More With Java

Errors and IO

Tim Jackman

BU Summer Challenge

July 9th, 2025

Grab a rubber duck from the box

Errors in Java

We all make mistakes

Yesterday I said the Boolean type was called `bool`...

It's actually `boolean`

`bool` is what it is called in C++

I don't even know C++?

I've fixed the slides from yesterday.

Let's get started...

Errors

- Just because Java is statically typed doesn't mean we avoid errors

Errors

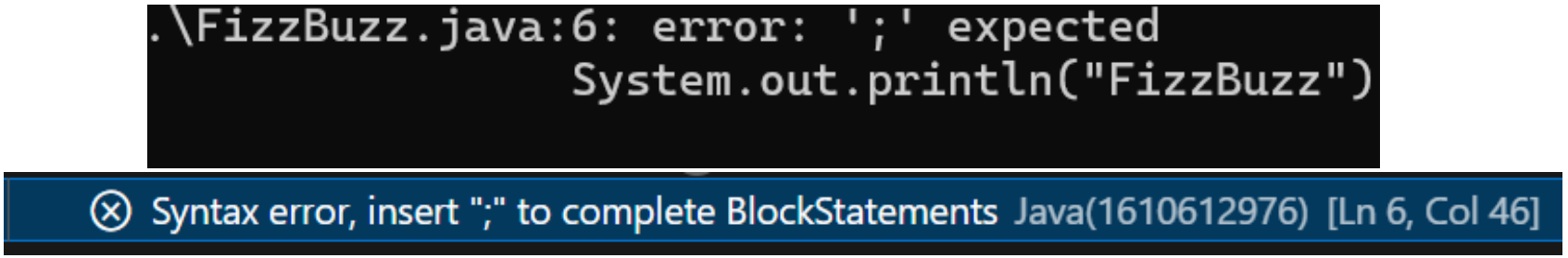
- Just because Java is statically typed doesn't mean we avoid errors
- Our files may have syntax errors (e.g. missing semi-colons, typos, etc.)

Errors

- Just because Java is statically typed doesn't mean we avoid errors
- Our files may have syntax errors (e.g. missing semi-colons, typos, etc.)
    - When we try to compile the file, the compiler will complain and try to tell us what is wrong

Errors

- Just because Java is statically typed doesn't mean we avoid errors
- Our files may have syntax errors (e.g. missing semi-colons, typos, etc.)
  - When we try to compile the file, the compiler will complain and try to tell us what is wrong



Errors

- Just because Java is statically typed doesn't mean we avoid errors
- Our files may have syntax errors (e.g. missing semi-colons, typos, etc.)
  - When we try to compile the file, the compiler will complain and try to tell us what is wrong
  - These errors are called *Compile Time Errors*

Errors

- Just because Java is statically typed doesn't mean we avoid errors
- Our files may have syntax errors (e.g. missing semi-colons, typos, etc.)
  - When we try to compile the file, the compiler will complain and try to tell us what is wrong
  - These errors are called *Compile Time Errors*
- Our program can also run into issues while being executed

Errors

- Just because Java is statically typed doesn't mean we avoid errors
- Our files may have syntax errors (e.g. missing semi-colons, typos, etc.)
  - When we try to compile the file, the compiler will complain and try to tell us what is wrong
  - These errors are called *Compile Time Errors*
- Our program can also run into issues while being executed
  - Division by 0, methods receiving bad inputs, etc.

Errors

- Just because Java is statically typed doesn't mean we avoid errors
- Our files may have syntax errors (e.g. missing semi-colons, typos, etc.)
  - When we try to compile the file, the compiler will complain and try to tell us what is wrong
  - These errors are called *Compile Time Errors*
- Our program can also run into issues while being executed
  - Division by 0, methods receiving bad inputs, etc.
  - What happens if we give the input "Hello" to our FizzBuzz.java program?

Errors

- Just because Java is statically typed doesn't mean we avoid errors
- Our files may have syntax errors (e.g. missing semi-colons, typos, etc.)
  - When we try to compile the file, the compiler will complain and try to tell us what is wrong
  - These errors are called *Compile Time Errors*
- Our program can also run into issues while being executed
  - Division by 0, methods receiving bad inputs, etc.
  - What happens if we give the input "Hello" to our FizzBuzz.java program?

