COMP 1020 - Java Programming

UNIT 1

Outline

- Using Java: text editors, JDK, running code
- Naming conventions
- Data types and operators
- Java syntax
  - Variables
  - Methods
  - Conditions
  - Loops
  - Arrays

Programming in Java

- You need a text editor! To write the code!

- You have many options

  1. Very basic editors (not recommended, but they are always available):
     - Notepad (Windows)
     - nano and pico (Linux)
     - TextEdit (Mac)

Programming in Java

- You need a text editor! To write the code!

- You have many options

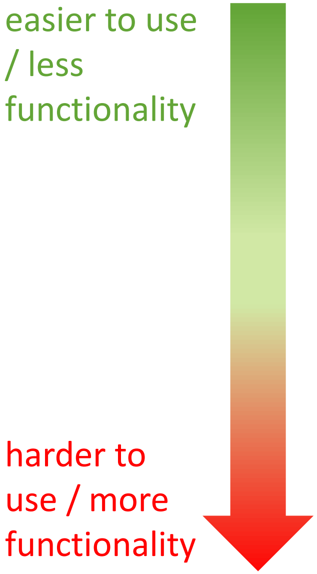
2. Lightweight-Midsized editors:
   - Notepad++ (Windows)
   - TextMate (Mac)
   - emacs, vim, Sublime Text (any platform)

Programming in Java

- You need a text editor! To write the code!

- You have many options

3. Integrated development environments (IDEs):



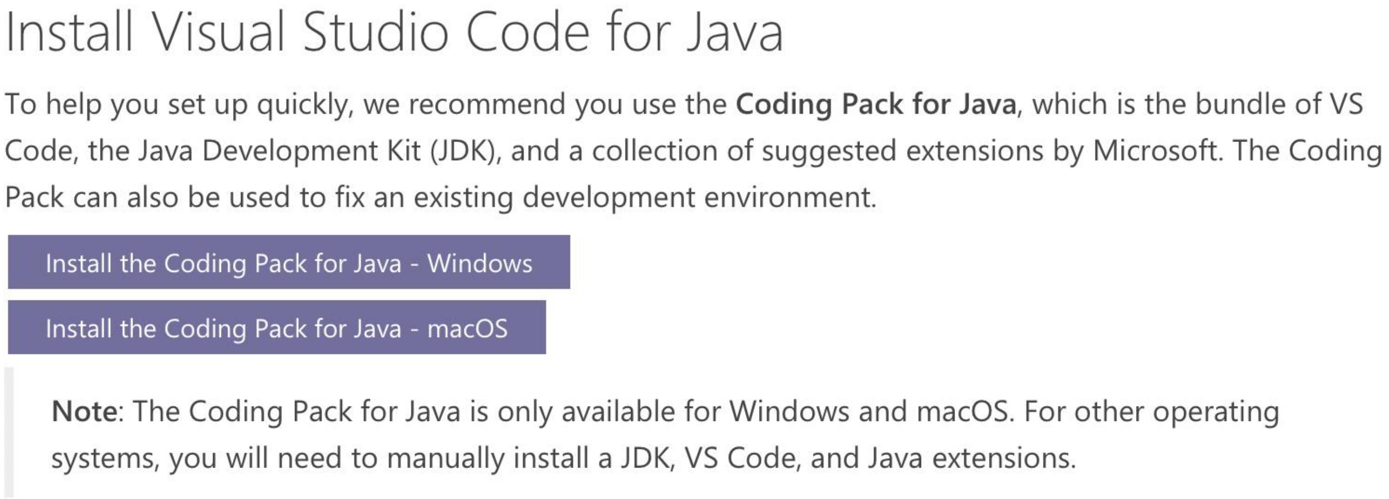
easier to use / less functionality

harder to use / more functionality

- BlueJ (https://bluej.org/)
- Geany (https://www.geany.org/)
- **Visual Studio Code** (https://code.visualstudio.com/)
- IntelliJ (https://www.jetbrains.com/idea/)
- Eclipse (https://www.eclipse.org/)

Visual Studio Code

- Visual Studio Code is the recommended environment
- Visit: https://code.visualstudio.com/docs/languages/java
- Install the ***Coding Pack for Java*** (not the other links)



this

or

this

Install Visual Studio Code for Java

To help you set up quickly, we recommend you use the **Coding Pack for Java**, which is the bundle of VS Code, the Java Development Kit (JDK), and a collection of suggested extensions by Microsoft. The Coding Pack can also be used to fix an existing development environment.

Install the Coding Pack for Java - Windows
Install the Coding Pack for Java - macOS

> **Note**: The Coding Pack for Java is only available for Windows and macOS. For other operating systems, you will need to manually install a JDK, VS Code, and Java extensions.

Visual Studio Code (“VS Code”)

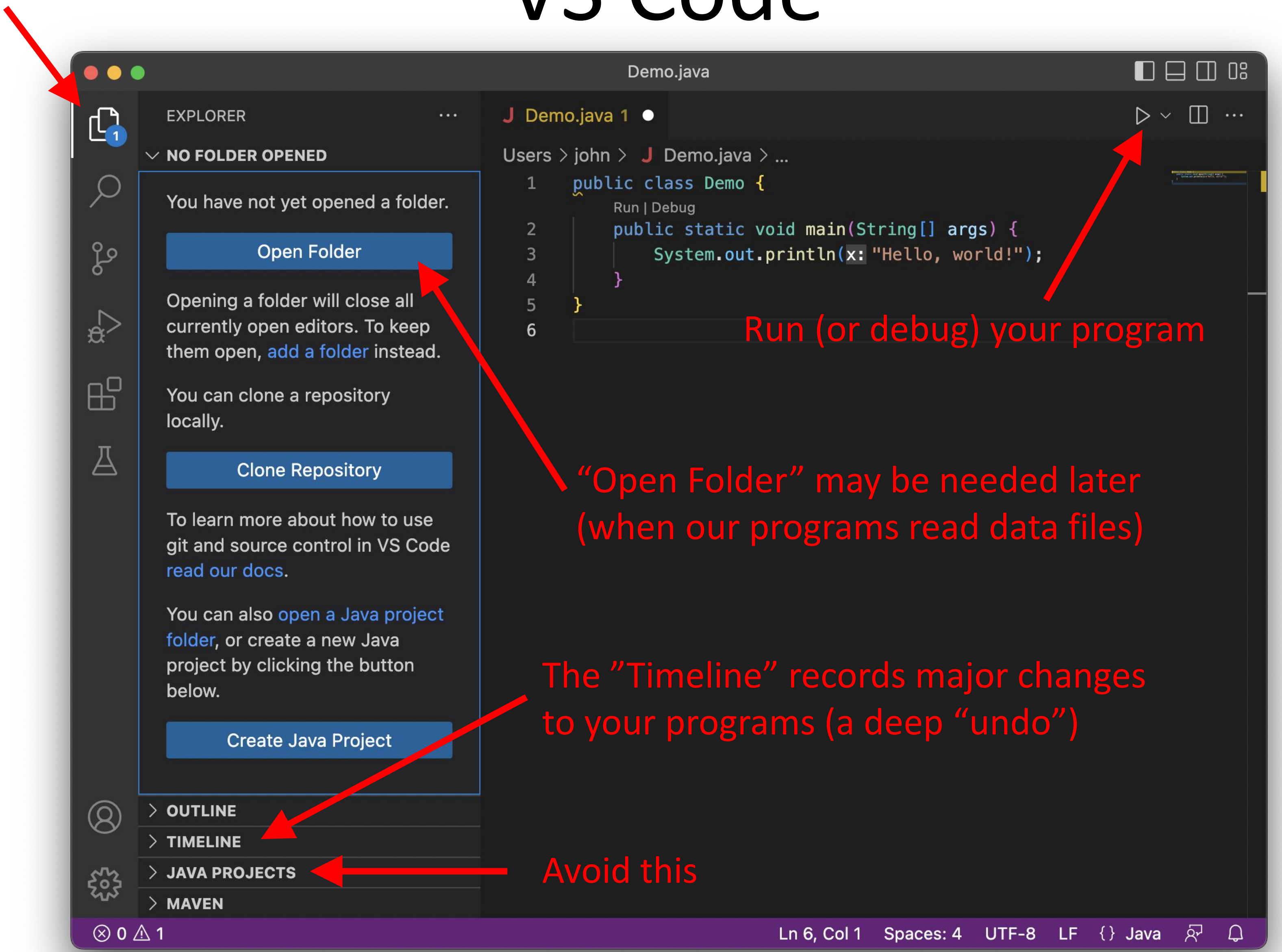
- The Coding Pack installer will first make sure Java is installed on your system, and allow you to download and install it if it is not. Once Java is installed, it will download and install VS Code.

- When you save a file with the extension `.java` VS Code will recognize it as a Java program and allow you to run it.

- Later in the course, you may want to use “Open Folder” to access related files. **Don’t** use “projects”.

VS Code

Turns the "Explorer" on and off



You have not yet opened a folder.

Opening a folder will close all currently open editors. To keep them open, add a folder instead.

You can clone a repository locally.

To learn more about how to use git and source control in VS Code read our docs.

You can also open a Java project folder, or create a new Java project by clicking the button below.

