Intro to Java

Our First Foray into a “Real” Programming Language

Tim Jackman

BU Summer Challenge

July 8th, 2025

Java Overview

- Developed in 1995, Java is one of the most popular programming languages in use today

Java Overview

- Developed in 1995, Java is one of the most popular programming languages in use today
- Java is designed to be platform independent, any computer with Java installed can run Java programs without needing to worry about the specific OS (Windows, Mac, Linux)

Java Overview

- Developed in 1995, Java is one of the most popular programming languages in use today
- Java is designed to be platform independent, any computer with Java installed can run Java programs without needing to worry about the specific OS (Windows, Mac, Linux)
- Java is an "Object-Oriented Language"

Java Overview

- Developed in 1995, Java is one of the most popular programming languages in use today
- Java is designed to be platform independent, any computer with Java installed can run Java programs without needing to worry about the specific OS (Windows, Mac, Linux)
- Java is an "Object-Oriented Language"
- Java is verbose: keywords (public, private, static), explicit types, semicolons and curly brackets

Java Overview

- Developed in 1995, Java is one of the most popular programming languages in use today
- Java is designed to be platform independent, any computer with Java installed can run Java programs without needing to worry about the specific OS (Windows, Mac, Linux)
- Java is an "Object-Oriented Language"
- Java is verbose: keywords (public, private, static), explicit types, semicolons and curly brackets
- Android apps are built using Java or languages that build on it (Kotlin)

A Basic Java File

- Java code is written in .java files

A Basic Java File

- Java code is written in .java files
- Code is written in a *public class* that is named the same as the file

A Basic Java File

- Java code is written in .java files
- Code is written in a *public class* that is named the same as the file
- The body of the class consists of *fields* (variables) and *methods* (functions)

A Basic Java File

- Java code is written in .java files
- Code is written in a *public class* that is named the same as the file
- The body of the class consists of *fields* (variables) and *methods* (functions)
- Blocks of code are grouped using curly braces { . . . }

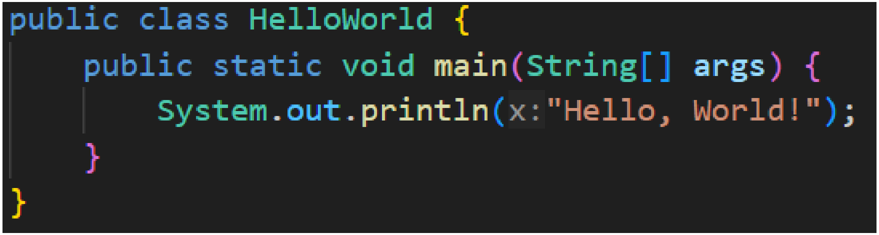
A Basic Java File

- Java code is written in .java files
- Code is written in a *public class* that is named the same as the file
- The body of the class consists of *fields* (variables) and *methods* (functions)
- Blocks of code are grouped using curly braces { . . . }
- We indent code nested in curly brackets as part of good Java style

A Basic Java File

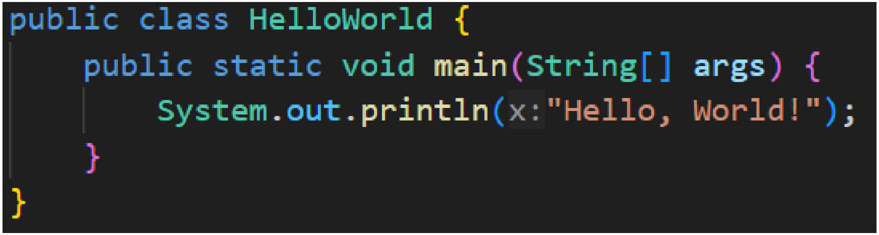
- Java code is written in .java files
- Code is written in a *public class* that is named the same as the file
- The body of the class consists of *fields* (variables) and *methods* (functions)
- Blocks of code are grouped using curly braces { . . . }
- We indent code nested in curly brackets as part of good Java style
- Statements (lines of code that perform an action) end using semicolons ;. This is required.

Example Class

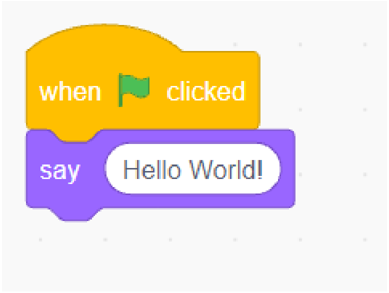


```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println(x:"Hello, World!");
    }
}
```

Example Class

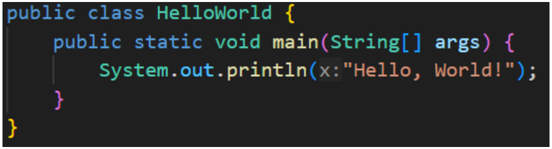


```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println(x:"Hello, World!");
    }
}
```



Breaking It Down

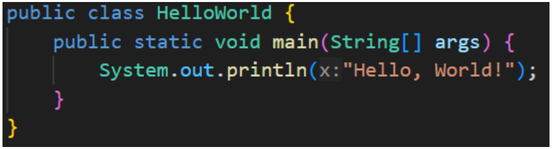
- This code defines the public class HelloWorld, and is found in the file HelloWorld.java



```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println(x:"Hello, World!");
    }
}
```

Breaking It Down

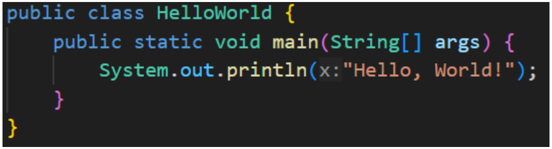
- This code defines the public class HelloWorld, and is found in the file HelloWorld.java
- The body of the class consists of a single method called "main"



```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println(x:"Hello, World!");
    }
}
```

