Computer Science 112 B1
Introduction to Computer Science II

Boston University, Spring 2023

Introduction to Computer Science II

Course Overview;
A First Look at Java

Computer Science 112
Boston University

David G. Sullivan, Ph.D.

Welcome to Computer Science 112!

- We will study fundamental *data structures.*
  - ways of imposing order on a collection of information
  - sequences: lists, stacks, and queues
  - trees
  - hash tables

- We will also:
  - study *algorithms* related to these data structures
  - learn how to *compare* data structures & algorithms

- Goals:
  - learn to think more intelligently about programming problems
  - acquire a set of useful tools and techniques

- We will use the Java language.
  - but learning Java is *not* the primary focus of the course!

Sample Problem: A Data "Dictionary"

- Given a large collection of data, how can we arrange it so that we can efficiently:
  - add a new item
  - search for an existing item

- Some data structures provide better performance than others for this application.

- More generally, we'll learn how to characterize the *efficiency* of different data structures and their associated algorithms.

Prerequisites

- CS 111, or the equivalent
  - ideally with a B- or better
  - if not CS 111, solid coding skills in one of the following:
    - Python
    - Java
    - C++
  - comfortable with recursion
  - some exposure to object-oriented programming

- Reasonable comfort level with mathematical reasoning
  - mostly simple algebra, but need to understand the basics of logarithms (we'll review this)
  - we'll do some simple proofs

Course Materials

- **Required**: *Coursepack*
  - use it during pre-lecture and lecture – need to fill in the blanks!
  - PDF version is available on Blackboard
  - recommended: get it printed
    - one option: FedEx Office (Cummington & Comm Ave)

- **Required** in-class software: Top Hat Pro platform
  - used for pre-lecture quizzes and in-lecture exercises
  - create your account and purchase a subscription ASAP (see Lab 0 for more details)

Preparing for Lecture

- Short video(s) and/or reading(s)

- Short online quiz

- Preparing for lecture is essential!
  - gets you ready for the lecture questions and discussions
  - we won't cover everything in lecture

Course Website
www.cs.bu.edu/courses/cs112
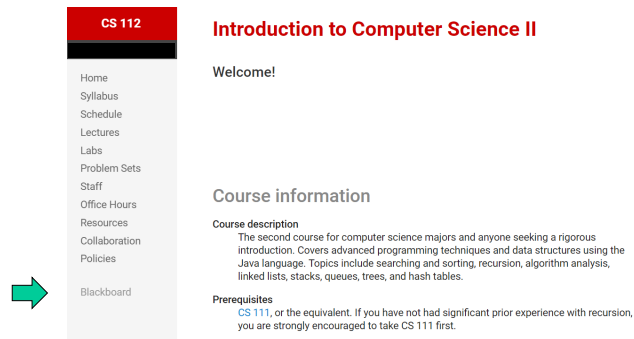
- *not* the same as the Blackboard site for the course

- use Blackboard to access info. on:
  - the pre-lecture videos/readings
  - the pre-lecture quizzes
  } posted by 36 hours before lecture
  - a note with the pages that we covered in lecture

Labs

- Will help you prepare for and get started on the assignments

- Will also reinforce essential skills

- **ASAP: Complete Lab 0** (on the course website)
  - short tasks to prepare you for the semester

Grading

1. Problem sets (20%) – see syllabus for due dates
2. Exams
   - two midterms (30%) – **Wed nights 6:30-8:00**; no makeups!
   - final exam (40%)
     - can replace lowest problem set and lowest midterm
     - **wait until you hear its dates/times from me;
       initial info posted by Registrar will likely be incorrect;
       make sure you're available for the entire exam period!**
3. Participation (10%)

*To pass the course, you must have
a passing PS average
<u>and</u> a passing exam average.*

Participation

- Full credit if you:
  - earn 85% of the Top Hat points over the entire semester
    (voting from outside classroom and voting for someone else
    are **not** allowed!)
  - attend 85% of the labs

- If you end up with x% for a given component where x < 85,
  you will get x/85 of the possible points.

- This policy is designed to allow for occasional absences
  for special circumstances – including isolation for Covid-19.

- If you need to miss a lecture:
  - watch its recording ASAP (available on Blackboard)
  - keep up with the pre-lecture tasks and the assignments
  - **do <u>not</u> email me!**

Course Staff

- **Instructors:**   David Sullivan (B1 lecture)
                       Christine Papadakis-Kanaris (A1/C1 lectures)

- **Teaching Assistants (TAs)**
  plus **Undergrad Course Assistants (CAs)**
  - see the course website:
    http://www.cs.bu.edu/courses/cs112/staff.html

- Office-hour calendar:
  http://www.cs.bu.edu/courses/cs112/office_hours.html

- For questions: **post on Piazza** or cs112-staff@cs.bu.edu

Getting Started with Java

- You all have a solid foundation in a programming language.

- For most of you, that language (Python) is *not* the one
  that we'll use (Java)!

- We'll cover some of the trickier aspects of Java together
  - in lecture
  - in the labs

- You'll need to learn the rest on your own.
  - something you will do many times in your career!
  - an important skill in its own right

Python vs. Java

- Here's a very simple Python program:

```
print('hello!')
```

- Here's the comparable program in Java:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("hello!");
    }
}
```

Python vs. Java

Python	**Java**

```
print('hello!')
```

```
public class HelloWorld {
    public static void
    main(String[] args) {
        System.out.println("hello!");
    }
}
```

- Classes are only used to define new data types.
  - *not* all programs need a class

- Functions/methods are useful but not required.
  - can put all code in the global scope

- Classes define data types, but they can also serve as "containers" for other code.
  - *all* programs need at least one class

- Every program needs at least method called `main()`.
  - with the header shown above
  - most code is inside methods

Python vs. Java (cont.)

Python

```python
def main():
    print('hello!')

main()
```

Java

```java
public class HelloWorld {
    public static void
    main(String[] args) {
        System.out.println("hello!");
    }
}
```

- Blocks of code are defined using indentation.

- Simple statements end at the end of the line.

- String literals can be surrounded with either single or double quotes.

- Blocks of code are (usually) defined using curly braces ({ }).
  - indentation isn't required, but we use it for readability!

- Simple statements end with a semi-colon.

- String literals must be surrounded with *double* quotes.

Python vs. Java (cont.)

Python

```python
def main():
    print('hello!')

main()
```

Java

```java
public class HelloWorld {
    public static void
    main(String[] args) {
        System.out.println("hello!");
    }
}
```

- You can run a program without pre-processing it.
  - can also execute code from the Shell/console

- When you run a program, the interpreter begins at the top of the file.
  - to execute a function, need to explicitly call it

- The program must be *compiled* before it can be run.
  - turns it into lower-level code
  - there is no Shell in Java

- When you run a program, Java begins with main().
  - even if there's code before it
  - it calls main() for us

Pre-Lecture
From Python to Java:
Converting a Simple Program

Computer Science 112
Boston University

Converting from Python to Java

- **Python:**

```python
quarters = 10
dimes = 3
nickels = 7
pennies = 6

cents = 25*quarters + 10*dimes + 5*nickels + pennies
print('total in cents is:', cents)
```

- **Java:** *this version still doesn't work!*

```java
public class ChangeAdder {
    public static void main(String[] args) {
        quarters = 10;
        dimes = 3;
        nickels = 7;
        pennies = 6;

        cents = 25*quarters + 10*dimes + 5*nickels + pennies;
        System.out.println("total in cents is: " + cents);
    }
}
```

Printing Values

- **Python:** a print statement can have multiple expressions:

    `print(`*expr₁*`, `*expr₂*`, ... `*exprₙ*`)`

  $\text{print}(expr_1, expr_2, ... expr_n)$

  - the resulting values are displayed on the same line, separated by spaces

- **Java:** you are limited to a *single* expression:

    `System.out.println(`*expr*`);`

  - if you need more than one value, use string concatenation

---

Printing Values: Other Examples

Python	**Java**
| ```
a = 3
b = 5
print(a, b)
``` | ```
a = 3;
b = 5;
System.out.println(a + " " + b);
``` |

output:
3 5

```
print('$' + str(dollars))
```

```
System.out.println("$" + dollars);
```

no conversion needed!

Declaring a Variable

- Java requires that we specify the type of data
  that a variable will store before we attempt to use it.

- This is called *declaring* the variable.
  - syntax:

    type   variable;

  - examples:

    ```
    int cents;
    ```

    type of   name of
    the variable   the variable

    ```
    int quarters = 10;
    ```

Converting from Python to Java

- **Python:**

```python
quarters = 10
dimes = 3
nickels = 7
pennies = 6

cents = 25*quarters + 10*dimes + 5*nickels + pennies
print('total in cents is: ', cents)
```

- **Java:** *here's the fixed version*

```java
public class ChangeAdder {
    public static void main(String[] args) {
        int quarters = 10;  // declare and assign, or...
        int dimes = 3;
        int nickels = 7;
        int pennies = 6;

        int cents;          // declare first, assign later
        cents = 25*quarters + 10*dimes + 5*nickels + pennies;
        System.out.println("total in cents is: " + cents);
    }
}
```

Pre-Lecture
From Python to Java:
Conditional Execution
and User Input

Computer Science 112
Boston University

Basic Changes

Python

```
avg = 85

if avg >= 90:
    grade = 'A'
elif avg >= 80:
    grade = 'B'
elif avg >= 70:
    grade = 'C'
elif avg >= 60:
    grade = 'D'
else:
    grade = 'F'

print(avg, '=', grade)
```

Java

```
avg = 85;

if avg >= 90:
    grade = "A";
elif avg >= 80:
    grade = "B";
elif avg >= 70:
    grade = "C";
elif avg >= 60:
    grade = "D";
else:
    grade = "F";

print(avg, "=", grade);
```

Printing

Python

```
avg = 85

if avg >= 90:
    grade = 'A'
elif avg >= 80:
    grade = 'B'
elif avg >= 70:
    grade = 'C'
elif avg >= 60:
    grade = 'D'
else:
    grade = 'F'

print(avg, '=', grade)
```

Java

```
avg = 85;

if avg >= 90:
    grade = "A";
elif avg >= 80:
    grade = "B";
elif avg >= 70:
    grade = "C";
elif avg >= 60:
    grade = "D";
else:
    grade = "F";

System.out.println(avg + " = " + grade);
```

Declaring Variables

Python

```
avg = 85

if avg >= 90:
    grade = 'A'
elif avg >= 80:
    grade = 'B'
elif avg >= 70:
    grade = 'C'
elif avg >= 60:
    grade = 'D'
else:
    grade = 'F'

print(avg, '=', grade)
```

Java

```
_____ avg = 85;

_____;
if avg >= 90:
    grade = "A";
elif avg >= 80:
    grade = "B";
elif avg >= 70:
    grade = "C";
elif avg >= 60:
    grade = "D";
else:
    grade = "F";

System.out.println(avg + " = " + grade);
```

Conditional Execution

Python

```python
avg = 85

if avg >= 90:
    grade = 'A'
elif avg >= 80:
    grade = 'B'
elif avg >= 70:
    grade = 'C'
elif avg >= 60:
    grade = 'D'
else:
    grade = 'F'

print(avg, '=', grade)
```

Java

```java
int avg = 85;

String grade;
if (avg >= 90) {
    grade = "A";
} else if (avg >= 80) {
    grade = "B";
} else if (avg >= 70) {
    grade = "C";
} else if (avg >= 60) {
    grade = "D";
} else {
    grade = "F";
}

System.out.println(avg + " = " + grade);
```

Getting User Input

Python

```python
avg = int(input('average: '))

if avg >= 90:
    grade = 'A'
elif avg >= 80:
    grade = 'B'
elif avg >= 70:
    grade = 'C'
elif avg >= 60:
    grade = 'D'
else:
    grade = 'F'

print(avg, '=', grade)
```

Java

```java
Scanner scan = new Scanner(System.in);

int avg = scan.nextInt();

String grade;
if (avg >= 90) {
    grade = "A";
} else if (avg >= 80) {
    grade = "B";
} else if (avg >= 70) {
    grade = "C";
} else if (avg >= 60) {
    grade = "D";
} else {
    grade = "F";
}

System.out.println(avg + " = " + grade);
```

Scanner Methods: A Partial List

- `scan.nextInt()`
  - read in an integer and return it

- `scan.nextDouble()`
  - read in a floating-point value and return it

- `scan.next()`
  - read in a single "word" and return it as a `String`

- `scan.nextLine()`
  - read in a "line" of input (could be multiple words) and return it as a `String`

Getting User Input

Python

```python
avg = int(input('average: '))

if avg >= 90:
    grade = 'A'
elif avg >= 80:
    grade = 'B'
elif avg >= 70:
    grade = 'C'
elif avg >= 60:
    grade = 'D'
else:
    grade = 'F'

print(avg, '=', grade)
```

Java

```java
Scanner scan = new Scanner(System.in);
System.out.print("average: ");
int avg = scan.nextInt();

String grade;
if (avg >= 90) {
    grade = "A";
} else if (avg >= 80) {
    grade = "B";
} else if (avg >= 70) {
    grade = "C";
} else if (avg >= 60) {
    grade = "D";
} else {
    grade = "F";
}

System.out.println(avg + " = " + grade);
```

Java Basics;
User Input and Conditional Execution

Computer Science 112
Boston University

Recall: Declaring a Variable

* Java requires that we specify the type of data
  that a variable will store before we attempt to use it.

* This is called *declaring* the variable.
    * syntax:

      type variable;

    * examples:

      ```
      int count;
      ```
      ← says that `count` will store an integer

      ```
      _____  area;
      ```
      ← says that `area` will store a
      floating-point number (one with a decimal)

* Optional: you can assign an initial value at the same time:
  ```
  int count = 0;
  double area = 125.5;
  ```

Commonly Used Data Types in Java

- `int` - an integer

    ```
    int count = 0;
    ```

- `double` - a floating-point number (one with a decimal)

    ```
    double area = 125.5;
    ```

- `String` - a sequence of 0 or more characters

    ```
    String message = "Welcome to CS 112!";
    ```

- `boolean` - either `true` or `false`

    ```
    boolean isPrime = false;
    ```

    unlike in Python, the boolean literals are *not* capitalized

---

Recall: Converting from Python to Java

- **Python:**

```python
quarters = 10     # we don't declare a variable's type
dimes = 3
nickels = 7
pennies = 6

cents = 25*quarters + 10*dimes + 5*nickels + pennies
print('total in cents is: ', cents)
```

- **Java:** *here's the fixed version*

```java
public class ChangeAdder {
    public static void main(String[] args) {
        int quarters = 10;  // declare and assign, or...
        int dimes = 3;
        int nickels = 7;
        int pennies = 6;

        int cents;          // declare first, assign later
        cents = 25*quarters + 10*dimes + 5*nickels + pennies;
        System.out.println("total in cents is: " + cents);
    }
}
```

Getting Input from the User

- **Python:**

```python
quarters = int(input('Enter the number of quarters: '))
dimes = int(input('Enter the number of dimes: '))
nickels = int(input('Enter the number of nickels: '))
pennies = int(input('Enter the number of pennies: '))

cents = 25*quarters + 10*dimes + 5*nickels + pennies
print('total in cents is: ', cents)
```

How should you fill in the blank?

- **Java:**

```java
import java.util.*;

public class ChangeAdder {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);

        System.out.print("Enter the number of quarters: ");
        int quarters = _____;
        int dimes = 3;
        int nickels = 7;
        int pennies = 6;

        int cents;
        cents = 25*quarters + 10*dimes + 5*nickels + pennies;
        System.out.println("total in cents is: " + cents);
    }
}
```

Recall: `Scanner` Methods: A Partial List

- `scan.nextInt()`

  where `scan` is a `Scanner` object

  - read in an integer and return it

- `scan.nextDouble()`
  - read in a floating-point value and return it

- `scan.next()`
  - read in a single "word" and return it as a `String`

- `scan.nextLine()`
  - read in a "line" of input (could be multiple words) and return it as a `String`

Getting Input from the User

- **Java:**

```java
import java.util.*;

public class ChangeAdder {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);

        System.out.print("Enter the number of quarters: ");
        int quarters = console.nextInt();
        System.out.print("Enter the number of dimes: ");
        int dimes = console.nextInt();
        System.out.print("Enter the number of nickels: ");
        int nickels = console.nextInt();
        System.out.print("Enter the number of pennies: ");
        int pennies = console.nextInt();

        int cents;
        cents = 25*quarters + 10*dimes + 5*nickels + pennies;
        System.out.println("total in cents is: " + cents);
    }
}
```

Recall: Conditional Execution

Python

```
avg = int(input('average: '))

if avg >= 60:
    print('passing')
    print('congrats!')
else:
    print('failing')
```

Java

```
Scanner scan = new Scanner(System.in);
System.out.print("average: ");
int avg = scan.nextInt();

if (avg >= 60) {
    System.out.println("passing");
    System.out.println("congrats!");
} else {
    System.out.println("failing");
}
```

- Snytax notes:
    - the condition is surrounded by parentheses (no colon)
    - each block is surrounded by curly braces

What does this print?

```
int x = 5;
if (x < 15) {
    if (x > 8) {
        System.out.println("one");
    } else {
        System.out.println("two");
    }
}
if (x > 2) {
    System.out.println("three");
}
```

What happens if I shift these two lines over?

```java
int x = 5;
if (x < 15) {
    if (x > 8) {
        System.out.println("one");
} else {
    System.out.println("two");
    }
}
if (x > 2) {
    System.out.println("three");
}
```

Java vs. Python

```java
int x = 5;                // the original Java version
if (x < 15) {
    if (x > 8) {
        System.out.println("one");
    } else {
        System.out.println("two");
    }
}
if (x > 2) {
    System.out.println("three");
}
```

output:
two
three

```python
x = 5                     # the equivalent Python code
if x < 15:
    if x > 8:
        print('one')
    else:
        print('two')
if x > 2:
    print('three')
```

output:
two
three

Java vs. Python

```java
int x = 5;
if (x < 15) {
    if (x > 8) {
        System.out.println("one");
    } else {
        System.out.println("two");
    }
}
if (x > 2) {
    System.out.println("three");
}
```

output:
two
three

```python
x = 5
if x < 15:
    if x > 8:
        print('one')
else:
    print('two')
if x > 2:
    print('three')
```

output?

To make the Java version behave the same…

```java
int x = 5;
if (x < 15) {
    if (x > 8) {
        System.out.println("one");
    }
} else {
    System.out.println("two");
}
if (x > 2) {
    System.out.println("three");
}
```

```python
x = 5
if x < 15:
    if x > 8:
        print('one')
else:
    print('two')
if x > 2:
    print('three')
```

Java's Logical Operators

operator	Python equivalent	example
&&	and	avg >= 80 && avg <= 85
||	or	avg < 0 || avg > 100
!	not	!(avg < 80 || avg > 90)

Operators and Data Types

- Each data type has its own set of operators.
  - the `int` version of an operator produces an `int` result
  - the `double` version produces a `double` result
  - etc.

- Rules for numeric operators:
  - if the operands are both of type `int`,
    the `int` version of the operator is used.
    - examples: `15 + 30`
      `1 / 2`
      `25 * quarters` *// quarters is an int*
  - if at least one of the operands is of type `double`,
    the `double` version of the operator is used.
    - examples: `15.5 + 30.1`
      `1 / 2.0`
      `25.0 * quarters`

Two Types of Division

- The `int` version of `/` performs *integer division*,
  which discards everything after the decimal.
  - like `//` in Python

- The `double` version of `/` performs *floating-point division*,
  which keeps the digits after the decimal.

- Examples:

statement	output
`System.out.println(5 / 3.0);`	`1.6666666666666667`
`System.out.println(5 / 3);`	`1`
`System.out.println(16.0 / 5);`	
`System.out.println(16 / 5);`	

An <u>Incorrect</u> Extended Change-Adder Program How Can We Fix It?

```java
import java.util.*;

public class ChangeAdder2 {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);

        System.out.print("Enter the number of quarters: ");
        int quarters = console.nextInt();
        System.out.print("Enter the number of dimes: ");
        int dimes = console.nextInt();
        System.out.print("Enter the number of nickels: ");
        int nickels = console.nextInt();
        System.out.print("Enter the number of pennies: ");
        int pennies = console.nextInt();

        int cents;
        cents = 25*quarters + 10*dimes + 5*nickels + pennies;
        System.out.println("total in cents is: " + cents);
        double dollars = cents / 100;
        System.out.print("total in dollars is: $" + dollars);
    }
}
```

An <u>Incorrect</u> Program for Computing a Grade

```java
/*
 * ComputeGrade.java
 *
 * This program computes a grade as a percentage.
 */

public class ComputeGrade {
    public static void main(String[] args) {
        int pointsEarned = 13;
        int possiblePoints = 15;

        // compute and print the grade as a percentage
        double grade;
        grade = pointsEarned / possiblePoints * 100;
        System.out.println("The grade is: " + grade);
    }
}
```
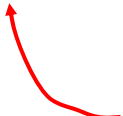
* What is the output?

Will This Fix Things?

```java
/*
 * ComputeGrade.java
 *
 * This program computes a grade as a percentage.
 */

public class ComputeGrade {
    public static void main(String[] args) {
        int pointsEarned = 13;
        int possiblePoints = 15;

        // compute and print the grade as a percentage
        double grade;
        grade = pointsEarned / possiblePoints * 100.0;
        System.out.println("The grade is: " + grade);
    }
}
```

Type Casts

- To compute the percentage, we need to tell Java to treat at least one of the operands as a `double`.

- We do so by performing a *type cast:*

  ```
  grade = (double)pointsEarned / possiblePoints * 100;
  ```
                          or

  ```
  grade = pointsEarned / (double)possiblePoints * 100;
  ```

- General syntax for a type cast:

  (*type*) *variable*

Corrected Program for Computing a Grade

```
/*
 * ComputeGrade.java
 *
 * This program computes a grade as a percentage.
 */

public class ComputeGrade {
    public static void main(String[] args) {
        int pointsEarned = 13;
        int possiblePoints = 15;

        // compute and print the grade as a percentage
        double grade;
        grade = (double)pointsEarned / possiblePoints * 100;
        System.out.println("The grade is: " + grade);
    }
}
```

Evaluating a Type Cast

- Example of evaluating a type cast:

```
pointsEarned = 13;
possiblePoints = 15;
grade = (double)pointsEarned / possiblePoints * 100;
        (double)13 / 15 * 100;
               13.0 / 15 * 100;
        0.8666666666666667 * 100;
        86.66666666666667;
```

- Note that the type cast occurs *after* the variable is replaced by its value.

- It does *not* change the value stored in the variable.
  - in the example above, `pointsEarned` is still 13

Type Conversions

- Java will automatically convert values from one type to another *provided there is no potential loss of information.*

- Example: we *don't* need a type cast here:

```
double d = 3;
```

variable of   value of
type `double`   type `int`

  - 3 is converted to `3.0`
  - `3.0` is assigned to `d`
  - *any* `int` can be assigned to a variable of type `double`

Type Conversions (cont.)

- The compiler will complain if the necessary type conversion could (at least in some cases) lead to a loss of information:

    ```
    int i = 7.5;     // won't compile
    ```

    variable of
    type int

    value of
    type double

- This is true regardless of the actual value being converted:

    ```
    int i = 5.0;     // won't compile
    ```

- In such cases, we need to perform the conversion ourselves:

    ```
    double area = scan.nextDouble(); // get from user
    int approximateArea = (int)area;
    System.out.println(approximateArea);
    ```

Type Conversions (cont.)

- When an automatic type conversion is performed as part of an assignment, the conversion happens <u>after</u> the evaluation of the expression to the right of the =.

- Example:

    ```
    double d = 1 / 3;
              = 0;      // uses integer division. why?
             = 0.0;
    ```

Pre-Lecture
From Python to Java:
Functions / Static Methods

Computer Science 112
Boston University

Functions / Methods

- Python distinguishes between:

  - *functions:* named blocks of code that:
    - take 0 or more inputs/parameters
    - return a value

  - *methods:* functions that are "inside" an object
    - have a `self` parameter

```python
def grade(avg):
    if avg >= 90:
        grade = 'A'
    elif avg >= 80:
        grade = 'B'
    elif avg >= 70:
        grade = 'C'
    elif avg >= 60:
        grade = 'D'
    else:
        grade = 'F'

    return grade
```

```python
class Rectangle:
    def __init__(self, w, h):
        self.width = w
        self.height = h

    def area(self):
        a = self.width * self.height
        return a
```

Functions / Methods

- Python distinguishes between:
  - *functions:* named blocks of code that:
    - take 0 or more inputs/parameters
    - return a value
  - *methods:* functions that are "inside" an object
    - have a `self` parameter

- In Java, both types of functions are called methods.
  - *static* methods – like Python functions
  - *non-static* or *instance* methods – like Python methods

Functions / Static Methods

Python	**Java**

```python
def grade(avg):
    if avg >= 90:
        grade = 'A'
    elif avg >= 80:
        grade = 'B'
    elif avg >= 70:
        grade = 'C'
    elif avg >= 60:
        grade = 'D'
    else:
        grade = 'F'

    return grade
```

```java
public static String grade(int avg) {
    String grade;
    if (avg >= 90) {
        grade = "A";
    } else if (avg >= 80) {
        grade = "B";
    } else if (avg >= 70) {
        grade = "C";
    } else if (avg >= 60) {
        grade = "D";
    } else {
        grade = "F";
    }

    return grade;
}
```

- Format of the header:

  `public static` *return-type* *method-name*(*parameters*)

  where each parameter is preceded by its type.

Pre-Lecture
From Python to Java:
for Loops

Computer Science 112
Boston University

Shortcut Operators

Python	Java

Python

```
result = 2
n = 5
result *= n
n += 1
```

Java

```
int result = 2;
int n = 5;
result *= n;
n++;
```

- Python has operators that combine arithmetic with assignment:

  ```
  +=
  -=
  *=
  etc.
  ```

- Java also has these operators.

- In addition, it has two special ones for adding/subtracting 1:

  ```
  x++    is the same as   x = x + 1
  x--    is the same as   x = x - 1
  ```

for Loops

Python

```
def fac(n):
    result = 1
    for x in range(2, n+1):
        result *= x
    return result
```

Java

```
public static int fac(int n) {
    int result = 1;
    for (int x = 2; x <= n; x++) {
        result *= x;
    }
    return result;
}
```

for Loops in Java

- Syntax:

  for (*initialization*; *continuation-test*; *update*) {
      body
  }

- In our example: *initialization* *continuation test*

  ```
  for (int x = 2; x <= n; x++) {
      result *= x;
  }
  ```
  update

Executing a `for` Loop

```
for (initialization; continuation-test; update) {
    body
}
```

```
perform the
initialization
```

```
is the
test true?
```
no

yes

```
execute the
body of the loop
```

```
perform the
update
```

```
execute statement
after the loop
```
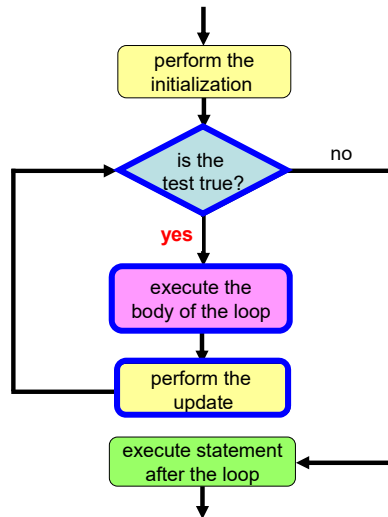
Notes:
- the initialization is only performed once
- the body is only executed if the test is true
- we repeatedly do:
  - test
  - body
  - update

  until the test is false

Tracing a `for` Loop

```
for (int x = 2; x <= n; x++) {
    result *= x;
}
```

fac(5)
n = 5

```
initialization:
int x = 2;
```

```
is x <= n
true?
```
no

yes

```
execute body:
result *= x;
```

```
perform update:
x++
```

```
execute statement
after the loop
```
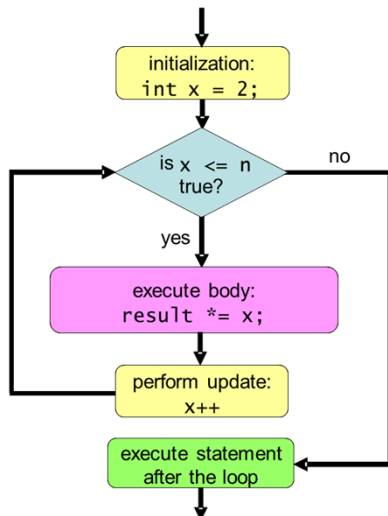
x	x <= n	result
		1
2	true	2

Static Methods;
A First Look at Loops

Computer Science 112
Boston University

Recall: Functions / Methods

- Python distinguishes between:
  - *functions:* named blocks of code that:
    - take 0 or more inputs/parameters
    - return a value
  - *methods:* functions that are "inside" an object
    - have a `self` parameter

- In Java, both types of functions are called methods.
  - *static* methods – like Python functions
  - *non-static* or *instance* methods – like Python methods
    - example?

A Side Note on Conditional Execution

Python

```python
def grade(avg):
    if avg >= 90:
        grade = 'A'
    elif avg >= 80:
        grade = 'B'
    elif avg >= 70:
        grade = 'C'
    elif avg >= 60:
        grade = 'D'
    else:
        grade = 'F'

    return grade
```

Java

```java
public static String grade(int avg) {
    String grade;
    if (avg >= 90) {
        grade = "A";
    } else if (avg >= 80) {
        grade = "B";
    } else if (avg >= 70) {
        grade = "C";
    } else if (avg >= 60) {
        grade = "D";
    } else if (avg < 60) {
        grade = "F";
    }

    return grade;  // error
}
```

* What if we changed the `else` to an `else if`?
    * compiler error: `grade` may not have been initialized
      (the compiler worries *all* of the conditions might be false)
    * always use an `else` if you know one choice must execute!

A method for finding the larger of two real numbers. What's the correct header?

```
public _____ max(_____) {
    if (val1 > val2) {
        return val1;
    } else {
        return val2;
    }
}
```

* Here's an example of calling it:

    ```
    double larger = max(10.5, 20.7);
    ```

Another Static Method

voidindicates that the method does *not* return a value

```
public static void printSquareInfo(int sideLength, String units) {
    int perim = 4 * sideLength;
    int area = sideLength * sideLength;        takes an int followed by a String

    System.out.println("side length = " + sideLength + " " + units);
    System.out.println("perimeter = " + perim + " " + units);
    System.out.println("area = " + area + " " + units + " squared")
}
```

* Here's an example of calling it:

```
    printSquareInfo(8, "inches");       // no value is returned!
                                        // in Python, would return None
```

* What's the output?

Practice: Computing Absolute Value

* Write a method named abs for computing the absolute value
  of a floating-point number n:

```
    public static _____ abs(_____) {




    }
```

The `Math` Class

- Java's built-in `Math` class contains static methods for mathematical operations.
    - examples:

        `abs(value)` – returns the absolute value of value

        `round(value)` – returns the result of rounding value to the nearest integer

        `pow(base, expon)` – returns the result of raising `base` to the `expon` power

        `sqrt(value)` – returns the square root of `value`

- To use a static method defined in another class, we prepend the name of the class.

    ```
    double numVals = Math.pow(2, 20);
    ```
    class name   static method name

Calling Functions / Static Methods

- If the function or static method is in the *current file*, just use its name:

Python	Java
    | ```
avg = 85
letter_grade = grade(avg)
``` | ```
int avg = 85;
String letter_grade = grade(avg);
``` |

- If the function or static method is in a different module/class:
    - import it as needed
    - prepend the module/class name

Python	Java
    | ```
import math
x = math.sqrt(100)
``` | ```
// we don't need to import Math!
double x = Math.sqrt(100);
``` |

Calling (Non-Static) Methods

- Because (non-static) methods are inside an object,
  we prepend the name of the object:

Python

```
s1 = 'hello'
s2 = s1.upper()
```

Java

```
Scanner scan = new Scanner(System.in);
int avg = scan.nextInt();
```

Tracing a `for` Loop

- Trace this loop by filling in the table::

```
for (int i = 2; i <= 10; i += 2) {
    System.out.println(i * 10);
}
```

i	i <= 10	value printed

To Get *N* Repetitions

Python   **Java**

- templates:

```
for i in range(N):
    body of loop
```

```
for (int i = 0; i < N; i++) {
    body of loop
}
```

- example: what do these print?

```
for i in range(5):
    print('Hip, hip!')
    print('Hooray!')
```

```
for (int i = 0; i < 5; i++) {
    System.out.println("Hip, hip!");
    System.out.println("Hooray!");
}
```

- to get 100 repetitions instead:

```
for i in range(100):
    print('Hip, hip!')
    print('Hooray!')
```

```
for (int i = 0; i < 100; i++) {
    System.out.println("Hip, hip!");
    System.out.println("Hooray!");
}
```

Which option(s) work?

- Fill in the blanks below to print the integers from 1 to 5:

```
for (_____; _____; _____) {
    System.out.println(i);
}
```

More Practice

- Fill in the blanks below to print the integers from 10 to 20:

```
for (_____; _____; _____) {
    System.out.println(i);
}
```

- Fill in the blanks below to print the integers from 10 down to 1:

```
for (_____; _____; _____) {
    System.out.println(i);
}
```

Pre-Lecture
Variable Scope in Java

Computer Science 112
Boston University

Variable Scope

* The *scope* of a variable is the portion of a program
  in which the variable can be used.

* By default, the scope of a variable in Java:
  * begins at the point at which it is declared
  * ends at the end of the innermost block
    that encloses the declaration

```
public static void printResults(int a, int b) {
    System.out.println("Here are the stats:");

    int sum = a + b;
    System.out.print("sum = ");
    System.out.println(sum);

    double avg = (a + b) / 2.0;
    System.out.print("average = ");
    System.out.println(avg);
}
```

scope of sum

scope of avg

Local Variables

- Variables that are declared inside a method are *local variables.*
  - they cannot be used outside that method.

```
public static void printResults(int a, int b) {
    System.out.println("Here are the stats:");
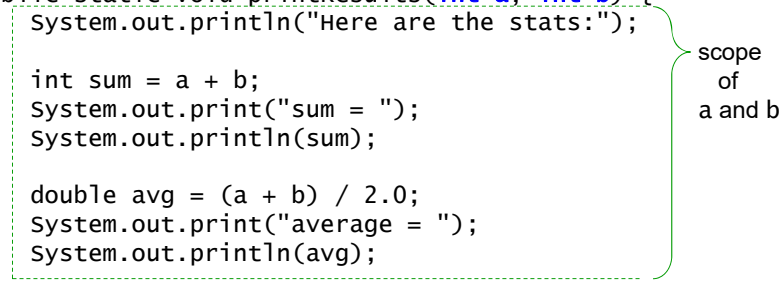
    int sum = a + b;
    System.out.print("sum = ");
    System.out.println(sum);                    scope of sum

    double avg = (a + b) / 2.0;
    System.out.print("average = ");    scope of
    System.out.println(avg);               avg
}
```

Special Case: Parameters

- What about the parameters of a method?
  - they do *not* follow the default scope rules!
  - their scope is limited to their method

```
public class MyClass {
    public static void printResults(int a, int b) {
        System.out.println("Here are the stats:");
                                                        scope
        int sum = a + b;                                 of
        System.out.print("sum = ");                    a and b
        System.out.println(sum);
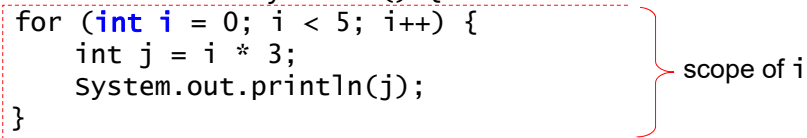
        double avg = (a + b) / 2.0;
        System.out.print("average = ");
        System.out.println(avg);
    }

    int c = a + b;      // does not compile!
}
```

Special Case: `for` Loops

- When a variable is declared in the initialization clause of a `for` loop, its scope is limited to the loop.

- Example:

```
public static void myMethod() {
    for (int i = 0; i < 5; i++) {           ⎫
        int j = i * 3;                       ⎬  scope of i
        System.out.println(j);              ⎭
    }
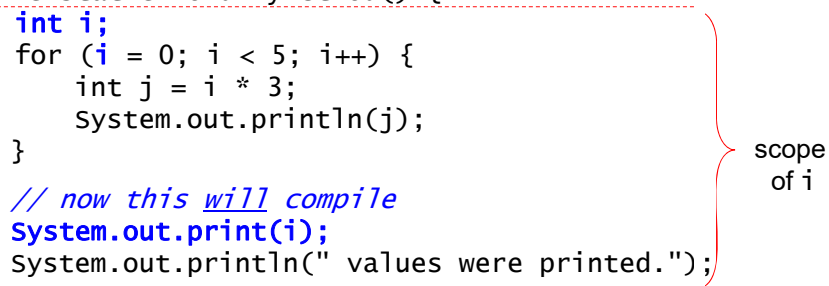
    // the following line won't compile
    System.out.print(i);
    System.out.println(" values were printed.");
}
```

Special Case: `for` Loops

- To allow `i` to be used outside the loop, we need to declare it outside the loop:

- Example:

```
public static void myMethod() {
    int i;                                  ⎫
    for (i = 0; i < 5; i++) {                ⎪
        int j = i * 3;                       ⎪
        System.out.println(j);               ⎬  scope
    }                                        ⎪  of i
                                             ⎪
    // now this will compile                 ⎪
    System.out.print(i);                     ⎪
    System.out.println(" values were printed.");⎭
}
```

From Python to Java:
Loops Revisited; Variable Scope

Computer Science 112
Boston University

Definite vs. Indefinite Loops

* for loops are *definite loops*
    * we use them when we know how many repetitions we need
    * templates for N repetitions:

Python	**Java**

```
for i in range(N):
    body of loop
```

```
for (int i = 0; i < N; i++) {
    body of loop
}
```

* *Indefinite loops* are used when:
    * we don't know how many repetitions are needed
    * it's harder to determine the number of repetitions
    * Java provides *two* options: `while` and `do…while`

while Loops

Python

```python
def fac(n):
    result = 1
    while n > 1:
        result *= n
        n -= 1
    return result
```

Java

```java
public static long fac(long n) {
    long result = 1;
    while (n > 1) {
        result *= n;
        n--;
    }
    return result;
}
```

- Here again, the Java version needs:
  - parens around the condition (no colon)
  - curly braces around the block (the *body* of the loop)

- The int type in Java uses 4 bytes per integer.
  - gives an approx. domain of [-2 billion, +2 billion]

- For larger integers, we can use the long type instead.
  - uses 8 bytes per integer

What is the *final* value printed by this loop?

```java
int a = 10;
while (a > 2) {
    a = a – 2;
    System.out.println(a * 2);
}
```

	a > 2	a	output
before loop		10	
1st iteration	10 > 2 (true)	8	16
2nd iteration			
3rd iteration (if any)			
4th iteration (if any)			

Using a Loop When Error-Checking

- Let's say we want the user to enter a positive integer.

- If the number is <= 0, we want to ask the user to try again.

- Here's one way of doing it using a `while` loop:

```
Scanner console = new Scanner(System.in);
System.out.print("Enter a positive integer: ");
int num = console.nextInt();
while (num <= 0) {
    System.out.print("Enter a positive integer: ");
    num = console.nextInt();
}
```

- Note that we end up duplicating code.

Error-Checking Using a `do-while` Loop

- a do…while loop allows us to eliminate the duplication:

```
Scanner console = new Scanner(System.in);
int num;
do {
    System.out.print("Enter a positive integer: ");
    num = console.nextInt();
} while (num <= 0);
```

- The code in the body of a do-while loop is always executed at least once.

do-while Loops

- In general, a do-while statement has the form
  ```
  do {
      one or more statements
  } while (test);
  ```
  - note the need for a semi-colon after the condition

- We do *not* use a semi-colon after a while loop's condition:
  ```
  while (test) {
      one or more statements
  }
  ```
  - beware of using one – it can actually create an infinite loop!

Evaluating a do-while Loop

Steps:

1. execute the statements in the body

2. evaluate the test

3. if it's true, go back to step 1

(if it's false, continue to the next statement)
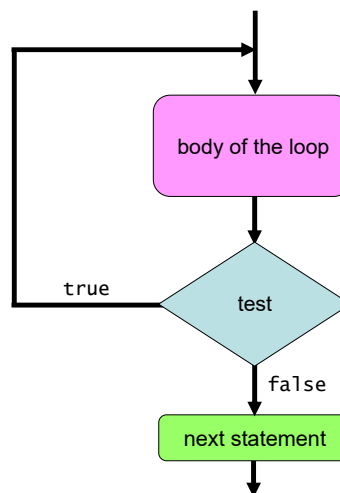
Comparing while and do-while
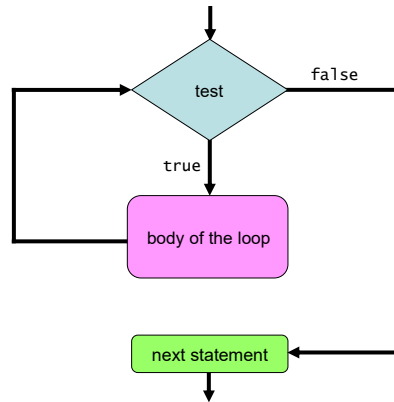
while loop

do-while loop
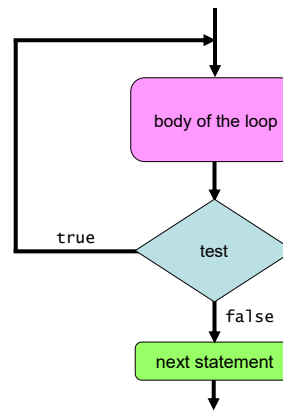
- In a do-while, the first test comes *after* executing the body.
  - thus, the body is always executed *at least once*

Which Type of Loop Should You Use?

- Use a for loop when the number of repetitions is known in advance – i.e., for a definite loop.

- Otherwise, use a while loop or do-while loop:
  - use a while loop if the body of the loop may not be executed at all
    - i.e., if the condition may be false at the start of the loop
  - use a do-while loop if:
    - the body will always be executed at least once
    - doing so will allow you to avoid duplicating code

Another Example of Variable Scope

```
public class MyProgram {
    public static void method1() {
        int i = 5;
        System.out.println(i * 3);
        int j = 10;
        System.out.println(j / i);
    }

    public static void main(String[] args) {
        // The following line won't compile.
        System.out.println(i + j);

        int i = 4;
        System.out.println(i * 6);
        method1();
    }
}
```

scope of j

scope of
method1's
version of i

scope of
main's
version of i

If we comment out the problematic line,
what is printed at the end of the program?
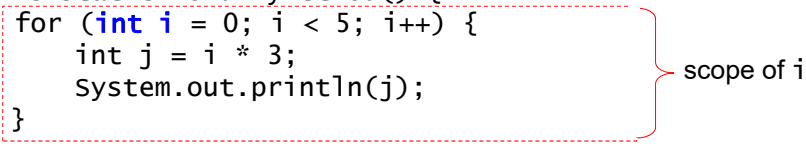
```
public class MyProgram {
    public static void method1() {
        int i = 5;
        System.out.println(i * 3);
        int j = 10;
        System.out.println(j / i);
    }

    public static void main(String[] args) {
        // The following line won't compile.
        // System.out.println(i + j);

        int i = 4;
        System.out.println(i * 6);
        method1();

        System.out.println(i);   // what would this print?
    }
}
```

Recall: `for` Loops and Variable Scope

- When a variable is declared in the initialization clause of a `for` loop, its scope is limited to the loop.

- Example:
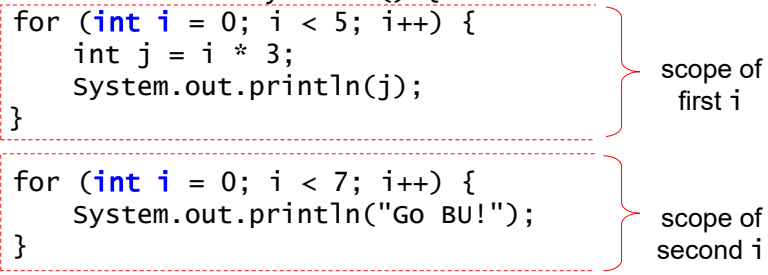
```java
public static void myMethod() {
    for (int i = 0; i < 5; i++) {
        int j = i * 3;
        System.out.println(j);
    }


    System.out.print(i);
    System.out.println(" values were printed.");
}
```

scope of `i`

Recall: `for` Loops and Variable Scope (cont.)

- Limiting the scope of a loop variable allows us to use the standard loop templates multiple times in the same method.

- Example:

```java
public static void myMethod() {
    for (int i = 0; i < 5; i++) {
        int j = i * 3;
        System.out.println(j);
    }

    for (int i = 0; i < 7; i++) {
        System.out.println("Go BU!");
    }
}
```

scope of first `i`

scope of second `i`

Practice with Scope

```
public static void drawRectangle(int height) {
    for (int i = 0; i < height; i++) {
        //  which variables could be used here?
        int width = height * 2;
        for (int j = 0; j < width; j++) {
            System.out.print("*");
            //  what about here?
        }
        //  what about here?
        System.out.println();
    }
    // what about here?
}

public static void repeatMessage(int numTimes) {
    //  what about here?
    for (int i = 0; i < numTimes; i++) {
        System.out.println("What is your scope?");
    }
}
```

Recall this static method…

*   Write a method named abs for computing the absolute value of a floating-point number n:

```
public static double abs(double n) {

    if (n >= 0) {

        return n;

    } else {

        return -1 * n;

    }

}
```

Here's an alternative approach. Why won't it work?

- Write a method named abs for computing the absolute value of a floating-point number n:

```java
public static double abs(double n) {

    if (n >= 0) {
        int result = n;
    } else {
        result = -1 * n;
    }
    return result;
}
```

How can we fix it?

- Write a method named abs for computing the absolute value of a floating-point number n:

```java
public static double abs(double n) {

    if (n >= 0) {
        int result = n;
    } else {
        result = -1 * n;
    }
    return result;
}
```

Pre-Lecture
From Python to Java:
Primitives, Objects, and References

Computer Science 112
Boston University

Recall: Variables and Values in Python

- In Python, when we assign a value to a variable,
  we're not actually storing the value *in* the variable.

- Rather:
  - the value is somewhere else in memory
  - the variable stores the *memory address* of the value.

- Example: x = 7

Memory

4000	
x 4001 → 4001	7
4002	
4003	
...	

Recall: References

Memory

```
        4000 ┌──────────┐
             │          │
x  4001      │          │
        4001 │    7     │
             │          │
        4002 │          │
             │          │
        4003 │          │
             │          │
        ...  │          │
             └──────────┘
```
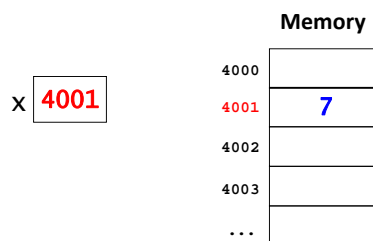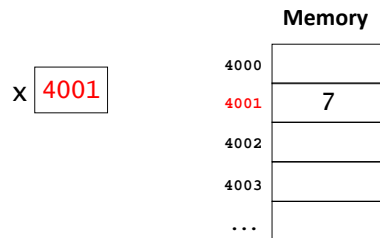
- We say that a variable stores a *reference* to its value.
  - also known as a *pointer*

- Because we don't care about the actual memory address, we use an arrow to represent a reference:

Memory

```
x ┌───┐
  │   │────────────────▶  7
  └───┘
```

Recall: Simplifying Our Mental Model

- In Python, when a variable represents certain types of values:
  - integers
  - floats
  - strings
  - other immutable (unchangeable) values

  it's okay to picture the value as being *inside* the variable.

```
x = 7              x │  7  │
```

Primitive Types

- In Java, some types of data <u>are</u> stored inside their variables:

      int x = 7;   x | 7 |

    - it's not a simplification
    - there is no reference!

- These data types are known as *primitive types:*
    - `int`
    - `long`
    - `double`
    - `boolean`
    - a few others

Object vs. Primitive

- Recall: An object is a construct that groups together:
    - one or more data values
      (the object's *attributes* or *fields*)
    - one or more functions
      (known as the object's *methods*)

- Every object is an *instance*
  of a class.

- Primitive values are *not* objects.
    - they are just "single" values
    - there is nothing else grouped with the value
    - they are not instances of a class
    - they only use a small number of bytes
        - that's why they can be easily stored inside their variables!

String object for `"hello"`

| contents | 'h' | 'e' | 'l' | 'l' | 'o' |

length | 5 |

```
replace()
split()
...
```

an int

| 112 |

Reference Types

- Java stores objects the same way that Python does:

      String s1 = "hello, world";

  s1 [→] → "hello, world"

  - the object is stored *outside* the variable
  - the variable stores a reference to the object

- Data types that work this way are known as *reference types.*
  - variables of those types are *reference variables*

Why Declare Variables?

- One reason:
  - primitives are stored inside variables
  - different primitive values require different amount of memory

primitive type	size
int	4 bytes
double	8 bytes
long	8 bytes
boolean	1 byte

- Declaring a variable tells the compiler how much memory to allocate!
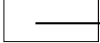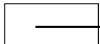
      int count = 1;   count [1] ← 4 bytes
      double result = 3.14159;
                                        result [3.14159] ← 8 bytes

What About Python?

- In Python, *everything* is an object.
  - thus, all variables hold references (memory addresses)

- As a result, Python can make every variable the same size.
  - and thus we don't need to declare the variable!

```
count = 1              count  [  ——→ ]  1

result = 3.14159       result [  ——→ ]  3.14159
```

Pre-Lecture
From Python to Java:
Working with Strings

Computer Science 112
Boston University

Representing Individual Characters in Java

- The `char` type is used to represent individual characters.

- It is a primitive type.

- To specify a `char` literal, we surround the character by single quotes:
  - examples: `'a'` `'z'` `'0'` `'7'` `'?'`
  - can only represent single characters   `'hi'` ← error!
  - if you want a `char`, don't use double-quotes!
    `"a"` is a string
    `'a'` is a char

Working with String Objects

Python	**Java**
| ```python
s1 = "hello"
s2 = "world!"
s1 = s1 + " " + s2
num_chars = len(s1)
s2 = s1[0:5] + s1[-1]

s3 = s2.upper()
``` | ```java
String s1 = "hello";
String s2 = "world!";
s1 = s1 + " " + s2;
int numChars = s1.length();
s2 = s1.substring(0, 5)
      + s1.charAt(numChars - 1);
String s3 = s2.toUpperCase();
``` |

A style convention:

* When a name in Python has more than one word, the convention is to separate the words with _ characters.

* In Java, we instead capitalize the first letter of each new word.

Working with String Objects

Python	**Java**
| ```python
s1 = "hello"
s2 = "world!"
s1 = s1 + " " + s2
num_chars = len(s1)
s2 = s1[0:5] + s1[-1]

s3 = s2.upper()
``` | ```java
String s1 = "hello";
String s2 = "world!";
s1 = s1 + " " + s2;
int numChars = s1.length();
s2 = s1.substring(0, 5)
      + s1.charAt(numChars - 1);
String s3 = s2.toUpperCase();
``` |

* Python has a len() function that takes a string as input.

* In Java, strings have a length() method.
    * inside the String object
    * a *non-static* method

Working with String Objects

Python

```python
s1 = "hello"
s2 = "world!"
s1 = s1 + " " + s2
num_chars = len(s1)
s2 = s1[0:5] + s1[-1]

s3 = s2.upper()
```

Java

```java
String s1 = "hello";
String s2 = "world!";
s1 = s1 + " " + s2;
int numChars = s1.length();
s2 = s1.substring(0, 5)
        + s1.charAt(numChars - 1);
String s3 = s2.toUpperCase();
```

- Python has special operators for slicing and indexing strings.

- In Java, we use non-static methods instead:
  - `substring(start, end)` for slicing
  - `charAt(index)` for indexing
- We can't use negative indices.
  - use `s.length() - 1` for the last character

Working with String Objects

Python

```python
s1 = "hello"
s2 = "world!"
s1 = s1 + " " + s2
num_chars = len(s1)
s2 = s1[0:5] + s1[-1]

s3 = s2.upper()
```

Java

```java
String s1 = "hello";
String s2 = "world!";
s1 = s1 + " " + s2;
int numChars = s1.length();
s2 = s1.substring(0, 5)
        + s1.charAt(numChars - 1);
String s3 = s2.toUpperCase();
```

- Python strings also have methods.

- In Java, the equivalent methods often have different names.

After running this, what is the value of s3?

Java

```java
String s1 = "hello";
String s2 = "world!";
s1 = s1 + " " + s2;
int numChars = s1.length();
s2 = s1.substring(0, 5)
        + s1.charAt(numChars - 1);
String s3 = s2.toUpperCase();
```

s1 ☐

s2 ☐

numChars ☐

s3 ☐

From Python to Java: Primitives, Objects, and References

Computer Science 112
Boston University

Recall: Variables and Values in Python

• In Python, when we assign a value to a variable,
  we're actually storing a *reference* to the value.

• Example: x = 7

Memory

x **4001**

4000	
4001	**7**
. . .	

• We depict a reference using an arrow.

Memory

x [——————→] 7
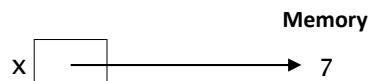
• When a value is immutable (unchangeable),
  we can picture it as being *inside* its variable.

x **7**
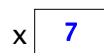
Recall: Primitive Types vs. Reference Types

- In Java, some types of data <u>are</u> stored inside their variables:

    `int x = 7;`   x `7`

    - it's not a simplification
    - there is no reference!

- These *primitive types* are *not* defined by a class.
    - their values are *not* objects
    - they are just "single" or "simple" values

- *Reference types* are types that *are* defined by a class.
    - their values *are* objects
        - they group together fields and methods
    - variables of these types hold a reference to an object

Which of these is a reference type in Java?

A. `char`   B. `double`   C. `String`   D. `Scanner`

E.   more than one of the above

sample variables of these types   *picturing the variables in memory*

`char ch = 'g';`   ch [ ]

`double avg = 85.3;`   avg [ ]

`String s = "hello";`   s [ ]

`Scanner in = new Scanner…;`   in [ ]

Which of these is a reference type in Java?

A. `char`   B. `double`   C. `String`   D. `Scanner`

   E.   more than one of the above

- Note:

   - _____ types begin with an upper-case letter
      - because we capitalize class names in Java

   - _____ types begin with a lower-case letter

Recall: Why do we declare variables in Java?

- One reason: doing so allows the compiler to give each variable the correct amount of memory!

