Deen Dayal Upadhyaya College
Department of Computer Science
Workshop on Java J2SE

Dr. Rajni Bala
Associate Professor
Dept. of Computer Science
DDUC

Object & Class

- Any real world entity is known as object
- Each object is represented by

1. attributes(characterstics)
2. Methods(behaviour)

- Collection of objects of similar type is known as class

Class

- Each class defines a new data type and once defined is used for creating objects of that datatype.
- Class is template for an object and object is an instance of class.

General form of class

```
class classname {
   type instance-variable1;
   type instance-variable2;
   // ...
   type instance-variableN;

   type methodname1(parameter-list) {
         // body of method
   }
   type methodname2(parameter-list) {
         // body of method


 }
      // ...
 type methodnameN(parameter-list) {
      // body of method
 }
}
```

Class Example

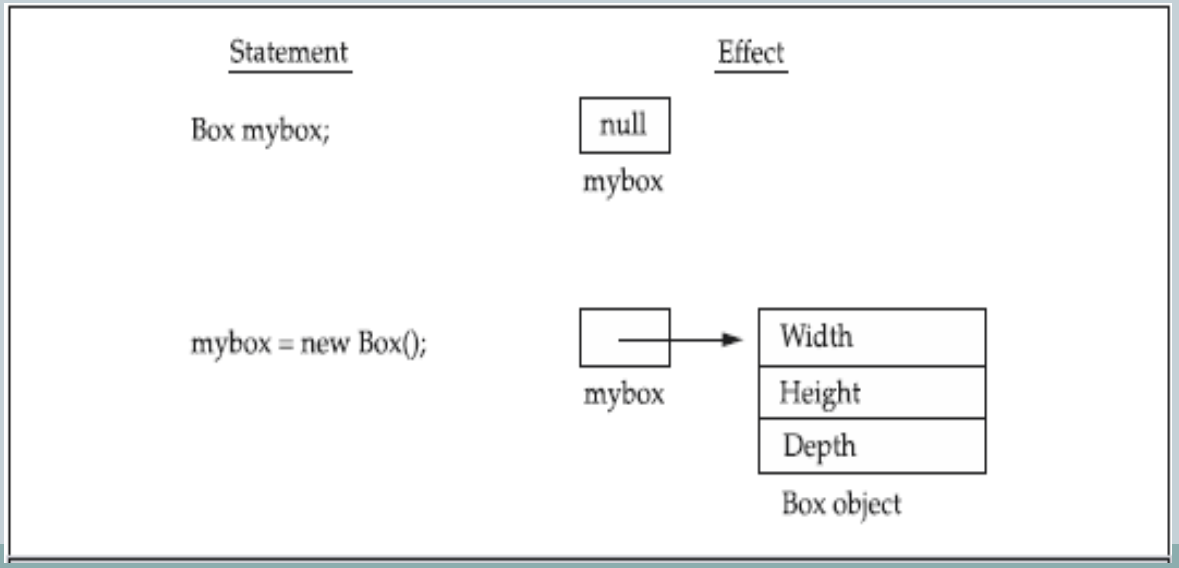
```java
/* A program that uses the Box class.
Call this file BoxDemo.java
*/
class Box
{
    double width;
    double height;
    double depth;
}
// This class declares an object of type Box.
class BoxDemo
{
    public static void main(String args[])
    {
        Box mybox = new Box();
        double vol;
        // assign values to mybox's instance variables
        mybox.width = 10;
        mybox.height = 20;
        mybox.depth = 15;
        // compute volume of box
        vol = mybox.width * mybox.height * mybox.depth;
        System.out.println("Volume is " + vol);
    }
}
```

```
Volume is 3000.0
```

