Unit 2

Object Oriented Principles in Java 6hrs

2.1 Review of Object-Oriented Principles

Object-Oriented Programming or OOPs refers to languages that use objects in programming. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc. in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

OOPs Concepts:

- Class
- Objects
- Data Abstraction
- Encapsulation
- Inheritance
- Polymorphism
- Dynamic Binding
- Message Passing

Class:

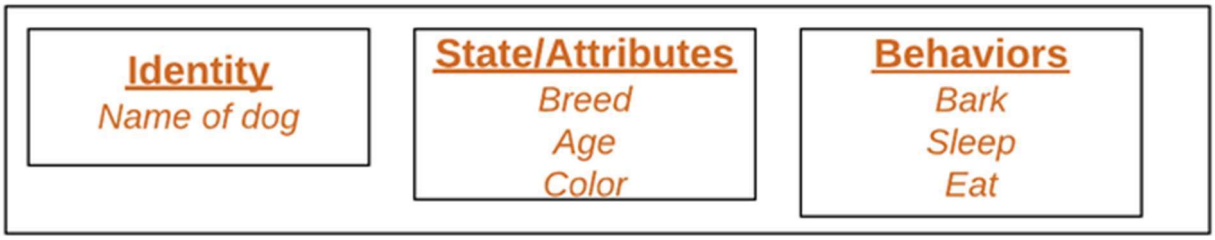
A class is a user-defined data type. It consists of data members and member functions, which can be accessed and used by creating an instance of that class. It represents the set of properties or methods that are common to all objects of one type. A class is like a blueprint for an object.

For Example: Consider the Class of Cars. There may be many cars with different names and brands but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range, etc. So here, Car is the class, and wheels, speed limits, mileage are their properties.

Object:

It is a basic unit of Object-Oriented Programming and represents the real-life entities. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated. An object has an identity, state, and behavior. Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other's data or code, it is sufficient to know the type of message accepted and type of response returned by the objects.

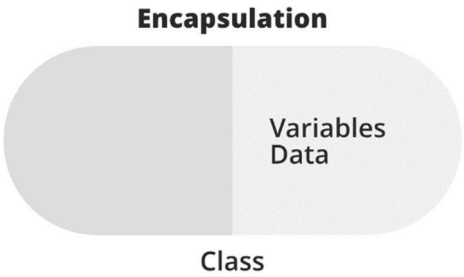
For example, "Dog" is a real-life Object, which has some characteristics like color, Breed, Bark, Sleep, and Eats.



Object

Data Abstraction:
Data abstraction is one of the most essential and important features of object-oriented programming. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation. Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of the car or applying brakes will stop the car, but he does not know about how on pressing the accelerator the speed is increasing, he does not know about the inner mechanism of the car or the implementation of the accelerator, brakes, etc. in the car. This is what abstraction is.

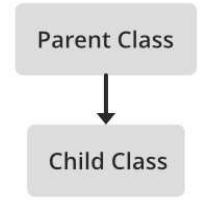
Encapsulation:
Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. In Encapsulation, the variables or data of a class are hidden from any other class and can be accessed only through any member function of their class in which they are declared. As in encapsulation, the data in a class is hidden from other classes, so it is also known as data-hiding.



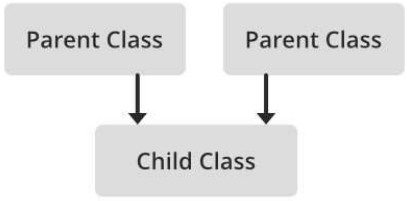
Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section, etc. The finance section handles all the financial transactions and keeps records of all the data related to finance. Similarly, the sales section handles all the sales-related activities and keeps records of all the sales. Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name “sales section”.

Inheritance:
Inheritance is an important pillar of OOP (Object-Oriented Programming). The capability of a class to derive properties and characteristics from another class is called Inheritance. When we write a class, we inherit properties from other classes. So, when we create a class, we do not need to write

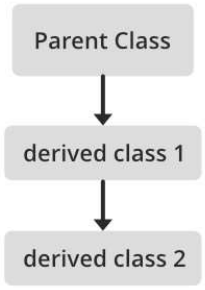
all the properties and functions again and again, as these can be inherited from another class that possesses it. Inheritance allows the user to reuse the code whenever possible and reduce its redundancy.