```
int count = 1;
```
count | 1 | ← 4 bytes

```
double result = 3.14159;
```
result | 3.14159 | ← 8 bytes

- In Python, we're *always* storing a reference, so all variables have the same size!
   - the size of a memory address (usually 8 bytes)

```
count = 1
```
count | → | → 1

```
result = 3.14159
```
result | → | → 3.14159

   - thus, we don't need to declare the type

Using Methods in an Existing Class

- Using an object from an existing class involves calling its *non-static* methods.
  - the ones inside the object

- How do we figure out:
  - which methods are available
  - what they do
  - how to call them

- Example: What if I want to figure out the length of a string, but I don't remember how?

The API of a Class

- The methods defined in a given class are the *API* of that class.
  - API = application programming interface

- The API of all classes that come with Java is available here:
  `https://docs.oracle.com/javase/8/docs/api/`

  - there's a link on the Resources page of the course website

Consulting the Java API

select the package name (optional)

String is in java.lang

Scanner is in java.util
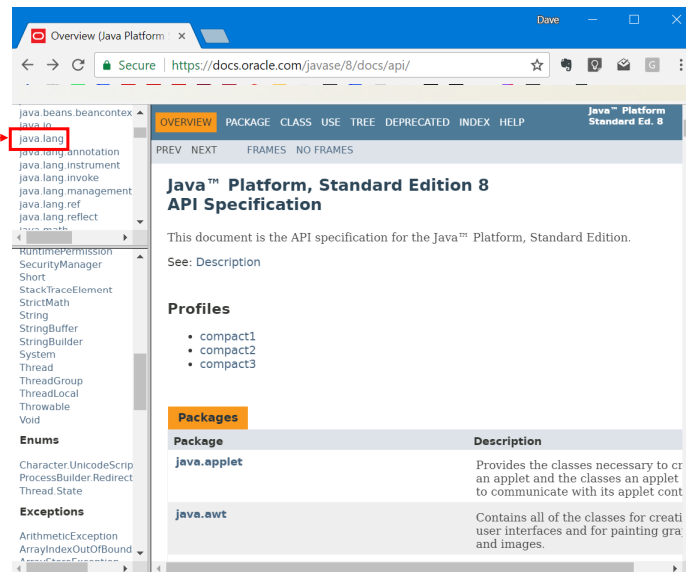
Consulting the Java API

select the class name
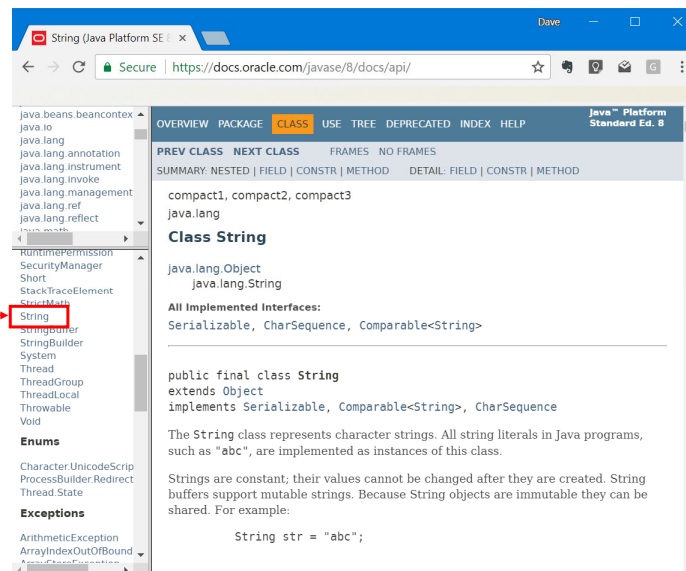
Consulting the Java API (cont.)

- Scroll down to see a summary of the available methods:
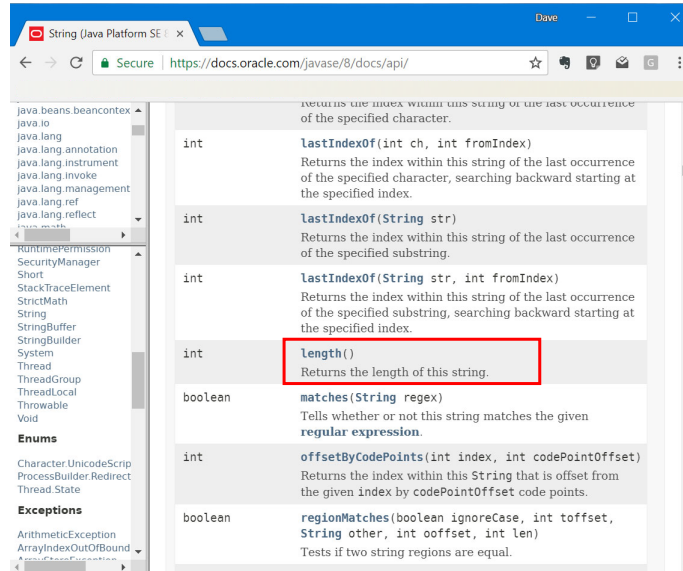
Consulting the Java API (cont.)

- Clicking on a method name gives you more information:
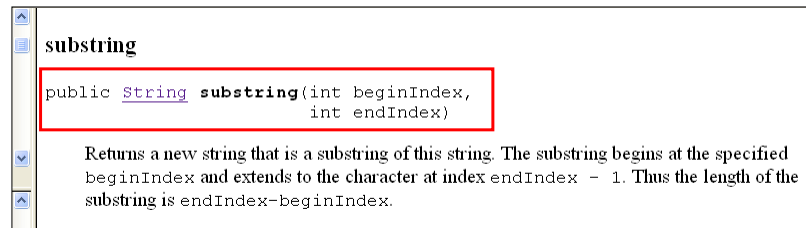
- From the header, we can determine:
  - the return type: `int`
  - the parameters we need to supply:
    the empty `()` indicates that `length` has no parameters

substring Method

substring

```
public String substring(int beginIndex,
                        int endIndex)
```

Returns a new string that is a substring of this string. The substring begins at the specified `beginIndex` and extends to the character at index `endIndex` - 1. Thus the length of the substring is `endIndex-beginIndex`.

`String substring(int beginIndex, int endIndex)`

* return type:

* parameters:
* behavior: returns the substring that:
    * begins at `beginIndex`
    * ends at `endIndex` – 1

Which of these correctly fills in the blank?

charAt

```
public char charAt(int index)
```

Returns the `char` value at the specified index. An index ranges from `0` to `length() - 1`.

```
String s = "PS 1 is due next week!";
int len = s.length();
_____  // get the last character in s
```

A. `String last = s.charAt(int len - 1);`

B. `String last = s.charAt(len - 1)`

C. `char last = s.charAt(int len - 1);`

D. `char last = s.charAt(len - 1);`

E. more than one of them

Adding chars…

charAt

```
public char charAt(int index)
```

Returns the char value at the specified index. An index ranges from 0 to length() - 1.

* Because charAt() returns a char,
  we have to be careful when we use its return value!
    * example:
      ```
      String s = "Perry Sullivan";
      System.out.println(s.charAt(0) + s.charAt(6));
      ```

      output:
      ??

Adding chars…*fixed!*

charAt

```
public char charAt(int index)
```

Returns the char value at the specified index. An index ranges from 0 to length() - 1.

* Because charAt() returns a char,
  we have to be careful when we use its return value!
    * example:
      ```
      String s = "Perry Sullivan";
      System.out.println(s.charAt(0) + "" + s.charAt(6));
      ```

After running this code, what are `s1` and `s3`?

```
String s1 = "Go";
String s2 = "Terriers!";
String s3 = s1 + " " + s2;
s2.substring(0, 5);
s1 = s2.toUpperCase();
```

	s1	s3
A.	`"TERRI"`	`"Go Terriers!"`
B.	`"TERRI"`	`"TERRI Terri"`
C.	`"TERRIERS!"`	`"Go Terriers!"`
D.	`"TERRIERS!"`	`"TERRIERS! Terriers!"`
E.	none of these	

Processing a String One Character at a Time

* Let's say that we want a method that prints a string vertically, one character per line.
    * example: `printVertical("hello")` should print:
        ```
        h
        e
        l
        l
        o
        ```

* Use a `for` loop, along with calls to methods inside the string:

    ```
    public static void printVertical(String s){

        for (int i = 0; i < _____; i++) {



        }
    }
    ```

Testing for Equivalent Primitive Values

* The `==` and `!=` operators are used to compare primitives.
    * `int`, `double`, `char`, etc.

* Example:
```
Scanner console = new Scanner(System.in);
System.out.print("Choose an option: ");
int choice = console.nextInt();
if (choice == 1) {     // this works just fine
    playSudoku();
} else if (choice == 2) {
    playChess();
} else {
    System.out.println("invalid input");
}
```

Testing for Equivalent Objects

* The `==` and `!=` operators do *not* typically work when comparing *objects*.

* Example:
```
Scanner console = new Scanner(System.in);
System.out.print("regular or diet? ");
String choice = console.next();
if (choice == "regular") {  // doesn't work
    processRegular();
} else {
    ...
}
```

* `choice == "regular"` compiles, but it evaluates to `false`, even when the user does enter `"regular"`!
    * it compares the memory addresses of the two strings, rather than their values!

Testing for Equivalent Objects (cont.)

- To test for equivalent objects, use a method called `equals`:
  - example:
    ```
    Scanner console = new Scanner(System.in);
    System.out.print("regular or diet? ");
    String choice = console.next();
    if (choice.equals("regular")) {
        processRegular();
    } else {
        ...
    }
    ```

- `choice.equals("regular")` compares the string represented by the variable `choice` with the string `"regular"`
  - returns `true` when they are equivalent
  - returns `false` when they are not

Pre-Lecture
From Python to Java:
Lists / Arrays, Part I

Computer Science 112
Boston University

Sequences

- A *sequence* is a collection of values in which each value has a *position* or *index.*

0	1	2	3	4	← indices
51	50	36	29	30	← elements

- The values are known as *elements* of the sequence.

Sequences in Python and Java

- In Python, a *list* is a sequence of *arbitrary* values.

      [2, 4, 6, 8]
      ['CS', 'math', 'english', 'psych']

  - the elements can have arbitrary types:

        ['Star Wars', 1977, 'PG', [35.9, 460.9]]

- Java provides a similar construct known as an *array*.
  - the elements must have the same type
  - less flexible than a list, with less built-in functionality
  - it also has less overhead, and it's easier to use efficiently

Array Variables

- We use a variable to represent the array as a whole.

- Example:

      int[] temps;

    - the [] indicates that it will represent an array
    - the int indicates that the elements will be ints

Basic Operations on Lists / Arrays

Python

```python
temps = [51, 50, 36, 29, 30]
first = temps[0]
num_temps = len(temps)
last = temps[-1]

temps[2] = 40
temps[3] += 5
print(temps[3])
print(temps)
```

Java

```java
int[] temps = {51, 50, 36, 29, 30};
int first = temps[0];
int numTemps = temps.length;
int last = temps[numTemps - 1];

temps[2] = 40;
temps[3] += 5;
System.out.println(temps[3]);
System.out.println(temps);
```

- Python uses [ ] to both:
  - surround list literals
  - index into the list

- Java uses:
  - { } to surround array literals
  - [ ] to index into the array
    - cannot use negative indices

Basic Operations on Lists / Arrays (cont.)

Python

```python
temps = [51, 50, 36, 29, 30]
first = temps[0]
num_temps = len(temps)
last = temps[-1]

temps[2] = 40
temps[3] += 5
print(temps[3])
print(temps)
```

Java

```java
int[] temps = {51, 50, 36, 29, 30};
int first = temps[0];
int numTemps = temps.length;
int last = temps[numTemps - 1];

temps[2] = 40;
temps[3] += 5;
System.out.println(temps[3]);
System.out.println(temps);  // no!
```

- `len(values)` gives the length of the list `values`

- `values.length` gives the length of the array `values`
  - `length` is *not* a method
  - in strings, it is: `s.length()`

- Printing a list displays its contents.

- Printing an array does *not* display its contents.

Arrays and References

- Arrays are objects.

- Thus, an array variable does *not* store the array itself.

- Rather, it stores a *reference* to the array.
  - the memory address of the array

```
int[] temps = {51, 50, 36, 29, 30};
```
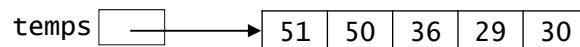
temps → | 51 | 50 | 36 | 29 | 30 |

- Printing an array displays a value based on its address!

```
System.out.println(temps);
```

output:
[I@1e1fd124

Printing the Contents of an Array

```java
import java.util.*;

public class FunWithArrays {
    public static void main(String[] args) {
        int[] temps = {51, 50, 36, 29, 30};
        int first = temps[0];
        int numTemps = temps.length;
        int last = temps[numTemps - 1];

        temps[2] = 40;
        temps[3] += 5;
        System.out.println(temps[3]);
        System.out.println(Arrays.toString(temps));
    }
}
```

```
temps [    ]
first [    ]
numTemps [    ]
last [    ]
```

output:

Pre-Lecture
From Python to Java:
Lists / Arrays, Part II

Computer Science 112
Boston University

Other Differences

Python	**Java**

```python
temps = [51, 50, 36, 29, 30]
first_two = temps[0:2]
temps = temps + [45, 29]
new_temps = [65] * 5
```

```java
int[] temps = {51, 50, 36, 29, 30};
// no operator for slicing!
// no operator for concatenating!
// no operator for multiplying!
```

* To get similar functionality in Java:
    * the `Arrays` class has *static* methods that take an array
        * example: `Arrays.copyOfRange(values, start, end)`
          returns the slice `values[start : end]`
    * there are built-in *collection classes* for lists
        * they can be used instead of an array
        * objects of these classes have *non-static* methods
          (methods inside them) for list operations
        * we'll soon be building our own collection classes!

Constructing and Filling a List / Array

Python

```
temps = [0] * 4
print('enter 4 temps:')
temps[0] = int(input())
temps[1] = int(input())
temps[2] = int(input())
temps[3] = int(input())
print(temps)
```

Java

```
Scanner scan = new Scanner(System.in);
int[] temps = new int[4];
System.out.println("enter 4 temps:");
temps[0] = scan.nextInt();
temps[1] = scan.nextInt();
temps[2] = scan.nextInt();
temps[3] = scan.nextInt();
System.out.println(
  Arrays.toString(temps));
```

* General pattern:

  type[] *variable* = new *type*[*length*];

  ```
  double[] vals = new double[100];   // room for 100 doubles
  String[] names = new String[10];   // room for 10 Strings
  ```

* Initially, the arrays are filled with the default value of their type:

int	0	boolean	false
double	0.0	objects	the special value null

Constructing and Filling a List / Array (cont.)

Python

```
temps = [0] * 100
print('enter 100 temps:')
for i in range(100):
    temps[i] = int(input())
print(temps)
```

Java

```
Scanner scan = new Scanner(System.in);
int[] temps = new int[100];
System.out.println("enter 100 temps:");
for (int i = 0; i < 100; i++) {
    temps[i] = scan.nextInt();
}
System.out.println(
  Arrays.toString(temps));
```

To make the code more flexible…

```
temps = [0] * 100
print('enter 100 temps:')
for i in range(len(temps)):
    temps[i] = int(input())
print(temps)
```

```
Scanner scan = new Scanner(System.in);
int[] temps = new int[100];
System.out.println("enter 100 temps:");
for (int i = 0; i < temps.length; i++) {
    temps[i] = scan.nextInt();
}
System.out.println(
  Arrays.toString(temps));
```

Processing a Sequence Using a Loop

Index-based:

Python	**Java**
```for i in range(len(list)):``` *do something with list*[i]	```for (int i = 0; i < array.length; i++) {``` *do something with array*[i] ```}```

where *list* is the list variable

where *array* is the array variable

Element-based:

Python	**Java**
```for val in list:``` *do something with* val	```for (int val : array) {``` *do something with* val ```}```

where *list* is the list variable

where *array* is the array variable

- Index-based is more flexible:
  - you can use it to *change* the element with index i
  - you can keep track of where you saw a given value

From Python to Java:
Lists / Arrays

Computer Science 112
Boston University

Recall: Sequences in Python and Java

- In Python, a *list* is a sequence of *arbitrary* values.
  ```
  [2, 4, 6, 8]
  ['CS', 'math', 'english', 'psych']
  ```
  - the elements can have arbitrary types:
    ```
    ['Star Wars', 1977, 'PG', [35.9, 460.9]]
    ```

- Java provides a similar construct known as an *array*.
  - the elements must have the same type
  - less flexible than a list, with less built-in functionality
  - it also has less overhead
    - example: a Java array of 1000 integers
      will use much less memory than
      a Python list of 1000 integers
  - it is easier to use it efficiently

Which of the following is valid Java?

A. `int[] vals = [2, 4, 5, 7, 3];`

B. `int vals = {2, 4, 5, 7, 3};`

C. `int[] vals = [2.0, 4, 5.0, 7.0, 3];`

D. `int[] vals = {2, 4, 5, 7, 3};`

What does this print?

```java
import java.util.*;

public class FunWithArrays {
    public static void main(String[] args) {
        int[] vals = {2, 4, 5, 7, 3};
        vals[1] = 6;
        vals[2] *= vals[1];
        System.out.println(Arrays.toString(vals));
    }
}
```

What does this modified version print?

```java
import java.util.*;

public class FunWithArrays {
    public static void main(String[] args) {
        int[] vals = {2, 4, 5, 7, 3};
        vals[1] = 6;
        vals[2] *= vals[1];
        System.out.println(vals);
    }
}
```

How can I make a list/array with room for 10 ints?

Python

```python
temps = [0] * 10
```

Java

```java
int[] temps = _____;
```

A Method for Finding the Smallest Value

```java
public static int minVal1(int[] values) {
    int min = values[0];
    for (int i = 0; _____; i++) {
        if (_____ < min) {
            min = _____;
        }
    }
    return min;
}
```

	first blank	second and third blanks
A.	i <= values.length	i
B.	i < values.length()	i
C.	i <= values.length()	values[i]
D.	i < values.length	values[i]
E.	i < values.length()	values[i]

Here's the method in the context of a program…

```java
public class ArrayMethods {
    /*
     * minVal1 - uses an index-based loop to find
     * and return the smallest value in the array values.
     */
    public static int minVal1(int[] values) {
        int min = values[0];
        for (int i = 0; _____; i++) {
            if (_____ < min) {
                min = _____;
            }
        }
        return min;
    }
    public static void main(String[] args) {
        int[] values = {7, 8, 9, 6, 10, 7, 9, 5};
        int min = minVal1(values);
        System.out.println("the min value is " + min);
    }
}
```

Finding the Smallest Value, version 2

```java
public class ArrayMethods {
    /*
     * minVal2 - uses an element-based loop to find
     * and return the smallest value in the array values.
     */
    public static int minVal2(int[] values) {
        int min = values[0];
        for (int val : values) {
            if (val < min) {
                min = val;
            }
        }
        return min;
    }

    public static void main(String[] args) {
        int[] values = {7, 8, 9, 6, 10, 7, 9, 5};
        int min = minVal2(values);
        System.out.println("the min value is " + min);
    }
}
```

What if we wanted to do this instead?

```java
public class ArrayMethods {
    /*
     * minIndex - uses an _____ loop to find and
     * and return the index of the smallest value in values.
     */
    public static int minIndex(int[] values) {
        int minIndex = _____;
        for (_____) {
            if (_____) {

            }
        }
        return minIndex;
    }

    public static void main(String[] args) {
        int[] values = {7, 8, 9, 6, 10, 7, 9, 5};
        int minInd = minIndex(values);
        System.out.println("min value is at index " + minInd);
    }
}
```

try this later on your own!

Copying a Reference Variable

- When we assign the value of one reference variable to another, we copy the reference to the object.
  We do *not* copy the object itself.

- Example involving objects:
  ```
  String s1 = "hello, world";
  String s2 = s1;
  ```

  s1 [→] ────────→ "hello, world"
  s2 [→] ───↗
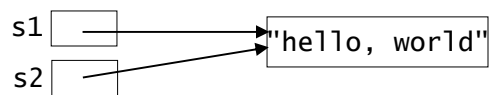
Copying a Reference Variable

- What does this do?
  ```
  int[] values = {7, 8, 9, 6, 10, 7, 9, 5};
  int[] other = values;
  ```

  values [→] ────────→ | 7 | 8 | 9 | 6 | 10 | 7 | 9 | 5 |

  other [   ]

- Given the lines of code above, what will the lines below print?
  ```
  other[2] = 4;
  System.out.println(values[2] + " " + other[2]);
  ```

Copying an Array

- To actually create a copy of an array, we can:
  - create a new array of the same length as the first
  - traverse the arrays and copy the individual elements

- Example:
```
int[] values = {7, 8, 9, 6, 10, 7, 9, 5};
int[] other = new int[values.length];
for (int i = 0; i < values.length; i++) {
    other[i] = values[i];
}
```

values	→	7	8	9	6	10	7	9	5
other	→	0	0	0	0	0	0	0	0
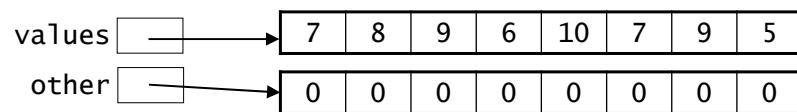
Copying an Array (cont.)

- To actually create a copy of an array, we can:
  - create a new array of the same length as the first
  - traverse the arrays and copy the individual elements

- Example:
```
int[] values = {7, 8, 9, 6, 10, 7, 9, 5};
int[] other = new int[values.length];
for (int i = 0; i < values.length; i++) {
    other[i] = values[i];
}
```
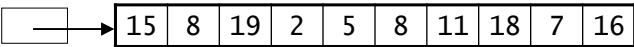
values	→	7	8	9	6	10	7	9	5
other	→	7	8	9	6	10	7	9	5
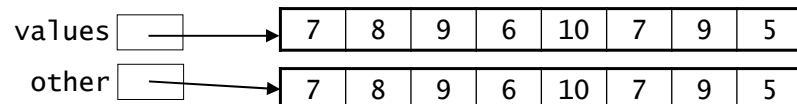
- What do the following lines print now?
```
other[2] = 4;
System.out.println(values[2] + " " + other[2]);
```

Shifting Values in an Array

- Let's say a small business is using an array to store the number of items sold in the past ten days.
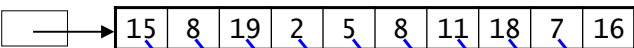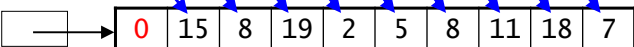
numSold ⟶ | 15 | 8 | 19 | 2 | 5 | 8 | 11 | 18 | 7 | 16 |

    numSold[0] gives the number of items sold today
    numSold[1] gives the number of items sold 1 day ago
    numSold[2] gives the number of items sold 2 days ago
    etc.

Shifting Values in an Array (cont.)

- At the start of each day, it's necessary to shift the values over to make room for the new day's sales.

numSold ⟶ | 15 | 8 | 19 | 2 | 5 | 8 | 11 | 18 | 7 | 16 |

numSold ⟶ | 0 | 15 | 8 | 19 | 2 | 5 | 8 | 11 | 18 | 7 |

  - the last value is lost, since it's now 10 days old

- In order to shift the values over, we need to perform assignments like the following:

```
numSold[9] = numSold[8];
numSold[6] = numSold[5];
numSold[2] = numSold[1];
```

  - what is the general form (the pattern) of these assignments?

Shifting Values in an Array (cont.)

* Here's one attempt at code for shifting all of the elements:

```
for (int i = 0; i < numSold.length; i++) {
    numSold[i] = numSold[i - 1];
}
```

* If we run this, we get an `ArrayIndexOutOfBoundsException`. Why?

Shifting Values in an Array (cont.)

* This version of the code eliminates the exception:

```
for (int i = 1; i < numSold.length; i++) {
    numSold[i] = numSold[i - 1];
}
```

* Let's trace it to see what it does:

numSold → | 15 | 8 | 19 | 2 | 5 | 8 | 11 | 18 | 7 | 16 |

* when i == 1, we perform `numSold[1] = numSold[0]` to get:

numSold → | 15 | *15* | 19 | 2 | 5 | 8 | 11 | 18 | 7 | 16 |

* when i == 2, we perform `numSold[2] = numSold[1]` to get:

numSold → | 15 | 15 | *15* | 2 | 5 | 8 | 11 | 18 | 7 | 16 |

this obviously doesn't work!

Shifting Values in an Array (cont.)

- How can we fix this code so that it does the right thing?

```
for (int i = 1; i < numSold.length; i++) {
    numSold[i] = numSold[i - 1];
}
```

⬇

```
for (          ;                  ;          ) {


}
```

- After performing all of the shifts, we would do: `numSold[0] = 0;`

numSold ☐→ | 15 | 15 | 8 | 19 | 2 | 5 | 8 | 11 | 18 | 7 |

⬇

numSold ☐→ | 0 | 15 | 8 | 19 | 2 | 5 | 8 | 11 | 18 | 7 |
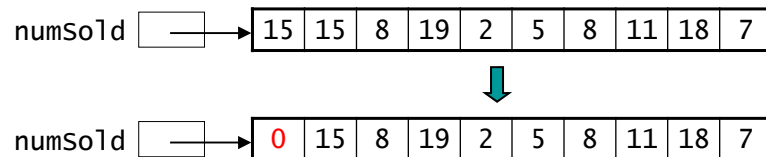
"Growing" an Array

- Once we have created an array, we can't increase its size.

- Instead, we need to do the following:
    1. create a new, larger array (use a temporary variable)
    2. copy the contents of the original array into the new array
    3. assign the new array to the original array variable

- Example for our `values` array:

```
int[] values = {7, 8, 9, 6, 10, 7, 9, 5};
...
int[] temp = new int[16];
for (int i = 0; i < values.length; i++) {
    temp[i] = values[i];
}
values = temp;
```

Pre-Lecture
From Python to Java: Writing Your Own Classes

Computer Science 112
Boston University

Recall: Classes As Blueprints

- A *class* is a blueprint – a definition of a data type.
  - specifies the data values and methods of that type

- Objects are built according to the blueprint provided by their class.
  - they are "values" / *instances* of that type

Example: A Rectangle Class

- Let's say that we want to create a data type for objects that represent rectangles.

- Every Rectangle object should have two variables inside it (`width` and `height`) for the rectangle's dimensions.
  - these variables are referred to as *fields*

width	200
height	150

- We'll also put functions/methods inside the object.

An Initial `Rectangle` Class

Python

```python
class Rectangle:
    def __init__(self, w, h):
        self.width = w
        self.height = h
```

Java

```java
public class Rectangle {
    int width;
    int heigh      non-static, no return type
    public Rectangle(int w, int h) {
        this.width = w;
        this.height = h;
    }
}
```

- `__init__` is the *constructor.*
  - used to create objects of the class

- The constructor has the same name as the class.

An Initial `Rectangle` Class

Python

```python
class Rectangle:
    # do not declare
    # the fields!

    def __init__(self, w, h):
        self.width = w
        self.height = h
```

Java

```java
public class Rectangle {
    int width;
    int height;

    public Rectangle(int w, int h) {
        this.width = w;
        this.height = h;
    }
}
```

* The fields are defined by assigning something to them in the constructor.

* The fields must be declared separately.
    * outside of any method
    * usually near the class header

Blueprint Class vs. Client Program

```java
public class Rectangle {
    int width;
    int height;

    public Rectangle(int w, int h) {
        this.width = w;
        this.height = h;
    }
}
```

r1 → width `100`
     height `50`

r2

```java
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(100, 50);
        Rectangle r2 = new Rectangle(20, 80);

        int area1 = r1.width * r1.height;
        System.out.println("r1's area = " + area1);

        int area2 = r2.width * r2.height;
        System.out.println("r2's area = " + area2);

        // grow both rectangles
        r1.width += 50;   r1.height += 10;
        r2.width += 5;    r2.height += 30;

        System.out.println("r1: " + r1.width + " x " + r1.height);
        System.out.println("r2: " + r2.width + " x " + r2.height);
    }
}
```

Adding Functionality to an Object

Python

```
class Rectangle:
    // do not declare
    // the fields!

    def __init__(self, w, h):
        self.width = w
        self.height = h

    def grow(self, dw, dh):
        self.width += dw
        self.height += dh
```

Java

```
public class Rectangle {
    int width;
    int height;

    public Rectangle(int w, int h) {
        this.width = w;
        this.height = h;
    }
            non-static
    public void grow(int dw, int dh) {
        this.width += dw;
        this.height += dh;
    }
}
```

- self *is* in the param list.

- it gets its value from the called object
  - ex: `r1.grow(50,10)`

- `this` is *not* in the parameter list

- it also gets its value from the called object
  - ex: `r1.grow(50,10)`

Adding Functionality to an Object

Python

```
class Rectangle:
    // do not declare
    // the fields!

    def __init__(self, w, h):
        self.width = w
        self.height = h

    def grow(self, dw, dh):
        self.width += dw
        self.height += dh

    def area(self):
        return self.width
               * self.height
```

Java

```
public class Rectangle {
    int width;
    int height;

    public Rectangle(int w, int h) {
        this.width = w;
        this.height = h;
    }

    public void grow(int dw, int dh) {
        this.width += dw;
        this.height += dh;
    }

    public int area() {
        return this.width * this.height;
    }
}
```

Simplifying the Client Program: Before

```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(100, 50);
        Rectangle r2 = new Rectangle(20, 80);

        int area1 = r1.width * r1.height;
        System.out.println("r1's area = " + area1);

        int area2 = r2.width * r2.height;
        System.out.println("r2's area = " + area2);

        // grow both rectangles
        r1.width += 50;   r1.height += 10;
        r2.width += 5;    r2.height += 30;

        System.out.println("r1: " + r1.width + " x " + r1.height);
        System.out.println("r2: " + r2.width + " x " + r2.height);
    }
}
```

Simplifying the Client Program: After

```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(100, 50);
        Rectangle r2 = new Rectangle(20, 80);

        int area1 = r1.area();
        System.out.println("r1's area = " + area1);

        int area2 = r2.area();
        System.out.println("r2's area = " + area2);

        // grow both rectangles
        r1.grow(50, 10);
        r2.grow(5, 30);

        System.out.println("r1: " + r1.width + " x " + r1.height);
        System.out.println("r2: " + r2.width + " x " + r2.height);
    }
}
```

Pre-Lecture
From Python to Java:
Adding Methods to an Object

Computer Science 112
Boston University

Recall: A Rectangle Class

- Let's say that we want to create a data type for objects that represent rectangles.

- Every Rectangle object should have
  two variables inside it (`width` and `height`)
  for the rectangle's dimensions.
  - these variables are referred to as *fields*

| width | 200 |
| height | 150 |

- We'll also put functions/methods inside the object.

Adding Functionality to an Object

Python

```
class Rectangle:
    // do not declare
    // the fields!

    def __init__(self, w, h):
        self.width = w
        self.height = h

    def grow(self, dw, dh):
        self.width += dw
        self.height += dh
```

Java

```
public class Rectangle {
    int width;
    int height;

    public Rectangle(int w, int h) {
        this.width = w;
        this.height = h;
    }                    non-static
    public void grow(int dw, int dh) {
        this.width += dw;
        this.height += dh;
    }
}
```

* self *is* in the param list.

* it gets its value from the called object

    * ex: `r1.grow(50,10)`

* `this` is *not* in the parameter list

* it also gets its value from the called object

    * ex: `r1.grow(50,10)`

Adding Functionality to an Object

Python

```
class Rectangle:
    // do not declare
    // the fields!

    def __init__(self, w, h):
        self.width = w
        self.height = h

    def grow(self, dw, dh):
        self.width += dw
        self.height += dh

    def area(self):
        return self.width
                * self.height
```

Java

```
public class Rectangle {
    int width;
    int height;

    public Rectangle(int w, int h) {
        this.width = w;
        this.height = h;
    }

    public void grow(int dw, int dh) {
        this.width += dw;
        this.height += dh;
    }

    public int area() {
        return this.width * this.height;
    }
}
```

Simplifying the Client Program: Before

```java
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(100, 50);
        Rectangle r2 = new Rectangle(20, 80);

        int area1 = r1.width * r1.height;
        System.out.println("r1's area = " + area1);

        int area2 = r2.width * r2.height;
        System.out.println("r2's area = " + area2);

        // grow both rectangles
        r1.width += 50;   r1.height += 10;
        r2.width += 5;    r2.height += 30;

        System.out.println("r1: " + r1.width + " x " + r1.height);
        System.out.println("r2: " + r2.width + " x " + r2.height);
    }
}
```

Simplifying the Client Program: After

```java
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(100, 50);
        Rectangle r2 = new Rectangle(20, 80);

        int area1 = r1.area();
        System.out.println("r1's area = " + area1);

        int area2 = r2.area();
        System.out.println("r2's area = " + area2);

        // grow both rectangles
        r1.grow(50, 10);
        r2.grow(5, 30);

        System.out.println("r1: " + r1.width + " x " + r1.height);
        System.out.println("r2: " + r2.width + " x " + r2.height);
    }
}
```

From Python to Java:
Defining New Types of Objects

Memory Management

Computer Science 112
Boston University

Recall: Classes As Blueprints

- A *class* is a blueprint – a definition of a data type.
  - specifies the data values and methods of that type

- Objects are built according to the blueprint provided by their class.
  - they are "values" / *instances* of that type

Another Analogy

- A class is like a cookie cutter.
  - specifies the "shape" that all objects of that type should have

- Objects are like the cookies.
  - created with the "shape" specified by their class

Another Example: A Class for `Time` Objects

- Let's say that we want to create a data type for objects that represent military times (e.g., 10:30 or 17:50).

- A `Time` object for 10:30 would look like this:

```
  hour   | 10
  minute | 30
```
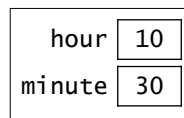
- We would create it as follows:

```
Time t1 = new Time(10, 30);
```

```
t1 [ —— ] ———→   hour   | 10
                 minute | 30
```
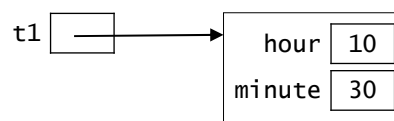
Which of these is a valid initial `Time` class?

A.

```
public class Time {
    public Time(int h, int m) {
        self.hour = h;
        self.min = m;
    }
}
```

B.

```
public class Time {
    int hour;
    int min;

    public Time(int h, int m) {
        this.hour = h;
        this.min = m;
    }
}
```

C.

```
public class Time {
    public void Time(int h, int m){
        self.hour = h;
        self.min = m;
    }
}
```

D.

```
public class Time {
    int hour;
    int min;

    public void Time(int h, int m) {
        this.hour = h;
        this.min = m;
    }
}
```

Which method call increases r's height by 5?

```
public class Rectangle {
    int width;
    int height;

    public Rectangle(int w, int h) {
        this.width = w;
        this.height = h;
    }
    public void grow(int dw, int dh) {
        this.width += dw;
        this.height += dh;
    }
    public int area() {
        return this.width * this.height;
    }
}
```

```
public class MyClient {
    public static void
    main(String[] args) {
        Rectangle r = new Rectangle(10, 15);
        _____???_____;

    }
}
```
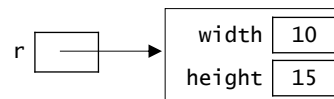
r → | width | 10 |
    | height | 15 |

Static vs. Non-Static

- When writing a method for growing a `Rectangle`,
  we could in theory have written a static method:

```
public static void grow(Rectangle r, int dw, int dh) {
    r.width += dw;
    r.height += dh;
}
```

- This would allow us to replace these statements in the client

```
    r1.width += 50;  r1.height += 10;
```

  with the method call

```
    Rectangle.grow(r1, 50, 10);
```

  (Note: We need to use the class name, because we're calling
  the method from outside the `Rectangle` class.)

Static vs. Non-Static (cont.)

- A better approach is to give each `Rectangle` object
  the ability to grow itself.

- This is what our non-static `grow()` method does.
  - also known as an *instance method*

```
public void grow(int dw, int dh) { // no static
    this.width += dw;
    this.height += dh;
}
```

- We <u>don't</u> pass the `Rectangle` object as an explicit parameter.

- Instead, the Java keyword `this` gives us access to
  the *called object*.
  - every non-static method has this special variable
  - referred to as the *implicit parameter*

Comparing the Static and Non-Static Versions

* Static:
```
public static void grow(Rectangle r, int dw, int dh) {
    r.width += dw;
    r.height += dh;
}
```

    * sample method call: `Rectangle.grow(r1, 50, 10);`

* Non-static:
```
public void grow(int dw, int dh) {
    this.width += dw;
    this.height += dh;
}
```

    * there's no keyword `static` in the method header
    * the `Rectangle` object is not an explicit parameter
    * the implicit parameter `this` gives access to the object
    * sample method call: `r1.grow(50, 10);`

Types of Instance Methods

* There are two main types of instance methods:
    * *mutators* – methods that change an object's internal state
    * *accessors* – methods that retrieve information from an object without changing its state

* Examples of mutators:
    * `grow()` in our `Rectangle` class

* Examples of accessors:
    * `area()` in our `Rectangle` class
    * `String` methods: `length()`, `substring()`, `charAt()`

Practice Defining Instance Methods

* Add a mutator method that *scales* the rectangle's dimensions by a specified factor (an integer).

```
public _____ scale(_____) {



}
```

* Add an accessor method that determines if the rectangle is a square (true or false).

```
public _____ isSquare(_____) {



}
```

Limiting Access to Fields

* The current version of our `Rectangle` class allows clients to directly access a `Rectangle` object's fields:

```
r1.width = 100;
r1.height += 20;
```

* This means that clients can make inappropriate changes:

```
r1.width = -100;
```

* To prevent this, we can declare the fields to be *private*:

```
public class Rectangle {
    private int x;
    private int y;
    private int width;
    private int height;
    ...
}
```

* This indicates that these fields can only be accessed or modified by methods that are part of the `Rectangle` class.

Limiting Access to Fields (cont.)

- Now that the fields are private, our client program won't compile:

```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(100, 50);
        Rectangle r2 = new Rectangle(20, 80);

        int area1 = r1.area();
        System.out.println("r1's area = " + area1);

        int area2 = r2.area();
        System.out.println("r2's area = " + area2);

        // grow both rectangles
        r1.grow(50, 10);
        r2.grow(5, 30);

        System.out.println("r1: " + r1.width + " x " + r1.height);
        System.out.println("r2: " + r2.width + " x " + r2.height);
    }
}
```

Adding Accessor Methods for the Fields

```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        this.width = w;
        this.height = h;
    }

    public int getWidth() {
        return this.width;
    }

    public int getHeight() {
        return this.height;
    }

    public void grow(int dw, int dh) {
        this.width += dw;
        this.height += dh;
    }
    ...
}
```

- These methods are *public*, which indicates that they <u>can</u> be used by code that is outside the Rectangle class.

Limiting Access to Fields (cont.)

- Now the client can use the accessor methods:

```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(100, 50);
        Rectangle r2 = new Rectangle(20, 80);

        int area1 = r1.area();
        System.out.println("r1's area = " + area1);

        int area2 = r2.area();
        System.out.println("r2's area = " + area2);

        // grow both rectangles
        r1.grow(50, 10);
        r2.grow(5, 30);

        System.out.println("r1: " + r1.getWidth() + " x " + r1.getHeight());
        System.out.println("r2: " + r2.getWidth() + " x " + r2.getHeight());
    }
}
```

Access Modifiers

- `public` and `private` are known as *access modifiers*.
  - they specify where a class, field, or method can be used

- A class is usually declared to be `public`:

    `public class Rectangle {`
  - indicates that objects of the class can be used anywhere, including in other classes

- Fields are usually declared to be `private`.

- Methods are usually declared to be `public`.

- We occasionally define `private` methods.
  - serve as *helper methods* for the `public` methods
  - cannot be invoked by code that is outside the class

Encapsulation

- *Encapsulation* is a key principle of object-oriented programming.

- It means that clients should *not* have *direct* access
  to the internals of objects of a class.
    - preventing inappropriate changes

- To achieve it, we must make the fields private.

```
public class Rectangle {
    private int width;
    private int height;

    ...
}
```

- To allow clients to have appropriate *indirect* access,
  we provide accessor and mutator methods.

Practice Exercise: Working with Private Fields

```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h){
        this.width = w;
        this.height = h;
    }
    public void grow(int dw, int dh){
        this.width += dw;
        this.height += dh;
    }
    public int area(){
        return this.width * this.height;
    }
}
```

```
// inside the main method
// of a client program
Rectangle r = new Rectangle(10, 15);
System.out.println(r.area());
r.width = r.width + 25;
r.height = r.height + 30;
System.out.println(r.width);
System.out.println(r.height);
```

- How many lines won't compile in the client above?

- Why can the client still call the `area()` method?

Practice Exercise: Working with Private Fields (cont.)

```
public class Rectangle {
  private int width;
  private int height;

  public Rectangle(int w, int h){
    this.width = w;
    this.height = h;
  }
  public void grow(int dw, int dh){
    this.width += dw;
    this.height += dh;
  }
  public int area(){
    return this.width * this.height;
  }
  public int getWidth() {
    return this.width;
  }
  public int getHeight() {
    return this.height;
  }
  public void setWidth(int w) {
    ...
  public void setHeight(int h) {
    ...
  }
```

```
// inside the main method
// of a client program
Rectangle r = new Rectangle(10, 15);
System.out.println(r.area());

r.width = r.width + 25;

r.height = r.height + 30;

System.out.println(r.width);

System.out.println(r.height);
```

- To allow a client to see a field's value, we need to add a public *accessor* method.
- To allow a client to change a field, we need to provide a public *mutator* method.

Make the necessary changes above.

A Common Mistake

```
public class Rectangle {
  private int width;
  private int height;

  public Rectangle(int w, int h){
    this.width = w;
    this.height = h;
  }
  public void grow(int dw, int dh){
    this.width += dw;
    this.height += dh;
  }
  public int area(){
    return this.width * this.height;
  }
  public int getWidth() {
    return this.width;
  }
  public int getHeight() {
    return this.height;
  }
}
```

```
// inside the main method
// of a client program
Rectangle r = new Rectangle(10, 15);
System.out.println(r.area());
r.getWidth() = r.getWidth() + 25;
r.getHeight() = r.getHeight() + 30;
System.out.println(r.getWidth());
System.out.println(r.getHeight());
```

- The above does <u>not</u> work!

- Accessors let us see a field's value, but they *don't* let us *change* it!

- The left-hand side of an assignment statement must be a variable!

Mutators: Allowing Only Appropriate Changes

```java
public void setWidth(int w) {
    if (w <= 0) {
        throw new IllegalArgumentException();
    }

    this.width = w;
}

public void setHeight(int h) {
    if (h <= 0) {
        throw new IllegalArgumentException();
    }

    this.height = h;
}
```

- Throwing an exception ends the method call early.
  - preventing the bad change
- Unless the client does something special,
  the exception will cause it to crash with an error!

Another Mutator…Fixed!

- Here's another mutator method that we already had:
```java
public void grow(int dw, int dh) {
    this.width += dw;
    this.height += dh;
}
```

- We fix it to prevent inappropriate changes:
```java
public void grow(int dw, int dh) {
    this.setWidth(this.width + dw);
    this.setHeight(this.height + dh);
}
```

  - rather than adding error-checking to this method, it calls the new mutator methods to:
    - do the error-checking for us
    - make the changes if appropriate
  - note: we use `this` to call other methods in the same object!

A Constructor…Fixed!

- To prevent bad initial values:

```
public Rectangle(int w, int h) {
    this.setWidth(w);
    this.setHeight(h);
}
```

  - here again, we take advantage of the error-checking done by setWidth and setHeight

  - if a bad value is passed in for w or h:
    - the mutator will throw an exception
    - that exception will end the constructor early, preventing the object from being created

---

What is printed?

```
Rectangle r1 = new Rectangle(20, 55);
Rectangle r2 = new Rectangle(20, 55);
Rectangle r3 = r1;
System.out.println((r1 == r2) + " " + (r1 == r3));
```

r1 [   ]

r2 [   ]

r3 [   ]

Testing for Equivalent Objects

- If o1 and o2 are variables that refer to objects,

    o1 == o2

    compares the *references* stored in the variables.

- It doesn't compare the objects themselves.

Testing for Equivalent Objects (cont.)

- To test for equivalent objects, we need an equals method.

- o1.equals(o2) should return:
  - true if object o1 is equivalent to object o2
  - false otherwise

- We already saw a method like this for String objects:
    ```
    Scanner console = new Scanner(System.in);
    System.out.print("regular or diet? ");
    String choice = console.next();
    if (choice.equals("regular")) {
        processRegular();
    } else {
        ...
    }
    ```

equals() Method for Our Rectangle Class

```
public boolean equals(Rectangle other) {
    if (other == null) {
        return false;
    } else if (this.width != other.width) {
        return false;
    } else if (this.height != other.height) {
        return false;
    } else {
        return true;
    }
}
```

* **Note:** The method is able to access the fields in other directly (without using accessor methods).

* Methods of a class can access the private fields of *any* object from their class.

equals() Method for Our Rectangle Class (cont.)

* Here's an alternative version:

```
public boolean equals(Rectangle other) {
    return (other != null
        && this.width == other.width
        && this.height == other.height);
}
```

Converting an Object to a String

- The `toString()` method allows objects to be displayed in a human-readable format.
    - it returns a string representation of the object

- This method is *called for us* when we:
    - attempt to print an object:
      ```
      Rectangle r1 = new Rectangle(10, 20);
      System.out.println(r1);

      // the line above is equivalent to:
      System.out.println(r1.toString());
      ```

    - concatenate an object with a string:
      ```
      String result = "dimensions of " + r1;

      // the line above is equivalent to:
      String result = "dimensions of " + r1.toString();
      ```

`toString()` Method for Our `Rectangle` Class

```
public String toString() {
    return this.width + " x " + this.height;
}
```

- Note: the method does not do any printing.

- It returns a `String` that can then be printed.

Client Program Before `toString()`

```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(100, 50);
        Rectangle r2 = new Rectangle(20, 80);

        int area1 = r1.area();
        System.out.println("r1's area = " + area1);

        int area2 = r2.area();
        System.out.println("r2's area = " + area2);

        // grow both rectangles
        r1.grow(50, 10);
        r2.grow(5, 30);

        System.out.println("r1: " + r1.getWidth() + " x " + r1.getHeight());
        System.out.println("r2: " + r2.getWidth() + " x " + r2.getHeight());
    }
}
```

Client Program *After* `toString()`

```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(100, 50);
        Rectangle r2 = new Rectangle(20, 80);

        int area1 = r1.area();
        System.out.println("r1's area = " + area1);

        int area2 = r2.area();
        System.out.println("r2's area = " + area2);

        // grow both rectangles
        r1.grow(50, 10);
        r2.grow(5, 30);

        System.out.println("r1: " + r1);
        System.out.println("r2: " + r2);
    }
}
```

Memory Management, Type I: Stack Storage

- Method parameters and other local variables are stored in a region of memory known as *the stack*.

- For each method call, a new *stack frame* is added to the top of the stack.

```
public class Foo {
  public static int y(int i) {
      int j = i * 3;
      return j;
  }
  public static int x(int i) {
      int j = i - 2;
      return y(i + j);
  }
  public static void
    main(String[] args) {
      System.out.println(x(5));
  }
}
```

```
  i    8
                } y(8)
  j   24

  i    5
                } x(5)
  j    3

args
```
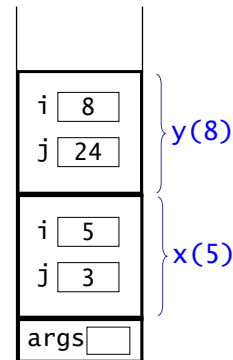
- When a method completes, its stack frame is removed.

Memory Management, Type II: Heap Storage

- Objects are stored in a memory region known as *the heap*.

- Memory on the heap is allocated using the new operator:

```
int[] values = new int[30];
Rectangle r = new Rectangle(10, 50);
```

- new returns the memory address of the start of the object on the heap.
  - a reference!

- An object stays on the heap until there are no remaining references to it.

- Unused objects are automatically reclaimed by a process known as *garbage collection*.
  - makes their memory available for other objects

Memory Management Example

```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(100, 50);
        Rectangle r2 = new Rectangle(20, 80);
        ...
        r1.grow(50, 10);
        r2.grow(5, 30);
        ...
```

* After the objects are created:
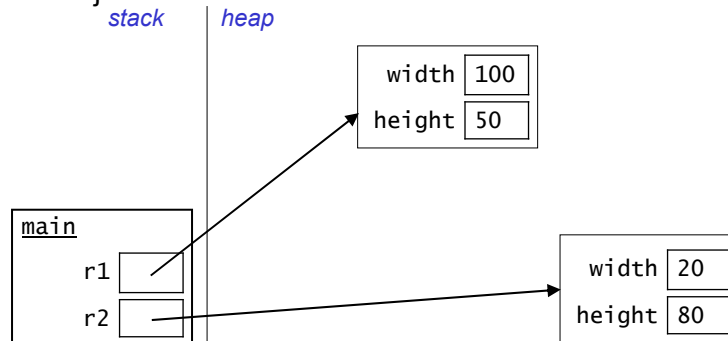
stack | *heap*

width	100
height	50

main

r1	
r2	

width	20
height	80

Memory Management Example

```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(100, 50);
        Rectangle r2 = new Rectangle(20, 80);
        ...
        r1.grow(50, 10);
        r2.grow(5, 30);
        ...
```

* During the method call `r1.grow(50, 10)`:

stack | *heap*

grow

this	
dw	50
dh	10

width	150
height	60

main

r1	
r2	

width	20
height	80

Memory Management Example

```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(100, 50);
        Rectangle r2 = new Rectangle(20, 80);
        ...
        r1.grow(50, 10);
        r2.grow(5, 30);
        ...
```
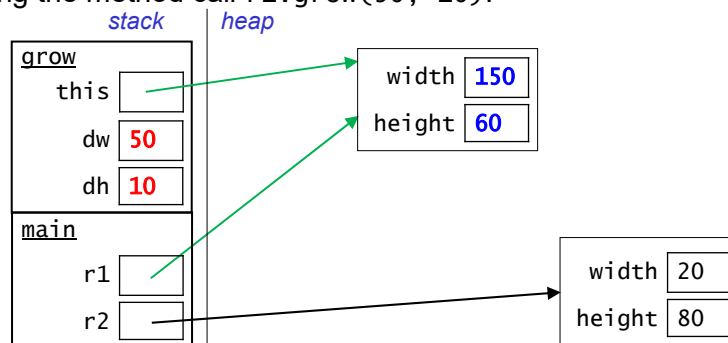
* <u>After</u> the method call `r1.grow(50, 10)`:

Memory Management Example

```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(100, 50);
        Rectangle r2 = new Rectangle(20, 80);
        ...
        r1.grow(50, 10);
        r2.grow(5, 30);
        ...
```
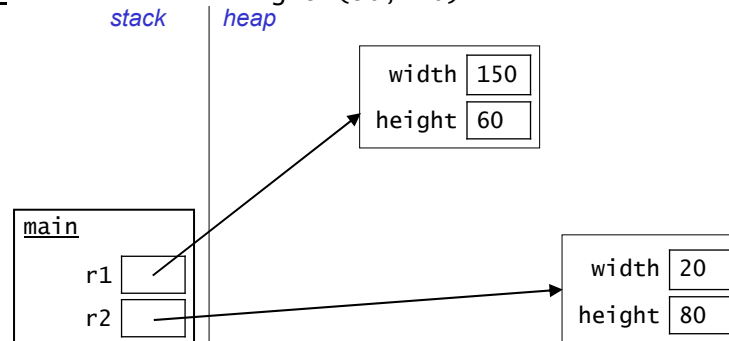
* During the method call `r2.grow(5, 30)`:
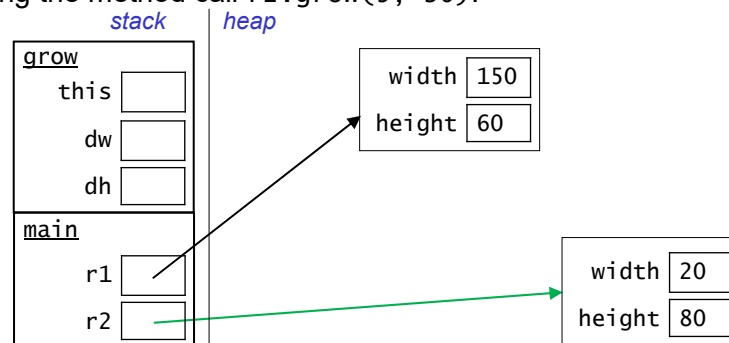
Memory Management Example

```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(100, 50);
        Rectangle r2 = new Rectangle(20, 80);
        ...
        r1.grow(50, 10);
        r2.grow(5, 30);
        ...
```

* After the method call `r2.grow(5, 30)`:
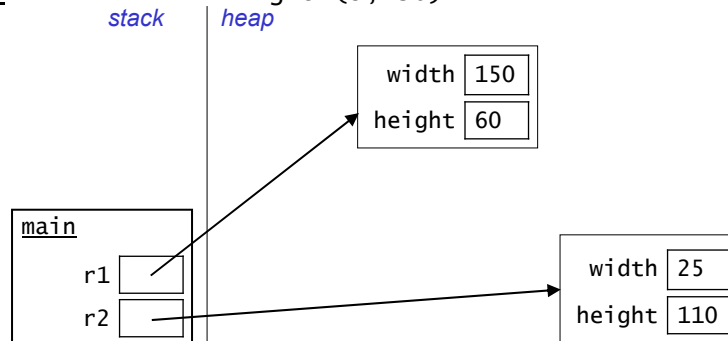
Why Mutators Don't Need to Return Anything

* A mutator operates directly on the called object,
  so any changes it makes will be there after the method returns.

  * example: the call `r2.grow(5, 30)` from the last slide
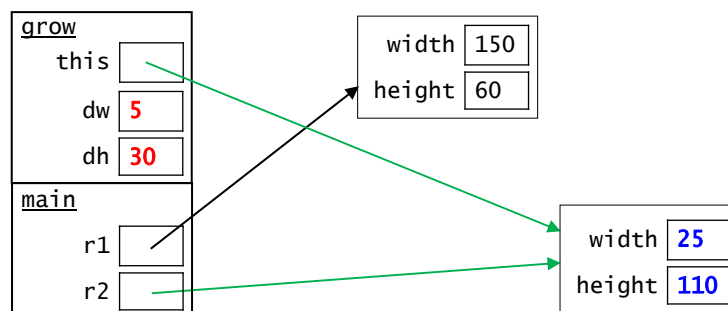
  * `grow` gets a copy of the reference in `r2`,
    so it changes the internals of the object to which `r2` refers

Memory Management, Type III: Static Storage

- Static storage is used for *class variables*, which are declared *outside any method* using the keyword `static`:

  ```
  public class MyMethods {
      public static int numCompares;
  ```

- There is only one copy of each class variable.
  - shared by all objects of the class
  - Java's version of a global variable

- To create a constant whose value can never change, make it both static and *final*:

  ```
          public static final double PI = 3.14159;
  ```

- The Java runtime allocates memory for class variables when the class is first encountered.
  - this memory stays fixed for the duration of the program

What if we want to keep track of how many `Rectangle` objects a program has created?