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "Hello"
        at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:67)
        at java.base/java.lang.Integer.parseInt(Integer.java:564)
        at java.base/java.lang.Integer.parseInt(Integer.java:661)
        at FizzBuzz.main(FizzBuzz.java:3)
```

Errors

- Just because Java is statically typed doesn't mean we avoid errors
- Our files may have syntax errors (e.g. missing semi-colons, typos, etc.)
  - When we try to compile the file, the compiler will complain and try to tell us what is wrong
  - These errors are called *Compile Time Errors*
- Our program can also run into issues while being executed
  - Division by 0, methods receiving bad inputs, etc.
  - What happens if we give the input "Hello" to our FizzBuzz.java program?

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "Hello"
        at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:67)
        at java.base/java.lang.Integer.parseInt(Integer.java:564)
        at java.base/java.lang.Integer.parseInt(Integer.java:661)
        at FizzBuzz.main(FizzBuzz.java:3)
```

- These errors are called *Runtime Exceptions*

Errors

- Just because Java is statically typed doesn't mean we avoid errors
- Our files may have syntax errors (e.g. missing semi-colons, typos, etc.)
  - When we try to compile the file, the compiler will complain and try to tell us what is wrong
  - These errors are called *Compile Time Errors*
- Our program can also run into issues while being executed
  - Division by 0, methods receiving bad inputs, etc.
  - What happens if we give the input "Hello" to our FizzBuzz.java program?

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "Hello"
        at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:67)
        at java.base/java.lang.Integer.parseInt(Integer.java:564)
        at java.base/java.lang.Integer.parseInt(Integer.java:661)
        at FizzBuzz.main(FizzBuzz.java:3)
```

- These errors are called *Runtime Exceptions*
- Unless *handled*, exceptions will cause your program to halt

Exception Handling

- Runtime exceptions can be responded to using *try-catch* blocks

Exception Handling

- Runtime exceptions can be responded to using *try-catch* blocks

```
try {
    // Code that may throw exceptions
} catch (Exception e) {
    // This code is run if an exception is thrown
}
```

Exception Handling

- Runtime exceptions can be responded to using *try-catch* blocks

```
try {
    // Code that may throw exceptions
} catch (Exception e) {
    // This code is run if an exception is thrown
}
```

- Different exceptions will contain different information about what went wrong, accessible via their `getMessage()` method (e.g. `e.getMessage()`)

Exception Handling

- Runtime exceptions can be responded to using *try-catch* blocks

```
try {
    // Code that may throw exceptions
} catch (Exception e) {
    // This code is run if an exception is thrown
}
```

- Different exceptions will contain different information about what went wrong, accessible via their `getMessage()` method (e.g. `e.getMessage()`)
- We can catch specific kinds of exceptions by specifying them in the `catch` block

Exception Handling

- Runtime exceptions can be responded to using *try-catch* blocks

```
try {
    // Code that may throw exceptions
} catch (Exception e) {
    // This code is run if an exception is thrown
}
```

- Different exceptions will contain different information about what went wrong, accessible via their `getMessage()` method (e.g. `e.getMessage()`)
- We can catch specific kinds of exceptions by specifying them in the `catch` block

`IllegalArgumentException`	`ArithmeticException`	`NumberFormatException`
`NullPointerException`	`ArrayIndexOutOfBoundsException`	`ClassCastException`

Improving Our FizzBuzz Program

- Let's add a `try-catch` block to our FizzBuzz

Improving Our FizzBuzz Program

- Let's add a `try-catch` block to our FizzBuzz

```java
int maxNum;

try {
    maxNum = Integer.parseInt(args[0]);
} catch (Exception e) {
    System.out.println(x:"Please Input An Integer!");
    return;
}
```

Improving Our FizzBuzz Program

- Let's add a `try-catch` block to our FizzBuzz

