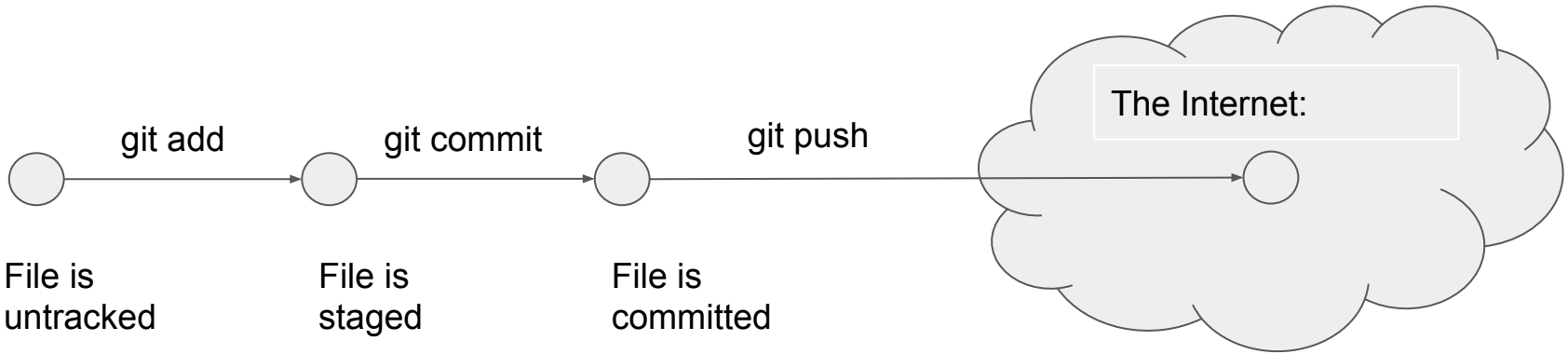
Weeks 1 & 2 Review

Day 1: Command Line Commands

- The main command to move around folder is cd. There are several variations of these:
  - cd ~ : Returns you to your home directory.
  - cd <directory name> : Takes you to a specified directory i.e. cd workspace takes you to a folder called workspace
  - cd .. : Takes you one level up.
- You can always see what directory you're in by typing pwd.
- The ls command lists all the files in the current directory.
- To create a directory we use the mkdir <directory name> command
- To copy a file from one directory to another: cp <source> <destination>
- To move a file from one directory to another: mv <source> <destination>
- We can remove a file using the rm <filename> command
- We can remove a directory using the rmdir <directory name> command.
  - Directories cannot be removed from command line shell if they are not empty.

Day 1: Source Control

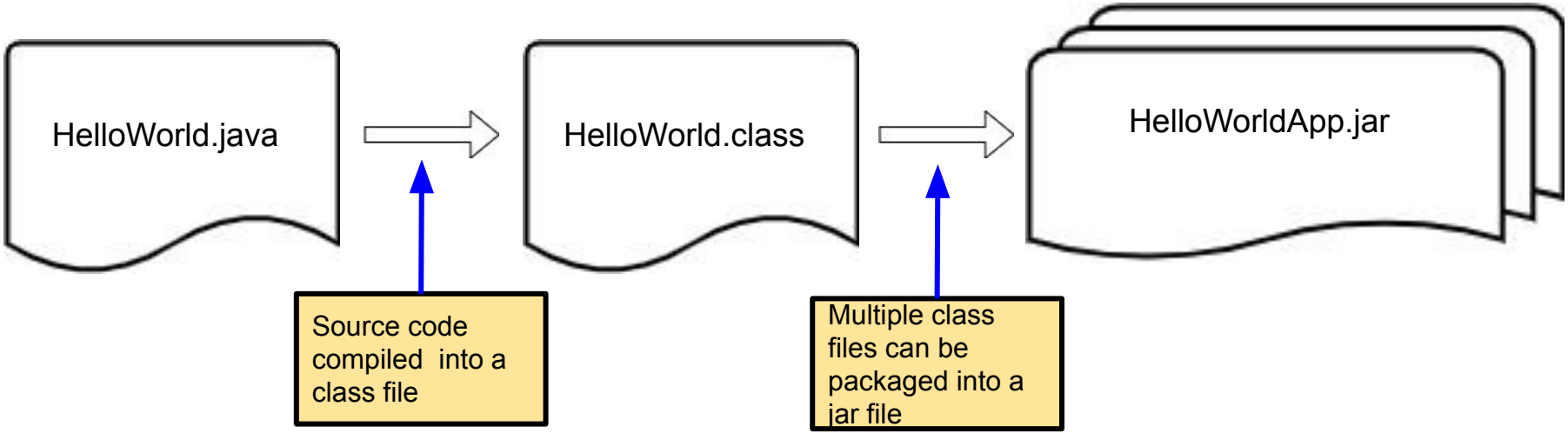
- **git status**: See the current status of your files.
- **git add -A**: Stage any files you have changed.
- **git commit -m “Commit message”**: Save files to your local repository.
- **git push origin main**: Push committed changes to network repository.