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

Run (or debug) your program

"Open Folder" may be needed later (when our programs read data files)

The "Timeline" records major changes to your programs (a deep "undo")

Avoid this

Other Environments

- Some other environments (like the command line / terminal shown on upcoming slides) require that you install a Java Development Kit (JDK).
- The “official” JDK: https://www.oracle.com/javadownload
- OpenJDK (better): https://adoptopenjdk.net
- This is already installed by the VS Coding Pack. If you use VS Code, you probably don’t need to install a JDK separately.

Programming in Java

- To actually run your code, you need the JDK (Java Development Kit)!

- Download and install JDK (not JRE) from

https://www.oracle.com/java/technologies/javase-downloads.html

- The latest release of JDK is version SE 14 (July 2020) but…

it’s now SE 15 (already)!

- … version 8 (SE 8) is still widely used (this is the version on the departmental servers, probably the one you should get)

Programming in Java

- For Windows users: after installing JDK, it is usually necessary to set the PATH Environment variable

- This allows you to run java anywhere (in a terminal) and also makes it easier for the IDE of your choice to find Java (when installing an IDE, it might also ask you for the location of Java)

- See: https://docs.oracle.com/javase/8/docs/technotes/guides/install/windows_jdk_install.html#BABGDJFH

Java execution

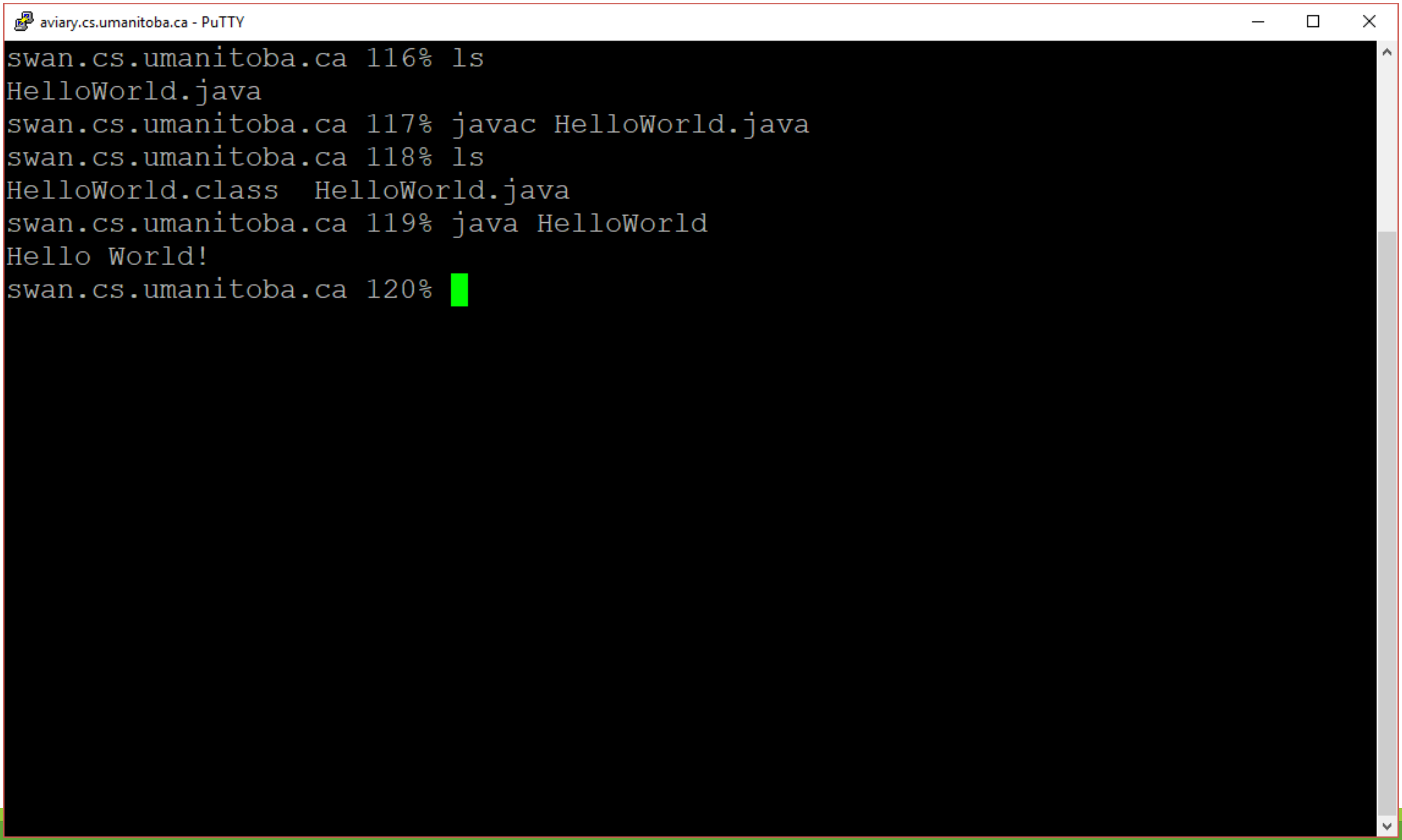
- Source code is stored in a file with the .java extension

- .java file is compiled to produce a .class file (which is a sort of generic machine language called Java Byte Code (JBC))

- The .class file is then interpreted by the Java Virtual Machine (JVM)
  - very portable: anyone with a JVM can run it on any platform
  - secure: .class file is unreadable

Java execution

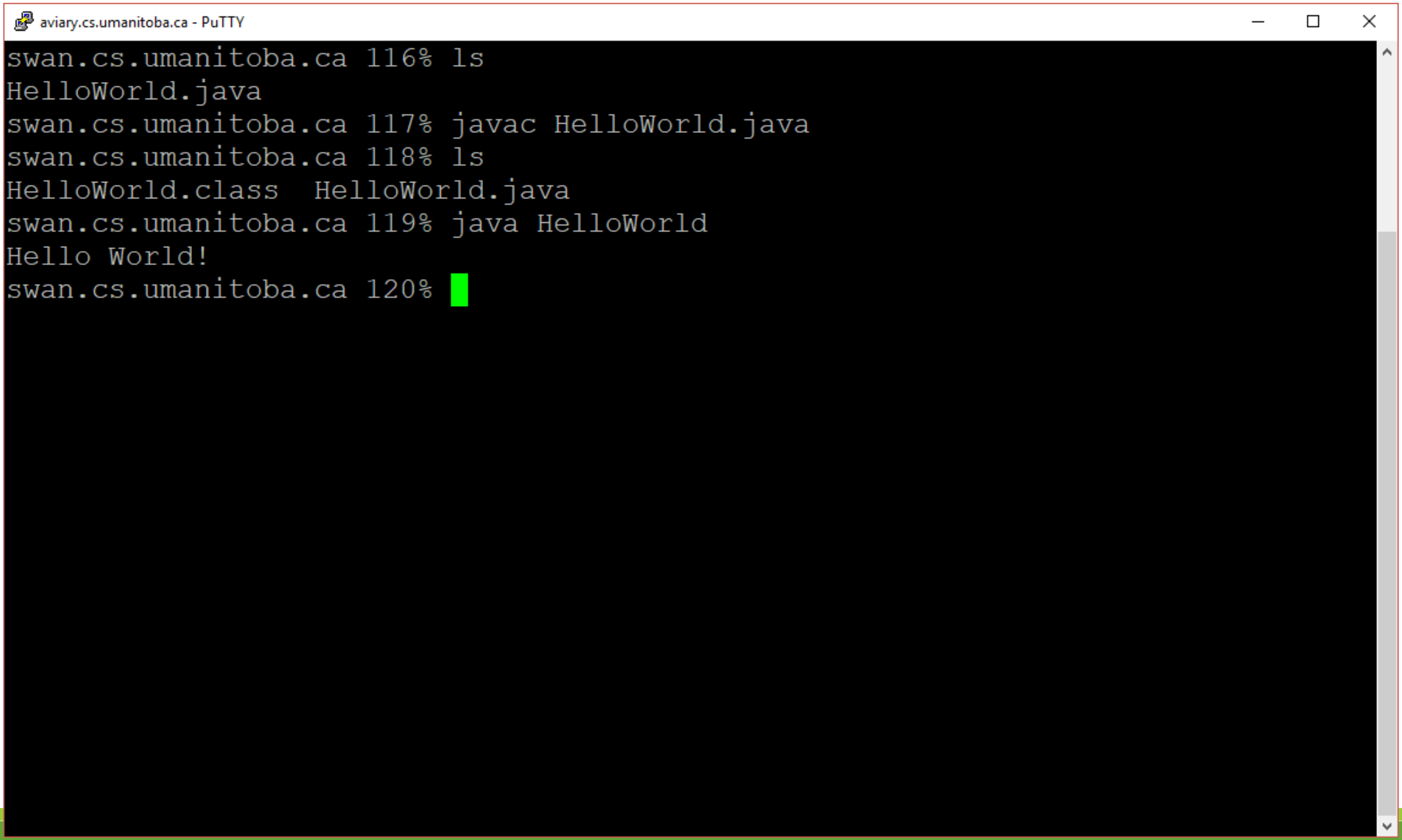
- You can also do everything from the command line in a terminal



```
swan.cs.umanitoba.ca 116% ls
HelloWorld.java
swan.cs.umanitoba.ca 117% javac HelloWorld.java
swan.cs.umanitoba.ca 118% ls
HelloWorld.class  HelloWorld.java
swan.cs.umanitoba.ca 119% java HelloWorld
Hello World!
swan.cs.umanitoba.ca 120%
```