```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h){
        this.setWidth(w);
        this.setHeight(h);
    }
    public void grow(int dw, int dh){
        this.setWidth(this.width + dw);
        this.setHeight(this.height + dh);
    }
    public int area(){
        return this.width * this.height;
    }
    ...
}
```

What if we want to keep track of how many `Rectangle` objects a program has created?

```java
public class Rectangle {
    private int width;
    private int height;
    private static int numCreated = 0;

    public Rectangle(int w, int h){
        this.setWidth(w);
        this.setHeight(h);
        numCreated++;
    }
    public void grow(int dw, int dh){
        this.setWidth(this.width + dw);
        this.setHeight(this.height + dh);
    }
    public int area(){
        return this.width * this.height;
    }
    ...
}
```

Increment the static variable every time that we create a `Rectangle`.

* Add a static variable – one that belongs to the class as a whole.
   * *not* final – because we want to be able to change it!

Review: Which of these fixes the line of client code?

```java
Rectangle r = new Rectangle(25, 10);
r.height = r.width + 10;   // how can we fix this?
```

A. `r.getHeight() = r.getWidth() + 10;`

B. `r.setHeight(r.getWidth() + 10);`

C. `this.setHeight(this.getWidth() + 10);`

D. `Rectangle.setHeight(Rectangle.getWidth() + 10);`

E. more than one of the above

What gets printed? (Draw a diagram below!)

```java
public class Rectangle {
  private int width;
  private int height;

  public Rectangle(int w, int h){
    this.setWidth(w);
    this.setHeight(h);
  }
  public void grow(int dw, int dh){
    this.setWidth(this.width + dw);
    this.setHeight(this.height + dh);
  }
  public int area(){
    return this.width * this.height;
  }

  // accessors and mutators
  // for the fields

  public String toString() {
    return
      this.width + " x " + this.height;
  }
}
```

```java
// in main method of a new client
Rectangle r1 = new Rectangle(80, 40);
Rectangle r2 = r1;
r1.grow(20, 5);
System.out.println(r2);
```

Memory diagram for the above exercise

* <u>During</u> the call to grow():
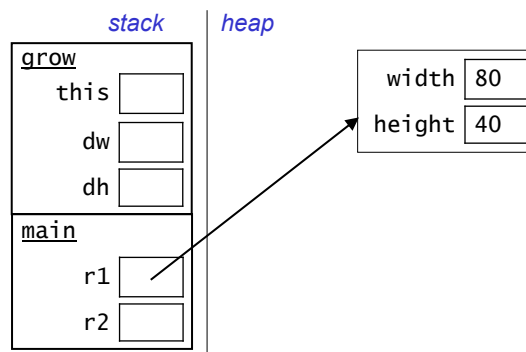
Recall: Functions / Methods

- Python distinguishes between:
    - *functions:* named blocks of code that:
        - take 0 or more inputs/parameters
        - return a value
    - *methods:* functions that are "inside" an object
        - have a `self` parameter

- In Java, both types of functions are called methods.
    - *static* methods – like Python functions
    - *non-static* or *instance* methods – like Python methods
        - methods that are "inside" an object
        - have a `this` parameter

Recall: Calling Functions / Static Methods

- If the function or static method is in the *current file*,
  just use its name:

Python	**Java**

```
avg = 85
letter_grade = grade(avg)
```

```
int avg = 85;
String letter_grade = grade(avg);
```

- If the function or static method is in a different module/class:
    - import the module or class as needed
    - prepend the module/class name

Python	**Java**

```
import math
x = math.sqrt(100)
```

```
// we don't need to import Math!
double x = Math.sqrt(100);
```

Recall: Calling Non-Static Methods

- Because non-static methods are inside an object,
  we prepend the name of the object:

Python

```
s1 = 'hello'
s2 = 'world'
s3 = s1.upper()
```

Java

```
String s1 = "hello";
String s2 = "world";
String s3 = s1.toUpperCase();
```

- Why can't we use the class name?
  ```
  String s2 = String.toUpperCase();
  ```

Which one makes a valid call to `methodOne()`?

- Consider a class called `Mystery` with the following API:

  ```
  public Mystery(int x)
  public static int methodOne(int b, int c)
  public double methodTwo(int d)
  ```

- You are writing client code that begins as follows:

  ```
  Mystery m1 = new Mystery(10);
  Mystery m2 = new Mystery(20);
  _____   // call methodOne()
  ```

  A.   `double d = Mystery.methodOne(5);`

  B.   `int n = Mystery.methodOne(5, 2);`

  C.   `double d = m1.methodOne(5, 2.5);`

  D.   `int n = m1.methodOne(10);`

  E.   more than one works

Which one makes a valid call to methodTwo()?

- Consider a class called Mystery with the following API:

```
public Mystery(int x)
public static int methodOne(int b, int c)
public double methodTwo(int d)
```

- You are writing client code that begins as follows:

```
Mystery m1 = new Mystery(10);
Mystery m2 = new Mystery(20);
_____    // call methodTwo()
```

A. `double d = Mystery.methodTwo(5);`

B. `int n = Mystery.methodTwo(5, 2);`

C. `int d = m1.methodTwo(5, 2.5);`

D. `double n = m1.methodTwo(10);`

E. more than one works

Pre-Lecture
From Python to Java: Inheritance

Computer Science 112
Boston University

Recall: A Class for `Rectangle` Objects

* Every `Rectangle` object has two fields:
  * `width`
  * `height`

width	200
height	150

* It also has methods inside it:
  * `grow()`
  * `area()`
  * `toString()`
  * etc.

Recall: A Class for `Rectangle` Objects

Python

```python
class Rectangle:
    // do not declare
    // the fields!

    def __init__(self, w, h):
        self.width = w
        self.height = h


    def grow(self, dw, dh):
        self.width += dw
        self.height += dh


    def area(self):
        return self.width
                * self.height
```

Java

```java
public class Rectangle {
    int width;
    int height;

    public Rectangle(int w, int h) {
        this.width = w;
        this.height = h;
    }

    public void grow(int dw, int dh) {
        this.width += dw;
        this.height += dh;
    }

    public int area() {
        return this.width * this.height;
    }
}
```

Squares == Special Rectangles!

- A square also has a width and a height.
  - but the two values must be the same

width	**40**
height	**40**
unit	**"cm"**

- Assume that we also want `Square` objects to have a field for the unit of measurement.

- `Square` objects should mostly behave like `Rectangle` objects:

  ```java
  Rectangle r = new Rectangle(20, 30);
  int area1 = r.area();

  Square sq = new Square(40, "cm");
  int area2 = sq.area();
  ```

- But there may be differences as well:

  ```java
  System.out.println(r);
  ```
  ⟹ output:
  `20 x 30`

  ```java
  System.out.println(sq);
  ```
  ⟹ output:
  `square with 40-cm sides`

Using Inheritance

Python

```
class Rectangle:
    // do not declare
    // the fields!

    def __init__(self, w, h):
        self.width = w
        self.height = h

    ...  # other methods
    def area(self):
        return self.width
                *self.height
```

Java

```
public class Rectangle {
    int width;
    int height;

    public Rectangle(int w, int h) {
        this.width = w;
        this.height = h;
    }
    ...  // other methods
    public int area() {
        return this.width * this.height;
    }
}
```

```
class Square(Rectangle):

    def __init__(self, side, unit):
        self.width = side
        self.height = side
        self.unit = unit

    # inherits other methods
```

```
public class Square extends Rectangle {
    String unit; // inherits other fields

    public Square(int side, String unit) {
        this.width = side;
        this.height = side;
        this.unit = unit;
    }

    // inherits other methods
}
```

Terminology

- Square is a *subclass* of Rectangle.

- Rectangle is a *superclass* of Square.

Using Inheritance

Python   **Java**

```
class Rectangle:
    // do not declare
    // the fields!

    def __init__(self, w, h):
        self.width = w
        self.height = h

    ...  # other methods
    def area(self):
        return self.width
                *self.height
```

```
public class Rectangle {
    int width;
    int height;

    public Rectangle(int w, int h) {
        this.width = w;
        this.height = h;
    }
    ...  // other methods
    public int area() {
        return this.width * this.height;
    }
}
```

```
class Square(Rectangle):

    def __init__(self, side, unit):
        # use superclass __init__
        super().__init__(side, side)
        self.unit = unit

    # inherits other methods
```

```
public class Square extends Rectangle {
    String unit; // inherits other fields

    public Square(int side, String unit) {
        // use superclass constructor
        super(side, side);
        this.unit = unit;
    }

    // inherits other methods
}
```

Example of an Inherited Method

* The `Rectangle` class has this `toString()` method:
    ```
    public String toString() {
        return this.width + " x " this.height;
    }
    ```

* The `Square` class inherits it from `Rectangle`.
    ```
    Square sq = new Square(40, "cm");
    System.out.println(sq);
    ```

    output:
    40 x 40

Overriding an Inherited Method

* A subclass can *override* / replace an inherited method with its own version, which must have the same:
  * return type
  * name
  * number and types of parameters

* Example: our `Square` class can define its own `toString()`:

```
public String toString() {
    String s = "square with ";
    s += this.width + "-" + this.unit + " sides";
}
```

* Printing a `Square` will now call this method, not the inherited one:

```
Square sq = new Square(40, "cm");
System.out.println(sq);
```

output:
square with 40-cm sides

Encapsulation

```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        this.setWidth(w);
        this.setHeight(h);
    }

    public int getWidth() {
        return this.width;
    }

    public int getHeight() {
        return this.height;
    }

    public void setWidth(int w) {
        if (w <= 0) {
            throw new IllegalArgumentException();
        }
        this.width = w;
    }

    public void setHeight(int w) {
        ...
```

Encapsulation and Inheritance

```
public class Rectangle {
    private int width;
    private int height;
    ...
    public int getWidth() {
        return this.width;
    }
    public int getHeight() {
        return this.height;
    }
    ...
```

```
public class Square extends Rectangle {
    private String unit;
    ...
    public String getUnit() {
        return this.unit;
    }
    public String toString() {
        String s = "square with ";
        s += this.width + "-" + this.unit + "sides";
        return s;
    }
}
```

Encapsulation and Inheritance

- A subclass has direct access to the *public* fields and methods of a superclass.
    - it **cannot** access its *private* fields and methods

- Example: we can think of a `Square` object as follows:
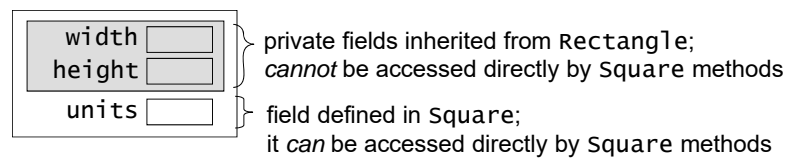
width ☐
height ☐
private fields inherited from `Rectangle`;
cannot be accessed directly by `Square` methods

units ☐
field defined in `Square`;
it *can* be accessed directly by `Square` methods

Encapsulation and Inheritance

```
public class Rectangle {
    private int width;
    private int height;
    ...
    public int getWidth() {
        return this.width;
    }
    public int getHeight() {
        return this.height;
    }
    ...
```

```
public class Square extends Rectangle {
    private String unit;
    ...
    public String getUnit() {
        return this.unit;
    }
    public String toString() {
        String s = "square with ";
        s += this.getWidth() + "-"
            + this.unit + "sides";
        return s;
    }
```

Writing a Constructor for a Subclass

- With private fields in `Rectangle`, this constructor won't compile:

  ```
  public Square(int side, String unit) {
      this.width = side;
      this.height = side;
      this.unit = unit;
  }
  ```

- To initialize inherited fields, a constructor should use `super()` to invoke a constructor from the superclass, as we did earlier:

  ```
  public Square(int side, String unit) {
      super(side, side);
      this.unit = unit;
  }
  ```

  - *must be done as the very first line of the constructor!*

From Python to Java: Inheritance and Polymorphism

Computer Science 112
Boston University

Recall: Using Inheritance

```java
public class Rectangle {
    int width;
    int height;

    public Rectangle(int w, int h) {
        this.setWidth(w);
        this.setHeight(h);
    }
    ... // other methods
    public String toString() {
        return this.width  + " x " + this.height;
    }
}
```

width	200
height	150

```java
public class Square extends Rectangle {
    String unit; // inherits other fields

    public Square(int side, String unit) {
        super(side, side);
        this.unit = unit;
    }
    public String toString() {  // overrides
        String s = "square with ";
        s += this.width + "-";
        s += this.unit + " sides";
        return s;
    } // inherits other methods
}
```

width	40
height	40
unit	"cm"

What change is needed if the fields are private?

```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        this.setWidth(w);
        this.setHeight(h);
    }
    ... // other methods
    public String toString() {
        return this.width  + " x " + this.height;
    }
}
```

width	200
height	150

```
public class Square extends Rectangle {
    private String unit; // inherits other fields

    public Square(int side, String unit) {
        super(side, side);
        this.unit = unit;
    }

    public String toString() {  // overrides
        String s = "square with ";
        s += this.width + "-";
        s += this.unit + " sides";
        return s;
    } // inherits other methods
}
```

width	40
height	40
unit	"cm"

Encapsulation and Inheritance (cont.)

- Faulty approach: redefine the inherited fields in the subclass

```
public class Rectangle {
    private int width;
    private int height;
    …
}

public class Square extends Rectangle {
    private int width;   // NOT a good idea!
    private int height;
    private String units;
    …
}
```

- You should <u>NOT</u> do this!

Another Example of Method Overriding

- The `Rectangle` class has the following mutator method:

```
public void setWidth(int w) {
    if (w <= 0) {
        throw new IllegalArgumentException();
    }
    this.width = w;
}
```

- The `Square` class inherits it. Why should we override it?

- One option: have the `Square` version change width *and* height.

Which of these works?

A.
```
// Square version, which overrides
// the version inherited from Rectangle
public void setWidth(int w) {
    this.width = w;
    this.height = w;
}
```

B.
```
// Square version, which overrides
// the version inherited from Rectangle
public void setWidth(int w) {
    this.setWidth(w);
    this.setHeight(w);
}
```

C. either version would work

D. neither version would work

Accessing Methods from the Superclass

* The solution: use `super` to access the inherited version
  of the method – the one we are overriding:

```
// Square version
public void setWidth(int w) {
    super.setWidth(w);   // call the Rectangle version
    super.setHeight(w);
}
```

* Only use `super` if you want to call a method from
  the superclass *that has been overridden*.

* If the method has *not* been overridden, use `this` as usual.

Accessing Methods from the Superclass

* We need to override *all* of the inherited mutators:

```
// Square versions
public void setWidth(int w) {
    super.setWidth(w);
    super.setHeight(w);
}

public void setHeight(int h) {
    super.setWidth(h);
    super.setHeight(h);
}

public void grow(int dw, int dh) {
    if (dw != dh) {
        throw new IllegalArgumentException();
    }
    super.setWidth(this.getWidth() + dw);
    super.setHeight(this.getHeight() + dh);
}
```

`getWidth()` and `getHeight()`
are not overridden, so we use `this`.

Inheritance Hierarchy

- Inheritance leads classes to be organized in a *hierarchy*:


```
public class Shape {
    // fields and methods
    // common to all shapes
    ...
}
```

```
public class Rectangle
extends Shape {

    ...

}
```
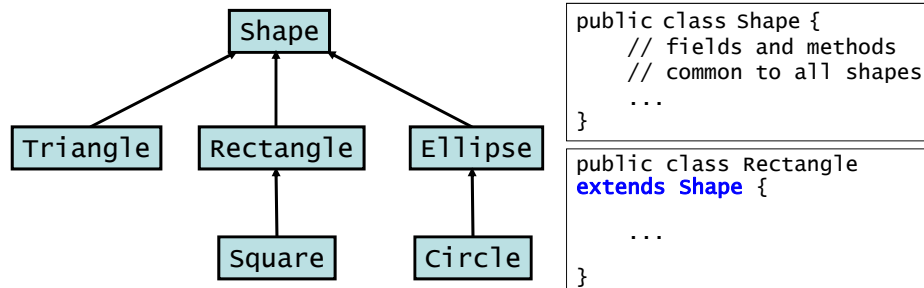
- A class in Java inherits *directly* from at most one class.

- However, a class can inherit *indirectly* from a class higher up in the hierarchy.
  - example: `Square` inherits indirectly from `Shape`

The `Object` Class

- If a class doesn't explicitly extend another class,
  it implicitly extends a special class called `Object`.

- Thus, the `Object` class is at the top of the class hierarchy.
  - *all* classes are subclasses of this class
  - the default `toString()` and `equals()` methods are defined in this class
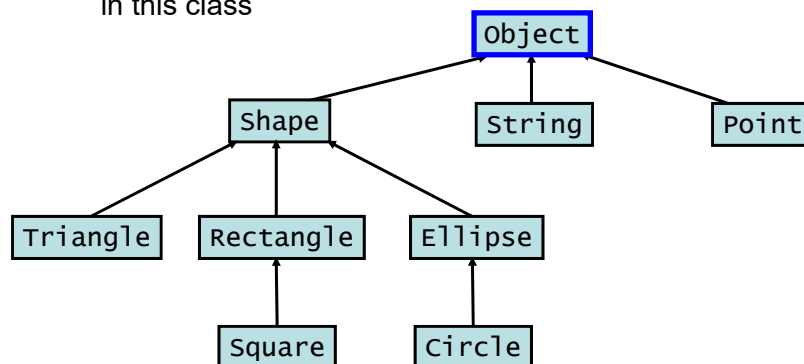
Polymorphism

- We've been using reference variables like this:

  `Rectangle r1 = new Rectangle(20, 30);`

  - variable `r1` is declared to be of type `Rectangle`
  - it holds a reference to a `Rectangle` object

- In addition, a reference variable of type T can hold a reference to an object from a *subclass* of T:

  `Rectangle r1 = new Square(50, "cm");`

  - this works because `Square` is a subclass of `Rectangle`
  - a square *is* a rectangle!

- The name for this feature of Java is *polymorphism*.
  - from the Greek for "many forms"
  - the same code can be used with objects of different types!

Polymorphism and Collections of Objects

- Polymorphism is useful when we have a collection of objects of different but related types.

- Example:
  - let's say that you need a collection of different shapes:
  - we can store all of them in an array of type `Shape`:

    ```
    Shape[] myShapes = new Shape[5];
    myShapes[0] = new Rectangle(20, 30);
    myShapes[1] = new Square(50, "cm");
    myShapes[2] = new Triangle(10, 8);
    myShapes[3] = new Circle(10);
    myShapes[4] = new Rectangle(50, 100);
    ```

Processing a Collection of Objects

- We can print out a description of each shape as follows:

```
Shape[] myShapes = new Shape[5];
myShapes[0] = new Rectangle(20, 30);
myShapes[1] = new Square(50, "cm");
myShapes[2] = new Triangle(10, 8);
myShapes[3] = new Circle(10);
myShapes[4] = new Rectangle(50, 100);

for (int i = 0; i < myShapes.length; i++) {
    System.out.println(myShapes[i]);
}
```

- For each element of the array, the appropriate `toString()` method is called!
  - `myShapes[0]`: the `Rectangle` version of `toString()` is called
  - `myShapes[1]`: the `Square` version of `toString()` is called
  - etc.

Practice with Polymorphism
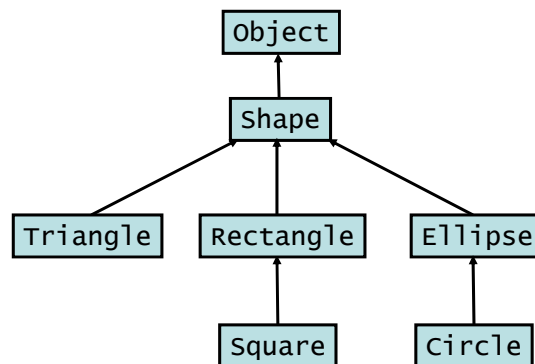
- Which of these assignments would be allowed?

```
Shape s1 = new Triangle(10, 8);
Square sq = new Rectangle(20, 30);
Rectangle r1 = new Circle(15);
Object o = new Circle(15);
```

Which of these would be allowed?

```
        Object
          ↑
        Shape
       ↗  ↑  ↖
Triangle Rectangle Ellipse
           ↑         ↑
         Square    Circle
```
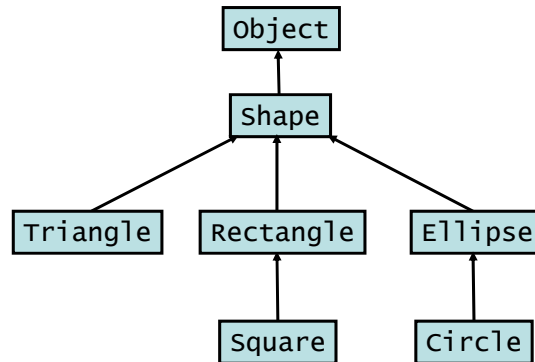
A. `Circle c = new Shape(5);`

B. `Shape s2 = new Square(8, "inch");`

C. both would be allowed

D. neither would be allowed

Another Hierarchy

- G has two non-static methods.

- E extends G
  - it inherits G's fields and methods
  - it *overrides* method2 with its own version
  - it adds a new method called method3

- F also extends G
  - it inherits G's fields and methods
  - it *overrides* method2

- H extends E
  - it inherits E's fields and methods
  - it *overrides* method1

```
                    G
                 method1
                 method2
                  ↗    ↖
          E              F
       method2        method2
       method3
          ↑
          H
       method1
```
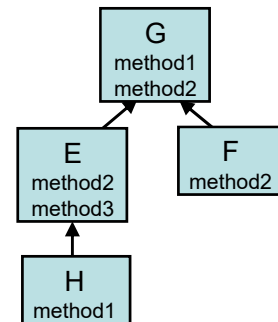
Which Version of a Method Will Run?

- Example:
  ```
  G g = new H();
  g.method2();    // which version runs?
  ```

- To determine which version of a method will run:
  - start at actual type of the object itself
  - go up the hierarchy as needed until you find the method
  - the first version you encounter is the one that will run

```
         ┌──────────┐
         │    G     │
         │ method1  │
         │ method2  │
         └──────────┘
          ↗        ↖
┌──────────┐    ┌──────────┐
│    E     │    │    F     │
│ method2  │    │ method2  │
│ method3  │    └──────────┘
└──────────┘
     ↑
┌──────────┐
│    H     │
│ method1  │
└──────────┘
```

- In this case:
  - start at H, since we have an H object
  - H doesn't have its own method2
  - go up to E
  - E does have a method2, so it's the version that runs!
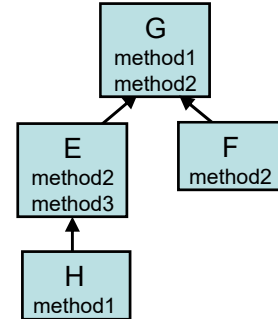
More Practice

- Consider the following object declarations:
  ```
  E e1 = new E();
  G g1 = new H();
  F f1 = new F();
  ```

- For each of the following calls using these objects:
  - will the call compile?
  - if so, which version of the method will run?

```
e1.method1()

e1.method2()

e1.method3()

g1.method1()

f1.method1()

f1.method2()

f1.method3()
```

```
         ┌──────────┐
         │    G     │
         │ method1  │
         │ method2  │
         └──────────┘
          ↗        ↖
┌──────────┐    ┌──────────┐
│    E     │    │    F     │
│ method2  │    │ method2  │
│ method3  │    └──────────┘
└──────────┘
     ↑
┌──────────┐
│    H     │
│ method1  │
└──────────┘
```
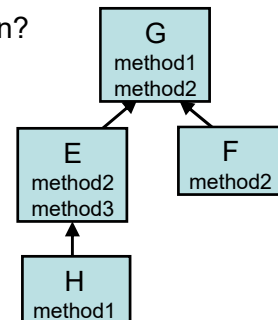
A Bag Data Structure

Computer Science 112
Boston University

What is a Bag?

- A bag is just a container for a group of data items.
  - analogy: a bag of candy

- The positions of the data items don't matter (unlike a sequence).
  - {3, 2, 10, 6} is equivalent to {2, 3, 6, 10}

- The items do *not* need to be unique (unlike a set).
  - {**7**, 2, 10, **7**, 5} isn't a set, but it is a bag

Implementing a Bag Data Structure

* We can create a blueprint class for bags.

* Each object of this class will represent an entire bag of items.
    * example: one object might represent {3, 2, 10, 6}

* This will be an example of a *collection class.*
    * each object will represent a collection of items

Implementing a Bag Data Structure (cont.)

* The operations we want each bag object to support:
    * `add(item)`: add `item` to the bag
    * `remove(item)`: remove one occurrence of `item` (if any) from the bag
    * `contains(item)`: check if `item` is in the bag
    * `numItems()`: get the number of items in the bag
    * `grab()`: get an item at random, without removing it
        * reflects the fact that the items don't have a position (and thus we *can't* say "get the 5th item in the bag")
    * `toArray()`: get an array containing the current contents of the bag

* We want the bag to be able to store objects of any type.

One Possible Bag Implementation

* One way to store the items in the bag is to use an array:
    ```
    public class ArrayBag {
        private _____[] items;


        ...
    }
    ```

* What type should the array be?

* This allows us to store *any* type of object in the `items` array, thanks to the power of polymorphism:
    ```
    ArrayBag bag = new ArrayBag();
    bag.add("hello");
    bag.add(new Rectangle(20, 30));
    ```

* How could we keep track of how many items are in a bag?

Two Methods with the Same Name

* In Java, two methods in a given class can have the same name.

* To do so, the methods must have:
    * a different number of parameters
        and/or
    * parameters of different types

* We saw this earlier with the `substring` methods in `String`:
    ```
    String substring(int beginIndex, int endIndex)
    String substring(int beginIndex)
    ```

* This is known as *method overloading*.

* When a method call is made, the compiler uses the values being passed in to figure out which version to call.

Two Constructors for the `ArrayBag` Class

```java
public class ArrayBag {
    private Object[] items;
    private int numItems;
    public static final int DEFAULT_MAX_SIZE = 50;

    public ArrayBag() {
        this.items = new Object[DEFAULT_MAX_SIZE];
        this.numItems = 0;
    }
    public ArrayBag(int maxSize) {
        ...
}
```

* We can have two different constructors!
    * the parameters must differ in some way

* The first one is useful for small bags.
    * creates an array with room for 50 items.

* The second one allows the client to specify the max # of items.

Two Constructors for the `ArrayBag` Class

```java
public class ArrayBag {
    private Object[] items;
    private int numItems;
    public static final int DEFAULT_MAX_SIZE = 50;

    public ArrayBag() {
        this.items = new Object[DEFAULT_MAX_SIZE];
        this.numItems = 0;
    }
    public ArrayBag(int maxSize) {
        if (maxSize <= 0) {
            throw new IllegalArgumentException(
                "maxSize must be > 0");
        }
        this.items = new Object[maxSize];
        this.numItems = 0;
    }
    ...
}
```

* If the user inputs an invalid `maxSize`, we throw an exception.

What if we want to keep track of how many `ArrayBag` objects a program has created?

Increment the static variable every time that we create an `ArrayBag`.

Static variables are like globals. They can be accessed by *any* method of the class.

```java
public class ArrayBag {
    private Object[] items;
    private int numItems;
    public static final int DEFAULT_MAX_SIZE = 50;
    private static int numBagsCreated = 0;

    public ArrayBag() {
        this.items = new Object[DEFAULT_MAX_SIZE];
        this.numItems = 0;
        numBagsCreated++;
    }
    public ArrayBag(int maxSize) {
        if (maxSize <= 0) {
            throw new IllegalArgumentException(...);
        }
        this.items = new Object[maxSize];
        this.numItems = 0;
        numBagsCreated++;
    } ...
}
```

- Add a static variable – one that belongs to the class as a whole.
  - *not* final – because we want to be able to change it!

Adding Items

- We fill the array from left to right. Here's an empty bag:

items | —→ | null | null | null | null
numItems | 0
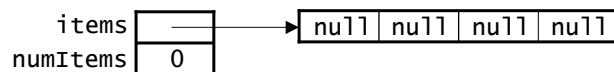
- After adding the first item:

items | —→ | | null | null | null
numItems | 1

"hello, world"
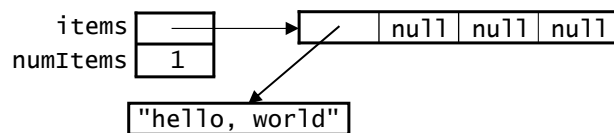
- After adding the second item:

items | —→ | | | null | null
numItems | 2

"hello, world" "howdy"
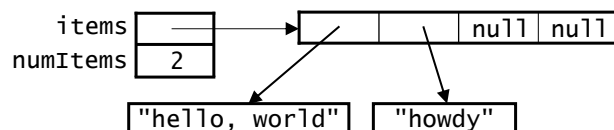
Adding Items (cont.)

- After adding the third item:

```
items   [  —— ] ——→ [ /  |    |    | null ]
numItems [ 3 ]              ↓    ↓     ↓
              [ "hello, world" ] [ "howdy" ] [ "bye" ]
```
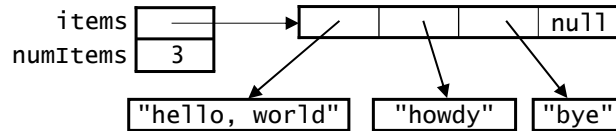
- After adding the fourth item:

```
items   [  —— ] ——→ [ /  |    |    |    ]
numItems [ 4 ]        ↓    ↓     ↓     ↓
        [ "hello, world" ] [ "howdy" ] [ "bye" ] [ "see ya!" ]
```
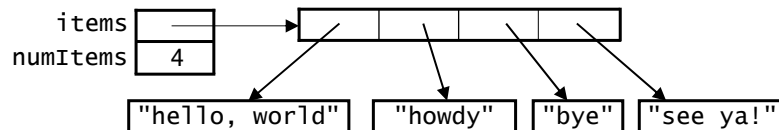
- At this point, the `ArrayBag` is full!
  - it's non-trivial to "grow" an array, so we don't!
  - additional items cannot be added until one is removed

A Method for Adding an Item to a Bag

```java
public class ArrayBag {
    private Object[] items;
    private int numItems;
    ...
    public boolean add(Object item) {
        if (item == null) {
            throw new IllegalArgumentException("no nulls");
        } else if (this.numItems == this.items.length) {
            return false;   // no more room!
        } else {
            this.items[this.numItems] = item;
            this.numItems++;
            return true;    // success!
        }
    }
    ...
}
```

- takes an object of any type!
- returns a `boolean` to indicate whether the operation succeeded

- Initially, `this.numItems` is 0, so the first item goes in position 0.

- We increase `this.numItems` because we now have 1 more item.
  - and so the *next* item added will go in the correct position!

In add(), could we replace `this.items.length` with `maxSize`?

```java
public class ArrayBag {
    private Object[] items;
    private int numItems;
    ...
    public ArrayBag(int maxSize) {
        if (maxSize <= 0) {
            throw new IllegalArgumentException(...);
        }
        this.items = new Object[maxSize];
        this.numItems = 0;
    }

    public boolean add(Object item) {
        if (item == null) {
            throw new IllegalArgumentException("no nulls");
        } else if (this.numItems == this.items.length) {
            return false;    // no more room!
        } else {
            this.items[this.numItems] = item;
            this.numItems++;
            return true;     // success!
        }
    } ...
```

Example: Adding an Item

```java
public static void main(String[] args) {
    String message = "hello, world";
    ArrayBag b = new ArrayBag(4);
    b.add(message);
    …
}
```

```java
public boolean add(Object item) {
    …
    else {
        this.items[this.numItems] = item;
        this.numItems++;
        return true;
    } …
```

stack   *heap*

```
this
item
b
message
args
```

```
items
numItems      0
```

```
null | null | null | null
```

```
"hello, world"
```
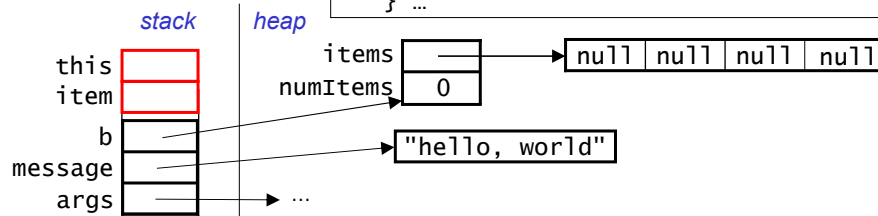
- add's stack frame includes:
  - `item`, which stores…
  - `this`, which stores…

Example: Adding an Item (cont.)

```
public static void main(String[] args) {
    String message = "hello, world";
    ArrayBag b = new ArrayBag(4);
    b.add(message);
    …
}
```

```
public boolean add(Object item) {
    …
    else {
        this.items[this.numItems] = item;
        this.numItems++;
        return true;
    } …
```


- The method modifies the `items` array and `numItems`.
  - note that the array holds a copy of the *reference* to the item, not a copy of the item itself.

Example: Adding an Item (cont.)

```
public static void main(String[] args) {
    String message = "hello, world";
    ArrayBag b = new ArrayBag(4);
    b.add(message);
    …
}
```

```
public boolean add(Object item) {
    …
    else {
        this.items[this.numItems] = item;
        this.numItems++;
        return true;
    } …
```


- The method modifies the `items` array and `numItems`.
  - note that the array holds a copy of the *reference* to the item, not a copy of the item itself.

Example: Adding an Item (cont.)

```
public static void main(String[] args) {
    String message = "hello, world";
    ArrayBag b = new ArrayBag(4);
    b.add(message);
    …
}
```

```
public boolean add(Object item) {
    …
    else {
        this.items[this.numItems] = item;
        this.numItems++;
        return true;
    } …
```

stack   *heap*

```
items                            null  null  null
numItems    1

b
message                "hello, world"
args        …
```

- After the method call returns, add's stack frame is removed from the stack.

Determining if a Bag Contains an Item
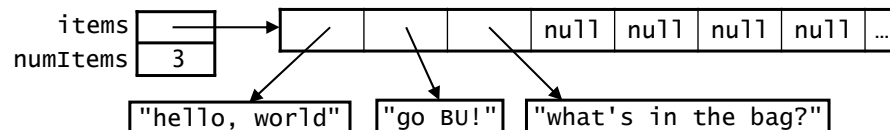
```
items                            null  null  null  null  …
numItems    3

        "hello, world"  "go BU!"  "what's in the bag?"
```

- Let's write the ArrayBag contains() method together.
  - should return true if an object equal to item is found, and false otherwise.

  _____ contains(_____ item) {

  }

Would this work instead?
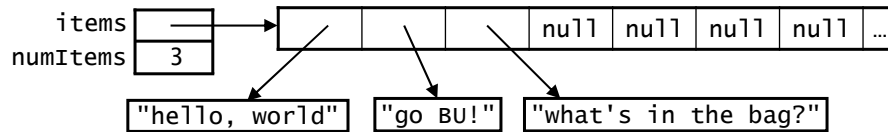
```
  items  ┌──┬──┐ ──────▶ ┌──┬──┬──┬────┬────┬────┬────┬───┐
numItems │  3  │         │ ╱│  │ ╲│null│null│null│null│...│
         └─────┘         └──┴──┴──┴────┴────┴────┴────┴───┘
              ┌──────────────┐ ┌────────┐ ┌────────────────────┐
              │"hello, world"│ │"go BU!"│ │"what's in the bag?"│
              └──────────────┘ └────────┘ └────────────────────┘
```

- Let's write the `ArrayBag` `contains()` method together.
  - should return `true` if an object equal to `item` is found, and `false` otherwise.

```
public boolean contains(Object item) {
    for (int i = 0; i < this.items.length; i++) {
        if (this.items[i].equals(item) {   // not ==
            return true;
        }
    }
    return false;
}
```

What about this?
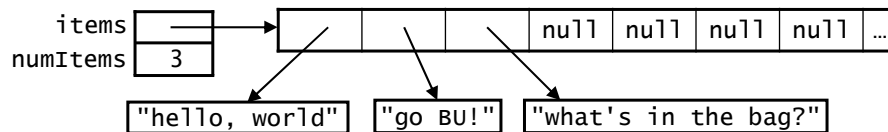
```
  items  ┌──┬──┐ ──────▶ ┌──┬──┬──┬────┬────┬────┬────┬───┐
numItems │  3  │         │ ╱│  │ ╲│null│null│null│null│...│
         └─────┘         └──┴──┴──┴────┴────┴────┴────┴───┘
              ┌──────────────┐ ┌────────┐ ┌────────────────────┐
              │"hello, world"│ │"go BU!"│ │"what's in the bag?"│
              └──────────────┘ └────────┘ └────────────────────┘
```

- Let's write the `ArrayBag` `contains()` method together.
  - should return `true` if an object equal to `item` is found, and `false` otherwise.

```
public boolean contains(Object item) {
    for (Object myItem : this.items) {
        if (myItem.equals(item)) {
            return true;
        }
    }
    return false;
}
```

element-based loop

Another Incorrect `contains()` Method

```java
public boolean contains(Object item) {
    for (int i = 0; i < this.numItems; i++) {
        if (this.items[i].equals(item)) {
            return true;
        } else {
            return false;
        }
    }
    return false;
}
```

* What's the problem with this?

A Method That Takes Another Bag as a Parameter

* Now let's add a method `containsAll(ArrayBag other)`
    * returns `true` if *all* of the items in the `ArrayBag` called `other` are in the called `ArrayBag`, and `false` otherwise.

```java
public boolean containsAll(ArrayBag other) {
    if (other == null || other.numItems == 0) {
        return false;
    }

    for (_____) {



    }

    return _____;
}
```

* Because this method is part of the `ArrayBag` class,
  it is able to access the private fields of the other `ArrayBag`.

A Type Mismatch

* Here are the headers of two `ArrayBag` methods:

    ```
    public boolean add(Object item)
    public Object grab()
    ```

* Polymorphism allows us to pass `String` objects into `add()`:

    ```
    ArrayBag stringBag = new ArrayBag();
    stringBag.add("hello");
    stringBag.add("world");
    ```

* However, this will <u>not</u> work:

    ```
    String str = stringBag.grab();   // compiler error
    ```
    * the return type of `grab()` is `Object`
    * `Object` isn't a subclass of `String`, so polymorphism doesn't help!

* Instead, we need to use a *type cast*:

    ```
    String str = (String)stringBag.grab();
    ```
    * this cast doesn't actually change the value being assigned
    * it just reassures the compiler that the assignment is okay

Recursion

Computer Science 112
Boston University

Printing a Series of Integers

- `printSeries(n1, n2)` should print the series of integers from n1 to n2, where n1 <= n2.
  - example: `printSeries(5, 10)` should print the following:

    `5, 6, 7, 8, 9, 10`

- Here's an *iterative* solution - one that uses a loop:

```
public static void printSeries(int n1, int n2) {
    for (int i = n1; i < n2; i++) {
        System.out.print(i + ", ");
    }
    System.out.println(n2);
}
```

Another Solution

- Here's an alternative solution to the same problem:

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {             // base case
        System.out.println(n2);
    } else {                    // recursive case
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);
    }
}
```

- A method that calls itself is a *recursive* method.

- This approach to problem-solving is known as *recursion*.

Tracing a Recursive Method

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {
        System.out.println(n2);
    } else {
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);
    }
}
```

- What happens when we execute `printSeries(5, 7)`?

```
printSeries(5, 7):
    System.out.print(5 + ", ");
    printSeries(6, 7):
        System.out.print(6 + ", ");
        printSeries(7, 7):
            System.out.println(7);
            return
        return
    return
```

Tracing a Recursive Method: Second Example

```
public static void mystery(int i) {
    if (i <= 0) {      // base case
        return;
    }
    // recursive case
    System.out.println(i);
    mystery(i - 1);
    System.out.println(i);
}
```

* What is the output of `mystery(2)`?

A.	2	**B.**	2	**C.**	2	**D.**	2
	1		1		1		1
			0		0		1
					1		2
					2		

A Recursive Method That Returns a Value

* Simple example: summing the integers from 1 to n

```
public static int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    int rest = sum(n - 1);
    return n + rest;
}
```

Tracing a Recursive Method on the Stack

```
public static int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    int rest = sum(n - 1);
    return n + rest;
}
```

The final result gets built up *on the way back* from the base case!

Example: `sum(3)`

base case

n | 0
rest |
return 0

rest = sum(0) = 0

n | 1
rest |

n | 1
rest |

n | 1
rest | 0
return 1+0

n | 2
rest |

n | 2
rest |

n | 2
rest |

n | 2
rest |

n | 2
rest | 1
return 2+1

n | 3
rest |

n | 3
rest |

n | 3
rest |

n | 3
rest |

n | 3
rest |

n | 3
rest |

n | 3
rest | 3
return 3+3

time ⟶

final result: 6
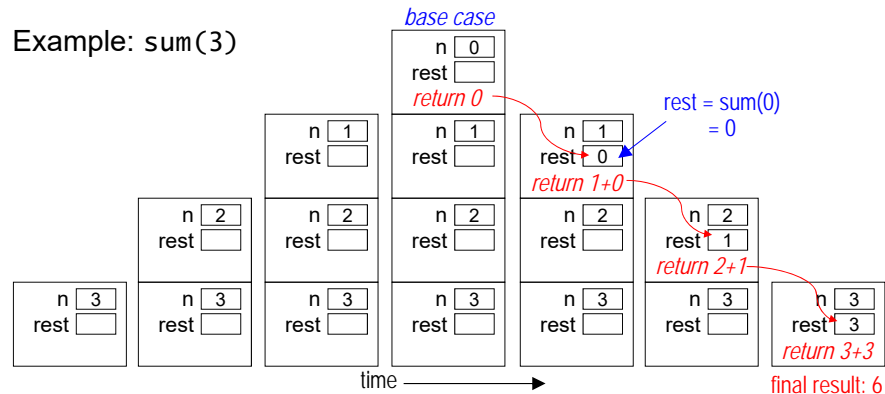
Designing a Recursive Method

1. Start by programming the base case(s).
   - *What instance(s) of this problem can I solve directly (without looking at anything smaller)?*

2. Find the recursive substructure.
   - *How could I use the solution to **any smaller version** of the problem to solve the overall problem?*

3. Solve the smaller problem using a recursive call!
   - store its result in a variable

4. Do your one step.
   - build your solution from the result of the recursive call
   - use *concrete cases* to figure out what you need to do

Processing a String Recursively

- A string is a recursive data structure. It is either:
  - empty ("")
  - a single character, followed by a string

- Thus, we can easily use recursion to process a string.
  - process one or two of the characters ourselves
  - make a recursive call to process the rest of the string

- Example: print a string vertically, one character per line:

```
public static void printVertical(String str) {
    if (str == null || str.equals("")) {
        return;
    }

    System.out.println(str.charAt(0)); // first char
    printVertical(str.substring(1));   // rest of string
}
```

Counting Occurrences of a Character in a String

- `numOccur(c, s)` should return the number of times that the character `c` appears in the string `s`
  - `numOccur('n', "banana")` should return 2
  - `numOccur('a', "banana")` should return 3

- Will the following typical approach work?
  - base case: empty string (or null)
  - delegate `s.substring(1)` to the recursive call
  - we're responsible for handling `s.charAt(0)`

Which combination is correct?

```
public static int numOccur(char c, String s) {
    if (s == null || s.equals("")) {  // base case
        return _____;
    } else {                    // recursive case
        int rest = _____;
        // do our one step!
    }
}
```

	first blank	**second blank**
A.	""	s.substring(1)
B.	""	numOccur(c, s.substring(1))
C.	0	s.substring(1)
D.	0	numOccur(c, s.substring(1))

Determining Our One Step

```
public static int numOccur(char c, String s) {
    if (s == null || s.equals("")) {
        return 0;
    } else {
        int rest = numOccur(c, s.substring(1));
        // do our one step!
```

* In our one step, we take care of s.charAt(0).
  * we build the solution to the larger problem on the solution to the smaller problem (in this case, rest)
  * does what we do depend on the value of s.charAt(0)?

* *Use concrete cases to figure out the logic!*

Consider this concrete case...

```
public static int numOccur(char c, String s) {
    if (s == null || s.equals("")) {
        return 0;
    } else {
        int rest = numOccur(c, s.substring(1));
        // do our one step!
        ...
```

numOccur('r', "recurse")

---

numOccur('r', "recurse")
  c = 'r', s = "recurse"

What value is eventually assigned to `rest`?
(i.e., what does the recursive call return?)

```
public static int numOccur(char c, String s) {
    if (s == null || s.equals("")) {
        return 0;
    } else {
        int rest = numOccur(c, s.substring(1));
        // do our one step!
        ...
```

numOccur('r', "recurse")

A. 2

B. 1

C. 0

D. none of the above

numOccur('r', "recurse")
  c = 'r', s = "recurse"
  int rest = ???

Consider Concrete Cases

```
numOccur('r', "recurse")      # first char is a match
```
- what is its solution?
- what is the next smaller subproblem?
- what is the solution to that subproblem?
- how can we use the solution to the subproblem?
  What is our one step?

```
numOccur('a', "banana")      # first char is not a match
```
- what is its solution?
- what is the next smaller subproblem?
- what is the solution to that subproblem?
- how can we use the solution to the subproblem?
  What is our one step?

Now complete the method!