Day 1: Source Control

- **git clone**: Pulls the entire repository (including all previous commits) to your local workstation.
- **git pull upstream main**: Pulls latest changes from the remote repository.
- In this class we make a distinction between “upstream main” and “origin main”. Always pull from upstream main and push to origin main! There are some circumstances where this will change - the instructor will let you know.

Day 2: Java Development Workflow

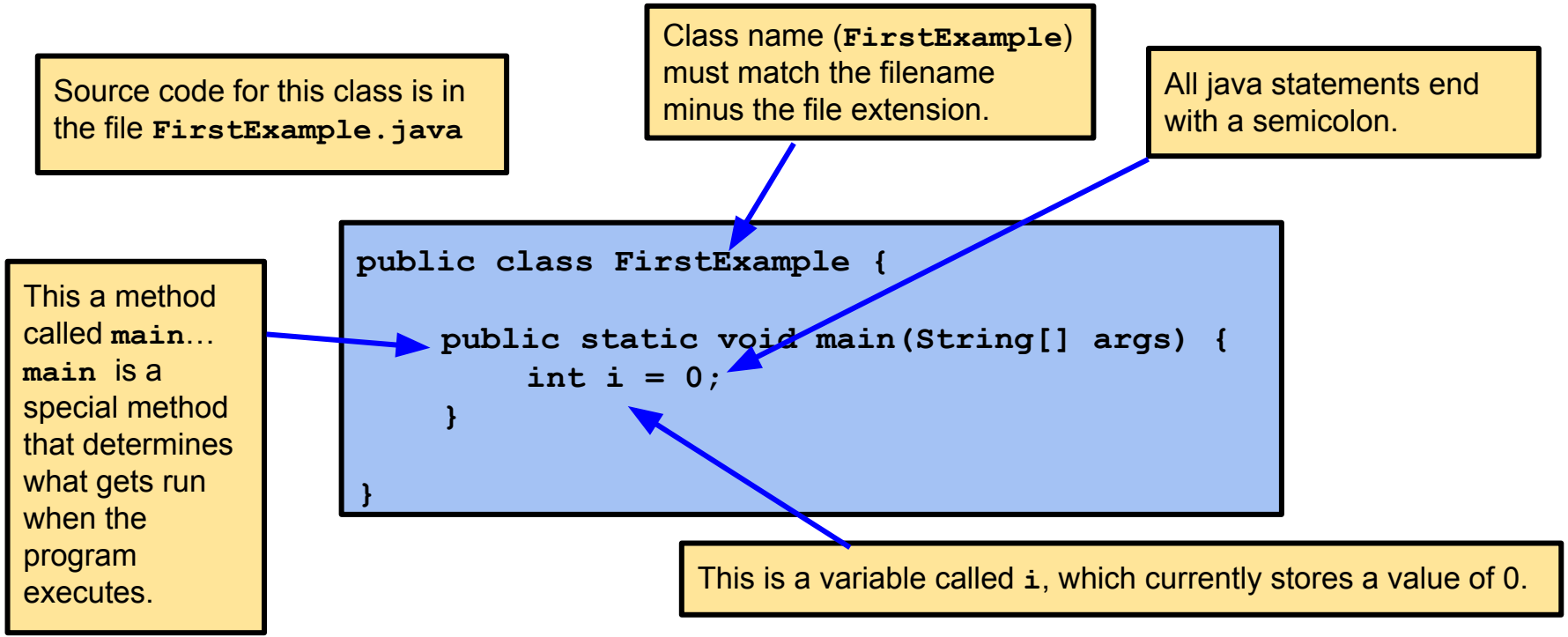


Compiling is the process by which source code (in this case the file HelloWorld.java) is transformed into a language the computer can understand.

In the past, the command **javac** was used to compile code, since the advent of modern development tools, manually typing this command is no longer needed.

Day 2: Java Program Structure

- The main unit of organization in Java is a class. Here is a simple example:



Day 2: Data Types

Data Type	Description	Example
int	Integers (whole numbers)	int i = 1;
long	Long integers (whole numbers) that can hold larger numbers than int	long l = 5000000L; // Note that to declare a long you must explicitly state that the number is a long by appending the L.
double	Decimals	double x = 3.14;
float	Also decimals, but older format. Avoid, use doubles instead.	float x = 3.14f; // Note that to declare a float you must explicitly state that the number is a float by appending the f.
char	One character. **Note the single quotes**.	char myChar = 'a';
String	A bunch or characters. **Note the double quotes**.	String myName = "Horatio"; String mySentence = "This is a beautiful day.";
boolean	true or false	boolean isItTurnedOn = true; boolean paidBills = false;

Day2: Java Variables: Rules & Conventions

Conventions:

- Ideally, variables should be named using “Camel Case.”
  - Examples of variable names:
    - playerOneScore
    - cityTemperature
    - shirtSize
- Ideally, variables should never start with an upper case.
- Variable names should be descriptive and of reasonable length.