Java execution

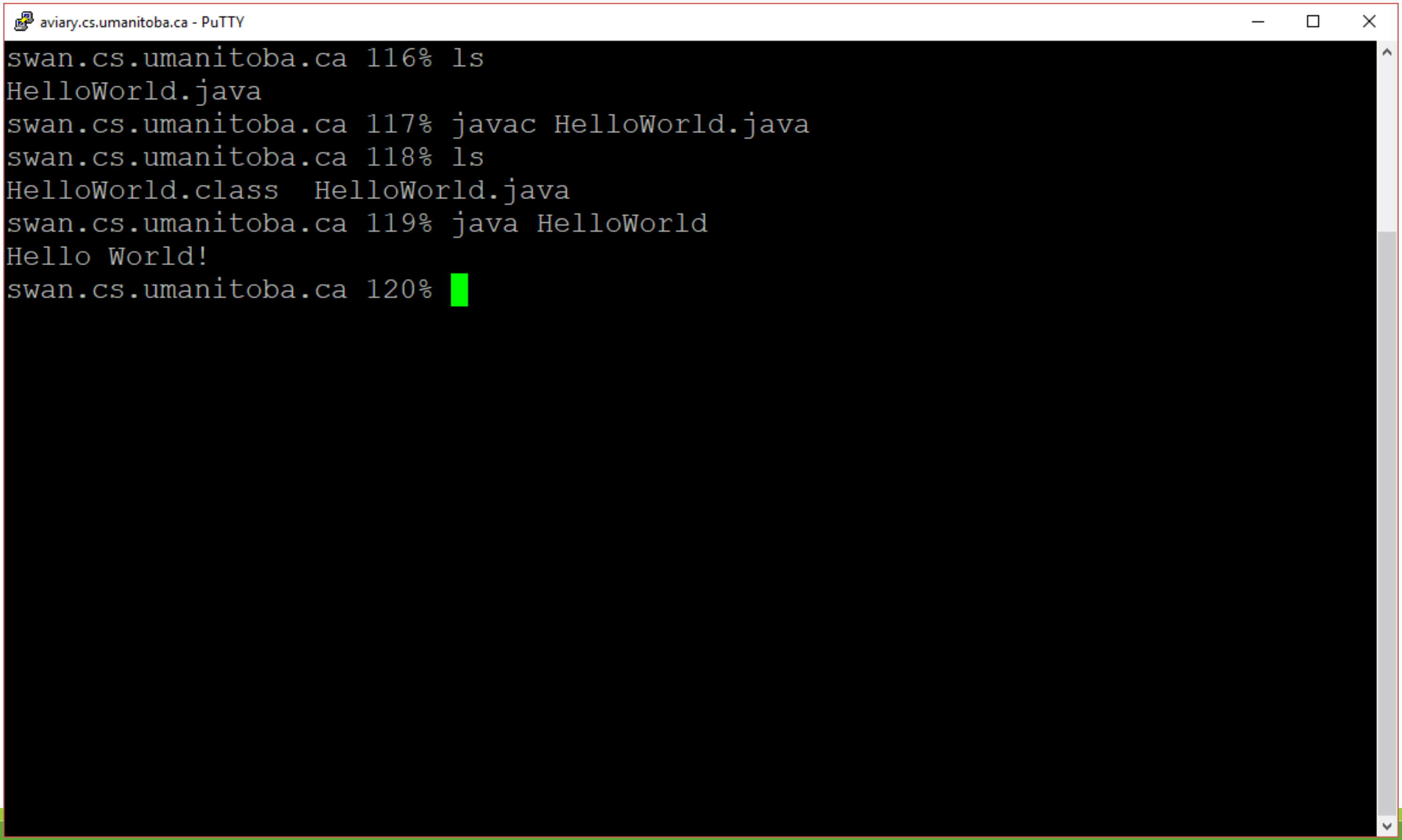
- javac <programName>.java → compiles the code (to get the .class file)



```
swan.cs.umanitoba.ca 116% ls
HelloWorld.java
swan.cs.umanitoba.ca 117% javac HelloWorld.java
swan.cs.umanitoba.ca 118% ls
HelloWorld.class  HelloWorld.java
swan.cs.umanitoba.ca 119% java HelloWorld
Hello World!
swan.cs.umanitoba.ca 120% 
```

Java execution

- java <programName> → interprets (runs) the compiled code



```
swan.cs.umanitoba.ca 116% ls
HelloWorld.java
swan.cs.umanitoba.ca 117% javac HelloWorld.java
swan.cs.umanitoba.ca 118% ls
HelloWorld.class  HelloWorld.java
swan.cs.umanitoba.ca 119% java HelloWorld
Hello World!
swan.cs.umanitoba.ca 120%
```

Building a program

- This is very different from Python, but similar to Processing

- A Java program (source code) will usually being with the following line:

```
public class xxxx {
```

and it must be stored in a file named xxxx.java

- After being compiled, you will get a xxxx.class file, which you can then execute

Let’s build our first program

- First, we put the class line, which we’ve just seen

```
public class HelloWorld {








}
```

Let’s build our first program

- First, we put the class line, which we’ve just seen

```
public class HelloWorld {





 }
```

This has to go in a file named HelloWorld.java

Let’s build our first program

- Second, we need a main method

```
public class HelloWorld {

   public static void main (String[] args){



   }
}
```

Let’s build our first program

- Second, we need a main method

```
public class HelloWorld {

   public static void main (String[] args){



   }
}
```

This is the syntax for the main method → the main method is called once when the program starts (similar to setup() in processing)

Let’s build our first program

- Then, we can add some java statements in the main

```
public class HelloWorld {

  public static void main (String[] args){

    System.out.println(“Hello World!”);

  }
}
```

Let’s build our first program

- Then, we can add some java statements in the main

```
public class HelloWorld {

  public static void main (String[] args){

    System.out.println(“Hello World!”);

  }
}
```

This is a statement that prints a String to the console.

In this case, the String is “Hello World!” and the quotes are there to indicate that this is a String.

System.out.println() prints the String inside the brackets and adds a newline; System.out.print() does not add a newline

Let’s build our first program

- Then, we can add some java statements in the main

```
public class HelloWorld {

   public static void main (String[] args){

      System.out.println(“Hello World!”);

   }
}
```

Important: reserved words in Java are case sensitive (e.g. `public` not `Public`) and statements must end with a semicolon (;) unlike Python

Naming conventions

- You shall follow these naming conventions:
  - MyProgram → class name starts with an uppercase letter
  - myVariable → variable name starts with a lowercase letter, then use camel case for the rest
  - MY_CONSTANT → constant name is in all caps

Comments

- Style 1

```
//Everything to the right of // is a comment
```

- Style 2

```
/* Everything
between /* and */
over any number of lines
is a comment
*/
```

Data types

- Java is a strongly typed language → Java requires that every piece of data (every variable) always has a designated type

- Python is very different → it is dynamically typed (variables don’t need to have a defined type)

Data types

- Each type has:
  - A name for that type, i.e. a type identifier

    Examples: `String   int   double   boolean`
  - A syntax for *literals* (constant values) of that type

    Examples: `“Hi”   53   4.557   false`
  - Operations that you can do on that type

    Examples: `+   –   *   &&`

The String type

- a String is a collection of characters

- Must use double quotes for literals “ “ (single quotes ‘ ‘ are not allowed for Strings, unlike Python)

- Examples:
  - “Hello World!”
  - “This is an entire sentence.”
  - “T” //You can put only one character in a String
  - “” //or 0 characters! → called null or empty String

The String type

- You cannot break a String literal over two or more lines

```
“This is
not
allowed for example”
```

The String type

- A useful String operation (more later): concatenation
- Operator: +
- It joins two Strings back-to-back into one longer String

The String type

- A useful String operation (more later): concatenation

- Operator: +

- It joins two Strings back-to-back into one longer String

- Example:

```
System.out.print(“This is a long sentence that I ” +
“am separating into different Strings over ” +
“ different lines, and this works!”);
```

Primitive types

- aka the most basic data types in Java:
  - Integer types: int, long, short, byte
  - Floating point types: double, float
  - Others: char, boolean

Primitive types

- aka the most basic data types in Java:
  - Integer types: int, long, short, byte
  - Floating point types: double, float
  - Others: char, boolean
- Note that all primitive types start with a lowercase letter. All other types will start with a capital letter (e.g. String).

Integer types

- There are 4 of them, and they vary in the amount of memory they use, and the range of values that they can represent
- int: 4 bytes (32 bits) ± 2147483647
- long: 8 bytes (64 bits) ± 9223372036854775807
- short: 2 bytes (16 bits) ± 32767
- byte: 1 byte (8 bits) ± 127

Integer types

- There are 4 of them, and they vary in the amount of memory they use, and the range of values that they can represent

- int: 4 bytes (32 bits) ± 2147483647 ← Most common

- long: 8 bytes (64 bits) ± 9223372036854775807

- short: 2 bytes (16 bits) ± 32767

- byte: 1 byte (8 bits) ± 127

Integer types

- There are 4 of them, and they vary in the amount of memory they use, and the range of values that they can represent

- int: 4 bytes (32 bits) ± 2147483647

Used only when you require big numbers (above 2 billion)

- long: 8 bytes (64 bits) ± 9223372036854775807

- short: 2 bytes (16 bits) ± 32767

- byte: 1 byte (8 bits) ± 127

Integer types

- There are 4 of them, and they vary in the amount of memory they use, and the range of values that they can represent

- int: 4 bytes (32 bits) ± 2147483647

- long: 8 bytes (64 bits) ± 9223372036854775807

- short: 2 bytes (16 bits) ± 32767

- byte: 1 byte (8 bits) ± 127

Used very rarely, basically only when you are told to do so

Integer constants

- You can use the minus sign (-) in front
- When you write an Integer constant (e.g. 1234), you will get an int, unless you add an L at the end (e.g. 1234L) and then it’s a long

Integer constants

- You can use the minus sign (-) in front

- When you write an Integer constant (e.g. 1234), you will get an int, unless you add an L at the end (e.g. 1234L) and then it's a long

  - 123456789012 is an error – it's too big to be an int
  - 123456789012L is OK – it's a long
  - 1234567890123456789012L is an error - it's too big to be a long!

