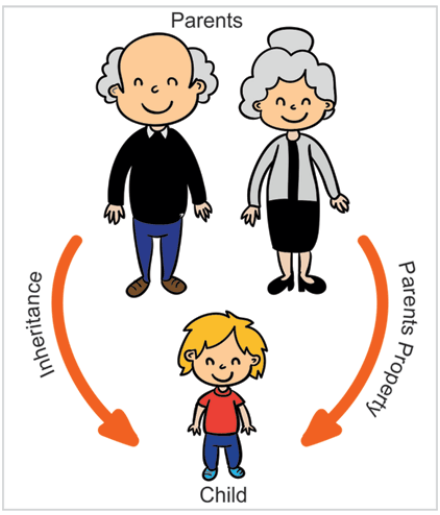
## MODULE II

# 1. INHERITANCE

## 1.1 Understanding Inheritance:




- Inheritancein java is a mechanism in which one object acquires all the properties and behaviors of parent object.
- Using inheritance we can create a general class that defines common members to a set of related items. This class can be inherited by other specific classes each adding those things that are unique to it.
- The inherited class is called a *superclass / Parent class*. The class that does the inheriting is called a *subclass / Child class*. Therefore, a subclass is a specialized version of a superclass.
- To inherit a class, we incorporate the definition of one class into another by using the ***extends*** keyword.
- The idea behind inheritance in java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of parent class, and you can add new methods and fields also.
- Inheritance represents the **IS-A** relationship, also known as parent-child relationship.
- Inheritance is used in Java for:
  - Code Reusability.
  - Method Overriding (so runtime polymorphism can be achieved).
- **Advantages:**
  - Application performance is improved because of less amount of development and execution time.
  - Redundancy (repetition) of the code is reduced or minimized so that we get consistence results and less storage cost.
- **Syntax:**

```
class Subclass-name extends Superclass-name
{
   //methods and fields
}
```

**Example:**

```
class Box
{
	double width;
	double height;
	double depth;

	Box(Box ob)	// passing object to constructor
	{
		width = ob.width;
		height = ob.height;
		depth = ob.depth;
	}

	// constructor used when all dimensions specified
	Box(double w, double h, double d)
	{
		width = w;
		height = h;
		depth = d;
	}
```

```
        // default constructor used when no dimensions specified
        Box()
        {
              width = -1; // use -1 to indicate
              height = -1; // an uninitialized
              depth = -1; // box
        }

        // constructor used when cube is created
        Box(double len)
        {
              width = height = depth = len;
        }

        // compute and return volume
        double volume()
        {
              return width * height * depth;
        }
  }

  // Here, Box is extended to include weight.
  class BoxWeight extends Box
  {
        double weight; // weight of box

        // constructor for BoxWeight
        BoxWeight(double w, double h, double d, double m)
        {
              width = w;
              height = h;
              depth = d;
              weight = m;
        }
  }

  class DemoBoxWeight
  {
        public static void main(String args[])
        {
              BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
              BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
              double vol;

              vol = mybox1.volume();
              System.out.println("Volume of mybox1 is " + vol);
              System.out.println("Weight of mybox1 is " + mybox1.weight);
              System.out.println();

              vol = mybox2.volume();
              System.out.println("Volume of mybox2 is " + vol);
              System.out.println("Weight of mybox2 is " + mybox2.weight);
        }
  }
```

**Output:**

```
  Volume of mybox1 is 3000.0
```

```
Weight of mybox1 is 34.3
Volume of mybox2 is 24.0
Weight of mybox2 is 0.076
```

One of the major advantages of inheritance is that once you have created a superclass that defines the attributes common to a set of objects, it can be used to create any number of more specific subclasses. Each subclass can precisely tailor its own classification.

