

Study Guide For OOP in JAVA Part 2

Spencer Barnes

SID: 991728219

OOP in JAVA

Binsy Salin

Sheridan College

Terms:.....	3
Inheritance: .....	5
Overriding:.....	6
Inheritance Chains:.....	7
Abstract Classes &Methods:.....	7
Object class and toString():.....	7
The .equals() and .hashCode() Methods:.....	8
Casting:.....	8
Interfaces: .....	9
Coupling vs Cohesion .....	9
ArrayLists: .....	10
Generics: .....	10
Inner Classes.....	11
Anonymous Classes.....	11
Functional Programming: .....	12
Lambda:.....	12
Streams .....	12

## Terms:

Constructor: Constructors are special methods used to initialize objects of a class. They have the same name as the class and are invoked using the new keyword when an object is created. They can be declared with no input parameters or with required parameters.

Abstract Classes: a class that cannot be instantiated directly and may contain abstract methods, concrete methods, or both

Abstract Methods: a method declared without implementation, and which must be implemented in any child class.

toString(): a method defined in the Object class, which returns a string representation of an object. It's often overridden in subclasses to provide meaningful information about the object.

Object Class: The Object class is the root of the Java class hierarchy. All classes in Java are subclasses of Object by default.

Polymorphism: allows objects of different types to be treated as objects of a common superclass. It enables methods to be invoked on objects without knowing their specific type at compile time.

Visibility Modifiers: control the access levels of classes, variables, methods, and constructors in Java. The main modifiers are public, private, protected, and default (no modifier).

	Class	Package	Subclass (same pkg)	Subclass (diff pkg)	World
public	+	+	+	+	+
protected	+	+	+	+	
no modifier	+	+	+		
private	+				

Garbage Collection: is the process of automatically reclaiming memory occupied by objects that are no longer in use. Java's garbage collector manages memory automatically, allowing developers to focus on coding without worrying about memory management.

Functional Interfaces: are interfaces that contain only one abstract method. They can have multiple default or static methods but only one abstract method, making them suitable for use with lambda expressions.

Terminal Operations: Operations which must terminate a stream, and which only one can be used per stream. Include: `forEach()`, `collect()`, `count()`, `findFirst()`, `min()`, `max()`, `toArray()` and `anyMatch()`

Non-Terminal Operations: Stream Operations which can be stacked infinitely preceding a terminal operation. These include: `filter()`, `map()`, `distinct()`, `limit()`, `sorted()`, `flatMap()` and `peek()`

## Inheritance:

Inheritance is one of the four pillars of Object-Oriented Programming. With Inheritance we can create subclasses which inherit methods and properties from another class, which becomes their superclass.

Example:

```
public class Person {
    protected String name;

    public Person(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void speak(String sentence) {
        System.out.println(name + " says, " + sentence);
    }

    public void sleep() {
        System.out.println(name + " is sleeping");
    }
}

public class Pirate extends Person {
    private int numEyePatches;

    public Pirate(String name) {
        super(name);
        this.numEyePatches = 0;
    }

    public void pillage() {
        this.numEyePatches += 1;
        System.out.println(this.name + " has "
            + this.numEyePatches + " eye patches!");
    }

    public void eat() {
        System.out.println(this.name + " is eating crumpets.");
    }

    public void eat(String food) {
        System.out.println(this.name + " is eating " + food + ".");
    }
}
```

As you can see above, Person class is declared with a protected attribute name, and several methods.

Protected attributes are accessible within the same package and to subclasses, regardless of whether they are within the same package.

Then we look at Pirate which extends Person. Inside the Pirate constructor, you can see that we call `super(name)` which looks at which class we have extended and calls that constructor, which is why we needed to pass in a String name. Now here's where things get interesting:

```
1 package inheritance;
2
3 public class Start {
4     public static void main(String[] args)
5     {
6
7
8         Person person1 = new Person(name:"Bart");
9         person1.setName(name:"Bart");
10        person1.speak(sentence:"Have a nice day!");
11
12
13        Pirate pirate1 = new Pirate(name:"John");
14
15        pirate1.speak(sentence:"hi");
16        pirate1.pillage();
17        pirate1.eat();
18        pirate1.eat(food:"cucumber");
19    }
20 }
```