Floating-point types

- Two of them (float and double)
  - float: 4 bytes - approx. 7 significant digits
  - double: 8 bytes - approx. 15 significant digits

- Examples:
  - double: `1.0  -0.34E-5  2.0d`
  - float: `1.0f  -0.34E-5f  2.0f`

Floating-point types

- Two of them (float and double)
  - float: 4 bytes - approx. 7 significant digits
  - double: 8 bytes - approx. 15 significant digits

- Examples:
  - double: `1.0   -0.34E-5   2.0d`
  - float: `1.0f   -0.34E-5f   2.0f`

- You can always determine the exact type of any literal value:

  5 is type int   5L is type long

  5.0 is type double   5.0f is type float

Floating-point types

- Two of them (float and double)
  - float: 4 bytes - approx. 7 significant digits
  - double: 8 bytes - approx. 15 significant digits
- Examples:
  - double: `1.0   -0.34E-5   2.0d`
  - float: `1.0f   -0.34E-5f   2.0f`

Important: d at the end gives you a double, f gives you a float. If you don't put d or f at the end of the number, it will default to double

- You can always determine the exact type of any literal value:

5 is type int	5L is type long
5.0 is type double	5.0f is type float

Other primitive types

- char: used to represent a single character, need to use the single quote to represent it (e.g. ‘a’ or ‘z’)

Other primitive types

- char: used to represent a single character, need to use the single quote to represent it (e.g. ‘a’ or ‘z’)

- boolean: `true` or `false` are the only 2 possible values

Types and the + operator

- + is a binary operator (i.e. it needs two operands)
- It can accept any two primitive data types (except boolean) or String
- There are rules for what type of data you get as the result, and some combinations don’t work

Types and the + operator

- + is a binary operator (i.e. it needs two operands)
- It can accept any two primitive data types (except boolean) or String
- There are rules for what type of data you get as the result, and some combinations don't work
  - `‘a’ + true` → error: bad operand types for binary operator '+'

Types and the + operator

- + is a binary operator (i.e. it needs two operands)
- It can accept any two primitive data types (except boolean) or String
- There are rules for what type of data you get as the result, and some combinations don't work
  - `‘a’ + true` → error: bad operand types for binary operator '+'
  - `‘a’ + 1` → what do we get?

Types and the + operator

- + is a binary operator (i.e. it needs two operands)

- It can accept any two primitive data types (except boolean) or String

- There are rules for what type of data you get as the result, and some combinations don't work

    - `‘a’ + true` → error: bad operand types for binary operator '+'

    - `‘a’ + 1` → what do we get? `‘b’`

Types and the + operator

- + is a binary operator (i.e. it needs two operands)
- It can accept any two primitive data types (except boolean) or String
- There are rules for what type of data you get as the result, and some combinations don't work
  - `'a' + true` → error: bad operand types for binary operator '+'
  - `'a' + 1` → what do we get? `'b'`

chars are represented internally by a code, so by adding 1 you get the next char

Types and the + operator

- If either side of a + operation is a String, then:
  - + means concatenation automatically
  - if the other operand is not a String, it is first converted to one (any data type in Java can be converted to a String), and then concatenated

Types and the + operator

- Example with Strings:

```
System.out.println(“I am “ + 1000 + “ years old”);
//prints: I am 1000 years old
```

Types and the + operator

- Example with Strings:

```
System.out.println(“I am “ + 1000 + “ years old”);
//prints: I am 1000 years old


System.out.println(“When will ” + 20 + 20 + “ end?”);
//prints: When will 2020 end?
```

Types and the + operator

- If the two operands are numbers (any type of number, Integer or floating-point), you will get an addition

- Example:

`31 + 0.5` → result?

int

double

Types and the + operator

- If the two operands are numbers (any type of number, Integer or floating-point), you will get an addition

- Example:

`31 + 0.5` → result?

int double

Java always converts the number with the lowest “precision” to the highest one, before making the addition. Then the results will have the highest “precision”. In this case, the int is converted to a double and the addition is made.

Types and the + operator

- If the two operands are numbers (any type of number, Integer or floating-point), you will get an addition

- Example:

`31 + 0.5` → result? `31.5`

int double double

Java always converts the number with the lowest “precision” to the highest one, before making the addition. Then the results will have the highest “precision”. In this case, the int is converted to a double and the addition is made.
Order is: double > float > long > int > short > byte

Standard arithmetic operators

- + and - : addition and subtraction (binary), and unary -
- / : division → remember that integer division discards the remainder: 5/2 = 2
- * : multiplication
- % : modulo (gives you the remainder of a division)

Standard arithmetic operators

- + and - : addition and subtraction (binary), and unary -
- / : division → remember that integer division discards the remainder: 5/2 = 2
- * : multiplication
- % : modulo (gives you the remainder of a division)
- There is no exponentiation or power operator (we have to use the Math library for that, e.g. Math.pow(4,3))

Variable declaration

- You have to declare a variable before you can use it

- Variable declaration is when you define the type of the variable

  - Remember that Java is a strongly typed language: it needs you to *declare* what is the type of each variable that you use

  - This step does not exist in Python

Variable declaration

- When declaring a variable:

1. you put the type of the variable first
2. followed by the name you want to give to the variable
3. optionally, you can initialize the variable at the same time: assign a value to it, using the = operator
4. end the line with a semicolon (;)

Variable declaration

- Examples

```
int hours;

double price;
```

Declaration only

```
int age = 55;

String name = “John”;
```

Declaration + assignment (initialization)

Choosing variable names

These rules will make your code more readable:

1. Choose meaningful names

```
int i;
double num;
```

Could refer to anything! Not meaningful!

```
int height;
double gasPrice;
```

More specific and meaningful!
Good!

Choosing variable names

These rules will make your code more readable:

2. Use short names

```
//The following name is way too long
int theNumberOfWeeksInTheSemester;
```

```
//This is much better
int numWeeks;
```

Choosing variable names

These rules will make your code more readable:

3. Use comments to describe the purpose of variables

```
int numWeeks; //number of weeks in the term

double avgGasPrice; //average price of gas in
                    //Manitoba
```

Assignment statement

- The assignment operator is =
- It is a binary operator: it has a **left** and a **right** operand

```
int age = 20;
```

- It assigns what is on its **right** to the variable that is on its **left**

Assignment statement

- As mentioned previously, each variable in Java has a type (you choose it when you declare the variable)

- Every expression or piece of data that you put on the right side of the assignment operator also has a type

```
int variable = 2 + 2;
```

Assignment statement

- As mentioned previously, each variable in Java has a type (you choose it when you declare the variable)

- Every expression or piece of data that you put on the right side of the assignment operator also has a type

```
int variable = 2 + 2;
```

This expression will be evaluated first (result will be int), and then assigned

Assignment statement

- As mentioned previously, each variable in Java has a type (you choose it when you declare the variable)

- Every expression or piece of data that you put on the right side of the assignment operator also has a type

```
int variable = 2 + 2;
```

This expression will be evaluated first (result will be int), and then assigned

- Both types, on each side of the assignment operator, must match, or be compatible

Assignment compatibility

- There are rules for what types of data you can assign to what types of variables

Assignment compatibility

- There are rules for what types of data you can assign to what types of variables

1. Strings are only compatible with Strings

Examples:

```
String name;
name = 57.3; //error – no automatic conversion is done.
```

Assignment compatibility

- There are rules for what types of data you can assign to what types of variables

1. Strings are only compatible with Strings

Examples:

```
String name;
name = 57.3; //error – no automatic conversion is done.
name = "57.3"; //OK
```

Assignment compatibility

- There are rules for what types of data you can assign to what types of variables

1. Strings are only compatible with Strings

Examples:

```
String name;
name = 57.3; //error – no automatic conversion is done.
name = "57.3"; //OK
name = ""+57.3; //Cheap trick. Now it’s a String. OK.
```

Assignment compatibility

- There are rules for what types of data you can assign to what types of variables

2. Numbers can be converted to “bigger” (or more “precise”) forms, but not the other way around

double > float > long > int > short > byte

bigger

smaller

Assignment compatibility

- There are rules for what types of data you can assign to what types of variables

2. Numbers can be converted to “bigger” (or more “precise”) forms, but not the other way around

Examples:

```
int intValue = 0; double doubleValue = 0; float floatValue = 0;
```

Assignment compatibility

- There are rules for what types of data you can assign to what types of variables

2. Numbers can be converted to “bigger” (or more “precise”) forms, but not the other way around

Examples:

```
int intValue = 0; double doubleValue = 0; float floatValue = 0;
intValue = 57.3; //Error. Can’t handle the .3
```