**Example:**

```
class Box
{
	double width;
	double height;
	double depth;

	Box(Box ob)	// pass object to constructor
	{
		width = ob.width;
		height = ob.height;
		depth = ob.depth;
	}

	// constructor used when all dimensions specified
	Box(double w, double h, double d)
	{
		width = w;
		height = h;
		depth = d;
	}

	// default constructor used when no dimensions specified
	Box()
	{
		width = -1; // use -1 to indicate
		height = -1; // an uninitialized
		depth = -1; // box
	}

	// constructor used when cube is created
	Box(double len)
	{
		width = height = depth = len;
	}

	// compute and return volume
	double volume()
	{
		return width * height * depth;
	}
}

// Here, Box is extended to include weight.
class BoxWeight extends Box
{
	double weight; // weight of box

	// constructor for BoxWeight
	BoxWeight(double w, double h, double d, double m)
```

```
        {
                width = w;
                height = h;
                depth = d;
                weight = m;
        }
}
class ColorBox extends Box
{
        String color;

        // constructor for ColorBox
        ColorBox(double w, double h, double d, String c)
        {
                width = w;
                height = h;
                depth = d;
                color = c;
        }
}
class DemoBoxWeight
{
        public static void main(String args[])
        {
                BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
                BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
                ColorBox mybox3 = new ColorBox(2, 3, 4, "Red");
                double vol;

                vol = mybox1.volume();
                System.out.println("Volume of mybox1 is " + vol);
                System.out.println("Weight of mybox1 is " + mybox1.weight);
                System.out.println();

                vol = mybox2.volume();
                System.out.println("Volume of mybox2 is " + vol);
                System.out.println("Weight of mybox2 is " + mybox2.weight);

                vol = mybox3.volume();
                System.out.println("Volume of mybox3 is " + vol);
                System.out.println("Color of mybox3 is " + mybox3.color);
        }
}
```

**Output:**

```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3
Volume of mybox2 is 24.0
Weight of mybox2 is 0.076
Volume of mybox3 is 24.0
Color of mybox3 is Red
```

**Member Access and Inheritance:**

Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private**.

**Example:**

```
// Create a superclass.
```

```
class A
{
	int i;
	private int j; // private to A
	void setij(int x, int y)
	{
		i = x;
		j = y;
	}
}

// A's j is not accessible here.
class B extends A
{
	int total;
	void sum()
	{
		total = i + j; // ERROR, j is not accessible here
	}
}
class Access
{
	public static void main(String args[])
	{
		B subOb = new B();
		subOb.setij(10, 12);
		subOb.sum();
		System.out.println("Total is " + subOb.total);
	}
}
```

**A Superclass variable can reference a Subclass Object:**

A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass.

**Example:**

```
class RefDemo
{
	public static void main(String args[])
	{
		BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
		Box plainbox = new Box();
		double vol;
		vol = weightbox.volume();
		System.out.println("Volume of weightbox is " + vol);
		System.out.println("Weight of weightbox is " +
					weightbox.weight);
		System.out.println();

		// assign BoxWeight reference to Box reference
		plainbox = weightbox;
		vol = plainbox.volume(); // OK, volume() defined in Box
		System.out.println("Volume of plainbox is " + vol);

		/* The following statement is invalid because plainbox
		does not define a weight member. */
		// System.out.println("Weight of plainbox is " +
					plainbox.weight);
	}
```

```
}
```

**NOTE:**

- `Plainbox` is the type of the reference variable, not the type of the object that it refers to (which determines what members can be accessed).
- When a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by the superclass. This is why `plainbox` can't access `weight` even when it refers to a `BoxWeight` object. The reason for this is the superclass has no knowledge of what a subclass adds to it.

**1.2 Using super:**

- In the preceding examples, classes derived from Box were not implemented efficiently.
- The constructor for BoxWeight explicitly initializes the width, height, and depth fields of Box(). This duplicate code found in its superclass, which is inefficient. Also, it implies that a subclass has access to the members of superclass directly.
- As per the encapsulation feature of OOP a superclass should keep the details of its implementation to itself (that is, that keeps its data members private).
- In this case, there would be no way for a subclass to directly access or initialize these variables.
- Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword ***super***.
- The keyword super has two general forms:
  - It calls the immediate parent class constructor.
  - It is used to access a member of the superclass that has been hidden by a member of a subclass**:** We can use super keyword to access the member (instance variable of method) of parent class. It is used if parent class and child class have same member.