Breaking It Down

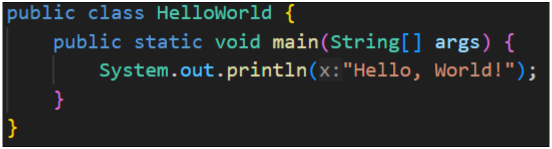
- This code defines the public class HelloWorld, and is found in the file HelloWorld.java
- The body of the class consists of a single method called "main"
    - It has two *keywords* "public" and "static" and it's *return type* is "void"



```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println(x:"Hello, World!");
    }
}
```

Breaking It Down

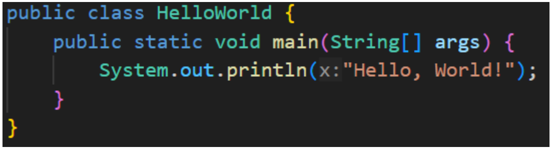
- This code defines the public class HelloWorld, and is found in the file HelloWorld.java
- The body of the class consists of a single method called "main"
  - It has two *keywords* "public" and "static" and it's *return type* is "void"
  - It takes in one *argument* (input) called *args* that has type String[]



```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println(x:"Hello, World!");
    }
}
```

Breaking It Down

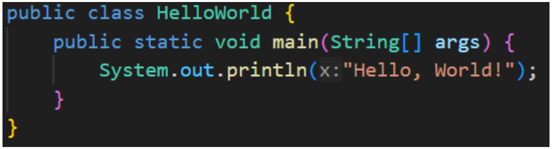
- This code defines the public class HelloWorld, and is found in the file HelloWorld.java
- The body of the class consists of a single method called “main”
  - It has two *keywords* “public” and “static” and it’s *return type* is “void”
  - It takes in one *argument* (input) called *args* that has type String[]
  - We’ll see what all this means later
- When a java file is run the class’s main method is what is run



```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println(x:"Hello, World!");
    }
}
```

Breaking It Down

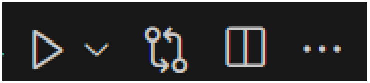
- This code defines the public class HelloWorld, and is found in the file HelloWorld.java
- The body of the class consists of a single method called “main”
  - It has two *keywords* “public” and “static” and it’s *return type* is “void”
  - It takes in one *argument* (input) called *args* that has type String[]
  - We’ll see what all this means later
- When a java file is run the class’s main method is what is run
- `System.out.println(‘‘Hello World!’’);` tells the computer to print out the line “Hello World” into the output (terminal)



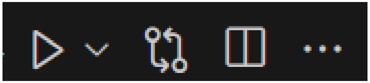
```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println(x:"Hello, World!");
    }
}
```

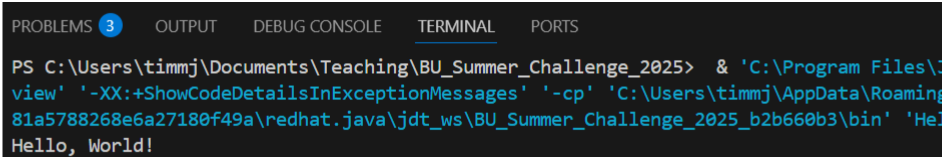
How do we run our program?

Running Code in Visual Code Studio



Running Code in Visual Code Studio





```
PS C:\Users\timmj\Documents\Teaching\BU_Summer_Challenge_2025>  & 'C:\Program Files\J
view' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\timmj\AppData\Roaming
81a5788268e6a27180f49a\redhat.java\jdt_ws\BU_Summer_Challenge_2025_b2b660b3\bin' 'Hel
Hello, World!
```

Running Code from the Terminal

- You can also run Java programs from the Terminal/Command Line/Command Prompt

Running Code from the Terminal

- You can also run Java programs from the Terminal/Command Line/Command Prompt
- First, navigate to the folder containing the file using the `cd` command

Running Code from the Terminal

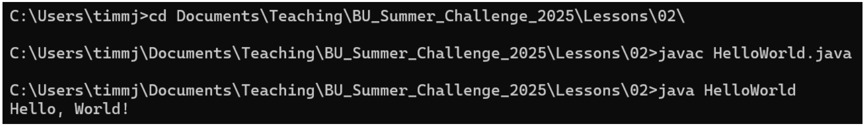
- You can also run Java programs from the Terminal/Command Line/Command Prompt
- First, navigate to the folder containing the file using the `cd` command
- Run `javac FileName.java` to *compile* the Java code into .class files

Running Code from the Terminal

- You can also run Java programs from the Terminal/Command Line/Command Prompt
- First, navigate to the folder containing the file using the `cd` command
- Run `javac FileName.java` to *compile* the Java code into .class files
- Run `java FileName` to *run* your program

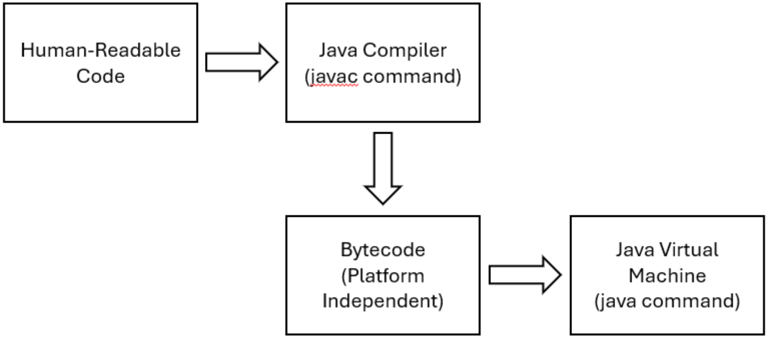
Running Code from the Terminal

- You can also run Java programs from the Terminal/Command Line/Command Prompt
- First, navigate to the folder containing the file using the `cd` command
- Run `javac FileName.java` to *compile* the Java code into .class files
- Run `java FileName` to *run* your program



```
C:\Users\timmj>cd Documents\Teaching\BU_Summer_Challenge_2025\Lessons\02\

C:\Users\timmj\Documents\Teaching\BU_Summer_Challenge_2025\Lessons\02>javac HelloWorld.java

C:\Users\timmj\Documents\Teaching\BU_Summer_Challenge_2025\Lessons\02>java HelloWorld
Hello, World!
```

Running a Java Program


Human-Readable Code
Java Compiler (javac command)
Bytecode (Platform Independent)
Java Virtual Machine (java command)

How do we give inputs to our program?