Rules:

- Variables can begin with an underscore (_), a dollar sign ($), or a letter.
- Subsequent characters can be letters, numbers, or underscores.
- Variable names cannot be the same as java keywords. https://www.w3schools.in/java-tutorial/keywords/

Day 2: Expressions

- An expression is statement of code which can be evaluated to produce a result.
- These are the basic operators:
  - + (addition)
  - - (subtraction)
  - * (multiplication)
  - / (division)
  - % (modulo, aka remainder)
- Operator/assignment operators:
  - i += 5; i -= 5; i *= 5; i /= 5; i %= 5;
- Order of operations (Please Excuse My Dear Aunt Sally):
  - Anything inside parentheses first.
  - Exponents
  - Multiplication
  - Division
  - Addition
  - Subtraction

Day 2: Working with Numbers

ints, doubles and floats can be used together in the same statement, but Java will apply certain rules:

- Bytes and Shorts are automatically promoted to an int, when used together.
- A long with anything else will become a long.
- A double with anything else becomes a double.
- We can overcome type incompatibility by doing a cast.
  - + (addition)
  - - (subtraction)
  - * (multiplication)
  - / (division)
  - % (modulo, aka remainder)
- Operator/assignment operators:
  - i += 5; i -= 5; i *= 5; i /= 5; i %= 5;
- Order of operations (Please Excuse My Dear Aunt Sally):
  - Anything inside parentheses first.
  - Exponents
  - Multiplication
  - Division
  - Addition
  - Subtraction

Day 3: Java Statements

- Statements are roughly equivalent to sentences in natural languages.
- A statement forms a complete unit of execution.
- Some expressions can be made into a statement by terminating the expression with a semicolon (;).
  - Assignment expressions
    - aValue = 8933.234;
  - Any use of ++ or --
    - aValue++; // (more on this shortly)
  - Method invocations
    - System.out.println("Hello World!");
- Declaration Statements
  - double aValue;
- Control flow statements
  - (... more on these later this week)

Day 3: Blocks

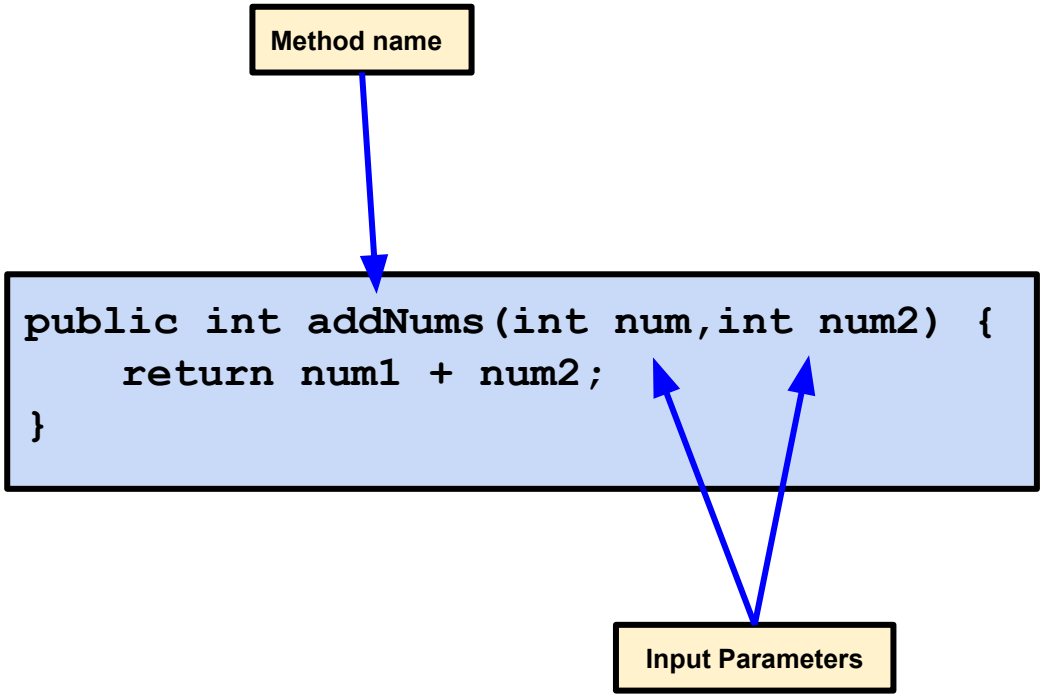
- Code that is related (either to conform to the Java language standard or by choice) is enclosed in a set of curly braces (`{ … }`). The contents inside the curly braces is known as a "**block**."
- A block is a group of zero or more statements between balanced braces and can be used anywhere a single statement is allowed.
- Blocks are used in:
  - Conditional Statements (we will talk about this today)
  - Methods (ditto)
  - Loops
- Blocks allow multiple statements where normally only one is allowed.