```
public static int numOccur(char c, String s) {
    if (s == null || s.equals("")) {
        return 0;
    } else {
        int rest = numOccur(c, s.substring(1));
        if (s.charAt(0) == c) {

            return _____;
        } else {

            return _____;
        }
    }
}
```

Tracing a Recursive Method on the Stack

```
public static int numOccur(char c, String s) {
    if (s == null || s.equals("")) {
        return 0;
    } else {
        int rest = numOccur(c, s.substring(1));
        if (s.charAt(0) == c) {
            return 1 + rest;
        } else {
            return rest;
        }
    }
}
    numOccur('a', "aha")
```

The final result gets built up *on the way back* from the base case!

base case

time ⟶
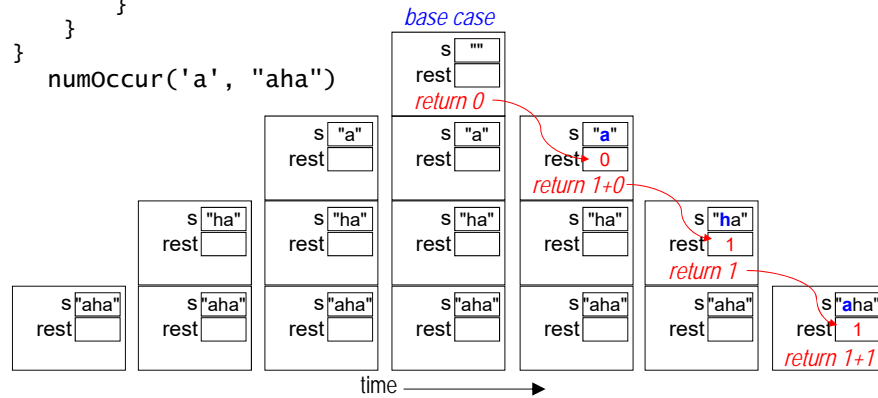
Common Mistake

- This version of the method does *not* work:

```
public static int numOccur(char c, String s) {
    if (s == null || s.equals("")) {
        return 0;
    }

    int count = 0;
    if (s.charAt(0) == c) {
        count++;
    }

    numOccur(c, s.substring(1));
    return count;
}
```

Another Faulty Approach

- Some people make `count` "global" to fix the prior version:

```java
public static int count = 0;

public static int numOccur(char c, String s) {
    if (s == null || s.equals("")) {
        return 0;
    }
    if (s.charAt(0) == c) {
        count++;
    }
    numOccur(c, s.substring(1));
    return count;
}
```

- Not recommended, and not allowed on the problem sets!

- Problems with this approach?

Extra Practice: Removing Vowels From a String

- `removeVowels(s)` - removes the vowels from the string `s`, returning its "vowel-less" version!

    `removeVowels("recursive")` should return `"rcrsv"`

    `removeVowels("vowel")` should return `"vwl"`

- Can we take the usual approach to recursive string processing?
    - base case: empty string
    - delegate `s.substring(1)` to the recursive call
    - we're responsible for handling `s.charAt(0)`

Applying the String-Processing Template

```java
public static String removeVowels(String s) {
    if (s.equals("")) {      // base case

        return _____;
    } else {                       // recursive case

        String rem_rest = _____;
        // do our one step!
    }
}
```

Consider Concrete Cases

`removeVowels("after")`   # first char is a vowel
- what is its solution?
- what is the next smaller subproblem?
- what is the solution to that subproblem?
- how can we use the solution to the subproblem?
  What is our one step?

`removeVowels("recurse")`   # first char is not a vowel
- what is its solution?
- what is the next smaller subproblem?
- what is the solution to that subproblem?
- how can we use the solution to the subproblem?
  What is our one step?

removeVowels()

```
public static String removeVowels(String s) {
    if (s.equals("")) {          // base case
        return "";
    } else {                     // recursive case
        String rem_rest = removeVowels(s.substring(1));
        if ("aeiou".indexOf(s.charAt(0)) != -1) {

            _____

        } else {

            _____

        }
    }
}
```

Recursive Backtracking

Computer Science 112
Boston University

The n-Queens Problem

- **Goal:** to place n queens on an n x n chessboard
  so that no two queens occupy:
  - the same row
  - the same column
  - the same diagonal.

- Sample solution for n = 8:
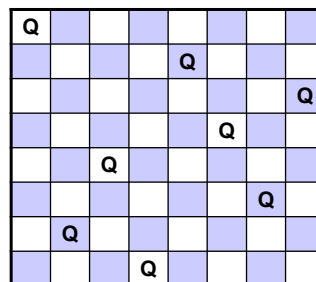
- This problem can be solved using a technique called
  recursive backtracking.

Recursive Strategy for n-Queens

* `findSolution(row)` – to place a queen in the specified `row`:
    * try one column at a time, looking for a "safe" one
    * if we find one: – place the queen there
        – *make a recursive call* to go to the next row
    * if we can't find one: – *backtrack* by returning from the call
        – try to find another safe column
        in the previous row

* Example:
    * row 0:
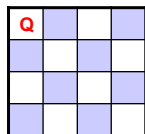
    col 0: safe

    * row 1:
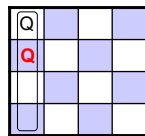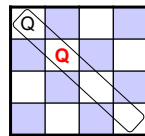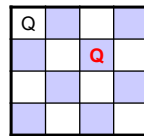
    col 0: same col   *col 1: same diag*   *col 2: safe*

4-Queens Example (cont.)
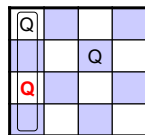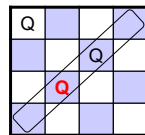
* row 2:
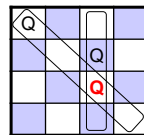
    col 0: same col   *col 1: same diag*   *col 2: same col/diag*   *col 3: same diag*
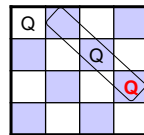
* We've run out of columns in row 2!

* *Backtrack* to row 1 by returning from the recursive call.
    * pick up where we left off
    * we had already tried columns 0-2, so now we try column 3:
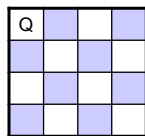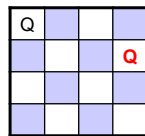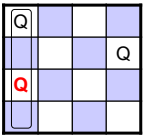
    we left off in col 2   *try col 3: safe*
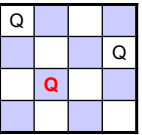
* Continue the recursion as before.

4-Queens Example (cont.)
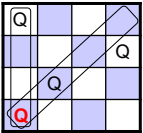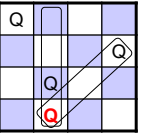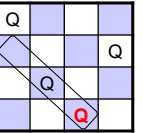
- row 2:

col 0: same col *col 1: safe*

- row 3:
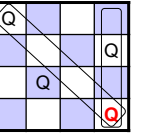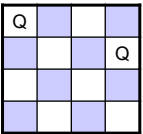
col 0: same col/diag *col 1: same col/diag* *col 2: same diag* *col 3: same col/diag*
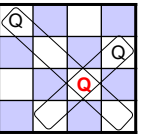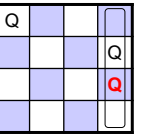
- Backtrack to row 2:
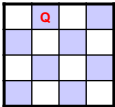
we left off in col 1 *col 2: same diag* *col 3: same col*
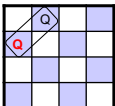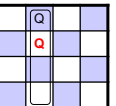
- Backtrack to row 1. No columns left, so backtrack to row 0!
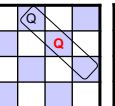
4-Queens Example (cont.)
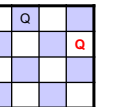
- row 0:
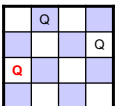
- row 1:
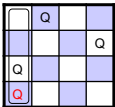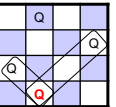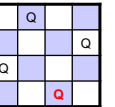
- row 2:

- row 3:

A solution!

A Blueprint Class for an N-Queens Solver

```
public class NQueens {
    private boolean[][] board;  // state of the chessboard
    // other fields go here...

    public NQueens(int n) {
        this.board = new boolean[n][n];
        // initialize other fields here...
    }

    ...
```
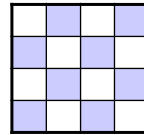
- Here's what the object looks like initially:

NQueens object

| board |
| //other fields |

false	false	false	false
false	false	false	false
false	false	false	false
false	false	false	false

A Blueprint Class for an N-Queens Solver

```
public class NQueens {
    private boolean[][] board;  // state of the chessboard
    // other fields go here...

    public NQueens(int n) {
        this.board = new boolean[n][n];
        // initialize other fields here...
    }

    private void placeQueen(int row, int col) {
        this.board[row][col] = true;
        // modify other fields here...
    }
```

Q			Q
		Q	
	Q		
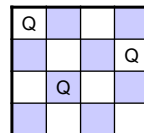
- Here's what it looks like after placing some queens:

NQueens object
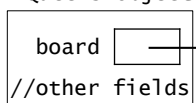
| board |
| //other fields |

true	false	false	false
false	false	false	true
false	true	false	false
false	false	false	false

A Blueprint Class for an N-Queens Solver

```java
public class NQueens {
    private boolean[][] board;  // state of the chessboard
    // other fields go here...

    public NQueens(int n) {
        this.board = new boolean[n][n];
        // initialize other fields here...
    }

    private void placeQueen(int row, int col) {
        this.board[row][col] = true;
        // modify other fields here...
    }

    private void removeQueen(int row, int col){
        this.board[row][col] = false;
        // modify other fields here...
    }

    private boolean isSafe(int row, int col) {
        // returns true if [row][col] is "safe", else false
    }

    private boolean findSolution(int row) {
        // see next slide!
    ...
```

private helper methods that will only be called by code within the class.

Making them private means we don't need to do error-checking!

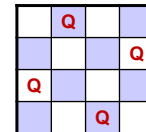## Recursive-Backtracking Method

```java
private boolean findSolution(int row) {
    if (row == this.board.length) {
        this.displayBoard();
        return true;
    }
    for (int col = 0; col < this.board.length; col++) {
        if (this.isSafe(row, col)) {
            this.placeQueen(row, col);
            if (this.findSolution(row + 1)) {
                return true;
            }
            this.removeQueen(row, col);
        }
    }
    return false;
}
```

* takes the index of a row (initially 0)

* uses a loop to consider all possible columns in that row

* makes a recursive call to move onto the next row

* returns `true` if a solution has been found; `false` otherwise

Tracing findSolution()

```
private boolean findSolution(int row) {
    if (row == this.board.length) {
        // code to process a solution goes here...
    }
    for (int col = 0; col < this.board.length; col++) {
        if (this.isSafe(row, col)) {
            this.placeQueen(row, col);
            if (this.findSolution(row + 1)) {
                return true;
            }
            this.removeQueen(row, col);
        }
    }
    return false;
}
```
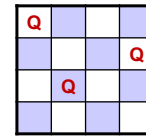
Note: row++ will not work here!

We can pick up where we left off, because row and col are stored in the stack frame!

backtrack!

		row: 2 col:0,1,2,3,4 return false				*backtrack!* row: 3 col:**0,1,2,3,4** return false
row: 1 col: 0,1,2	row: 1 col: 0,1,2	row: 1 col: 0,1,2	row: 1 col: 0,1,2	row: 1 col: 0,1,2,**3**	row: 2 col: 0,1	row: 2 col: 0,1
row: 0 col: 0	row: 0 col: 0	row: 0 col: 0	row: 0 col: 0	row: 0 col: 0	row: 1 col: 0,1,2,3	row: 1 col: 0,1,2,3
					row: 0 col: 0	row: 0 col: 0

time ————————→

Once we place a queen in the last row…

```
private boolean findSolution(int row) {
    if (row == this.board.length) {
        this.displayBoard();
        return true;
    }
    for (int col = 0; col < this.board.length; col++) {
        if (this.isSafe(row, col)) {
            this.placeQueen(row, col);
            if (this.findSolution(row + 1)) {
                return true;
            }
            this.removeQueen(row, col);
        }
    }
    return false;
}
```
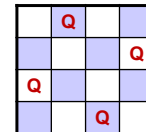
| row: 3 col:0,1,**2** |
| row: 2 col: **0** |
| row: 1 col: 0,1,2,**3** |
| row: 0 col: **1** |

…

time ————————→

…we make one more recursive call…

```
private boolean findSolution(int row) {
    if (row == this.board.length) {
        this.displayBoard();
        return true;
    }
    for (int col = 0; col < this.board.length; col++) {
        if (this.isSafe(row, col)) {
            this.placeQueen(row, col);
            if (this.findSolution(row + 1)) {
                return true;
            }
            this.removeQueen(row, col);
        }
    }
    return false;
}
```

	row: **4**
row: 3 col:0,1,**2**	row: 3 col:0,1,**2**
row: 2 col: **0**	row: 2 col: **0**
row: 1 col: 0,1,2,**3**	row: 1 col: 0,1,2,**3**
row: 0 col: **1**	row: 0 col: **1**

…

time ——————→

…and hit the base case!

```
private boolean findSolution(int row) {
    if (row == this.board.length) {
        this.displayBoard();
        return true;
    }
    for (int col = 0; col < this.board.length; col++) {
        if (this.isSafe(row, col)) {
            this.placeQueen(row, col);
            if (this.findSolution(row + 1)) {
                return true;
            }
            this.removeQueen(row, col);
        }
    }
    return false;
}
```

	row: **4** **return true**
row: 3 col:0,1,**2**	row: 3 col:0,1,**2**
row: 2 col: **0**	row: 2 col: **0**
row: 1 col: 0,1,2,**3**	row: 1 col: 0,1,2,**3**
row: 0 col: **1**	row: 0 col: **1**

…

time ——————→

true is sent back...

```java
private boolean findSolution(int row) {
    if (row == this.board.length) {
        this.displayBoard();
        return true;
    }
    for (int col = 0; col < this.board.length; col++) {
        if (this.isSafe(row, col)) {
            this.placeQueen(row, col);
            if (this.findSolution(row + 1)) { // if (true)
                return true;
            }
            this.removeQueen(         );
        }
    }
    return false;
}
```

	row: 4 **return true**	
row: 3 col:0,1,**2**	row: 3 col:0,1,**2**	row: 3 col:0,1,**2**
row: 2 col: **0**	row: 2 col: **0**	row: 2 col: **0**
row: 1 col: 0,1,2,**3**	row: 1 col: 0,1,2,**3**	row: 1 col: 0,1,2,**3**
row: 0 col: **1**	row: 0 col: **1**	row: 0 col: **1**

...

time ——————→

...and all the earlier calls also return true!

```java
private boolean findSolution(int row) {
    if (row == this.board.length) {
        this.displayBoard();
        return true;
    }
    for (int col = 0; col < this.board.length; col++) {
        if (this.isSafe(row, col)) {
            this.placeQueen(row, col);
            if (this.findSolution(row + 1)) { // if (true)
                return true;
            }
            this.removeQueen(         );
        }
    }
    return false;
}
```

		row: 4 **return true**	
row: 3 col:0,1,**2**	row: 3 col:0,1,**2**	row: 3 col:0,1,**2** **return true**	
row: 2 col: **0**	row: 2 col: **0**	row: 2 col: **0**	row: 2 col: **0** **return true**
row: 1 col: 0,1,2,**3**	row: 1 col: 0,1,2,**3**	row: 1 col: 0,1,2,**3**	row: 1 col: 0,1,2,**3**
row: 0 col: **1**	row: 0 col: **1**	row: 0 col: **1**	row: 0 col: **1**

...

time ——————→

Using a "Wrapper" Method

- The key recursive method is private:
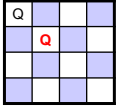
```
private boolean findSolution(int row) {
    ...
}
```

- We use a separate, public *"wrapper" method*
  to start the recursion:

```
public boolean findSolution() {
    return this.findSolution(0);
}
```

  - an example of overloading – two methods with
    the same name, but different parameters
  - this method takes no parameters
  - it makes the initial call to the recursive method
    and returns whatever that call returns
  - it allows us to ensure that the correct initial value
    is passed into the recursive method

Recursive Backtracking in General

- Useful for *constraint satisfaction problems*
  - involve assigning values to variables according to
    a set of constraints
  - n-Queens: variables = Queen's position in each row
                  constraints = no two queens in same row/col/diag
  - many others: factory scheduling, room scheduling, etc.

- Backtracking greatly reduces the number of possible solutions
  that we consider.
  - ex:
    - there are 16 possible solutions that
      begin with queens in these two positions
    - backtracking doesn't consider any of them!

- Recursion makes it easy to handle an arbitrary problem size.
  - stores the state of each variable in a separate stack frame

Template for Recursive Backtracking

```
// n is the number of the variable that the current
// call of the method is responsible for
boolean findSolution(int n, possibly other params) {
    if (found a solution) {
        this.displaySolution();
        return true;
    }

    // loop over possible values for the nth variable
    for (val = first to last) {
        if (this.isValid(val, n)) {
            this.applyValue(val, n);
            if (this.findSolution(n + 1, other params)) {
                return true;
            }
            this.removeValue(val, n);
        }
    }
    return false;        // backtrack!
}
```

Note: n++ will *not* work here!

Template for Finding Multiple Solutions
(up to some target number of solutions)

```
boolean findSolutions(int n, possibly other params) {
    if (found a solution) {
        this.displaySolution();
        this.solutionsFound++;
        return (this.solutionsFound >= this.target);
    }

    // loop over possible values for the nth variable
    for (val = first to last) {
        if (isValid(val, n)) {
            this.applyValue(val, n);
            if (this.findSolutions(n + 1, other params)) {
                return true;
            }
            this.removeValue(val, n);
        }
    }

    return false;
}
```

Data Structures for n-Queens

- Three key operations:
  - `isSafe(row, col)`: check to see if a position is safe
  - `placeQueen(row, col)`
  - `removeQueen(row, col)`

- In theory, our 2-D array of booleans would be sufficient:

  ```
  public class NQueens {
      private boolean[][] board;
  ```

- It's easy to place or remove a queen:

  ```
  private void placeQueen(int row, int col) {
      this.board[row][col] = true;
  }
  private void removeQueen(int row, int col) {
      this.board[row][col] = false;
  }
  ```
  …

- Problem: `isSafe()` takes a lot of steps. What matters more?

Additional Data Structures for n-Queens

- To facilitate `isSafe()`, add three arrays of booleans:

  ```
  private boolean[] colEmpty;
  private boolean[] upDiagEmpty;
  private boolean[] downDiagEmpty;
  ```

- An entry in one of these arrays is:
  - `true` if there are no queens in the column or diagonal
  - `false` otherwise

- Numbering diagonals to get the indices into the arrays:

upDiag = _____

downDiag =
(boardSize − 1) + row − col
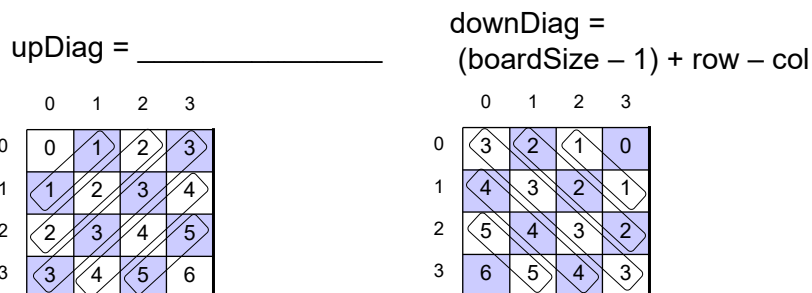
Using the Additional Arrays

* Placing and removing a queen now involve updating four arrays instead of just one. For example:

```
private void placeQueen(int row, int col) {
    this.board[row][col] = true;
    this.colEmpty[col] = false;
    this.upDiagEmpty[row + col] = false;
    this.downDiagEmpty[
        (this.board.length – 1) + row – col] = false;
}
```

* However, checking if a square is safe is now more efficient:

```
private boolean isSafe(int row, int col) {
    return (this.colEmpty[col]
        && this.upDiagEmpty[row + col]
        && this.downDiagEmpty[
                (this.board.length – 1) + row – col]);
}
```

Recursive Backtracking II: Map Coloring

* We want to color a map using **only four colors**.

* Bordering states or countries *cannot* have the same color.
  * example:

not allowed!
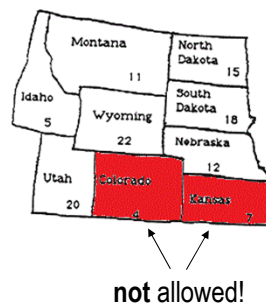
Applying the Template to Map Coloring

```
boolean findSolution(n, perhaps other params) {
    if (found a solution) {
        this.displaySolution();
        return true;
    }
    for (val = first to last) {
        if (this.isValid(val, n)) {
            this.applyValue(val, n);
            if (this.findSolution(n + 1, other params)) {
                return true;
            }
            this.removeValue(val, n);
        }
    }
    return false;
}
```

template element	meaning in map coloring
n	
found a solution	
val	
isValid(val, n)	
applyValue(val, n)	
removeValue(val, n)	

Map Coloring Example

consider the states in alphabetical order. colors = { red, yellow, green, blue }.
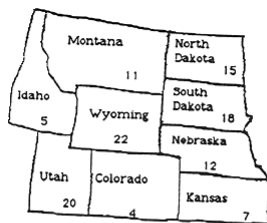
We color Colorado through
Utah without a problem.

    Colorado:
    Idaho:
    Kansas:
    Montana:
    Nebraska:
    North Dakota:
    South Dakota:
    Utah:
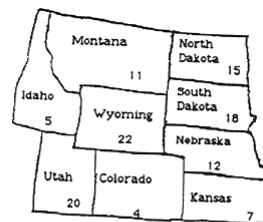
No color works for Wyoming,
so we backtrack…

Map Coloring Example (cont.)
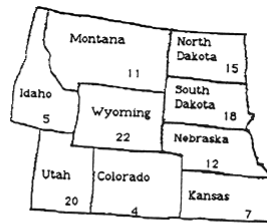
Now we can complete the coloring:

A First Look at Sorting and Algorithm Analysis

Computer Science 112
Boston University

Sorting an Array of Integers
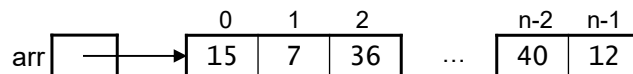
- Ground rules:
  - sort the values in increasing order
  - sort "in place," using only a small amount of additional storage

- Terminology:
  - position: one of the memory locations in the array
  - element: one of the data items stored in the array
  - element i: the element at position i

- Goal: minimize the number of **comparisons C** and the number of **moves M** needed to sort the array.
  - move = copying an element from one position to another
    example: `arr[3] = arr[5];`

Defining a Class for our Sort Methods

```java
public class Sort {
    public static void bubbleSort(int[] arr) {
        ...
    }
    public static void insertionSort(int[] arr) {
        ...
    }
    ...
}
```

* Our `Sort` class is simply a collection of methods like Java's built-in `Math` class.

* Because we never create `Sort` objects, all of the methods in the class must be *static*.
    * outside the class, we invoke them using the class name: e.g., `Sort.bubbleSort(arr)`

A Method for Swapping Elements

* A private helper method used by several of the algorithms:
```java
private static void swap(int[] arr, int a, int b) {
    int temp = arr[a];
    arr[a] = arr[b];
    arr[b] = temp;
}
```

* For example:
```java
int[] arr = {15, 7, 3, 6, 12};
swap(arr, 0, 1);
System.out.println(Arrays.toString(arr));
```

output:
[7, 15, 3, 6, 12]

* Note that every swap requires 3 moves.

Selection Sort

- Basic idea:
  - consider the positions in the array from left to right
  - for each position, find the element that belongs there and put it in place by swapping it with the element that's currently there

- Example:

0	1	2	3	4
15	6	*2*	12	4

0	1	2	3	4
2	6	15	12	*4*

0	1	2	3	4
2	4	15	12	*6*
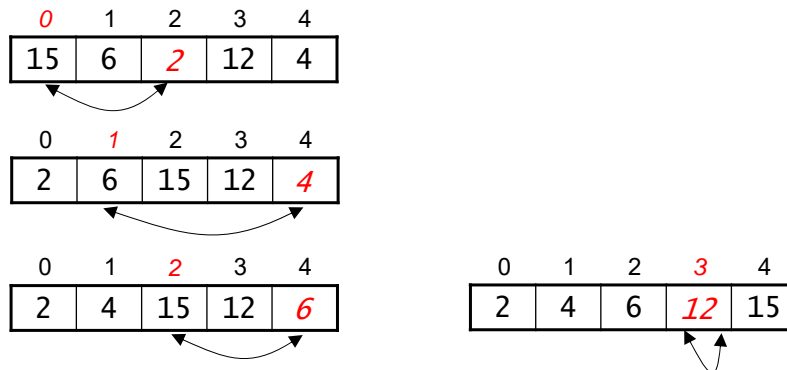
0	1	2	3	4
2	4	6	*12*	15

Why don't we need to consider position 4?

Selecting an Element

- When we consider position `i`, the elements in positions 0 through `i – 1` are already in their final positions.

example for `i = 3`:

0	1	2	3	4	5	6
2	4	7	21	25	10	17

- To select an element for position `i`:
  - consider elements `i, i+1, i+2, …, arr.length – 1`, and keep track of `indexMin`, the index of the smallest element seen thus far

indexMin: 3, 5

0	1	2	3	4	5	6
2	4	7	21	25	*10*	17

  - when we finish this pass, `indexMin` is the index of the element that belongs in position `i`.
  - swap `arr[i]` and `arr[indexMin]`:

0	1	2	3	4	5	6
2	4	7	10	25	21	17
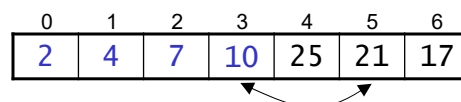
What will things look like after 2 positions are handled?

	0	1	2	3	4	5
	12	5	2	13	18	4

Implementation of Selection Sort

- Use a helper method to find the index of the smallest element:

```
private static int indexSmallest(int[] arr, int start) {
    int indexMin = start;

    for (int i = start + 1; i < arr.length; i++) {
        if (arr[i] < arr[indexMin]) {
            indexMin = i;
        }
    }

    return indexMin;
}
```

- The actual sort method is very simple:

```
public static void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        int j = indexSmallest(arr, i);
        swap(arr, i, j);
    }
}
```

Time Analysis

* Some algorithms are much more efficient than others.

* The *time efficiency* or *time complexity* of an algorithm is some measure of the number of operations that it performs.
    * for sorting, we'll focus on comparisons and moves

* We want to characterize how the number of operations depends on the size, n, of the input to the algorithm.
    * for sorting, n is the length of the array
    * how does the number of operations grow as n grows?

* We'll express the number of operations as functions of n
    * C(n) = number of comparisons for an array of length n
    * M(n) = number of moves for an array of length n

Counting Comparisons by Selection Sort

```
private static int indexSmallest(int[] arr, int start){
    int indexMin = start;

    for (int i = start + 1; i < arr.length; i++) {
        if (arr[i] < arr[indexMin]) {
            indexMin = i;
        }
    }

    return indexMin;
}
public static void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        int j = indexSmallest(arr, i);
        swap(arr, i, j);
    }
}
```

* To sort n elements, selection sort performs n – 1 passes:

    on 1st pass, it performs _____ comparisons to find indexSmallest
    on 2nd pass, it performs _____ comparisons
       …
    on the (n–1)st pass, it performs 1 comparison

* Adding them up: C(n) = 1 + 2 + … + (n - 2) + (n - 1)

Counting Comparisons by Selection Sort (cont.)

- The resulting formula for $c(n)$ is the sum of an arithmetic sequence:

$$c(n) = 1 + 2 + \ldots + (n - 2) + (n - 1) = \sum_{i=1}^{n-1} i$$

- Formula for the sum of this type of arithmetic sequence:

$$\sum_{i=1}^{m} i = \frac{m(m+1)}{2}$$

- Thus, we can simplify our expression for $C(n)$ as follows:

$$c(n) = \sum_{i=1}^{n-1} i$$
$$= \frac{(n-1)((n-1)+1)}{2}$$
$$= \frac{(n-1)n}{2}$$

$$\boxed{c(n) = n^2/2 - n/2}$$

Focusing on the Largest Term

- When n is large, mathematical expressions of n are dominated by their "largest" term — i.e., the term that grows fastest as a function of n.

  - example:

n	$n^2/2$	$n/2$	$n^2/2 - n/2$
10	50	5	45
100	5000	50	4950
10000	50,000,000	5000	49,995,000

- In characterizing the time complexity of an algorithm, we'll focus on the largest term in its operation-count expression.

  - for selection sort, $c(n) = n^2/2 - n/2 \approx n^2/2$

- In addition, we'll typically ignore the coefficient of the largest term (e.g., $n^2/2 \rightarrow n^2$).

Big-*O* Notation

- We specify the largest term using big-*O* notation.
  - e.g., we say that $c(n) = n^2/2 - n/2$ is $O(n^2)$

- Common classes of algorithms:

name	example expressions	big-O notation
constant time	$1, 7, 10$	$O(1)$
logarithmic time	$3\log_{10}n, \log_2 n + 5$	$O(\log n)$
linear time	$5n, 10n - 2\log_2 n$	$O(n)$
n log n time	$4n\log_2 n, n\log_2 n + n$	$O(n \log n)$
quadratic time	$2n^2 + 3n, n^2 - 1$	$O(n^2)$
exponential time	$2^n, 5e^n + 2n^2$	$O(c^n)$

  slower →

- For large inputs, efficiency matters more than CPU speed.
  - e.g., an $O(\log n)$ algorithm on a slow machine will outperform an $O(n)$ algorithm on a fast machine

---

Ordering of Functions

- We can see below that: n^2 grows faster than $n\log_2 n$
  $n\log_2 n$ grows faster than n
  n grows faster than $\log_2 n$
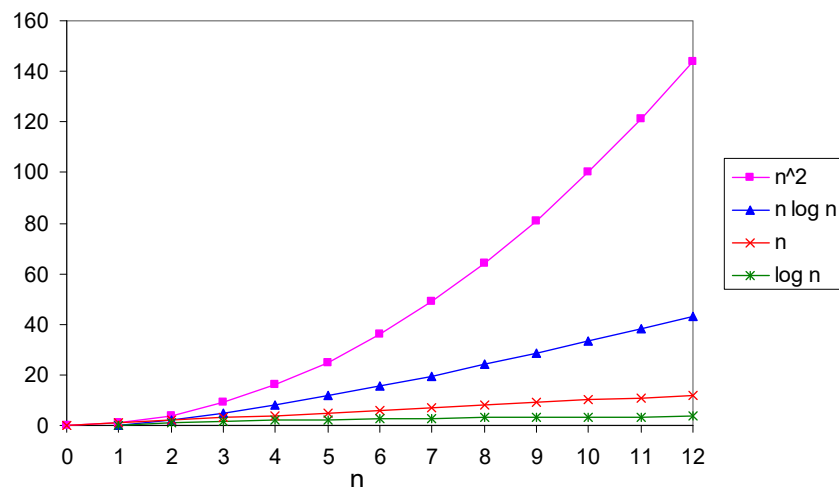
Ordering of Functions (cont.)

- Zooming in, we see that: $n^2 >= n$ for all $n >= 1$
  $n\log_2 n >= n$ for all $n >= 2$
  $n > \log_2 n$ for all $n >= 1$
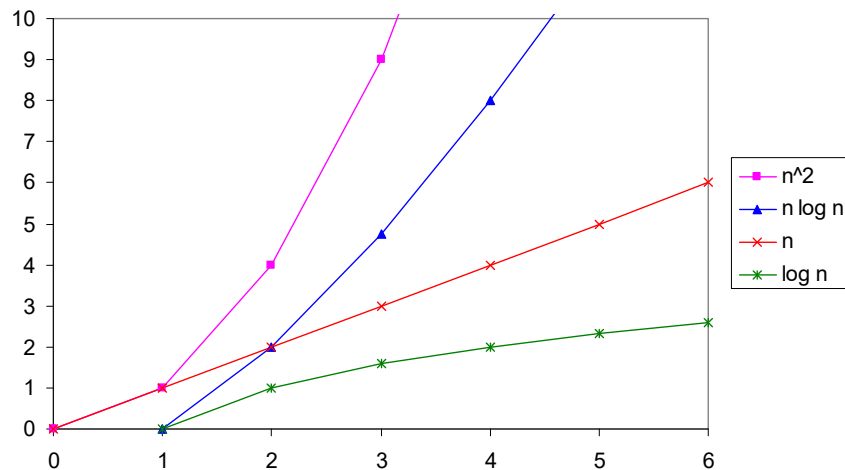
Big-O Time Analysis of Selection Sort

- Comparisons: we showed that $c(n) = n^2/2 - n/2$
  - selection sort performs $O(n^2)$ comparisons

- Moves: after each of the $n-1$ passes, the algorithm does one swap.
  - $n-1$ swaps, 3 moves per swap
  - $M(n) = 3(n-1) = 3n-3$
  - selection sort performs $O(n)$ moves.

- Running time (i.e., total operations): ?

Insertion Sort

- Basic idea:
  - going from left to right, "insert" each element into its proper place with respect to the elements to its left
  - "slide over" other elements to make room

- Example:

0	1	2	3	4
15	*4*	2	12	6

4	15	*2*	12	6

2	4	15	*12*	6

2	4	12	15	*6*

2	4	6	12	15
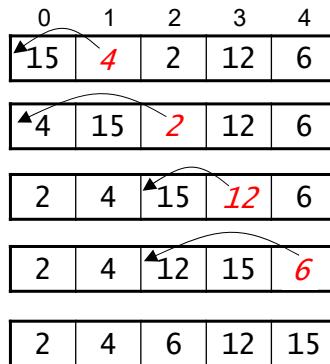
Inserting an Element

- When we consider element `i`, elements 0 through `i – 1` are already sorted with respect to each other.

example for `i = 3`:

0	1	2	3	4
6	14	19	9	...

- To insert element `i`:
  - make a copy of element `i`, storing it in the variable `toInsert`:

toInsert | 9 |

0	1	2	3
6	14	19	9

  - consider elements `i-1, i-2, …`
    - if an element `> toInsert`, slide it over to the right
    - stop at the first element `<= toInsert`

toInsert | 9 |

0	1	2	3
6		14	19

  - copy `toInsert` into the resulting "hole":

0	1	2	3
6	9	14	19

Insertion Sort Example (done together)

description of steps

12	5	2	13	18	4

Implementation of Insertion Sort

```java
public class Sort {
  ...
  public static void insertionSort(int[] arr) {
      for (int i = 1; i < arr.length; i++) {
          if (arr[i] < arr[i-1]) {
              int toInsert = arr[i];

              int j = i;
              do {
                  arr[j] = arr[j-1];
                  j = j - 1;
              } while (j > 0  &&  toInsert < arr[j-1]);

              arr[j] = toInsert;
          }
      }
  }
}
```

Time Analysis of Insertion Sort

- The number of operations depends on the contents of the array.
- *best case:* array is sorted
    - each element is only compared to the element to its left
    - we never execute the do-while loop!
    - C(n) =_____, M(n) = _____, running time = _____

    also true if array is almost sorted

- *worst case:* array is in reverse order
    - each element is compared to *all* of the elements to its left:
        arr[1] is compared to 1 element (arr[0])
        arr[2] is compared to 2 elements (arr[0] and arr[1])
        …
        arr[n-1] is compared to n-1 elements
    - C(n) = 1 + 2 + … + (n − 1) = _____
    - similarly, M(n) = _____, running time = _____

- *average case:* elements are randomly arranged
    - on average, each element is compared to *half* of the elements to its left
    - still get C(n) = M(n) = _____, running time = _____

Bubble Sort

- Perform a sequence of passes from left to right
    - each pass swaps adjacent elements if they are out of order
    - larger elements "bubble up" to the end of the array

- At the end of the kth pass:
    - the k rightmost elements are in their final positions
    - we don't need to consider them in subsequent passes.

- Example:

	0	1	2	3	4
	28	24	37	15	5
after the first pass:	24	28	15	5	*37*
after the second:	24	15	5	*28*	*37*
after the third:	15	5	*24*	*28*	*37*
after the fourth:	5	*15*	*24*	*28*	*37*

Implementation of Bubble Sort

```
public class Sort {
    ...
    public static void bubbleSort(int[] arr) {
        for (int i = arr.length - 1; i > 0; i--) {
            for (int j = 0; j < i; j++) {
                if (arr[j] > arr[j+1]) {
                    swap(arr, j, j+1);
                }
            }
        }
    }
}
```

* Nested loops:
  * the inner loop performs a single pass
  * the outer loop governs:
    * the number of passes (`arr.length - 1`)
    * the ending point of each pass (the current value of `i`)

Time Analysis of Bubble Sort

* Comparisons (n = length of array):
  * they are performed in the inner loop
  * *how many repetitions does each execution of the inner loop perform?*

value of i	number of comparisons
n − 1	n − 1
n − 2	n − 2
...	...
2	2
1	1

$$1 + 2 + \ldots + n - 1 =$$

```
public static void bubbleSort(int[] arr) {
    for (int i = arr.length - 1; i > 0; i--) {
        for (int j = 0; j < i; j++) {
            if (arr[j] > arr[j+1]) {
                swap(arr, j, j+1);
            }
        }
    }
}
```

Time Analysis of Bubble Sort

* Comparisons: the kth pass performs $n - k$ comparisons,

  so we get $\;\; C(n) = \sum_{i=1}^{n-1} i = n^2/2 - n/2 = O(n^2)$

* Moves: depends on the contents of the array
    * in the worst case:

        * $M(n) =$
    * in the best case:

* Running time:
    * $C(n)$ is always $O(n^2)$, $M(n)$ is never worse than $O(n^2)$
    * therefore, the largest term of $C(n) + M(n)$ is $O(n^2)$

* Bubble sort is a quadratic-time or $O(n^2)$ algorithm.
    * can't do much worse than bubble!

Practicing Time Analysis

* Consider the following static method:

```java
public static int mystery(int n) {
    int x = 0;
    for (int i = 0; i < n; i++) {
        x += i;          // statement 1
        for (int j = 0; j < i; j++) {
            x += j;
        }
    }
    return x;
}
```

* What is the big-O expression for the number of times that statement 1 is executed as a function of the input **n**?

What about now?

- Consider the following static method:

```
public static int mystery(int n) {
    int x = 0;
    for (int i = 0; i < 3*n + 4; i++) {
        x += i;         // statement 1
        for (int j = 0; j < i; j++) {
            x += j;
        }
    }
    return x;
}
```

- What is the big-O expression for the number of times that statement 1 is executed as a function of the input **n**?

Practicing Time Analysis

- Consider the following static method:

```
public static int mystery(int n) {
    int x = 0;
    for (int i = 0; i < n; i++) {
        x += i;          // statement 1
        for (int j = 0; j < i; j++) {
            x += j;      // statement 2
        }
    }
    return x;
}
```

- What is the big-O expression for the number of times that statement 2 is executed as a function of the input **n**?

  | value of i | number of times statement 2 is executed |

Extra Practice: Which algorithm is this?

original array

| 12 | 5 | 2 | 13 | 18 | 4 |

➡️

array after some number of passes

| 2 | 5 | 12 | 13 | 18 | 4 |

A. selection sort

B. insertion sort

C. bubble sort

D. it could be more than one of them

Sorting II: Quicksort and Mergesort

Computer Science 112
Boston University

Quicksort

- Like bubble sort, quicksort uses an approach based on swapping out-of-order elements, but it's more efficient.

- A recursive, divide-and-conquer algorithm:
  - *divide:* rearrange the elements so that we end up with two subarrays that meet the following criterion:

    each element in left array <= each element in right array

    example:

    | 12 | 8 | 14 | 4 | 6 | 13 |

    ⟹

    | 6 | 8 | 4 | 14 | 12 | 13 |
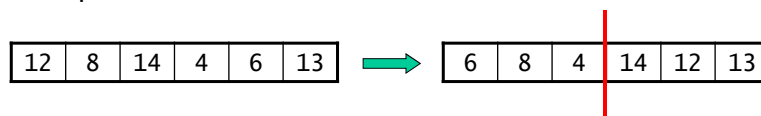
  - *conquer:* apply quicksort recursively to the subarrays, stopping when a subarray has a single element

  - *combine:* nothing needs to be done, because of the way we formed the subarrays

Partitioning an Array Using a Pivot

- The process that quicksort uses to rearrange the elements is known as *partitioning* the array.

- It uses one of the values in the array as a *pivot,* rearranging the elements to produce two subarrays:
  - left subarray: all values <= pivot   *equivalent to the criterion*
  - right subarray: all values >= pivot   *on the previous page.*

7	15	4	9	6	18	9	12

partition using a pivot of 9

7	9	4	6	9	18	15	12

all values <= 9   all values >= 9
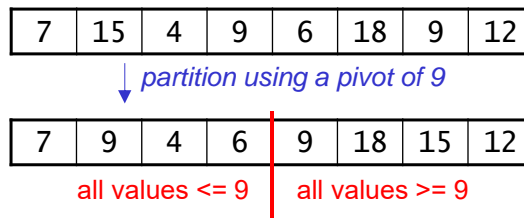
- The subarrays will *not* always have the same length.

- This approach to partitioning is one of several variants.

Possible Pivot Values

- First element or last element
  - risky, can lead to terrible worst-case behavior
  - especially poor if the array is almost sorted

4	8	14	12	6	18

⟹

4	8	14	12	6	18

pivot = 18
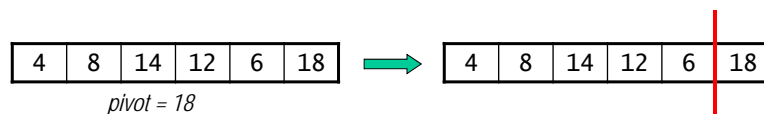
- Middle element (what we will use)

- Randomly chosen element

- Median of three elements
  - left, center, and right elements
  - three randomly selected elements
  - taking the median of three decreases the probability of getting a poor pivot

Partitioning an Array: An Example

```
                first                              last
  arr  [ --]------>  7 | 15 |  4 |  9 |  6 | 18 |  9 | 12
                                pivot = 9
```
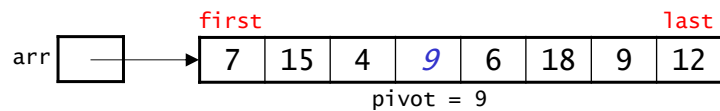
- Maintain indices `i` and `j`, starting them "outside" the array:

```
   i = first – 1        i                                        j
   j = last + 1            7 | 15 |  4 |  9 |  6 | 18 |  9 | 12
```
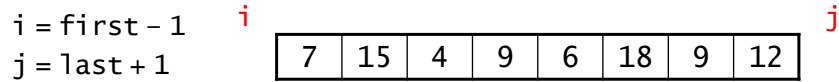
- *Find* "out of place" elements:
  - increment `i` until `arr[i] >= pivot`
  - decrement `j` until `arr[j] <= pivot`
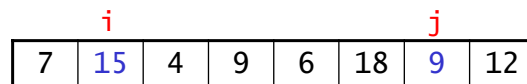
```
                           i                    j
                         7 | 15 |  4 |  9 |  6 | 18 |  9 | 12
```

- *Swap* `arr[i]` and `arr[j]`:

```
                           i                    j
                         7 |  9 |  4 |  9 |  6 | 18 | 15 | 12
```

Partitioning Example (cont.)

```
                             i                    j
from prev. page:           7 |  9 |  4 |  9 |  6 | 18 | 15 | 12
```
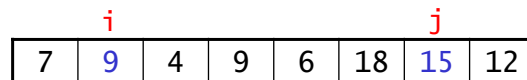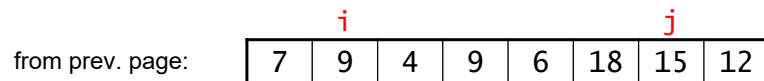
- Find:
```
                                 i    j
                           7 |  9 |  4 |  9 |  6 | 18 | 15 | 12
```

- Swap:
```
                                 i    j
                           7 |  9 |  4 |  6 |  9 | 18 | 15 | 12
```

- Find:
```
                                 j    i
                           7 |  9 |  4 |  6 |  9 | 18 | 15 | 12
```

and now the indices have crossed, so we return `j`.

- Subarrays: left = from `first` to `j`, right = from `j+1` to `last`

```
        first                  j   i                last
                           7 |  9 |  4 |  6 |  9 | 18 | 15 | 12
```
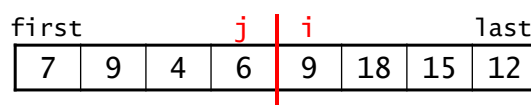
Partitioning Example 2

- Start
  (pivot = 13):

	i							j
	24	5	2	*13*	18	4	20	19

- Find:

	i					j		
	24	5	2	13	18	4	20	19

- Swap:

	i					j		
	4	5	2	13	18	24	20	19

- Find:

			i	j				
	4	5	2	13	18	24	20	19

and now the indices are equal, so we return j.

- Subarrays:

			i	j				
	4	5	2	13	18	24	20	19

Partitioning Example 3 (done together)

- Start
  (pivot = 5):

	i							j
	4	14	7	*5*	2	19	26	6

- Find:

	4	14	7	5	2	19	26	6

Partitioning Example 4

- Start
  (pivot = 15):

  i   j

8	10	7	*15*	20	9	6	18

- Find:

8	10	7	15	20	9	6	18

---

partition() Helper Method

```
private static int partition(int[] arr, int first, int last)
{
    int pivot = arr[(first + last)/2];
    int i = first - 1;  // index going left to right
    int j = last + 1;   // index going right to left
    while (true) {
        do {
            i++;
        } while (arr[i] < pivot);
        do {
            j--;
        } while (arr[j] > pivot);
        if (i < j) {
            swap(arr, i, j);
        } else {
            return j;   // arr[j] = end of left array
        }
    }
}
```

first   last

…	7	15	4	9	6	18	9	12	…

Implementation of Quicksort

```java
public static void quickSort(int[] arr) { // "wrapper" method
    qSort(arr, 0, arr.length - 1);
}

private static void qSort(int[] arr, int first, int last) {
    int split = partition(arr, first, last);

    if (first < split) {  // if left subarray has 2+ values
        qSort(arr, first, split);  // sort it recursively!
    }
    if (last > split + 1) {        // if right has 2+ values
        qSort(arr, split + 1, last);  // sort it!
    }
}   // note: base case is when neither call is made,
    // because both subarrays have only one element!
```

```
                          split
        first              (j)                last
     ┌─────┬───┬───┬───┬───┬───┬────┬────┬────┬─────┐
     │  …  │ 7 │ 9 │ 4 │ 6 │ 9 │ 18 │ 15 │ 12 │  …  │
     └─────┴───┴───┴───┴───┴───┴────┴────┴────┴─────┘
```
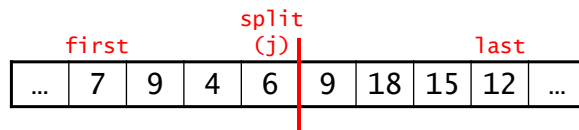
A Quick Review of Logarithms

- $\log_b n$ = the exponent to which b must be raised to get n

  - $\log_b n = p$ if $b^p = n$

  - examples: $\log_2 8 = 3$ because $2^3 = 8$
    $\log_{10} 10000 = 4$ because $10^4 = 10000$

- Another way of looking at logs:
  - let's say that you repeatedly divide n by b (using integer division)
  - $\log_b n$ is an upper bound on the number of divisions needed to reach 1
  - example: $\log_2 18$ is approx. 4.17
    $18/2 = 9$ $9/2 = 4$ $4/2 = 2$ $2/2 = 1$

A Quick Review of Logs (cont.)

- $O(\log n)$ algorithm – one in which the number of operations is proportional to $\log_b n$ for any base b

- $\log_b n$ grows much more slowly than n

n	$\log_2 n$
2	1
1024 (1K)	10
1024*1024 (1M)	20
1024*1024*1024 (1G)	30

- Thus, for large values of n:
  - a $O(\log n)$ algorithm is much faster than a $O(n)$ algorithm
    - $\log n$ << n
  - a $O(n \log n)$ algorithm is much faster than a $O(n^2)$ algorithm
    - $n * \log n$ << $n * n$
      $n \log n$ << n^2

Time Analysis of Quicksort

- Partitioning an array of length n requires approx. n comparisons.
  - most elements are compared with the pivot once; a few twice
- *best case:* partitioning always divides the array in half
  - repeated recursive calls give:

	comparisons
n	n
n/2 n/2	2*(n/2) = n
n/4 n/4 n/4 n/4	4*(n/4) = n
...	...
1 1 1 1 1 1 ... 1 1 1 1	0
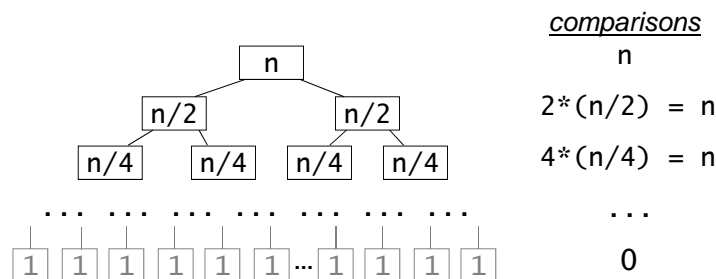
- at each "row" except the bottom, we perform n comparisons
- there are _____ rows that include comparisons
- C(n) = ?
- Similarly, M(n) and running time are both _____

Time Analysis of Quicksort (cont.)

- *worst case:* pivot is always the smallest or largest element
  - one subarray has 1 element, the other has n – 1
  - repeated recursive calls give:

comparisons

n

n–1

n–2

n–3

. . .

2
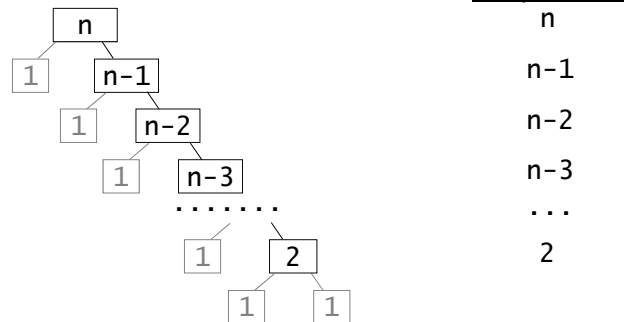
- $C(n) = \sum_{i=2}^{n} i = O(n^2)$. $M(n)$ and run time are also $O(n^2)$.

- *average case* is harder to analyze
  - $C(n) > n\log_2 n$, but it's still $O(n\log n)$

Mergesort

- The algorithms we've seen so far have sorted the array in place.
  - use only a small amount of additional memory

- Mergesort requires an additional temporary array of the same size as the original one.
  - it needs $O(n)$ additional space, where n is the array size

- It is based on the process of *merging* two sorted arrays.
  - example:
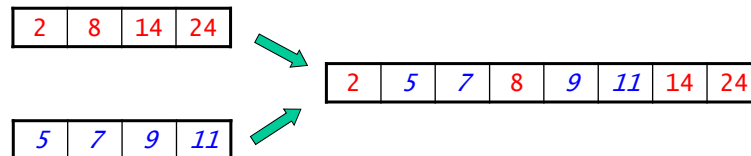
Merging Sorted Arrays

- To merge sorted arrays A and B into an array C, we maintain three indices, which start out on the first elements of the arrays:

```
      i
A [ 2 | 8 | 14 | 24 ]          k
                          C [   |   |   |   |   |   |   |   ]
      j
B [ 5 | 7 | 9 | 11 ]
```
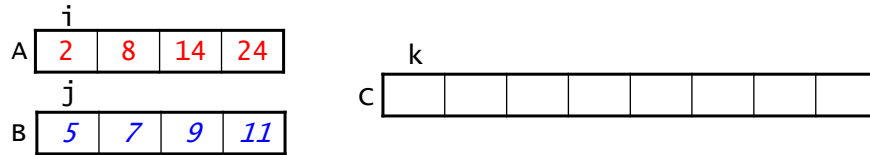
- We repeatedly do the following:
  - compare A[i] and B[j]
  - copy the smaller of the two to C[k]
  - increment the index of the array whose element was copied
  - increment k

```
          i
A [ 2 | 8 | 14 | 24 ]              k
                          C [ 2 |   |   |   |   |   |   |   ]
      j
B [ 5 | 7 | 9 | 11 ]
```
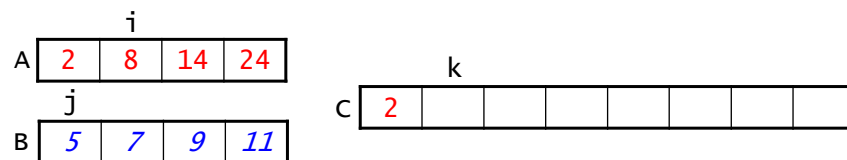
Merging Sorted Arrays (cont.)

- Starting point:

```
      i
A [ 2 | 8 | 14 | 24 ]          k
                          C [   |   |   |   |   |   |   |   ]
      j
B [ 5 | 7 | 9 | 11 ]
```

- After the first copy:

```
          i
A [ 2 | 8 | 14 | 24 ]              k
                          C [ 2 |   |   |   |   |   |   |   ]
      j
B [ 5 | 7 | 9 | 11 ]
```
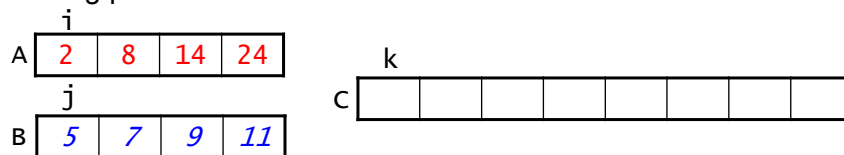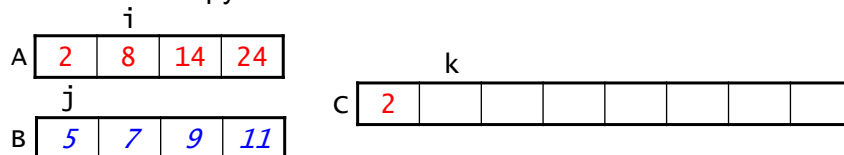
- After the second copy:

```
          i
A [ 2 | 8 | 14 | 24 ]                  k
                          C [ 2 | 5 |   |   |   |   |   |   ]
          j
B [ 5 | 7 | 9 | 11 ]
```
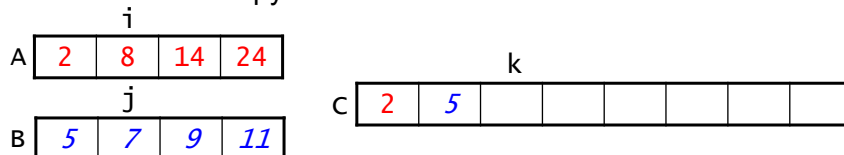
Merging Sorted Arrays (cont.)

- After the third copy:

```
         i
A   2    8   14   24                        k
                              C   2    5    7
         j
B   5    7    9   11
```

- After the fourth copy:

```
              i
A   2    8   14   24                             k
                              C   2    5    7    8
         j
B   5    7    9   11
```

- After the fifth copy:

```
              i
A   2    8   14   24                                  k
                              C   2    5    7    8    9
              j
B   5    7    9   11
```

Merging Sorted Arrays (cont.)

- After the sixth copy:

```
              i
A   2    8   14   24                                       k
                         j    C   2    5    7    8    9   11
B   5    7    9   11
```

- There's nothing left in B, so we simply copy the remaining elements from A:

```
                   i
A   2    8   14   24                                            k
                         j    C   2    5    7    8    9   11   14   24
B   5    7    9   11
```

Divide and Conquer

- Like quicksort, mergesort is a divide-and-conquer algorithm.
  - *divide:* split the array in half, forming two subarrays
  - *conquer:* apply mergesort recursively to the subarrays, stopping when a subarray has a single element
  - *combine:* merge the sorted subarrays

	12	8	14	4	6	33	2	27
split	12	8	14	4	6	33	2	27
split	12	8	14	4	6	33	2	27
split	12	8	14	4	6	33	2	27
merge	8	12	4	14	6	33	2	27
merge	4	8	12	14	2	6	27	33
merge	2	4	6	8	12	14	27	33
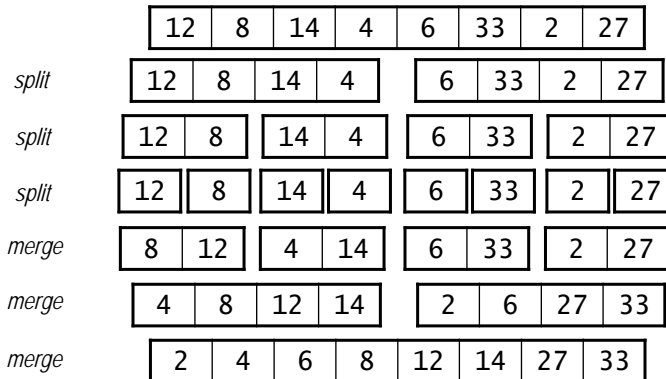
Tracing the Calls to Mergesort

the initial call is made to sort the entire array:

12	8	14	4	6	33	2	27
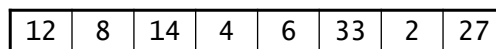
split into two 4-element subarrays, and make a recursive call to sort the left subarray:

12	8	14	4	6	33	2	27

12	8	14	4
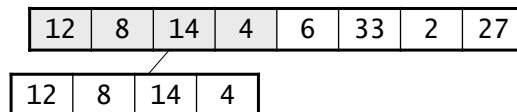
split into two 2-element subarrays, and make a recursive call to sort the left subarray:

12	8	14	4	6	33	2	27

12	8	14	4

12	8
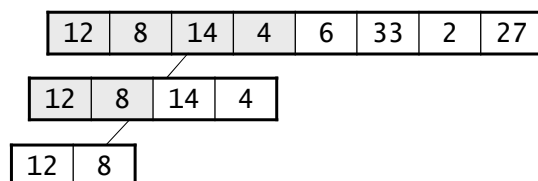
Tracing the Calls to Mergesort

split into two 1-element subarrays, and make a recursive call to sort the left subarray:
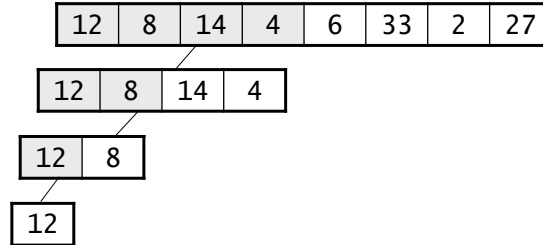
12	8	14	4	6	33	2	27

12	8	14	4

12	8

12

base case, so return to the call for the subarray {12, 8}:

12	8	14	4	6	33	2	27

12	8	14	4

12	8

Tracing the Calls to Mergesort

make a recursive call to sort its right subarray:

12	8	14	4	6	33	2	27

12	8	14	4

12	8

8

base case, so return to the call for the subarray {12, 8}:

12	8	14	4	6	33	2	27

12	8	14	4

12	8
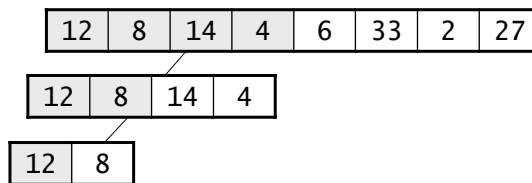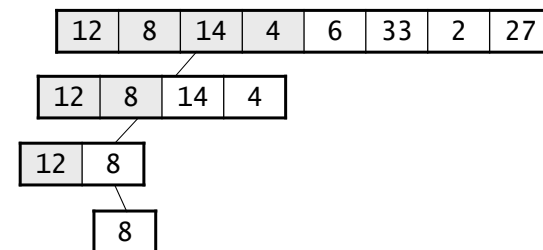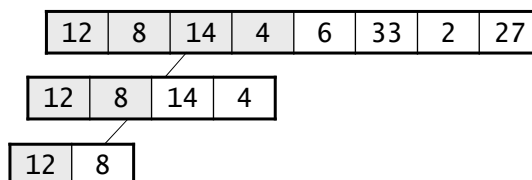
Tracing the Calls to Mergesort

merge the sorted halves of {12, 8}:
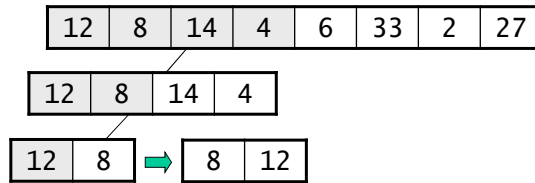
| 12 | 8 | 14 | 4 | 6 | 33 | 2 | 27 |

| 12 | 8 | 14 | 4 |

| 12 | 8 | ➡ | 8 | 12 |

end of the method, so return to the call for the 4-element subarray, which now has a sorted left subarray:

| 12 | 8 | 14 | 4 | 6 | 33 | 2 | 27 |

| **8** | **12** | 14 | 4 |
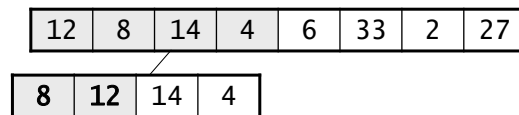
Tracing the Calls to Mergesort

make a recursive call to sort the right subarray of the 4-element subarray
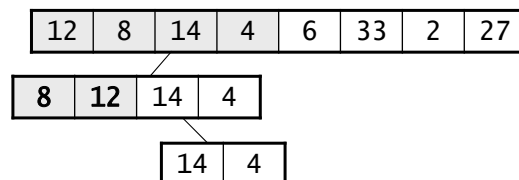
| 12 | 8 | 14 | 4 | 6 | 33 | 2 | 27 |

| **8** | **12** | 14 | 4 |

| 14 | 4 |

split it into two 1-element subarrays, and make a recursive call to sort the left subarray:

| 12 | 8 | 14 | 4 | 6 | 33 | 2 | 27 |

| **8** | **12** | 14 | 4 |

| 14 | 4 |

| 14 | base case…

Tracing the Calls to Mergesort

return to the call for the subarray {14, 4}:

| 12 | 8 | 14 | 4 | 6 | 33 | 2 | 27 |

| **8** | **12** | 14 | 4 |

| 14 | 4 |

make a recursive call to sort its right subarray:

| 12 | 8 | 14 | 4 | 6 | 33 | 2 | 27 |

| **8** | **12** | 14 | 4 |

| 14 | 4 |

| 4 | base case…

Tracing the Calls to Mergesort

return to the call for the subarray {14, 4}:

| 12 | 8 | 14 | 4 | 6 | 33 | 2 | 27 |

| 8 | 12 | 14 | 4 |

| 14 | 4 |

merge the sorted halves of {14, 4}:

| 12 | 8 | 14 | 4 | 6 | 33 | 2 | 27 |

| 8 | 12 | 14 | 4 |

| 14 | 4 | ➡ | 4 | 14 |
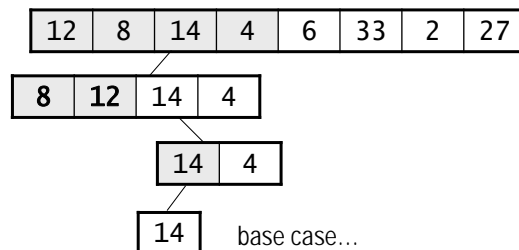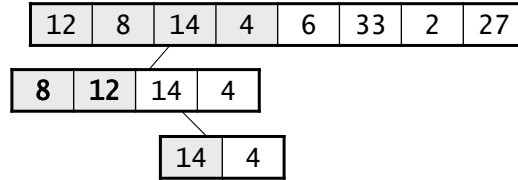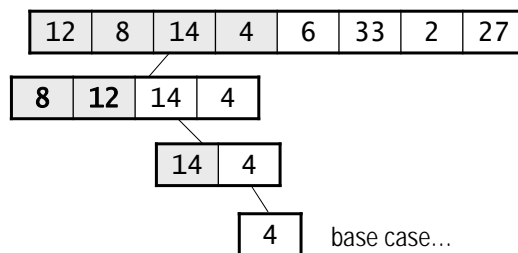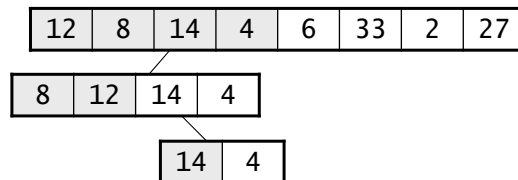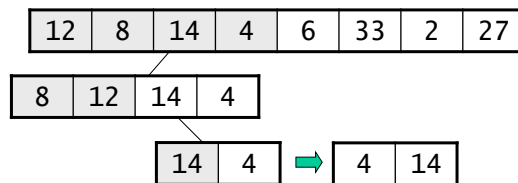
Tracing the Calls to Mergesort

end of the method, so return to the call for the 4-element subarray, which now has two sorted 2-element subarrays:

12	8	14	4	6	33	2	27

8	12	4	14
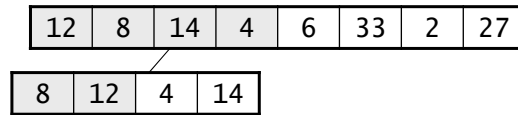
merge the 2-element subarrays:

12	8	14	4	6	33	2	27

8	12	4	14
➡
4	8	12	14
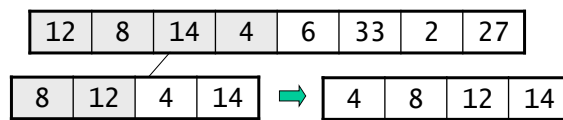
Tracing the Calls to Mergesort

end of the method, so return to the call for the original array, which now has a sorted left subarray:

4	8	12	14	6	33	2	27
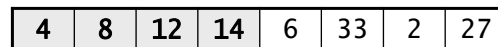
perform a similar set of recursive calls to sort the right subarray. here's the result:

4	8	12	14	2	6	27	33
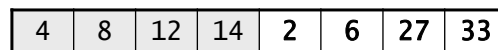
finally, merge the sorted 4-element subarrays to get a fully sorted 8-element array:

4	8	12	14	2	6	27	33

⬇

2	4	6	8	12	14	27	33
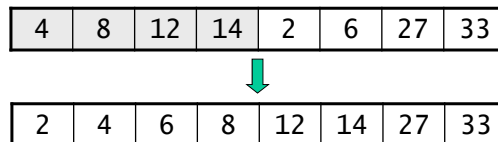
Implementing Mergesort

- In theory, we could create new arrays for each new pair of subarrays, and merge them back into the array that was split.

- Instead, we'll create a temp. array of the same size as the original.
  - pass it to each call of the recursive mergesort method
  - use it when merging subarrays of the original array:

arr | 8 | 12 | 4 | 14 | 6 | 33 | 2 | 27 |

temp | 4 | 8 | 12 | 14 | | | | |
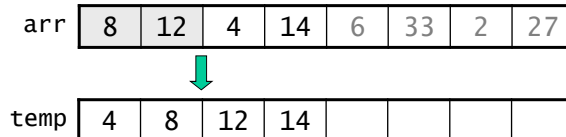
  - after each merge, copy the result back into the original array:
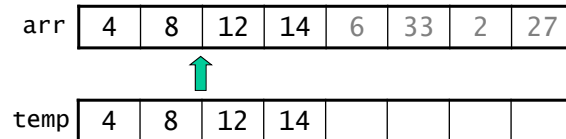
arr | 4 | 8 | 12 | 14 | 6 | 33 | 2 | 27 |

temp | 4 | 8 | 12 | 14 | | | | |

A Method for Merging Subarrays

```
private static void merge(int[] arr, int[] temp,
  int leftStart, int leftEnd, int rightStart, int rightEnd) {
    int i = leftStart;     // index into left subarray
    int j = rightStart;    // index into right subarray
    int k = leftStart;     // index into temp

    while (i <= leftEnd && j <= rightEnd) {
        if (arr[i] < arr[j]) {
            temp[k] = arr[i];
            i++; k++;
        } else {
            temp[k] = arr[j];
            j++; k++;
        }
    }

    while (i <= leftEnd) {
        temp[k] = arr[i];
        i++; k++;
    }
    while (j <= rightEnd) {
        temp[k] = arr[j];
        j++; k++;
    }

    for (i = leftStart; i <= rightEnd; i++) {
        arr[i] = temp[i];
    }
}
```

A Method for Merging Subarrays

```
private static void merge(int[] arr, int[] temp,
  int leftStart, int leftEnd, int rightStart, int rightEnd) {
    int i = leftStart;     // index into left subarray
    int j = rightStart;    // index into right subarray
    int k = leftStart;     // index into temp

    while (i <= leftEnd && j <= rightEnd) { // both subarrays still have values
        if (arr[i] < arr[j]) {
            temp[k] = arr[i];
            i++; k++;
        } else {
            temp[k] = arr[j];
            j++; k++;
        }
    }
    ...
}
```
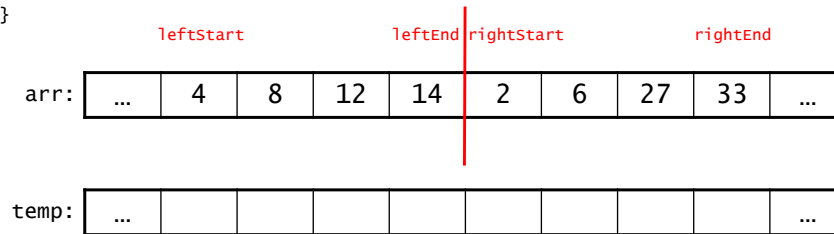
		leftStart			leftEnd	rightStart			rightEnd	
arr:	…	4	8	12	14	2	6	27	33	…

temp:	…									…

Methods for Mergesort

- Here's the key recursive method:

```
private static void mSort(int[] arr, int[] temp, int start, int end){
    if (start >= end) {  // base case: subarray of length 0 or 1
        return;
    } else {
        int middle = (start + end)/2;

        mSort(arr, temp, start, middle);
        mSort(arr, temp, middle + 1, end);

        merge(arr, temp, start, middle, middle + 1, end);
    }
}
```

		start							end	
arr:	…	12	8	14	4	6	33	2	27	…
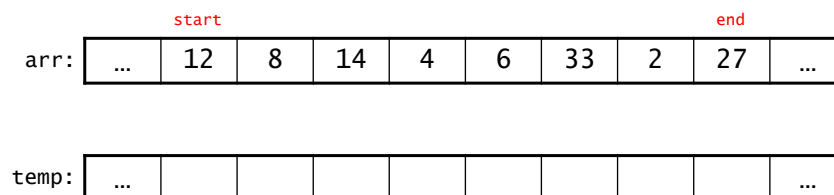
temp:	…									…

What does the array look like after the first recursive call returns?

- Here's the key recursive method:

```
private static void mSort(int[] arr, int[] temp, int start, int end){
    if (start >= end) {  // base case: subarray of length 0 or 1
        return;
    } else {
        int middle = (start + end)/2;

        mSort(arr, temp, start, middle);
        mSort(arr, temp, middle + 1, end);

        merge(arr, temp, start, middle, middle + 1, end);
    }
}
```

		start						end	
arr:	...	12	8	14	4	6	33	2	27	...
A:	...	4	8	12	14	6	33	2	27	...
B:	...	8	12	14	4	6	33	2	27	...
C:	...	12	8	14	4	2	6	27	33	...
D:	...	4	8	12	14	2	6	27	33	...

Methods for Mergesort

- Here's the key recursive method:

```
private static void mSort(int[] arr, int[] temp, int start, int end){
    if (start >= end) {  // base case: subarray of length 0 or 1
        return;
    } else {
        int middle = (start + end)/2;

        mSort(arr, temp, start, middle);
        mSort(arr, temp, middle + 1, end);

        merge(arr, temp, start, middle, middle + 1, end);
    }
}
```

- We use a "wrapper" method to create the `temp` array,
  and to make the initial call to the recursive method:

```
public static void mergeSort(int[] arr) {
    int[] temp = new int[arr.length];
    mSort(arr, temp, 0, arr.length - 1);
}
```

Time Analysis of Mergesort

- Merging two halves of an array of size n requires 2n moves. Why?

- Mergesort repeatedly divides the array in half, so we have the following call tree (showing the sizes of the arrays):

$$\underline{moves}$$
$$2n$$

$$2*2*(n/2) = 2n$$

$$4*2*(n/4) = 2n$$

$$\ldots$$
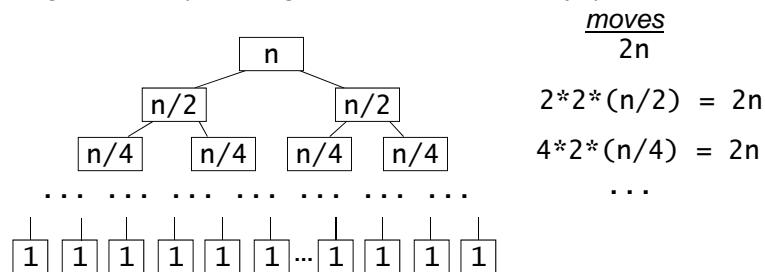
- at all but the last level of the call tree, there are 2n moves
- how many levels are there?
- `M(n) = ?`
- `C(n) = ?`

Summary: Sorting Algorithms

algorithm	best case	avg case	worst case	extra memory
selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	*best/avg:* $O(\log n)$ *worst:* $O(n)$
mergesort	$O(n \log n)$	$O(n \log n)$	$O(n\log n)$	$O(n)$

- Insertion sort is best for nearly sorted arrays.

- Mergesort has the best worst-case complexity, but requires O(n) extra memory – and moves to and from the temp. array.

- Quicksort is comparable to mergesort in the best/average case.
    - efficiency is also $O(n \log n)$, but less memory and fewer moves
    - its extra memory is from…
    - with a reasonable pivot choice, its worst case is seldom seen

Extra Practice: What does partitioning give here?

- Start
(pivot = 9):

8	10	7	*9*	20	3	6	18

A.

8	6	7	3	20	10	9	18

B.

7	8	9	10	20	3	6	18

C.

8	6	7	9	20	3	10	18

D.

8	6	7	3	20	9	10	18

Linked Lists

Computer Science 112
Boston University

Representing a Sequence of Data

- Sequence – an ordered collection of items (position matters)
  - we will look at several types: lists, stacks, and queues

- Most common representation = an array

- Advantages of using an array:
  - easy and efficient access to *any* item in the sequence
    - `items[i]` gives you the item at position i in O(1) time
    - known as *random access*
  - very compact (but can waste space if positions are empty)

- Disadvantages of using an array:
  - have to specify an initial array size and resize it as needed
  - inserting/deleting items can require shifting other items
    - ex: insert 63 between 52 and 72

| items | | 31 | 52 | 72 | ... |
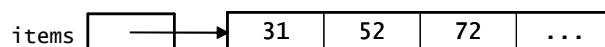
Alternative Representation: A Linked List

items → 31 → 52 → 72 → null
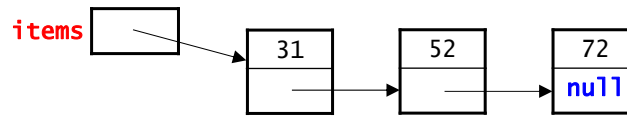
- A linked list stores a sequence of items in separate *nodes*.

- Each node is an object that contains:
  - a single item
  - a "link" (i.e., a reference) to
    the node containing the next item

  31 →

- The last node in the linked list has a link value of null.

- The linked list as a whole is represented by a variable that
  holds a reference to the first node.
  - e.g., items in the example above
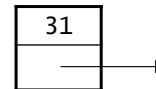
Arrays vs. Linked Lists in Memory

- In an array, the elements occupy consecutive memory locations:

items → 31 | 52 | 72 | ...

0x100 *0x104* *0x108*
items 0x100   31 | 52 | 72 | ...
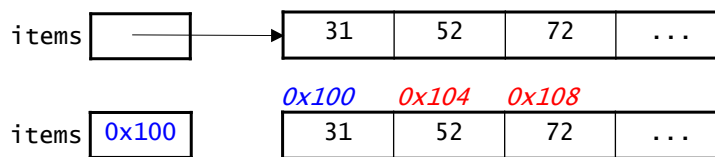
- In a linked list, the nodes are distinct objects.
  - do *not* have to be next to each other in memory
  - that's why we need the links to get from one node to the next!

items → 31 → 52 → 72 → null

0x520   *0x812*   *0x208*
items 0x520   31 | 52 | 72
               0x812 | 0x208 | null
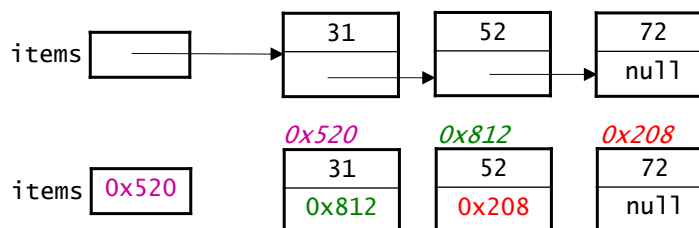
Linked Lists in Memory
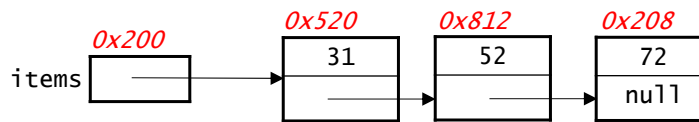
* Here's how the above linked list might actually look in memory:

Features of Linked Lists

* They can grow without limit (provided there is enough memory).

* Easy to insert/delete an item – no need to "shift over" other items.
    * for example, to insert 63 between 52 and 72:
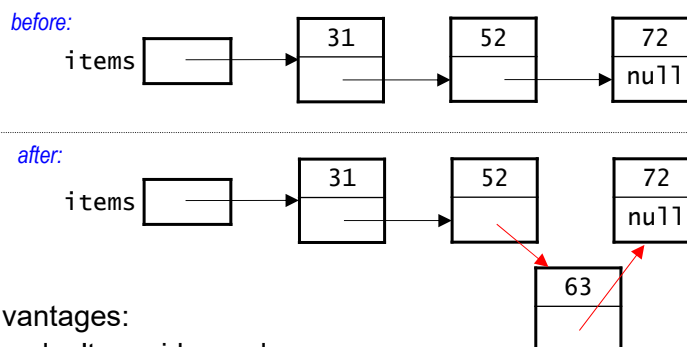
* Disadvantages:
    * they don't provide random access
        * need to "walk down" the list to access an item
    * the links take up additional memory

A String as a Linked List of Characters
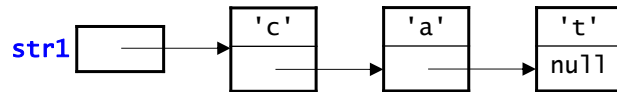
- Each node represents one character.

- Java class for this type of node:

```
public class StringNode {
    private char ch;
    private StringNode next;
             same type as the node itself!

    public StringNode(char c, StringNode n) {
        this.ch = c;
        this.next = n;
    }
    ...
}
```

- The string as a whole is represented by a variable that holds a reference to the node for the first character (e.g., `str1` above).

A String as a Linked List (cont.)

- An empty string will be represented by a null value.
    example:
    ```
    StringNode str2 = null;
    ```

- We will use *static* methods that take the string as a parameter.
    - e.g., we'll write `length(str1)` instead of `str1.length()`
    - outside the class, call the methods using the class name:
        ```
        StringNode.length(str1)
        ```

- This approach allows the methods to handle empty strings.
    - if `str1 == null`:
        - `length(str1)` will work
        - `str1.length()` will throw a `NullPointerException`

Building a Linked List of Characters I

```
public StringNode(char c,
                      StringNode n) {
    this.ch = c;
    this.next = n;
}
```
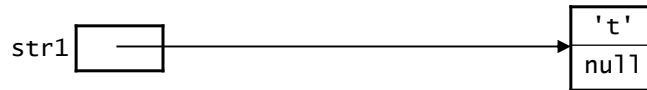
str1 → 't' / null

- We can use the StringNode constructor to build the linked list from the previous slide.

- One way is to start with the last node and work towards the front:

  ```
  StringNode str1 = new StringNode('t', null);
  ```

Building a Linked List of Characters II

```
public StringNode(char c,
                      StringNode n) {
    this.ch = c;
    this.next = n;
}
```
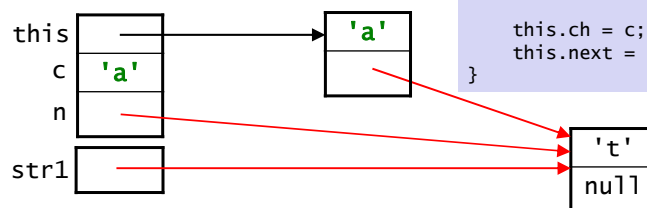
this → 'a'
c 'a'
n
str1 → 't' / null

- We can use the StringNode constructor to build the linked list from the previous slide.

- One way is to start with the last node and work towards the front:

  ```
  StringNode str1 = new StringNode('t', null);
  str1 = new StringNode('a', str1);
  ```

Building a Linked List of Characters III

```
public StringNode(char c,
                  StringNode n) {
    this.ch = c;
    this.next = n;
}
```
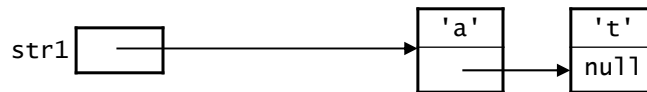
str1 → 'a' → 't' / null

* We can use the `StringNode` constructor to build the linked list from the previous slide.

* One way is to start with the last node and work towards the front:

```
StringNode str1 = new StringNode('t', null);
str1 = new StringNode('a', str1);
```

Building a Linked List of Characters IV

```
public StringNode(char c,
                  StringNode n) {
    this.ch = c;
    this.next = n;
}
```
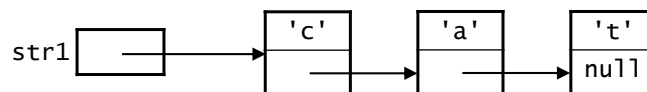
str1 → 'c' → 'a' → 't' / null

* We can use the `StringNode` constructor to build the linked list from the previous slide.

* One way is to start with the last node and work towards the front:

```
StringNode str1 = new StringNode('t', null);
str1 = new StringNode('a', str1);
str1 = new StringNode('c', str1);
```

* Later, we'll see methods that can be used to build a linked list and add nodes to it.

Review of Variables

- A variable or variable expression represents both:
    - a "box" or location in memory (the *address* of the variable)
    - the contents of that "box" (the *value* of the variable)

- Practice:

0x200 str

0x520 ch 'd'   *0x812* 'o'   *0x208* 'g'

0x204 temp   next   null

```
StringNode str;    // points to the first node
StringNode temp;   // points to the second node
```
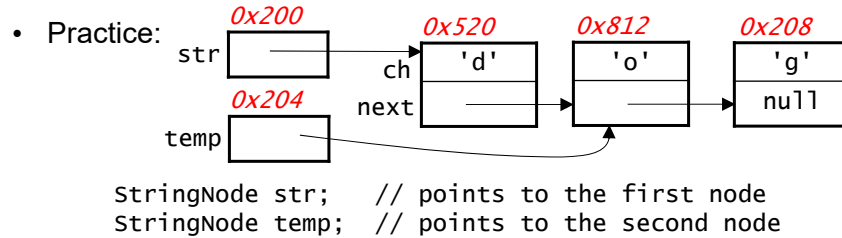
expression	address	value
str	0x200	0x520 (ref to the 'd' node)
str.ch		
str.next		

Assumptions:
- `ch` *field has the same memory address as the node itself.*
- `next` *field comes 2 bytes after the start of the node.*

More Complicated Expressions

0x200 str

0x520 'd'   *0x812* 'o'   *0x208* 'g'

0x204 temp   null
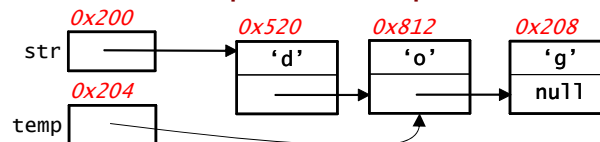
- Example: `temp.next.ch`

- Start with the beginning of the expression: `temp.next`
  It represents the next field of the node to which `temp` refers.
    - address =
    - value =

- Next, consider `temp.next.ch`
  It represents the ch field of the node to which `temp.next` refers.
    - address =
    - value =

Find the address and value of `s.next.next.ch`

	0xc000	0xbe00	0x3004	0xbb00	0xa004
s →	'b'	'r'	'e'	'a'	'k' / null
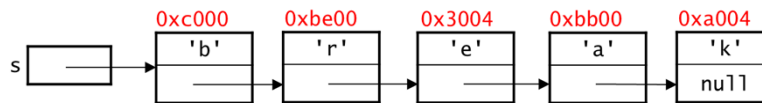
- `s.next` is…
  - it holds…
- thus, `s.next.next` is…
  - it holds…
- thus, `s.next.next.ch` is…
  - it holds…

	address	value
A.	0xbe00	'r'
B.	0x3004	'e'
C.	0xbb00	'a'
D.	none of these	

Review of Assignment Statements

- An assignment of the form

      var1 = var2;

  - takes the value inside `var2`
  - copies it into `var1`

- Example involving integers:

      int x = 5;
      int y = x;
            5

  0x400
  x | 5 |
  0x804
  y | 5 |
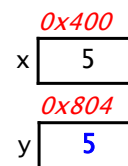
- Example involving references:

      int[] a1 = {3, 4, 5};
      int[] a2 = a1;
            0x320

  0x600   0x320
  a1 | → | 3 | 4 | 5 |
  0x256
  a2 | |
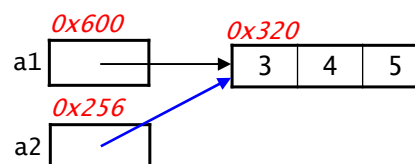
What About These Assignments?
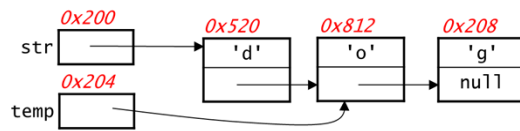
- *Identify the two boxes.*
- *Determine the value in the box specified by the right-hand side.*
- *Copy that value into the box specified by the left-hand side.*

1) `str.next = temp.next;`

2) `temp.next = temp.next.next;`

Writing an Appropriate Assignment

- If `temp` didn't already refer to the `'o'` node, what assignment would be needed to make it refer to that node?
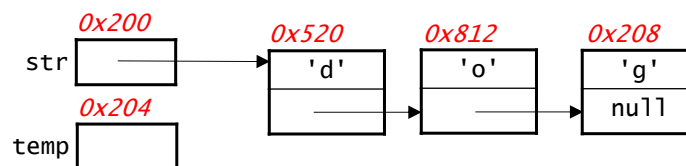
- start by asking: where do I currently have a reference to the `'o'` node?

- then ask: what expression can I use for that box?

- then write the assignment:

A Linked List Is a Recursive Data Structure!

- Recursive definition: a linked list is either
  - a) empty or
  - b) a single node, followed by a linked list

- Viewing linked lists in this way allows us to write recursive methods that operate on linked lists.

Recursively Finding the Length of a String

- For a Java `String` object:
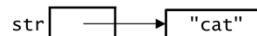

```java
public static int length(String str) {
    if (str.equals("")) {
        return 0;
    } else {
        int lenRest = length(str.substring(1));
        return 1 + lenRest;
    }
}
```

- For a linked-list string:
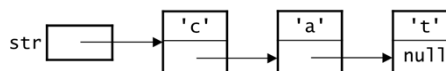

```java
public static int length(StringNode str) {
    if (str == null) {
        return 0;
    } else {
        int lenRest = length(str.next);
        return 1 + lenRest;
    }
}
```

An Alternative Version of the Method

- Original version:
```
public static int length(StringNode str) {
    if (str == null) {
        return 0;
    } else {
        int lenRest = length(str.next);
        return 1 + lenRest;
    }
}
```

- Version without a variable for the result of the recursive call:
```
public static int length(StringNode str) {
    if (str == null) {
        return 0;
    } else {
        return 1 + length(str.next);
    }
}
```

Tracing `length()`

```
public static int length(StringNode str) {
    if (str == null) {
        return 0;
    } else {
        return 1 + length(str.next);
    }
}
```
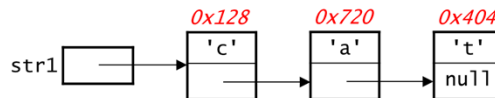

- Example: `StringNode.length(str1)`
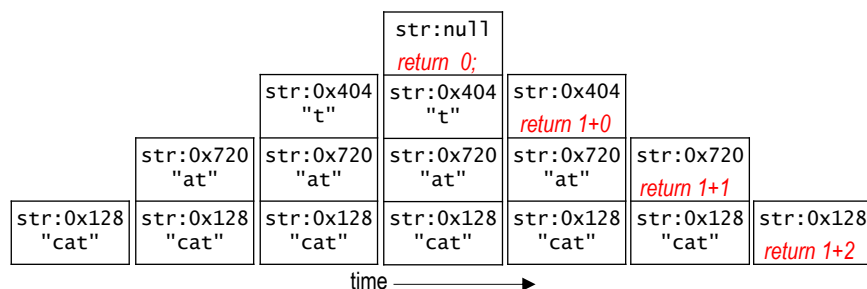
Using Iteration to Traverse a Linked List

- Many tasks require us to traverse or "walk down" a linked list.

- We just saw a method that used recursion to do this.

- It can also be done using iteration (for loops, while loops, etc.).

- We make use of a variable (call it `trav`) that keeps track of where we are in the linked list.
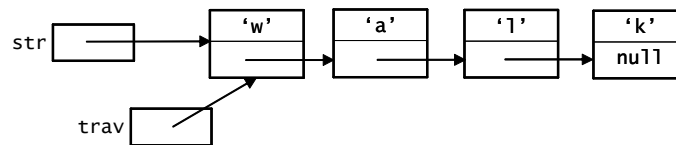
- Template for traversing an entire linked list:
```
StringNode trav = str;      // start with first node
while (trav != null) {
    // process the current node here
    trav = trav.next;       // move trav to next node
}
```

Example of Iterative Traversal

- `toUpperCase(str)`: converting `str` to all upper-case letters
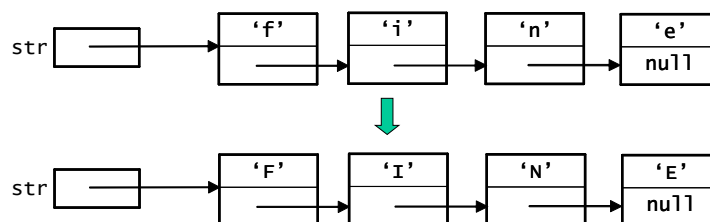
- Similar to the built-in method for Java `String` objects.

- This method processes linked-list strings:
  - uses a loop to process one `StringNode` at a time
  - modifies the internals of the string (unlike the built-in version)
  - thus, it doesn't need to return anything

Example of Iterative Traversal (cont.)

- `toUpperCase(str)`: converting `str` to all upper-case letters
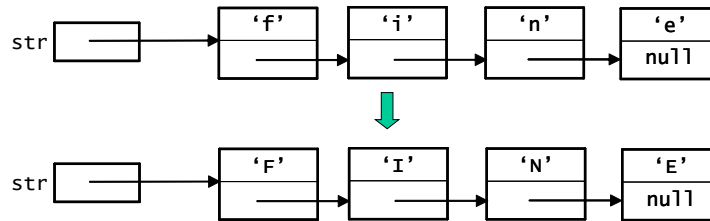
- Here's the method:

```
public static void toUpperCase(StringNode str) {
    StringNode trav = str;
    while (trav != null) {
        trav.ch = Character.toUpperCase(trav.ch);
        trav = trav.next;
    }
}
```

- uses a built-in static method from the `Character` class to convert a single `char` to upper case

Tracing `toUpperCase()`: Before the Loop
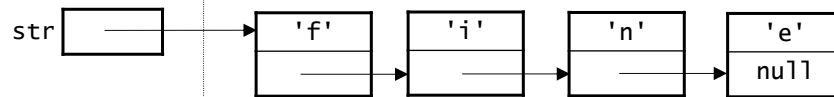
Calling `StringNode.toUpperCase(str)` adds a stack frame to the stack:
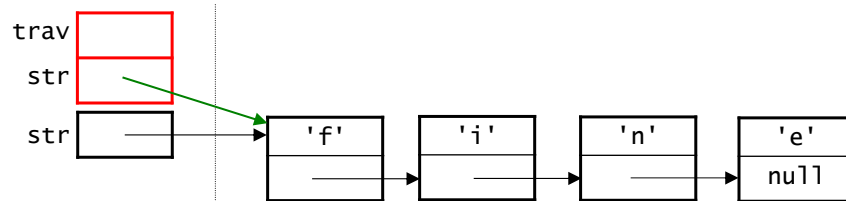
`StringNode trav = str;`
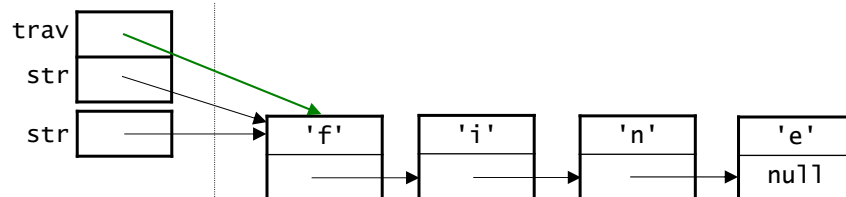
Tracing `toUpperCase()`: First Iteration of Loop

```
while (trav != null) {
    trav.ch = Character.toUpperCase(trav.ch);
    trav = trav.next;
}
```

after updating `trav.ch`:

trav

str

str 'F' 'i' 'n' 'e'
null

after updating `trav`:

trav

str

str 'F' 'i' 'n' 'e'
null
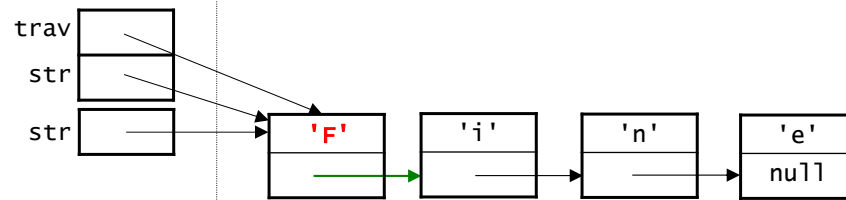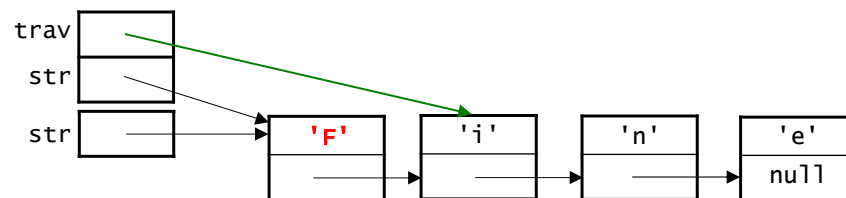
Tracing `toUpperCase()`: Second Iteration

```
while (trav != null) {
    trav.ch = Character.toUpperCase(trav.ch);
    trav = trav.next;
}
```

after updating `trav.ch`:

trav

str

str 'F' 'I' 'n' 'e'
null

after updating `trav`:

trav

str

str 'F' 'I' 'n' 'e'
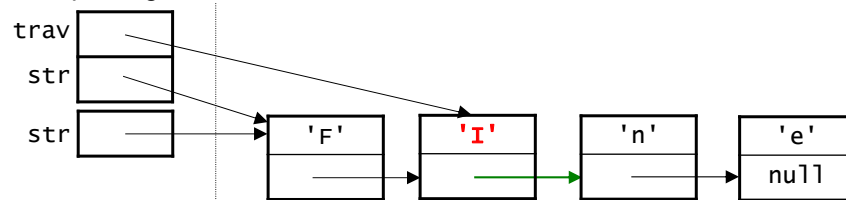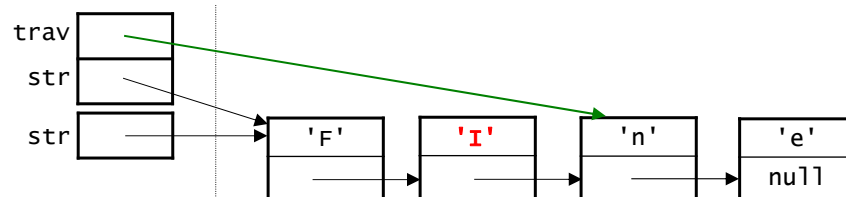
Tracing `toUpperCase()`: Third Iteration

```
while (trav != null) {
    trav.ch = Character.toUpperCase(trav.ch);
    trav = trav.next;
}
```

after updating `trav.ch`:

trav

str

str | 'F' | 'I' | **'N'** | 'e' |
| | | | null |

after updating `trav`:

trav

str

str | 'F' | 'I' | **'N'** | 'e' |
| | | | null |
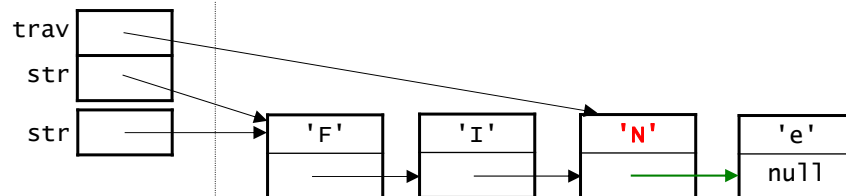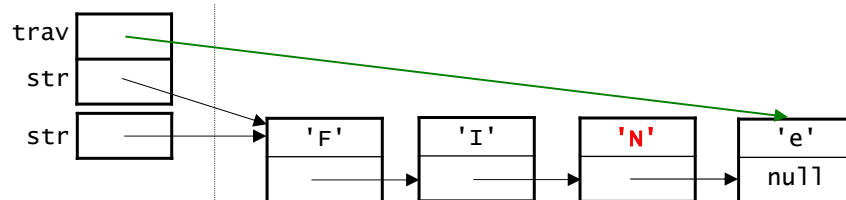
Tracing `toUpperCase()`: Fourth Iteration

```
while (trav != null) {
    trav.ch = Character.toUpperCase(trav.ch);
    trav = trav.next;
}
```

after updating `trav.ch`:

trav

str

str | 'F' | 'I' | 'N' | **'E'** |
| | | | null |

after updating `trav`:

trav **null**

str

str | 'F' | 'I' | 'N' | **'E'** |
| | | | null |
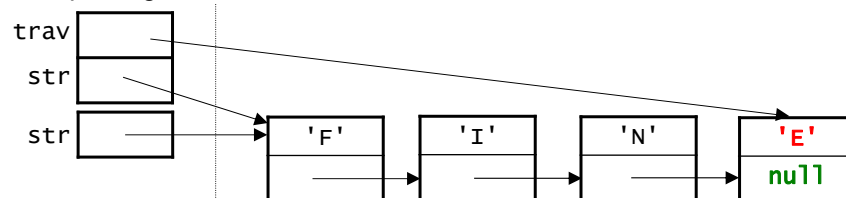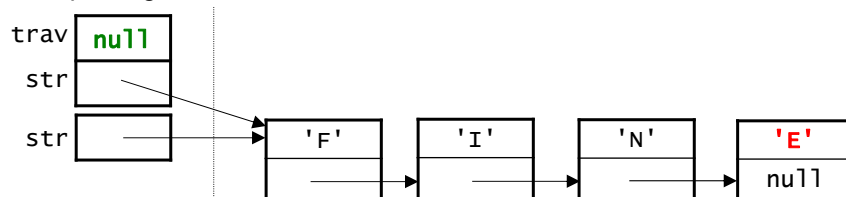
Tracing `toUpperCase()`: Finishing Up

```
while (trav != null) {
    trav.ch = Character.toUpperCase(trav.ch);
    trav = trav.next;
}
```

results of the final iteration:

trav	**null**	
str		
str		→ 'F' → 'I' → 'N' → 'E' / null
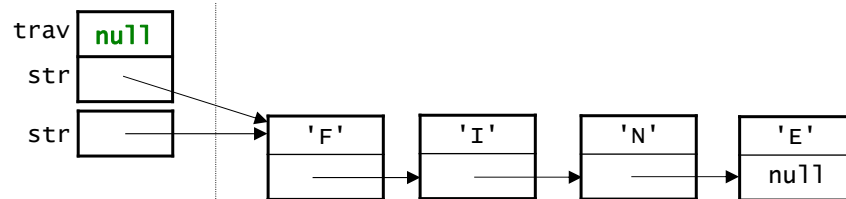
and now `trav == null`, so we end the loop and return:

str → 'F' → 'I' → 'N' → 'E' / null
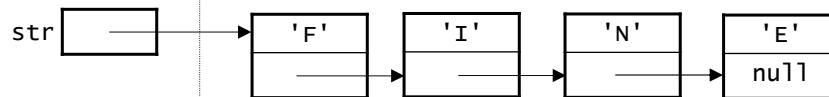
Getting the Node at Position i in a Linked List

- `getNode(str, i)` – should return a reference to the `i`th node in the linked list to which `str` refers

  str → 'f' → 'i' → 'n' → 'e' / null
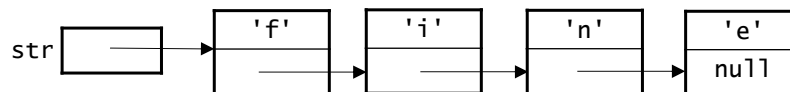
- Examples:
  - `getNode(str, 0)` should return a ref. to the `'f'` node
  - `getNode(str, 3)` should return a ref. to the `'e'` node
  - `getNode(`**`str.next`**`, 2)` should return a ref. to…?

- More generally, when $0 < i <$ *length of list*,
  `getNode(str, i)` is equivalent to `getNode(str.next, i-1)`

Getting the Node at Position i in a Linked List
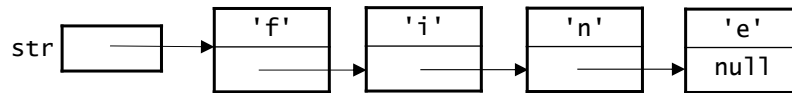
- Recursive approach to `getNode(str, i)`:
    - if `i == 0`, return `str` (base case)
    - else call `getNode(str.next, i-1)` and return what it returns!
    - other base case?

- Here's the method:
```
private static StringNode getNode(StringNode str, int i) {
    if (i < 0 || str == null) {  // base case 1
        return null;
    } else if (i == 0) {         // base case 2: just found
        return str;
    } else {
        return getNode(str.next, i-1);
    }
}
```

Deleting the Item at Position i

- <u>Special case:</u> `i == 0` (deleting the first item)

- Update our reference to the first node by doing:
    `str = str.next;`
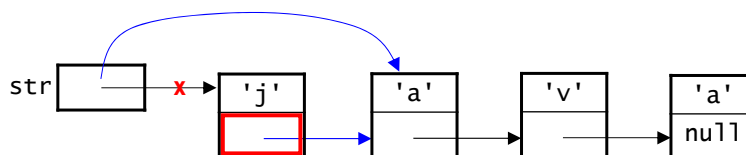
Deleting the Item at Position i (cont.)

- <u>General case:</u> `i > 0`

    1. Obtain a reference to the *previous* node:

        `StringNode prevNode = getNode(i - 1);`

    (example for i == 1)
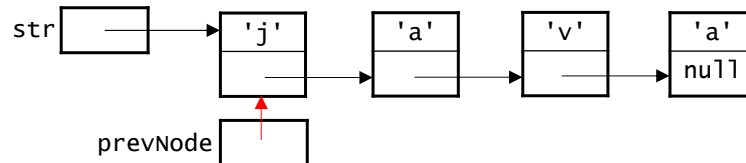
Deleting the Item at Position i (cont.)

- <u>General case:</u> `i > 0`

    2. Update the references to remove the node

    (example for i == 1)

    before:

    after:

$$\underline{\hspace{3cm}} = \underline{\hspace{4cm}};$$
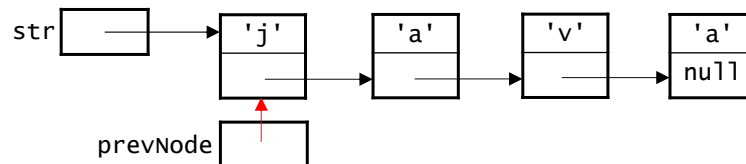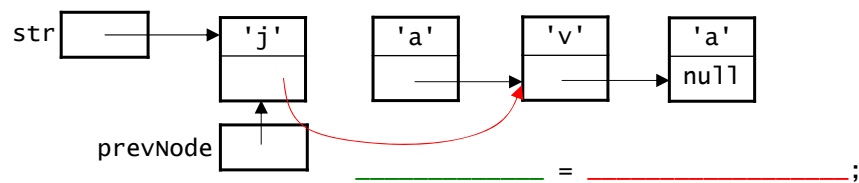
Inserting an Item at Position i

- Special case: `i == 0` (insertion at the front of the list)

- Step 1: *Create* the new node. *Fill in the blanks!*

before:

```
ch  'f'
```

```
str  →  'a' → 'c' → 'e'
                      null
```

after:

```
ch  'f'        'f'
newNode  →
str  →         'a' → 'c' → 'e'
                            null
```
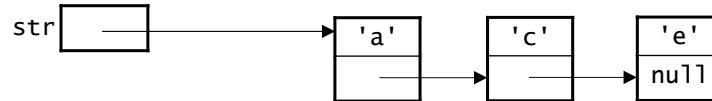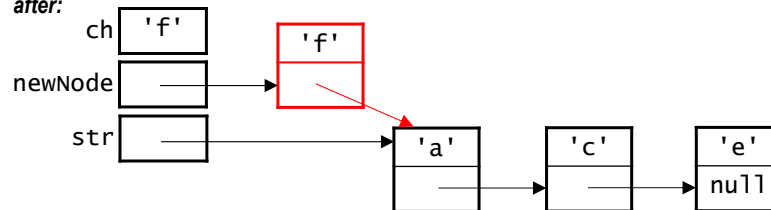
```
StringNode newNode = new StringNode(_____, _____);
```

Inserting an Item at Position i (cont.)

- Special case: `i == 0` (continued)

- Step 2: *Insert* the new node. *Write the assignment!*

before (result of previous slide):

```
ch  'f'        'f'
newNode  →
str  →         'a' → 'c' → 'e'
                            null
```

after:

```
ch  'f'        'f'
newNode  →
str  →         'a' → 'c' → 'e'
                            null
```
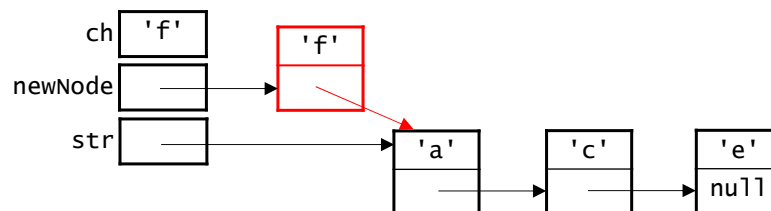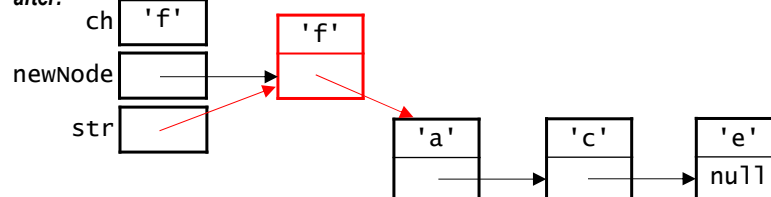
Inserting an Item at Position i (cont.)

* <u>General case:</u> `i > 0` (insert *before* the item currently in posn i)

  before:
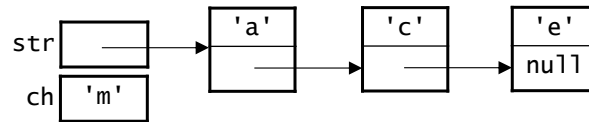
  after *(assume that i == 2):*
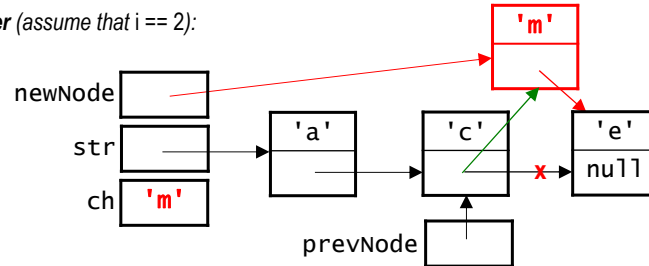

```
StringNode prevNode = getNode(i - 1);
StringNode newNode = new StringNode(ch, _____);
_____  // one more line
```

Returning a Reference to the First Node

* Both `deleteChar()` and `insertChar()` return a reference to the first node in the linked list. For example:

```
public static StringNode deleteChar(StringNode str, int i) {
    …
    if (i == 0) {                    // special case
        str = str.next;
    } else {                         // general case
        StringNode prevNode = getNode(str, i-1);
        if (prevNode != null && prevNode.next != null) {
            prevNode.next = prevNode.next.next;
            …
        }
    }

    return str;
}
```

* Clients should call them as part of an assignment:

```
s1 = StringNode.deleteChar(s1, 0);
s2 = StringNode.insertChar(s2, 0, 'h');
```

* If the first node changes, the client's variable will be updated to point to the new first node.

Creating a Copy of a Linked List

- `copy(str)` – create a copy of *the entire list* to which `str` refers

- Recursive approach:
  - base case: if `str` is empty, return `null`
  - else: – make a recursive call to copy the rest of the linked list
    – create and return a copy of the first node,
       with its next field pointing to the copy of the rest

```
public static StringNode copy(StringNode str) {
    if (str == null) {          // base case
        return null;
    }

    // make a recursive call to copy the rest of the list
    StringNode copyRest = copy(str.next);

    // create and return a copy of the first node,
    // with its next field pointing to the copy of the rest
    return new StringNode(str.ch, copyRest);
}
```

Tracing `copy()`: the initial call

- From a client: `StringNode s2 = StringNode.copy(s1);`

```
public static StringNode copy(StringNode str) {
    if (str == null) {
        return null;
    }