Declaring Objects

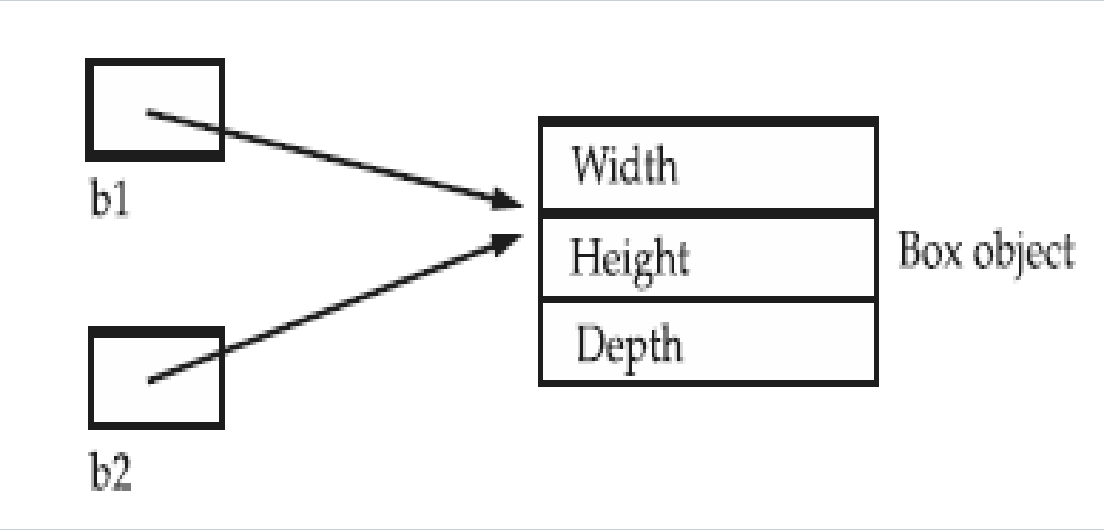
- Box mybox = new Box();

- Box mybox;// Declares a reference
- mybox = new Box();// allocates a memory



Assigning Object reference variables

- Box b1= new Box();
- Box b2 = b1;



When you assign one object reference variable to another object reference variable, you are not creating a copy of the object, you are only making a copy of the reference.

Adding Methods

```
class Box
{   double width;
    double height;
    double depth;
    // display volume of a box
    void volume()
    {
        System.out.print("Volume is ");
        System.out.println(width * height * depth);
    }
}
class BoxDemo
{
    public static void main(String args[])
    {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        // assign values to mybox1's instance variables
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;
        /* assign different values to mybox2's
        instance variables */
        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;
        // display volume of first box
        mybox1.volume();
        // display volume of second box
        mybox2.volume();
    }
}
```

```
Volume is 3000.0
Volume is 162.0
```

Constructors

```
class Box
{   double width;
    double height;
    double depth;
    // This is the constructor for Box.
    Box()
    {
        System.out.println("Constructing Box");
        width = 10;
        height = 10;
        depth = 10;
    }
    // This is the constructor for Box.
    Box(double w, double h, double d)
    {
        width = w;
        height = h;
        depth = d;
    }
    // display volume of a box
    void volume()
    {
        System.out.print("Volume is ");
        System.out.println(width * height * depth);
    }
}
```

```
class BoxDemo
{
    public static void main(String args[])
    {
        Box mybox1 = new Box();
        Box mybox2 = new Box(5,6,7);
        // display volume of first box
        mybox1.volume();
        // display volume of second box
        mybox2.volume();
    }
}
```

```
Constructing Box
Volume is 1000.0
Volume is 210.0
```

Passing parameters to functions

- Call-by-value: All primitive
- Call-by-reference : All Objects are passed by reference. Since array are implemented as objects, therefore arrays are also passed by reference.

this keyword

- This is a reference to the current object
- It can used inside any method to refer to current object.

```
Box(double width, double height, double depth) {
  this.width = width;
  this.height = height;
  this.depth = depth;
}
```

Garbage Collection

- All the object are allocated memory dynamically using a new operator
- In c++ memory for dynamically allocated objects is released manually by using delete operator.
- Java takes a different approach for de-allocating memory. It handles de-allocation for you automatically.
- when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.
- There is no explicit need to destroy objects as in C++. Garbage collection only occurs sporadically (if at all) during the execution of your program.

Finalize Method

- Sometimes an object will need to perform some action when it is destroyed. To handle such situations, Java provides a mechanism called *finalization*.
- To add a finalizer to a class, you simply define the **finalize( )** method. The Java runtime calls that method whenever it is about to recycle an object of that class.
- Inside the **finalize( )** method you will specify those actions that must be performed before an object is destroyed.
- The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects. Right before an asset is freed, the Java run time calls the **finalize( )** method on the object.

Finalize method

- The **finalize( )** method has this general form:

```
protected void finalize( )
{
  // finalization code here
}
```

Exercise

- Create a class stack
- Write a main class that reads a postfix expression and evaluates it.