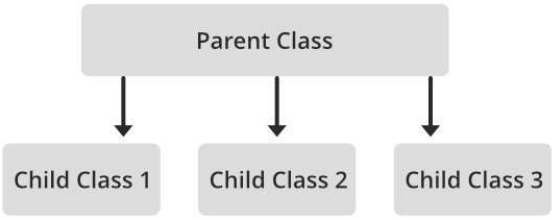
Single inheritance



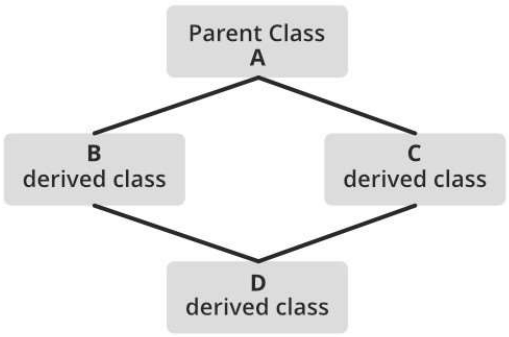
Multiple inheritance



Multilevel inheritance



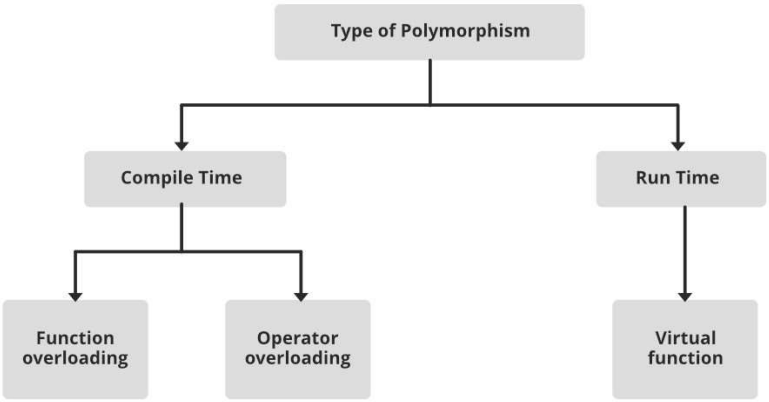
Hierarchical inheritance



Hybrid Class

Polymorphism:

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. For example, A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So, the same person possesses different behavior in different situations. This is called polymorphism.



Dynamic Binding:
In dynamic binding, the code to be executed in response to the function call is decided at runtime. Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run time. Dynamic Method Binding One of the main advantages of inheritance is that some derived class D has all the members of its base class B. Once D is not hiding any of the public members of B, then an object of D can represent B in any context where a B could be used. This feature is known as subtype polymorphism.

Message Passing:
It is a form of communication used in object-oriented programming as well as parallel programming. Objects communicate with one another by sending and receiving information to each other. A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing involves specifying the name of the object, the name of the function, and the information to be sent.

Why do we need object-oriented programming?
- To make the development and maintenance of projects more effortless.
- To provide the feature of data hiding that is good for security concerns.
- We can solve real-world problems if we are using object-oriented programming.
- It ensures code reusability.
- It lets us write generic code: which will work with a range of data, so we don't have to write basic stuff over and over again.

2.2 super class, sub-class, inheritance and member access

Inheritance is the process by which a class inherits the property of another class. The class which is being inherited is known as the super class (or base class) and the class that inherits the property is known sub class (or derived class). To inherit the content of superclass, the base class uses the **extends** keyword. Java provides multilevel inheritance but does not support multiple inheritance.

Example:

```
class A
{
	int i, j;
	void showij()
	{
		System.out.println("i and j: " + i + " " + j);
	}
}
class B extends A
{
	int k;
	void showk()
	{
```

```
            System.out.println("k: " + k);
      }
      void sum()
      {
            System.out.println("i+j+k: " + (i+j+k));
      }
}
class SimpleInheritance
{
      public static void main(String args[])
      {
            B subOb = new B(); subOb.i = 7;
            subOb.j = 8;
            subOb.k = 9;
            System.out.println("Contents of subOb: "); subOb.showij();
            subOb.showk();
            System.out.println("Sum of i, j and k in subOb:"); subOb.sum();
      }
}
```

The output from this program is shown here: Contents of subOb:

i and j: 7 8

k: 9

Sum of i, j and k in subOb: i+j+k: 24

2.3 Types of inheritance

Inheritance

The process of deriving a new class from an old one is called inheritance or derivation. The old class is referred to as the base class or super class or parent class and new one is called the derived class or subclass or child class. The derived class inherits some or all of the properties from the base class. A class can inherit properties from more than one class or from more than one level. The main purpose of derivation or inheritance is reusability. Java strongly supports the concept of reusability. The Java classes can be reused in several ways. Once a class has been written and tested, it can be used by another by another class by inheriting the properties of parent class. Defining a subclass A subclass can be defined as follows:

```
class SubClassName extends SuperClassName{
      // instance variables
```

```
      //Methods
}
```

The keyword extends signifies that the properties of the SuperClassName are extended to the SubClassName. The subclass will now contain its own variables and methods as well those of the super class. This kind of situation occurs when we want to add some more properties to an existing class without actually modifying it.

