Introduction to Java

Java is a high-level, compiled, strongly typed object-oriented programming (OOP) language. The advantages of Java are many: it is platform independent, has a C-language inspired syntax, provides automatic memory management, has an extensive built-in runtime library, is supported by the Oracle Corporation, and has a rich open source community.

When we say Java is *object-oriented*, we mean that it has the constructs of **classes** and **objects** built into the language. It also allows us to use various principles of object-oriented programming, which are covered in a separate module. An object in code can represent a real-world entity, or a conceptual entity. Classes are the *blueprints* for how to create objects that contain a certain **state** - which is represented by *fields* (variables) - and **behavior** - which is defined via *methods*. Objects are instances of class definitions. However, Java is not 100% object-oriented because it still has **primitive values** (or just: primitives), which are defined below:

Primitive type	Size	Description
`boolean`	not specified (JVM-dependent)	represents true and false values
`byte`	8-bit	numerical, integral value
`short`	16-bit	signed numerical, integral value
`char`	16-bit	unsigned numerical, Unicode character
`int`	32-bit	numerical, integral value
`long`	64-bit	numerical, integral value
`float`	32-bit	floating point value
`double`	64-bit	floating point value

JVM, JRE, JDK

When we say Java is platform independent, we mean that Java programs are not constrained to a single operating system - Java code can be run on many different systems. Another way of saying this is that with Java, we can **write once, run anywhere** (WORA). Portability is possible because Java code is compiled to **byte code** and runs on a **JVM - or Java Virtual Machine**. The JVM is specific to the operating system - there is a JVM for Windows, one for Mac, one for Linux, etc. The JVM reads the compiled Java byte code and translates it to machine code to be executed on the given system. Actually, in order to run Java code, you also need a **JRE - Java Runtime Environment**, which contains all the runtime libraries that your code will be calling and using. The JRE contains the JVM within it, so if you want to run a Java program, all you need to install is the JRE.

But how do we actually compile the Java code that we write down to byte code that the JVM will understand? For that, you need a **JDK - Java Development Kit**, which provides developer tools like a compiler, debugger, documentation tools (`javadoc`), and other

command-line utilities. The JDK also has a JRE inside of it, so if you install a JDK you can compile your Java code as well as execute it.

So, to recap - the JDK contains tools for Java development as well as a JRE, which contains the built-in Java libraries as well as a JVM, which actually executes your Java byte code and runs it on the specific operating system it is installed upon.

Class vs. Object

In object-oriented programming, a class is a blueprint for how to create an object. We instantiate objects from their class using a constructor.

Class	Object
Class is declared using `class` keyword. For example, `class Student{}`	Object is created through `new` keyword. For example, `Student s1=new Student();`
Object is created many times as per requirement.	Class is declared once.
Object allocates memory when it is created.	Class doesn't allocated memory when it is created.

The Four Pillars of OOP

There are four principles that are often referenced when talking about object-oriented programming. They are as follows:

- Abstraction
- Polymorphism
- Inheritance
- Encapsulation

An easy way to remember these is with the acronym **A PIE**. These concepts provide guidance when designing object-oriented programs. Let's walk through each of these "pillars" of OOP one by one.

Inheritance

We start with inheritance as it is easiest to understand conceptually. In Java (and other OOP languages), classes contain a blueprint for the state and behavior of objects. Most languages have a method whereby classes (and thus, objects) can **inherit** the state and behavior (read: fields and methods) of other classes. The class from which other classes

inherit from is called a "base" or "parent" class, and the class which inherit the parent is called a "child" or "sub"-class.

In Java, all non-`private` fields and methods are inherited from a parent class when using the `extends` keyword in the class declaration. For example, we could have an `Animal` class as our base class which has a field `numOfLegs` and a method `speak`, and a `Dog` class which extends `Animal`. Thus `Dog` will also have the `numOfLegs` field and `speak` method:

```
public class Animal {
  public int numOfLegs;

  public void speak() {
    System.out.println("*generic animal noise*");
  }
}

public class Dog extends Animal {}
```

`Dog` will get the `speak` method from `Animal`, but can **override** it and implement its own method which is specific to `Dog` if desired. This will be covered later with polymorphism.