- The easiest way to give inputs to your program is with *command line arguments*

How do we give inputs to our program?

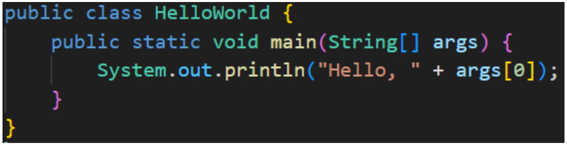
- The easiest way to give inputs to your program is with *command line arguments*
- Every string (separated by a space) after the filename is supplied to your program as an input

How do we give inputs to our program?

- The easiest way to give inputs to your program is with *command line arguments*
- Every string (separated by a space) after the filename is supplied to your program as an input
- The *first* input can be accessed by your code with `args[0]`, the *second* is given by `args[1]`, etc.

How do we give inputs to our program?

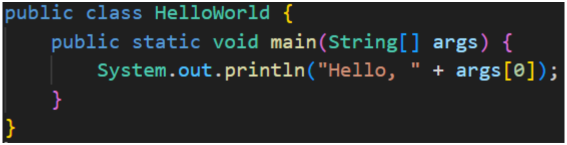
- The easiest way to give inputs to your program is with *command line arguments*
- Every string (separated by a space) after the filename is supplied to your program as an input
- The *first* input can be accessed by your code with `args[0]`, the *second* is given by `args[1]`, etc.



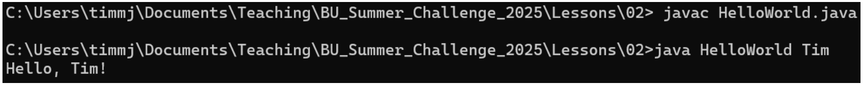
```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, " + args[0]);
    }
}
```

How do we give inputs to our program?

- The easiest way to give inputs to your program is with *command line arguments*
- Every string (separated by a space) after the filename is supplied to your program as an input
- The *first* input can be accessed by your code with `args[0]`, the *second* is given by `args[1]`, etc.



```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, " + args[0]);
    }
}
```



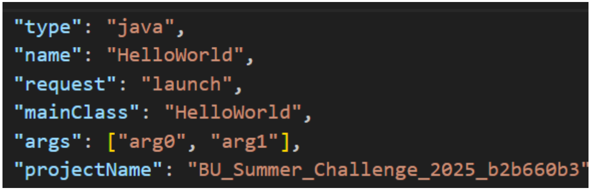
```
C:\Users\timmj\Documents\Teaching\BU_Summer_Challenge_2025\Lessons\02> javac HelloWorld.java

C:\Users\timmj\Documents\Teaching\BU_Summer_Challenge_2025\Lessons\02>java HelloWorld Tim
Hello, Tim!
```

Command Line Args in VS Code

- Command Line Args can be supplied to VS Code in the `launch.json` files

Command Line Args in VS Code

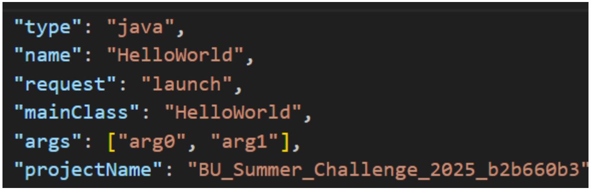
- Command Line Args can be supplied to VS Code in the `launch.json` files



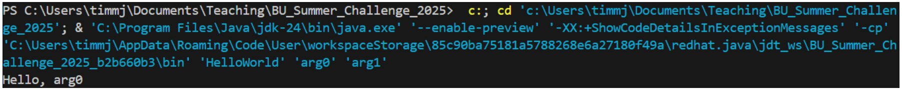
```
"type": "java",
"name": "HelloWorld",
"request": "launch",
"mainClass": "HelloWorld",
"args": ["arg0", "arg1"],
"projectName": "BU_Summer_Challenge_2025_b2b660b3"
```

Command Line Args in VS Code

- Command Line Args can be supplied to VS Code in the `launch.json` files



```
"type": "java",
"name": "HelloWorld",
"request": "launch",
"mainClass": "HelloWorld",
"args": ["arg0", "arg1"],
"projectName": "BU_Summer_Challenge_2025_b2b660b3"
```



```
PS C:\Users\timmj\Documents\Teaching\BU_Summer_Challenge_2025>  c:; cd 'c:\Users\timmj\Documents\Teaching\BU_Summer_Challen
ge_2025'; & 'C:\Program Files\Java\jdk-24\bin\java.exe' '--enable-preview' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp'
'C:\Users\timmj\AppData\Roaming\Code\User\workspaceStorage\85c90ba75181a5788268e6a27180f49a\redhat.java\jdt_ws\BU_Summer_Ch
allenge_2025_b2b660b3\bin' 'HelloWorld' 'arg0' 'arg1'
Hello, arg0
```

Types in Java

- Java is a *statically typed* languages

Types in Java

- Java is a *statically typed* languages
    - The type of every variable and the output of any function must be *declared*

Types in Java

- Java is a *statically typed* languages
    - The type of every variable and the output of any function must be *declared*
    - Java does not automatically cast between similar types

Types in Java

- Java is a *statically typed* languages
  - The type of every variable and the output of any function must be *declared*
  - Java does not automatically cast between similar types
- This allows the compiler to check for any type errors before we ever run any code

Primitive Types

- Java has a few special types called primitives

Primitive Types

- Java has a few special types called primitives
- These are the only types that are “lowercase”