```
int maxNum;

try {
    maxNum = Integer.parseInt(args[0]);
} catch (Exception e) {
    System.out.println(x:"Please Input An Integer!");
    return;
}
```

- Now if the user forgets to input a number or inputs something else:

Improving Our FizzBuzz Program

- Let's add a `try-catch` block to our FizzBuzz

```
int maxNum;

try {
    maxNum = Integer.parseInt(args[0]);
} catch (Exception e) {
    System.out.println(x:"Please Input An Integer!");
    return;
}
```

- Now if the user forgets to input a number or inputs something else:

```
Please Input An Integer!
```

Side Note: Scope

- We've already talked a little bit about *scope*

Side Note: Scope

- We've already talked a little bit about *scope*
- Variables are only accessible *within* the curly brackets where they are defined

Side Note: Scope

- We've already talked a little bit about *scope*
- Variables are only accessible *within* the curly brackets where they are defined

```
if ( ...  ) {
    int x = 5;
} else {
    // x is not accessible here
}
```

Side Note: Scope

- We've already talked a little bit about *scope*
- Variables are only accessible *within* the curly brackets where they are defined

```
if ( ...  ) {
    int x = 5;
} else {
    // x is not accessible here
}
```

- This is true of methods, loops, conditionals, try-catch blocks, etc.

Side Note: Scope

- We've already talked a little bit about *scope*
- Variables are only accessible *within* the curly brackets where they are defined

```
if ( ...  ) {
    int x = 5;
} else {
    // x is not accessible here
}
```

- This is true of methods, loops, conditionals, try-catch blocks, etc.
- If we want to use a variable outside of a loop/conditional, we need to declare it before

Side Note: Scope

- We've already talked a little bit about *scope*
- Variables are only accessible *within* the curly brackets where they are defined
- This is true of methods, loops, conditionals, try-catch blocks, etc.
- If we want to use a variable outside of a loop/conditional, we need to declare it before

```
int x;
if ( ...  )  {
    x = 5;
} else {
    // x is accessible here
}
```

Throwing Exceptions

- We can make our own code throw exceptions using the `throw` keyword

Throwing Exceptions

- We can make our own code throw exceptions using the `throw` keyword

```java
public static void printBobNTimes(int n) {
    if (n <= 0) {
        throw new IllegalArgumentException("n must be greater than 0, receiv
    }
```

Throwing Exceptions

- We can make our own code throw exceptions using the `throw` keyword

```java
public static void printBobNTimes(int n) {
    if (n <= 0) {
        throw new IllegalArgumentException("n must be greater than 0, receiv
    }
```

```
Exception in thread "main" java.lang.IllegalArgumentException: n must be greater
        at FizzBuzz.printBobNTimes(FizzBuzz.java:8)
        at FizzBuzz.main(FizzBuzz.java:3)
```

Logical Errors

- Not all errors in our program cause it to crash, sometimes our program just doesn't do what we want

Logical Errors

- Not all errors in our program cause it to crash, sometimes our program just doesn't do what we want
- Logical errors or *bugs* are mistakes in code that cause it to produce incorrect results

Logical Errors

- Not all errors in our program cause it to crash, sometimes our program just doesn't do what we want
- Logical errors or *bugs* are mistakes in code that cause it to produce incorrect results
- Often they are hard to spot because they're subtle

Logical Errors

- Not all errors in our program cause it to crash, sometimes our program just doesn't do what we want
- Logical errors or *bugs* are mistakes in code that cause it to produce incorrect results
- Often they are hard to spot because they're subtle