The benefit of inheritance lies mainly in the **re-usability of code** - our `Dog` class does not need to re-declare the methods and fields it gets from `Animal`. Thus, we can abide by the common programming guideline **DRY (don't repeat yourself)**.

Interfaces and the Diamond Problem

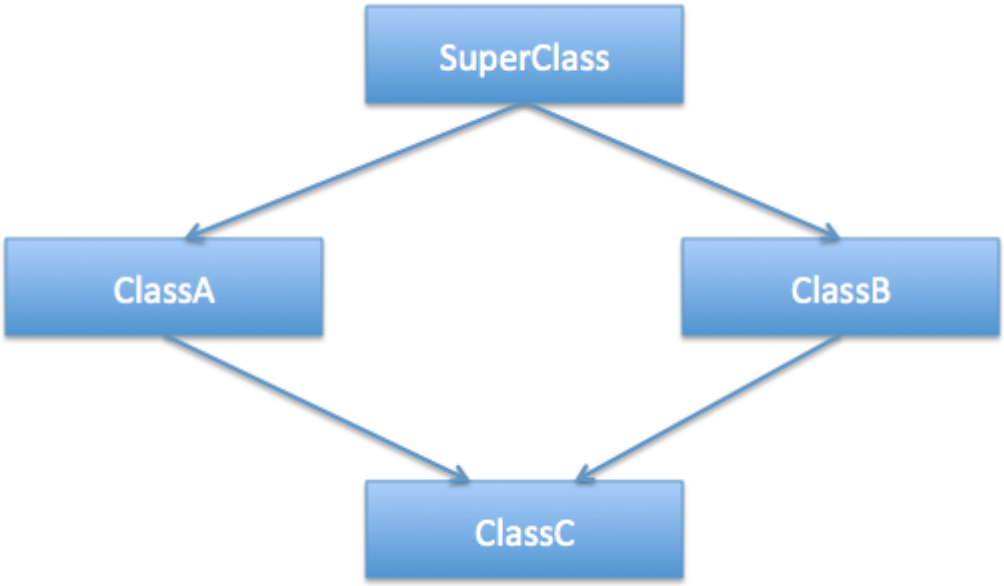
Another way for classes to inherit is through **interfaces**. Interfaces are like a contract which specify behaviors that other classes must provide. Interfaces only supply method signatures which other classes must implement and provide method bodies for. Thus, methods in interfaces are implicitly `public` and `abstract`, even when those keywords are not used. Interfaces can also have variables, which are implicitly `public static final` - meaning they are attached to the interface itself and are constants that cannot be reassigned. Below is an example of an interface:

```
public interface Purchasable {
  int maxPrice; // this is implicitly static!

  boolean buy(Item i); // NOTE: no method body. This method is public abstract even t
hough we did not use the keywords
}
```

In Java, classes can only extend from one other class - **there is no multiple inheritance via classes. However, we *can* have multi-level inheritance** - for example, class `C` extends `B` which extends `A`. If multiple inheritance were allowed, imagine what would happen if a class were to extend multiple other classes which both contain the same

method. Which method implementation would the subclass inherit? This ambiguity is called the "diamond problem", shown below.



In Java we can have multiple inheritance and still bypass the diamond problem by using interfaces. A class can implement an arbitrary number of interfaces. There is no ambiguity because interfaces do not have method bodies (with the exception of Java 8 `default` and `static` interface methods) - the implementing class resolves any ambiguity because it must implement the methods defined in the interfaces.

Abstraction

Abstraction is a programming principle in which we centralize common characteristics and generalize behavior into conceptual classes. In the example above, the `Animal` class contains characteristics and behaviors common to all animals.

Through abstraction, we hide underlying complexity through a simplified interface. Think of a car - you do not need to know how the car works, just how to use the accelerator, brake, and steering wheel. Thus, a car "abstracts" away the internal details of the engine, motor, driveshaft, and other parts. If our `Animal` class were part of a library for creating animals in Java, the user of the library wouldn't need to know exactly how each animal speaks, because the `speak` method is defined on the `Animal` class. We can also use the generic `Animal` type for reference variables without worrying about which specific animal the object is.

```
Animal a = new Dog();

a.speak();
```

In the example above, `a` is of type `Animal`, not `Dog`. The advantage of writing code this way is that later if we decide we instead need to create a `Cat` here, we can easily change the

constructor being invoked. Because the `.speak()` method belongs to the `Animal` class, it is guaranteed to exist for all animals. (Here we are assuming `Cat` would extend `Animal`.)