Day 3: Method Signature

Methods have a signature, which is made up of:

- Name (should be descriptive)
- Input Parameters

- **Return Type is not considered part of the identifying signature**



Day 3: Conditional Statements

- A conditional statement allows for the execution of code only if a certain condition is met. The condition must be, or must evaluate to a boolean value (true or false).
- The if statement follows this pattern:

```
if (condition) {
// do something if condition is true.
}
else {
// do something if condition is false.
}
```

Day 3: Comparison Operators

The following operators allow you to compare numbers:

- == : Are 2 numbers equal to each other?
- != : Are 2 numbers not equal to each other?
- > : Is a number greater than another number?
- < : Is a number less than another number?
- >= : Is a number greater or equal to another number?
- <= : Is a number less than or equal to another number?

Day 3: Ternary Operator

The ternary operator can sometimes be used to simplify conditional statements.

The following format is used:

- (condition to evaluate) ? //do this if condition is true : //do this if condition is false;

You can assign the result of the above statement to a variable if needed. The data type of this variable would be what the statements on both sides of the colon resolve to.

Day 3: AND / OR: Truth Table

We evaluate AND / OR using truth tables:

- For AND statement:
  - True AND True is True
  - True AND False is False
  - False AND True is False
  - False AND False is False
- For OR statement:
  - True AND True is True
  - True AND False is True
  - False AND True is True
  - False AND False is False

A	B	!A	A && B	A || B	A ^ B
TRUE	TRUE	FALSE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	FALSE	TRUE	TRUE
FALSE	TRUE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	TRUE	FALSE	FALSE	FALSE

Day 3: AND / OR: Exclusive OR

There is a third case called an “Exclusive Or” or XOR for short. The operator is the carrot symbol (^).

- For XOR statements:
  - True XOR True is False
  - True XOR False is True
  - False XOR True is True
  - False XOR False is False

In most day to day programming, XOR is not used very often.

Day 4: Loops

There are several types of loops in Java:

- For Loop (by far the most common)
  - for (int i = 0; i <= 10; i++)
  - Used when number of elements can be know ahead of time

- While Loop
  - Executes as long as a condition is true

- Do-While Loop
  - Same as while loop but guaranteed to run at least once

Day 4: Variable Scope

A variable's **scope** defines where in the program a variable exists (i.e. can be referenced). When code execution reaches a point where a variable is no longer referenceable, the variable is said to be **out of scope**.

Rules of Scope:

1. Variables declared inside of a function or block `{ . . }` are local variables and only available within that block. This includes loops.

2. Blocks can be nested within other blocks. Therefore, if a variable is declared outside of a block, it is accessible within the inner block.

Day 4: Arrays

Declaration:

- `int [] team1Score = new int [4];`
- `int[] team1Score = {5, 4, 3, 2};`

Assignment:

- `team1Score[0]= 20;`

Iterating:

- Can use array length property to create for loop to iterate through all elements in the array.

Day 4: The Increment/Decrement Operator

- The increment (++) and decrement operator (--) increases or decreases a number by 1 respectively.
- If the operator is behind a variable it is a postfix operator (i.e. x++).
  - A variable with a postfix operator is evaluated first, then incremented.
- If the operator is in front a variable it is a prefix operator (i.e. ++x).
  - A variable with a prefix operator is incremented first, then evaluated.
- Choosing prefix vs. postfix can affect code differently

Day 5: Methods

- General syntax:
  <access Modifier> **<return type>** **<name of the method>** (... arguments…) {
  // method code.
  }
- The return type can be one of the data types (**`boolean`**, **`int`**, **`float`**, etc.) we have seen so far.
- If the return type is **`void`** it means nothing is returned by the method.
- Methods can be called from other methods.
- Once a method has been defined, it can be called from somewhere else.

Day 5: Command Line Input/Output

- Scanner class can be used to read command line input.
- Declaring:
  - `Scanner userInput = new Scanner(System.in);`
- Using:
  - `String name = userInput.nextLine();`:
  - There are methods to read `ints`, `longs`, etc but they should be avoided
    - `nextLine()` removes carriage return from buffer but others do not so won't always wait for next input
    - Use `nextLine` and parse `Strings` to numbers using wrapper class if needed
      - `int height = Integer.parseInt(heightInput);`
  - Can use String split method to parse values delimited by spaces into a String array
    - `String [] inputArray = lineInput.split(" ");`

Day 6: Objects, the Stack, and the Heap

- **Primitive variables** exist in memory in containers sized to fit their max values in an area known as the **Stack**.
- **Non-primitive variables** (objects) exist on the **heap**.
- A **class** is a grouping of variables and methods in a source code file that we can generate objects out of.
- An **object** is an in-memory **instance** of the class "blueprint".
  - Can have state (properties) and behavior (methods),
  - Each **instance** will have its own and behavior.
  - Use **`new`** keyword to create an object
    - **`String aString = new String("Hello");`**
  - Variables that don't reference any memory are set to value **`null`**.

