ALL EXAMPLES MUST HAVE 2 CLASSES. DRIVER, AND IMPLEMENTATION

Java Inheritance.

Inheritance can be defined as the process where one object acquires the properties of another. With the use of inheritance, the information is made manageable in a hierarchical order.

When we talk about inheritance, the most commonly used keyword would be **extends** and **implements**.
These words would determine whether one object **IS-A type of another**. By using these keywords we can make one object acquire the properties of another object.

IS-- A Relationship:

IS-A is a way of saying : This object is a type of that object. Let us see how the extends keyword is used to achieve inheritance.

```
public class Animal{
}



public class Mammal extends Animal{
}

public class Reptile extends Animal{
}

public class Dog extends Mammal{
}
```

Now, based on the above example, In Object Oriented terms the following are true:

- Animal is the superclass of Mammal class.
- Animal is the superclass of Reptile class.
- Mammal and Reptile are subclasses of Animal class.
- Dog is the subclass of both Mammal and Animal classes.

Now, if we consider the IS-A relationship, we can say:
- Mammal IS-A Animal
- Reptile IS-A Animal
- Dog IS-A Mammal
- Hence : Dog IS-A Animal as well

With use of the extends keyword the subclasses will be able to inherit all the properties of the superclass except for the private properties of the superclass.

The **implements** keyword is used by classes by inherit from interfaces. **Interfaces can never be extended by the classes.**

```
public interface Animal{

}

public class Mammal implements Animal
{
}


public class Dog extends Mammal
{
}



public interface Animal {

   public void eat();
   public void travel();
}


public class MammalInt implements Animal{

   public void eat(){
      System.out.println("Mammal eats");
   }

   public void travel(){
      System.out.println("Mammal travels");
   }

   public int noOfLegs(){
      return 0;
   }


}
```

Using super and this keywords

```
//example of super keyword

class Vehicle{
  int speed=50;
}



class Bike4 extends Vehicle{
  int speed=100;

  void display(){
 System.out.println(super.speed);//will print speed of Vehicle -50
 System.out.println(speed);//will print speed of Vehicle now   -100 -Local Variables have preference
 System.out.println(this.speed);//will print speed of Vehicle now   -100
  }

  public static void main(String args[]){
   Bike4 b=new Bike4();
   b.display();

}
}
```

Java Overriding

I n the previous chapter, we talked about superclasses and subclasses. **If a class inherits a method from its superclass, then there is a chance to override the method provided that it is not marked final.**

The benefit of overriding is: **ability to define a behavior that's specific to the subclass type which means a subclass can implement a parent class method based on its requirement.**
In object-oriented terms, overriding means to override the functionality of an existing method.

Rules for method overriding:

• The argument list should be exactly the same as that of the overridden method.
• The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.
• The access level cannot be more restrictive than the overridden method's access level. For example, if the superclass method is declared public, then the overriding method in the subclass cannot be either private or protected.

• Instance methods can be overridden only if they are inherited by the subclass.
• A method declared final cannot be overridden.
• A method declared static cannot be overridden but can be re-declared.
• If a method cannot be inherited, then it cannot be overridden.
• A subclass within the same package as the instance's superclass can override any superclass method that is
not declared private or final.
• A subclass in a different package can only override the non-final methods declared public or protected.
• An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws
exceptions or not. However the overriding method should not throw checked exceptions that are new or broader
than the ones declared by the overridden method. The overriding method can throw narrower or fewer
exceptions than the overridden method.
• Constructors cannot be overridden.

Using the super keyword: When invoking a superclass version of an overridden method the super keyword is used.

Java Polymorphism

Polymorphism is the **ability of an object to take on many forms.** The **most common use of polymorphism in OOP, occurs when a parent class reference is used to refer to a child class object.**
Any Java **object that can pass more than one** IS-A test is considered to be polymorphic.

In Java, all Java objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object.

It is important to know that the only possible way to access an object is through a reference variable. A reference variable can be of only one type. Once declared, the type of a reference variable cannot be changed.

The reference variable can be reassigned to other objects provided that it is not declared final. The type of the reference variable would determine the methods that it can invoke on the object.

A reference variable can refer to any object of its declared type or any subtype of its declared type. A reference variable can be declared as a class or interface type.

```
public interfaceVegetarian{}
public class Animal{}
public class Deer extends Animal implements Vegetarian{}
```

Now, the Deer class is considered to be polymorphic since this has multiple inheritance. Following are true for the
above example:
• A Deer IS-A Animal
• A Deer IS-A Vegetarian
• A Deer IS-A Deer
• A Deer IS-A Object

Virtual Methods (NOT DONE)

```
public class Employee
{
.
.
.
public void mailCheck()
{
System.out.println("Mailing a check to "+this.name +" "+this.address);
}
.
.
.
}
```

```
public class Salary extends Employee
{
.
.
.
public void mailCheck() //OVERRIDE
{
System.out.println("Within mailCheck of Salary class ");
System.out.println("Mailing check to "+ getName() +" with salary "+ salary);
}
.
.
.
}
```

```
public static void main(String[] args)
{
Salary s =new Salary("Mohd Mohtashim","Ambehta, UP", 3,3600.00);
Employee e =new Salary("John Adams","Boston, MA", 2,2400.00);

System.out.println("Call mailCheck using Salary reference --"); s.mailCheck();
System.out.println("\n Call mailCheck usingEmployee reference--"); e.mailCheck();
}
```

Here, we instantiate two Salary objects, one using a Salary reference s, and the other using an Employee reference e.
While invoking s.mailCheck() the compiler sees mailCheck() in the Salary class at compile time, and the JVM invokes mailCheck() in the Salary class at run time.

Invoking mailCheck() on e is quite different because e is an Employee reference. When the

compiler sees e.mailCheck(), the compiler sees the mailCheck() method in the Employee class. Here, at compile time, t**he compiler used mailCheck() in Employee to validate this statement. At run time, however, the JVM invokes mailCheck() in the Salary class.**

This behavior is referred to as **virtual method invocation,** and the methods are referred to as virtual methods. **All methods in Java behave in this manner, whereby an overridden method is invoked at run time, no matter what data type the reference is that was used in the source code at compile time.**