**Java Inheritance**

*Inheritance*, is one of three object oriented concepts, which helps to separate out *common data and behavior* (or member variables and methods) from multiple related classes.

it is necessary to remove duplicate code across classes. *Inheritance* greatly helps in simplifying the code, enhancing its the re-usability and maintainability.

**Example**

A picture containing diagram

Description automatically generated

To achieve *inheritance*, we need to create a *super class* with the common code and behavior from multiple classes. Then we use the extends keyword to use that common code in the *sub classes*.

class CompareEntertainmentsUsingInheritance  
{  
    public static void main(String arg[])  
    {  
        Movie julai = new Movie();  
        julai.name = "Julai";  
        julai.director = "Trivikram";  
        julai.stuntMaster = "Peter Hein";  
        julai.numberOfArtists = 57;  
        julai.releaseDate = "15-Aug-2012";  
          
        julai.collectionsFirstWeek = 215467.8;  
        julai.collectionsRestOfTheDays = 541132.5;  
          
        Drama ramayan = new Drama();  
        ramayan.name = "Ramayana";  
        ramayan.writer = "Valmiki";  
        ramayan.stageSetter = "Anjaneya";  
        ramayan.numberOfArtists = 200000;  
        ramayan.releaseDate = "17-Mar-1659 BC";  
          
        ramayan.collectionsFirstWeek = 3282937242.86;  
        ramayan.collectionsRestOfTheDays = 93488272349.51;  
          
          
        Circus jumbo = new Circus();  
        jumbo.name = "Jumbo";  
        jumbo.ringMaster = "Antony";  
        jumbo.numberOfArtists = 316;  
        jumbo.releaseDate = "16-Dec-1997";  
          
        jumbo.collectionsFirstWeek = 2123132.21;  
        jumbo.collectionsRestOfTheDays = 234936725.09;  
          
        if((jumbo.getTotalCollections() > julai.getTotalCollections()) && (jumbo.getTotalCollections() > ramayan.getTotalCollections()))  
        {  
            jumbo.print();  
        }  
        else if (julai.getTotalCollections() > ramayan.getTotalCollections())  
        {  
            julai.print();  
        }  
        else  
        {  
            ramayan.print();  
        }  
      
    }  
}  
  
class Entertainment  
{  
    String name;  
    int numberOfArtists;  
    String releaseDate;  
    double collectionsFirstWeek;  
    double collectionsRestOfTheDays;  
  
    double getTotalCollections()  
    {  
        return collectionsFirstWeek + collectionsRestOfTheDays;  
    }  
  
    void printEntertainment()  
    {  
        System.out.println( name + " got the following collections " );  
        System.out.println("First Week : " + collectionsFirstWeek);  
        System.out.println("Rest Of The Days : " + collectionsRestOfTheDays);  
        System.out.println("Total Collections : " + getTotalCollections());  
        System.out.println("Total Artists : " + numberOfArtists);  
        System.out.println("Release Date : " + releaseDate);  
    }  
  
}  
  
class Movie extends Entertainment  
{  
    String director;  
    String stuntMaster;  
  
    void print()  
    {  
        printEntertainment();  
        System.out.println("Director : " + director);  
        System.out.println("Stunt Master : " + stuntMaster);  
    }  
}  
  
class Drama extends Entertainment  
{  
    String writer;  
    String stageSetter;  
  
  
    void print()  
    {  
        printEntertainment();  
        System.out.println("Writer : " + writer);  
        System.out.println("Stage Setter : " + stageSetter);  
    }  
}  
  
class Circus extends Entertainment  
{  
    String ringMaster;  
  
    void print()  
    {  
        printEntertainment();  
        System.out.println("Ring Master : " + ringMaster);  
    }  
}

**Inheritance supports is-a relationship**

Every sub-class object is also a super-class object, but every super-class object need not be a sub-class object. e.g., Every Movie is an Entertainment, but every Entertainment need not be a Movie, similarly, every Drama is an Entertainment, but every Entertainment need not be a Drama.

public void printName(Entertainment e)  
{  
    System.out.println("Name of the entertainment is " + e.name);  
}  
  
Movie businessMan = new Movie();  
businessMan.name = "Business Man";  
businessMan.directorName = "Puri Jagannadh";  
  
Drama devdas = new Drama();  
devdas.name = "Devdas";  
  
printName(businessMan); // Calling printName method by passing the Movie object  
printName(devdas); // Calling printName method by passing the Drama object

**This wont work:**

public void printMovie(Movie m)  
{  
    System.out.println(m.directorName + " is the director for movie " + m.name);  
}  
  
Movie businessMan = new Movie();  
businessMan.name = "Business Man";  
businessMan.directorName = "Puri Jagannadh";  
  