Primitive Types vs. Object References

Primitive Types

```
public void createData() {
      int age = 40;
      char grade = 'A';
}
```

stack

age	40
grade	A

Object (Reference Type)

```
public static void createBooks() {
      Book firstBook = new Book("Fly Eagles, Fly");
      Book secondBook = new Book("Gritty's Big Day");
}
```

instantiation creates objects on the heap

stack

firstBook	1AFF8
secondBook	1B00A

stack holds address where object exists in the heap

heap

firstBook

title	Fly Eagles, Fly

secondBook

title	Gritty's Big Day

Day 6: Strings Methods

- length()
  - returns int indicating length of String
- charAt(index)
  - returns the char at the specified index
- indexOf(searchdata)
  - returns the starting position of a character or String
  - index is 0 based
  - returns -1 if searchdata is not found in String
- contains(searchdata)
  - returns true if searchdata found in String, false otherwise.
- substring(startIndex, stopIndex)
  - returns substring of String starting at startIndex and stopping at (but not including) stopIndex
- toLowerCase() / toUpperCase()
  - returns String in all lower/uppercase

Day 6: Immutability/Equality

- Strings are immutable - once created they cannot be changed. The result of the substring operation has no bearing on the original String.

- The only way to get a new String value containing the smaller String is by re-assigning myString using the = operator to a new variable.

- When we need the String to update its value we can assign the result back to the same variable
  - `myString = myString.substring(0, 6);`
  - This still creates a new String but updates the variable to reference the new String
- For Strings (and other objects) == indicates reference comparison and equals(object) is a content comparison

Day 6: BigDecimal

`BigDecimal` objects can be created using **`new`** and a parameter such as a **`String`**

`BigDecimal` objects can also be created using the **`BigDecimal`** static method **`valueOf`** with a **`double`** value as a parameter. This should be used rather than using **`new`** with a **`double`** parameter to avoid prevision problems inherent in **`doubles`**

```
BigDecimal bigDecimalFromString = new BigDecimal("0.04");
BigDecimal bigDecimalFromDouble = BigDecimal.valueOf(0.03);

BigDecimal difference = bigDecimalFromString.subtract(bigDecimalFromDouble);
```

`BigDecimal` objects perform math operations using object methods such as **`add`**, **`subtract`**, **`multiply`**, **`divide`**, **`pow`**.

Day 7: Lists

- **`List`**:
  - Is a Collection of Objects of the same type
    - The Collection type must be an object.
    - We can use **wrapper classes** if we need to add primitive types to a **`List`**.
  - Is **Zero-indexed**, similar to arrays
  - Is an **ordered set of elements** accessible by index
  - **Allows duplicates**
  - **Can grow and shrink** as elements are added and removed
    - Methods: **`add()`** and **`remove()`**

- List<T> : <T> indicates we must specify the class. A List can be of any object type but we must specify the type.

- In Java, we use an **`ArrayList`** (object) to implement a **`List`** (interface)

Day 7: Lists

- **`List`** methods:
  - **`add(object)`**
  - **`add(index, object)`**
  - **`remove(index)`**
  - **`contains(object)`**
  - **`indexOf(object)`**
  - **`toArray(object[])`**
    - If the passed array doesn't have enough space, a new array is created with same type and size of given list.
  - **`Collections.sort(List)`**
  - **`Collections.reverse(List)`**

- Can iterate through elements using for-each loop

Day 7: Primitive Wrapper Classes

- Lists and other collections can hold objects.
  - Wait… what if I want a list of ints? Or floats? Or doubles?

- Java has a wrapper class for each primitive data type.

Java Primitive Type	Wrapper Class	Constructor Argument
`byte`	`Byte`	`byte` or `String`
`short`	`Short`	`short` or `String`
`int`	`Integer`	`int` or `String`
`long`	`Long`	`long` or `String`
`float`	`Float`	`float`, `double`, or `String`
`double`	`Double`	`double` or `String`
`char`	`Character`	`char`
`boolean`	`Boolean`	`boolean` or `String`

Day 7: Autoboxing and Unboxing

- **Autoboxing** is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes.
- **Unboxing** is converting an object of a wrapper type to its corresponding primitive value.
- The Java compiler applies unboxing when an object of a wrapper class is:
  - Passed as a parameter to a method that expects a value of the corresponding primitive type.
  - Assigned to a variable of the corresponding primitive type.