Assignment compatibility

- There are rules for what types of data you can assign to what types of variables

2. Numbers can be converted to “bigger” (or more “precise”) forms, but not the other way around

Examples:

```
int intValue = 0; double doubleValue = 0; float floatValue = 0;
intValue = 57.3; //Error. Can’t handle the .3
doubleValue = 57; //OK. Java can add .0
```

Assignment compatibility

- There are rules for what types of data you can assign to what types of variables

2. Numbers can be converted to “bigger” (or more “precise”) forms, but not the other way around

Examples:

```
int intValue = 0; double doubleValue = 0; float floatValue = 0;
intValue = 57.3; //Error. Can’t handle the .3
doubleValue = 57; //OK. Java can add .0
floatValue = doubleValue; //Error. Too many digits to fit.
```

Assignment compatibility

- There are rules for what types of data you can assign to what types of variables

2. Numbers can be converted to “bigger” (or more “precise”) forms, but not the other way around

Examples:

```
int intValue = 0; double doubleValue = 0 ;
intValue = 57.3; //Error. Can’t handle th
doubleValue = 57; //OK. Java can add .0
floatValue = doubleValue; //Error. Too many digits to fit.
```

Actual Java error message:

incompatible types: possible lossy conversion from double to float

Assignment compatibility

- There are rules for what types of data you can assign to what types of variables

2. Numbers can be converted to “bigger” (or more “precise”) forms, but not the other way around

Examples:

```
int intValue = 0; double doubleValue = 0; float floatValue = 0;
intValue = 57.3; //Error. Can’t handle the .3
doubleValue = 57; //OK. Java can add .0
floatValue = doubleValue; //Error. Too many digits to fit.
doubleValue = floatValue; //OK. double is bigger, so float fits
```

Assignment compatibility

- There are rules for what types of data you can assign to what types of variables

- Remember to always pay attention to types

  5, 5L, 5.0 and 5.0f are not the same!

Defining methods

- A method (Processing called them *functions*) is made up of two parts:
    - A signature (which defines a modifier, a return type, the name of the method and a list of parameters)
    - A body (block of statements performed when the method is called)

Defining methods

- Example:

```
static void printMessage (String message)
{
    System.out.println(message);
}
```

Defining methods

- Example:

```
static void printMessage (String message)
{
    System.out.println(message);
}
```

The first line is the signature, it declares the method

Defining methods

- Example:

Modifier (for now, always use static)

```
static void printMessage (String message)
{
    System.out.println(message);
}
```

Defining methods

- Example:

Return type → always needed, must be void if nothing is returned by the method

```
static void printMessage (String message)
{
    System.out.println(message);
}
```

Defining methods

- Example:

Name of the method (you choose the name, convention is for the first word to be a verb)

```
static void printMessage (String message)
{
    System.out.println(message);
}
```

Defining methods

- Example:

List of parameters, separated by commas → each parameter must be declared as a normal variable (type + name)

```
static void printMessage (String message)
{
    System.out.println(message);
}
```

Defining methods

- Example:

```
static void printMessage (String message)
{
    System.out.println(message);
}
```

The rest is the body, enclosed within { }

Defining methods

- If your method returns something, you need to define the return type in the signature (instead of using void)

- Then the last statement of the body should be a return statement (return followed by an expression or variable to be returned, and a semicolon to end the statement)

Defining methods

- Example with 2 parameters, and a returned value:

```
static double calculateTotal (double tax, double subTotal)
{
    return subTotal + subTotal * tax;
}
```

Defining methods

- Example with 2 parameters, and a returned value:

return type is double in this case

```
static double calculateTotal (double tax, double subTotal)
{
    return subTotal + subTotal * tax;
}
```

return statement

Defining methods

- A void method does not need a return statement (it will stop when the end of the method is reached)

- A non-void method must have a return statement, and the returned value must be of the same return type (or compatible with the return type) defined in the signature

Where do methods go?

- In the Java file (of course)

- In the class body

- Typically after the main
    - The order of method definitions does not matter to Java, but by convention we usually put the main method first (or last sometimes)
    - Reason: we want to be able to find the main easily → makes the code more readable

Calling methods

- We call a method using its name, followed by parentheses ()

  - If the method does not use parameters, leave the parentheses empty ()

  - If the method requires parameters, put them inside the parentheses, separated by commas, in the same order as they were listed in the signature

Calling methods

- If the method returns a value, the method call can be used as a data item

```
int bigNb = 6 * Math.max(23, 45) + 1;
```

- If the method does not return a value (void), use it as a statement ending with a semicolon ;

```
System.out.println("The big number is " + bigNb);
```

Calling methods

- Example:

```
public class MyTest{
    public static void main (String[] args){









    }
    //Assume that methods defined earlier are here
}
```

Calling methods

- Example:

```
public class MyTest{
    public static void main (String[] args){
        double itemPrice = 59.99;







    }
    //Assume that methods defined earlier are here
}
```

Calling methods

- Example:

```
public class MyTest{
    public static void main (String[] args){
        double itemPrice = 59.99;
        String myMessage = “Hello!”;




    }
    //Assume that methods defined earlier are here
}
```

Calling methods

- Example:

```
public class MyTest{
     public static void main (String[] args){
          double itemPrice = 59.99;
          String myMessage = “Hello!”;
          printMessage(myMessage);




     }
     //Assume that methods defined earlier are here
}
```

Calling methods

- Example:

```
public class MyTest{
    public static void main (String[] args){
        double itemPrice = 59.99;
        String myMessage = “Hello!”;
        printMessage(myMessage);
        printMessage(“I can also put a String here directly”);



    }
    //Assume that methods defined earlier are here
}
```

Calling methods

- Example:

```
public class MyTest{
    public static void main (String[] args){
        double itemPrice = 59.99;
        String myMessage = “Hello!”;
        printMessage(myMessage);
        printMessage(“I can also put a String here directly”);
        double total = calculateTotal(0.13, itemPrice);


    }
    //Assume that methods defined earlier are here
}
```

Calling methods

- Example:

```
public class MyTest{
    public static void main (String[] args){
        double itemPrice = 59.99;
        String myMessage = “Hello!”;
        printMessage(myMessage);
        printMessage(“I can also put a String here directly”);
        double total = calculateTotal(0.13, itemPrice);
        printMessage(“The total is $” + total);
    }
    //Assume that methods defined earlier are here
}
```

Global variables

- Global variables must be declared outside of any method
  - for now be sure to add the static keyword in front (will make sense in a few weeks)

```
public class GlobalExample{
    static int id = 1001;
}
```

Global variables

- Example

```
public class GlobalExample{
    static int id = 1001; //global var is accessible
                          //anywhere in the class
    public static void main (String[] args){
        int local = 55; //local var, exists only in this block
        System.out.println(“Id is: “ + id);
        System.out.println(“local is: “ + local);
    }
}
```

Named constants

- Adding the keyword final before a declaration
  - makes it a “constant” not a “variable”
  - promises that its value will never change
    - produces an error if you ever try to change it

Named constants

- Adding the keyword final before a declaration
  - makes it a “constant” not a “variable”
  - promises that its value will never change
    - produces an error if you ever try to change it

- Naturally, it should be given a value with =

- Convention: use ALL_UPPER_CASE for constants

```
int userInput; //This is a regular variable
final double TAX_RATE = 0.13; //This is a constant
```

Named constants

- You can have a global constant → just add static

```
public class MyProgram{
    static final double TAX_RATE = 0.13; //global constant

    public static void main (String[] args){
        //statements here
    }
}
```

Formatting output

- Instead of print() or println() you can use printf() or format() to control output exactly
  - printf and format are two names for the same method

Formatting output

- Here's how to use them:

```
System.out.printf("Casting %f to int gives %d %n",
                              23.8, (int)23.8  );
```

- The first parameter is a String that indicates exactly how you want the data printed
  - The red codes that start with % are where the data goes
  - Except %n which just gives a newline character
- There can be any number of other parameters
  - These supply the actual data to print (in blue)

Formatting codes

- Commonly used codes:
  - %d – print a decimal integer here (base 10 integer)
  - %6d – use at least 6 characters to do that
  - %f – print a floating-point value here
  - %6f – use at least 6 characters to do that
  - %6.2f – with exactly 2 of them after the decimal point
  - %s – print a String here
  - %n – print a newline (\n character) here

- There must be one additional parameter (after the String) for each code used (except %n), and it must be the correct type

Formatting codes

- Previous style:

  ```
  System.out.println(a+" plus "+b+" is "+(a+b)+".\n");
  ```

- Formatted style:

  ```
  System.out.printf("%d plus %d is %d.%n",a,b,a+b);
  ```

- Most useful to
  - Line up decimal points – perhaps use %7.2f
  - Round off a number
    - Use %4.1f to get 98.6 and not 98.59999999999999

Input using Scanner

- Scanner can be used to get input (keyboard input) from the user during the execution of a program

- Very useful if you need to interact/prompt the user for some information

Input using Scanner

- To use Scanner, you first need to import the class from the library, using this statement at the very top of the file:

```
import java.util.Scanner;
```

Input using Scanner

- Then, in your program, use the special declaration statement to create the Scanner object:

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

this is a variable name, you can call it whatever you want

Input using Scanner

- Now, on this Scanner object that we just created, we can use any of these methods to get input

keyboardInput.nextInt()	→ int
keyboardInput.nextLong()	→ long
keyboardInput.nextFloat()	→ float
keyboardInput.nextDouble()	→ double
keyboardInput.next()	→ String
keyboardInput.nextLine()	→ String

Input using Scanner

- Now, on this Scanner object that we just created, we can use any of these methods to get input

keyboardInput.nextInt()	→	int
keyboardInput.nextLong()	→	long
keyboardInput.nextFloat()	→	float
keyboardInput.nextDouble()	→	double
keyboardInput.next()	→	String
keyboardInput.nextLine()	→	String

Return only the next token: sequence of non-blank characters

Input using Scanner

- Now, on this Scanner object that we just created, we can use any of these methods to get input

keyboardInput.nextInt()	→ int
keyboardInput.nextLong()	→ long
keyboardInput.nextFloat()	→ float
keyboardInput.nextDouble()	→ double
keyboardInput.next()	→ String
keyboardInput.nextLine()	→ String

Returns the entire line, including blank spaces

Input using Scanner

- Now, on this Scanner object that we just created, we can use any of these methods to get input

keyboardInput.nextInt()	→	int
keyboardInput.nextLong()	→	long
keyboardInput.nextFloat()	→	float
keyboardInput.nextDouble()	→	double
keyboardInput.next()	→	String
keyboardInput.nextLine()	→	String

These methods automatically convert the keyboard input to the specified type. You will get an error if the next token entered by the user is not of the expected format.

See H_ScannerTest.java

Type conversions

- We have seen before how to convert a number to a String (using the empty string and concatenation)
- You will often have to do the opposite: from String to number (e.g. from command line arguments)
- “57” is not the same as 57 → you cannot store a String in an int variable

Type conversions

- String to primitive type conversion:

Integer.parseInt(*String*)	→ to int
Long.parseLong(*String*)	→ to long
Double.parseDouble(*String*)	→ to double
Float.parseFloat(*String*)	→ to float
Boolean.parseBoolean(*String*)	→ to boolean

Replace *String* in the above methods by any String you want to convert.
Once again, the String must be convertible to the corresponding type, otherwise you'll get an error.

Type conversions

- Primitive type to String conversion:

Integer.toString(*int*)
Long.toString(*long*)
Double.toString(*double*)
Float.toString(*float*)
Boolean.toString(*boolean*)

Replace *int/long/double/float/boolean* in the above methods by the variable of that type you want to convert.

These methods are called automatically by Java when you concatenate these primitive types with a String.

Conversion by casting

- You can also force conversion between some types of values using type casting

- Use (*desiredType*) in front of the variable/value to convert

Conversion by casting

- You can also force conversion between some types of values using type casting

- Use (*desiredType*) in front of the variable/value to convert

- Example:

```
double d = 102.3
int i = (int) d;
System.out.println(i);  //102 will be printed, the .3 is dropped
```

Conversion by casting

- When converting a floating-point number to an integer type, all decimals just disappear (equivalent to rounding down)

- Example:

```
double d = 102.999999999
int i = (int) d;
System.out.println(i);  //102 will be printed, it’s not rounded to
                        //the nearest integer
```

Conversion by casting

- Type casting is used to go from a “bigger” to a “smaller” numeric type (double → float → long → int)
- Casting can make you lose information (e.g. when going from double to int, you lose the decimals)
- You cannot cast Strings to/from numbers
    - Use the methods shown previously

Operators ++ and --

- They must be used on a variable only
- ++ is the incrementation operator (adds 1 to the value of the variable)

  → equivalent to x = x + 1;
- -- is the decrementation operator (removes 1 to the value of the variable)

  → equivalent to x = x - 1;

Operators ++ and --

- They can be used as a prefix or postfix to the variable

- x++ → returns the value of x and then increments it
- ++x → increments x and then returns its value

➡ same principle for decrement

Operators ++ and --

- They can be used as a prefix or postfix to the variable

- x++ → returns the value of x and then increments it
- ++x → increments x and then returns its value

  ➡ same principle for decrement

- Example:

```
int x = 5;
System.out.println(x++);  //prints 5
System.out.println(x);  //prints 6
System.out.println(++x);  //prints 7
```

Other ways of incrementing

- ++ only increments by one (same for --, decrements by one)

- If you want to increment (decrement) by more than one, use += (-=)

- Examples:

x = x + 2 → equivalent to x += 2
x = x - 5 → equivalent to x -= 5

Other ways of incrementing

- ++ only increments by one (same for --, decrements by one)
- If you want to increment (decrement) by more than one, use += (-=)
- Examples:

x = x + 2 → equivalent to x += 2
x = x - 5 → equivalent to x -= 5
x = x / 2 → equivalent to x /= 2
x = x * 10 → equivalent to x *= 10

Works with other math operators as well!

The boolean type

- Recall that boolean can take 2 values: true or false

```
boolean myBool = true;
myBool = false;
```

- Three operations on boolean:
  - && → and (binary)
  - || → or (binary)
  - ! → not (unary)

Relational operators

- Operators that test a relation between two values and return a boolean

- Six relational operators:
  - == → equal to
  - != → not equal to
  - < → less than
  - <= → less than or equal to
  - > → greater than
  - >= → greater than or equal to

Relational operators

- Special note on ==
  - Do not use on Strings → <u>not</u> appropriate for comparing Strings
  - Use instead:

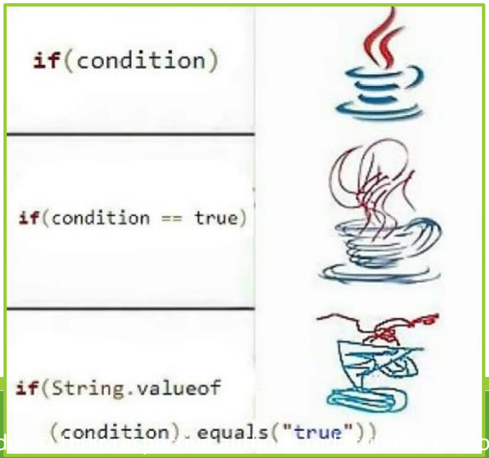
```
string1.equals(string2)  //returns true or false

string1.compareTo(string2)  //returns an int (char value difference
                             //between first 2 dissimilar chars) →
                             //returns 0 if Strings are identical
```

Relational operators

- Also, do not use == or != for booleans
    - It’s completely redundant and useless

flag == true → equivalent to: flag

flag != true → equivalent to: !flag



```
if(condition)
```

```
if(condition == true)
```

```
if(String.valueof
  (condition).equals("true"))
```

Conditions - if, else if, else

- The if, else if and else statements give us choices

- The if and else if statements must be followed by an expression giving a boolean result within parentheses

  - else is not followed by an expression

- Then you open a block (using curly brackets {}) containing the statements to be executed if the expression is true

Conditions - if, else if, else

- Example:

```
int number = -5;

if (number > 0){
    System.out.println(“Positive number”);
}
else if (number == 0)
    System.out.println(“Zero”);
else
    System.out.println(“Negative number”);
```

Conditions - if, else if, else

- Example:

```
int number = -5;

if (number > 0){
      System.out.println("Positive number");
}
else if (number == 0)
      System.out.println("Zero");
else
      System.out.println("Negative number");
```

Note that the curly brackets are not necessary if there is only one statement inside the block, but you can still put them anyway.

If there is more than one statement, you absolutely need the curly brackets.

Conditions - if, else if, else

- Example:

```
int number = -5;

if (number > 0){
    System.out.println(“Positive number”);
}
else if (number == 0)
    System.out.println(“Zero”);
else
    System.out.println(“Negative number”);
```

Unlike Python, the indentation is not required by the compiler, but it is extremely important for readability

Loops

- There are three different types of loops that you can use:
  - for loops
  - while loops
  - do - while

- You can always get the same end result with any of the three types, but in most cases one specific type of loop will be more appropriate for the task

For loops

- Probably the one we use most often

- Example:

initializes a counter

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

For loops

- Probably the one we use most often

- Example:

loop continues while this expression is true

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

For loops

- Probably the one we use most often

- Example:

this determines how to update the counter after each iteration

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

For loops

- Probably the one we use most often

- Example:

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

Then you open a block and put statements that must be executed at each iteration of the loop

For loops

- Probably the one we use most often

- Example:

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

Note that the int i variable will exist only inside the for loop → once you exit the for loop, it won’t be accessible anymore

For loops

- Probably the one we use most often

- Example:

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

For COMP 1012 people: this for loop is equivalent, in Python, to
for i in range(0,11)

While loops

- While loops only require a boolean expression inside parentheses
- Example:

```
int counter = 0;

while ( counter <= 10 ) {
    System.out.println(counter);
    counter++;
}
```

The block is executed while the expression is true

While loops

- While loops only require a boolean expression inside parentheses

- Example:

```
int counter = 0;

while ( counter <= 10 ) {
    System.out.println(counter);
    counter++;
}
```

Remember to increment the counter inside the body

While loops

- While loops only require a boolean expression inside parentheses

- Example:

```
int counter = 0;

while ( counter <= 10 ) {
    System.out.println(counter);
    counter++;
}
```

This is just an example of how to achieve the same results with a while loop.

Usually, we would use a for loop to do this specific task.

