Java Programming
MODULE-1
(Inheritance)

By
Dr. N. Ramana
Associate Professor
UCE(KSM), Kakatiya University

Course Outcomes

- Able to solve real world problems using OOP techniques.
- Able to understand the use of abstract classes.
- Able to solve problems using java collection framework and I/o classes.
- Able to develop multithreaded applications with synchronization.
- Able to develop applets for web applications.
- Able to design GUI based applications

Agenda

- ✓ Inheritance basics
- ✓ Member access
- ✓ Constructors
- ✓ Creating Multilevel hierarchy
- ✓ super uses, using final with inheritance
- ✓ Polymorphism-ad hoc polymorphism pure polymorphism
- ✓ abstract classes, Object class
- ✓ forms of inheritance
- ✓ benefits of inheritance & costs of inheritance.

Java Inheritance Basics

The inheritance can be defined as follows:

- The inheritance is the process of acquiring the properties of one class to another class.

Inheritance Basics

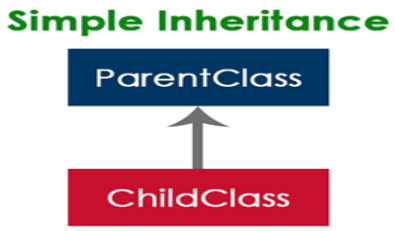
- In inheritance, we use the terms like parent class, child class, base class, derived class, superclass, and subclass.
- The Parent class is the class which provides features to another class. The parent class is also known as Base class or Superclass.
- The Child class is the class which receives features from another class. The child class is also known as the Derived Class or Subclass.

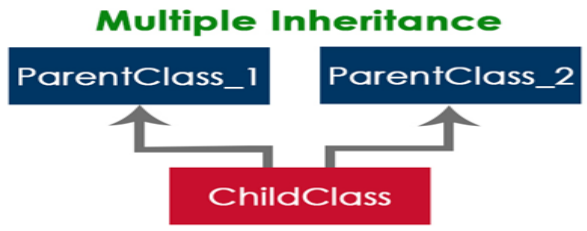
Types of Inheritance

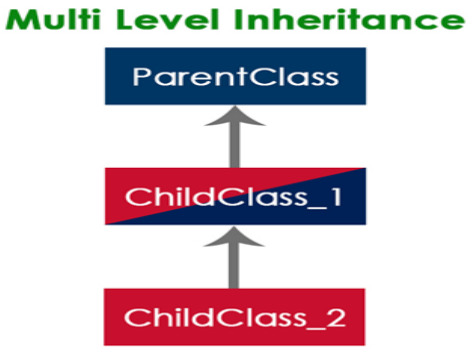
There are five types of inheritances, and they are as follows.

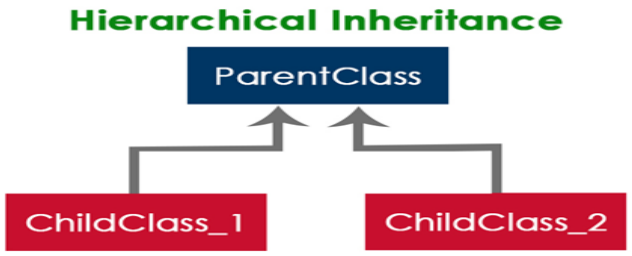
- Simple Inheritance (or) Single Inheritance
- Multiple Inheritance
- Multi-Level Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

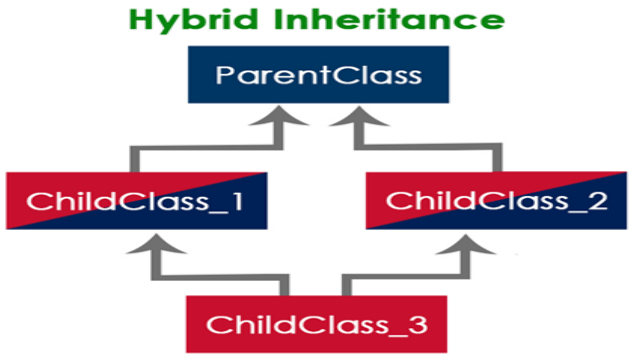
Types of Inheritance











Creating Child Class in java

In java, we use the keyword **extends** to create a child class. The following syntax used to create a child class in java.