Drama devdas = new Drama();  
devdas.name = "Devdas";  
  
printMovie(businessMan); // Calling printMovie method by passing the Movie object  
printMovie(devdas); // THIS WON'T WORK SINCE DRAMA IS NOT A MOVIE

**Comapre different entertainments:**

public static void compareEntertainments(Entertainment ent1, Entertainment ent2, Entertainment ent3)  
    {  
    if((ent1.getTotalCollections() > ent2.getTotalCollections()) && (ent1.getTotalCollections() > ent3.getTotalCollections()))  
        {  
            System.out.println(ent1.name + " has the highest collections.");  
        }  
        else if (ent2.getTotalCollections() > ent3.getTotalCollections())  
        {  
            System.out.println(ent2.name + " has the highest collections.");  
        }  
        else  
        {  
            System.out.println(ent3.name + " has the highest collections.");  
        }  
    }  
  
}

**Downcasting**

since every sub-class object is also a super-class object, we can assign a sub-class object to a super-class reference. This is also called as *downcasting*. e.g., the object of type Drama can be assigned to the reference of the type Entertainment.

Entertainment e;  
  
Movie m = new Movie();  
e = m; // LINE A  
  
Drama d = new Drama();  
e = d; // LINE B  
  
m = d; // LINE C - WON'T WORK, SINCE A DRAMA OBJECT IS NOT OF TYPE MOVIE.

**Assigning super class reference to A sub class reference**

Entertainment ent1;  
Drama d1 = new Drama();  
ent1 = d1;  
  
Drama d2 = (Drama) ent1; // LINE A

As shown here, we are first creating a reference of Entertainment called ent1, a reference of Drama called d1 and a Drama object. Since we know that ent1 is referring to an object of type Drama, we can up cast the reference as shown in LINE A and assign it to another Drama reference d2.  
Instead, if we try to up cast a reference which does not refer to the correct object type, then it throws a ClassCastException during run time.

Entertainment ent1;  
Circus c1 = new Circus();  
ent1 = c1;  
  
Drama d2 = (Drama) ent1; // LINE A - THROWS ClassCastException  
Circus c2 = (Circus) ent1; // LINE B - WILL WORK

Here a ClassCastException is thrown at LINE A, since ent1 points to Circus object and hence can not be cast to a Drama. But the casting as done in LINE B will work since ent1 points to the Circus object.

**Multiple Inheritance in Java**

class TestVehiclesHierarchy  
{  
    public static void main(String arg[])  
    {  
        MountainBike mb = new MountainBike();  
        mb.numberOfWheels = 2;  
        mb.registrationNumber = "APXX WWW";  
        mb.hasHelmet = true;  
        mb.maxElevation = 3000.0;  
          
        System.out.print("Mountain Bike with registration Number " + mb.registrationNumber);  
        System.out.println(" is supported till the elevation of " + mb.maxElevation + " feet.");      
    }  
}  
  
class Vehicle  
{  
    int numberOfWheels;  
}  
  
class RegisteredVehicle extends Vehicle  
{  
    String registrationNumber;  
}  
  
class Bike extends RegisteredVehicle // LINE A  
{  
    boolean hasHelmet;  
}  
  
class MountainBike extends Bike  
{  
    double maxElevation;  
}  
  
class Car extends RegisteredVehicle  
{  
    boolean hasAC;  
}  
  
class Cycle extends Vehicle  
{  
    boolean hasBackSeat;  
}

**Java - Modifier Types**

**Access Modifiers**

**Default, private, public , protected**

1. Visible to the package, the **default**. No modifiers are needed.

Eg:

String version = "1.5.1";

boolean processOrder() {

return true;

}

2. Visible to the class only (**private**).

public class Logger {

private String format;

public String getFormat() {

return this.format;

}

public void setFormat(String format) {

this.format = format;

}

}

3. Visible to the world (**public**).

public static void main(String[] arguments) {

// ...

}

4. Visible to the package and all subclasses (**protected**). - We will cover this later

class AudioPlayer {

protected boolean openSpeaker(Speaker sp) {

// implementation details

}

}

class StreamingAudioPlayer extends AudioPlayer {

boolean openSpeaker(Speaker sp) {

// implementation details

}

}

**Java - Non Access Modifiers**

1. The **Static** Modifier

Static Variables

The static keyword is used to create variables that will exist independently of any instances created for the class. Only one copy of the static variable exists regardless of the number of instances of the class.

Static variables are also known as class variables. Local variables cannot be declared static.

Static Methods

The static keyword is used to create methods that will exist independently of any instances created for the class.

Static methods do not use any instance variables of any object of the class they are defined in. Static methods take all the data from parameters and compute something from those parameters, with no reference to variables.

Class variables and methods can be accessed using the class name followed by a dot and the name of the variable or method.