**Example:**

```
// Demo showing access of superclass member through super.
class Animal
{
      String color="white";
}
class Dog extends Animal
{
      String color="black";
      void printColor()
      {
            System.out.println(color);//prints color of Dog class
            System.out.println(super.color);//prints color of Animal class
      }
}
class TestSuper1
{
      public static void main(String args[])
      {
            Dog d=new Dog();
            d.printColor();
      }
}
```

**Output:**

```
black
white
```

**Example:**

```
// Demo showing call of superclass' constructor through super.
class Box
```

```
{
	private double width;
	private double height;
	private double depth;

	// construct clone of an object
	Box(Box ob) // pass object to constructor
	{
		width = ob.width;
		height = ob.height;
		depth = ob.depth;
	}

	// constructor used when all dimensions specified
	Box(double w, double h, double d)
	{
		width = w;
		height = h;
		depth = d;
	}
	// constructor used when no dimensions specified
	Box()
	{
		width = -1; // use -1 to indicate
		height = -1; // an uninitialized
		depth = -1; // box
	}
	// constructor used when cube is created
	Box(double len)
	{
		width = height = depth = len;
	}
	// compute and return volume
	double volume()
	{
		return width * height * depth;
	}
}

// BoxWeight now fully implements all constructors.
class BoxWeight extends Box
{
	double weight; // weight of box
	// construct clone of an object
	BoxWeight(BoxWeight ob) // pass object to constructor
	{
		super(ob);
		weight = ob.weight;
	}
	// constructor when all parameters are specified
	BoxWeight(double w, double h, double d, double m) {
		super(w, h, d); // call superclass constructor
		weight = m;
	}
	// default constructor
	BoxWeight() {
```

```
                super();
                weight = -1;
        }
        // constructor used when cube is created
        BoxWeight(double len, double m)
        {
                super(len);
                weight = m;
        }
}
class DemoSuper {
        public static void main(String args[]) {
                BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
                BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
                BoxWeight mybox3 = new BoxWeight(); // default
                BoxWeight mycube = new BoxWeight(3, 2);
                BoxWeight myclone = new BoxWeight(mybox1);
                double vol;

                vol = mybox1.volume();
                System.out.println("Volume of mybox1 is " + vol);
                System.out.println("Weight of mybox1 is " + mybox1.weight);
                System.out.println();

                vol = mybox2.volume();
                System.out.println("Volume of mybox2 is " + vol);
                System.out.println("Weight of mybox2 is " + mybox2.weight);
                System.out.println();

                vol = mybox3.volume();
                System.out.println("Volume of mybox3 is " + vol);
                System.out.println("Weight of mybox3 is " + mybox3.weight);
                System.out.println();

                vol = myclone.volume();
                System.out.println("Volume of myclone is " + vol);
                System.out.println("Weight of myclone is " + myclone.weight);
                System.out.println();

                vol = mycube.volume();
                System.out.println("Volume of mycube is " + vol);
                System.out.println("Weight of mycube is " + mycube.weight);
                System.out.println();
        }
}
```

**Output:**

```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3
Volume of mybox2 is 24.0
Weight of mybox2 is 0.076
Volume of mybox3 is -1.0
Weight of mybox3 is -1.0
Volume of myclone is 3000.0
Weight of myclone is 34.3
Volume of mycube is 27.0
```