In Java, we achieve abstraction through abstract classes and interfaces. Abstract classes are classes which cannot be instantiated and are declared with the `abstract` keyword. Generally, it wouldn't make sense to instantiate instantiate our `Animal` class - instead, we use it as a general type and instantiate specific animals. So we could make the `Animal` class abstract instead of concrete.

Abstract Classes and Interfaces

Abstract classes, as mentioned above, are more general classes which cannot be instantiated. They instead act as templates for other classes to extend from. Abstract classes can have both concrete and abstract methods - the `abstract` methods must be implemented by concrete subclasses.

Interfaces also cannot be instantiated. They instead serve as contracts for methods that classes must implement. In order to inherit from interfaces, a class declares that it `implements` some interface, or multiple interfaces. Methods declared on an interface are implicitly `public` and `abstract`. Interfaces can have variables, but they are implicitly `public`, `static`, and `final`. Since Java 8, interfaces can also provide method implementations if the method is marked `static` or `default`.

Abstract classes are better suited for defining *common characteristics* of objects and are named as nouns by convention, whereas interfaces are better for defining common *behavior* the implementing class promises to provide.

Polymorphism

By definition, **polymorphism** means "taking on many forms". In the realm of programming, it describes how objects can behave differently in different contexts. The most common examples of polymorphism in Java are method overloading and overriding.

Method Overloading

Method overloading is when there exist two or more methods in a class with the same method name, but different method signatures by changing the parameter list. For example:

```
public class MyClass {
  public static void main(String[] args) {
    MyClass mc = new MyClass();
        mc.foo("str", 1); // prints "version 1"
        mc.foo("str");    // prints "version 2"
  }
  public void foo(String a, int b) {
    System.out.println("version 1");
  }
```

```
  public void foo(String a) {
    System.out.println("version 2");
  }
}
```

We can change the number of parameters (like above), the types of the parameters, or the order in which the parameters are passed. Which version of the method is executed is determined by the parameters passed when the method is invoked. Note that varying the return type of the method is **not** permitted - this will not compile.

Because the argument list is known at compilation, the compiler knows which version of the method will be executed. Therefore, method overloading is a type of **compile-time** - or **static** - polymorphism.

Method Overriding

Method overriding is when a method in a child class has the same method signature as a method in the parent class, but with a different implementation. Thus, child classes can change the default behavior given by a parent's method. Overriding methods makes class hierarchies more flexible and dynamic. Let's change our `Dog` class we used before and have it override the `speak` method from `Animal`:

```
public class Dog extends Animal {
  public static void main(String[] args){
    Animal d = new Dog();
        d.speak(); // Woof!
  }

  public void speak() {
    System.out.println("Woof!");
  }
}
```

When we declare the reference variable `d`, it uses the class `Animal` as its type but refers to an actual *instance* of a `Dog` in memory. So we can call the `speak` method declared in `Animal`, but since `d` refers to a `Dog` object, it will use that `Dog`'s implementation of `speak`. This is referred to as **virtual method invocation** and is key to method overriding. The `Animal` class (if it is abstract) does not even need to define any implementation for `speak`, since the method to be executed will be determined at runtime depending on the object referred to in memory. This is the reason why method overriding is classified as **runtime** - or **dynamic** - polymorphism.

One more item to note with method overriding is that `static` methods cannot be overridden. Instead, if a subclass implements the same `static` method as its parent, the method is hidden. **Method hiding** replaces the parent method in the calls defined in the child class.