Primitive Types

- Java has a few special types called primitives
- These are the only types that are “lowercase”
- Most represent different sets of numbers:

Primitive Types

- Java has a few special types called primitives
- These are the only types that are “lowercase”
- Most represent different sets of numbers:
  - `byte`: integers between -128 and 127

Primitive Types

- Java has a few special types called primitives
- These are the only types that are "lowercase"
- Most represent different sets of numbers:
    - `byte`: integers between -128 and 127
    - `short`: integers between -32,768 and 32,767

Primitive Types

- Java has a few special types called primitives
- These are the only types that are "lowercase"
- Most represent different sets of numbers:
    - `byte`: integers between -128 and 127
    - `short`: integers between -32,768 and 32,767
    - `int`: integers between -2^{-31} and $2^{31} - 1$

Primitive Types

- Java has a few special types called primitives
- These are the only types that are "lowercase"
- Most represent different sets of numbers:
  - `byte`: integers between -128 and 127
  - `short`: integers between -32,768 and 32,767
  - `int`: integers between -2^{-31} and $2^{31} - 1$
  - `long`: integers between -2^{-63} and $2^{63} - 1$

Primitive Types

- Java has a few special types called primitives
- These are the only types that are “lowercase”
- Most represent different sets of numbers:
  - `byte`: integers between -128 and 127
  - `short`: integers between -32,768 and 32,767
  - `int`: integers between -2^{-31} and $2^{31} - 1$
  - `long`: integers between -2^{-63} and $2^{63} - 1$
  - `float`: *32 bit floating point* decimals (arithmetic is not precise)

Primitive Types

- Java has a few special types called primitives
- These are the only types that are “lowercase”
- Most represent different sets of numbers:
  - `byte`: integers between -128 and 127
  - `short`: integers between -32,768 and 32,767
  - `int`: integers between -2^{-31} and $2^{31} - 1$
  - `long`: integers between -2^{-63} and $2^{63} - 1$
  - `float`: *32 bit floating point* decimals (arithmetic is not precise)
  - `double`: *64 bit floating point* decimals (arithmetic is not precise)

Primitive Types

- Java has a few special types called primitives
- These are the only types that are "lowercase"
- Most represent different sets of numbers:
  - `byte`: integers between -128 and 127
  - `short`: integers between -32,768 and 32,767
  - `int`: integers between -2^{-31} and $2^{31} - 1$
  - `long`: integers between -2^{-63} and $2^{63} - 1$
  - `float`: *32 bit floating point* decimals (arithmetic is not precise)
  - `double`: *64 bit floating point* decimals (arithmetic is not precise)
- `boolean`: Booleans (`true` and `false`)

Primitive Types

- Java has a few special types called primitives
- These are the only types that are "lowercase"
- Most represent different sets of numbers:
    - `byte`: integers between -128 and 127
    - `short`: integers between -32,768 and 32,767
    - `int`: integers between -2^{-31} and $2^{31} - 1$
    - `long`: integers between -2^{-63} and $2^{63} - 1$
    - `float`: *32 bit floating point* decimals (arithmetic is not precise)
    - `double`: *64 bit floating point* decimals (arithmetic is not precise)
- `boolean`: Booleans (`true` and `false`)
- `char`: Unicode character (individual letters and symbols), wrapped in single quotes (e.g. 'a')

Numerical Operators

- The basic arithmetic operators are $+, -, *, /$, and % (modulo)

Numerical Operators

- The basic arithmetic operators are $+, -, *, /$, and % (modulo)
- Be careful with / on ints, you get an integer back (the result is rounded towards 0, e.g. 2/3 = -2/3 = 0)

Numerical Operators

- The basic arithmetic operators are $+, -, *, /$, and % (modulo)
- Be careful with / on ints, you get an integer back (the result is rounded towards 0, e.g. $2/3 =$ -$2/3 = 0$)
- Number comparison operators are `<, <=, ==, !=, >=, >`

Boolean Operators

- The AND, OR, and NOT operators are &&, ||, and ! respectively

Boolean Operators

- The AND, OR, and NOT operators are &&, ||, and ! respectively
- Java also has & and | for AND and OR but these are less efficient

Boolean Operators

- The AND, OR, and NOT operators are &&, ||, and ! respectively
- Java also has & and | for AND and OR but these are less efficient
  - If you write `p & q` in Java, it always evaluates both *p* and then *q* and then compute `p & q`

Boolean Operators

- The AND, OR, and NOT operators are &&, ||, and ! respectively
- Java also has & and | for AND and OR but these are less efficient
  - If you write `p & q` in Java, it always evaluates both p and then q and then compute `p & q`
  - If you write `p && q` in Java, it will evaluate p, and if it is false, it *short circuits* and evaluates the & to false

Boolean Operators

- The AND, OR, and NOT operators are &&, ||, and ! respectively
- Java also has & and | for AND and OR but these are less efficient
  - If you write `p & q` in Java, it always evaluates both *p* and then *q* and then compute `p & q`
  - If you write `p && q` in Java, it will evaluate *p*, and if it is false, it *short circuits* and evaluates the & to false
- Remember || is *inclusive or*: it evaluates to true if *at least one* of its inputs is true

Boolean Operators

- The AND, OR, and NOT operators are &&, ||, and ! respectively
- Java also has & and | for AND and OR but these are less efficient
    - If you write `p & q` in Java, it always evaluates both *p* and then *q* and then compute `p & q`
    - If you write `p && q` in Java, it will evaluate *p*, and if it is false, it *short circuits* and evaluates the & to false
- Remember || is *inclusive or*: it evaluates to true if *at least one* of its inputs is true
- If you want *exclusive or* (exactly one of the two is true), you can use ^or ! =

Strings

- Every other type in Java is a *Reference Type*, and will be capitalized using UpperCamelCase