More info about Autoboxing/Unboxing can be found at:

https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html

Day 7: For-Each Loops

- For-each is a loop specifically created to iterate through collections.
- Cannot modify contents of the collection used by the for-each loop during iteration.
- Useful to work with the elements when we don't need to know index. When working with collections, the content is usually what is most useful, not the actual index position.

```
for (String currentString : myList) {
    // do something with currentString
    // (which represents the current
    // element in the loop
}
```

Day 7: Queues & Stacks

- **`Queue`** is a collection that provides additional functionality when adding and removing items that operates as a **FIFO (First In, First Out)** structure

- **`Stack`** is a collection that provides additional functionality when adding and removing items that operates as a **LIFO (Last In, First Out)** structure

Day 8: Maps

- **`Map`**:
  - Is a Collection of keys and objects
    - keys must all be the same object type
    - values must all be the same object type
    - key do NOT need to be the same object type as values
    - We can use **wrapper classes** if we need to add primitive types to a **`List`**.
  - HashMap<T, T> is an unordered form of Map
  - Elements accessible by key
  - Allows duplicates in values but **keys must be unique**
    - If a duplicate key is added, the original value associated with it will be updated
  - **Can grow and shrink** as elements are added and removed
    - Methods: **`put(<T>, <T>)`** and **`remove(<T>, <T>)`**

Day 8: Maps

- Declaring a Map
  - `Map<Integer, String> myMap = new HashMap<Integer, String>();`
- **`Map`** methods:
  - `put(key object, value object)`
  - `remove(key object)`
  - `get(key object)`
  - `contains(key object)`
  - `keySet()`
  - `values()`
  - `entrySet()`

- Can iterate through elements for-each loop with **`keySet`** or **`EntrySet`**

Day 8: Sets

- **`Set`**:
  - Is a Collection of Objects of the same type
    - The Collection type must be an object.
    - We can use **wrapper classes** if we need to add primitive types to a **`List`**.
  - Is an **unordered set of elements** accessible by object
  - **Does NOT allow duplicates**
  - **Can grow and shrink** as elements are added and removed
    - Methods: **`add()`** and **`remove()`**

- Set<T> : <T> indicates we must specify the class. A Set can be of any object type but we must specify the type.

- In Java, we use an **`HashSet`**(object) to implement a **`Set`**(interface)

Day 8: Maps

- Declaring a Set
  - `Set<Integer> primeNumbersLessThan10 = new HashSet<>();`
- **`Map`** methods:
  - `add(object)`
  - `remove(key object)`
  - `contains(object)`

- Can iterate through elements using for-each loop

Day 8: Making the Decision: Arrays vs Lists vs Maps vs Sets

- Use **Arrays** when … you know the maximum number of elements, and you know you will primarily be working with primitive data types**.
- Use **Lists** when … you want something that works like an array, but you don't know the maximum number of elements.
- Use **Maps** when … you have key value pairs.
- Use **Sets** when … you know your data does not contain repeating elements.

** This "rule" is debatable in that you can declare Object[] arrays; they have their place but List<T> is far more common and meets the majority of use-cases.

Day 9: Classes

- A **class** is a blueprint to create an object
  - Specifies **state**/variables
  - Defines **behavior**/methods
- Class Naming
  - Use singular nouns, not verbs
  - Class must match the file name
  - Use Pascal casing
  - A Fully Qualified Name is unambiguous and includes the package and class name