Types of Inheritance The following kinds of inheritance are there in Java.

• Simple Inheritance

• Multilevel Inheritance

• Hierarchical Inheritance

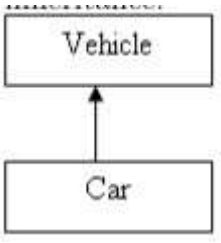
• Multiple Inheritance

• Hybrid Inheritance

In java programming, Multiple and Hybrid inheritance is not directly supported. It is supported indirectly by using interface.

Simple Inheritance

When a subclass is derived simply from it's parent class then this mechanism is known as simple inheritance or (a parent class with only one child class is known as single inheritance). In case of simple inheritance there is only one subclass and it's parent class. It is also called single inheritance or one level inheritance.



Pictorial Representation of Simple Inheritance

For Example,

```
class Parent
{
      int x;
```

```
	int y;
	void get(int p, int q)
	{
		x=p; y=q;
	}
	void Show()
	{
		System.out.println("x=" +x + " and y= " +y);
	}
}
class Child extends Parent
{
	public static void main(String args[])
	{
		Parent p = new Parent();
		p.get(5,6);
		p.Show();
	}
}
```

Subclass constructor A subclass constructor is used to construct the instance variables of both the subclass and the superclass. The subclass constructor uses the keyword super to invoke the constructor method of the superclass. The keyword super is used as

• Super may only be used within a subclass constructor method.

• The call to superclass constructor must appear as the first statement within the subclass constructor.

• The parameters in the super call must match the order and type of the instance variable declared in the superclass.

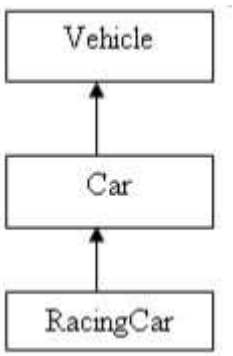
For Example, WAP that demonstrate the use of the keyword super

```
class Room
{
	int length;
	int breadth;
	Room(int x, int y)
	{
		length= x;
		breadth = y;
	}
	int area()
	{
```

```
            return(length*breadth);
      }
}
class BedRoom extends Room
{
      int height;
      BedRoom(int
                  x, int y, int z)
      {
            super(x,y);
            //call the superclass constructor with value x and y
            height = z;
      }
      int volume()
      {
            return(length*breadth*height);
      }
}
class SuperTest
{
      public static void main(String args[]){
            BedRoom room1 = new BedRoom(2,3,4);//call subclass
constructor
            int area1 = room1.area();//superclass method
            int volume1 = room1.volume();//call subclass method
            System.out.println("Area of the room is= " +area1);
            System.out.println("Volume of the bedroom is = "
                        +volume1);
      }
}
```

Multilevel Inheritance

When a subclass is derived from a derived class then this mechanism is known as the multilevel inheritance. The derived class is called the subclass or child class for it's parent class and this parent class works as the child class for it's just above (parent) class. Multilevel inheritance can go up to any number of levels.



Pictorial Representation of Simple and Multilevel Inheritance In Multilevel Inheritance a derived class with multilevel base classes is declared as follows:

```
class vehicle{
      //Instance variables;
      //Methods;
}
class Car extends Vehicle{
      //first level
      //Instance variables;
      //Methods;
      //..............
}
class RacingCar extends Car{
      //second level
      //Instance variables;Methods;
      //................................
}
```