Run | Debug

PROBLEMS 12 DEBUG CONSOLE OUTPUT TERMINAL PORTS GITLENS

Bart says, Have a nice day!  
John says, hi  
John has 1 eye patches!  
John is eating crumpets.  
John is eating cucumber.  
PS C:\Users\srd\_\OneDrive\Documents\Sheridan\Year 1 project

As you can see, we create a person named Bart, and Bart can access all of the functions within person, as you would expect. However, when we create a pirate named John, we are still able to call the functions declared in the person class, even though John is a Pirate. This demonstrates inheritance perfectly. A subclass can access all its super classes' functions.

What do we do then, if we want to declare a function that does a similar but distinctly different thing in a subclass. An example of this is if we want pirates to speak, but in pirate speech.

## Overriding:

This is where overriding comes in. Inside the subclass, we are allowed to redeclare a method already declared inside the super class, and have it have different functionality by declaring it as an overridden class.

```
@Override
public void speak(String sentence)
{
    System.out.println(name + "says, ARRRR, " + sentence + " Matey!");
}
```

We do this by making sure that above the method declaration we provide an @Override, which lets the compiler know that we understand that we are overriding the superclass function for pirates, and now when we run the same program as before we get a totally different output.

```
Bart says, Have a nice day!
Johnsays, ARRRR, hi Matey!
John has 1 eye patches!
John is eating crumpets.
John is eating cucumber.
PS C:\Users\srd_\OneDrive\Do
```

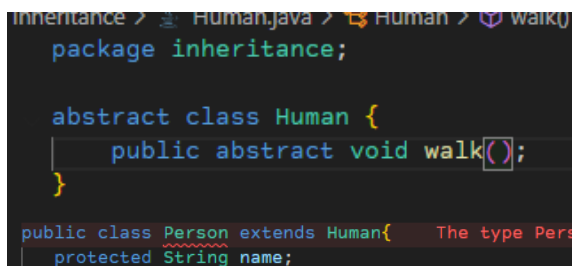
## Inheritance Chains:

In Java, a subclass can only inherit from one parent class, but a parent class can have many subclasses. By making a parent class have its own parent class, you can create a chain of inheritance, allowing for cohesive programming and establishing rules for similar classes.

## Abstract Classes & Methods:

What if, following that logic you want to make sure that any child of this class has to implement its own version of a method. You may choose to do this to allow certain other methods to properly function, allowing for multiple class types to be used interchangeably in the rest of your code. However, you don't want to have to create an entire parent class every time you want to enforce such a rule.

This is where Abstract Classes and Methods come in. An abstract class is a class that cannot be instantiated directly and may contain abstract methods, concrete methods, or both. An abstract method is declared without implementation and must be implemented in any child class.



```
package inheritance;

abstract class Human {
    public abstract void walk();
}

public class Person extends Human {
    protected String name;
```

Here we can see the declaration of an abstract class and method:

When the Person class extends Human, we can see that it MUST implement the walk() method. What we decide to declare is up to us, but any child class, including Pirate, must also include the walk() method.

## Object class and toString():

Following polymorphism, you may have realized that all classes are defined and therefore children of the Object class. This is important for many reasons, but the main one to remember is that any child class can access, or override the methods of its parent, and this includes the Object class! An example of this is the toString() function. By overriding the toString() method, we can decide what will happen when it is called, for example we could print out a person's name, all of their stored attributes, or even prepare our own custom message.

```
3 public class Start {
4     public static void main(String[] args)
5     {
6
7
8         Person person1 = new Person(name:"Bart");
9         person1.setName(name:"Bart");
10        person1.speak(sentence:"Have a nice day!");
11        System.out.println(person1.toString());
12    }
13 }
```

PROBLEMS 12 DEBUG CONSOLE OUTPUT TERMINAL PORTS GITLENS

```
PS C:\Users\srdb_\OneDrive\Documents\Sheridan\Year 1 projects\sem2\java>
PS C:\Users\srdb_\OneDrive\Documents\Sheridan\Year 1 projects\sem2\java>
PS C:\Users\srdb_\OneDrive\Documents\Sheridan\Year 1 projects\sem2\java>
b\AppData\Local\Programs\Eclipse Adoptium\jdk-17.0.9-hotspot\bin\java.exe
Bart says, Have a nice day!
Hey you can't just sum up all of a persons parts into one sentence!
```

Here we can see that with the default toString() we get the package.classname, followed by some random looking numbers, which are really just the memory address where the object is stored.

```
@Override
public String toString(){
    return "Hey you can't just sum up all of a persons parts into one sentence!";
}
```

However if we declare a method override on the toString() method, we can see that it now prints whatever information we want!