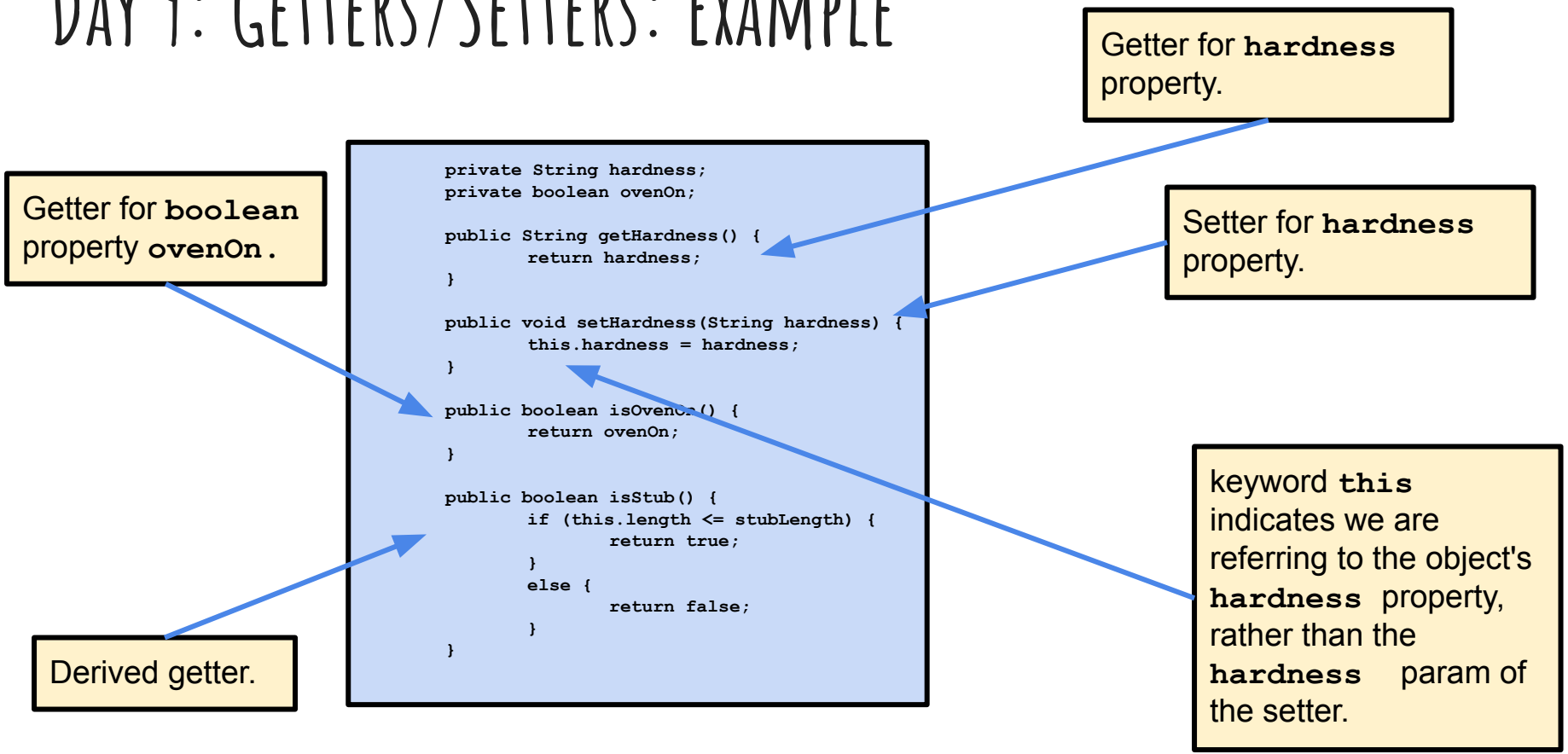
Day 9: Properties & Methods

- Instance properties are variables that maintain an objects state.
- Instance methods are methods that provide an object's behavior.
- Each instance of a class has its own copies of properties and methods
- Access modifiers indicate who can access properties or methods in a class
  - **`public`**
    - Can be accessed by any other object
  - **`private`**
    - Can only be accessed by the current instance of a class

Day 9: Getters/Setters

- To follow the encapsulation principle, instance variables are often declared **`private`** so that only the current instance of an object can access its internal state.
- **`public`** **getters and setters** can be used to provide access to **`private`** properties to other objects. We can provide none, either, or both for any **`private`** properties to manage access by other objects.
- Getters typically take no parameters and return the value of a property.
- Setters typically take a value which is used to set the desired property.
    - The keyword **`this`** represents the current object and can be used to resolve ambiguity between object properties and setter variables when their names are the same.
- Getters and setters should follow the following naming conventions as the conventions are leveraged by many Java extensions to implement behavior:
    - Getters should prepend the word **`get`** to the property name and the resulting method name should be camel-cased: getter for **`personAge`** would be **`getPersonAge`**
        - getters for **`boolean`** properties should start with **`is`** rather than **`get`**: **`isOvenOn`**
    - Setters should prepend the word **`set`** to property name and the resulting method name should be camel-cased: setter for **`personAge`** would be **`setPersonAge`**
- Getters and setters do not need to directly correspond to properties. Those that compute the resulting value rather than being tied directly to a property are called **derived getters/setters**.