    StringNode copyRest = copy(str.next);
    return new StringNode(str.ch, copyRest);
}
```
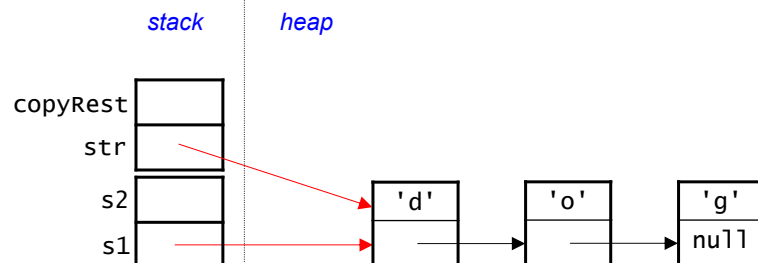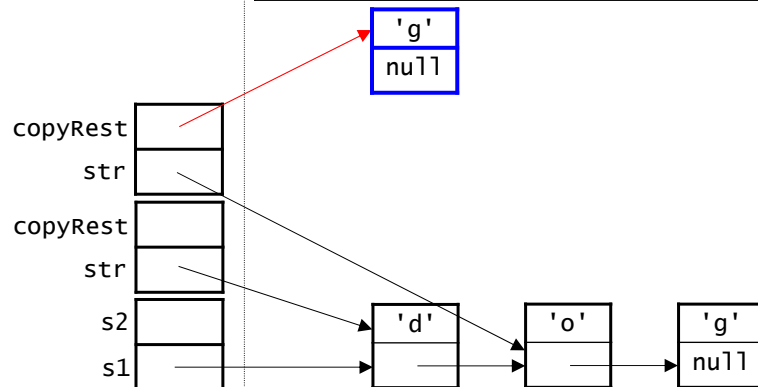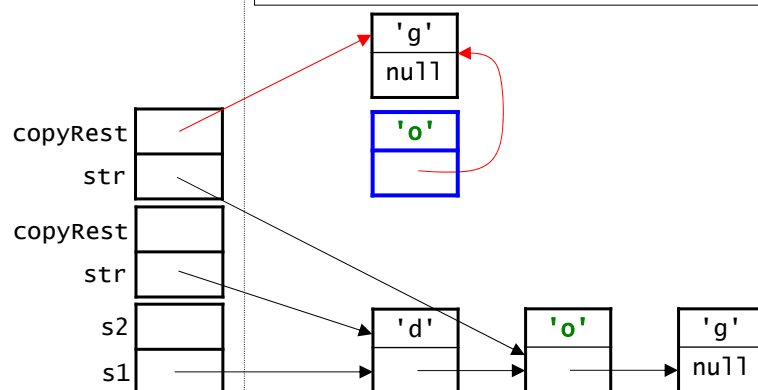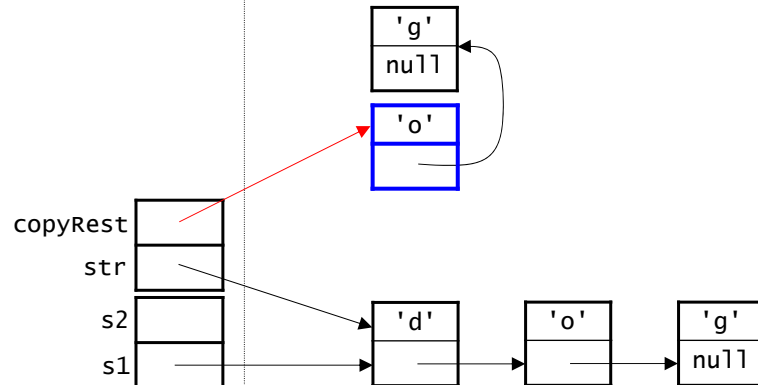
Tracing copy(): the recursive calls

- From a client: `StringNode s2 = StringNode.copy(s1);`

```
public static StringNode copy(StringNode str) {
    if (str == null) {
        return null;
    }
    StringNode copyRest = copy(str.next);
    return new StringNode(str.ch, copyRest);
}
```


Tracing copy(): the recursive calls

```
public static StringNode copy(...) {
    if (str == null) {
        return null;
    }
    StringNode copyRest = copy(str.next);
    return new StringNode(str.ch,
                          copyRest);
}
```
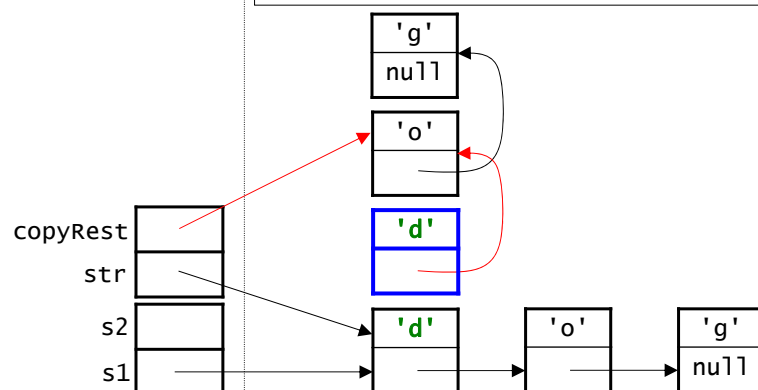
Tracing copy(): the recursive calls

```
public static StringNode copy(...) {
    if (str == null) {
        return null;
    }

    StringNode copyRest = copy(str.next);
    return new StringNode(str.ch,
                          copyRest);
}
```

copyRest

str **null**

copyRest

str

copyRest

str

copyRest

str

s2

s1

'd'   'o'   'g'
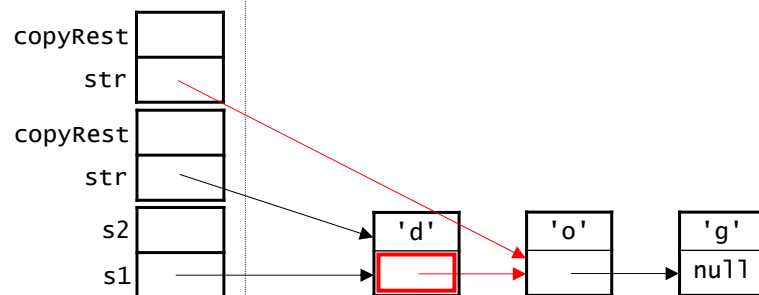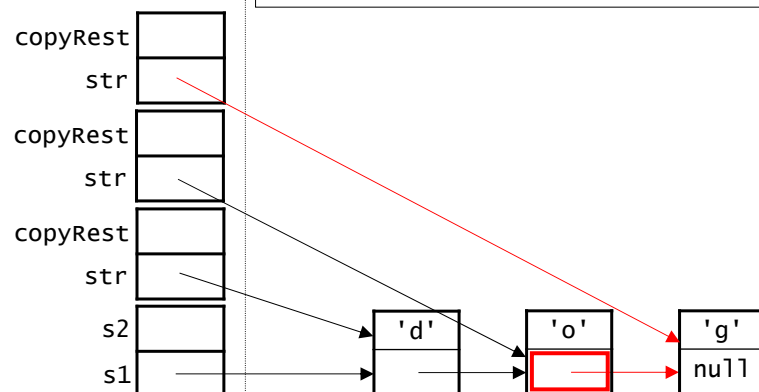
null

Tracing copy(): the base case

```
public static StringNode copy(...) {
    if (str == null) {
        return null;
    }

    StringNode copyRest = copy(str.next);
    return new StringNode(str.ch,
                          copyRest);
}
```

copyRest

str **null**

copyRest

str

copyRest

str

copyRest

str

s2

s1

'd'   'o'   'g'

Tracing copy(): returning from the base case

```
public static StringNode copy(...) {
  if (str == null) {
      return null;
  }

  StringNode copyRest = copy(str.next);
  return new StringNode(str.ch,
                          copyRest);
}
```

copyRest | null
str |
copyRest |
str |
copyRest |
str |
s2 |
s1 |

'd' | → 'o' | → 'g' | null

Tracing copy(): returning from the base case

```
public static StringNode copy(...) {
  if (str == null) {
      return null;
  }

  StringNode copyRest = copy(str.next);
  return new StringNode(str.ch,
                          copyRest);
}
```

copyRest | null
str |
copyRest |
str |
copyRest |
str |
s2 |
s1 |

'g'
null

'd' | → 'o' | → 'g' | null

Tracing copy(): returning from the base case

```
public static StringNode copy(...) {
  if (str == null) {
      return null;
  }

  StringNode copyRest = copy(str.next);
  return new StringNode(str.ch,
                        copyRest);
}
```

copyRest

str

copyRest

str

s2

s1

'g'
null

'd'   'o'   'g'
              null

Tracing copy(): returning from the base case

```
public static StringNode copy(...) {
  if (str == null) {
      return null;
  }

  StringNode copyRest = copy(str.next);
  return new StringNode(str.ch,
                        copyRest);
}
```

copyRest

str

copyRest

str

s2

s1

'g'
null

'o'

'd'   'o'   'g'

Tracing copy(): returning from the base case

```
public static StringNode copy(...) {
    if (str == null) {
        return null;
    }

    StringNode copyRest = copy(str.next);
    return new StringNode(str.ch,
                          copyRest);
}
```

'g'
null

'o'

copyRest
str
s2
s1

'd'   'o'   'g'
                null

Tracing copy(): returning from the base case

```
public static StringNode copy(...) {
    if (str == null) {
        return null;
    }

    StringNode copyRest = copy(str.next);
    return new StringNode(str.ch,
                          copyRest);
}
```

'g'
null

'o'

copyRest
str
s2
s1

'd'

'd'   'o'   'g'

Tracing `copy()`: Final Result

- From a client: `StringNode s2 = StringNode.copy(s1);`

- `s2` now holds a reference to a linked list that is a copy of the linked list to which `s1` holds a reference.
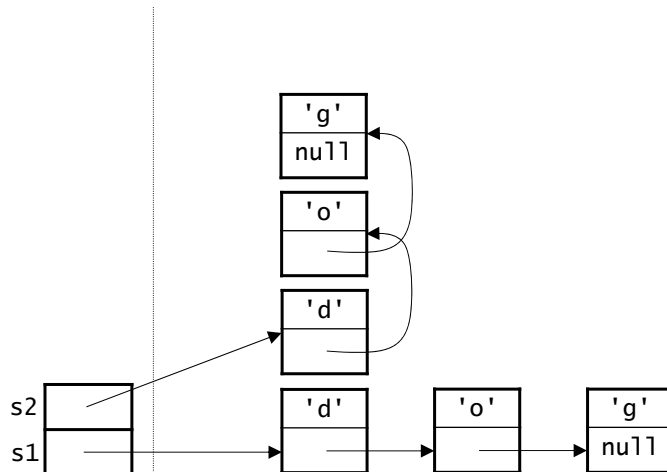
Using a "Trailing Reference" During Traversal

- When traversing a linked list, one `trav` may not be enough.

- Ex: insert `ch = 'n'` at the right place in this *sorted* linked list:
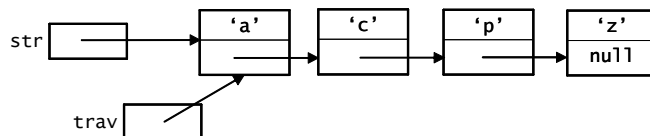
- Traverse the list to find the right position:

```
StringNode trav = str;
while (trav != null && trav.ch < ch) {
    trav = trav.next;
}
```

- When we exit the loop, where will `trav` point? Can we insert 'n'?

- The following changed version doesn't work either. Why not?

```
while (trav != null && trav.next.ch < ch) {
    trav = trav.next;
}
```

Using a "Trailing Reference" (cont.)

* To get around the problem seen on the previous page,
  we traverse the list using two different references:
  * `trav`, which we use as before
  * `trail`, which stays one node behind `trav`
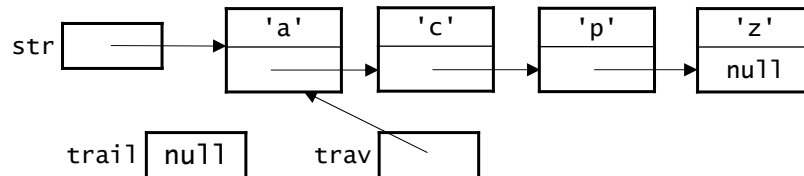

```
StringNode trav = str;
StringNode trail = null;
while (trav != null && trav.ch < ch) {
    trail = trav;
    trav = trav.next;
}
// if trail == null, insert at the front of the list
// else insert after the node to which trail refers
```

Using a "Trailing Reference" (cont.)

* To get around the problem seen on the previous page,
  we traverse the list using two different references:
  * `trav`, which we use as before
  * `trail`, which stays one node behind `trav`
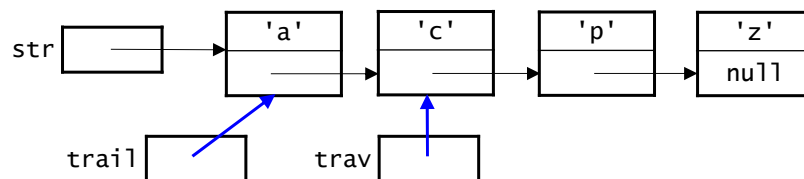

```
StringNode trav = str;
StringNode trail = null;
while (trav != null && trav.ch < ch) {
    trail = trav;
    trav = trav.next;
}
// if trail == null, insert at the front of the list
// else insert after the node to which trail refers
```

Doubly Linked Lists
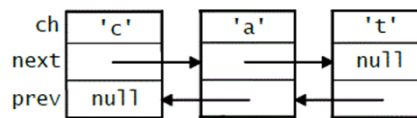
- In a doubly linked list, every node stores *two* references:
  - `next`, which works the same as before
  - `prev`, which holds a reference to the previous node
    - in the first node, `prev` has a value of `null`

- The `prev` references allow us to "back up" as needed.
  - remove the need for a trailing reference during traversal!

- Insertion and deletion must update both types of references.

What expression using `t` would give us `'e'`?
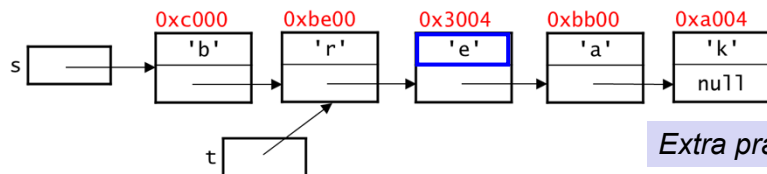
Extra practice!

A. `t.next.ch`
B. `t.next`
C. `t.next.next.ch`
D. `t.next.next`

What are the address and value of `str.next.next`?
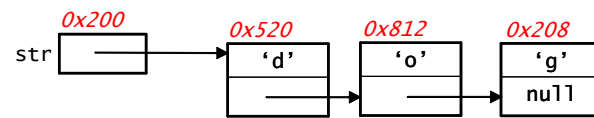
Extra practice!

	address	value
A.	0x522	0x812
B.	0x812	'o'
C.	0x814	0x208
D.	0x208	'g'
E.	0x210	null

ADTs and Interfaces

The List ADT

Computer Science 112
Boston University

Representing a Sequence: Arrays vs. Linked Lists

- Sequence – an ordered collection of items (position matters)
  - we will look at several types: lists, stacks, and queues

- Can represent any sequence using an array *or* a linked list

	array	*linked list*
representation in memory	elements occupy consecutive memory locations	nodes can be at arbitrary locations in memory; the links connect the nodes together
advantages	- provide random access (access to any item in constant time) - no extra memory needed for links	- can grow to an arbitrary length - allocate nodes as needed - inserting or deleting does *not* require shifting items
disadvantages	- have to preallocate the memory needed for the maximum sequence size - inserting or deleting can require shifting items	- no random access (may need to traverse the list) - need extra memory for links

Abstract Data Types

- An *abstract data type* (ADT) is a model of a data structure that specifies:
    - the characteristics of the collection of data
    - the operations that can be performed on the collection

- It's *abstract* because it doesn't specify *how* the ADT will be implemented.

- A given ADT can have multiple implementations.

The List ADT

- A list is a sequence in which items can be accessed, inserted, and removed *at any position in the sequence*.

- The operations supported by our List ADT:
    - `getItem(i)`: get the item at position i
    - `addItem(item, i)`: add the specified item at position i
    - `removeItem(i)`: remove the item at position i
    - `length()`: get the number of items in the list
    - `isFull()`: test if the list already has the maximum number of items

- Note that we *don't* specify *how* the list will be implemented.

Specifying an ADT Using an Interface

- In Java, we can use an *interface* to specify an ADT:

```java
public interface List {
    Object getItem(int i);
    boolean addItem(Object item, int i);
    Object removeItem(int i);
    int length();
    boolean isFull();
}
```

- An interface specifies a set of methods.
  - includes only their headers
  - does *not* typically include the full method definitions

- Like a class, it must go in a file with an appropriate name.
  - in this case: `List.java`

- Methods specified in an interface *must* be public,
  so we don't need the keyword `public` in the headers.

Implementing an ADT Using a Class

- To implement an ADT, we define a class.

- We specify the corresponding interface in the class header:

```java
public class ArrayList implements List {
    ...
```

  - tells the compiler that the class will define *all* of the methods in the interface
  - if the class doesn't define them, it won't compile

- We'll look at two implementations of the `List` interface:
  - `ArrayList` – uses an array to store the items
  - `LLList` – uses a linked list to store the items

Recall: Polymorphism

- A reference variable of type T can hold a reference to an object from a *subclass* of T:

  `Rectangle r1 = new Square(50, "cm");`

    - this works because `Square` is a subclass of `Rectangle`
    - a square *is* a rectangle!

Another Example of Polymorphism

- An interface can be used as the type of a variable:

  ```
  List myList;
  ```

- We can then assign an object of any class that implements the interface:

  ```
  List l1 = new ArrayList(20);
  List l2 = new LLList();
  ```

- This allows us write code that works with *any* implementation of an ADT:

  ```
  public static void processList(List vals) {
      for (int i = 0; i < vals.length(); i++) {
          …
      }
  }
  ```

    - `vals` can be an object of *any* class that implements `List`
    - regardless of which class `vals` is from, we know it has all of the methods in the `List` interface

Implementing a List Using an Array

```
public class ArrayList implements List {
    private Object[] items;
    private int length;

    public ArrayList(int maxSize) {
        // code to check for invalid maxSize goes here...
        this.items = new Object[maxSize];
        this.length = 0;
    }

    public int length() {
        return this.length;
    }

    public boolean isFull() {
        return (this.length == this.items.length);
    }
    ...
}
```


list — *a variable of type* `ArrayList`

items / length / 2 — *an* `ArrayList` *object*

null ... / "if" / "for"
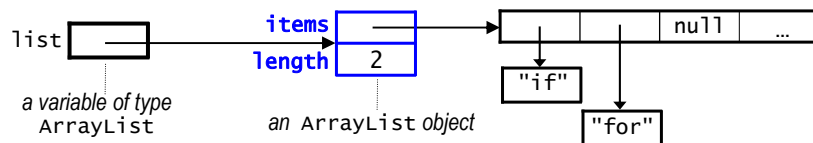
Recall: The Implicit Parameter

```
public class ArrayList implements List {
    private Object[] items;
    private int length;

    public ArrayList(int maxSize) {
        this.items = new Object[maxSize];
        this.length = 0;
    }

    public int length() {
        return this.length;
    }

    public boolean isFull() {
        return (this.length == this.items.length);
    }
    ...
}
```

* All non-static methods have an implicit parameter (`this`)
  that refers to the called object.

* In most cases, we're allowed to omit it!
  * we'll do so in the remaining notes

Omitting The Implicit Parameter

```java
public class ArrayList implements List {
    private Object[] items;
    private int length;

    public ArrayList(int maxSize) {
        items = new Object[maxSize];
        length = 0;
    }
    public int length() {
        return length;
    }
    public boolean isFull() {
        return (length == items.length);
    }
    ...
}
```

* In a non-static method, if we use a variable that
    * isn't declared in the method
    * has the name of one of the fields

    Java assumes that we're using the field.

Adding an Item to an `ArrayList`

* Adding at position i (shifting items i, i+1, … to the right by one):

```java
public boolean addItem(Object item, int i) {
    if (item == null || i < 0 || i > length) {
        throw new IllegalArgumentException();
    } else if (isFull()) {
        return false;
    }

    // make room for the new item
    for (int j = length - 1; j >= i; j--) {
        items[j + 1] = items[j];
    }

    items[i] = item;
    length++;
    return true;
}
```

example for i = 3:

```
                           0   1   2   3   4   5   6   7   8
  items ┌──┬──┐ ──────→  ┌───┬───┬───┬───┬───┬───┬───┬───┬───┐
        │  │──┼─→        │   │   │   │   │   │   │   │   │   │
        └──┴──┘          └───┴───┴───┴───┴───┴───┴───┴───┴───┘
 length │  6  │
        └─────┘
```

Adding an Item to an `ArrayList`

- Adding at position i (shifting items i, i+1, … to the right by one):

```
public boolean addItem(Object item, int i) {
    if (item == null || i < 0 || i > length) {
        throw new IllegalArgumentException();
    } else if (isFull()) {
        return false;
    }

    // make room for the new item
    for (int j = length - 1; j >= i; j--) {
        items[j + 1] = items[j];
    }

    items[i] = item;
    length++;
    return true;
}
```

example for i = 3:

	0	1	2	3	4	5	6	7	8
items   length   **7**
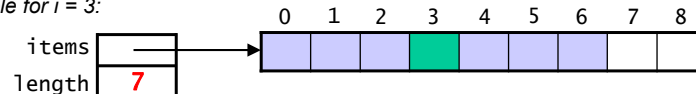
Removing an Item from an `ArrayList`

- Removing item i (shifting items i+1, i+2, … to the left by one):

```
public Object removeItem(int i) {
    if (i < 0 || i >= length) {
        throw new IndexOutOfBoundsException();
    }
    Object removed = items[i];

    // shift items after items[i] to the left
    for (int j = i; j < length - 1; j++) {
        _____;
    }
    items[length - 1] = null;

    length--;
    return removed;
}
```

example for i = 1:

	0	1	2	3	4	5	6	7	8
						null	null	null	null

items   length   **5**

"Ivan"   "Aileen"   "Kelly"   "Melissa"

removed   "Tristan"
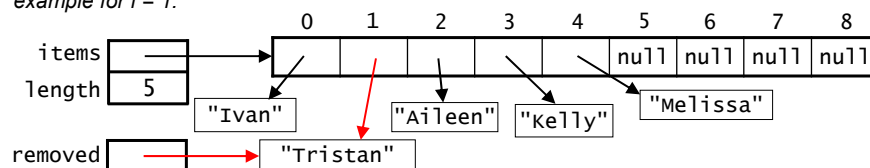
Getting an Item from an `ArrayList`

* Getting item i (without removing it):

```
public Object getItem(int i) {
    if (i < 0 || i >= length) {
        throw new IndexOutOfBoundsException();
    }
    return items[i];
}
```

Implementing a List Using a Linked List

```
public class LLList implements List {
    private Node head;
    private int length;
    ...
}
```


variable of type
LLList

LLList *object*

dummy head node

Node *objects*
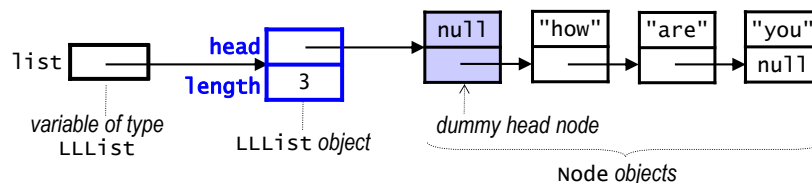
* Differences from the linked lists we used for strings:
  * we "embed" the linked list inside another class
    * users of our `LLList` class won't actually touch the nodes
  * we use non-static methods instead of static ones
    `myList.length()` instead of `length(myList)`
  * we use a special *dummy head node* as the first node

Using a Dummy Head Node
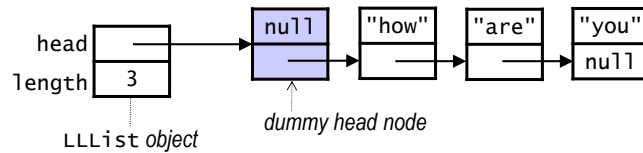
- The dummy head node is always at the front of the linked list.
  - like the other nodes in the linked list, it's of type `Node`
  - it does *not* store an item
  - it does *not* count towards the length of the list

- Using it allows us to avoid special cases when adding and removing nodes from the linked list.

- An empty `LLList` still has a dummy head node:
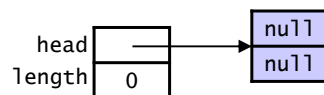
An Inner Class for the Nodes

```
public class LLList implements List {
    private class Node {
        private Object item;
        private Node next;

        private Node(Object i, Node n) {
            item = i;
            next = n;
        }
    }
    ...
}
```

private since only `LLList` *will use it*

item `"hi"`
next

Node *object*

- We make `Node` an *inner class*, defining it within `LLList`.
  - allows the `LLList` methods to directly access `Node`'s private fields, while restricting access from outside `LLList`
  - the compiler creates this class file: `LLList$Node.class`

- For simplicity, our diagrams may show the items inside the nodes.

`"hi"`   *instead of*   `"hi"`
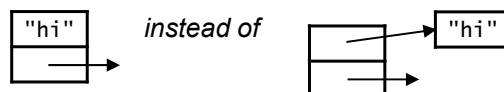
Other Details of Our `LLList` Class

```
public class LLList implements List {
    private class Node {
        // see previous slide
    }

    private Node head;
    private int length;

    public LLList() {
        head = new Node(null, null);
        length = 0;
    }

    public boolean isFull() {
        return false;
    }
    ...
}
```

* Unlike `ArrayList`, there's no need to preallocate space for the items. The constructor simply creates the dummy head node.

* The linked list can grow indefinitely, so the list is never full!

Getting a Node

* Private helper method for getting node i
  * to get the dummy head node, use i = -1

```
private Node getNode(int i) {
    // private method, so we assume i is valid!

    Node trav = _____;
    int travIndex = -1;

    while ( _____ ) {
        travIndex++;

        _____;
    }

    return trav;
}
```

example for i = 1:
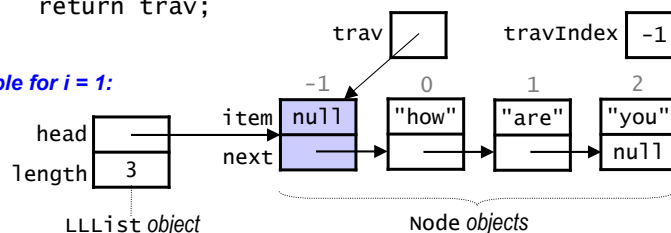
Getting an Item

```
public Object getItem(int i) {
    if (i < 0 || i >= length) {
        throw new IndexOutOfBoundsException();
    }

    Node n = getNode(i);
    return _____;
}
```

example for i = 1:
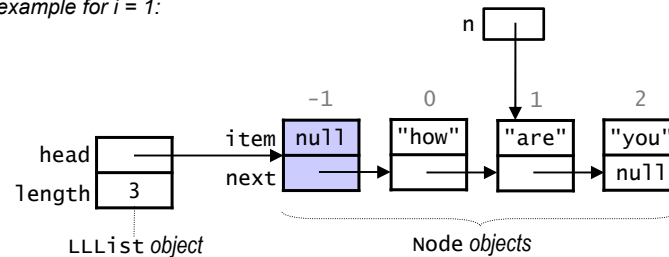
Adding an Item to an `LLList`

```
public boolean addItem(Object item, int i) {
    if (item == null || i < 0 || i > length) {
        throw new IllegalArgumentException();
    }
    Node newNode = new Node(item, null);
    Node prevNode = getNode(i - 1);
    newNode.next = prevNode.next;
    prevNode.next = newNode;

    length++;
    return true;
}
```

* This works even when adding at the front of the list (i = 0):
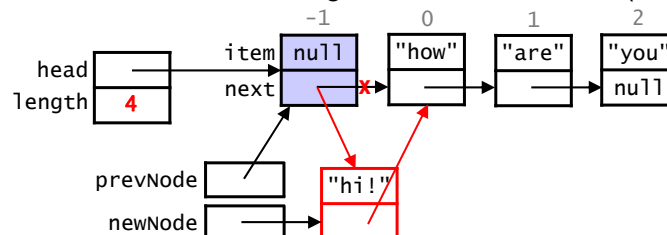
addItem() Without a Dummy Head Node

```
public boolean addItem(Object item, int i) {
    if (item == null || i < 0 || i > length) {
        throw new IllegalArgumentException();
    }
    Node newNode = new Node(item, null);

    if (i == 0) {                        // case 1: add to front
        newNode.next = head;
        head = newNode;
    } else {                             // case 2: i > 0
        Node prevNode = getNode(i - 1);
        newNode.next = prevNode.next;
        prevNode.next = newNode;
    }

    length++;
    return true;
}
```

(the gray code shows what we would need to add if we didn't have a dummy head node)

Removing an Item from an LLList

```
public Object removeItem(int i) {
    if (i < 0 || i >= length) {
        throw new IndexOutOfBoundsException();
    }
    Node prevNode = getNode(i - 1);
    Object removed = prevNode.next.item;
                            // what line goes here?

    length--;
    return removed;
}
```

* This works even when removing the first item (i = 0):
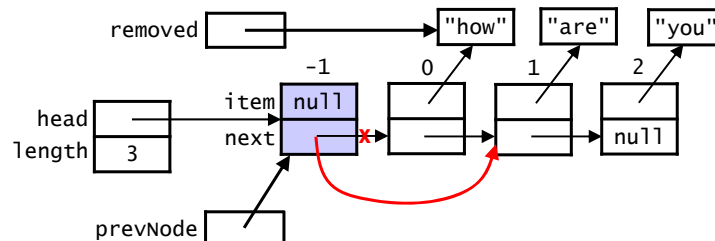
Efficiency of the List ADT Implementations

n = number of items in the list

	ArrayList	LLList
getItem()	only one case:	*best:* *worst:* *average:*
addItem()	*best:* *worst:* *average:*	*best:* *worst:* *average:*

Example of Using a Reference to the Last Node

mylist.addItem("you", 99)

- before the call is made:

  LLList *object*
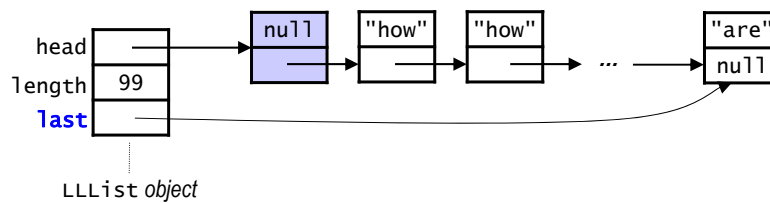
- use last to add the new item's node to the end of the linked list:
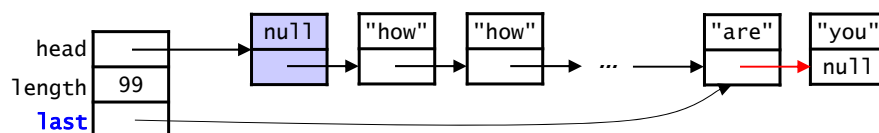
Example of Using a Reference to the Last Node (cont.)

mylist.addItem("you", 99)

- after the call is made:

```
head                    null    "how"    "how"           "are"    "you"
length  100                                                        null

last

      LLList object
```
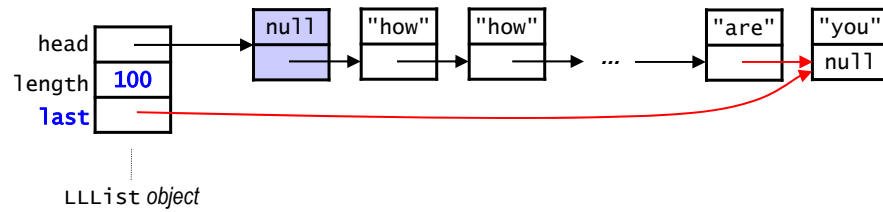
Efficiency of the List ADT Implementations (cont.)

n = number of items in the list

	ArrayList	LLList
removeItem()	*best:*	*best:*
	worst:	*worst:*
	average:	*average:*
space efficiency		

A Reference to the Second-to-Last Node Doesn't Help

`mylist.removeItem(99)`

- before the call is made:

LLList *object*
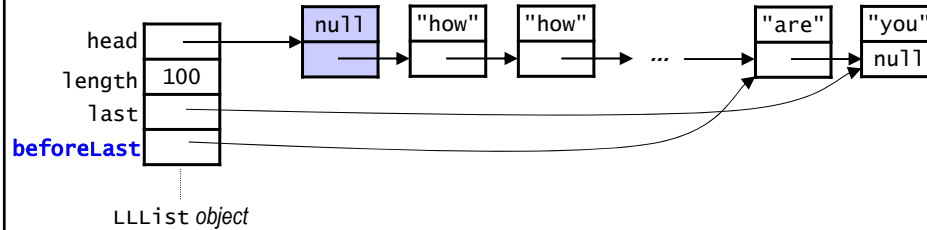
- we can use `beforeLast` to remove the last node and update `last`:
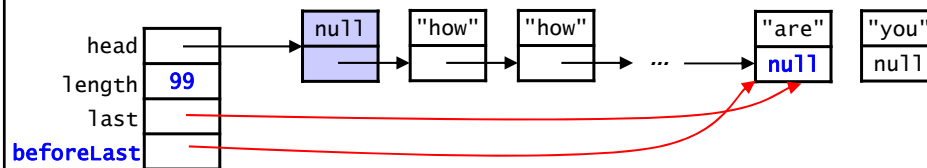
A Reference to the Second-to-Last Node Doesn't Help

`mylist.removeItem(99)`

- but in order to update `beforeLast`, we need to walk down the linked list!
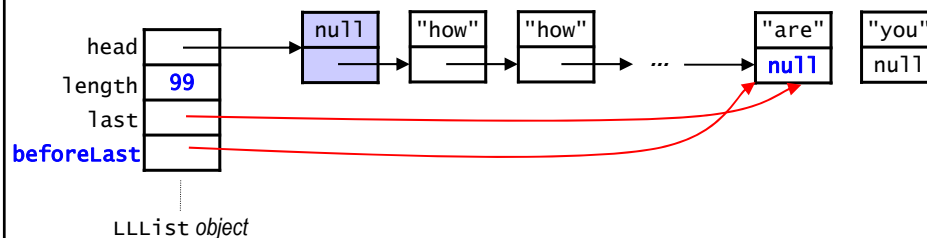
LLList *object*

Recall: Another Example of Polymorphism

* An interface can be used as the type of a variable:
  ```
  List myList;
  ```

* We can then assign an object of any class that implements the interface:
  ```
  List l1 = new ArrayList(20);
  List l2 = new LLList();
  ```

Counting the Number of Occurrences of an Item

```
public class MyClass {
    public static int numOccur(List l, Object item) {
        int numOccur = 0;
        for (int i = 0; i < l.length(); i++) {
            Object itemAt = l.getItem(i);
            if (itemAt.equals(item)) {
                numOccur++;
            }
        }
        return numOccur;
    } ...
```

* This method works fine if we pass in an `ArrayList` object.
  * time efficiency (as a function of the length, n) = ?

* However, it's *not* efficient if we pass in an `LLList`.
  * each call to `getItem()` calls `getNode()`
  * to access item 0, `getNode()` accesses 2 nodes (dummy + node 0)
  * to access item 1, `getNode()` accesses 3 nodes
  * to access item i, `getNode()` accesses i+2 nodes
  * 2 + 3 + ... + (n+1) = ?

Solution: Provide an Iterator

```java
public class MyClass {
    public static int numOccur(List l, Object item) {
        int numOccur = 0;
        ListIterator iter = l.iterator();
        while (iter.hasNext()) {
            Object itemAt = iter.next();
            if (itemAt.equals(item)) {
                numOccur++;
            }
        }
        return numOccur;
    } ...
```

* We add an `iterator()` method to the `List` interface.
  * it returns a separate *iterator object* that can efficiently iterate over the items in the list

* The iterator has two key methods:
  * `hasNext()`: tells us if there are items we haven't seen yet
  * `next()`: returns the next item *and* advances the iterator

An Interface for List Iterators

* Here again, the interface only includes the method headers:
  ```java
  public interface ListIterator { // in ListIterator.java
      boolean hasNext();
      Object next();
  }
  ```

* We can then implement this interface for each type of list:
  * `LLListIterator` for an iterator that works with `LLLists`
  * `ArrayListIterator` for an iterator for `ArrayLists`

* We use the interfaces when declaring variables in client code:
  ```java
  public class MyClass {
      public static int numOccur(List l, Object item) {
          int numOccur = 0;
          ListIterator iter = l.iterator();
          ...
  ```
  * doing so allows the code to work for any type of list!

Using an Inner Class for the Iterator

```
public class LLList {
    private Node head;
    private int length;

    private class LLListIterator implements ListIterator {
        private Node nextNode;  // points to node with the next item

        public LLListIterator() {
            nextNode = head.next;  // skip over dummy head node
        }
        ...
    }
    public ListIterator iterator() {
        return new LLListIterator();
    }
    ...
```

* Using an inner class gives the iterator access to the list's internals.

* The `iterator()` method is an `LLList` method.
    * it creates an instance of the inner class and returns it
    * its return type is the interface type
        * so it will work in the context of client code

Full `LLListIterator` Implementation

```
private class LLListIterator implements ListIterator {
    private Node nextNode;      // points to node with the next item

    public LLListIterator() {
        nextNode = head.next;  // skip over the dummy head node
    }
    public boolean hasNext() {
        return (nextNode != null);
    }
    public Object next() {
        // throw an exception if nextNode is null

        Object item = _____;

        nextNode = _____;

        return item;
    }
}
```
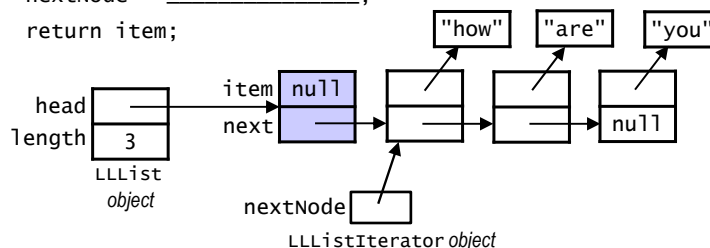
The Stack and Queue ADTs

Computer Science 112
Boston University

Stack ADT

- A stack is a sequence in which:
  - items can be added and removed only at one end (the *top*)
  - you can only access the item that is currently at the top

- Operations:
  - push: add an item to the top of the stack
  - pop: remove the item at the top of the stack
  - peek: get the item at the top of the stack, but don't remove it
  - isEmpty: test if the stack is empty
  - isFull: test if the stack is full

- Example: a stack of integers

start:	*push 8:*	*pop:*	*pop:*	*push 3:*
	8			
15	15	15		3
7	7	7	7	7
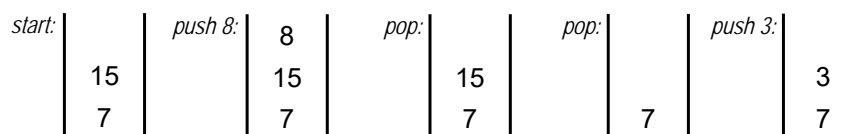
A Stack Interface: First Version

```
public interface Stack {
    boolean push(Object item);
    Object pop();
    Object peek();
    boolean isEmpty();
    boolean isFull();
}
```

* push() returns false if the stack is full, and true otherwise.

* pop() and peek() take no arguments, because we know that we always access the item at the top of the stack.
  * return null if the stack is empty.

* The interface provides no way to access/insert/delete an item at an arbitrary position.
  * encapsulation allows us to ensure that our stacks are only manipulated in appropriate ways

Implementing a Stack Using an Array: First Version

```
public class ArrayStack implements Stack {
    private Object[] items;
    private int top;     // index of the top item

    public ArrayStack(int maxSize) {
        // code to check for invalid maxSize goes here...
        items = new Object[maxSize];
        top = -1;
    }
    ...
```

* Example: the stack
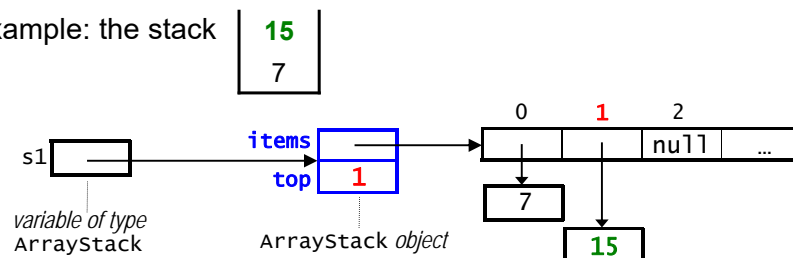
* Items are added from left to right (top item = the rightmost one).
  * *why does this approach make sense?*

Collection Classes and Data Types

```
public class ArrayStack implements Stack {
    private Object[] items;
    private int top;     // index of the top item
     ...
}
```
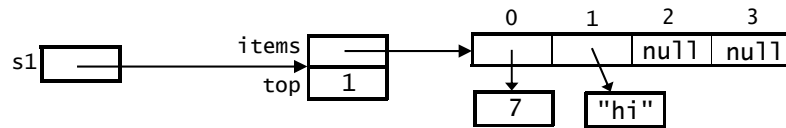

* So far, our collections have allowed us to add objects of any type.

```
ArrayStack s1 = new ArrayStack(4);
s1.push(7);     // 7 is turned into an Integer object for 7
s1.push("hi");
String item = s1.pop();          // won't compile
String item = (String)s1.pop();   // need a type cast
```

* We'd like to be able to limit a given collection to one type.

```
ArrayStack<String> s2 = new ArrayStack<String>(10);
s2.push(7);                 // won't compile
s2.push("hello");
String item = s2.pop();    // no cast needed!
```

Limiting a Stack to Objects of a Given Type

* We can do this by using a *generic* interface and class.

* Here's a generic version of our Stack interface:

```
public interface Stack<T> {
    boolean push(T item);
    T pop();
    T peek();
    boolean isEmpty();
    boolean isFull();
}
```

* It includes a *type variable* **T** in its header and body.
  * used as a placeholder for the actual type of the items

A Generic `ArrayStack` Class

```
public class ArrayStack<T> implements Stack<T> {
    private T[] items;
    private int top;     // index of the top item
    …
    public boolean push(T item) {
        …
    }
    …
}
```

* Once again, a type variable `T` is used as a placeholder for the actual type of the items.

* When we create an `ArrayStack`, we specify the type of items that we intend to store in the stack:

```
ArrayStack<String> s1 = new ArrayStack<String>(10);
ArrayStack<Integer> s2 = new ArrayStack<Integer>(25);
```

* We can still allow for a mixed-type collection:

```
ArrayStack<Object> s3 = new ArrayStack<Object>(20);
```

Using a Generic Class

```
public class ArrayStack<String> {
    private String[] items;
    private int top;

    ...
    public boolean push(String item) {
        ...
```

```
ArrayStack<String> s1 =
    new ArrayStack<String>(10);
```

```
public class ArrayStack<T> ... {
    private T[] items;
    private int top;

    ...
    public boolean push(T item) {
        ...
```

```
ArrayStack<Integer> s2 =
    new ArrayStack<Integer>(25);
```

```
public class ArrayStack<Integer> {
    private Integer[] items;
    private int top;

    ...
    public boolean push(Integer item) {
        ...
```

`ArrayStack` Constructor

- Java doesn't allow you to create an object or array using a type variable. Thus, we *cannot* do this:

```
public ArrayStack(int maxSize) {
    // code to check for invalid maxSize goes here...
    items = new T[maxSize];   // not allowed
    top = -1;
}
```

- Instead, we do this:

```
public ArrayStack(int maxSize) {
    // code to check for invalid maxSize goes here...
    items = (T[])new Object[maxSize];
    top = -1;
}
```

- The cast generates a compile-time warning, but we'll ignore it.

- Java's built-in `ArrayList` class takes this same approach.

Testing if an `ArrayStack` is Empty or Full

- Empty stack:
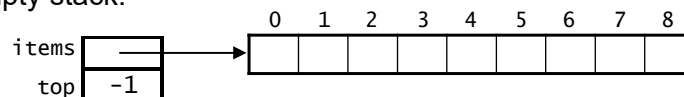

```
public boolean isEmpty() {
    return (top == -1);
}
```

- Full stack:
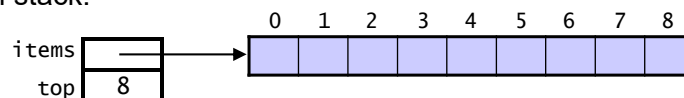

```
public boolean isFull() {
    return (top == items.length - 1);
}
```

Pushing an Item onto an `ArrayStack`

```
        0   1   2   3   4   5   6   7   8
items ___ _____
top  4  [ ][ ][ ][ ][ ][ ][ ][ ][ ]
```
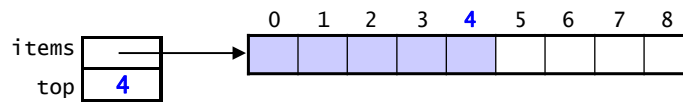
```java
public boolean push(T item) {
    // code to check for a null item goes here
    if (isFull()) {
        return false;
    }
    top++;
    items[top] = item;
    return true;
}
```

`ArrayStack pop()` and `peek()`

```
        0   1   2   3    4     5     6     7
items ___ _____
top  3  [ ][ ][ ][ ]null null null null
         |  |  |  |
        10  5  9  13
                      removed ___
```
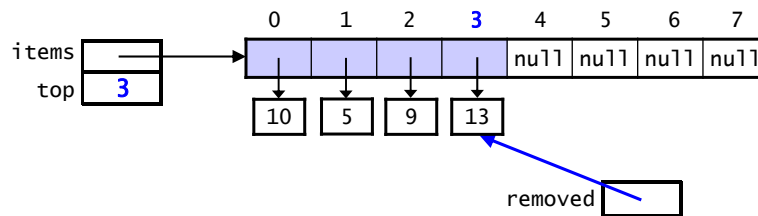
```java
public T pop() {
    if (isEmpty()) {
        return null;
    }

    _____ removed = items[top];
    items[top] = null;
    top--;
    return removed;
}
```

- peek just returns `items[top]` without decrementing `top`.

Implementing a Generic Stack Using a Linked List

```
public class LLStack<T> implements Stack<T> {
    private Node top;    // top of the stack
    …
}
```

* Example: the stack
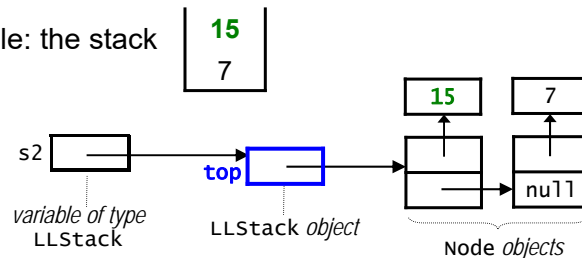
* Things worth noting:
  * our `LLStack` class needs only a single field:
    a reference to the first node, which holds the top item
  * top item = leftmost item (vs. rightmost item in `ArrayStack`)
  * we don't need a dummy node
    * only one case: always insert/delete at the front of the list!

Other Details of Our `LLStack` Class

```
public class LLStack<T> implements Stack<T> {
    private class Node {
        private T item;
        private Node next;
        ...
    }

    private Node top;

    public LLStack() {
        top = null;
    }
    public boolean isEmpty() {
        return (top == null);
    }
    public boolean isFull() {
        return false;
    }
}
```

* The inner `Node` class uses the type parameter `T` for the item.

* We don't need to preallocate any memory for the items.

* The stack is never full!

LLStack push()
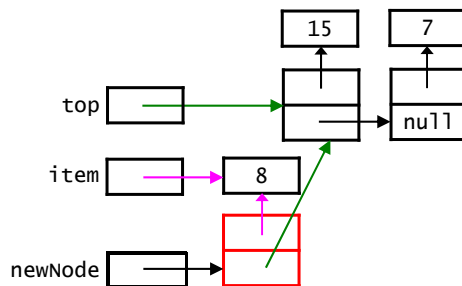

```
public boolean push(T item) {
    // code to check for a null item goes here
    Node newNode = new Node(item, top);
    top = newNode;
    return true;
}
```
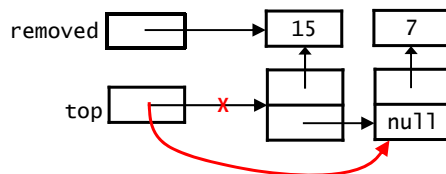
LLStack push()
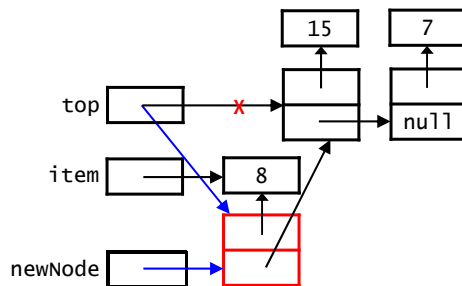

```
public boolean push(T item) {
    // code to check for a null item goes here
    Node newNode = new Node(item, top);
    top = newNode;
    return true;
}
```

LLStack pop() and peek()

```
removed [ | ]────▶[ 15 ]   [ 7 ]
                       ▲      ▲
                      [ ]    [ ]
top [ ]     x         [ |──]▶[null]
```

```
public T pop() {
    if (isEmpty()) {
        return null;
    }
    T removed = _____;

    _____;
    return removed;
}
public T peek() {
    if (isEmpty()) {
        return null;
    }
    return top.item;
}
```

Efficiency of the Stack Implementations

	ArrayStack	LLStack
push()	O(1)	O(1)
pop()	O(1)	O(1)
peek()	O(1)	O(1)
space efficiency	O(m) where m is the *anticipated* maximum number of items	O(n) where n is the number of items currently on the stack

Applications of Stacks

- Converting a recursive algorithm to an iterative one
  - use a stack to emulate the runtime stack

- Making sure that delimiters (parens, brackets, etc.) are balanced:
  - push open (i.e., left) delimiters onto a stack
  - when you encounter a close (i.e., right) delimiter, pop an item off the stack and see if it matches
  - example:
    ```
    5 * [3 + {(5 + 16 – 2)]
    ```

push [	*push* {	*push* (		*)*, so pop. get (, which matches		*]*, so pop. get {, which doesn't match	
			(		{		
	{	{	{		{		
[	[	[	[		[		[
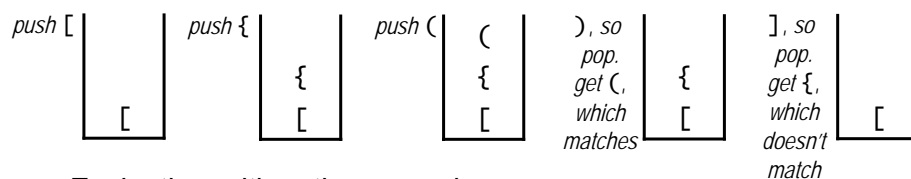
- Evaluating arithmetic expressions

Queue ADT
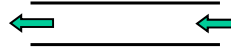
- A queue is a sequence in which:
  - items are added at the rear and removed from the front
    - first in, first out (FIFO) (vs. a stack, which is last in, first out)
  - you can only access the item that is currently at the front

- Operations:
  - insert: add an item at the rear of the queue
  - remove: remove the item at the front of the queue
  - peek: get the item at the front of the queue, but don't remove it
  - isEmpty: test if the queue is empty
  - isFull: test if the queue is full

- Example: a queue of integers
  - *start:* 12 8
  - *insert 5:* 12 8 5
  - *remove:* 8 5

Our Generic Queue Interface

```
public interface Queue<T> {
    boolean insert(T item);
    T remove();
    T peek();
    boolean isEmpty();
    boolean isFull();
}
```

- insert() returns false if the queue is full, and true otherwise.

- remove() and peek() take no arguments, because
  we always access the item at the front of the queue.
    - return null if the queue is empty.

- Here again, we will use encapsulation to ensure that the
  data structure is manipulated only in valid ways.

Implementing a Queue Using an Array

```
public class ArrayQueue<T> implements Queue<T> {
    private T[] items;
    private int front;
    private int rear;
    private int numItems;

    ...
}
```

- Example:
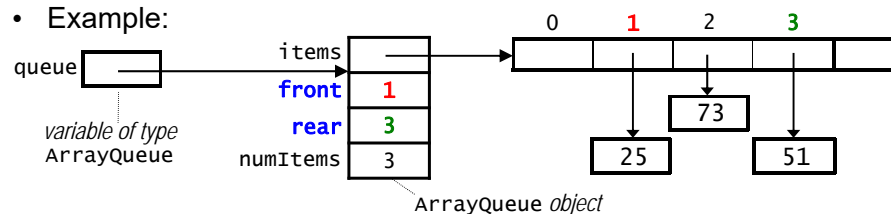
- We maintain two indices:
    - front: the index of the item at the front of the queue
    - rear: the index of the item at the rear of the queue

Avoiding the Need to Shift Items

* Problem: what do we do when we reach the end of the array?

  example: a queue of integers:

  front   rear

  | 54 | 4 | 21 | 17 | 89 | 65 | | |

  the same queue after removing two items and inserting two:

  front   rear

  | | | 21 | 17 | 89 | 65 | 43 | 81 |

  we have room for more items, but shifting to make room is inefficient

* Solution: maintain a *circular queue*. When we reach the end of the array, we wrap around to the beginning.

  insert 5: wrap around!
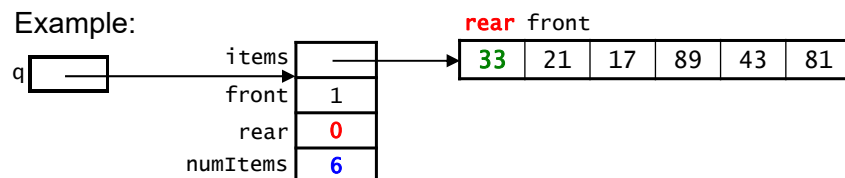
  rear   front

  | 5 | | 21 | 17 | 89 | 65 | 43 | 81 |

Maintaining a Circular Queue

* We use the mod operator (%) when updating front or rear:

  ```
  front = (front + 1) % items.length;
  rear = (rear + 1) % items.length;
  ```

* Example:

  ```
         items
  q                      front        rear
  front    1        | | 21 | 17 | 89 | 43 | |
  rear     4
  numItems 4
  ```

Maintaining a Circular Queue

- We use the mod operator (%) when updating `front` or `rear`:

      front = (front + 1) % items.length;
      rear = (rear + 1) % items.length;

- Example:

```
                          front                    rear
            items   ┌──────┐      ┌────┬────┬────┬────┬────┐
q  ┌──────┐         │  ──  │─────▶│ 21 │ 17 │ 89 │ 43 │ 81 │
   └──────┘         └──────┘      └────┴────┴────┴────┴────┘
            front   │  1   │
            rear    │  5   │
         numItems   │  5   │
                    └──────┘
```
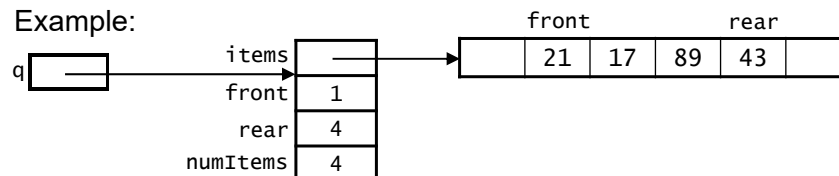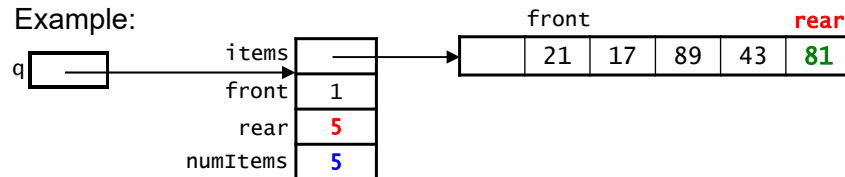
- q.insert(**81**):   *// rear is <u>not</u> at end of array*
  - rear = (rear + 1) % items.length;
          = (4 + 1) %   6
          =   5 %   6   = **5** *(% has no effect)*

Maintaining a Circular Queue

- We use the mod operator (%) when updating `front` or `rear`:

      front = (front + 1) % items.length;
      rear = (rear + 1) % items.length;

- Example:

```
                      rear front
            items   ┌──────┐      ┌────┬────┬────┬────┬────┬────┐
q  ┌──────┐         │  ──  │─────▶│ 33 │ 21 │ 17 │ 89 │ 43 │ 81 │
   └──────┘         └──────┘      └────┴────┴────┴────┴────┴────┘
            front   │  1   │
            rear    │  0   │
         numItems   │  6   │
                    └──────┘
```

- q.insert(81):   *// rear is <u>not</u> at end of array*
  - rear = (rear + 1) % items.length;
          = (4 + 1) %   6
          =   5 %   6   = 5 *(% has no effect)*
- q.insert(**33**):   *// rear is at end of array*
  - rear = (rear + 1) % items.length;
          = (5 + 1) %   6
          =   6 %   6   = **0**   *wrap around!*

Inserting an Item in an `ArrayQueue`

* We increment `rear` before adding the item:

before:

front   rear

after:

front   rear
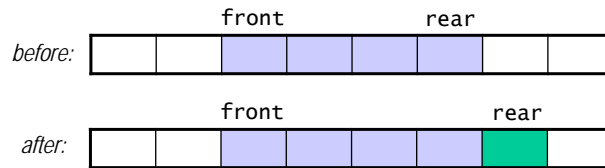
```
public boolean insert(T item) {
    // code to check for a null item goes here
    if (isFull()) {
        return false;
    }
    rear = (rear + 1) % items.length;
    items[rear] = item;
    numItems++;
    return true;
}
```
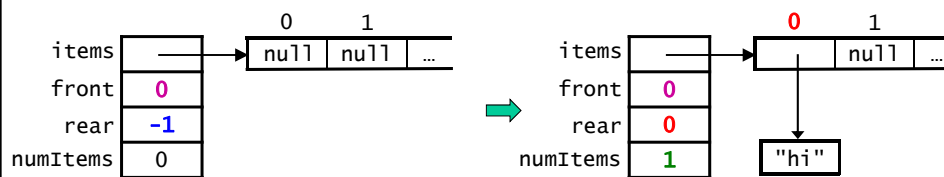
ArrayQueue remove()

before:

front   rear

| 10 | 5 | 9 | 13 |

after:

front   rear

| null |

removed

| 10 | 5 | 9 | 13 |
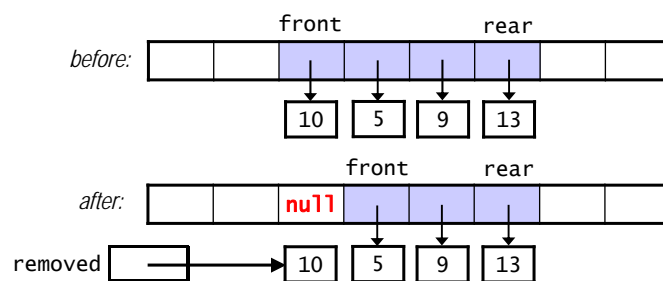
```
public T remove() {
    if (isEmpty()) {
        return null;
    }
    T removed = _____;


    numItems--;
    return removed;
}
```

Constructor

```
public ArrayQueue(int maxSize) {
    // code to check for an invalid maxSize goes here...
    items = (T[])new Object[maxSize];
    front = 0;
    rear = -1;
    numItems = 0;
}
```

* When we insert the first item in a newly created `ArrayQueue`,
  we want it to go in position 0. Thus, we need to:
  * start `rear` at **-1**, since then it will be incremented to **0**
    and used to perform the insertion
  * start `front` at **0**, since it is not changed by the insertion

Testing if an `ArrayQueue` is Empty or Full

* In both empty and full queues, `rear` is one "behind" `front`:

initial configuration:

after two insertions and two removals:

after 7 more insertions: | 5 | 36 | 21 | 17 | 89 | 65 | 43 |
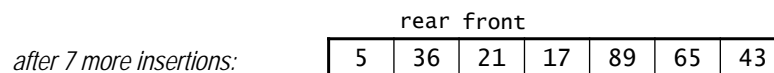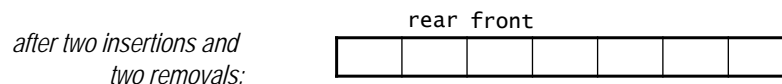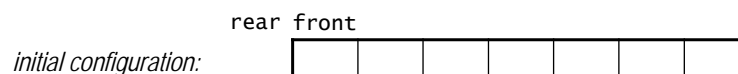
* This is why we maintain `numItems`!