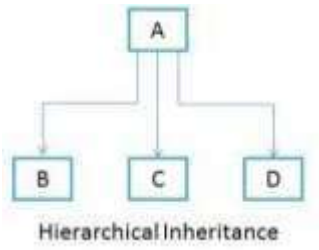
Q. WAP that demonstrate the concept of multilevel inheritance class

```
class Student{
      int roll_number;
      Student(int x){
            roll_number = x;
      }
      void put_number(){
            System.out.println("Roll number = " +roll_number);
      }
}
class Test extends Student{
      double sub1;double sub2;
      Test(int a, double b, double c){
            super(a);
            sub1 = b;sub2 = c;
      }
```

```
	void put_marks(){
		System.out.println("Marks in subject first is = "
				+sub1);
		System.out.println("Marks in subject second is = "
				+sub2);
	}
}
class Result extends Test{
	double total;
	Result(int a, double b, double c){
		super(a,b,c);
	}void display(){
		total = sub1 +sub2;
		put_number();
		put_marks();
		System.out.println("Total marks = " +total);
	}
}
class MultiLevelDemo{
	public static void main(String args[]){
		Result ram = new Result(5,55.5,67.5);
		ram.display();
	}
}
```

Hierarchical Inheritance

As you can see in the diagram below that when a class has more than one child classes (sub classes) or in other words more than one child classes have the same parent class then such kind of inheritance is known as hierarchical



Syntax:

```
	class A{
	//Instance variables;
	//Methods;
```

```
}
class B extends A{
      //Instance variables of Parent and its own;
      //Methods of parent and its own
      //..............
}
class C extends A{
      //Instance variables of Parent and its own;
      //Methods of parent and its own
      //..............
}
class D extends A{
      //Instance variables of Parent and its own;
      //Methods of parent and its own
      //..............
}
```

WAP to demonstrate the hierarchical inheritance

```
class A{
      public void methodA(){
            System.out.println("method of Class A");
      }
}
class B extends A{
      public void methodB(){
            System.out.println("method of Class B");
      }
}
class C extends A{
      public void methodC(){
            System.out.println("method of Class C");
      }
}
class D extends A{
      public void methodD(){
            System.out.println("method of Class D");
      }
}
class MyClass{
      public void methodB(){
            System.out.println("method of Class B");
      }
      public static void main(String args[]) {
            B obj1 = new B();
```

```
            C obj2 = new C();
            D obj3 = new D();
            obj1.methodA();
            obj2.methodA();
            obj3.methodA();
      }
}

}
```

Java Does not support Multiple and Hybrid inheritance directly it is supported by package and interface

2.4 extends and super keywords

extends keyword

The extends keyword extends a class (indicates that a class is inherited from another class). In Java, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

- subclass (child) - the class that inherits from another class
- superclass (parent) - the class being inherited from

To inherit from a class, use the extends keyword.

super keyword

The super keyword refers to superclass (parent) objects. It is used to call superclass methods, and to access the superclass constructor. The most common use of the super keyword is to eliminate the confusion between superclasses and subclasses that have methods with the same name.

2.5 Overloading/Overriding

Method Overloading

Method overloading in Java means having two or more methods (or functions) in a class with the same name and different arguments (or parameters). It can be with a different number of arguments or different data types of arguments.

Method overriding:

When a method is defined in subclass that has same name, same arguments and same return type as a method in the superclas and when that method is called, the method defined in the cubclass is invoked and executed instead of the one in superclass. This is known as method overriding. That is the subclass method override the superclass method.

WAP that demonstrate the concept of method overriding

```
class A{
	int i,j;
	A(int x, int y)   {
		i = x;
		j = y;
	}
	void display()    {
		System.out.println("i and j = " +i+ " " +j);
	}
}

class B extends A{
	int k;
	B(int a, int b, int c){
		super(a,b); k = c;
	}
	void display()    {
		System.out.println (" k = " +k);
	}
}

class MethodOverrideDemo{
	public static void main(String args[])    {
		B b1 = new B(2,3,4); // call constructor in class Bs
		b1.display();     // this call method in B
	}
}
```

Output:

K=4

In the above example, when the display() method is called with an object of type B, the version of display() defined in B is executed. That is, the version of display() in B overrides the version in superclass A.

2.6 final class and methods

- A final class is a class that can't be extended.
- A final method is a function in the base class that can't be overridden by any subclass.
- A final variable is considered read only. Once defined and initialized, it can't be modified. This is useful when defining constants.
- To mark a class, method, or variable as final, use the final keyword in its declaration.

final classes

The class that is declared as final is known as final class. When a class is declared as final, we cannot inherit such class that is we cannot create subclass of such class. Any attempt to inherit these classes will cause an error and the compiler will not allow it. Declaring a class as final implicitly declares all of its method as final.

Syntax:

```
final class A{
      ……;
      ……
}

class B extends A{      // error cannot subclass A


}
```

final variable and methods

All methods and variables can be overridden by default in subclass. If we wish to prevent the subclasses from overriding the members of the superclass, we can declare them as final using the keyword final as a modifier.