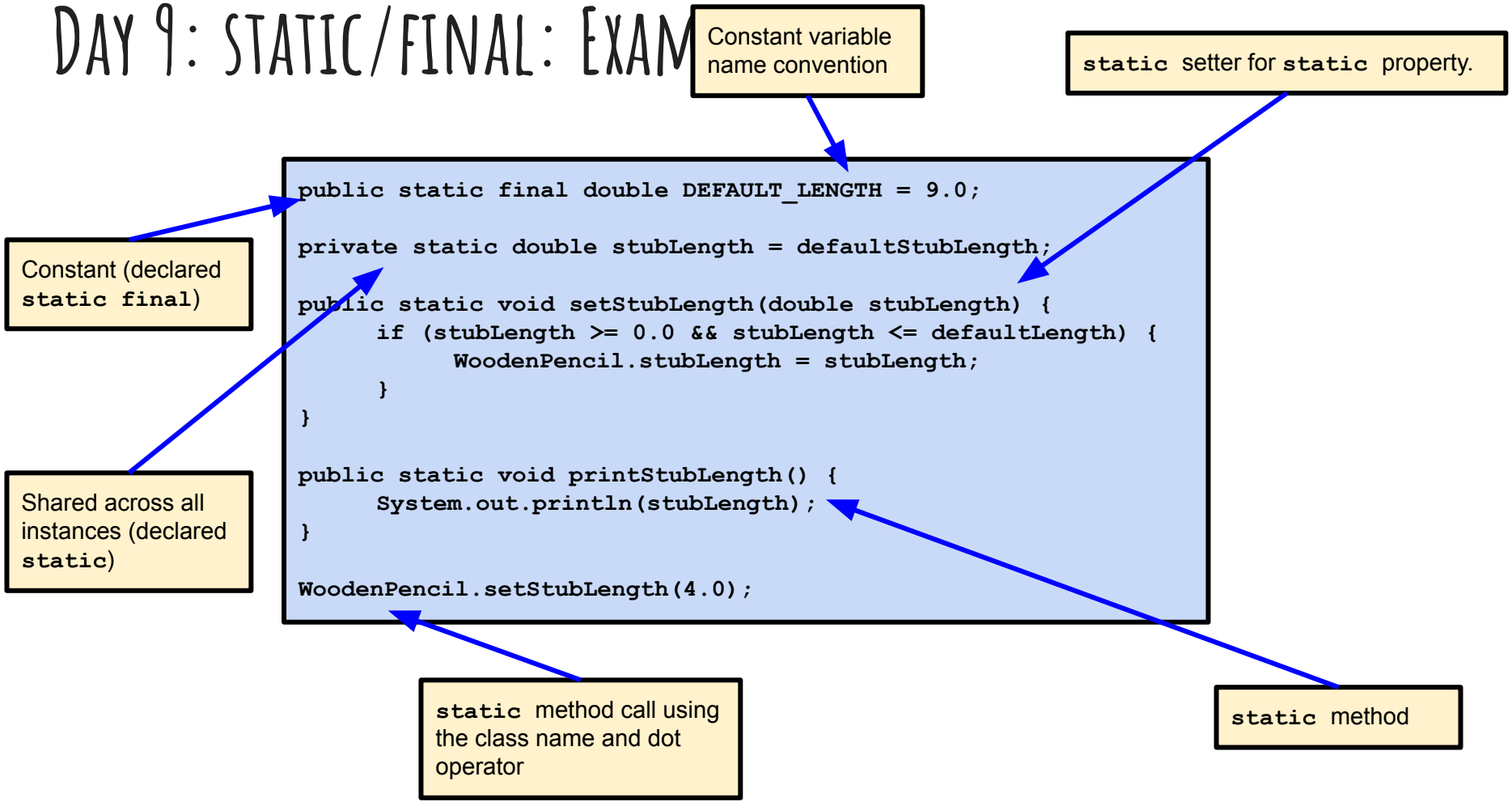
Day 9: Getters/Setters: Example



Day 9: Special Cases: static and final

- The keyword **`static`** indicates that a property or method is shared across all instances of a class. If a method or property is declared as **`static`**, it means there is **NOT** a separate copy of it for each instance of the object - rather there is one copy shared by all instances.
- Since methods and properties marked **`static`** are not specific to an instance, **`static`** methods can only access the **`static`** properties and methods of a class.
- **`static`** methods and properties which are **`public`** can be accessed using the class name and the dot operator. They can also be accessed through a specific instance of an object but the convention is not to do so because the property or method is not tied directly to the instance.
- If the modifier **`final`** is added to a property, the property can't be modified.
- **Constants** in Java are declared using both **`static`** and **`final`** since there is only one copy of the property in the class and the property should not be modified.
    - Constants should be declared using all caps with underscores between words rather than using camel-case: **`DEFAULT_LENGTH`**

Day 9: static/final: Exam




```
public static final double DEFAULT_LENGTH = 9.0;

private static double stubLength = defaultStubLength;

public static void setStubLength(double stubLength) {
      if (stubLength >= 0.0 && stubLength <= defaultLength) {
            WoodenPencil.stubLength = stubLength;
      }
}

public static void printStubLength() {
      System.out.println(stubLength);
}

WoodenPencil.setStubLength(4.0);
```

Day 9: Constructors

- When an object is created, its **constructor** is called before its reference is returned. You can think of it as an initializer.
- Constructors have the same name as the class and have no return value.
- They can have parameters, but are not required to.
- Every class has a built-in no-argument constructor which is called when the object is created but as soon as any other constructor is defined in a class, the built in one is no longer available. This means that if you define a constructor with parameters and still want the ability to create an object with a no-argument constructor, you have to define the no-argument constructor manually in your class.
- Constructors can call other constructors using this as the method name.

Day 9: Constructor: Exam

No-arg constructor. This is normally built in to a class but can be overridden to add functionality.

Uses `this` to call the constructor with args.

```
public WoodenPencil() {
      // standard wooden pencils are assumed to be unsharpened when first instantiated (i.e. 0.0)
      this(defaultLength, defaultShape, defaultHardness, defaultColor, 0.0);
}

public WoodenPencil(double length, int shape, String hardness, Color color, double sharpness) {
      this.length = length;
      // more code
}

//

WoodenPencil pencil = new WoodenPencil();

WoodenPencil pencilWithValues = new WoodenPencil(5.9, 4, "hardness", WoodenPencil.defaultColor,
7.0);
```

Constructor overloaded with params. Once this is defined, the built in no-arg constructor is no longer available and must be redeclared if needed. Note this is used to initialize the properties when created.

Creating using constructor with args.

Creating using no-arg constructor