OOP

Private, Public, Static

Private

- Private access modifier is the most restrictive access level. Class and interfaces cannot be private.
- Methods, variables, and constructors that are declared private can only be accessed within the declared class itself.
- Variables that are declared private can be accessed outside the class, if public getter methods are present in the class.
- Using the private modifier is the main way that an object **encapsulates** itself and hides data from the outside world.

Public

- The public keyword is an access modifier for class, method and variable.
- It is the access modifier that has least restriction on the object it modifies.
- When a **class** is marked as public, it can be accessed from anywhere, including outside packages.
- When a **method** is marked as public, it can be invoked not only from the enclosing class, but also from outside classes.
- When a **variable** is marked as public, it can be accessed and updated from outside classes.

Static
The keyword **static** indicates that the particular member belongs to a type itself, rather than to an instance of that type. This means that only one instance of that static member is created which is shared across all instances of the class. The static keyword is used mainly for memory management purposes.

When to use "static":

- When the value of variable is independent of objects
- When the value is supposed to be shared across all objects

OOP Principles

My article about OOP: https://blog.usejournal.com/introduction-to-oop-in-java-coding-bootcamp-series-4d9f849915ef

Inheritance

This is the OOP principle that allows classes to derive from other classes. Often times, a program includes several classes that are very similar to each other—but not entirely the same. OOP programming allows us to create a parent class with the common attributes and children classes that define the specific properties.
In Java, each class can only be derived from one other class. That class is called a superclass, or parent class. The derived class is called subclass, or child class.

Use keyword "**extends**".

```
public Plant(int waterAmount, String color, String type, int waterNeeded,
double absorption) {
        this.waterAmount = waterAmount;
        this.color = color;
        this.type = type;
        this.waterNeeded = waterNeeded;
        this.absorption = absorption;
    }
```

```
public class Tree extends Plant {

    public Tree(int waterAmount, String color, String type) {
        super(waterAmount, color, type, 10, 0.4);
    }
}
```

```
public class Flower extends Plant {

    String type;
```

```
    public Flower(int waterAmount, String color, String type) {
        super(waterAmount, color, type, 5, 0.75);
    }
}
```

Encapsulation

If you have an attribute that is not visible from the outside of an object, and bundle it with methods that provide read or write access to it, then you can hide specific information and control access to the internal state of the object.
You implement this information-hiding mechanism by making your class attributes inaccessible from the outside and by providing getter and/or setter methods for attributes that shall be readable or updatable by other classes.

- Coffee machine example

Benefits of encapsulation

- The fields of a class can be made read-only or write-only.
- A class can have total control over what is stored in its fields.

Polymorphism

Polymorphism is the ability of an object to take on many forms.
Any Java object that can pass more than one IS-A test is considered to be polymorphic.

```
public class StringedInstrument extends Instrument {
    int numberOfStrings;

 public String sound() {
        return null;
    }
}
```

```
public class BassGuitar extends StringedInstrument {


 @Override
    public String sound() {
        return "Duum-duum-duum";
    }
}
```

Abstraction

Abstraction means using simple things to represent complexity. We all know how to turn the TV on, but we don't need to know how it works in order to enjoy it. In Java, abstraction means simple things like objects, classes, and variables represent more complex underlying code and data. This is important because it lets us avoid repeating the same work multiple times.

Interfaces and Abstract Classes

Interfaces
An interface is a reference type in Java. It is similar to class. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.
Interfaces are used to achieve polymorphism. They can be declared public or package private (no access modifier). Classes can implement interfaces through the keyword "**implements**". It is possible for a class to implement several interfaces, like so:

```
public class MyInterfaceImpl
    implements MyInterface, MyOtherInterface
```

An interface is similar to a class in the way that both of them can contain any number of methods.

Interfaces have the following properties:

- An interface is implicitly abstract. You do not need to use the abstract keyword while declaring an interface
- Each method in an interface is also implicitly abstract, so the abstract keyword is not needed
- Methods in an interface are implicitly public.

Abstract Classes

An abstract class is a class that is declared abstract—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed. When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, then the subclass must also be declared abstract.

Interfaces default Abstract Classes

Abstract classes are similar to interfaces. You cannot instantiate them, and they may contain a mix of methods declared with or without an implementation. However, with abstract classes, you can declare fields that are not static and final, and define public, protected, and private concrete methods. With interfaces, all fields are automatically public, static, and final, and all methods that you declare or define (as default methods) are public. In addition, you can extend only one class, whether or not it is abstract, whereas you can implement any number of interfaces.