Syntax

```
class <ChildClassName> extends <ParentClassName>
   {
    ...
                //Implementation of child class
   ...
   }
```

Example for Single Inheritance in java- Example

```
class ParentClass
  {
        int a;
         void setData(int a)
         {
             this.a = a;
          }
  }
class ChildClass extends ParentClass
  {
      void showData()
          {
          System.out.println("Value of a is " + a);
           }
  }
```

```
public class SingleInheritance
  {
        public static void main(String[] args)
            {
            ChildClass obj = new ChildClass();
            obj.setData(100);
            obj.showData();
            }
  }
```

Multi-level Inheritance in java- Example

```
class ParentClass
  {
        int a;
        void setData(int a)
        {
        this.a = a;
        }
  }
class ChildClass extends ParentClass
  {
        void showData()
        {
        System.out.println("Value of a is " + a);
        }
  }
class ChildChildClass extends ChildClass
  {
        void display()
        {
        System.out.println("Inside ChildChildClass!");
         }  }
```

```
public class MultipleInheritance
  {
  public static void main(String[] args)
        {
        ChildChildClass obj = new ChildChildClass();
        obj.setData(100);
        obj.showData();
        obj.display();
        }
  }
```

Hierarchical Inheritance in java- Example

```
class ParentClass
  {
        int a;
        void setData(int a)
        {
        this.a = a;
        }
   }
 class ChildClass extends ParentClass
  {
     void showData()
        {
        System.out.println("Inside ChildClass!");
        System.out.println("Value of a is " + a);
        }
  }
class ChildClassToo extends ParentClass
   {
      void display()
        {
        System.out.println("Inside ChildClassToo!");
         System.out.println("Value of a is " + a);
         }
}
public class HierarchicalInheritance
   {
   public static void main(String[] args)
   {
      ChildClass child_obj = new ChildClass();
      child_obj.setData(100);
      child_obj.showData();
      ChildClassToo childToo_obj = new ChildClassToo();
      childToo_obj.setData(200);
       childToo_obj.display();
   }
   }
```

Java Access Modifiers

In Java, the access specifiers (also known as access modifiers) used to restrict the scope or accessibility of a class, constructor, variable, method or data member of class and interface. There are four access specifiers, and their list is below.

- default (or) no modifier
- public
- protected
- private

Java Access Modifiers

- ✓ The public members can be accessed everywhere.
- ✓ The private members can be accessed only inside the same class.
- ✓ The protected members are accessible to every child class (same package or other packages).
- ✓ The default members are accessible within the same package but not outside the package.

Access control for members of class and interface in java

Accessibility / Access Specifier	Same Class	Same Package		Other Package	
		Child class	Non-child class	Child class	Non-child class
Public	Yes	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	Yes	No
Default	Yes	Yes	Yes	No	No
Private	Yes	No	No	No	No

Example

```
class ParentClass
  {
  int a = 10;
  public int b = 20;
  protected int c = 30;
  private int d = 40;
  void showData()
  {
  System.out.println("Inside ParentClass");
  System.out.println("a = " + a);
  System.out.println("b = " + b);
  System.out.println("c = " + c);
  System.out.println("d = " + d);
  } }
```

```
class ChildClass extends ParentClass
{
void accessData()
{
System.out.println("Inside ChildClass");
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
//System.out.println("d = " + d); // private member can't
be accessed
}
  }
public class AccessModifiersExample
{
public static void main(String[] args)
{
ChildClass obj = new ChildClass();
obj.showData();
obj.accessData();
}
```

Java Constructors in Inheritance

- ✓ It is very important to understand how the constructors get executed in the inheritance concept. In the inheritance, the constructors never get inherited to any child class.
- ✓ In java, the default constructor of a parent class called automatically by the constructor of its child class. That means when we create an object of the child class, the parent class constructor executed, followed by the child class constructor executed.