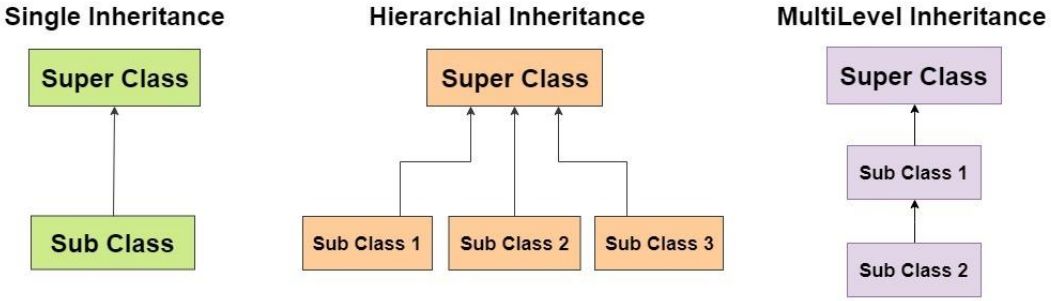
Weight of mycube is 2.0

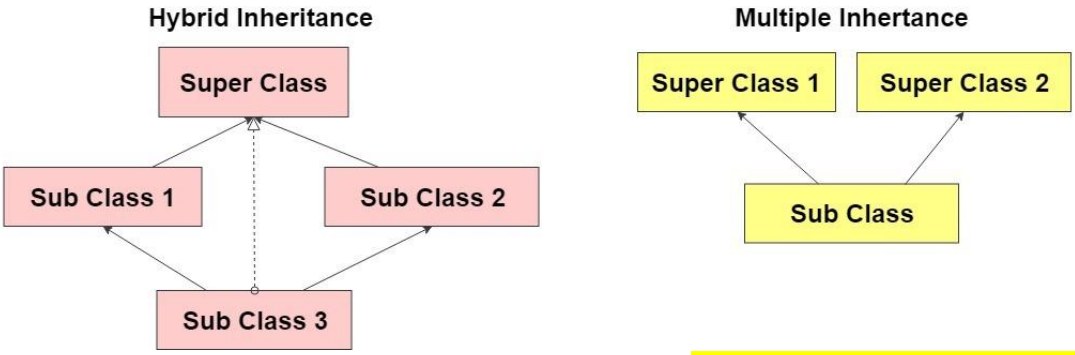
**NOTE:**

- When a subclass calls **super( )**, it is calling the constructor of its immediate superclass. Thus, **super( )** always refers to the super class immediately above the calling class.
- Also, **super( )** must always be the first statement executed inside a subclass constructor.

### 1.3 Types of Inheritance:

- Based on number of ways inheriting the feature of base class into derived class we have the following five types of inheritances:




1. **Single Inheritance:** In single inheritance there exists single base class and single derived class.
2. **Multilevel Inheritance:** In Multilevel inheritances there exists single base class, single derived class and multiple intermediate base classes.
3. **Multiple Inheritance:** In multiple inheritance there exist multiple classes and single derived class. The concept of multiple inheritance is not supported in java through concept of classes but it can be supported through the concept of interface.
4. **Hybrid Inheritance:** Combination of any inheritance type discussed so far. In the combination if one of the combination is multiple inheritance then the inherited combination is not supported by java through the class concept but it can be supported through the concept of interface.
5. **Hierarchical Inheritance:** 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 this type of inheritance is known as hierarchical inheritance.

**Example 1:Multilevel Inheritance**

```
import java.lang.*;
import java.io.*;
class Account
{
      String cust_name;
      int acc_no;
      Account(String a, int b)
      {
```

```
                cust_name=a;
                acc_no=b;
        }
        void display()
        {
                System.out.println ("Customer Name: "+cust_name);
                System.out.println ("Account No: "+acc_no);
        }
}
class Saving_Acc extends Account
{
        int min_bal,saving_bal;
        Saving_Acc(String a, int b, int c, int d)
        {
                super(a,b);
                min_bal=c;
                saving_bal=d;
        }
        void display()
        {
                super.display();
                System.out.println ("Minimum Balance: "+min_bal);
                System.out.println ("Saving Balance: "+saving_bal);
        }
}
class Acct_Details extends Saving_Acc
{
        int deposits, withdrawals;
        Acct_Details(String a, int b, int c, int d, int e, int f)
        {
                super(a,b,c,d);
                deposits=e;
                withdrawals=f;
        }
        void display()
        {
                super.display();
                System.out.println ("Deposit: "+deposits);
                System.out.println ("Withdrawals: "+withdrawals);
        }
}
class Multilevel
{
        public static void main(String args[])
        {
        Acct_Details A =new Acct_Details("Ajit Ram",666,1000,5000,500,900);
                A.display();
        }
}
```