Use abstract classes if:

- You want to share code among several closely related classes.
- You expect that classes that extend your abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).
- You want to declare non-static or non-final fields. This enables you to define methods that can access and modify the state of the object to which they belong.

Use interfaces if:

- You expect that unrelated classes would implement your interface. For example, the interfaces Comparable and Cloneable are implemented by many unrelated classes.
- You want to specify the behavior of a particular data type, but not concerned about who implements its behavior.
- You want to take advantage of multiple inheritance of type.

Helpful link: https://beginnersbook.com/2013/05/abstract-class-default-interface-in-java/

	Abstract Class	Interface
1	An abstract class can extend only one class or one abstract class at a time	An interface can extend any number of interfaces at a time
2	An abstract class can extend another concrete (regular) class or abstract class	An interface can only extend another interface
3	An abstract class can have both abstract and concrete methods	An interface can have only abstract methods
4	In abstract class keyword “abstract” is mandatory to declare a method as an abstract	In an interface keyword “abstract” is optional to declare a method as an abstract
5	An abstract class can have protected and public abstract methods	An interface can have only have public abstract methods
6	An abstract class can have static, final or static final variable with any access specifier	interface can only have public static final (constant) variable

Static / Final

Static - The keyword static indicates that the particular member belongs to a type itself, rather than to an instance of that type. This means that only one instance of that static member is created which is shared across all instances of the class. The static keyword is used mainly for memory management purposes.

Final - the "final" keyword is used to define an entity that can only be assigned once.
Variable - can only be assigned once
Classes - a "final" class cannot be subclassed
Methods - a "final" method cannot be overridden or hidden by subclasses

Reasons to use "final"

- Performance
- Obtain encapsulated data
- Reliability and contract

Static default Final

Static means it belongs to the class not an instance, this means that there is only one copy of that variable/method shared between all instances of a particular Class.
Final is entirely unrelated, it is a way of defining a once only initialization. You can either initialize when defining the variable or within the constructor, nowhere else.

Expected skills (4/5)

- **Able to explain difference between a class and an instance**

You can make multiple instances of your class.
Say you have 5 apples in your basket. Each of those apples is an object of type Apple, which has some characteristics (i.e. big, round, grows on trees). In programming terms, you can have a class called Apple, which has variables size:big, shape:round, habitat:grows on trees. To have 5 apples in your basket, you need to instantiate 5 apples. Apple apple1, Apple apple2, Apple apple3, etc.

- **Able to explain the role of constructors**

A constructor in Java is a block of code similar to a method that's called when an instance of an object is created. Here are the key differences between a constructor and a method:

- A constructor doesn't have a return type.
- The name of the constructor must be the same as the name of the class.
- Unlike methods, constructors are not considered members of a class.
- A constructor is called automatically when a new instance of an object is created.

Types of constructors:

- **Default -** If you do not implement any constructor in your class, Java compiler inserts a default constructor into your code on your behalf. This constructor is known as default constructor (it's a no-args constructor). If you implement any constructor then you no longer receive a default constructor from Java compiler.
- **No-args** - Constructor with no arguments is known as no-arg constructor. The signature is same as default constructor, however body can have any code unlike default constructor where the body of the constructor is empty.
- **Parameterized constructor** - Constructor with arguments(or you can say parameters)

A class that extends another class does not inherit its constructors. However, the subclass must call a constructor in the superclass inside one of the subclass constructors!

Look at the following two Java classes. The class Car extends (inherits from) the class Vehicle.

```
public class Vehicle {
    private String regNo = null;

    public Vehicle(String no) {
        this.regNo = no;
    }
}
public class Car extends Vehicle {
    private String brand = null;

    public Car(String br, String no) {
        super(no);
        this.brand = br;
    }
}
```

Notice the constructor in the Car class. It calls the constructor in the superclass using this Java statement:

```
super(no);
```

- **Able to explain where to use inheritance**

Inheritance should only be used when:

- Both classes are in the same logical domain
- The subclass is a proper subtype of the superclass
- The superclass's implementation is necessary or appropriate for the subclass
- The enhancements made by the subclass are primarily additive

- **Able to identify source of inherited methods**

TO BE COMPLETED

- **Able to demonstrate good example of encapsulation**

```
private myInt = 0;
public void setMyInt(int value) {
    if (value < 0)
        throw new IllegalArgumentException("myInt should be positive");
    myInt = value;
}
```

```
public class Door {
    private boolean locked;
    private boolean open;

    public boolean openDoor() {
        if (locked || open) {
            return false; //You can't open a locked or already open door
        }
        open = true;
        return true;
    }
}
```