## The .equals() and .hashCode() Methods:

The .equals() method is used to compare the equality of objects, usually paired with a .hashCode() method, which takes the class attributes and uses a chosen algorithm to generate a number that is constant for that class and will always return that number given the same object.

## Casting:

Casting is the process of converting a reference of one data type to another. In Java, casting can be done implicitly for widening conversions or explicitly for narrowing conversions.

```
public class Dog extends Animal {
    public void bark() {
        System.out.println(x:"Dog is barking.");
    }
}

public class Animal {
    public void eat() {
        System.out.println(x:"Animal is eating.");
    }
}
```

Here we have two classes, one parent and one child. In main we declare an instance of Dog but immediately downcast it as an Animal, this works, as upcasting can be implicitly done. Any method that the Dog object would need to perform as an animal already exists.

However, you can see below that that we then explicitly case the animal to be a dog again, but putting (Dog) animal, and assigning it to a new Dog variable, dog. This is explicit casting, we are



```
20     Animal animal = new Dog();
21     animal.eat();
22     Dog dog = (Dog) animal;
23     dog.bark();
...
PROBLEMS 12 DEBUG CONSOLE OUTPUT TERMINAL
PS C:\Users\srd\OneDrive\Documents\Sherida\adoption\src\main\java\com\sherida\adoption\
package inheritance;

abstract interface Human {
    public abstract void walk();
}
```

telling the compiler that we know that this animal has the methods required to function as a dog, and this can only be done when that animal object is created using the dog constructor.

## Interfaces:

Similar to Abstract classes, Interfaces define a contract for classes to follow. They contain method signatures but no

method bodies. Classes implement interfaces to provide concrete implementations for those

methods. Unlike Abstract classes, there are no concrete methods in an interface.

```
public class Person implements Human { The type Person must implement the inherited abstract method Human.walk()
}
```

## Dependency

Dependency refers to the relationship between classes when one class relies on another to perform its functions. Managing dependencies is essential for creating maintainable and modular code.

### Coupling vs Cohesion

Coupling measures how much a class is dependent on other classes, while cohesion measures how well the elements within a class are related to each other. High cohesion and low coupling are desirable for better software design.

```

class Car {
    void start() {
        System.out.println(x:"Car started");
    }
}

class Person {
    void drive(Car car) { // Dependency: Person depends on Car
        car.start(); // Dependency: Person uses Car's start method
        System.out.println(x:"Person is driving");
    }
}

class Engine {
    void start() {
        System.out.println(x:"Engine started");
    }
}

class Car {
    Engine engine;

    Car() {
        this.engine = new Engine();
        // this is composition, the engine lives
        //and dies with the car
    }
}

class Department {
    String name;

    Department(String name) {
        this.name = name;
    }
}

class University {
    String name;
    ArrayList<Department> departments;

    University(String name) {
        this.name = name;
        this.departments = new ArrayList<Department>();
    }

    void addDepartment(Department dept) {
        departments.add(dept);
        //This is aggregation as the department exists
        //outside of the class and it simply holds values
    }
}

```

## ArrayLists:

```

ArrayList<String> list = new ArrayList<>();
list.add(e:"Apple");
list.add(e:"Banana");
list.add(e:"Orange");
for (String fruit : list) {
    System.out.println(fruit);
}
list.remove(o:"Banana");

```

ArrayList is a resizable array implementation provided by Java's java.util package. It dynamically grows and shrinks as elements are added or removed. In the example to the left, you can see how we can implement an array list.

Unlike regular java arrays, Array Lists come with several inbuilt methods to add, remove or even sort the arrays.