OOPS Concept

- Encapsulation
- Inheritance
- Polymorphism
- Abstraction

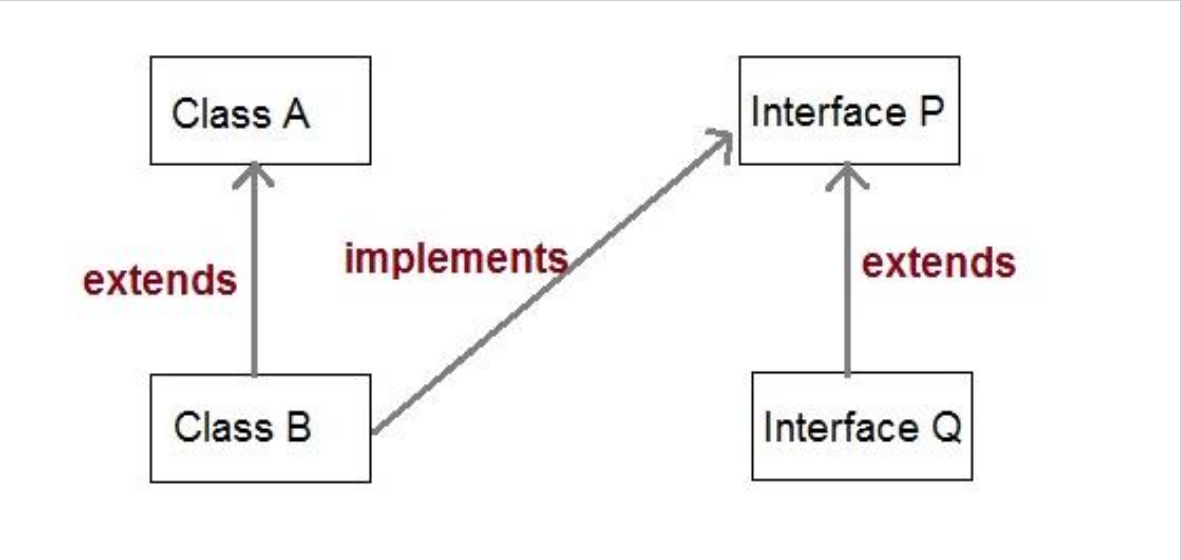
Encapsulation

- Binding data and methods as a single unit.
- Achieved through classes.

Inheritance

- Ability of an object to inherit the properties of other class is known as inheritance.
- When a class inherits another class it inherits all the data members and methods of that class.
- Inheritance is best understood by parent – child relationship (Super – sub class).
- It defines IS_A relationship between super(parent) and sub(child).
- Extends and implements are two keywords used for implementing inheritance.

Inheritance



Inheritance

- Vehicle is a super-class of Car
- Car is a sub-class of Vehicle
- Car IS_A Vehicle

```
class Vehicle.
{
  ......
}
class Car extends Vehicle
{
 .......   //extends the property of vehicle class.
}
```

Purpose of Inheritance

- Code re-use
- To use polymorhism(run-time)