Strings

- Every other type in Java is a *Reference Type*, and will be capitalized using UpperCamelCase
- Most Reference Types are *classes* (we will explore what that means later in the seminar)

Strings

- Every other type in Java is a *Reference Type*, and will be capitalized using UpperCamelCase
- Most Reference Types are *classes* (we will explore what that means later in the seminar)
- However, the String class has special built in support and are therefore like pseudo-primitives

Strings

- Every other type in Java is a *Reference Type*, and will be capitalized using UpperCamelCase
- Most Reference Types are *classes* (we will explore what that means later in the seminar)
- However, the String class has special built in support and are therefore like pseudo-primitives
- Strings are essentially ordered lists of chars, and are written by wrapping chars with double quotes: "hello"

Strings

- Every other type in Java is a *Reference Type*, and will be capitalized using UpperCamelCase
- Most Reference Types are *classes* (we will explore what that means later in the seminar)
- However, the String class has special built in support and are therefore like pseudo-primitives
- Strings are essentially ordered lists of chars, and are written by wrapping chars with double quotes: “hello”
- There are a bunch of String methods to work with Strings:

Strings

- Every other type in Java is a *Reference Type*, and will be capitalized using UpperCamelCase
- Most Reference Types are *classes* (we will explore what that means later in the seminar)
- However, the String class has special built in support and are therefore like pseudo-primitives
- Strings are essentially ordered lists of chars, and are written by wrapping chars with double quotes: “hello”
- There are a bunch of String methods to work with Strings:

```
length()                               concat(String str)     indexOf(String str)
replace(char oldChar, char newChar)    contains(String str)   compareTo(String str)
substring(int beginIndex, int endIndex) equals(String str)    charAt(int index)
```

Strings

- Every other type in Java is a *Reference Type*, and will be capitalized using UpperCamelCase
- Most Reference Types are *classes* (we will explore what that means later in the seminar)
- However, the String class has special built in support and are therefore like pseudo-primitives
- Strings are essentially ordered lists of chars, and are written by wrapping chars with double quotes: “hello”
- There are a bunch of String methods to work with Strings:

```
length()                                concat(String str)    indexOf(String str)
replace(char oldChar, char newChar)     contains(String str)  compareTo(String str)
substring(int beginIndex, int endIndex) equals(String str)    charAt(int index)
```

- To call these methods, you use the *dot operator* on a specific String, followed by the method name: `‘‘Hello’’.length()` evaluates to 5

Strings

- Every other type in Java is a *Reference Type*, and will be capitalized using UpperCamelCase
- Most Reference Types are *classes* (we will explore what that means later in the seminar)
- However, the String class has special built in support and are therefore like pseudo-primitives
- Strings are essentially ordered lists of chars, and are written by wrapping chars with double quotes: "hello"
- There are a bunch of String methods to work with Strings:

```
length()                              concat(String str)    indexOf(String str)
replace(char oldChar, char newChar)   contains(String str)  compareTo(String str)
substring(int beginIndex, int endIndex) equals(String str)  charAt(int index)
```

- To call these methods, you use the *dot operator* on a specific String, followed by the method name: `‘‘Hello’’.length()` evaluates to 5
- Java indexes starting at 0: the first character of "Hello" is at the 0th position: `‘‘Hello’’.charAt(0)` evaluates to 'H'.

Strings

- Every other type in Java is a *Reference Type*, and will be capitalized using UpperCamelCase
- Most Reference Types are *classes* (we will explore what that means later in the seminar)
- However, the String class has special built in support and are therefore like pseudo-primitives
- Strings are essentially ordered lists of chars, and are written by wrapping chars with double quotes: “hello”
- There are a bunch of String methods to work with Strings:

`length()`	`concat(String str)`	`indexOf(String str)`
`replace(char oldChar, char newChar)`	`contains(String str)`	`compareTo(String str)`
`substring(int beginIndex, int endIndex)`	`equals(String str)`	`charAt(int index)`

- To call these methods, you use the *dot operator* on a specific String, followed by the method name: `‘‘Hello’’.length()` evaluates to 5
- Java indexes starting at 0: the first character of “Hello” is at the 0th position: `‘‘Hello’’.charAt(0)` evaluates to ‘H’.
- You can check the Java Documentation to see all of the methods and how they work

Checking String Equality? use .equals()

Equality Testing

Do not use == to compare reference data types, use the .equals() method

Checking String Equality? use .equals()

Equality Testing

Do not use == to compare reference data types, use the .equals() method

- == will check whether the two things are taking up the same the memory address on your computer, not whether the content of them is the same

Checking String Equality? use .equals()

Equality Testing

Do not use == to compare reference data types, use the .equals() method

- == will check whether the two things are taking up the same the memory address on your computer, not whether the content of them is the same
- Because of how Java handles Strings, this is ok like 95% of the time

Checking String Equality? use .equals()

Equality Testing

Do not use == to compare reference data types, use the .equals() method

- == will check whether the two things are taking up the same the memory address on your computer, not whether the content of them is the same
- Because of how Java handles Strings, this is ok like 95% of the time
- But 5% of the time your program will be wrong and you'll be confused!