Syntax:

```
final type variable_name;
final int num=10;

final return_type methodname(){
      …
}

final void show(){
      …
}
```

2.7 abstract class and methods

abstract class

Any class that is declared as abstract is known as abstract class. Any class that contains one or more abstract methods must also be declared abstract. The general syntax of the abstract class is:

```
abstract class ClassName{
      ……
      …….
      abstract type methodName(parameterlist);

}
```

Example:

```
abstract class Shape{   //abstract class declaration

      ……
      ……
      abstract void draw(); // abstract method declaration

}
```

The abstract class has following properties

- We cannot create instance of the abstract class using new operator.

  For example Shape a = new Shape();
  Above declaration is illegal because shape is an abstract class.

- The abstract methods of an abstract class must be defined in its subclass.
- We cannot declare abstract constructor or abstract static methods. Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared abstract.

Abstract Methods and Classes more information

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.

An *abstract method* is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

abstract void moveTo(double deltaX, double deltaY);

If a class includes abstract methods, then the class itself *must* be declared abstract, as in:

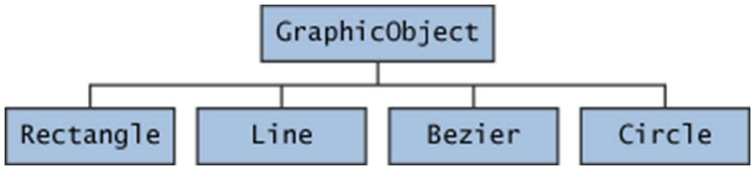
public abstract class GraphicObject {
  // declare fields
  // declare nonabstract methods

```
  abstract void draw();
}
```

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.

An Abstract Class Example

In an object-oriented drawing application, you can draw circles, rectangles, lines, Bezier curves, and many other graphic objects. These objects all have certain states (for example: position, orientation, line color, fill color) and behaviors (for example: moveTo, rotate, resize, draw) in common. Some of these states and behaviors are the same for all graphic objects (for example: position, fill color, and moveTo). Others require different implementations (for example, resize or draw). All GraphicObjects must be able to draw or resize themselves; they just differ in how they do it. This is a perfect situation for an abstract superclass. You can take advantage of the similarities and declare all the graphic objects to inherit from the same abstract parent object (for example, GraphicObject) as shown in the following figure.



Classes Rectangle, Line, Bezier, and Circle Inherit from GraphicObject

First, you declare an abstract class, GraphicObject, to provide member variables and methods that are wholly shared by all subclasses, such as the current position and the moveTo method. GraphicObject also declares abstract methods for methods, such as draw or resize, that need to be implemented by all subclasses but must be implemented in different ways. The GraphicObject class can look something like this:

```
abstract class GraphicObject {
   int x, y;
   ...
```

```
    void moveTo(int newX, int newY) {
        ...
    }
    abstract void draw();
    abstract void resize();
}
```

Each nonabstract subclass of GraphicObject, such as Circle and Rectangle, must provide implementations for the draw and resize methods:

```
class Circle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
class Rectangle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
```

When an Abstract Class Implements an Interface

In the section on Interfaces, it was noted that a class that implements an interface must implement *all* of the interface's methods. It is possible, however, to define a class that does not implement all of the interface's methods, provided that the class is declared to be abstract. For example,

```
abstract class X implements Y {
```

```
  // implements all but one method of Y
}

class XX extends X {
  // implements the remaining method in Y
}
```

In this case, class X must be abstract because it does not fully implement Y, but class XX does, in fact, implement Y.

Class Members

An abstract class may have static fields and static methods. You can use these static members with a class reference (for example, AbstractClass.staticMethod()) as you would with any other class.

2.8 Upcasting and down casting

Typecasting is one of the most important concepts which basically deals with the conversion of one data type to another datatype implicitly or explicitly.