## Generics:

Generics allow classes and methods to operate on objects of various types while providing compile-time type safety. They enable the creation of parameterized types.

Here we have an example of how it can all come together, keeping in mind the other things we have learned. You can declare an interface, create classes that all implement that interface, and

```
interface Printable {  
    void print();  
}  
  
public class Printer<T extends Printable> {  
    private T item;  
  
    public Printer(T item) {  
        this.item = item;  
    }  
  
    public void printItem() {  
        item.print();  
    }  
}  
  
public class MyClass implements Printable {  
    @Override  
    public void print() {  
        System.out.println(x:"Printing from MyClass");  
    }  
}  
  
MyClass myClass = new MyClass();  
Printer<MyClass> printer = new Printer<>(myClass);  
printer.printItem();  
  
public class OuterClass {  
    private int outerField = 10;  
  
    // Inner class definition  
    public class InnerClass {  
        public void display() {  
            System.out.println("Value of outerField from InnerClass: " + outerField);  
        }  
    }  
}
```

then make a generic that also requires that the object passed into it also implements that interface, leading to a highly cohesive, but lowly coupled codebase.

## Inner Classes

Inner classes are classes defined within another class. They are used to logically group classes that are only used in one place and increase encapsulation and code readability.

## Anonymous Classes

Anonymous classes are inner classes without a name. They are typically used for one-time use and are defined and instantiated simultaneously. They require a functional interface to with one abstract method, so the compiler knows which method you are making.

To the left we are creating a new instance of a honkable implementing class, and since we are instantiating it as a new Honkable, the compiler doesn't need the class to be named. Then we

```
public interface Honkable {  
    void honk();  
}  
  
Honkable h3 = new Honkable() {  
    @Override  
    public void honk()  
    {  
        System.out.println(x:"beep beep");  
    }  
};
```

can call h3, and get the result we are looking for. However in general when you might use an anonymous class, you might as well use a lambda

## Functional Programming:

Before we talk about Lambdas, we need to talk about Functional programming. This is a programming paradigm focused on writing

software using pure functions and immutable data. This means we want to achieve a single objective with no ability for changes or modification elsewhere in our code. We don't want to store the data, but simply want the effect of a defined function once.

## Lambda:

Essentially a more concise version of Anonymous Classes, lambdas allow us to define a single method that doesn't need to be stored, and run that method, it as little as one line of code. Looking back to our example of an anonymous class, we can accomplish the same or similar results with much less code:

```
Honkable h4 = () -> System.out.println(x:"meep meep");  
h4.honk();
```

Because we are storing the method in Honkable h4, we don't need to use the new Honkable, and because we only want to execute one line of code, we don't even need our method parenthesis. This is the simplest form of a lambda, and you can pass in values and execute multi-line blocks of code, just like a normal method.

## Streams

Streams allow multiple cores of your CPU to be utilized by your programs at any given time, allowing for parallel tasks. We can use functional programming to great extent here. Its important to remember that streams collect the results without ever changing the original

structure of the data. This means we can input data and keep that data safe from modification by the stream and still do work.

Every Stream must be finished with a terminal operation and must only include one, and may be preceded by any number of non-terminal operations.

```
ArrayList<Student> students = new ArrayList<>();
students.add(new Student(name:"Ralph", Program.BUSI, gpa:14));
students.add(new Student(name:"Lisa", Program.COMP, gpa:98));
students.add(new Student(name:"Martin", Program.COMP, gpa:88));
students.add(new Student(name:"Millhouse", Program.COMP, gpa:38));
students.add(new Student(name:"Bart", Program.PHYS, gpa:52));
students.add(new Student(name:"Uter", Program.MATH, gpa:72));
students.add(new Student(name:"Uter", Program.MATH, gpa:72));

students.stream()
    .filter(student -> student.getGpa() > 50)
    .map(Student::getName)
    .distinct()
    .sorted()
    .forEach(System.out::println);
```

Bart
Lisa
Martin
Uter

In the example above, we create an arraylist of students. Below we use a stream and several functions afterwards to first filter the students by their gpa. Secondly, we map those students who had a high enough GPA to a new stream. Finally, we check to make sure that all the names are distinct, sort the list, and then print out each name.