**Output:**

Customer Name: Ajit Ram

Account No: 666

Minimum Balance: 1000

Saving Balance: 5000

Deposit: 5000

Withdrawals: 900

**Example 2: Multilevel Inheritance**

```
class person
{
	private String name;
	person(String s)
	{
		setName(s);
	}
	public void setName(String s)
	{
		name = s;
	}
	public String getName()
	{
		return name;
	}
	public void display()
	{
		System.out.println("Name = " + name);
	}
}
class Employee extends person
{
	private int empid;
	Employee(String sname,int id) //Constructor Method
	{
		super(sname);
		setEmpid(id);
	}
	public void setEmpid(int id)
	{
		empid = id;
	}
	public int getEmpid()
	{
		return empid;
	}
	public void display()
	{
		super.display();
		System.out.println("Empid = " + empid);
	}
};

class HourlyEmployee extends Employee
{
	private double hourlyRate;
	private int hoursWorked;

	HourlyEmployee(String sname,int id,double hr,int hw)
	{
		super(sname,id);
		hourlyRate = hr;
		hoursWorked = hw;
```

```
        }
        public double GetGrosspay()
        {
              return (hourlyRate * hoursWorked);
        }
        public void display()
        {
              super.display();
              System.out.println("Hourly Rate = " + hourlyRate);
              System.out.println("Hours Worked = " + hoursWorked);
              System.out.println("Gross pay = " + GetGrosspay());
        }
    };

class MultilevelInheritance
{
      public static void main(String[] args)
      {
      HourlyEmployee emp = new HourlyEmployee("Dinesh Thakur",1,15,1800);
            emp.display();
      }
}
```

**Output:**

Name =Dinesh Thakur

Empid = 1

Hourly Rate = 15.0

Hours Worked = 1800

Gross pay = 27000.0

**Example: Hierarchical Inheritance**

```
import java.io.*;
class A
{
      int rno;
      void setrno(int a)
      {
            rno=a;
      }
      void show()
      {
            System.out.println("Roll no is "+rno);
      }
}

class B extends A
{
      int sm;
      void setmarks(int x)
      {
            sm=x;
      }

      void show1()
```

```
	{
		System.out.println("Subject marks "+sm);
	}
}

class C extends A
{
	int lm;
	void getlab(int y)
	{
		lm=y;
	}

	void show2()
	{
		System.out.println("Lab marks are "+lm);
	}
}

class hirarc
{
	public static void main(String args[])
	{
		B b=new B();
		b.setrno(83);
		b.setmarks(40);

		C c=new C();
		c.getlab(50);

		b.show();
		b.show1();
		c.show2();
	}

}
```

**Output:**

Roll no is 83

Subject marks 40

Lab marks are 50

**Key Points about Inheritance:**

- In java programming one derived class can extends only one base class because java programming does not support multiple inheritance through the concept of classes, but it can be supported through the concept of Interface.
- Whenever we develop any inheritance application first creates an object of bottom most derived class but not for top most base class.
- When we create an object of bottom most derived class, first we get the memory space for the data members of top most base class, and then we get the memory space for data member of other bottom most derived class.
- Bottom most derived class contains logical appearance for the data members of all top most base classes.

- If we donot want to give the features of base class to the derived class then the definition of the base class must be preceded by final hence final base classes are not reusable or not inheritable.
- If we do not want to give some of the features of base class to derived class then such features of base class must be as private hence private features of base class are not inheritable or accessible in derived class.
- Data members and methods of a base class can be inherited into the derived class but constructors of base class cannot be inherited because every constructor of a class is made for initializing its own data members but not made for initializing the data members of other classes.
- An object of base class can contain details about features of same class but an object of base class never contains the details about special features of its derived class (this concept is known as scope of base class object).
- For each and every class in java there exists an implicit predefined super class called java.lang.Object, because it providers garbage collection facilities to its sub classes for collecting un-used memory space and improved the performance of java application.
- **Why multiple inheritance is not supported in java?**: To reduce the complexity and simplify the language, multiple inheritance is not supported in java. Consider a scenario where A, B and C are three classes. The C class inherits A and B classes. If A and B classes have same method and you call it from child class object, there will be ambiguity to call method of A or B class. Since compile time errors are better than runtime errors, java renders compile time error if you inherit two classes. So whether you have same method or different, there will be compile time error now.

## 2. POLYMORPHISM

### 2.1 Understanding Polymorphism:

- Polymorphism in java is a concept by which we can perform a *single action by different ways*. Polymorphism is derived from two Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.
- There are two types of polymorphism in java:
    - **Compile time polymorphism:** The type of polymorphism that is implemented when the compiler compiles a program is called compile-time polymorphism. Java supports compile-time polymorphism through method overloading. This type of polymorphism is also called as **static polymorphism** or **early binding**.
    - **Runtime polymorphism:** The type of polymorphism that is implemented dynamically when a program being executed is called run-time polymorphism. Java supports run-time polymorphism by dynamically dispatching methods at run time through method overriding. For this type of polymorphism, method invocations are resolved at run time by the JVM and not at the compile time. The run-time polymorphism is also called **dynamic polymorphism** or **late binding**.