```
public boolean isEmpty() {
    return (numItems == 0);
}

public boolean isFull() {
    return (numItems == items.length);
}
```

Implementing a Queue Using a Linked List

```
public class LLQueue<T> implements Queue<T> {
    private Node front;     // front of the queue
    private Node rear;      // rear of the queue
    …
}
```

* Example:
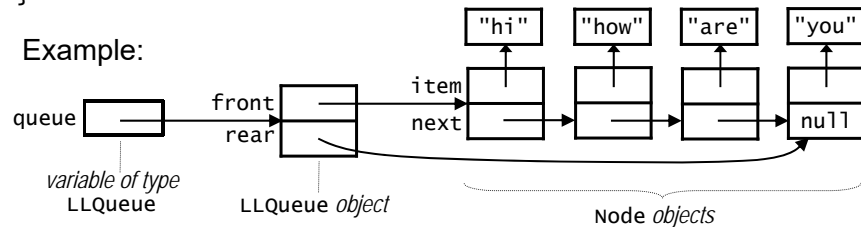
* In a linked list, we can efficiently:
  * remove the item at the front
  * add an item to the rear (if we have a ref. to the last node)

* Thus, this implementation is simpler than the array-based one!

Other Details of Our `LLQueue` Class

```
public class LLQueue<T> implements Queue<T> {
    private class Node {
        private T item;
        private Node next;
        ...
    }

    private Node front;
    private Node rear;

    public LLQueue() {
        front = null;
        rear = null;
    }
    public boolean isEmpty() {
        return (front == null);
    }
    public boolean isFull() {
        return false;
    }
    …
}
```

Inserting an Item in an Empty `LLQueue`

front `null`
rear `null`

item ☐──→ `"now"`
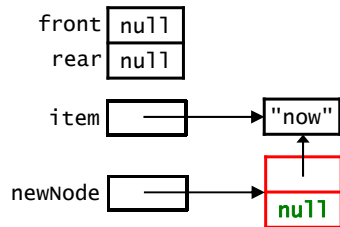
newNode ☐──→ ☐ `null`

The `next` field in the `newNode` will be `null` regardless of whether the queue is empty. Why?

```
public boolean insert(T item) {
    // code to check for a null item goes here
    Node newNode = new Node(item, null);
    if (isEmpty()) {
        front = newNode;
        rear = newNode;
    } else {
        // we'll add this later!

    }
    return true;
}
```

Inserting an Item in a Non-Empty `LLQueue`

`"hi"` `"how"` `"are"` `"you"`

front ☐──→ ☐──→ ☐──→ ☐──→ ☐

rear ☐ ⤵ X `"now"`

item ☐──→ ☐ `null`

newNode ☐──→

```
public boolean insert(T item) {
    // code to check for a null item goes here
    Node newNode = new Node(item, null);
    if (isEmpty()) {
        front = newNode;
        rear = newNode;
    } else {


    }
    return true;
}
```

A. `rear = newNode;`
   `rear.next = newNode;`

B. `rear.next = newNode;`
   `rear = newNode;`

C. either A or B

D. neither A nor B
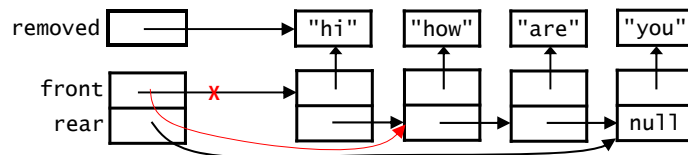
Removing from an `LLQueue` with One Item
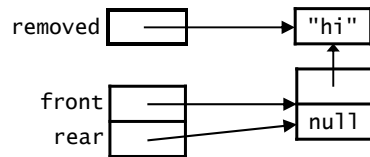

```
public T remove() {
    if (isEmpty()) {
        return null;
    }

    T removed = _____;
    if (front == rear) {      // removing the only item
        front = null;
        rear = null;
    } else {
        // we'll add this later
    }

    return removed;
}
```

Removing from an `LLQueue` with Two or More Items


```
public T remove() {
    if (isEmpty()) {
        return null;
    }

    T removed = _____;
    if (front == rear) {      // removing the only item
        front = null;
        rear = null;
    } else {

    }

    return removed;
}
```

Efficiency of the Queue Implementations

	ArrayQueue	LLQueue
insert()	$O(1)$	$O(1)$
remove()	$O(1)$	$O(1)$
peek()	$O(1)$	$O(1)$
space efficiency	$O(m)$ where m is the *anticipated* maximum number of items	$O(n)$ where n is the number of items currently in the queue

Applications of Queues

* first-in first-out (FIFO) inventory control

* OS scheduling: processes, print jobs, packets, etc.

* simulations of banks, supermarkets, airports, etc.

Binary Trees

Computer Science 112
Boston University

Motivation: Implementing a Dictionary

- A *data dictionary* is a collection of data with two main operations:
  - *search* for an item (and possibly delete it)
  - *insert* a new item

- If we use a *sorted* list to implement it, efficiency = $O(n)$.

data structure	searching for an item	inserting an item
a list implemented using an array	$O(\log n)$ using binary search	$O(n)$ because we need to shift items over
a list implemented using a linked list	$O(n)$ using linear search (binary search in a linked list is $O(n \log n)$)	$O(n)$ ($O(1)$ to do the actual insertion, but $O(n)$ to find where it belongs)

- In the next few lectures, we'll look at how we can use a *tree* for a data dictionary, and we'll try to get better efficiency.

- We'll also look at other applications of trees.

What Is a Tree?
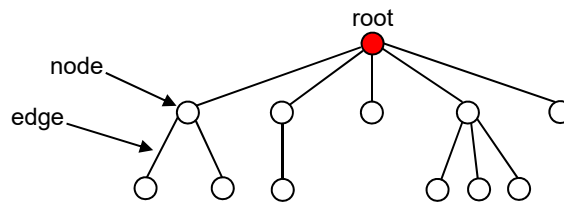
- A tree consists of:
  - a set of *nodes*
  - a set of *edges*, each of which connects a pair of nodes

- Each node may have one or more *data items.*
  - each data item consists of one or more fields
  - *key field* = the field used when searching for a data item
  - data items with the same key are referred to as *duplicates*

- The node at the "top" of the tree is called the *root* of the tree.

Relationships Between Nodes
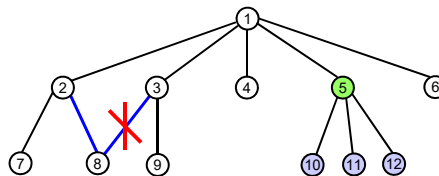
- If a node N is connected to nodes directly below it in the tree:
  - N is referred to as their *parent*
  - they are referred to as its *children*.
    - example: node 5 is the parent of nodes 10, 11, and 12

- Each node is the child of *at most one* parent.

- Nodes with the same parent are *siblings.*

## Relationships Between Nodes (cont.)
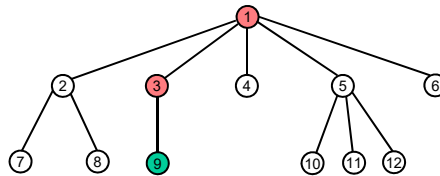
- A node's *ancestors* are its parent, its parent's parent, etc.
  - example: node 9's ancestors are 3 and 1

- A node's *descendants* are its children, their children, etc.
 - example: node 1's descendants are *all* of the other nodes

Types of Nodes
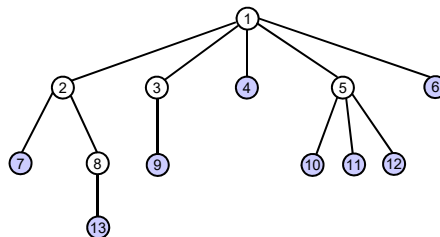
- A *leaf node* is a node without children.

- An *interior node* is a node with one or more children.

A Tree is a Recursive Data Structure
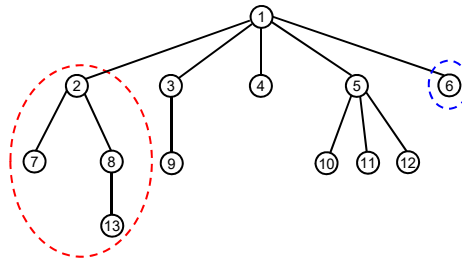
- Each node in the tree is the root of a smaller tree!
  - refer to such trees as *subtrees* to distinguish them from the tree as a whole
  - example: node 2 is the root of the subtree circled above
  - example: node 6 is the root of a subtree with only one node

- We'll see that tree algorithms often lend themselves to recursive implementations.
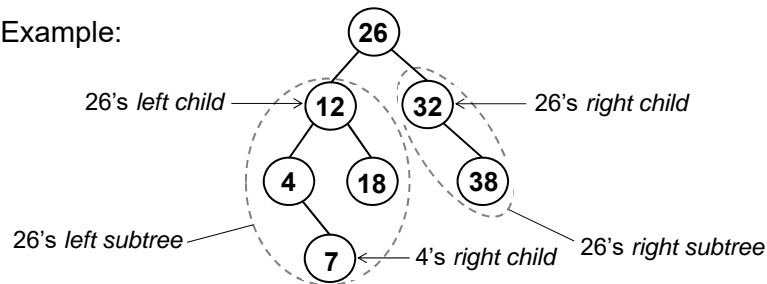
Path, Depth, Level, and Height
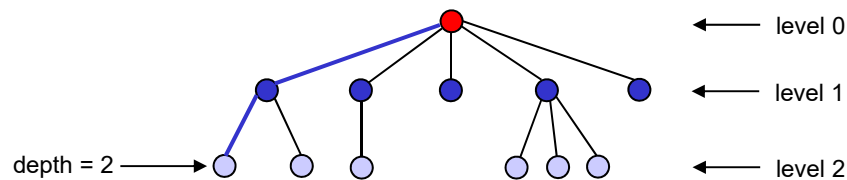
- There is exactly one *path* (one sequence of edges) connecting each node to the root.

- *depth* of a node = # of edges on the path from it to the root

- Nodes with the same depth form a *level* of the tree.

- The *height* of a tree is the maximum depth of its nodes.
 - example: the tree above has a height of 2

Binary Trees

- In a *binary tree*, nodes have *at most two* children.
  - distinguish between them using the direction *left* or *right*

- Example:

26's *left child* → 12
26's *right child* → 32
26's *left subtree*
4's *right child*
26's *right subtree*

- Recursive definition: a binary tree is either:
  1) empty, or
  2) a node (the root of the tree) that has:
     - one or more pieces of data (the key, and possibly others)
     - a *left subtree*, which is itself a binary tree
     - a *right subtree*, which is itself a binary tree

## Which of the following is/are <u>not</u> true?
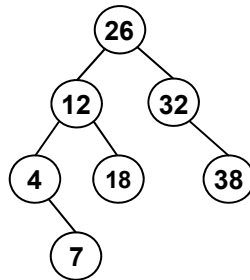
A. This tree has a height of 4.

B. There are 3 leaf nodes.

C. The 38 node is the right child of the 32 node.

D. The 12 node has 3 children.

E. more than one of the above are <u>not</u> true (which ones?)

Representing a Binary Tree Using Linked Nodes

```java
public class LinkedTree {
    private class Node {
        private int key;      // limit ourselves to int keys
        private LLList data;  // list of data for that key
        private Node left;    // reference to left child
        private Node right;   // reference to right child
        …
    }

    private Node root;
    …
}
```
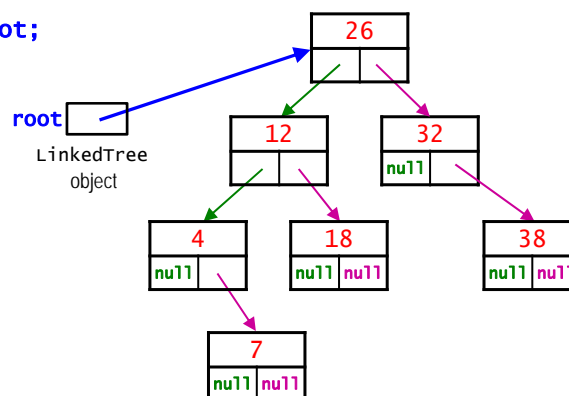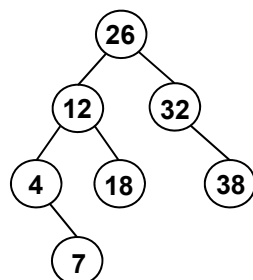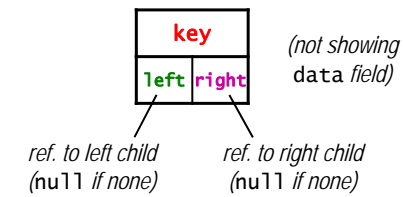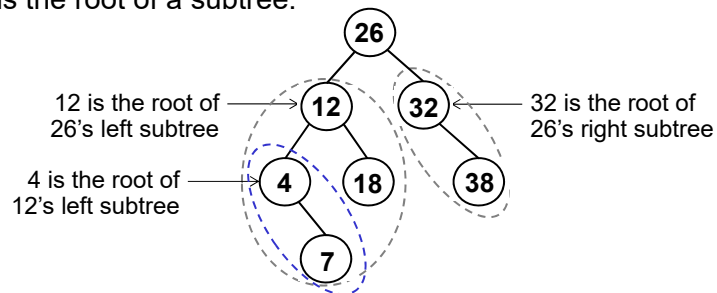
Traversing a Binary Tree

* Traversing a tree involves *visiting* all of the nodes in the tree.
  * visiting a node = processing its data in some way
    * example: print the key

* We'll look at four types of traversals.
  * each visits the nodes in a different order

* To understand traversals, it helps to remember that every node is the root of a subtree.

12 is the root of 26's left subtree

32 is the root of 26's right subtree

4 is the root of 12's left subtree

Nodes: 26, 12, 32, 4, 18, 38, 7

1: Preorder Traversal

* preorder traversal of the tree whose root is N:
  1) visit the root, N
  2) recursively perform a preorder traversal of N's left subtree
  3) recursively perform a preorder traversal of N's right subtree

Tree: 7 (root), left child 9, right child 5; 9 has children 8 and 6; 5 has child 2; 8 has child 4
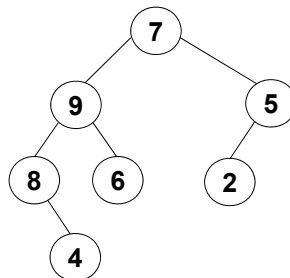
* *pre*order because a node is visited *before* its subtrees

* The root of the tree as a whole is visited first.

Implementing Preorder Traversal

```java
public class LinkedTree {
    ...
    private Node root;

    public void preorderPrint() {
        if (root != null) {
            preorderPrintTree(root);
        }
        System.out.println();
    }
    private static void preorderPrintTree(Node root) {
        System.out.print(root.key + " ");
        if (root.left != null) {
            preorderPrintTree(root.left);
        }
        if (root.right != null) {
            preorderPrintTree(root.right);
        }
    }
}
```
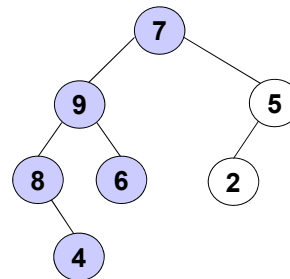
Not always the same as the root of the entire tree.

* `preorderPrintTree()` is a static, recursive method that takes the root of the tree/subtree that you want to print.
* `preorderPrint()` is a non-static "wrapper" method that makes the initial call. It passes in the root of the entire tree.

Tracing Preorder Traversal

```java
void preorderPrintTree(Node root) {
    System.out.print(root.key + " ");
    if (root.left != null) {
        preorderPrintTree(root.left);
    }
    if (root.right != null) {
        preorderPrintTree(root.right);
    }
}
```


base case, since neither recursive call is made!

we go back up the tree by returning!

order in which nodes are visited:

			root: ④ **print 4**			
	root: ⑧ **print 8**	root: ⑧	root: ⑧	root: ⑧		root: ⑥ **print 6**
	root: ⑨ **print 9**	root: ⑨	root: ⑨	root: ⑨	root: ⑨	root: ⑨
root: ⑦ **print 7**	root: ⑦	root: ⑦	root: ⑦	root: ⑦	root: ⑦	root: ⑦

time ⟶

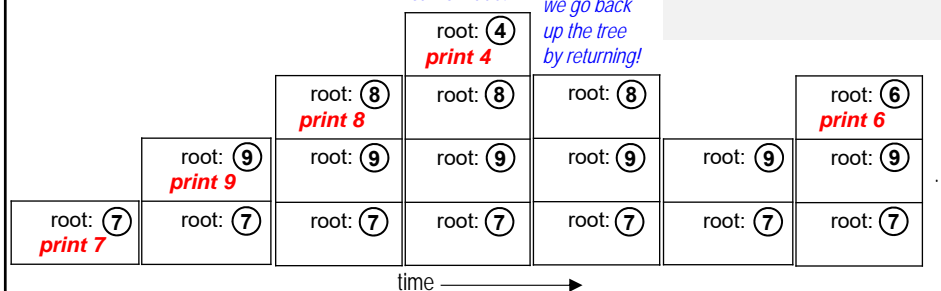## Using Recursion for Traversals

```
void preorderPrintTree(Node root) {
    System.out.print(root.key + " ");
    if (root.left != null) {
        preorderPrintTree(root.left);
    }
    if (root.right != null) {
        preorderPrintTree(root.right);
    }
}
```
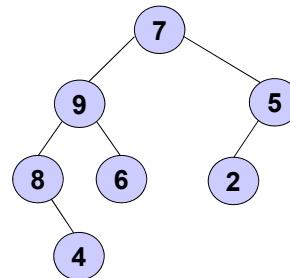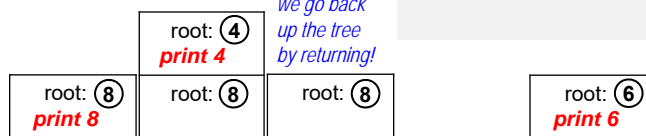
base case, since neither recursive call is made!

we go back up the tree by returning!

order in which nodes are visited:

root: ④
print 4

root: ⑧
print 8

root: ⑧

root: ⑧

root: ⑥
print 6

...

- Using recursion allows us to easily go back up the tree.

- Using a loop would be harder. Why?

2: Postorder Traversal

- postorder traversal of the tree whose root is N:
  1) recursively perform a postorder traversal of N's left subtree
  2) recursively perform a postorder traversal of N's right subtree
  3) visit the root, N
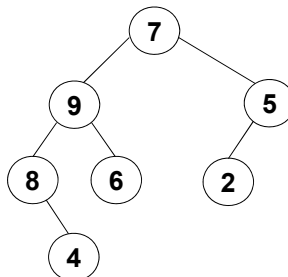
- *post*order because a node is visited *after* its subtrees

- The root of the tree as a whole is visited last.

Implementing Postorder Traversal

```java
public class LinkedTree {
    …
    private Node root;

    public void postorderPrint() {
        if (root != null) {
            postorderPrintTree(root);
        }
        System.out.println();
    }

    private static void postorderPrintTree(Node root) {
        if (root.left != null) {
            postorderPrintTree(root.left);
        }
        if (root.right != null) {
            postorderPrintTree(root.right);
        }
        System.out.print(root.key + " ");
    }
}
```
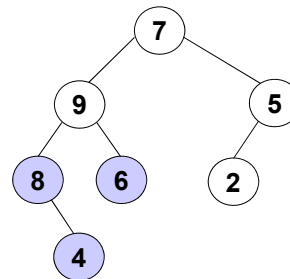
*   Note that the root is printed *after* the two recursive calls.

Tracing Postorder Traversal

```java
void postorderPrintTree(Node root) {
    if (root.left != null) {
        postorderPrintTree(root.left);
    }
    if (root.right != null) {
        postorderPrintTree(root.right);
    }
    System.out.print(root.key + " ");
}
```

order in which nodes are visited:

			root: ④ print 4			
		root: ⑧	root: ⑧	root: ⑧ print 8		root: ⑥ print 6
	root: ⑨	root: ⑨	root: ⑨	root: ⑨	root: ⑨	root: ⑨
root: ⑦	root: ⑦	root: ⑦	root: ⑦	root: ⑦	root: ⑦	root: ⑦

time ——————→

3: Inorder Traversal

- inorder traversal of the tree whose root is N:
    1) recursively perform an inorder traversal of N's left subtree
    2) visit the root, N
    3) recursively perform an inorder traversal of N's right subtree

- The root of the tree as a whole is visited between its subtrees.

- We'll see later why this is called *in*order traversal!

Implementing Inorder Traversal

```java
public class LinkedTree {
    …
    private Node root;

    public void inorderPrint() {
        if (root != null) {
            inorderPrintTree(root);
        }
        System.out.println();
    }
    private static void inorderPrintTree(Node root) {
        if (root.left != null) {
            inorderPrintTree(root.left);
        }
        System.out.print(root.key + " ");
        if (root.right != null) {
            inorderPrintTree(root.right);
        }
    }
}
```
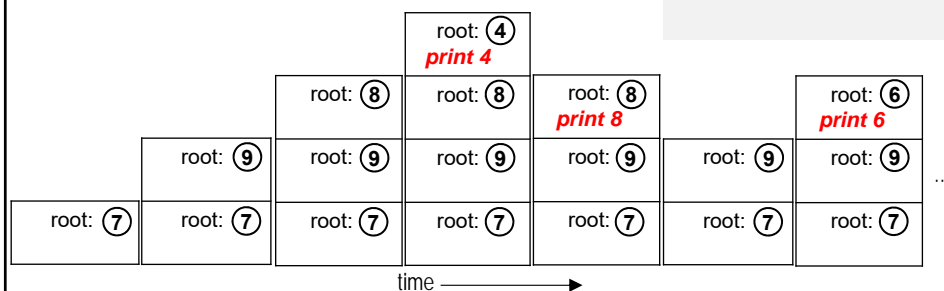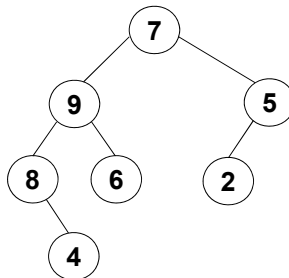
- Note that the root is printed *between* the two recursive calls.

Tracing Inorder Traversal

```
void inorderPrintTree(Node root) {
    if (root.left != null) {
        inorderPrintTree(root.left);
    }
    System.out.print(root.key + " ");
    if (root.right != null) {
        inorderPrintTree(root.right);
    }
}
```
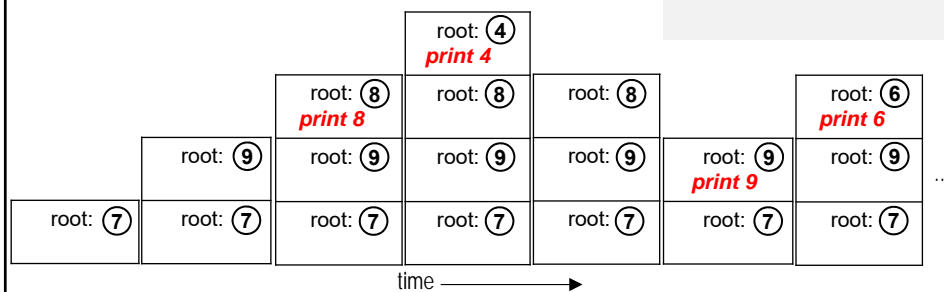
order in which nodes are visited:

	root: ⑦	root: ⑦	root: ⑦	root: ⑦	root: ⑦	root: ⑦
		root: ⑨	root: ⑨	root: ⑨	root: ⑨ *print 9*	root: ⑨
		root: ⑧ *print 8*	root: ⑧	root: ⑧		root: ⑥ *print 6*
			root: ④ *print 4*			
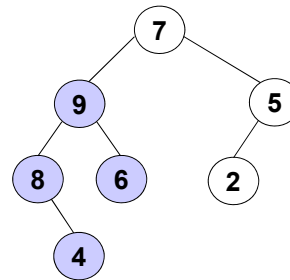
time ⟶
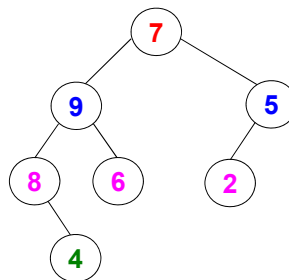

Level-Order Traversal

- Visit the nodes one level at a time, from top to bottom and left to right.

- Level-order traversal of the tree above: **7 9 5 8 6 2 4**

- We can implement this type of traversal using a queue.

Tree-Traversal Summary

preorder: root, left subtree, right subtree
postorder: left subtree, right subtree, root
inorder: left subtree, root, right subtree
level-order: top to bottom, left to right

- Extra practice: perform each type of traversal on the tree below:
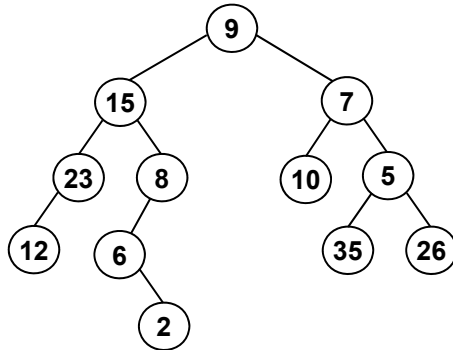
Tree Traversal Puzzle

- preorder traversal: A M P K L D H T

- inorder traversal: P M L K A H T D

- Draw the tree!

- What's one fact that we can easily determine from one
  of the traversals?

- How could we determine the nodes in each of the root's subtrees?

- *What are the roots of each subtree?*

Using a Binary Tree for an Algebraic Expression

- We'll restrict ourselves to fully parenthesized expressions using the following binary operators: +, –, *, /

- Example: `((a + (3 * c)) – (d / 2))`

- Leaf nodes are variables or constants.

- Interior nodes are operators.
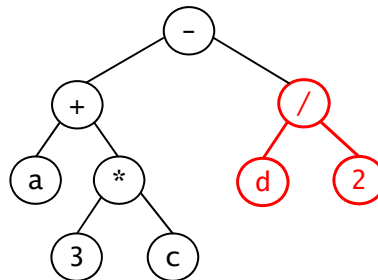 - their children are their operands

Traversing an Algebraic-Expression Tree

- Inorder gives conventional algebraic notation.
  - print '(' before the recursive call on the left subtree
  - print ')' after the recursive call on the right subtree
  - for tree at right: `((a + (b * c)) – (d / e))`

- Preorder gives functional notation.
  - print '('s and ')'s as for inorder, and commas after the recursive call on the left subtree
  - for tree above: `subtr(add(a, mult(b, c)), divide(d, e))`

- Postorder gives the order in which the computation must be carried out on a stack/RPN calculator.
  - for tree above: `push a, push b, push c, multiply, add,…`
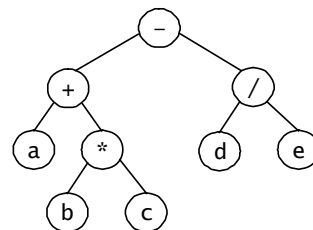
Search Trees

Computer Science 112
Boston University

Binary *Search* Trees

- Search-tree property: for each node *k* (*k* is the key):
  - all nodes in *k*'s left subtree are < *k*
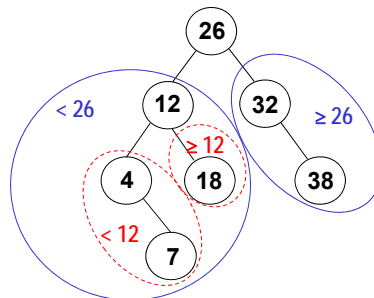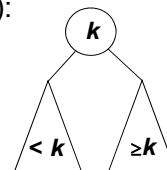 - all nodes in *k*'s right subtree are >= *k*

- Our earlier binary-tree example is
  a search tree:

- With a search tree, an inorder traversal visits the nodes in order!
  - in order of increasing key values

Searching for an Item in a Binary Search Tree

- Algorithm for searching for an item with a key *k*:

    if *k* == the root node's key, you're done
    else if *k* < the root node's key, search the left subtree
    else search the right subtree

- Example: search for 7
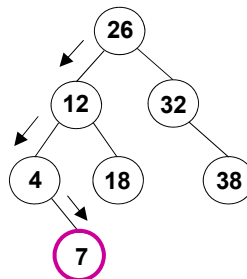
Implementing Binary-Tree Search

```
public class LinkedTree {    // Nodes have keys that are ints
    …
    private Node root;

    public LLList search(int key) {      // "wrapper method"
        Node n = searchTree(root, key);  // get Node for key
        if (n == null) {
            return null;     // no such key
        } else {
            return n.data;   // return list of values for key
        }
    }

    private static Node searchTree(Node root, int key) {
        if (              ) {

        } else if (              ) {

        } else if (              ) {

        } else {

        }
    }
```

two base cases
(order matters!)

two
recursive cases

Inserting an Item in a Binary Search Tree

* `public void insert(int key, Object data)`
  will add a new (key, data) pair to the tree

* Example 1: a search tree containing student records
  * key = the student's ID number (an integer)
  * data = a string with the rest of the student record
  * we want to be able to write client code that looks like this:
    ```
    LinkedTree students = new LinkedTree();
    students.insert(23, "Jill Jones,sophomore,comp sci");
    students.insert(45, "Al Zhang,junior,english");
    ```

* Example 2: a search tree containing scrabble words
  * key = a scrabble score (an integer)
  * data = a word with that scrabble score
    ```
    LinkedTree tree = new LinkedTree();
    tree.insert(4, "lost");
    ```

Inserting an Item in a Binary Search Tree (cont.)

* To insert an item (*k*, *d*),
  we start by searching for *k.*

* If we find a node with key *k*, we add
  d to the list of data values for that node.
  * example: `tree.insert(4, "sail")`

* If we don't find *k*, the last node seen
  in the search becomes the parent **P**
  of the new node **N**.
  * if *k* < **P**'s key, make **N** the left child of **P**
  * else make **N** the right child of **P**

* *Special case:* if the tree is empty,
  make the new node the root of the tree.

* ***Important:*** The resulting tree is still a search tree!

example:
`tree.insert(35,`
`  "photooxidizes")`
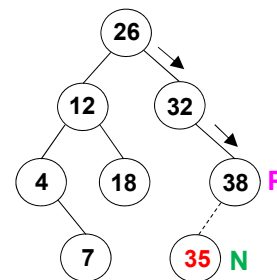
Implementing Binary-Tree Insertion

- We'll implement part of the `insert()` method together.

- We'll use iteration rather than recursion.

- Our method will use two references/pointers:
  - `trav`: performs the traversal down to the point of insertion
  - `parent`: stays one behind `trav`
    - like the `trail` reference that we sometimes use when traversing a linked list

parent

26
12   32
4   18   38
7
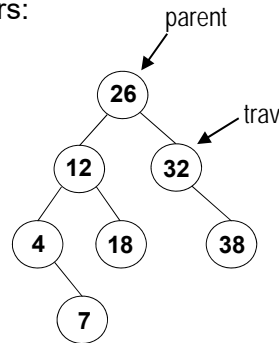
trav

Implementing Binary-Tree Insertion

parent

trav

insert 35:

```
public void insert(int key, Object data) {
    Node parent = null;
    Node trav = root;
    while (trav != null) {
        if (trav.key == key) {
            trav.data.addItem(data, 0);
            return;
        }
        // what should go here?



    }
    Node newNode = new Node(key, data);
    if (root == null) {    // the tree was empty
        root = newNode;
    } else if (key < parent.key) {
        parent.left = newNode;
    } else {
        parent.right = newNode;
    }
}
```

26
12   32
4   18   38
7
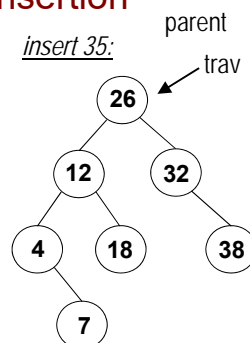
Deleting Items from a Binary Search Tree

- Three cases for deleting a node *x*

- **Case 1:** *x* has no children.
  Remove *x* from the tree by setting its parent's reference to null.

  ex: delete 4
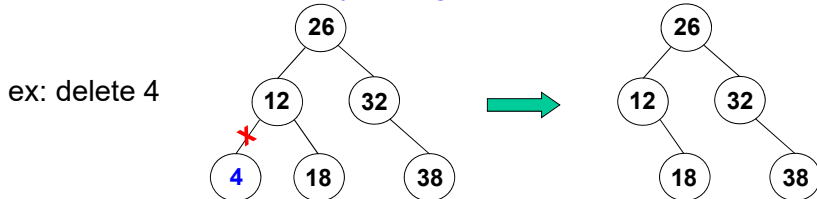
- **Case 2:** *x* has one child.
  Take the parent's reference to *x* and make it refer to *x*'s child.

  ex: delete 12
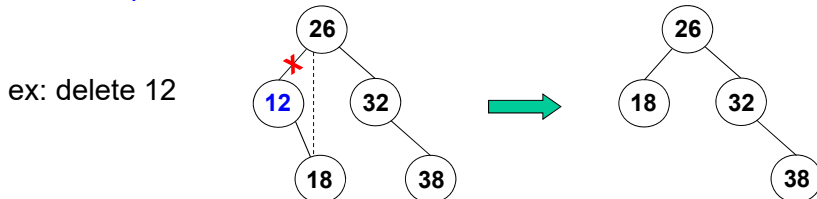
Deleting Items from a Binary Search Tree (cont.)

- **Case 3:** *x* has two children

  - we can't give both children to the parent. why?

  - instead, we leave *x's* node where it is, and we replace its key and data with those from another node
    - the replacement must maintain the search-tree inequalities

ex:
delete 12

two options: *which ones?*
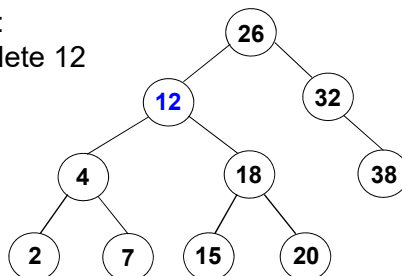
Deleting Items from a Binary Search Tree (cont.)

- **Case 3:** *x* has two children (continued):
  - replace *x*'s key and data with those from the smallest node in *x*'s right subtree—call it *y*
  - we then delete *y*
    - it will either be a leaf node or will have one right child. why?

    - thus, we can delete it using case 1 or 2

ex: delete 12
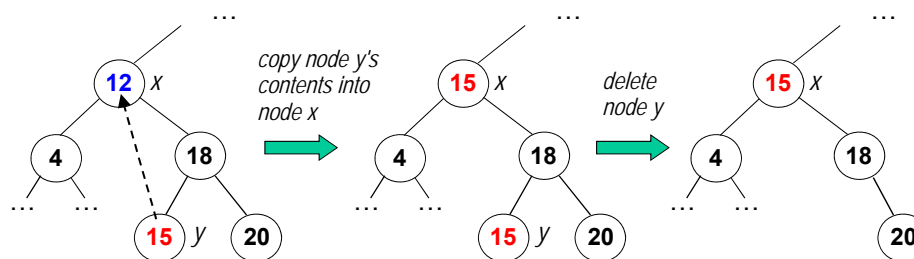
Which Node Would Be Used To Replace 9?
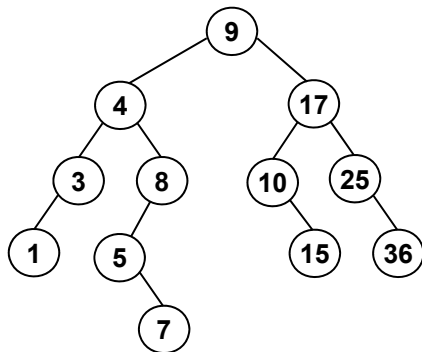
A. **4**

B. **8**

C. **10**

D. **15**

E. **17**

Implementing Deletion

delete 26:

```
public LLList delete(int key) {
    // Find the node and its parent.
    Node parent = null;
    Node trav = root;
    while (trav != null && trav.key != key) {
        parent = trav;
        if (key < trav.key) {
            trav = trav.left;
        } else {
            trav = trav.right;
        }
    }

    // Delete the node (if any) and return the removed items.
    if (trav == null) {    // no such key
        return null;
    } else {
        LLList removedData = trav.data;
        deleteNode(trav, parent);   // call helper method
        return removedData;
    }
}
```

```
        50
       /  parent
     15    trav
       \
        26
       /  \
     18    45
          /
        30
          \
          35
```
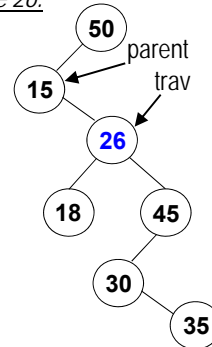
Implementing Case 3

```
private void deleteNode(Node toDelete, Node parent) {
    if (toDelete.left != null && toDelete.right != null) {
        // Find a replacement – and
        // the replacement's parent.
        Node replaceParent = toDelete;

        // Get the smallest item
        // in the right subtree.
        Node replace = toDelete.right;
        // what should go here?




        // Replace toDelete's key and data
        // with those of the replacement item.
        toDelete.key = replace.key;
        toDelete.data = replace.data;

        // Recursively delete the replacement
        // item's old node. It has at most one
        // child, so we don't have to
        // worry about infinite recursion.
        deleteNode(replace, replaceParent);
    } else {
        ...
    }
}
```

```
     toDelete
         |
        26
       /  \
     18    45
          /
        30
          \
          35
```
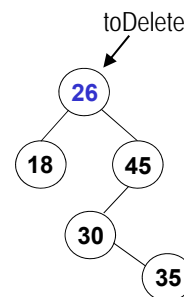
Implementing Cases 1 and 2

```
private void deleteNode(Node toDelete, Node parent) {
    if (toDelete.left != null && toDelete.right != null) {
        ...
    } else {
        Node toDeleteChild;
        if (toDelete.left != null)
            toDeleteChild = toDelete.left;
        else
            toDeleteChild = toDelete.right;
        // Note: in case 1, toDeleteChild
        // will have a value of null.

        if (toDelete == root)
            root = toDeleteChild;
        else if (toDelete.key < parent.key)
            parent.left = toDeleteChild;
        else
            parent.right = toDeleteChild;
    }
}
```

30
parent
18   45
toDelete
30
35
toDeleteChild
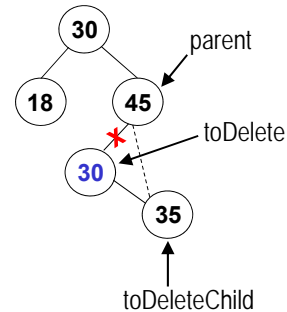
Recall: Path, Depth, Level, and Height
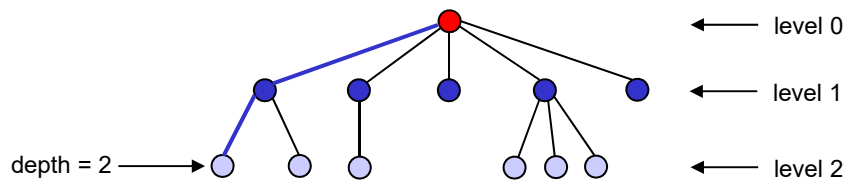
depth = 2

level 0
level 1
level 2

- There is exactly one *path* (one sequence of edges) connecting each node to the root.

- *depth* of a node = # of edges on the path from it to the root

- Nodes with the same depth form a *level* of the tree.

- The *height* of a tree is the maximum depth of its nodes.
    - example: the tree above has a height of 2

Efficiency of a Binary Search Tree

- For a tree containing n items, what is the efficiency
  of any of the traversal algorithms?
  - you process all n of the nodes
  - you perform $O(1)$ operations on each of them

- Search, insert, and delete all have the same time complexity.
  - insert is a search followed by $O(1)$ operations
  - delete involves either:
    - a search followed by $O(1)$ operations (cases 1 and 2)
    - a search partway down the tree for the item,
      followed by a search further down for its replacement,
      followed by $O(1)$ operations (case 3)

Efficiency of a Binary Search Tree (cont.)

- Time complexity of searching:
  - best case:

  - worst case:
    - you have to go all the way down to level h
      before finding the key or realizing it isn't there
    - along the path to level h, you process $h + 1$ nodes
  - average case:

- What is the height of a tree containing n items?

Balanced Trees

- A tree is *balanced* if, for *each* of its nodes, the node's subtrees have the same height or have heights that differ by 1.
    - example:
        - 26: both subtrees have a height of 1
        - 12: left subtree has height 0
            right subtree is empty (height = -1)
        - 32: both subtrees have a height of 0
        - all leaf nodes: both subtrees are empty

- For a balanced tree with n nodes, height = $O(\log n)$
    - each time that you follow an edge down the longest path, you cut the problem size roughly in half!

- Therefore, for a *balanced* binary search tree, the worst case for search / insert / delete is $O(h) = O(\log n)$
    - the "best" worst-case time complexity
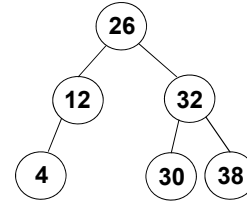
What If the Tree Isn't Balanced?

- Extreme case: the tree is equivalent to a linked list
    - height = $n - 1$

- Therefore, for a unbalanced
  binary search tree, the worst case
  for search / insert / delete is $O(h) = O(n)$
    - the "worst" worst-case time complexity

- We'll look next at search-tree variants
  that take special measures to ensure balance.
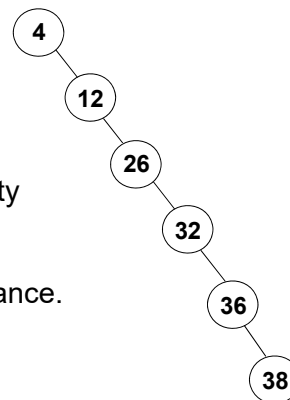
2-3 Trees

* A 2-3 tree is a balanced tree in which:
    * *all* nodes have equal-height subtrees (perfect balance)
    * each node is either
        * a **2-node**, which contains one data item and 0 or 2 children
        * a **3-node**, which contains two data items and 0 or 3 children
    * the keys form a search tree

* Example:

```
                        28  61
            ┌─────────────┼─────────────┐
           10            40           77  90
         ┌──┴──┐       ┌──┴──┐     ┌────┼────┐
         3   14 20    34    51    68  80 87  93 97
```

2-node:

```
      k
     ╱ ╲
   <k   ≥k
```

3-node:

```
     k1    k2
    ╱  │  ╲
  <k1  ≥k1  ≥k2
       <k2
```
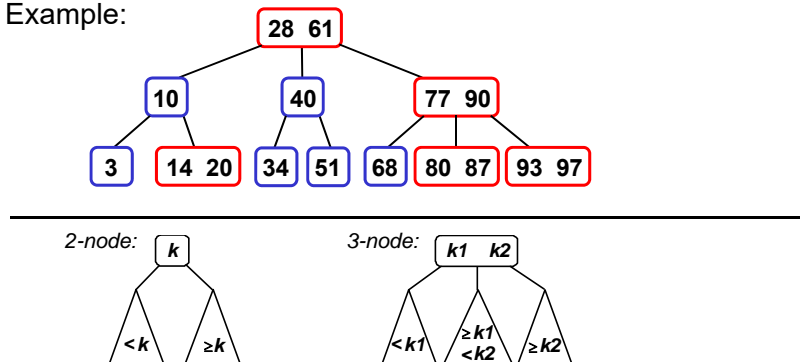
Search in 2-3 Trees

* Algorithm for searching for an item with a key *k*:

      if *k* == one of the root node's keys, you're done
      else if *k* < the root node's first key
          search the left subtree
      else if the root is a 3-node and k < its second key
          search the middle subtree
      else
          search the right subtree