Do - while loops

- Do - while is similar to a while loop, except that it executes the block first, and then checks the boolean expression → guarantees at least 1 execution of block

- Example:

```
int counter = 0;
do {
    System.out.println(counter);
    counter++;
}
while ( counter <= 10 );
```

The block is executed first (at least once for sure), and then it will keep being executed while the expression is true

Do - while loops

- Do - while is similar to a while loop, except that it executes the block first, and then checks the boolean expression → guarantees at least 1 execution of block

- Example:

```
int counter = 0;
do {
     System.out.println(counter);
     counter++;
}
while ( counter <= 10 );
```

Note that the do-while loop requires a semicolon (;) at then end of the while(expression);

Do - while loops

- Do - while is similar to a while loop, except that it executes the block first, and then checks the boolean expression → guarantees at least 1 execution of block

- Example:

```
int counter = 11;
do {
    System.out.println(counter);
    counter++;
}
while ( counter <= 10 );
```

In this example, the block (following do) will be executed once, even though the expression is false → that's because the expression in a do-while is only checked after executing the block

Loops - special statements

- Two specific keywords can be used inside any type of loop:
  - break → immediately terminates the inner loop
  - continue → immediately skips to the next iteration of the loop
- **NOTE**: These are shown for informational purposes only. Programming standards in COMP 1020 do not permit the use of break or continue.

Loops - special statements

- Example:

```
for ( int i = 0; i < 5; i++ ) {
    if ( i == 2)
        break;
    System.out.println(i);
}
```

What is going to be printed?

- **NOTE**: These are shown for informational purposes only. Programming standards in COMP 1020 do not permit the use of break or continue.

Loops - special statements

- Example:

```
for ( int i = 0; i < 5; i++ ) {
    if ( i == 2)
        break;
    System.out.println(i);
}
```

What is going to be printed?

```
0
1
```

- **NOTE**: These are shown for informational purposes only. Programming standards in COMP 1020 do not permit the use of break or continue.

Loops - special statements

- Example 2:

```
for ( int i = 0; i < 5; i++ ) {
      if ( i == 2)
            continue;
      System.out.println(i);
}
```

What is going to be printed?

- **NOTE**: These are shown for informational purposes only. Programming standards in COMP 1020 do not permit the use of break or continue.

Loops - special statements

- Example 2:

```
for ( int i = 0; i < 5; i++ ) {
    if ( i == 2)
        continue;
    System.out.println(i);
}
```

What is going to be printed?

0
1
3
4

- **NOTE**: These are shown for informational purposes only. Programming standards in COMP 1020 do not permit the use of break or continue.

More on Strings: Escape (\)

- Escape character: \ (backslash)

- It is used to “escape” characters or sequences of characters in a String that otherwise would have a specific meaning in the context of a String literal

Escape (\)

- Imagine you want to put a double quote (“) inside a String → normally it would be recognized by Java as the end of the String → we need to escape it!

- Example:

```
String myString = “String ending with a double quote \””;
```

Escape (\)

- Other uses of \
  - \\ → gives a backslash character
  - \n → gives a newline character (enter)
  - \t → gives a tab character

String methods

- You can call methods on Strings
- There are quite a few of them, and they are very useful
- Syntax looks like this:

```
someString.methodName(parameters);
```

String methods

- To check if two Strings are identical: equals()

```
String s = “hello”;
if ( s.equals(“Hello”))
    System.out.println(“Strings are equal”);
```

String methods

- To check if two Strings are identical: equals()

```
String s = “hello”;
if ( s.equals(“Hello”))
    System.out.println(“Strings are equal”);
```

In this case, it’s false! (because of h != H)

String methods

- To check if two Strings are identical, but ignoring case (lowercase vs uppercase does not matter): equalsIgnoreCase()

```
String s = “hello”;
if ( s.equalsIgnoreCase(“Hello”))
   System.out.println(“Strings are equal, ignoring case”);
```

In this case, it’s true!

String methods

- To get the length of a String (number of characters): length()

```
String s = “hello”;
System.out.println(s.length());  //prints 5
```

String methods

- To get the char at the position i: charAt(i)

```
String s = “hello”;
System.out.println(s.charAt(1));  //prints??
```

String methods

- To get the char at the position i: charAt(i)

```
String s = “hello”;
System.out.println(s.charAt(1));  //prints e → first
                                 //position of the String is 0!
```

Math operations

- There is a library for math operations: it’s called Math
- You don’t need to “import” it, it’s always available
- Some constants are also accessible from Math
  - e.g. Math.PI → double value of 3.14159…

Math operations

- double Math.pow(double x, double y)
  - takes x to the power y
- double Math.sqrt(double x)
  - gives the square root of x
- int Math.min(int x, int y)
- int Math.max(int x, int y)
  - give the minimum or maximum of the two
  - there are also versions for long, float, and double
- double Math.random( )
  - gives a random double in the range 0 ≤ x <1
  - note there is nothing inside ( ) – but they still must be there!

Math operations

- double Math.pow(double x, double y)
  - takes x to the power y
- double Math.sqrt(double x)
  - gives the square root of x
- int Math.min(int x, int y)
- int Math.max(int x, int y)
  - give the minimum or maximum of the two
  - there are also versions for long, float, and double
- double Math.random( )
  - gives a random double in the range 0 ≤ x <1
  - note there is nothing inside ( ) – but they still must be there!

There are many many more…
You can always visit the online documentation to learn more:
https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html

Arrays

- An array is an object that stores a group of values of the same type

- An array can contain any number of elements (including 0)

- Once the size of an array is set, it cannot be changed

- *You can store the reference (address) of an array (or any object) in a variable, but not the array itself → more on that in a few weeks

Creating an array

```
int[] arrayOfInts = new int[10];
```

Creating an array

```
int[] arrayOfInts = new int[10];
```

The type declaration, in this case, an array of int. [] must stay empty here.

Creating an array

```
int[] arrayOfInts = new int[10];
```

Just like a regular variable declaration, here is the name you want to give to the variable

Creating an array

```
int[] arrayOfInts = new int[10];
```

To create the array, we use the new keyword, followed by the type of values again and within the square brackets you put the size of the array. The size is necessary to allocate the appropriate amount of space in memory.

Creating an array

```
int[] arrayOfInts = new int[10];
```

Once the array has been created, its size cannot be changed, but the elements contained inside can.

Arrays

- When created as shown previously, the newborn array is filled with default values for each type:

  - int[] → 0
  - double[] → 0.0d
  - float[] → 0.0f
  - boolean[] → false
  - char[] → ‘\u0000’
  - any object, including String → null

Arrays

- You can create an array in two steps (declaration first, and then creation of the array)

```
String[] myArray;  //after this, the myArray variable is
                 //created but it points to null for now

myArray = new String[50];
```

Creating an initialized array

```
    int[] data = new int[]{1, 2, 3, 4, 5};
OR
    int[] data = {1, 2, 3, 4, 5};  //OK to omit “new int[ ]”, only
                                  //if you declare the variable on the same line
```

- This creates an array and initializes the values inside at the same time
- In this example, we’ll get this array of size 5:

1	**2**	**3**	**4**	**5**

Accessing arrays

- To access the element stored at a specific position in the array, use [position] after the variable name (of the array)

- Example:

```
int[] arrayOfInts = new int[]{1, 2, 3, 4, 5};

System.out.println(arrayOfInts[4]);  //prints??
```

Accessing arrays

- To access the element stored at a specific position in the array, use [position] after the variable name (of the array)

- Example:

```
int[] arrayOfInts = new int[]{1, 2, 3, 4, 5};

System.out.println(arrayOfInts[4]);  //prints 5 → first
                                   //position has index 0!
```

Accessing arrays

- To access the element stored at a specific position in the array, use [position] after the variable name (of the array)

- Example:

```
int[] arrayOfInts = new int[]{1, 2, 3, 4, 5};

System.out.println(arrayOfInts[4]);  //prints 5 → first
                                   //position has index 0!
```

Remember: valid indices go from 0 to length - 1

Accessing arrays

- You can always modify what is contained at a specific position, using the same syntax → [position]

```
int[] arrayOfInts = new int[]{1, 2, 3, 4, 5};

arrayOfInts[0] = 7;
arrayOfInts[1] = arrayOfInts[0] + arrayOfInts[1];

System.out.println(arrayOfInts[1]);  //prints??
```

Accessing arrays

- You can always modify what is contained at a specific position, using the same syntax → [position]

```
int[] arrayOfInts = new int[]{1, 2, 3, 4, 5};

arrayOfInts[0] = 7;
arrayOfInts[1] = arrayOfInts[0] + arrayOfInts[1];

System.out.println(arrayOfInts[1]);  //prints 9
```

Accessing arrays

- Getting the length of an array (number of cells) is easy: use myArray.length;

- Example:

```
double[] myArray = new double[]{1.0, 2.5, 3.44};

System.out.println(myArray.length);  //prints??
```

Note: no () after length

Accessing arrays

- Getting the length of an array (number of cells) is easy: use myArray.length;

- Example:

```
double[] myArray = new double[]{1.0, 2.5, 3.44};

System.out.println(myArray.length);  //prints 3
```

Printing an array

- Unlike Python, you cannot just put your array variable in a print statement (it will print the reference…)

- You have to print the array yourself, by traversing the array using a loop:

```
for(int i=0; i < data.length; i++) {
      System.out.print(data[i]+" ");
}
System.out.println( );  //just adding a newline at the end
```