Variables and Assignments

- To create a variable we *declare* it:

Variables and Assignments

- To create a variable we *declare* it:

```
int exampleVar;
```

Variables and Assignments

- To create a variable we *declare* it:

```
int exampleVar;
```

- Since Java is *statically typed*, we must include types

Variables and Assignments

- To create a variable we *declare* it:

```
int exampleVar;
```

- Since Java is *statically typed*, we must include types
- We use = to assign values when declaring variables

Variables and Assignments

- To create a variable we *declare* it:

```
int exampleVar;
```

- Since Java is *statically typed*, we must include types
- We use = to assign values when declaring variables

```
int exampleVar = 5;
```

Variables and Assignments

- To create a variable we *declare* it:

```
int exampleVar;
```

- Since Java is *statically typed*, we must include types
- We use = to assign values when declaring variables

```
int exampleVar = 5;
```

- If we don't initially define them, variables will default to some value depending on their type (e.g. 0, false, *null*)

Variables and Assignments

- To create a variable we *declare* it:

```
int exampleVar;
```

- Since Java is *statically typed*, we must include types
- We use = to assign values when declaring variables

```
int exampleVar = 5;
```

- If we don't initially define them, variables will default to some value depending on their type (e.g. 0, false, *null*)
- We can reassign variables by using = again:

Variables and Assignments

- To create a variable we *declare* it:

```
int exampleVar;
```

- Since Java is *statically typed*, we must include types
- We use = to assign values when declaring variables

```
int exampleVar = 5;
```

- If we don't initially define them, variables will default to some value depending on their type (e.g. 0, false, *null*)
- We can reassign variables by using = again:

```
int exampleVar = 5;
exampleVar = 10;
System.out.println(exampleVar);
```

Variables and Assignments

- To create a variable we *declare* it:

```
int exampleVar;
```

- Since Java is *statically typed*, we must include types
- We use = to assign values when declaring variables

```
int exampleVar = 5;
```

- If we don't initially define them, variables will default to some value depending on their type (e.g. 0, false, *null*)
- We can reassign variables by using = again:

```
int exampleVar = 5;
exampleVar = 10;
System.out.println(exampleVar);
```

- Naming Conventions: lowerCamelCase for regular variables, UPPER_SNAKE_CASE for constants (e.g. `double PI = 3.14;`)

Converting Between Types

- While we can't change a variables type we can *cast* from one type to another

Converting Between Types

- While we can't change a variables type we can *cast* from one type to another
- Java will automatically "cast up" number primitives to the larger sets of numbers:

Converting Between Types

- While we can't change a variables type we can *cast* from one type to another
- Java will automatically "cast up" number primitives to the larger sets of numbers:

```
int exampleInt = 5;
double exampleDouble = exampleInt;
```

Converting Between Types

- While we can't change a variables type we can *cast* from one type to another
- Java will automatically "cast up" number primitives to the larger sets of numbers:

```
int exampleInt = 5;
double exampleDouble = exampleInt;
```

- We can manually "cast down" number primitives into the smaller sets of numbers:

Converting Between Types

- While we can't change a variables type we can *cast* from one type to another
- Java will automatically "cast up" number primitives to the larger sets of numbers:

```
int exampleInt = 5;
double exampleDouble = exampleInt;
```

- We can manually "cast down" number primitives into the smaller sets of numbers:

```
double exampleDouble = 5.6;
int exampleInt = (int) exampleDouble; //equals 5
```

Converting Between Types

- While we can't change a variables type we can *cast* from one type to another
- Java will automatically "cast up" number primitives to the larger sets of numbers:

```
int exampleInt = 5;
double exampleDouble = exampleInt;
```

- We can manually "cast down" number primitives into the smaller sets of numbers:

```
double exampleDouble = 5.6;
int exampleInt = (int) exampleDouble; //equals 5
```

- All Reference Types will have a built-in .toString() method to convert to String

Methods

- We have seen a number of methods (functions) so far, now let's see how to define them

Methods

- We have seen a number of methods (functions) so far, now let's see how to define them

```
public static boolean hidesBob(String str) {
   return str.toLowerCase().contains("bob");
}
```

Methods

- We have seen a number of methods (functions) so far, now let's see how to define them

```
public static boolean hidesBob(String str) {
   return str.toLowerCase().contains("bob");
}
```

- Every method starts with its *signature* which uniquely identifies it

Methods

- We have seen a number of methods (functions) so far, now let's see how to define them

```
public static boolean hidesBob(String str) {
    return str.toLowerCase().contains("bob");
}
```

- Every method starts with its *signature* which uniquely identifies it
  - The first part of the signature are any *modifiers* (e.g. `public`, `private`, `static`)

Methods

- We have seen a number of methods (functions) so far, now let's see how to define them

```
public static boolean hidesBob(String str) {
   return str.toLowerCase().contains("bob");
}
```

- Every method starts with its *signature* which uniquely identifies it
  - The first part of the signature are any *modifiers* (e.g. `public`, `private`, `static`)
  - The second part is the *return type*

Methods

- We have seen a number of methods (functions) so far, now let's see how to define them

```
public static boolean hidesBob(String str) {
   return str.toLowerCase().contains("bob");
}
```

- Every method starts with its *signature* which uniquely identifies it
  - The first part of the signature are any *modifiers* (e.g. `public`, `private`, `static`)
  - The second part is the *return type*
    - If a method returns a value it will have a `return` statement inside the body of the code
    - If a method doesn't return anything, it's return type is `void`

Methods

- We have seen a number of methods (functions) so far, now let's see how to define them

```
public static boolean hidesBob(String str) {
   return str.toLowerCase().contains("bob");
}
```

- Every method starts with its *signature* which uniquely identifies it
  - The first part of the signature are any *modifiers* (e.g. `public`, `private`, `static`)
  - The second part is the *return type*
    - If a method returns a value it will have a `return` statement inside the body of the code
    - If a method doesn't return anything, it's return type is `void`

Methods

- We have seen a number of methods (functions) so far, now let's see how to define them

```
public static boolean hidesBob(String str) {
   return str.toLowerCase().contains("bob");
}
```

- Every method starts with its *signature* which uniquely identifies it
  - The first part of the signature are any *modifiers* (e.g. `public`, `private`, `static`)
  - The second part is the *return type* (e.g. boolean, int, void)

Methods

- We have seen a number of methods (functions) so far, now let's see how to define them

```
public static boolean hidesBob(String str) {
   return str.toLowerCase().contains("bob");
}
```

- Every method starts with its *signature* which uniquely identifies it
  - The first part of the signature are any *modifiers* (e.g. `public`, `private`, `static`)
  - The second part is the *return type* (e.g. boolean, int, void)
  - The third part is the method name, we use lowerCamelCase