```
     k1    k2
    ╱  │  ╲
  <k1  ≥k1  ≥k2
       <k2
```

* Example: search for 87
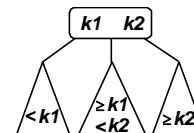
```
                        28  61
            ┌─────────────┼──────────↘
           10            40           77  90
         ┌──┴──┐       ┌──┴──┐     ┌───┼↓───┐
         3   14 20    34    51    68  80 87  93 97
```
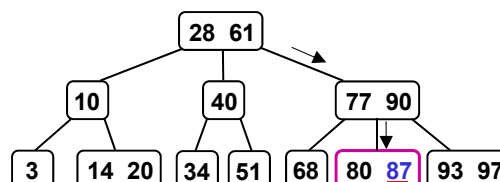
Insertion in 2-3 Trees

- Algorithm for inserting an item with a key *k*:

    search for *k,* but don't stop until you hit a leaf node
    let L be the leaf node at the end of the search
    if L is a 2-node
        add *k* to L, making it a 3-node

    else if L is a 3-node
        split L into two 2-nodes containing the items with the
            smallest and largest of: *k,* L's 1st key, L's 2nd key
        the middle item is "sent up" and inserted in L's parent
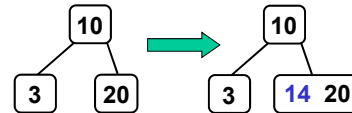
    example: add 52
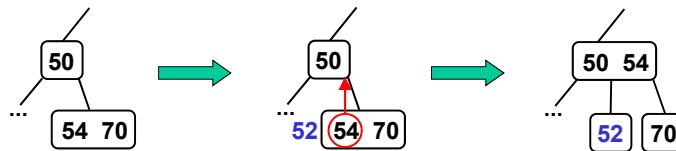
Example 1: Insert 8

- Search for 8:
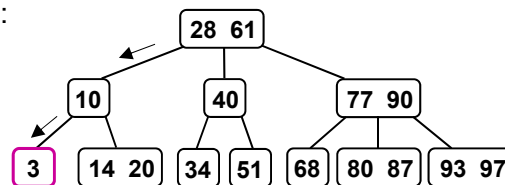
- Add 8 to the leaf node, making it a 3-node:
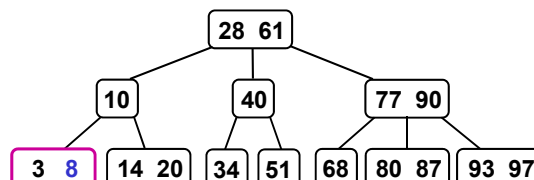
Example 2: Insert 17

- Search for 17:
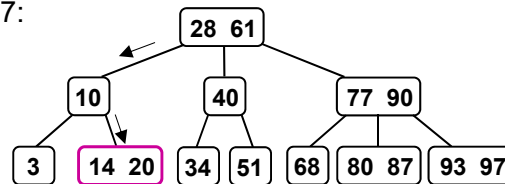
- Split the leaf node, and send up the middle of 14, 17, 20 and insert it the leaf node's parent:
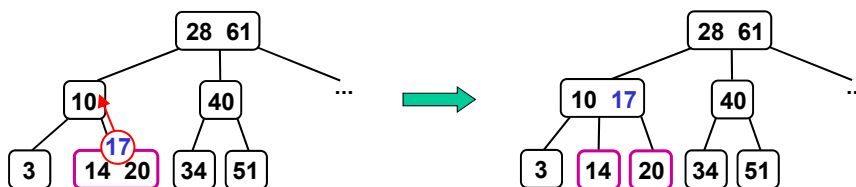
Example 3: Insert 92
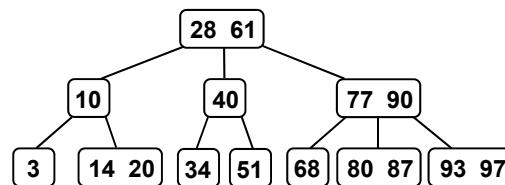
- In which node will we initially try to insert it?

Example 3: Insert 92

- Search for 92:
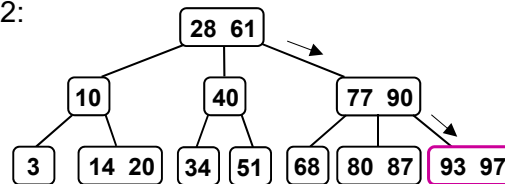
- Split the leaf node, and send up the middle of 92, 93, 97 and insert it the leaf node's parent:
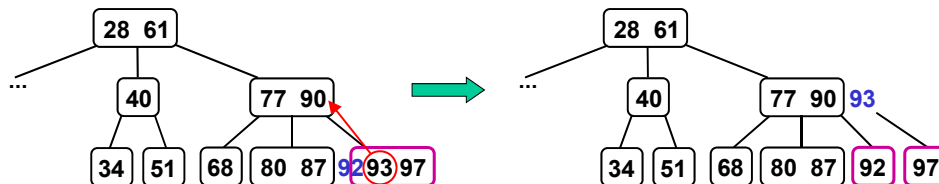
- In this case, the leaf node's parent is also a 3-node, so we need to split is as well...

Example 3 (cont.)

- We split the [77 90] node and we send up the middle of 77, 90, 93:

- We try to insert it in the root node, but the root is also full!
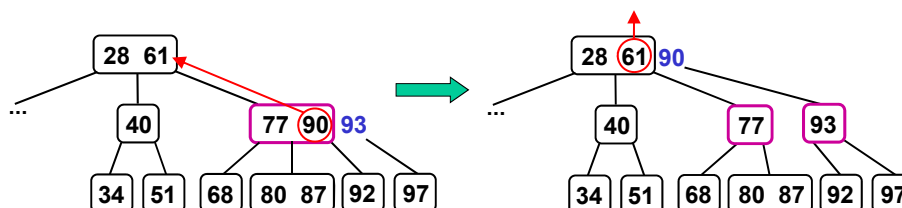
- Then we split the root, which increases the tree's height by 1, but the tree is still balanced.

- This is only case in which the tree's height increases.
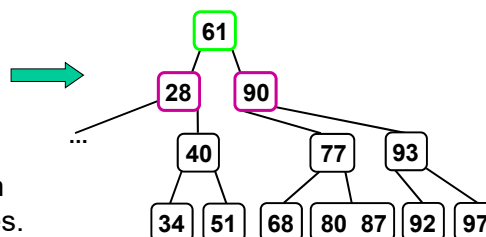
Efficiency of 2-3 Trees
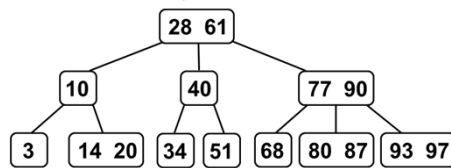
- A 2-3 tree containing n items has a height h <= $\log_2 n$.

- Thus, search and insertion are both *O*(log n).
  - search visits at most h + 1 nodes
  - insertion visits at most 2h + 1 nodes:
    - starts by going down the full height
    - in the worst case, performs splits all the way back up to the root

- Deletion is tricky – you may need to coalesce nodes!
  However, it also has a time complexity of *O*(log n).

- Thus, we can use 2-3 trees for a *O*(log n)-time data dictionary!

Extra Practice

- Starting with an empty 2-3 tree, insert the following
  sequence of keys:

  51, 3, 40, 77, 20, 10, 34, 28, 61, 80, 68, 93, 90, 97, 87, 14

You should get back the tree from the previous slide!

Hash Tables

Computer Science 112
Boston University

Data Dictionary Revisited

- We've considered several data structures that allow us to store and search for data items using their key fields:

data structure	searching for an item	inserting an item
a list implemented using an array	$O(\log n)$ using binary search	$O(n)$
a list implemented using a linked list	$O(n)$ using linear search	$O(n)$
binary search tree		
balanced search trees (2-3 tree, others)		

- We'll now look at hash tables, which can do better than $O(\log n)$.

Ideal Case: Searching = Indexing

* We would achieve optimal efficiency if we could treat the key as an index into an array.

* Example: storing data about members of a sports team
    * key = jersey number (some value from 0-99).
    * class for an individual player's record:
      ```
      public class Player {
          private int jerseyNum;
          private String firstName;
          …
      }
      ```
    * store the player records in an array:
      ```
      Player[] teamRecords = new Player[100];
      ```

* In such cases, search and insertion are *O*(1):
  ```
  public Player search(int jerseyNum) {
      return teamRecords[jerseyNum];
  }
  ```

Hashing: Turning Keys into Array Indices

* In most real-world problems, indexing is not as simple as the sports-team example. Why?
    *
    *
    *

* To handle these problems, we perform *hashing*:
    * use a *hash function* to convert the keys into array indices
        "Sullivan" → 18   "Papadakis" → 25
    * use techniques to handle cases in which multiple keys are assigned the same hash value

* The resulting data structure is known as a *hash table*.

Hash Functions

- A hash function defines a mapping from keys to integers.

- We then use the modulus operator to get a valid array index.

    key value ⇒ | **hash function** | ⇒ integer $\overset{\%}{⇒}$ integer in [0, n – 1]
    (n = array length)
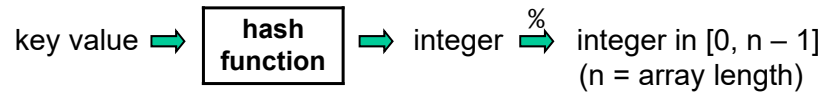
- Here's a very simple hash function for keys of lower-case letters:
    h(key) = ASCII value of first char – ASCII value of 'a'
    - examples:
        h("ant") = ASCII for 'a' – ASCII for 'a' = 0
        h("cat") = ASCII for 'c' – ASCII for 'a' = 2

- h(key) is known as the key's *hash code.*

- A *collision* occurs when items with different keys are assigned the same hash code.

Dealing with Collisions I: Separate Chaining

- Each position in the hash table serves as a *bucket* that can store multiple data items.

- Two options:
    1. each bucket is itself an array
        - need to preallocate, and a bucket may become full

    2. each bucket is a linked list
        - items with the same hash code are "chained" together
        - each "chain" can grow as needed

```
0 |    →   "ant"  →   "ape"
1 | null      |        null
2 |    →   "cat"
3 | null     null
… |  ...
```
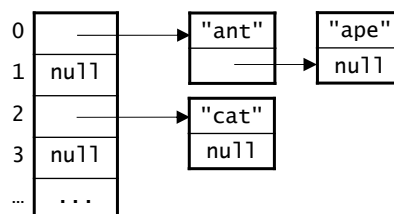
Dealing with Collisions II: Open Addressing

- When the position assigned by the hash function is occupied, find another open position.

- Example: "wasp" has a hash code of 22, but it ends up in position 23 because position 22 is occupied.

- We'll consider three ways of finding an open position – a process known as *probing*.

- We also perform probing when searching.
  - example: search for "wasp"
    - look in position 22
    - then look in position 23
  - need to figure out when to safely stop searching (more on this soon!)

0	"ant"
1	
2	"cat"
3	
4	"emu"
5	
6	
7	
…	...
22	"wolf"
23	**"wasp"**
24	"yak"
25	"zebra"

Linear Probing

- Probe sequence: h(key), h(key) + 1, h(key) + 2, …, wrapping around as necessary.

- Examples:
  - "ape" (h = 0) would be placed in position 1, because position 0 is already full.
  - "bear" (h = 1): try 1, 1 + 1, 1 + 2 – open!
  - where would "zebu" end up?

- Advantage: if there is an open cell, linear probing will eventually find it.

- Disadvantage: get "clusters" of occupied cells that lead to longer subsequent probes.
  - probe length = the number of positions considered during a probe

0	"ant"
1	"ape"
2	"cat"
3	"bear"
4	"emu"
5	
6	
7	
…	...
22	"wolf"
23	"wasp"
24	"yak"
25	"zebra"

Quadratic Probing

- Probe sequence: h(key), h(key) + 1^2, h(key) + 2^2, h(key) + 3^2, …, wrapping around as necessary.

- Examples:
  - "ape" (h = 0): try 0, 0 + 1 – open!
  - "bear" (h = 1): try 1, 1 + 1, 1 + 4 – open!
  - "zebu"?

- Advantage: smaller clusters of occupied cells

- Disadvantage: may fail to find an existing open position. For example:

0	"ant"
1	"ape"
2	"cat"
3	
4	"emu"
5	"bear"
6	
7	
…	. . .
22	"wolf"
23	"wasp"
24	"yak"
25	"zebra"

table size = 10
x = occupied

trying to insert a
key with h(key) = 0

offsets of the probe
sequence in italics

0	x		5	x	*25*
1	x	*1 81*	6	x	*16 36*
2			7		
3			8		
4	x	*4 64*	9	x	*9 49*

Double Hashing

- Use two hash functions:
  - h1 computes the hash code
  - h2 computes the increment for probing
  - probe sequence: h1, h1 + h2, h1 + 2*h2, …

0	"ant"
1	"bear"
2	"cat"
3	"ape"
4	"emu"
5	
6	
7	
…	. . .
22	"wolf"
23	"wasp"
24	"yak"
25	"zebra"

- Examples:
  - h1 = our previous h
  - h2 = number of characters in the string
  - "ape" (h1 = 0, h2 = 3): try 0, 0 + 3 – open!
  - "bear" (h1 = 1, h2 = 4): try 1 – open!
  - "zebu"?

- Combines good features of linear and quadratic:
  - reduces clustering
  - will find an open position if there is one, provided the table size is a prime number

Removing Items Under Open Addressing

- Problematic example (using linear probing):
  - insert "ape" (h = 0): try 0, 0 + 1 – open!
  - insert "bear" (h = 1): try 1, 1 + 1, 1 + 2 – open!
  - remove "ape"
  - search for "ape": try 0, 0 + 1 – conclude not in table
  - search for "bear": **try 1 – conclude not in table, but "bear" is further down in the table!**

0	"ant"
1	
2	"cat"
3	"bear"
4	"emu"
5	
…	...
22	"wolf"
23	"wasp"
24	"yak"
25	"zebra"

- To fix this problem, distinguish between:
  - *removed positions* that previously held an item
  - *empty positions* that have never held an item

- During probing, we *don't* stop if we see a removed position.
  ex: search for "bear": try 1 (removed), 1 + 1, **1 + 2 – found!**

- We can insert items in either empty or removed positions.

An Interface For Hash Tables

```
public interface HashTable {
    boolean insert(Object key, Object value);
    Queue<Object> search(Object key);
    Queue<Object> remove(Object key);
}
```

- `insert()` takes a key-value pair and returns:
  - `true` if the key-value pair can be added
  - `false` if it cannot be added (referred to as *overflow*)

- `search()` and `remove()` both take a key, and return a queue containing all of the values associated with that key.
  - example: an index for a book
    - key = word
    - values = the pages on which that word appears
  - return `null` if the key is not found

An Implementation Using Open Addressing

```
public class OpenHashTable implements HashTable {
    private class Entry {
        private Object key;
        private LLQueue<Object> values;
        …
    }
    …
    private Entry[] table;
    private int probeType;
}
```

table [| —]
probeType [LINEAR]

0	
1	
2	null
3	null
4	null
…	…

"ant"
LLQueue object

"ape"
LLQueue object
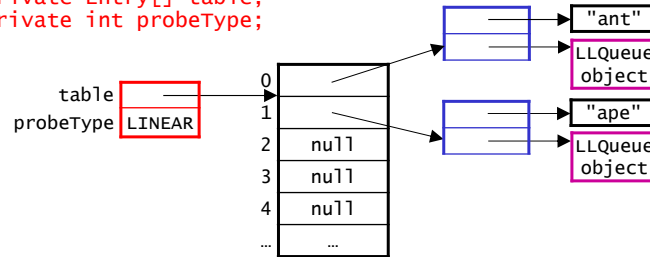
- We use a private inner class for the entries in the hash table.

- We use an `LLQueue` for the values associated with a given key.

Empty vs. Removed

- When we remove a key and its values, we:
  - leave the `Entry` object in the table
  - set the `Entry` object's `key` and `values` fields to `null`
  - example: after `remove("ape")`:

table [| —]
probeType [LINEAR]

0	
1	
2	null
3	null
4	null
…	…

"ant"
LLQueue object

null
null

"ape"
LLQueue object
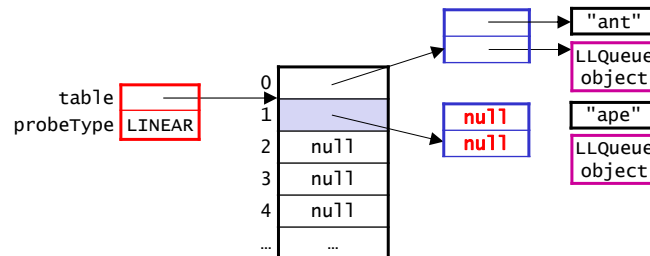
- Note the difference:
  - a truly empty position has a value of `null` in the table
    (example: positions 2, 3 and 4 above)

  - a removed position refers to an `Entry` object whose
    `key` and `values` fields are `null` (example: position 1 above)

Probing Using Double Hashing

```
private int probe(Object key) {
    int i = h1(key);    // first hash function
    int h2 = h2(key);   // second hash function

    // keep probing until we get an empty position or match
    while (table[i] != null && !key.equals(table[i].key)) {
        i = (i + h2) % table.length;
    }

    return i;
}
```

* It is essential that we:
  * check for `table[i] != null` first. why?

  * call the `equals` method on `key`, not `table[i].key`. why?

Avoiding an Infinite Loop

* The while loop in our probe method could lead to an infinite loop.

  ```
  while (table[i] != null && !key.equals(table[i].key)) {
      i = (i + h2) % table.length;
  }
  ```

* When would this happen?

* We can stop probing after checking n positions (n = table size), because the probe sequence will just repeat after that point.
  * for quadratic probing:
    $(h1 + n^2) \% n = h1 \% n$
    $(h1 + (n+1)^2) \% n = (h1 + n^2 + 2n + 1) \% n = (h1 + 1) \% n$
  * for double hashing:
    $(h1 + n*h2) \% n = h1 \% n$
    $(h1 + (n+1)^*h2) \% n = (h1 + n*h2 + h2) \% n = (h1 + h2) \% n$

Avoiding an Infinite Loop (cont.)

```java
private int probe(Object key) {
    int i = h1(key);     // first hash function
    int h2 = h2(key);    // second hash function
    int numChecked = 1;

    // keep probing until we get an empty position or a match
    while (table[i] != null && !key.equals(table[i].key)) {
        if (numChecked == table.length) {
            return -1;
        }
        i = (i + h2) % table.length;
        numChecked++;
    }

    return i;
}
```

Search and Removal

```java
public LLQueue<Object> search(Object key) {
    // throw an exception if key == null
    int i = probe(key);
    if (i == -1 || table[i] == null) {
        return null;
    } else {
        return table[i].values;
    }
}

public LLQueue<Object> remove(Object key) {
    // throw an exception if key == null
    int i = probe(key);
    if (i == -1 || table[i] == null) {
        return null;
    }

    LLQueue<Object> removedVals = table[i].values;
    table[i].key = null;
    table[i].values = null;
    return removedVals;
}
```

Insertion

- We begin by probing for the key.

- Several cases:
    1. the key is already in the table (we're inserting a duplicate)
       → add the value to the values in the key's `Entry`

    2. the key is not in the table: three subcases:
        a. encountered 1 or more removed positions while probing
           → put the (key, value) pair in the *first* removed position seen during probing. why?

        b. no removed position; reached an empty position
           → put the (key, value) pair in the empty position

        c. no removed position or empty position
           → overflow; return `false`

Tracing Through Some Examples

- Start with the hash table at right with:
    - double hashing
    - our earlier hash functions h1 and h2

- Perform the following operations:
    - insert "bear" (h1 = 1, h2 = 4):
    - insert "bison" (h1 = 1, h2 = 5):

    - insert "cow" (h1 = 2, h2 = 3):

    - delete "emu" (h1 = 4, h2 = 3):

    - search "eel" (h1 = 4, h2 = 3):

    - insert "bee" (h1 = ___, h2 = ____):

0	"ant"
1	
2	"cat"
3	
4	"emu"
5	"fox"
6	
7	
8	
9	
10	

Dealing with Overflow

* Overflow = can't find a position for an item

* When does it occur?
  * linear probing:
  * quadratic probing:
    *
    *
  * double hashing:
    * if the table size is a prime number: same as linear
    * if the table size is not a prime number: same as quadratic

* To avoid overflow (and reduce search times), grow the hash table when the % of occupied positions gets too big.
  * problem: we need to rehash **all** of the existing items. why?

Implementing the Hash Function

* Characteristics of a good hash function:
  1) efficient to compute
  2) uses the entire key
     * changing any char/digit/etc. should change the hash code
  3) distributes the keys more or less uniformly across the table
  4) must be a function!
     * a key must always get the same hash code

* In Java, every object has a `hashCode()` method.
  * the version inherited from `Object` returns a value based on an object's memory location
  * classes can override this version with their own

Hash Functions in `OpenHashTable`

* Initial hash function: returns a value in [0, `table.length` - 1]

```java
public int h1(Object key) {
    int h1 = key.hashCode() % table.length;
    if (h1 < 0) {
        h1 += table.length;
    }
    return h1;
}
```

* Second hash function (for double hashing):

```java
public h2(Object key) {
    int h2 = key.hashCode() % 5;
    if (h2 < 0) {
        h2 += 11;
    }
    h2 += 5;
    return h2;
}
```

  * 5 and 11 are values that could be adjusted as needed
  * provide a range of possible increments >= 5

Hash Table Efficiency

* In the best case, search and insertion are $O(1)$.

* In the worst case, search and insertion are linear.
  * open addressing: $O(m)$, where m = the size of the hash table
  * separate chaining: $O(n)$, where n = the number of keys

* With good choices of hash function and table size,
  complexity is generally better than $O(\log n)$ and approaches $O(1)$.

* *load factor* = # keys in table / size of the table.
  To prevent performance degradation:
  * open addressing: try to keep the load factor < 1/2
  * separate chaining: try to keep the load factor < 1

* Time-space tradeoff: bigger tables have better performance,
  but they use up more memory.

Hash Table Limitations

- It can be hard to come up with a good hash function for a particular data set.

- The items are not ordered by key. As a result, we can't easily:
  - print the contents in sorted order
  - perform a range search (find all values between v1 and v2)
  - perform a rank search – get the kth largest item

  We *can* do all of these things with a search tree.

Extra Practice

- Start with the hash table at right with:
  - double hashing
  - h1(key) = ASCII of first letter – ASCII of 'a'
  - h2(key) = key.length()
  - shaded cells are removed cells

- What is the *probe sequence* for "baboon"? (the sequence of positions seen during probing)

0	"ant"
1	
2	"cat"
3	
4	"emu"
5	
6	
7	
8	
9	
10	

- If we insert "baboon", in what position will it go?

Heaps and Priority Queues

Computer Science 112
Boston University

Priority Queue

- A *priority queue* (PQ) is a collection in which each item
  has an associated number known as a *priority*.
  - ("Jean Morrison", 10), ("Robert Brown", 15),
    ("Dave Sullivan", 5)
  - use a higher priority for items that are "more important"

- Example application: scheduling a shared resource like the CPU
  - give some processes/applications a higher priority,
    so that they will be scheduled first and/or more often

- Key operations:
  - *insert:* add an item (with a position based on its priority)
  - *remove:* remove the item with the highest priority

- One way to implement a PQ efficiently is using a type of
  binary tree known as a *heap.*

Complete Binary Trees

- A binary tree of height *h* is *complete* if:
  - levels 0 through *h* - 1 are fully occupied
  - there are no "gaps" to the left of a node in level *h*

- Complete:
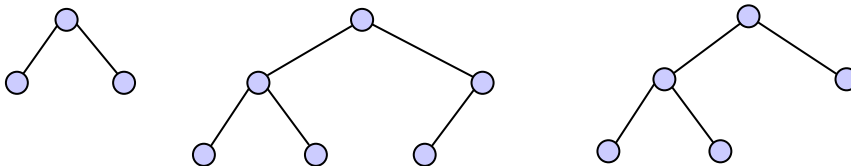
- Not complete (⭘ = missing node):
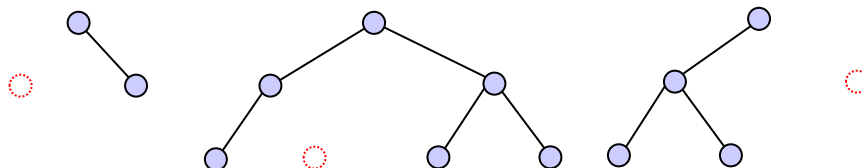
Representing a Complete Binary Tree

- A complete binary tree has a simple array representation.

- The tree's nodes are stored in the array in the order given by a level-order traversal.
  - top to bottom, left to right
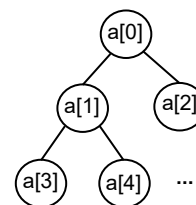
- Examples:
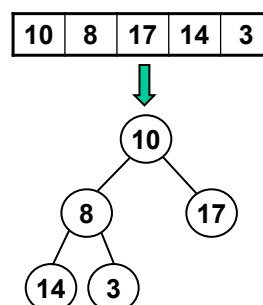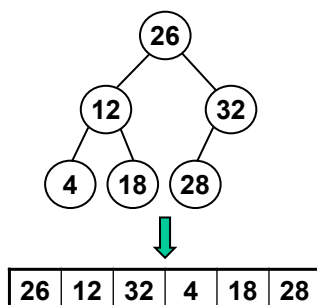
Navigating a Complete Binary Tree in Array Form

- The root node is in `a[0]`

- Given the node in `a[i]`:
    - its left child is in `a[2*i + 1]`
    - its right child is in `a[2*i + 2]`
    - its parent is in `a[(i - 1)/2]` (using integer division)
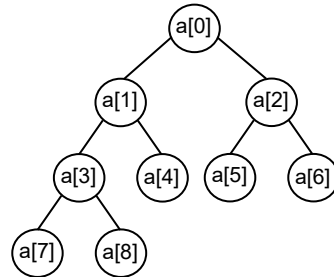
- Examples:
    - the left child of the node in `a[1]` is in `a[2*1 + 1]` = `a[3]`
    - the left child of the node in `a[2]` is in `a[2*2 + 1]` = `a[5]`
    - the right child of the node in `a[3]` is in `a[2*3 + 2]` = `a[8]`

    - the right child of the node in `a[2]` is in _____
    - the parent of the node in `a[4]` is in `a[(4-1)/2]` = `a[1]`

    - the parent of the node in `a[7]` is in _____

What is the left child of 24?

- Assume that the following array represents a complete tree:

0	1	2	3	4	5	6	7	8
26	12	32	24	18	28	47	10	9

Heaps

- Heap: a complete binary tree in which each interior node is greater than or equal to its children
  - examples:

```
        28                  18                  12
       /  \                /  \                /  \
     16    20             8    2              7    10
    / \   /             / \
  12   8 5             3   7
```
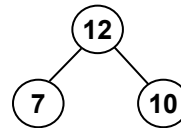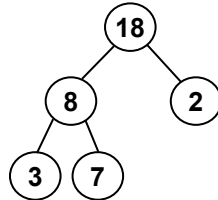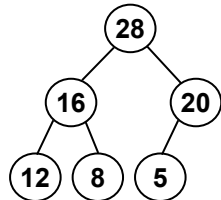
- The largest value is always at the root of the tree.

- The smallest value can be in *any* leaf node - there's no guarantee about which one it will be.

- We're using *max-at-top* heaps.
  - in a *min-at-top* heap, every interior node <= its children

---

Which of these is a heap?

```
A.      28          B.      18          C.      12
       /  \                /  \                /  \
     16    20             8    2              7    10
    / \   /             / \                  /      \
  12  18 5             3   7                2        5
```
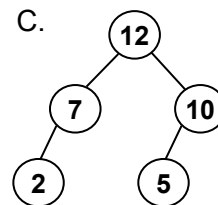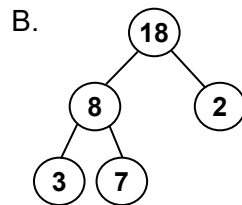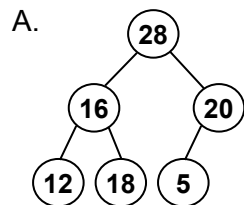
D. more than one (which ones?)

E. none of them

How to Compare Objects

- We need to be able to compare items in the heap.

- If those items are objects, we can't just do something like this:

      if (item1 < item2)

   Why not?

- Instead, we need to use a method to compare them.

An Interface for Objects That Can Be Compared

- The `Comparable` interface is a built-in generic Java interface:

```
public interface Comparable<T> {
    public int compareTo(T other);
}
```

- It is used when defining a class of objects that can be ordered.

- Examples from the built-in Java classes:

```
public class String implements Comparable<String> {
    ...
    public int compareTo(String other) {
        ...
}
public class Integer implements Comparable<Integer> {
    ...
    public int compareTo(Integer other) {
        ...
}
```

An Interface for Objects That Can Be Compared (cont.)

```
public interface Comparable<T> {
    public int compareTo(T other);
}
```

* `item1.compareTo(item2)` should return:
    * a negative integer if `item1` "comes before" `item2`
    * a positive integer if `item1` "comes after" `item2`
    * 0 if `item1` and `item2` are equivalent in the ordering

* These conventions make it easy to construct appropriate method calls:

<u>numeric comparison</u>   <u>comparison using `compareTo`</u>
```
item1 < item2          item1.compareTo(item2) < 0
item1 > item2          item1.compareTo(item2) > 0
item1 == item2         item1.compareTo(item2) == 0
```

Heap Implementation

```
public class Heap<T extends Comparable<T>> {
    private T[] contents;
    private int numItems;

    public Heap(int maxSize) {
        contents = (T[])new Comparable[maxSize];
        numItems = 0;
    }
    ...
}
```


a Heap *object*
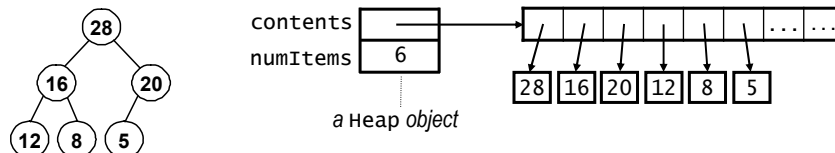
* `Heap` is another example of a generic collection class.
    * as usual, `T` is the type of the elements
    * `extends Comparable<T>` specifies `T` must implement `Comparable<T>`
    * must use `Comparable` (not `Object`) when creating the array

Heap Implementation (cont.)

```
public class Heap<T extends Comparable<T>> {
    private T[] contents;
    private int numItems;

    …
}
```


a Heap object
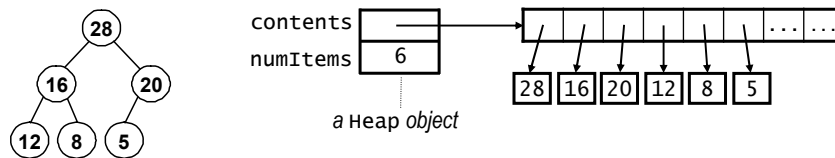
* The picture above is a heap of integers:

    `Heap<Integer> myHeap = new Heap<Integer>(20);`
    * works because `Integer` implements `Comparable<Integer>`
    * could also use `String` or `Double`

Removing the Largest Item from a Heap

* Remove and return the item in the root node.

* In addition, need to move the largest remaining item to the root, while maintaining a complete tree with each node >= children

* Algorithm:
    1. make a copy of the largest item
    2. move the last item in the heap
       to the root
    3. "sift down" the new root item
       until it is >= its children (or it's a leaf)
    4. return the largest item
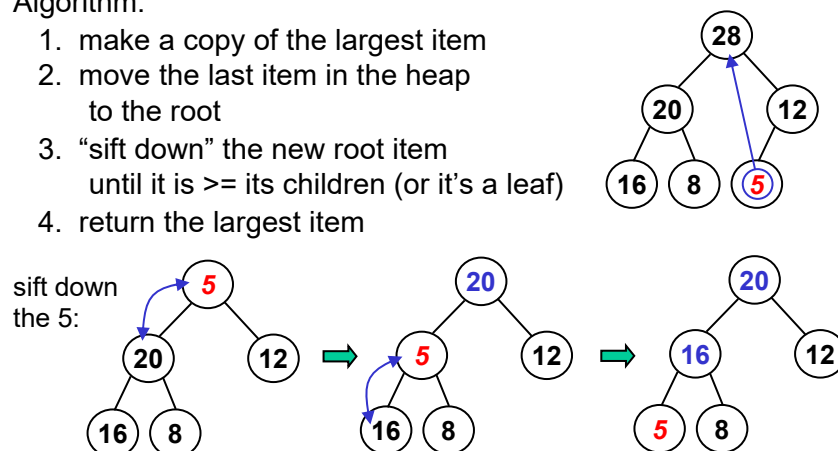
sift down the 5:

Sifting Down an Item

- To sift down item x (i.e., the item whose key is x):
    1. compare x with the larger of the item's children, y
    2. if $x < y$, swap x and y and repeat

- Other examples:

sift down
the 10:

```
        10                         18
    7        18          7             10
  3   5    8   6       3   5        8     6
```

sift down
the 7:

```
        7
    26      23
  15  18  10
```
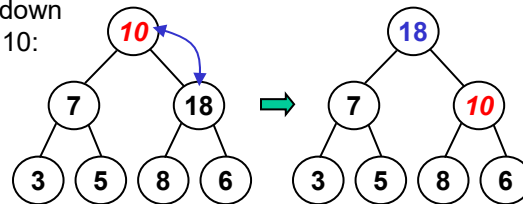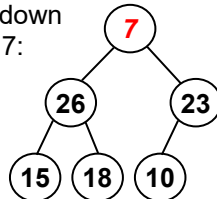
Inserting an Item in a Heap

- Algorithm:
    1. put the item in the next available slot (grow array if needed)
    2. "sift up" the new item until it is <= its parent
       (or it becomes the root item)

- Example: insert 35

put it in
place:

```
      20                     20
   16     12             16      12
  5   8               5    8   35
```

sift it up:

```
      20              20                    35
   16     12       16     35           16       20
  5   8   35     5   8   12           5   8    12
```
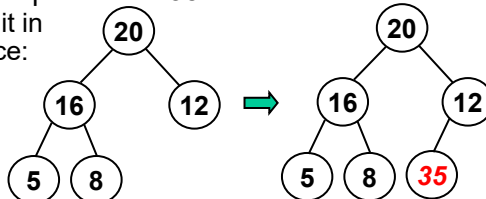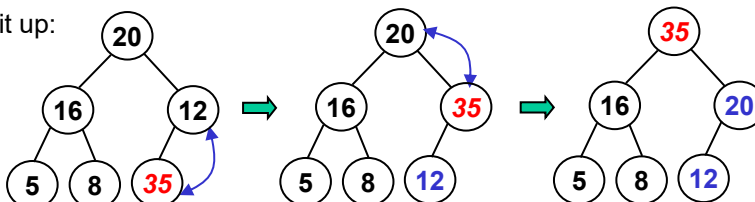
remove() Method

```
public T remove() {
    // check for empty heap goes here
    T toRemove = contents[0];

    contents[0] = contents[numItems - 1];
    contents[numItems - 1] = null;
    numItems--;
    siftDown(0);

    return toRemove;
}
```



```
   28                    5                    20
  /  \                  / \                  /  \
 20   12               20  12               16   12
 / \  /               /  \                 / \
16  8 5              16   8               5   8
```

0	1	2	3	4	5
28	20	12	16	8	5

numItems: **6**
toRemove: **28**

0	1	2	3	4	5
5	20	12	16	8	null

numItems: **5**
toRemove: **28**

0	1	2	3	4	5
20	16	12	5	8	null

numItems: **5**
toRemove: **28**
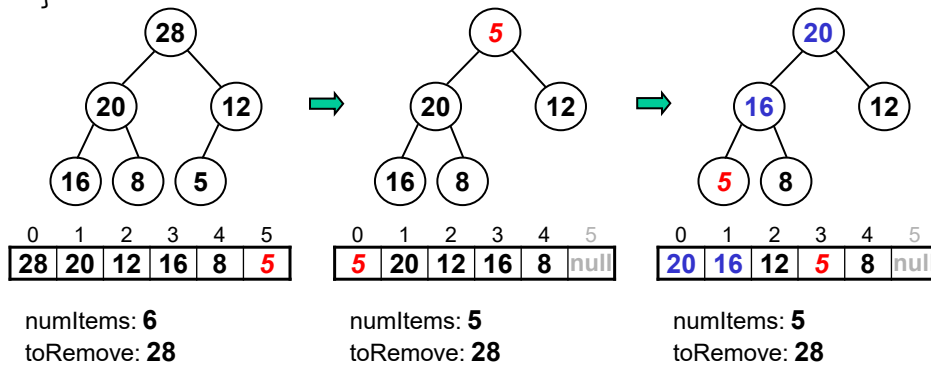
Extra practice: After calling remove() on this heap, what will it look like?


```
        35
       /  \
      26   23
     / \   / \
    3  18 8   7
```

A.
```
      26
     /  \
    23   18
   / \   /
  3   8 7
```

B.
```
      26
     /  \
    23    8
   / \   /
  3  18 7
```

C.
```
      26
     /  \
    18   23
   / \   /
  3   7 8
```
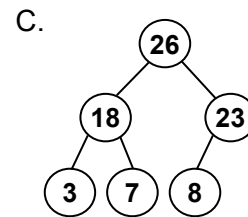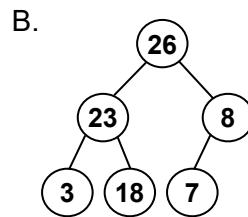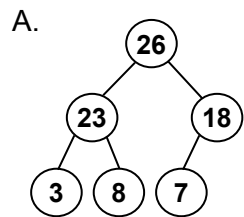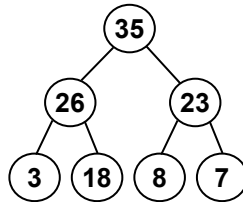
D. none of these

If Time Permits: `siftDown()` Method

```
private void siftDown(int i) {     // assume i = 0
    T toSift = contents[i];

    int parent = i;
    int child = 2 * parent + 1;
    while (child < numItems) {
        // If the right child is bigger, set child to be its index.
        if (child < numItems - 1  &&
            contents[child].compareTo(contents[child + 1]) < 0) {
            child = child + 1;
        }
        if (toSift.compareTo(contents[child]) >= 0) {
            break;  // we're done
        }
        // Move child up and move down one level in the tree.
        contents[parent] = contents[child];
        parent = child;
        child = 2 * parent + 1;
    }

    contents[parent] = toSift;
}
```

* We don't actually swap items. We put the sifted item in place at the end.

toSift: **7**

parent	child
0	1
1	3
1	4
4	9

```
   0    1    2    3    4    5
 ┌────┬────┬────┬────┬────┬────┐
 │ 26 │ 18 │ 23 │ 15 │ 7  │ 10 │
 └────┴────┴────┴────┴────┴────┘
```
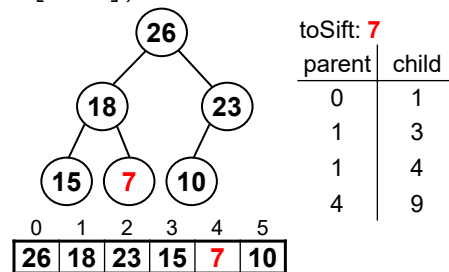
`insert()` Method

```
public void insert(T item) {
    if (numItems == contents.length) {
        // code to grow the array goes here…
    }

    contents[numItems] = item;
    siftUp(numItems);
    numItems++;
}
```

```
   0    1    2    3    4   5
 ┌────┬────┬────┬────┬────┬────┐
 │ 20 │ 16 │ 12 │ 5  │ 8  │    │
 └────┴────┴────┴────┴────┴────┘
```

numItems: **5**
item: **35**

```
   0    1    2    3    4   5
 ┌────┬────┬────┬────┬────┬────┐
 │ 20 │ 16 │ 12 │ 5  │ 8  │ 35 │
 └────┴────┴────┴────┴────┴────┘
```

numItems: **5**
item: **35**

```
   0    1    2    3    4    5
 ┌────┬────┬────┬────┬────┬────┐
 │ 35 │ 16 │ 20 │ 5  │ 8  │ 12 │
 └────┴────┴────┴────┴────┴────┘
```

numItems: **6**
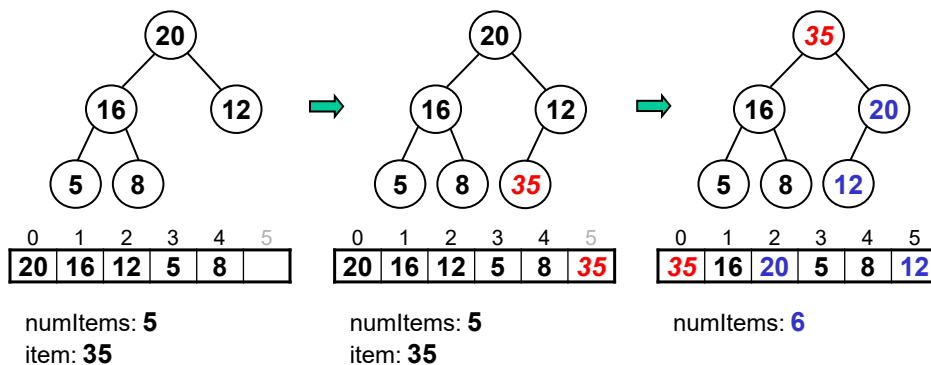
Time Complexity of a Heap

```
         ( 5 )
        /     \
     (16)     ( 8)
     /  \     /  \
  (14)(20) ( 1)(26)
```
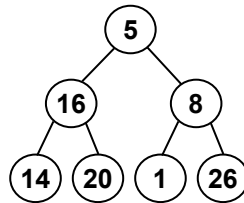
* A heap containing n items has a height $<= \log_2 n$. Why?

* Thus, removal and insertion are both $O(\log n)$.
    * remove: go down at most $\log_2 n$ levels when sifting down;
              do a constant number of operations per level
    * insert: go up at most $\log_2 n$ levels when sifting up;
              do a constant number of operations per level

* This means we can use a heap for a $O(\log n)$-time priority queue.

Using a Heap for a Priority Queue

* Recall: a *priority queue* (PQ) is a collection in which each item has an associated number known as a *priority*.
    * ("Jean Morrison", 10), ("Robert Brown", 15),
      ("Dave Sullivan", 5)
    * use a higher priority for items that are "more important"

* To implement a PQ using a heap:
    * order the items in the heap according to their priorities
        * every item in the heap will have a priority >= its children
        * the highest priority item will be in the root node
    * get the highest priority item by calling `heap.remove()`!

* For this to work, we need a "wrapper" class for items that we put in the priority queue.
    * will group together an item with its priority
    * with a `compareTo()` method that compares priorities!

A Class for Items in a Priority Queue

```java
public class PQItem implements Comparable<PQItem> {
    // group an arbitrary object with a priority
    private Object data;
    private int priority;
    ...

    public int compareTo(PQItem other) {
        // error-checking goes here…
        return (priority - other.priority);
    }
}
```

* Example: `PQItem item = new PQItem("Dave Sullivan", 5);`

* Its `compareTo()` compares `PQItems` based on their priorities.

* `item1.compareTo(item2)` returns:
  * a negative integer if `item1` has a lower priority than `item2`
  * a positive integer if `item1` has a higher priority than `item2`
  * 0 if they have the same priority

Using a Heap for a Priority Queue
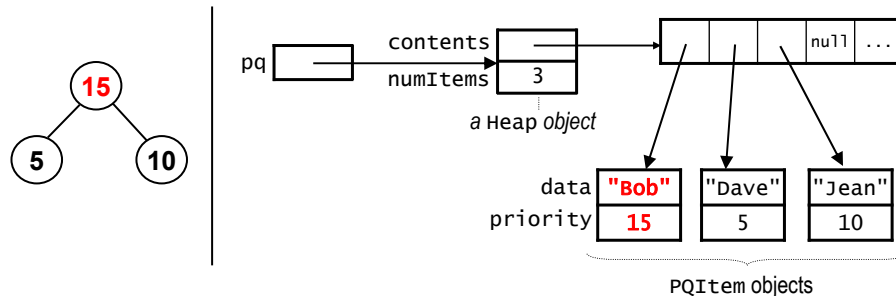
* Sample client code:
```java
Heap<PQItem> pq = new Heap<PQItem>(50);
pq.insert(new PQItem("Dave", 5));
pq.insert(new PQItem("Jean", 10));
pq.insert(new PQItem("Bob", 15));

PQItem mostImportant = pq.remove(); // will get Bob!
```

Using a Heap to Sort an Array

- Recall selection sort: it repeatedly finds the smallest remaining element and swaps it into place:

0	1	2	3	4	5	6
5	16	8	14	20	1	26

0	1	2	3	4	5	6
1	16	8	14	20	5	26

0	1	2	3	4	5	6
1	*5*	8	14	20	16	26

  …

- It isn't efficient, because it performs a linear scan to find the smallest remaining element ($O(n)$ steps per scan).

- Heapsort is a sorting algorithm that repeatedly finds the *largest* remaining element and puts it in place.

- It *is* efficient, because it turns the array into a heap.
  - it can find/remove the largest remaining in $O(\log n)$ steps!

Converting an Arbitrary Array to a Heap

- To convert an array (call it `contents`) with n items to a heap:
  1. start with the parent of the last element:
     `contents[i]`, where `i = ((n – 1) – 1)/2 = (n – 2)/2`
  2. sift down `contents[i]` and all elements to its left

- Example:

0	1	2	3	4	5	6
5	16	8	14	20	1	26

- Last element's parent = `contents[(7 – 2)/2]` = `contents[2]`. Sift it down:
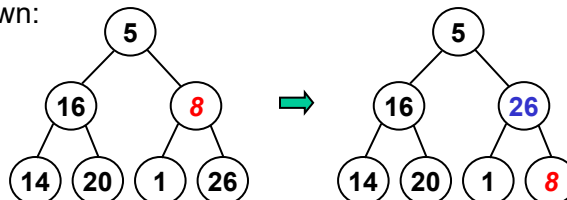
Converting an Array to a Heap (cont.)

- Next, sift down `contents[1]`:
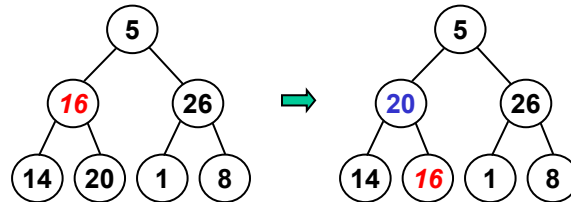
- Finally, sift down `contents[0]`:
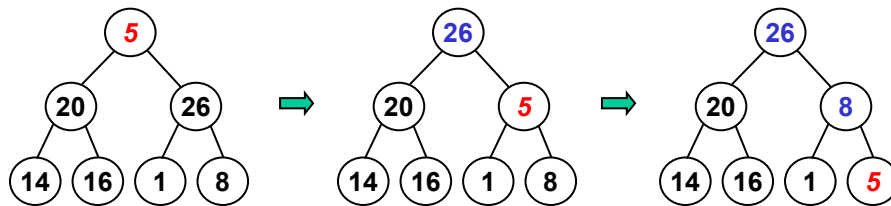
Creating a Heap from an Array

```
public class Heap<T extends Comparable<T>> {
    private T[] contents;
    private int numItems;
    ...
    public Heap(T[] arr) {
        // Note that we don't copy the array!
        contents = arr;
        numItems = arr.length;
        makeHeap();
    }

    private void makeHeap() {
        int last = contents.length - 1;
        int parentOfLast = (last - 1)/2;
        for (int i = parentOfLast; i >= 0; i--) {
            siftDown(i);
        }
    }
    ...
}
```

Heapsort

```java
public static <T extends Comparable<T>> void
heapSort(T[] arr) {
    // Turn the array into a max-at-top heap.
    Heap<T> heap = new Heap<T>(arr);

    int endUnsorted = arr.length - 1;
    while (endUnsorted > 0) {
        // Get the largest remaining element and put it
        // at the end of the unsorted portion of the array.
        T largestRemaining = heap.remove();
        arr[endUnsorted] = largestRemaining;

        endUnsorted--;
    }
}
```

- We define a *generic method*, with a type variable in the method header. It goes right before the method's return type.

- T is a placeholder for the type of the array – and the heap's items.
  - can be any type T that implements Comparable<T>.

Heapsort Example

- Sort the following array:

0	1	2	3	4	5	6
13	6	45	10	3	22	5

- Here's the corresponding complete tree:
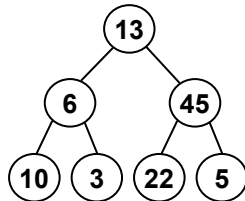
- Begin by converting it to a heap:
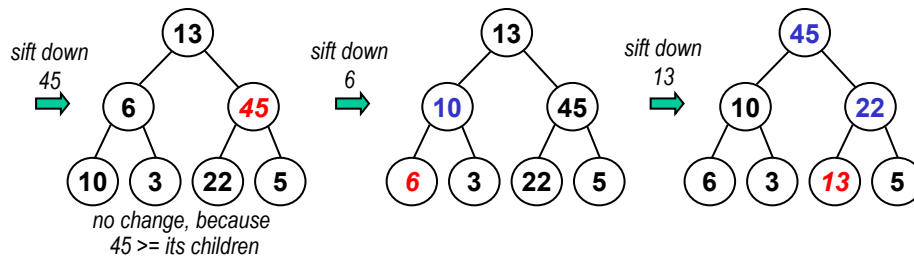
Heapsort Example (cont.)

- Here's the heap in both tree and array forms:
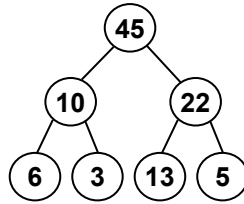
0	1	2	3	4	5	6
45	10	22	6	3	13	5

endUnsorted: **6**

- We begin looping:

```
while (endUnsorted > 0) {
    // Get the largest remaining element and put it
    // at the end of the unsorted portion of the array.
    largestRemaining = heap.remove();
    arr[endUnsorted] = largestRemaining;

    endUnsorted--;
}
```

Heapsort Example (cont.)

- Here's the heap in both tree and array forms:

0	1	2	3	4	5	6
45	10	22	6	3	13	5

endUnsorted: **6**

- Remove the largest item and put it in place:

remove() copies 45; moves 5 to root

toRemove: **45**

remove() sifts down 5; returns 45

0	1	2	3	4	5	6
22	10	13	6	3	5	5

endUnsorted: **6**
largestRemaining: **45**

heapSort() puts 45 in place; decrements endUnsorted

0	1	2	3	4	5	6
22	10	13	6	3	5	45

endUnsorted: **5**

Heapsort Example (cont.)

copy 22;
move 5
to root
➡

⑤ 22

(10) (13)

(6) (3) (5)

toRemove: **22**

sift down 5;
return 22
➡

13

(10) (5)

(6) (3)

0	1	2	3	4	5	6
13	10	*5*	6	3	5	45

endUnsorted: **5**
largestRemaining: **22**

put 22 in place;
decrement endUnsorted
➡

(13)

(10) (5)

(6) (3)

0	1	2	3	**4**	5	6
13	10	5	6	3	22	45

endUnsorted: **4**

---

copy 13;

move ___
to root
➡

(13)

(10) (5)

(6) (3)

toRemove: **13**

➡

( )

( ) ( )

( )

0	1	2	3	4	5	6
					22	45

endUnsorted: **4**
largestRemaining: **13**

put 13 in place;
decrement
➡

( )

( ) ( )

( )

0	1	2	**3**	4	5	6
			13		22	45

endUnsorted: **3**

Heapsort Example (cont.)

copy 10;
move 3
to root

sift down 3;
return 10

put 10 in place;
decrement

toRemove: **10**

0	1	2	3	4	5	6
6	***3***	**5**	**3**	**13**	**22**	**45**

endUnsorted: **3**
largestRemaining: **10**

0	1	**2**	3	4	5	6
6	**3**	**5**	**10**	**13**	**22**	**45**

endUnsorted: **2**

---

copy 6;
move 5
to root

sift down 5;
return 6

put 6 in place;
decrement

toRemove: **6**

0	1	2	3	4	5	6
5	**3**	**5**	**10**	**13**	**22**	**45**

endUnsorted: **2**
largestRemaining: **6**

0	**1**	2	3	4	5	6
5	**3**	**6**	**10**	**13**	**22**	**45**

endUnsorted: **1**

Heapsort Example (cont.)

copy 5;
move 3
to root

sift down 3;
return 5

put 5 in place;
decrement

toRemove: **5**

0	1	2	3	4	5	6
3	**3**	**6**	**10**	**13**	**22**	**45**

endUnsorted: **1**
largestRemaining: **5**

0	1	2	3	4	5	6
3	**5**	**6**	**10**	**13**	**22**	**45**

endUnsorted: **0**

- And now we terminate the loop:
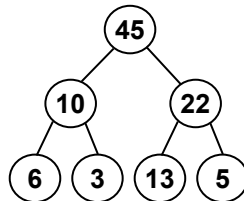
```
while (endUnsorted > 0) {
    // Get the largest remaining element and put it
    // at the end of the unsorted portion of the array.
    largestRemaining = heap.remove();
    arr[endUnsorted] = largestRemaining;

    endUnsorted--;
}
```

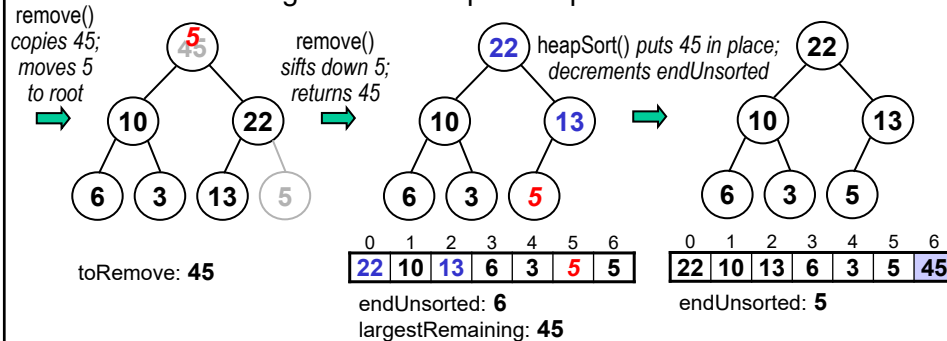# Time Complexity of Heapsort

```
        ( 5 )
       /     \
    (16)     ( 8 )
    /   \    /   \
 (14) (20)(1)  (26)
```
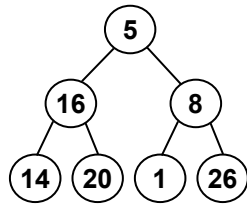
- Time complexity of creating a heap from an array?

- Time complexity of sorting the array?

- Thus, total time complexity = ?

How Does Heapsort Compare?

algorithm	best case	avg case	worst case	extra memory
selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$ worst: $O(n)$
mergesort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$

- Heapsort matches mergesort for the best worst-case time complexity, but it has better space complexity.

- Insertion sort is still best for arrays that are almost sorted.
  - heapsort will scramble an almost sorted array before sorting it!

- Quicksort is still typically fastest in the average case.

End-of-Semester Lessons

- Object-oriented programming allows us to capture the abstractions in the programs that we write.
  - creates reusable building blocks
  - key concepts: encapsulation, inheritance, polymorphism

- Abstract data types allow us to organize and manipulate collections of data.
  - a given ADT can be implemented in different ways
  - fundamental building blocks: arrays, linked nodes

- Efficiency matters when dealing with large collections of data.
  - some solutions can be *much* faster or more space efficient
  - what's the best data structure/algorithm for *your* workload?
    - example: sorting an almost sorted collection

End-of-Semester Lessons (cont.)

- Use the tools in your toolbox!
  - interfaces, generic data structures
  - lists/stacks/queues, trees, heaps, hash tables
  - recursion, recursive backtracking, divide-and-conquer

- Use built-in/provided collections/interfaces:
  - `java.util.ArrayList<T>` (implements `List<T>`)
  - `java.util.LinkedList<T>` (implements `List<T>` and `Queue<T>`)
  - `java.util.Stack<T>`
  - `java.util.TreeMap<K, V>` (a balanced search tree) ⎫ implement
  - `java.util.HashMap<K, V>` (a hash table) ⎬ `Map<K, V>`
  - `java.util.PriorityQueue<T>` (a heap) ⎭

- But use them intelligently!
  - ex: `LinkedList` maintains a reference to the last node in the list
  - `list.add(item, n)` will add `item` to the end in O(n) time
  - `list.addLast(item)` will add `item` to the end in O(1) time!

Practice: After turning this array into a heap, what will it look like?

```
  0   1   2   3   4
  6 | 4 | 18| 11| 8
```

A.
```
  0   1   2   3   4
 18 | 11| 8 | 6 | 4
```

B.
```
  0   1   2   3   4
 18 | 11| 6 | 4 | 8
```

C.
```
  0   1   2   3   4
 11 | 6 | 18| 4 | 8
```
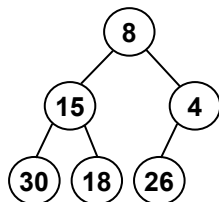
D. none of the above

Heapsort Extra Practice

- Sort the following array:
```
  0   1   2   3   4   5
  8 | 15| 4 | 30| 18| 26
```

- Here's the corresponding complete tree:

- What does it look like after we turn it into a heap?

- What do the heap and the array look like after each repetition of the loop in heapsort?