```
public int average(int a, int b) {
    return a + b / 2;
}
```

Why do we call them bugs?

- American engineers have called small malfunctions *bugs* at least as far back as Thomas Edison

Why do we call them bugs?

- American engineers have called small malfunctions *bugs* at least as far back as Thomas Edison
- The term in relation to computers was popularized by mathematician, computer scientist, and Navy Admiral Grace Hopper

Why do we call them bugs?

- American engineers have called small malfunctions *bugs* at least as far back as Thomas Edison
- The term in relation to computers was popularized by mathematician, computer scientist, and Navy Admiral Grace Hopper

Admiral Grace Hopper

Why do we call them bugs?

- American engineers have called small malfunctions *bugs* at least as far back as Thomas Edison
- The term in relation to computers was popularized by mathematician, computer scientist, and Navy Admiral Grace Hopper
- While at Harvard, her and her team removed a *moth* from their computer, the first instance of "debugging!"

Admiral Grace Hopper

Why do we call them bugs?

- American engineers have called small malfunctions *bugs* at least as far back as Thomas Edison
- The term in relation to computers was popularized by mathematician, computer scientist, and Navy Admiral Grace Hopper
- While at Harvard, her and her team removed a *moth* from their computer, the first instance of "debugging!"
- Grace Hopper also was the first to conceive of *machine-independent* programming languages

Admiral Grace Hopper

Why do we call them bugs?

- American engineers have called small malfunctions *bugs* at least as far back as Thomas Edison
- The term in relation to computers was popularized by mathematician, computer scientist, and Navy Admiral Grace Hopper
- While at Harvard, her and her team removed a *moth* from their computer, the first instance of "debugging!"
- Grace Hopper also was the first to conceive of *machine-independent* programming languages
- She also created one of the first "high level" programming languages (where code is written in English-like syntax rather than binary/symbols)

Admiral Grace Hopper

Debugging Your Code

- Avoiding bugs in your code is *impossible*: you're not that good

Debugging Your Code

- Avoiding bugs in your code is *impossible*: you're not that good
- If we identify that our program doesn't work as expected we need to carefully examine our code

Debugging Your Code

- Avoiding bugs in your code is *impossible*: you're not that good
- If we identify that our program doesn't work as expected we need to carefully examine our code
- VS Code comes with a debugger, a tool that helps us find bugs, that can

Debugging Your Code

- Avoiding bugs in your code is *impossible*: you're not that good
- If we identify that our program doesn't work as expected we need to carefully examine our code
- VS Code comes with a debugger, a tool that helps us find bugs, that can
  - Pause the program at certain lines

Debugging Your Code

- Avoiding bugs in your code is *impossible*: you're not that good
- If we identify that our program doesn't work as expected we need to carefully examine our code
- VS Code comes with a debugger, a tool that helps us find bugs, that can
  - Pause the program at certain lines
  - Step through the program one line at a time, seeing what methods are called

Debugging Your Code

- Avoiding bugs in your code is *impossible*: you're not that good
- If we identify that our program doesn't work as expected we need to carefully examine our code
- VS Code comes with a debugger, a tool that helps us find bugs, that can
  - Pause the program at certain lines
  - Step through the program one line at a time, seeing what methods are called
  - See the current values of variables

Debugging Your Code

- Avoiding bugs in your code is *impossible*: you're not that good
- If we identify that our program doesn't work as expected we need to carefully examine our code
- VS Code comes with a debugger, a tool that helps us find bugs, that can
  - Pause the program at certain lines
  - Step through the program one line at a time, seeing what methods are called
  - See the current values of variables
- Later in the week we'll talk about *testing*, code that runs other to check that it works as expected

Demo Time!

Rubber Duck Debugging

- Another technique for solving bugs is known as “rubber duck” debugging

Rubber Duck Debugging

- Another technique for solving bugs is known as “rubber duck” debugging
- Comes from the observation that you often realize what’s wrong when you try to explain the problem to someone else

Rubber Duck Debugging

- Another technique for solving bugs is known as “rubber duck” debugging
- Comes from the observation that you often realize what’s wrong when you try to explain the problem to someone else
- To do the method:

Rubber Duck Debugging

- Another technique for solving bugs is known as “rubber duck” debugging
- Comes from the observation that you often realize what’s wrong when you try to explain the problem to someone else
- To do the method:
  - Take a rubber duck (or other inanimate object)

Rubber Duck Debugging

- Another technique for solving bugs is known as "rubber duck" debugging
- Comes from the observation that you often realize what's wrong when you try to explain the problem to someone else
- To do the method:
  - Take a rubber duck (or other inanimate object)
  - Explain to it what the code *is* doing

Rubber Duck Debugging

- Another technique for solving bugs is known as "rubber duck" debugging
- Comes from the observation that you often realize what's wrong when you try to explain the problem to someone else
- To do the method:
  - Take a rubber duck (or other inanimate object)
  - Explain to it what the code *is* doing
  - Explain to it what the code *should be* doing

Rubber Duck Debugging

- Another technique for solving bugs is known as "rubber duck" debugging
- Comes from the observation that you often realize what's wrong when you try to explain the problem to someone else
- To do the method:
  - Take a rubber duck (or other inanimate object)
  - Explain to it what the code *is* doing
  - Explain to it what the code *should be* doing
  - Walk through your program line by line, explaining each line

Rubber Duck Debugging

- Another technique for solving bugs is known as "rubber duck" debugging
- Comes from the observation that you often realize what's wrong when you try to explain the problem to someone else
- To do the method:
  - Take a rubber duck (or other inanimate object)
  - Explain to it what the code *is* doing
  - Explain to it what the code *should be* doing
  - Walk through your program line by line, explaining each line
- This method is also effective with *animate* things too (like friends!)

Documentation

- As projects grow in size and age, it can be hard to read and understand code, which itself can lead to bugs

Documentation

- As projects grow in size and age, it can be hard to read and understand code, which itself can lead to bugs
- Try to use descriptive names that "self-document" (e.g. `int posX` vs `int a`)

Documentation

- As projects grow in size and age, it can be hard to read and understand code, which itself can lead to bugs
- Try to use descriptive names that "self-document" (e.g. `int posX` vs `int a`)
- However, sometimes we need explicit documentation

Documentation

- As projects grow in size and age, it can be hard to read and understand code, which itself can lead to bugs
- Try to use descriptive names that "self-document" (e.g. `int posX` vs `int a`)
- However, sometimes we need explicit documentation
- We can write *comments* into our code (text that is for humans reading the code, not the compiler)

Documentation

- As projects grow in size and age, it can be hard to read and understand code, which itself can lead to bugs
- Try to use descriptive names that "self-document" (e.g. `int posX` vs `int a`)
- However, sometimes we need explicit documentation
- We can write *comments* into our code (text that is for humans reading the code, not the compiler)
- In Java, we can write single line comments using //, anything after // on a line will not be part of your program

Documentation

- As projects grow in size and age, it can be hard to read and understand code, which itself can lead to bugs
- Try to use descriptive names that "self-document" (e.g. `int posX` vs `int a`)
- However, sometimes we need explicit documentation
- We can write *comments* into our code (text that is for humans reading the code, not the compiler)
- In Java, we can write single line comments using //, anything after // on a line will not be part of your program
- We can use /* ... */ to write longer, multiline comments, anything after /* will be commented out until the matching */

JavaDocs

- Java has official support for providing documentation to methods and classes: JavaDocs

JavaDocs

- Java has official support for providing documentation to methods and classes: JavaDocs
- A JavaDoc is wrapped in /** ... */, and has a specific format:

JavaDocs

- Java has official support for providing documentation to methods and classes: JavaDocs
- A JavaDoc is wrapped in /** ... */, and has a specific format:
  - The first part of a JavaDoc is the *description* and it is where you describe what the class or method does

JavaDocs

- Java has official support for providing documentation to methods and classes: JavaDocs
- A JavaDoc is wrapped in /** ... */, and has a specific format:
  - The first part of a JavaDoc is the *description* and it is where you describe what the class or method does
  - The second part of a JavaDoc is a series of *tags* like @param, @return, @throws, which describe the inputs, outputs, and error behavior of that code

JavaDocs

- Java has official support for providing documentation to methods and classes: JavaDocs
- A JavaDoc is wrapped in /** ... */, and has a specific format:
  - The first part of a JavaDoc is the *description* and it is where you describe what the class or method does
  - The second part of a JavaDoc is a series of *tags* like @param, @return, @throws, which describe the inputs, outputs, and error behavior of that code