Printing an array

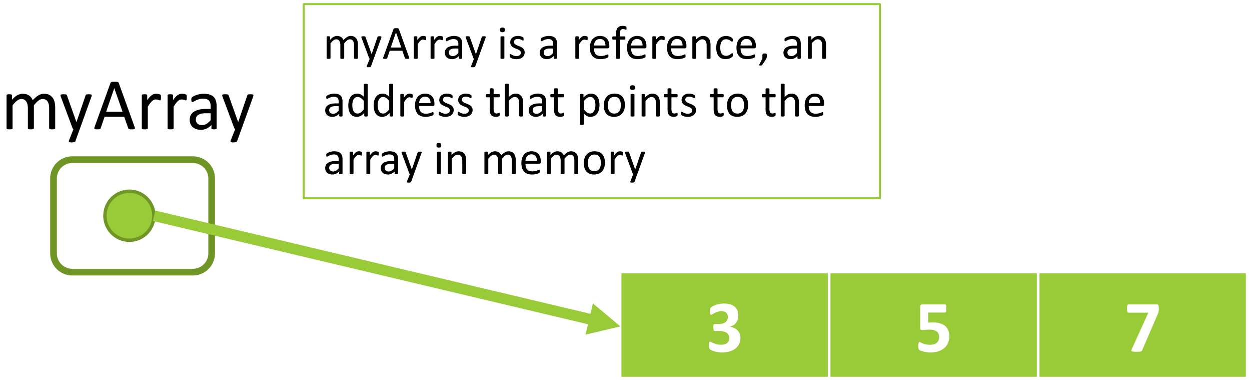
- You could also use the “Arrays” class (needs import statement)

```
import java.util.Arrays;  //at top of file
//
//
//
System.out.println(Arrays.toString(data));  //returns a
                              //String representation of the array
```

Copying an array

- Here’s how not to copy an array:

```
int[] myArray = new int[] {3, 5, 7};
```

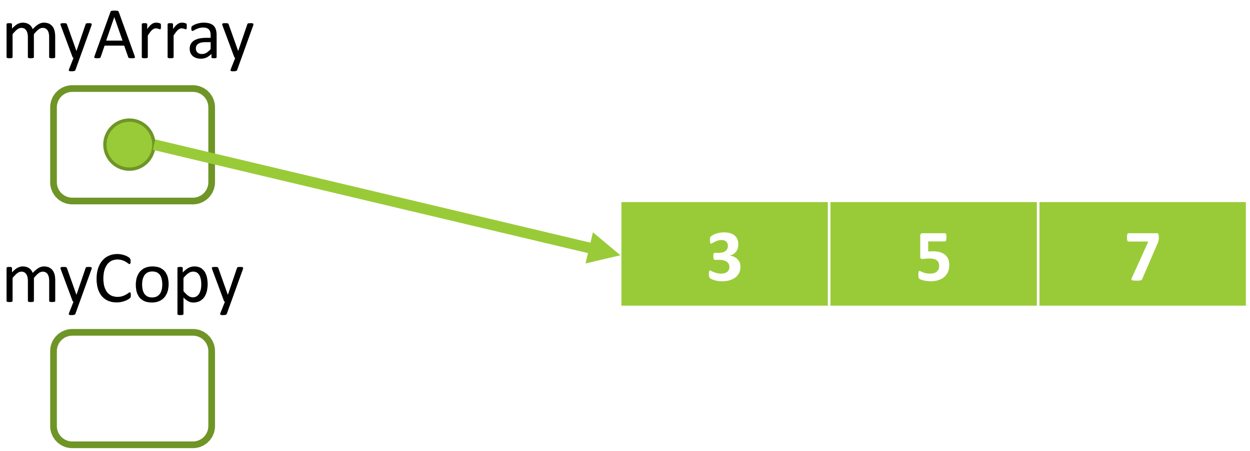


myArray is a reference, an address that points to the array in memory

Copying an array

- Here's how not to copy an array:

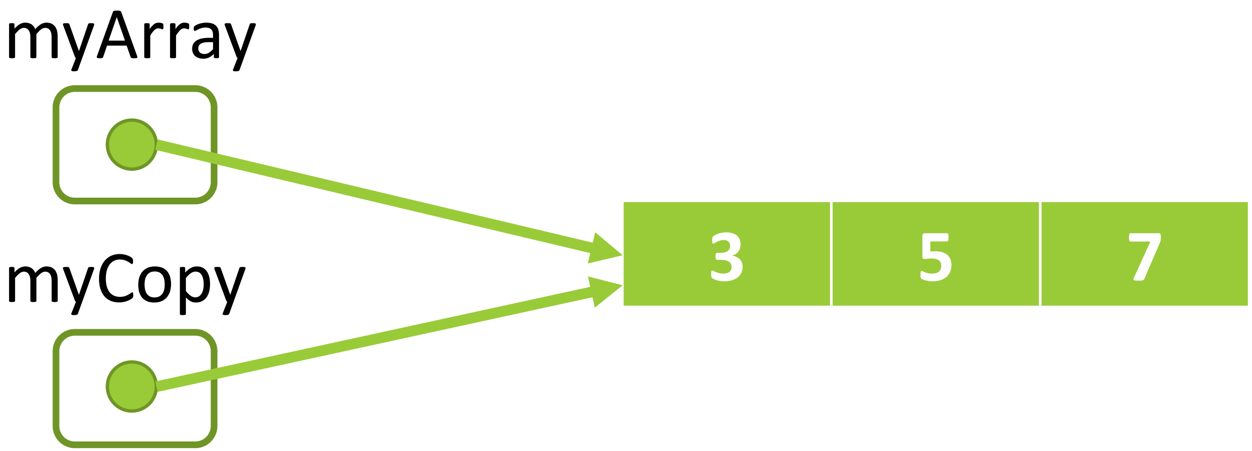
```
int[] myArray = new int[] {3, 5, 7};
int[] myCopy;
```



Copying an array

- Here's how not to copy an array:

```
int[] myArray = new int[] {3, 5, 7};
int[] myCopy;
myCopy = myArray;
```

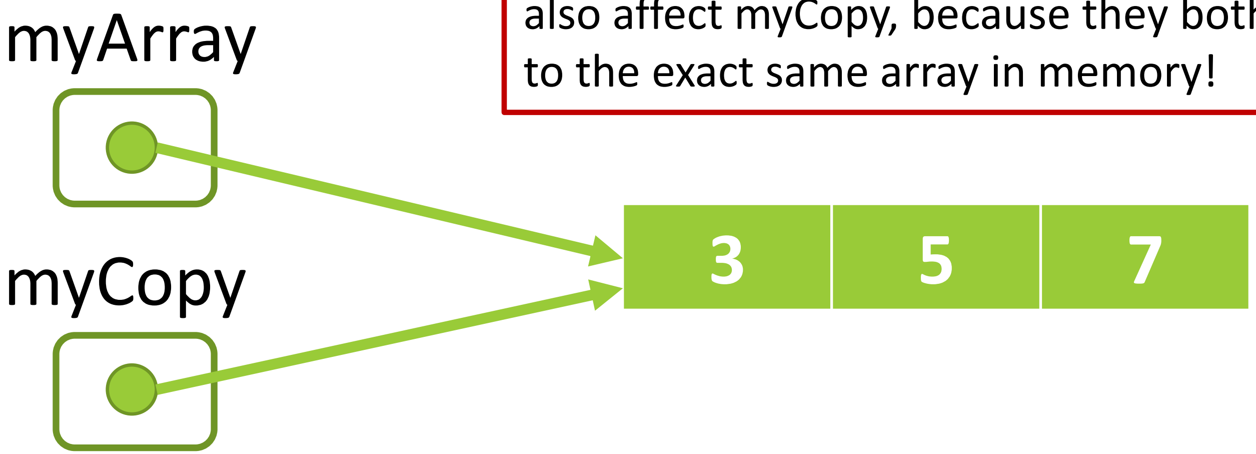


Copying an array

- Here's how not to copy an array:

```
int[] myArray = new int[] {3, 5, 7};
int[] myCopy;
myCopy = myArray;
```

If you do this, you don't get two independent copies of the array, you just get two references to the same location in memory! Modifying myArray's elements will also affect myCopy, because they both point to the exact same array in memory!



Copying an array

- Here’s the appropriate way of copying arrays:

```
int[] myArray = new int[] {3, 5, 7};
int[] myCopy = new int[myArray.length];  //set same size

for (int i = 0; i < myArray.length; i++) {
    myCopy[i] = myArray[i];  //copies each element
}
```

- Alternative to using a for loop:

```
    System.arraycopy(myArray, 0, myCopy, 0, myArray.length);
```

Syntax shortcut

- There's a syntax shortcut for iterating over all elements in an array

```
for (int i = 0; i < data.length; i++) {
    System.out.println(data[i]);
}
```

→ you can do instead

```
for (int element : data)
    System.out.println(element);
```

Syntax shortcut

- There’s a syntax shortcut for iterating over all elements in an array

```
for (int i = 0; i < data.length; i++) {
    System.out.println(data[i]);
}
```

→ you can do instead

```
for (int element : data)
    System.out.println(element);
```

This has to match the type that is contained inside the array named data

Syntax shortcut

- There's a syntax shortcut for iterating over all elements in an array

```
for (int i = 0; i < data.length; i++) {
    System.out.println(data[i]);
}
```

→ you can do instead

```
for (int element : data)
    System.out.println(element);
```

At each iteration, element will be one of the elements inside the data array