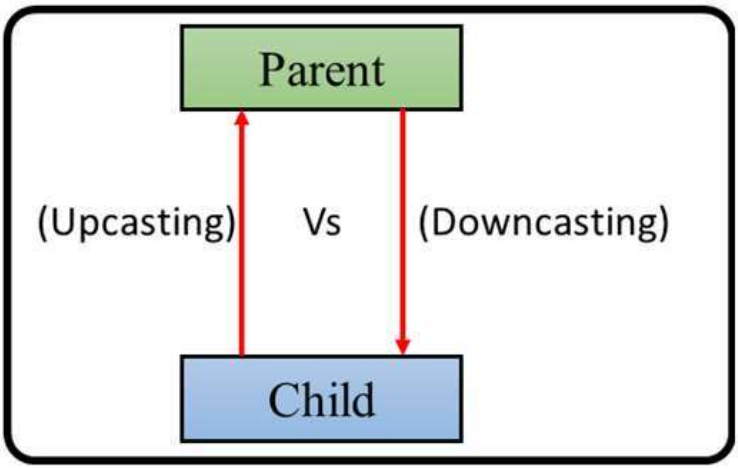
Just like the data types, the objects can also be typecasted. However, in objects, there are only two types of objects, i.e. parent object and child object. Therefore, typecasting of objects basically means that one type of object (i.e.) child or parent to another.

There are two types of typecasting. They are:

Upcasting: Upcasting is the typecasting **of a child object to a parent object**. Upcasting can be done implicitly. Upcasting gives us the flexibility to access the parent class members but it is not possible to access all the child class members using this feature. Instead of all the members, we can access some specified members of the child class. For instance, we can access the overridden methods.

Downcasting: Similarly, downcasting means the typecasting of a **parent object to a child object**. Downcasting cannot be implicit. The following image illustrates the concept of upcasting and

downcasting:



Animal.java

```
public class Animal {
	String name;

	public void makeNoise() {
		System.out.println("Animal Noise");
	}
}
```

Cat.java

```
public class Cat extends Animal {

	@Override
	public void makeNoise() {
		System.out.println("Mawo");
	}

}
```

Dog.java

```
public class Dog extends Animal{

	@Override
	public void makeNoise() {
		System.out.println("Wof wof!");
```

```
	}

	public void grawl() {
		System.out.println("grawl grawl");
	}
}
```

UpDownCasting.java

```
public class UpDownCasting {
	public static void main(String[] args) {

		//up casting
		Animal animal = new Dog();
		animal.makeNoise();

		/*in up casting Parent class reference only
		 *  call its overridden method not other
		 *  methods of child class
		 */

		if(animal instanceof Dog) {
			//down casting
			Dog dog = (Dog) animal;
			dog.grawl();
		}

		animal = new Cat();
		animal.makeNoise();

	}
}
```

2.9 Interface and implementations

Using the keyword **interface**, we can specify what a class must do, but not how it does it. Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body. Once it is defined, any number of classes can implement an **interface**. Also, one class can implement any number of interfaces. To implement an interface, a class must create the complete set of methods defined by the interface. However, each class is free to determine the details of its own implementation.

Defining an Interface

An interface is defined much like a class. The general form of an interface:
Syntax:

```
interface InterfaceName{
      type varName1 = value;
      type varName2 = value;
      ...
      ...
      type varnameN = value;

      return-type methodName1(parameter-list);
      return-type methodName2(parameter-list);
      ...
      ...
      return-type     methodNameN(parameter-list);
}
```

InterfaceName is the name of the interface, and can be any valid identifier. The methods that are declared have no bodies. They end with a semicolon after the parameter list. Each class that includes an interface must implement all of the methods. Variables can be declared inside of interface declarations. All methods and variables are implicitly **public**.

Example:

```
interface Callback{
      void callback(int param);
}
```

Implementing Interfaces

Once an **interface** has been defined, one or more classes can implement that interface. To implement an interface, include the **implements** clause in a class definition, and then create the methods defined by the interface. The general form of a class that includes the **implements** clause is:

```
class className implements InterfaceName{
// class-body
}
```

If a class implements more than one interface, the interfaces are separated with a comma. The methods that implement an interface must be declared **public**.

Here is a small example class that implements the **Callback** interface shown earlier.

```
class Client implements Callback{
	public void callback(int p)
	{
		System.out.println("callback called with " + p);
	}
	void nonIfaceMeth()  {
		System.out.println("Classes that implement interfaces " +
"may also define other members, too.");
	}
}
```

It is both permissible and common for classes that implement interfaces to define additional members of their own. For example, the following version of **Client** implements **callback()** and adds the method **nonIfaceMeth( )**:

The following example calls the **callback( )** method via an interface reference variable:

```
class TestIface{
	public static void main(String args[]){
		Client c = new Client();
		c.callback(42);
	}
}
```

The output of this program is shown here:
callback called with 42

Assignment:
Difference between Abstract class and an Interface with code example.
Write down the code to handle user defined exceptions.