Eg:

public class InstanceCounter {

private static int numInstances = 0;

protected static int getCount() {

return numInstances;

}

private static void addInstance() {

numInstances++;

}

InstanceCounter() {

InstanceCounter.addInstance();

}

public static void main(String[] arguments) {

System.out.println("Starting with " + InstanceCounter.getCount() + " instances");

for (int i = 0; i < 500; ++i) {

new InstanceCounter();

}

System.out.println("Created " + InstanceCounter.getCount() + " instances");

}

}

2. The **Final** Modifier

Final Variables

A final variable can be explicitly initialized only once. A reference variable declared final can never be reassigned to refer to an different object.

However, the data within the object can be changed. So, the state of the object can be changed but not the reference.

With variables, the final modifier often is used with static to make the constant a class variable.

Eg:

public class Test {

final int value = 10;

// The following are examples of declaring constants:

public static final int BOXWIDTH = 6;

static final String TITLE = "Manager";

public void changeValue() {

value = 12; // will give an error

}

}

**Final Methods**

A final method cannot be overridden by any subclasses. As mentioned previously, the final modifier prevents a method from being modified in a subclass.

The main intention of making a method final would be that the content of the method should not be changed by any outsider.

Example

You declare methods using the final modifier in the class declaration, as in the following example −

public class Test {

public final void changeName() {

// body of method

}

}

**Final Classes**

The main purpose of using a class being declared as final is to prevent the class from being subclassed. If a class is marked as final then no class can inherit any feature from the final class.

Example

public final class Test {

// body of class

}

The **abstract** Modifier

Abstract Class

An abstract class can never be instantiated. If a class is declared as abstract then the sole purpose is for the class to be extended.

A class cannot be both abstract and final (since a final class cannot be extended). If a class contains abstract methods then the class should be declared abstract. Otherwise, a compile error will be thrown.

An abstract class may contain both abstract methods as well normal methods.

Example

abstract class Caravan {

private double price;

private String model;

private String year;

public abstract void goFast(); // an abstract method

public abstract void changeColor();

}

**Abstract Methods**

An abstract method is a method declared without any implementation. The methods body (implementation) is provided by the subclass. Abstract methods can never be final or strict.

Any class that extends an abstract class must implement all the abstract methods of the super class, unless the subclass is also an abstract class.

If a class contains one or more abstract methods, then the class must be declared abstract. An abstract class does not need to contain abstract methods.

The abstract method ends with a semicolon. Example: public abstract sample();

Example

public abstract class SuperClass {

abstract void m(); // abstract method

}

class SubClass extends SuperClass {

// implements the abstract method

void m() {

.........

}

}

The **Synchronized** Modifier

The synchronized keyword used to indicate that a method can be accessed by only one thread at a time. The synchronized modifier can be applied with any of the four access level modifiers.

Example

public synchronized void showDetails() {

.......

}

The Transient Modifier - We can cover later

An instance variable is marked transient to indicate the JVM to skip the particular variable when serializing the object containing it.

This modifier is included in the statement that creates the variable, preceding the class or data type of the variable.

Example

public transient int limit = 55; // will not persist

public int b; // will persist

The Volatile Modifier - We can cover later

The volatile modifier is used to let the JVM know that a thread accessing the variable must always merge its own private copy of the variable with the master copy in the memory.

Accessing a volatile variable synchronizes all the cached copied of the variables in the main memory. Volatile can only be applied to instance variables, which are of type object or private. A volatile object reference can be null.

Example

public class MyRunnable implements Runnable {

private volatile boolean active;

public void run() {

active = true;

while (active) { // line 1

// some code here

}

}

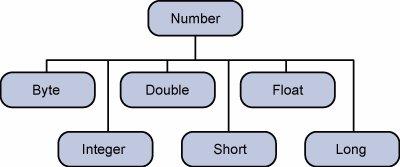
public void stop() {

active = false; // line 2

}

}

All the wrapper classes (Integer, Long, Byte, Double, Float, Short) are subclasses of the abstract class Number.



The object of the wrapper class contains or wraps its respective primitive data type. Converting primitive data types into object is called **boxing**, and this is taken care by the compiler. Therefore, while using a wrapper class you just need to pass the value of the primitive data type to the constructor of the Wrapper class.

And the Wrapper object will be converted back to a primitive data type, and this process is called unboxing. The **Number** class is part of the java.lang package.

public class Test {

public static void main(String args[]) {

Integer x = 5; // boxes int to an Integer object

x = x + 10; // unboxes the Integer to a int

System.out.println(x);

}

}

Strings, which are widely used in Java programming, are a sequence of characters. In Java programming language, strings are treated as objects.

The Java platform provides the String class to create and manipulate strings.