Example

```
class  ParentClass
   {
     int  a;
   ParentClass()
   {
   System.out.println("Inside ParentClass constructor!");
     }
     }
```

```
class  ChildClass  extends  ParentClass
    {
      ChildClass()
       {
           System.out.println("Inside ChildClass constructor!!");
       }     }
class  ChildChildClass  extends  ChildClass
    {
    ChildChildClass()
     {
          System.out.println("Inside ChildChildClass constructor!!");
     } }
 public  class  ConstructorInInheritance
    {
    public  static  void  main(String[]  args)
       {
       ChildChildClass  obj  =  new  ChildChildClass();
       }
    }
```

Java super keyword

In java, super is a keyword used to refers to the parent class object. The super keyword came into existence to solve the naming conflicts in the inheritance. When both parent class and child class have members with the same name, then the super keyword is used to refer to the parent class version.

In java, the super keyword is used for the following purposes.

1. To refer parent class data members
2. To refer parent class methods
3. To call parent class constructor

super : to refer parent class data members

When both parent class and child class have data members with the same name, then the super keyword is used to refer to the parent class data member from child class.

```
class ParentClass
  {
  int num = 10;
   }
class ChildClass extends ParentClass
  {
  int num = 20;
  void showData()
   {
  System.out.println("Inside the ChildClass");
  System.out.println("ChildClass num = " + num);
  System.out.println("ParentClass num = " + super.num);

   }
  }
```

```
public class SuperKeywordExample
 {
public static void main(String[] args)
  {
  ChildClass obj = new ChildClass();
  obj.showData();
  System.out.println("\nInside the non-child class");
  System.out.println("ChildClass num = " + obj.num);
//System.out.println("ParentClass num = " +
super.num); //super can't be used here
  }
 }
```

super: to refer parent class method

When both parent class and child class have method with the same name, then the super keyword is used to refer to the parent class method from child class.

```
class ParentClass
 {
 int num1 = 10;
 void showData()
 {
 System.out.println("\nInside the ParentClass
  showData method");
  System.out.println("ChildClass num = " +
  num1);
  }
  }
class ChildClass extends ParentClass
    {
    int num2 = 20;
 void showData()
 {
 System.out.println("\nInside the ChildClass showData
method"); System.out.println("ChildClass num = " +
num2);
 super.showData();
}
 }
public class SuperKeywordExample
 {
 public static void main(String[] args)
 {
ChildClass obj = new ChildClass();
 obj.showData();
 //super.showData(); // super can't be used here
}
}
```

super: to call parent class constructor

When an object of child class is created, it automatically calls the parent class default-constructor before it's own. But, the parameterized constructor of parent class must be called explicitly using the super keyword inside the child class constructor.

```
class ParentClass
 {
 int num1;
  ParentClass()
 {
 System.out.println("\nInside the ParentClass
 default constructor");
  num1 = 10;
 }
 ParentClass(int value)
 {
 System.out.println("\nInside the ParentClass
 parameterized constructor"); num1 = value;
}
}
```

```
class ChildClass extends ParentClass
{
int num2;
 ChildClass()
{
super(100);
 System.out.println("\nInside the ChildClass
constructor");
 num2 = 200;
}
}
public class SuperKeywordExample { public static
void main(String[] args)
 {
 ChildClass obj = new ChildClass(); }
```

Java final keyword

In java, the final is a keyword and it is used with the following things.