Methods

- We have seen a number of methods (functions) so far, now let's see how to define them

```
public static boolean hidesBob(String str) {
   return str.toLowerCase().contains("bob");
}
```

- Every method starts with its *signature* which uniquely identifies it
  - The first part of the signature are any *modifiers* (e.g. `public`, `private`, `static`)
  - The second part is the *return type* (e.g. boolean, int, void)
  - The third part is the method name, we use lowerCamelCase
  - The final is its arguments, a list of inputs (and their types) that must supplied to call the method

Methods

- We have seen a number of methods (functions) so far, now let's see how to define them

```
public static boolean hidesBob(String str) {
   return str.toLowerCase().contains("bob");
}
```

- Every method starts with its *signature* which uniquely identifies it
  - The first part of the signature are any *modifiers* (e.g. `public, private, static`)
  - The second part is the *return type* (e.g. boolean, int, void)
  - The third part is the method name, we use lowerCamelCase
  - The final is its arguments, a list of inputs (and their types) that must supplied to call the method
    - A method's inputs can only be accessed inside that specific method's body

Methods

- We have seen a number of methods (functions) so far, now let's see how to define them

```
public static boolean hidesBob(String str) {
   return str.toLowerCase().contains("bob");
}
```

- Every method starts with its *signature* which uniquely identifies it
  - The first part of the signature are any *modifiers* (e.g. `public`, `private`, `static`)
  - The second part is the *return type* (e.g. boolean, int, void)
  - The third part is the method name, we use lowerCamelCase
  - The final is its arguments, a list of inputs (and their types) that must supplied to call the method
- Two methods can have the same name but cannot have the same signature

Methods

- We have seen a number of methods (functions) so far, now let's see how to define them

Methods

- We have seen a number of methods (functions) so far, now let's see how to define them

```
public static boolean hidesBob(String str) {
   return str.toLowerCase().contains("bob");
}
```

- Between the curly brackets is a method's *body*, it is where we write our code

Methods

- We have seen a number of methods (functions) so far, now let's see how to define them

```
public static boolean hidesBob(String str) {
   return str.toLowerCase().contains("bob");
}
```

- Between the curly brackets is a method's *body*, it is where we write our code
- If the method's return type is not `void`, the body must use the `return` keyword to return a value of the correct type

Methods

- We have seen a number of methods (functions) so far, now let's see how to define them

```
public static boolean hidesBob(String str) {
   return str.toLowerCase().contains("bob");
}
```

- Between the curly brackets is a method's *body*, it is where we write our code
- If the method's return type is not `void`, the body must use the `return` keyword to return a value of the correct type
- In `void` methods we can use `return;` to halt the method (e.g. inside conditionals)

Methods

- We have seen a number of methods (functions) so far, now let's see how to define them

```
public static boolean hidesBob(String str) {
   return str.toLowerCase().contains("bob");
}
```

- Between the curly brackets is a method's *body*, it is where we write our code
- If the method's return type is not `void`, the body must use the `return` keyword to return a value of the correct type
- In `void` methods we can use `return;` to halt the method (e.g. inside conditionals)
- When we call a method, the expression has that method's type, and thus we can *curry* function calls:

Methods

- We have seen a number of methods (functions) so far, now let's see how to define them

```
public static boolean hidesBob(String str) {
    return str.toLowerCase().contains("bob");
}
```

- Between the curly brackets is a method's *body*, it is where we write our code
- If the method's return type is not `void`, the body must use the `return` keyword to return a value of the correct type
- In `void` methods we can use `return;` to halt the method (e.g. inside conditionals)
- When we call a method, the expression has that method's type, and thus we can *curry* function calls:
  - `str.toLowerCase()` is a `String`, so we can call `.contains("bob");` on it directly

Calling Methods

- We have seen how to call methods using the the dot operator

Calling Methods

- We have seen how to call methods using the the dot operator
- So far we've called *instance* methods (i.e. non-static)

Calling Methods

- We have seen how to call methods using the the dot operator
- So far we've called *instance* methods (i.e. non-static)
    - We use the dot operator on a specific *instance* of a class

Calling Methods

- We have seen how to call methods using the the dot operator
- So far we've called *instance* methods (i.e. non-static)
  - We use the dot operator on a specific *instance* of a class
  - `''Hello''.charAt(3);`

Calling Methods

- We have seen how to call methods using the the dot operator
- So far we've called *instance* methods (i.e. non-static)
  - We use the dot operator on a specific *instance* of a class
  - `‘‘Hello’’.charAt(3);`
  - Instance methods behave differently on different instances
- *static* methods do not depend on the specific instance and are for the class as a whole

Calling Methods

- We have seen how to call methods using the the dot operator
- So far we've called *instance* methods (i.e. non-static)
  - We use the dot operator on a specific *instance* of a class
  - `‘‘Hello’’.charAt(3);`
  - Instance methods behave differently on different instances
- *static* methods do not depend on the specific instance and are for the class as a whole
- static mathods often belong to utility classes that just bundle together useful functions

Calling Methods

- We have seen how to call methods using the the dot operator
- So far we've called *instance* methods (i.e. non-static)
    - We use the dot operator on a specific *instance* of a class
    - `‘‘Hello’’.charAt(3);`
    - Instance methods behave differently on different instances
- *static* methods do not depend on the specific instance and are for the class as a whole
- static mathods often belong to utility classes that just bundle together useful functions
- To call them we use the dot operator on the class name

Calling Methods

- We have seen how to call methods using the the dot operator
- So far we've called *instance* methods (i.e. non-static)
  - We use the dot operator on a specific *instance* of a class
  - `‘‘Hello’’.charAt(3);`
  - Instance methods behave differently on different instances
- *static* methods do not depend on the specific instance and are for the class as a whole
- static mathods often belong to utility classes that just bundle together useful functions
- To call them we use the dot operator on the class name
  - `Math.min(int a, int b);`