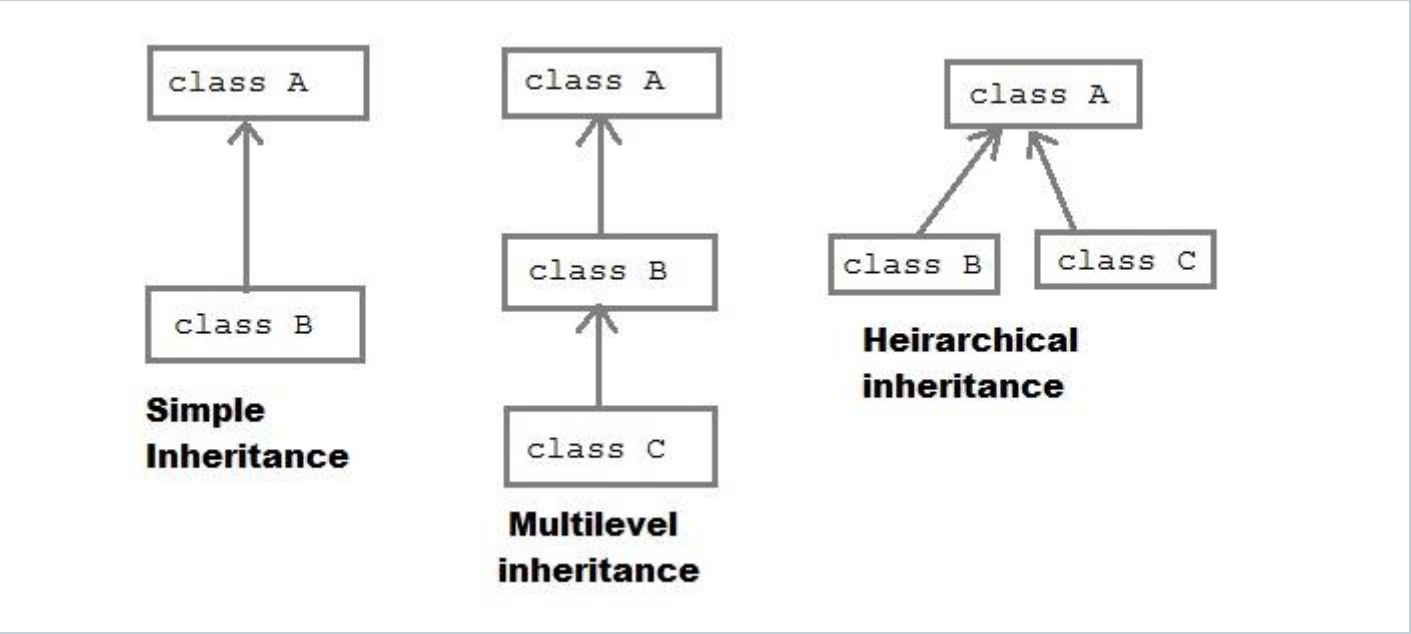
Example

```
class Vehicle
{
    String vehicleType;
}
public class Car extends Vehicle {

    String modelType;
    public void showDetail()
    {
        vehicleType = "Car";        //accessing Vehicle class member
        modelType = "sports";
        System.out.println(modelType+" "+vehicleType);
    }
    public static void main(String[] args)
    {
        Car car =new Car();
        car.showDetail();
    }
}
```