- ✓ With variable (to create constant)
- ✓ With method (to avoid method overriding)
- ✓ With class (to avoid inheritance)
- ✓ final with variables

When a variable defined with the final keyword, it becomes a constant, and it does not allow us to modify the value. The variable defined with the final keyword allows only a one-time assignment, once a value assigned to it, never allows us to change it again.

final with variables

```
public class FinalVariableExample
{
 public static void main(String[] args)
{
final int a = 10;
System.out.println("a = " + a);
a = 100; // Can't be modified
}
}
```

final with methods

When a method defined with the final keyword, it does not allow it to override. The final method extends to the child class, but the child class can not override or re-define it. It must be used as it has implemented in the parent class.

```
class ParentClass
{
int num = 10;
 final void showData()
 {
 System.out.println("Inside ParentClass showData() method");
System.out.println("num = " + num);
}
}
 class ChildClass extends ParentClass
{
void showData()
{
System.out.println("Inside ChildClass showData() method");
System.out.println("num = " + num);
}
}
public class FinalKeywordExample
 {
 public static void main(String[] args)
 {
 ChildClass obj = new ChildClass();
 obj.showData();
}
}
```

final with class

When a class defined with final keyword, it can not be extended by any other class.

```
final class ParentClass
{
  int num = 10;
void showData()
  {
System.out.println("Inside ParentClass showData() method"); System.out.println("num = " + num);
}
}
class ChildClass extends ParentClass
{


}
public class FinalKeywordExample
{
  public static void main(String[] args)
  {
  ChildClass obj = new ChildClass();
  }
  }
```

Java Polymorphism

➢The polymorphism is the process of defining same method with different implementation. That means creating multiple methods with different behaviors.

➢In java, polymorphism implemented using method overloading and method overriding.

➢Polymorphism can be

- Adhoc polymorphism
- Pure polymorphism

Ad hoc polymorphism

- The ad hoc polymorphism is a technique used to define the same method with different implementations and different arguments.
- In a java programming language, ad hoc polymorphism carried out with a method overloading concept. In ad hoc polymorphism the method binding happens at the time of compilation.
- Ad hoc polymorphism is also known as compile-time polymorphism. Every function call binded with the respective overloaded method based on the arguments.
- The ad hoc polymorphism implemented within the class only.

```
import java.util.Arrays;
public class AdHocPolymorphismExample
{
void sorting(int[] list)
{
Arrays.parallelSort(list);
System.out.println("Integers after sort: " + Arrays.toString(list) );
}
void sorting(String[] names)
{
  Arrays.parallelSort(names);
System.out.println("Names after sort: " + Arrays.toString(names) );
  }
public static void main(String[] args)
{
AdHocPolymorphismExample obj = new AdHocPolymorphismExample();
int list[] = {2, 3, 1, 5, 4};
obj.sorting(list); // Calling with integer array
String[] names = {"rama", "raja", "shyam", "seeta"};
obj.sorting(names); // Calling with String array
  }
  }
```

Pure polymorphism

- The pure polymorphism is a technique used to define the same method with the same arguments but different implementations. In a java programming language, pure polymorphism carried out with a method overriding concept.
- In pure polymorphism, the method binding happens at run time. Pure polymorphism is also known as run-time polymorphism. Every function call binding with the respective overridden method based on the object reference.
- When a child class has a definition for a member function of the parent class, the parent class function is said to be overridden.
- The pure polymorphism implemented in the inheritance concept only.

```
class ParentClass
{
  int num = 10;
  void showData()
  {
System.out.println("Inside ParentClass showData() method");
System.out.println("num = " + num);
  }
}
class ChildClass extends ParentClass
{
void showData()
{
System.out.println("Inside ChildClass showData() method");
System.out.println("num = " + num);
}
}
public class PurePolymorphism
{
public static void main(String[] args)
  {
ParentClass obj = new ParentClass();
obj.showData();
obj = new ChildClass();
obj.showData();
}
}
```

Rules for method overriding

- Static methods can not be overridden.
- Final methods can not be overridden.
- Private methods can not be overridden.
- Constructor can not be overridden.
- An abstract method must be overridden.
- Use super keyword to invoke overridden method from child class.
- The return type of the overriding method must be same as the parent has it.
- The access specifier of the overriding method can be changed, but the visibility must increase but not decrease.