- When we hover over a type or method in VS Code, its JavaDoc will be previewed (even our custom JavaDocs!)

JavaDocs

- Java has official support for providing documentation to methods and classes: JavaDocs
- A JavaDoc is wrapped in /** ... */, and has a specific format:
  - The first part of a JavaDoc is the *description* and it is where you describe what the class or method does
  - The second part of a JavaDoc is a series of *tags* like @param, @return, @throws, which describe the inputs, outputs, and error behavior of that code
- When we hover over a type or method in VS Code, its JavaDoc will be previewed (even our custom JavaDocs!)
- In practice, not every method needs a JavaDoc (if everything is clearly named and straightforward)

IO in Java

Basic IO

- We have seen basic IO using `System.out.println()` and command line arguments

Basic IO

- We have seen basic IO using `System.out.println()` and command line arguments
- What if we want a better way to supply inputs (how can we enter 100s of numbers?)

Basic IO

- We have seen basic IO using `System.out.println()` and command line arguments
- What if we want a better way to supply inputs (how can we enter 100s of numbers?)
- What if we want to save the output of our program?

Basic IO

- We have seen basic IO using `System.out.println()` and command line arguments
- What if we want a better way to supply inputs (how can we enter 100s of numbers?)
- What if we want to save the output of our program?
- How can we make our program interactive?

Scanner Class

- We can use the built-in Scanner reference type to achieve better IO

Scanner Class

- We can use the built-in Scanner reference type to achieve better IO
- The Scanner class allows us to read files and to parse inputs into the command prompt during execution

Scanner Class

- We can use the built-in Scanner reference type to achieve better IO
- The Scanner class allows us to read files and to parse inputs into the command prompt during execution
- To use the Scanner in our code we first *import* it

Scanner Class

- We can use the built-in Scanner reference type to achieve better IO
- The Scanner class allows us to read files and to parse inputs into the command prompt during execution
- To use the Scanner in our code we first *import* it
- At the top of our file, we write `import java.util.Scanner;`

Scanner Class

- We can use the built-in Scanner reference type to achieve better IO
- The Scanner class allows us to read files and to parse inputs into the command prompt during execution
- To use the Scanner in our code we first *import* it
- At the top of our file, we write `import java.util.Scanner;`
  - This tells the compiler that when we write "Scanner" in our code we mean the class defined in the file java.util.Scanner.java

Scanner Class

- We can use the built-in Scanner reference type to achieve better IO
- The Scanner class allows us to read files and to parse inputs into the command prompt during execution
- To use the Scanner in our code we first *import* it
- At the top of our file, we write `import java.util.Scanner;`
  - This tells the compiler that when we write "Scanner" in our code we mean the class defined in the file java.util.Scanner.java
  - The compiler automatically imports all the built-in files and classes defined in the package (folder) java.lang

Scanner Class

- We can use the built-in Scanner reference type to achieve better IO
- The Scanner class allows us to read files and to parse inputs into the command prompt during execution
- To use the Scanner in our code we first *import* it
- At the top of our file, we write `import java.util.Scanner;`
  - This tells the compiler that when we write "Scanner" in our code we mean the class defined in the file java.util.Scanner.java
  - The compiler automatically imports all the built-in files and classes defined in the package (folder) java.lang
  - Any java file in the same folder is also automatically imported

Scanner Class

- We can use the built-in Scanner reference type to achieve better IO
- The Scanner class allows us to read files and to parse inputs into the command prompt during execution
- To use the Scanner in our code we first *import* it
- At the top of our file, we write `import java.util.Scanner;`
  - This tells the compiler that when we write "Scanner" in our code we mean the class defined in the file java.util.Scanner.java
  - The compiler automatically imports all the built-in files and classes defined in the package (folder) java.lang
  - Any java file in the same folder is also automatically imported
- Before we can instantiate the Scanner, we need the File

Loading a File

- Java has a built-in File type which we can load with `import java.io.File;`

Loading a File

- Java has a built-in File type which we can load with `import java.io.File;`
- Given that our folder contains the file `moby-dick.txt`, we can load it into the program with:

Loading a File

- Java has a built-in File type which we can load with `import java.io.File;`
- Given that our folder contains the file `moby-dick.txt`, we can load it into the program with:

```
File inputFile = new File("moby-dick.txt");
```

Loading a File

- Java has a built-in File type which we can load with `import java.io.File;`
- Given that our folder contains the file `moby-dick.txt`, we can load it into the program with:

```
File inputFile = new File("moby-dick.txt");
```

- Instances of reference types are called *objects* and objects are created using special methods called *constructors*

Loading a File

- Java has a built-in File type which we can load with `import java.io.File;`
- Given that our folder contains the file `moby-dick.txt`, we can load it into the program with:

```
File inputFile = new File("moby-dick.txt");
```

- Instances of reference types are called *objects* and objects are created using special methods called *constructors*
- The File class has a constructor that takes in the desired filepath as a String

Creating the Scanner

- We now can try to instantiate the Scanner by supplying its constructor with the file

Creating the Scanner

- We now can try to instantiate the Scanner by supplying its constructor with the file

```
Scanner scanner = new Scanner(inputFile);
```

Creating the Scanner

- We now can try to instantiate the Scanner by supplying its constructor with the file

```
Scanner scanner = new Scanner(inputFile);
```

- However, Java will complain: `Unhandled exception type FileNotFoundException`

Creating the Scanner

- We now can try to instantiate the Scanner by supplying its constructor with the file

```
Scanner scanner = new Scanner(inputFile);
```

- However, Java will complain: `Unhandled exception type FileNotFoundException`
  - Some Runtime Exceptions are special *checked* exceptions that must be handled

Creating the Scanner

- We now can try to instantiate the Scanner by supplying its constructor with the file

```
Scanner scanner = new Scanner(inputFile);
```

- However, Java will complain: `Unhandled exception type FileNotFoundException`
  - Some Runtime Exceptions are special *checked* exceptions that must be handled
- We will wrap our code into a try-catch block

Scanner Example

```java
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;

public class IOExample {
    public static void main(String[] args) {
        File inputFile = new File(pathname:"moby-dick.txt");
        try {
            Scanner fileReader = new Scanner(inputFile);
        } catch (FileNotFoundException e) {
            System.out.println(x:"File 'moby-dick.txt' was not found.");
        }
    }
}
```

Using the Scanner

- The scanner breaks the text into individual *tokens* and then traverses these tokens using its methods

Using the Scanner

- The scanner breaks the text into individual *tokens* and then traverses these tokens using its methods
- Tokens that are read are the output of the method calls

Using the Scanner

- The scanner breaks the text into individual *tokens* and then traverses these tokens using its methods
- Tokens that are read are the output of the method calls
- `next()` reads the next token, `nextLine()` reads until a new line, `nextInt()` reads the next token *as an int*

Using the Scanner

- The scanner breaks the text into individual *tokens* and then traverses these tokens using its methods
- Tokens that are read are the output of the method calls
- `next()` reads the next token, `nextLine()` reads until a new line, `nextInt()` reads the next token *as an int*
- Once the scanner advances past a token it cannot go back

Using the Scanner

- The scanner breaks the text into individual *tokens* and then traverses these tokens using its methods
- Tokens that are read are the output of the method calls
- `next()` reads the next token, `nextLine()` reads until a new line, `nextInt()` reads the next token *as an int*
- Once the scanner advances past a token it cannot go back
- The scanner tries to read *even if there are no more tokens* leading to exceptions

Using the Scanner

- The scanner breaks the text into individual *tokens* and then traverses these tokens using its methods
- Tokens that are read are the output of the method calls
- `next()` reads the next token, `nextLine()` reads until a new line, `nextInt()` reads the next token *as an int*
- Once the scanner advances past a token it cannot go back
- The scanner tries to read *even if there are no more tokens* leading to exceptions
- We can check first using `hasNext()`, `hasNextLine()`, `hasNextInt()`

Scanner Example