Covariant return types

When overriding a method, we also have the option of changing the return type - *provided that the overridden return type **is a subtype of the original type***. This is called **covariant return types**. We can also choose to change the access modifier of an overridden method - *provided that the new modifier for the overriding method provides **more, not less, access** than the overridden method*. An example is below:

```
public class Thing {}
public class SpecificThing extends Thing {}


public class Foo {
  protected Thing get() {
    return new Thing();
  }
}


public class Bar extends Foo {
  public SpecificThing get() {
    return new SpecificThing();
  }
}
```

Encapsulation

Finally, **encapsulation** is the OOP principle of containing related state and behavior together inside a class, and also hiding and preventing change to an object's data members. When an object is encapsulated, it controls the access to its internal state. This prevents arbitrary external interference, which could bring the object into an invalid or inconsistent state.

In Java, encapsulation is achieved through declaring data members as `private`, with public "getter" and "setter" methods.

```
public class Person {
  private int age;


  public int getAge() {
    return this.age;
  }
```

```
  public void setAge(int age) {
    if (age > 0) {
      this.age = age;
        }
  }
}
```

In the `Person` class above, we prevent anything outside the class from changing the `age` of the person. We allow access to the field through the `getAge()` method, and we allow manipulation of the field through the `setAge()` method. Note that inside setter methods we can and should perform validation, like we do above - checking that the input number is a valid age (greater than 0).

Stack and Heap

Inside the JVM, our application memory is divided into the "stack" and the "heap". The stack is where method invocations and reference variables are stored in stack frames. For example, when the JVM invokes the `main` method a stack frame is created for it and placed on the stack. Multiple stacks are possible - for example, when a thread is created it is given its own stack.

The heap, in contrast, is a central location in memory where all objects are stored. New objects are created via the `new` keyword and (optionally) assigned to a reference variable, which can then be re-assigned to reference different objects later. Thus, multiple reference variables could point to the same object in memory.

Note: Errors at runtime can be thrown if a program runs out of memory addresses for new stack frames (`StackOverflowError`), or if no memory is available in the heap for object creation (`OutOfMemoryError`).

Garbage Collection

Consider the following Java code snippet:

```
Object o1 = new Object(); // 1
Object o2 = new Object(); // 2
Object o3 = o1;           // 3
o2 = o3;                  // 4
```

How many objects have we created on the heap? We determine this by counting the number of `new` keywords: in this case, 2. However, we have declared 3 reference variables that point to these objects in memory. The third line does not create a new object, it simply creates a new reference variable pointing to the object created on the first line. The last line

reassigns the `o2` variable to reference the object `o3` is referencing, which is the first object created.

But what happened to the object created on the second line? After line 4, there are no reference variables pointing to it, thus it can never be used again in the program. When a condition like this occurs, the object becomes eligible for garbage collection - **the process of removing objects from the heap which have no references to them**. In lower-level programming languages, memory is manipulated directly in the code, but Java abstracts these details away from the developer by allowing the JVM to handle memory management itself. We already know that the JVM creates objects in the heap when we invoke the `new` keyword. It also will clean up objects for us, freeing up memory for new objects to be created.

As demonstrated, garbage collection is run in the background by the JVM. **There is no way we can explicitly force garbage collection to happen**, but we can *request* garbage collection to be run through the use of one of the following:

- `System.gc()`
- `Runtime.getRuntime().gc()`
- `System.runFinalize()`

Control Flow Statements

Java, like most other languages, has a few keywords that define statements which control the flow of program execution:

- `if/else if/else`
- `for`
- `while`
- `do-while`
- `switch`

If statements

The basic syntax of `if` statements:

```
if (condition) { // this is the only block that is required - others are optional
  statement1;
} else if (condition2) {
  statement2;
} else {
  statement3;
}
```

For loops

For-loops are used to iterate over data structures. They include 3 statements in parentheses - a declaration, condition, and statement (typically increment or decrement).

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

Any object which implements the `Iterable` interface can be iterated over using an **enhanced for-loop**. The syntax is given in an example:

```
List<String> myList = getListOfStrings();
for (String myStr : listOfStrings) {
  System.out.println(myStr);
}
```

While statements

While statements test a condition - if the condition evaluates to true the block of code is run, otherwise the block is skipped. The block is looped through as long as the condition remains `true`.

```
while (true) {
  // infinite loop!
}
```

An alternative to `while` loops is the `do-while` loop. This guarantees that the block will always run at least once:

```
do {
  // always runs at least once!
} while(condition); // condition evaluated after the block of code
```