Java Abstract Class

- An abstract class is a class that created using abstract keyword. In other words, a class prefixed with abstract keyword is known as an abstract class.
- In java, an abstract class may contain abstract methods (methods without implementation) and also non-abstract methods (methods with implementation).
- We use the following syntax to create an abstract class.

Syntax

```
abstract class <ClassName>
{
  ...
 }
```

Java Object Class

- In java, the Object class is the super most class of any class hierarchy. The Object class in the java programming language is present inside the java.lang package.
- Every class in the java programming language is a subclass of Object class by default.
- The Object class is useful when you want to refer to any object whose type you don't know. Because it is the superclass of all other classes in java, it can refer to any type of object.

Methods of Object class

- The following table depicts all built-in methods of Object class in java.

Method	Description	Return Value
getClass()	Returns Class class object	object
hashCode()	returns the hashcode number for object being used.	int

equals(Object obj)	compares the argument object to calling object.	boolean
clone()	Compares two strings, ignoring case	int
concat(String)	Creates copy of invoking object	object
toString()	eturns the string representation of invoking object.	String
notify()	wakes up a thread, waiting on invoking object's monitor.	void
notifyAll()	wakes up all the threads, waiting on invoking object's monitor.	void
wait()	causes the current thread to wait, until another thread notifies.	void
wait(long,int)	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies.	void
finalize()	It is invoked by the garbage collector before an object is being garbage collected.	void

Java Forms of Inheritance

The following are the differnt forms of inheritance in java.

1. Specialization
2. Specification
3. Construction
4. Extension
5. Limitation
6. Combination

Java Forms of Inheritance

1. Specialization

   It is the most ideal form of inheritance. The subclass is a special case of the parent class. It holds the principle of substitutability.

2. Specification

   This is another commonly used form of inheritance. In this form of inheritance, the parent class just specifies which methods should be available to the child class but doesn't implement them.

3. Construction

   This is another form of inheritance where the child class may change the behavior defined by the parent class (overriding). It does not hold the principle of substitutability.

4. Extension

   This is another form of inheritance where the child class may add its new properties. It holds the principle of substitutability.

5. Limitation

This is another form of inheritance where the subclass restricts the inherited behavior. It does not hold the principle of substitutability.

6. Combination

This is another form of inheritance where the subclass inherits properties from multiple parent classes. Java does not support multiple inheritance type.

Benefits of Inheritance

- Inheritance helps in code reuse. The child class may use the code defined in the parent class without re-writing it.
- Inheritance can save time and effort as the main code need not be written again.
- Inheritance provides a clear model structure which is easy to understand.
- An inheritance leads to less development and maintenance costs.
- With inheritance, we will be able to override the methods of the base class so that the meaningful implementation of the base class method can be designed in the derived class. An inheritance leads to less development and maintenance costs.
- In inheritance base class can decide to keep some data private so that it cannot be altered by the derived class.

Costs of Inheritance

- Inheritance decreases the execution speed due to the increased time and effort it takes, the program to jump through all the levels of overloaded classes.
- Inheritance makes the two classes (base and inherited class) get tightly coupled. This means one cannot be used independently of each other.
- The changes made in the parent class will affect the behavior of child class too.
- The overuse of inheritance makes the program more complex.

Questions

1. What is inheritance in object-oriented programming?
2. How does inheritance promote code reusability?
3. What is the difference between single inheritance and multiple inheritance?
4. What is a superclass and a subclass?
5. How are members (fields and methods) accessed in a class hierarchy?
6. What is the difference between public, protected, and private access modifiers?
7. How does the protected access modifier differ from private in terms of inheritance?
8. How does a subclass call the constructor of its superclass?

Questions

9. How is the super keyword used in Java?

10. What does it mean to declare a class as final?

11. What is an abstract class?

12. How does an abstract class differ from a concrete class?