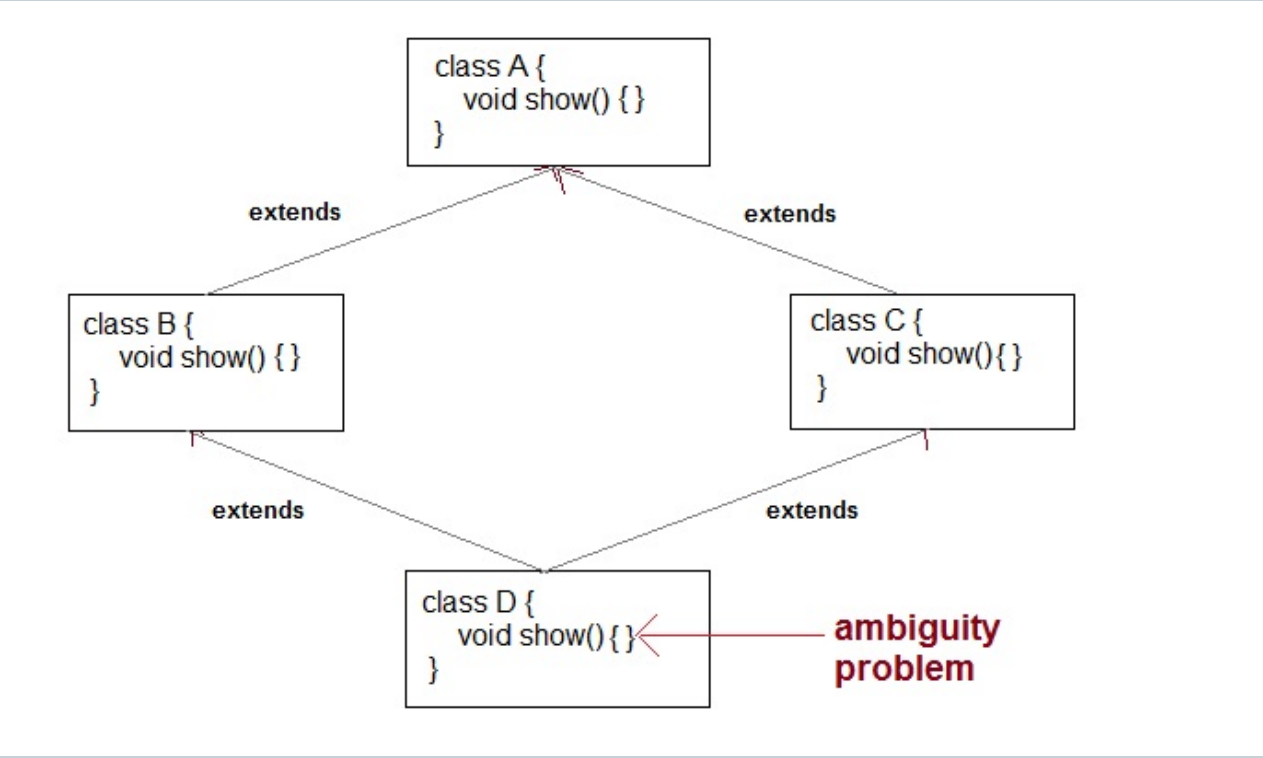
Types of inheritance

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance



Multiple inheritance

- Multiple inheritance not supported by Java.
- To remove ambiguity
- To provide more maintainable and clear design



Super Keyword

- It is used to refer to the immediate parent class

```
class Parent
{
    String name;

}
public class Child extends Parent {
    String name;
    public void details()
    {
        super.name = "Parent";      //refers to parent class member
        name = "Child";
        System.out.println(super.name+" and "+name);
    }
    public static void main(String[] args)
    {
        Child cobj = new Child();
        cobj.details();
    }
}
```

Super Keyword

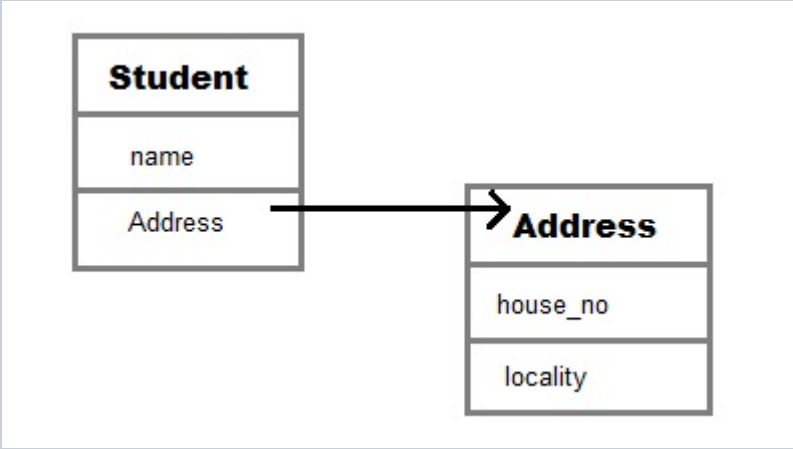
- Calling function from the parent class with the same name as in subclass.

```
class Parent
{
    String name;
    public void details()
    {
      name = "Parent";
        System.out.println(name);
    }
}
public class Child extends Parent {
    String name;
    public void details()
    {
        super.details();        //calling Parent class details() method
        name = "Child";
        System.out.println(name);
    }
    public static void main(String[] args)
    {
        Child cobj = new Child();
        cobj.details();
    }
}
```

Aggregation (HAS_A)

- Class A HAS_A relationship with class B if code in class A has a reference to an instance of class B.
- Student HAS_A address

```
class Student
{
 String name;
 Address ad;
}
```



Example of Aggregation

```
class Author
{
 String authorName;
 int age;
 String place;
 Author(String name,int age,String place)
 {
  this.authorName=name;
  this.age=age;
  this.place=place;
 }
 public String getAuthorName()
 {
  return authorName;
 }
 public int getAge()
 {
  return age;
 }
 public String getPlace()
 {
  return place;
 }
}
```

```
class Book
{
 String name;
 int price;
 Author auth;
 Book(String n,int p,Author at)
 {
  this.name=n;
  this.price=p;
  this.auth=at;
 }
 public void showDetail()
 {
  System.out.println("Book is"+name);
  System.out.println("price "+price);
  System.out.println("Author is "+auth.getAuthorName());
 }
}

class Test
{
 public static void main(String args[])
 {
  Author ath=new Author("Me",22,"India");
  Book b=new Book("Java",550,ath);
  b.showDetail();
 }
}
```

Polymorphism

- One interface, many methods
- Method overloading
- Method overriding

Method Overloading

- If two or more method in a class have same name but different parameters, it is known as overloading.
- Methods can be overloaded by changing the number of parameters or types of parameters.

```
class Calculate
{
 void sum (int a, int b)
 {
  System.out.println("sum is"+(a+b)) ;
 }
 void sum (float a, float b)
 {
  System.out.println("sum is"+(a+b));
 }
 Public static void main (String[] args)
 {
  Calculate  cal = new Calculate();
  cal.sum (8,5);      //sum(int a, int b) is method is called.
  cal.sum (4.6, 3.8); //sum(float a, float b) is called.
 }
}
```

Method overriding

- When a method in a sub-class has the same name and type signature as a method in super-class , the method is known as overridden method.
- It is also referred as runtime polymorphism.
- Static methods cannot be overridden.