### 2.2 Method Overloading:

- Method overloading is one type of static polymorphism. It supports polymorphism because it is one way that Java implements the "*one interface multiple methods*" paradigm.
- In languages that do not support method overloading, each method must be given a unique name.For instance, in C, the function abs( ) returns the absolute value of an integer, labs() returns the absolute value of a long integer, and fabs() returns the absolute value of a floating-point value.
- If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.
- If we have to perform only one operation, having same name of the methods increases the readability of the program.
- Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behaviour of the method because its name differs.

- So, we perform method overloading to figure out the program quickly.
- When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters.
- While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method.
- **Advantages:** Method overloading *increases the readability of the program*.
- There are two ways to overload the method in java
    1. By changing number of arguments
    2. By changing the data type

**Example:**

```
class OverloadDemo
{
	void test()
	{
		System.out.println("No parameters");
	}
	// Overload test for one integer parameter.
	void test(int a)
	{
		System.out.println("a: " + a);
	}

	// Overload test for two integer parameters.
	void test(int a, int b)
	{
		System.out.println("a and b: " + a + " " + b);
	}
	// overload test for a double parameter
	double test(double a)
	{
		System.out.println("double a: " + a);
		return a*a;
	}
}
class Overload
{
	public static void main(String args[])
	{
		OverloadDemo ob = new OverloadDemo();
		double result;

		// call all versions of test()
		ob.test();
		ob.test(10);
		ob.test(10, 20);
		result = ob.test(123.25);
		System.out.println("Result of ob.test(123.25): " + result);
	}
}
```

**Output:**

```
No parameters
a: 10
a and b: 10 20
```

double a: 123.25
Result of ob.test(123.25): 15190.5625

In some cases, Java's automatic type conversions can play a role in overload resolution. For example, consider the following program:

**Example:**

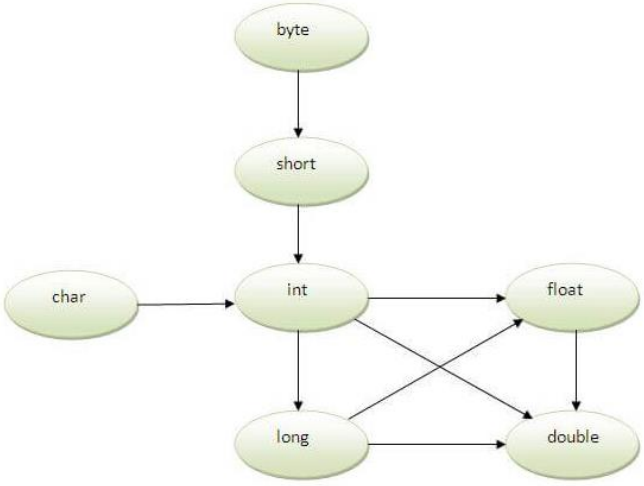
```
class OverloadDemo
{
	void test()
	{
		System.out.println("No parameters");
	}

	// Overload test for two integer parameters.
	void test(int a, int b)
	{
		System.out.println("a and b: " + a + " " + b);
	}
	// overload test for a double parameter
	void test(double a)
	{
		System.out.println("Inside test(double) a: " + a);
	}
}
class Overload
{
	public static void main(String args[])
	{
		OverloadDemo ob = new OverloadDemo();
		int i=88;

		ob.test();
		ob.test(10, 20);
		ob.test(i);
		ob.test(123.2);
	}
}
```

**Output:**

No parameters
a and b: 10 20
Inside test(double) a: 88.0
Inside test(double) a: 123.2




**NOTE:**

- Java will employ its automatic type conversions only if no exact match is found. One type is promoted to another implicitly if no matching data type is found. Let's understand the concept by the figure given here.
- There is no rule stating that overloaded methods must relate to one another. However, from a stylistic point of view, method overloading implies a relationship. Thus, while you can use the same name to overload unrelated methods, you should not. In practice, you should only overload closely related operations.
- If there are no matching type arguments in the method, and each method promotes similar number of arguments, there will be ambiguity.