Switch statements

`switch` statements attempt to match some variable with the value it contains. This type of statement works with `byte`, `short`, `char`, and `int` primitives, along with enumerated types (`enum`s) and - since Java 7 - `String`s. The basic syntax is:

```
switch(variable) {
  case 'A': System.out.println("Case A matches!"); break; // break is REQUIRED, unles
s you want control flow to "fall through" to the next case
  case 'B': System.out.println("Case B matches!"); break;
```

```
  case 'C': System.out.println("Case C matches!"); break;
  default: System.out.println("this will run if other cases don't match"); break;
}
```

Packages

```
package com.revature.mypackage;
```

This line declares the **package** in which the class will reside. This must always be the first (non-commented) line in a `.java` file. Packages are a way of organizing classes, interfaces, and enums in a hierarchical manner. Packages follow a naming convention of lowercase characters separated by periods in the reverse way you would specify a web domain - thus, `com.revature.mypackage` instead of `mypackage.revature.com`.

Also, classes can be referenced anywhere in a program by their "fully qualified class name" - which is the package declaration followed by the class, in order to uniquely identify the class. In our example, the fully qualified class name is `com.revature.mypackage.HelloWorld`.

But typically we do not want to write out a verbose package and class name together. Instead, we can use an `import` statement after our package declaration to pull in other classes. We can then just use the class name without the package. By default, everything in the `java.lang` package is imported (which gives us the `System` class we used in the example). Other packages and classes must be imported by the programmer explicitly.

Constructors

When we use the `new` keyword in order to create an object, the JVM is invoking a special class member called a **constructor**. A constructor declares how an object is to be instantiated and initialized from the class "blueprint". A constructor is declared like a method, except **its method signature does not contain a return type, and a constructor always has the same name as the class**. The new object returned from the constructor is always of the class in which the constructor is declared. A simple example is shown below:

```
public class ConstructorExample {


  int myNumber;


  public static void main(String[] args) {
    ConstructorExample ce = new ConstructorExample(3); // a
        System.out.println(ce.myNumber); // b
```

```
  }


  public ConstructorExample(int myNumber) { // c
    this.myNumber = myNumber; // d
  }
}
```

`this` keyword

When this program is run, it will print `3`. How does this happen? The constructor is defined on line "c" with one input parameter. Note that the constructor does not have a return type. On line "d", the **parameter** myNumber is assigned to the **instance variable** myNumber via the `this` keyword. `this` refers to the object which is being instantiated - it is used to initialize instance variables, or - to call other constructors (this is called constructor chaining).

When the program above is run, the `main` method is executed. On line "a", a new `ConstructorExample` object is created and assigned to the variable `ce`. The constructor is invoked with the `new` keyword and the int `3` is passed as the argument. This assigns the value 3 to the instance variable `myNumber` on the object returned, as explained above. Finally, the instance variable is printed out and the program finishes execution.

`super` keyword

There is another keyword important for constructors - the `super` keyword, which references the "super", or parent, class. When invoked as a method (`super()`), the parent class constructor will be called. A `super()` call (or a `this()` call) **must** be the first line of any constructor. If not explicitly provided, the compiler will inject it on the first line.

`Default` constructor

Earlier, in our `Example` class, we saw a constructor with no arguments used with the `new` keyword. But we didn't define a constructor, so how is this possible? It turns out the compiler will inject a "default" constructor for us if we are too lazy to define one ourselves. The "default" constructor takes no arguments and simply calls `super()` (see above) - sometimes it is referred to as the "default, no-args" constructor. However, be warned that if we define our own constructor(s) in the class, we will **not** receive a default constructor from the compiler.

Thus, the simple class:

```
public class MySimpleClass {}
```

has a default no-args constructor that can be called:

```
MySimpleClass someVariable = new MySimpleClass();
```

Methods and Parameters

A method is a block of code which only runs when it is called. You can pass data, known as parameters, into a method. A method must be declared within a class. It is defined with the name of the method, followed by parentheses `()`.