Overriding example

```
package overriding;
//Method overriding.
class A
{
    int i, j;
    A(int a, int b)
    {
        i = a;
        j = b;
    }
    //display i and j
    void show()
    {
        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;
    }
    //display k - this overrides show() in A
    void show()
    {
        System.out.println("k: " + k);
    }
}
```

```
public class Override
{
    public static void main(String[] args)
    {
        B subOb = new B(1, 2, 3);
        subOb.show(); // this calls show() in B
    }
}
```

Dynamic Method Dispatch

- Method overriding forms the basis for one of Java's most powerful concepts: *dynamic method dispatch*.
- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- Dynamic method dispatch is important because this is how Java implements run-time polymorphism.
- a superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time.
- When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.

Dynamic method dispatch

```
package dynamicdispatch;
//Using run-time polymorphism.
class Figure
{
    double dim1;
    double dim2;
    Figure(double a, double b)
    {
        dim1 = a;
        dim2 = b;
    }
    double area()
    {
        System.out.println("Area for Figure is undefined.");
        return 0;
    }
}
class Rectangle extends Figure
{
    Rectangle(double a, double b)
    {
        super(a, b);
    }
    //override area for rectangle
    double area()
    {
        System.out.println("Inside Area for Rectangle.");
        return dim1 * dim2;
    }
}
```

```
class Triangle extends Figure
{
    Triangle(double a, double b)
    {
        super(a, b);
    }
    //override area for right triangle
    double area()
    {
        System.out.println("Inside Area for Triangle.");
        return dim1 * dim2 / 2;
    }
}
class FindArea {
    public static void main(String args[])
    {
        Figure f = new Figure(10, 10);
        Rectangle r = new Rectangle(9, 5);
        Triangle t = new Triangle(10, 8);
        Figure figref;
        figref = r;
        System.out.println("Area is " + figref.area());
        figref = t;
        System.out.println("Area is " + figref.area());
        figref = f;
        System.out.println("Area is " + figref.area());
    }
}
```

Abstract class

- If a class contain any abstract method then the class is declared as abstract class.
- Abstract class is never instantiated.
- Abstract classes can have constructors, member variables and normal methods.
- When you extend Abstract class ,you must define the abstract method in the child class or make the chid class abstract.
- Syntax
  - abstract class class_name { }

Abstract Method

- Method that are declared without any body within a class is known as abstract method.
- The body will defined by its subclass.
- Abstract method can never be static and final.
- Syntax

  - abstract return_type function_name() ;

  // no_definition

Example: Abstract Class

```
//Using abstract methods and classes.
//Using run-time polymorphism.
abstract class Figure
{
    double dim1;
    double dim2;
    Figure(double a, double b)
    {
        dim1 = a;
        dim2 = b;
    }
    abstract double area();
}
class Rectangle extends Figure
{
    Rectangle(double a, double b)
    {
        super(a, b);
    }
    //override area for rectangle
    double area()
    {
        System.out.println("Inside Area for Rectangle.");
        return dim1 * dim2;
    }
}
```

```
class Triangle extends Figure
{
    Triangle(double a, double b)
    {
        super(a, b);
    }
    //override area for right triangle
    double area()
    {
        System.out.println("Inside Area for Triangle.");
        return dim1 * dim2 / 2;
    }
}
class FindArea {
    public static void main(String args[])
    {
        Rectangle r = new Rectangle(9, 5);
        Triangle t = new Triangle(10, 8);
        Figure figref;
        figref = r;
        System.out.println("Area is " + figref.area());
        figref = t;
        System.out.println("Area is " + figref.area());
    }
}
```

Using final keyword

- Using final with any method prevents method to be overridden by its subclasses.
- Using final with class prevents the class to be inherited by some other class.

```
class A {
  final void meth() {
    System.out.println("This is a final method.");
  }
}

class B extends A {
  void meth() { // ERROR! Can't override.

    System.out.println("Illegal!");
  }
}
```

```
final class A {
  // ...
}

// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
  // ...
}
```