Calling Methods

- We have seen how to call methods using the the dot operator
- So far we've called *instance* methods (i.e. non-static)
  - We use the dot operator on a specific *instance* of a class
  - `‘‘Hello’’.charAt(3);`
  - Instance methods behave differently on different instances
- *static* methods do not depend on the specific instance and are for the class as a whole
- static mathods often belong to utility classes that just bundle together useful functions
- To call them we use the dot operator on the class name
  - `Math.min(int a, int b);`
- `Math` and `Integer` are some classes with useful static methods

Conditionals in Java

- Conditionals in Java work just like in Scratch

Conditionals in Java

- Conditionals in Java work just like in Scratch

```
if (conditionA) {
    //runs if conditionA is true
} else if (conditionB) {
    //runs if conditionA is false and conditionB is true
} else {
    //runs if neither are true
}
```

Conditionals in Java

- Conditionals in Java work just like in Scratch

```
if (conditionA) {
    //runs if conditionA is true
} else if (conditionB) {
    //runs if conditionA is false and conditionB is true
} else {
    //runs if neither are true
}
```

- The conditions can be boolean expressions, boolean variables, or methods that return booleans

Conditionals in Java

- Conditionals in Java work just like in Scratch

```
if (conditionA) {
    //runs if conditionA is true
} else if (conditionB) {
    //runs if conditionA is false and conditionB is true
} else {
    //runs if neither are true
}
```

- The conditions can be boolean expressions, boolean variables, or methods that return booleans
- Avoid writing conditions involving booleans like `(boolVar == true)` or `(boolVar == false)`

Conditionals in Java

- Conditionals in Java work just like in Scratch

```
if (conditionA) {
    //runs if conditionA is true
} else if (conditionB) {
    //runs if conditionA is false and conditionB is true
} else {
    //runs if neither are true
}
```

- The conditions can be boolean expressions, boolean variables, or methods that return booleans
- Avoid writing conditions involving booleans like `(boolVar == true)` or `(boolVar == false)`
  - You can just write `(boolVar)` or `(!boolVar)`

Loops in Java

- There are two basic types of loops in Java: `while` and `for` loops

Loops in Java

- There are two basic types of loops in Java: `while` and `for` loops
- `while` loops are kind of like `until` loops in Scratch but the opposite

Loops in Java

- There are two basic types of loops in Java: `while` and `for` loops
- `while` loops are kind of like `until` loops in Scratch but the opposite

```
while (conditionA) {
    //iterates if conditionA is true
}
```

Loops in Java

- There are two basic types of loops in Java: `while` and `for` loops
- `while` loops are kind of like `until` loops in Scratch but the opposite
- `for` loops are like `repeat` loops in Scratch but more general

Loops in Java

- There are two basic types of loops in Java: `while` and `for` loops
- `while` loops are kind of like `until` loops in Scratch but the opposite
- `for` loops are like `repeat` loops in Scratch but more general

```
for (statementA; statementB; statementC) {
   //iterates until statementB is false
}
```

Loops in Java

- There are two basic types of loops in Java: `while` and `for` loops
- `while` loops are kind of like `until` loops in Scratch but the opposite
- `for` loops are like `repeat` loops in Scratch but more general

```
for (statementA; statementB; statementC) {
    //iterates until statementB is false
}
```

- Before the first iteration of the loop statementA is executed: typically initiating a temporary variable that controls the loop

Loops in Java

- There are two basic types of loops in Java: `while` and `for` loops
- `while` loops are kind of like `until` loops in Scratch but the opposite
- `for` loops are like `repeat` loops in Scratch but more general

```
for (statementA; statementB; statementC) {
    //iterates until statementB is false
}
```

- Before the first iteration of the loop statementA is executed: typically initiating a temporary variable that controls the loop
- statementB is a condition. Before any iteration of the loop, statementB is evaluated and the nested code only runs if it evaluates to true

Loops in Java

- There are two basic types of loops in Java: `while` and `for` loops
- `while` loops are kind of like `until` loops in Scratch but the opposite
- `for` loops are like `repeat` loops in Scratch but more general

```
for (statementA; statementB; statementC) {
    //iterates until statementB is false
}
```

- Before the first iteration of the loop statementA is executed: typically initiating a temporary variable that controls the loop
- statementB is a condition. Before any iteration of the loop, statementB is evaluated and the nested code only runs if it evaluates to true
- At the end of a loop iteration, statementC is run: typically incrementing the temporary variable

Example Loop

```
for (int i = 0; i < 10; i++) {
    System.out.println(i);
}
```

Example Loop

```
for (int i = 0; i < 10; i++) {
    System.out.println(i);
}
```

- Typically our temporary loop variables are called i, j, k (if we're nesting loops)

Example Loop

```
for (int i = 0; i < 10; i++) {
    System.out.println(i);
}
```

- Typically our temporary loop variables are called i, j, k (if we're nesting loops)
- Temporary loop variables are removed after the loop is done, they can only be referenced in the loop (we call this the variable's *scope*)

Example Loop

```
for (int i = 0; i < 10; i++) {
    System.out.println(i);
}
```

- Typically our temporary loop variables are called i, j, k (if we're nesting loops)
- Temporary loop variables are removed after the loop is done, they can only be referenced in the loop (we call this the variable's *scope*)
- We have a lot of control here with the third statement

Challenge

- Create a FizzBuzz file that when run prints the integers 1 through 100, replacing multiples of 3 with Fizz, multiples of 5 with Buzz, and multiples of both with FizzBuzz.
- As an extra challenge, modify the program to take a command line argument for the upper bound (e.g. `java FizzBuzz 45` does 1 through 45). You'll need to look at the Integer class documentation to find a helpful static method.
- As an extra extra challenge, modify it to loop through only the even integers, and replace multiples of 6 and 10 with Fizz and Buzz (and FizzBuzz)!