Example:

```
public class MyClass {
  public static void myMethod() {
    // code to be executed
  }
}
```

Parameters are specified after the method name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

Example:

```
public class MyClass {
  public static void myMethod(String name) {
    System.out.println(name);
  }

  public static void main(String[] args) {
    myMethod("Liam");
    myMethod("Jenny");
  }
}
//Output:
//Liam
//Jenny
```

Variable scopes

When a variable is declared in a Java program, it is attached to a specific **scope** within the program, which determines where the variable resides. The different **scopes of a variable** in Java are:

- Instance, or object, scope
- Class, or static, scope

- Method scope
- Block scope

Instance scope means that the variable is attached to individual objects created from the class. When an instance-scoped variable is modified, it has no effect on other, distinct objects of the same class.

Class scoped variables reside on the class definition itself. This means that when objects update a class-scoped variable, the change is reflected across all instances of the class. Class scope is declared with the `static` keyword. Methods can also be declared as class scope. However, `static` methods cannot invoke instance methods or variables (think about it: which specific object would they reference?). Static methods and variables should be referenced through the class directly, not through an object. For example: `MyClass.myStaticMethod()` or `MyClass.myStaticVariable`.

Method scope is the scope of a variable declared within a method block, whether static or instance. Method-scoped variables are only available within the method they are declared; they do not exist after the method finishes execution (the stack frame is popped from the stack and removed from memory after execution).

Block scoped variables only exist within the specific control flow block, of which there are several in Java: `for`, `while`, and `do-while` loops, `if/else-if/else` blocks, `switch` cases, or even just regular blocks of code declared via curly braces (`{}`). After the block ends, variables declared within it are no longer available.

Access Modifiers

Modifiers

We already saw two different **modifiers** on the `main` method above - one is an **access modifier** and the other is a **non-access modifier**.

Access Modifiers

Access modifiers are keywords which define the ability of other code to access the given entity. Modifiers can be placed on classes, interfaces, enums, and class members. The access modifiers are listed below:

Modifier	Access Level
`public`	Available anywhere
`protected`	Within the same package, and this class' sub-classes
`default`	Within the same package
`private`	Only within the same class

The `default` access level requires additional clarification - this access level is "default" because **there is no keyword** to be used. This access level is also known as "package private".

Using `private` modifiers on instance variables - along with public getter and setter methods - helps with **encapsulation**, which is one of the pillars of object-oriented programming.

Non-Access Modifiers

Java also has **non-access** modifiers which can be placed on various class members:

- `static` - denotes "class" scope, meaning the member resides on the class itself, not object instances.
  - `static` variables can be accessed through the class, e.g. `MyClass.staticVariable`
  - `static` methods can be called directly without needing an instance of the class, e.g. `MyClass.someMethod()`
- `final`
  - when applied to a variable, it means the variable **cannot be re-assigned**
  - when applied to a class, it means the class **cannot be extended**
  - when applied to a method, it means the method **cannot be overridden**
- `abstract`
  - when applied to a class, the class **cannot be instantiated** directly (instead, it should be *inherited*)
  - when applied to a method, only the method signature is defined, not the implementation. Also, the class where the method resides must also be abstract. Concrete subclasses **must implement the abstract method**.
- `synchronized` - relevant to threads and preventing deadlock phenomena (discussed in a separate module)
- `transient` - marks a variables as non-serializable, meaning it will not be persisted when written to a byte stream (discussed in another module)
- `volatile` - marks a variable to never be cached thread-locally. Obscure, rarely-used keyword.
- `strictfp` - restricts floating point calculations for portability. Obscure, rarely-used keyword.

Below is a class with various class members which have different modifiers associated with them:

```
package com.revature.mypackage;


public class Example {

    // <-- this notation specifies a comment

        public boolean a; // this variable can be accessed anywhere, on an object of type Example
```

```
        protected byte b; // accessible within com.revature.mypackage or subclasses
of Example
        static int c; // class scope with default access
        private String d; // can only be accessed within this class

        public static void main(String[] args) {
                Example myExample = new Example();
                myExample.printValues();
                // prints out the default values:
                // The value of a is: false
                // The value of b is: 0
                // The value of c is: 0
                // The value of d is:
        }

        public void printValues() {
                System.out.println("The value of a is: " + a);
                System.out.println("The value of b is: " + b);
                System.out.println("The value of c is: " + c);
                System.out.println("The value of d is: " + d);
        }
}
```