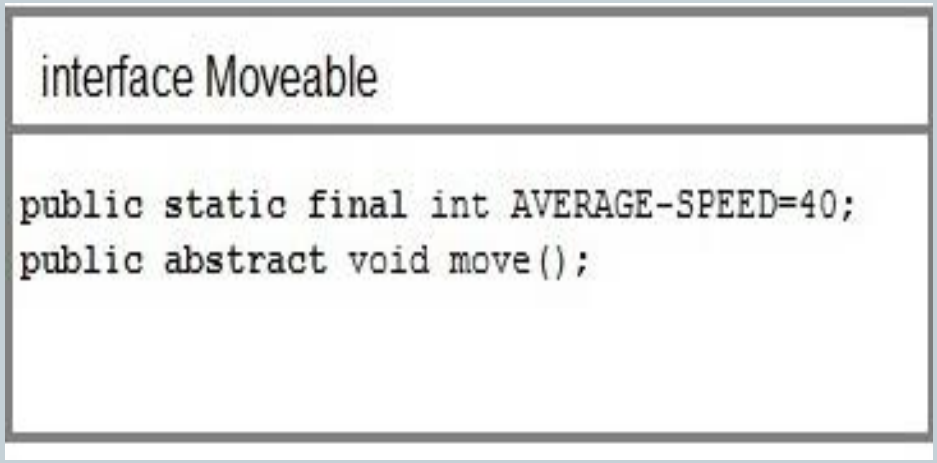
Interface

- It is pure abstract class.
- Syntactically similar to classes.
- Cannot create an instance of interface.
- In interface all methods are abstract.
- Interface defines what a class can do without saying anything about how a class will do it.
- Syntax

```
interface interface_name {  }
```

Example

```
interface Moveable
{
 int AVERAGE-SPEED=40;
 void move();
}
```



Rules for using interface

- Methods inside interface must not be static, final
- All variables declared inside interface are implicitly public static final variables.
- All methods inside Java interfaces are public and abstract.
- Interface can extend one or more interfaces.
- Interface cannot implement a class

Example

```
interface Moveable
{
 int AVG-SPEED = 40;
 void move();
}

class Vehicle implements Moveable
{
 public void move()
 {
  System .out. print in ("Average speed is"+AVG-SPEED");
 }
 public static void main (String[] arg)
 {
  Vehicle vc = new Vehicle();
  vc.move();
 }
}
Output:
Average speed is 40.
```

Applying Interfaces

```
// Define an integer stack interface.
interface IntStack {
  void push(int item); // store an item
  int pop(); // retrieve an item
}
```

```
// An implementation of IntStack that uses fixed storage.
class FixedStack implements IntStack {
  private int stck[];
  private int tos;

  // allocate and initialize stack
  FixedStack(int size) {
    stck = new int[size];
    tos = -1;
  }

  // Push an item onto the stack
  public void push(int item) {
    if(tos==stck.length-1) // use length member
      System.out.println("Stack is full.");
    else
      stck[++tos] = item;
  }

  // Pop an item from the stack
  public int pop() {
    if(tos < 0) {
      System.out.println("Stack underflow.");
      return 0;
    }
    else
      return stck[tos--];
  }
}
```

Object Class

- There is one special class, **Object**, defined by Java.
- All other classes are subclasses of **Object**. That is, **Object** is a superclass of all other classes.
- This means that a reference variable of type **Object** can refer to an object of any other class.
- Also, since arrays are implemented as classes, a variable of type **Object** can also refer to any array.

Methods in Object Class

Method	Purpose
Object clone( )	Creates a new object that is the same as the object being cloned.
boolean equals(Object *object*)	Determines whether one object is equal to another.
void finalize( )	Called before an unused object is recycled.
Class getClass( )	Obtains the class of an object at run time.
int hashCode( )	Returns the hash code associated with the invoking object.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
void wait( ) void wait(long *milliseconds*) void wait(long *milliseconds*, int *nanoseconds*)	Waits on another thread of execution.

Using Command-line arguments

- A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy—they are stored as strings in the **String** array passed to **main( )**.

```
// Display all command-line arguments.
class CommandLine {
  public static void main(String args[]) {
    for(int i=0; i<args.length; i++)
      System.out.println("args[" + i + "]: " +
                         args[i]);
  }
}
```