```java
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;

public class IOExample {
    public static void main(String[] args) {
        System.out.println(System.getProperty(key:"user.dir"));
        File inputFile = new File(pathname:"moby-dick.txt");
        try {
            Scanner fileReader = new Scanner(inputFile);
            int numLines = 0;

            while (fileReader.hasNext()) {
                fileReader.nextLine();
                numLines = numLines + 1;
            }
            System.out.println(numLines);
            fileReader.close();

        } catch (FileNotFoundException e) {
            System.out.println(x:"File 'moby-dick.txt' was not found.");
        }
```

Reading from the Terminal

- The scanner can also read from the terminal:

Reading from the Terminal

- The scanner can also read from the terminal:

```
Scanner scanner = new Scanner(System.in);
```

Reading from the Terminal

- The scanner can also read from the terminal:

```
Scanner scanner = new Scanner(System.in);
```

```java
Scanner scanner = new Scanner(System.in);
System.out.println(x:"Enter an int!");
int input = scanner.nextInt();
System.out.println("You input: " + Integer.toString(input));
```

Reading from the Terminal

- The scanner can also read from the terminal:

```
Scanner scanner = new Scanner(System.in);
```

```
Scanner scanner = new Scanner(System.in);
System.out.println(x:"Enter an int!");
int input = scanner.nextInt();
System.out.println("You input: " + Integer.toString(input));
```

```
Enter an int!
10
You input: 10
```

Closing the Scanner

- Typically Java automatically clears up resources once it is no longer needed (e.g. after exiting a function it forgets any temporary variables)

Closing the Scanner

- Typically Java automatically clears up resources once it is no longer needed (e.g. after exiting a function it forgets any temporary variables)
- Once a Scanner is open, it will not be cleared up until manually closed with `.close()`

Closing the Scanner

- Typically Java automatically clears up resources once it is no longer needed (e.g. after exiting a function it forgets any temporary variables)
- Once a Scanner is open, it will not be cleared up until manually closed with `.close()`
- This will also automatically close the file it has loaded

Closing the Scanner

- Typically Java automatically clears up resources once it is no longer needed (e.g. after exiting a function it forgets any temporary variables)
- Once a Scanner is open, it will not be cleared up until manually closed with `.close()`
- This will also automatically close the file it has loaded

Closing System.In

Do not close a Scanner reading System.in, it cannot be reopen!

Closing the Scanner

- Typically Java automatically clears up resources once it is no longer needed (e.g. after exiting a function it forgets any temporary variables)
- Once a Scanner is open, it will not be cleared up until manually closed with `.close()`
- This will also automatically close the file it has loaded

Closing System.In

Do not close a Scanner reading System.in, it cannot be reopen!

- Important for making perfomant programs and other languages require more of this manual resource management

Creating and Writing to a File

- We can create a new file by instanting a File object for it and using the `createNewFile()` command

Creating and Writing to a File

- We can create a new file by instanting a File object for it and using the `createNewFile()` command
- This throws the checked exception `java.io.IOException`

Creating and Writing to a File

- We can create a new file by instanting a File object for it and using the `createNewFile()` command
- This throws the checked exception `java.io.IOException`
- We can write to a file using the java.io.FileWriter class which works very similar to Scanner

Creating and Writing to a File

- We can create a new file by instanting a File object for it and using the `createNewFile()` command
- This throws the checked exception `java.io.IOException`
- We can write to a file using the java.io.FileWriter class which works very similar to Scanner
- `write(String str)` command writes the String to the file

Creating and Writing to a File

- We can create a new file by instanting a File object for it and using the `createNewFile()` command
- This throws the checked exception `java.io.IOException`
- We can write to a file using the java.io.FileWriter class which works very similar to Scanner
- `write(String str)` command writes the String to the file
- FileWriter also needs to be closed!

Creating and Writing to a File

- We can create a new file by instanting a File object for it and using the `createNewFile()` command
- This throws the checked exception `java.io.IOException`
- We can write to a file using the java.io.FileWriter class which works very similar to Scanner
- `write(String str)` command writes the String to the file
- FileWriter also needs to be closed!

Challenge

Create a program that creates a new file and writes "Hello World!" in it! As an extra challenge, use System.in to allow the user to choose the file name! Also try handling the case where the file already exists