**Example:**

```
class OverloadingCalculation3
{
	void sum(int a,long b){System.out.println("a method invoked");}
	void sum(long a,int b){System.out.println("b method invoked");}

	public static void main(String args[]){
		OverloadingCalculation3 obj=new OverloadingCalculation3();
		obj.sum(20,20);//now ambiguity
	}
}
```

**Output:**

Compile Time Error

**2.3 Constructor Overloading**:

- In addition to overloading normal methods, we can also overload constructor methods.

**Example:**

```
class Box
{
	double width; double height; double depth;
	// constructor used when all dimensions specified
	Box(double w, double h, double d)
	{
		width = w; height = h; depth = d;
	}
	// constructor used when no dimensions specified
	Box()
	{
		width = -1;  // use -1 to indicate
		height = -1; // an uninitialized
		depth = -1;  // box
	}

	// constructor used when cube is created
	Box(double len) { width = height = depth = len; }

	// compute and return volume
	double volume() { return width * height * depth; }
}
class OverloadCons
{
	public static void main(String args[])
	{
		// create boxes using the various constructors
		Box mybox1 = new Box(10, 20, 15);
		Box mybox2 = new Box();
		Box mycube = new Box(7);
		double vol;

		// get volume of first box
		vol = mybox1.volume();
		System.out.println("Volume of mybox1 is " + vol);

		// get volume of second box
		vol = mybox2.volume();
		System.out.println("Volume of mybox2 is " + vol);
```

```
                // get volume of cube
                vol = mycube.volume();
                System.out.println("Volume of mycube is " + vol);
        }
}
```

**Output:**

Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of mycube is 343.0

- We can pass objects to methods. One of the most common uses of object parameters involves constructors. Frequently, we construct a new object so that it is initially the same as some existing object. To do this, we must define a constructor that takes an object of its class as a parameter.

**Example:**

```
class Box
{
        double width;
        double height;
        double depth;

        Box(Box ob)
        {
                width = ob.width;
                height = ob.height;
                depth = ob.depth;
        }

        Box(double w, double h, double d)
        {
                width = w;
                height = h;
                depth = d;
        }

        Box()
        {
                width = -1;
                height = -1;
                depth = -1;
        }

        Box(double len)
        {
                width = height = depth = len;
        }

        double volume()
        {
                return width * height * depth;
        }
}
class OverloadCons2
{
        public static void main(String args[])
        {
                Box mybox1 = new Box(10, 20, 15);
```

```
                Box mybox2 = new Box();
                Box mycube = new Box(7);
                Box myclone = new Box(mybox1); // create copy of mybox1
                double vol;

                vol = mybox1.volume();
                System.out.println("Volume of mybox1 is " + vol);

                vol = mybox2.volume();
                System.out.println("Volume of mybox2 is " + vol);

                vol = mycube.volume();
                System.out.println("Volume of cube is " + vol);

                vol = myclone.volume();
                System.out.println("Volume of clone is " + vol);
        }
}
```

**Output:**

Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of cube is 343.0
Volume of clone is 3000.0

**2.4 Method Overriding**:

- In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to **override** the method in the superclass.
- When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

**Example:**

```
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);
        }
}

// Method overriding.
class B extends A
{
        int k;
        B(int a, int b, int c)
        {
                super(a, b);
                k = c;
        }
```

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

**Output:**k: 3

- If you wish to access the superclass version of an overridden method, you can do so by using super.

**Example:**

```
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);
        }
}

// Method overriding.
class B extends A
{
        int k;
        B(int a, int b, int c)
        {
                super(a, b);
                k = c;
        }

        void show()
        {
                super.show();
                System.out.println("k: " + k);
        }
}
class Override
{
        public static void main(String args[])
        {
                B subOb = new B(1, 2, 3);
                subOb.show(); // this calls show() in B
        }
```

```
}
```

**Output:** i and j: 1 2
k: 3

- Method overriding occurs only when the names and the type signatures of the two methods are identical. If they are not, then the two methods are simply overloaded.
- **Usage of Java Method Overriding**
  1. Method overriding is used to provide specific implementation of a method that is already provided by its super class.
  2. Method overriding is used for runtime polymorphism
- **Rules for Java Method Overriding**
  1. Method must have same name as in the parent class
  2. The argument list should be exactly the same as that of the overridden method.
  3. The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.
  4. Must be IS-A relationship (inheritance).
  5. 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 overridding method in the sub class cannot be either private or protected. i.e. While overriding, visibility of a method can be increased but can not be reduced.

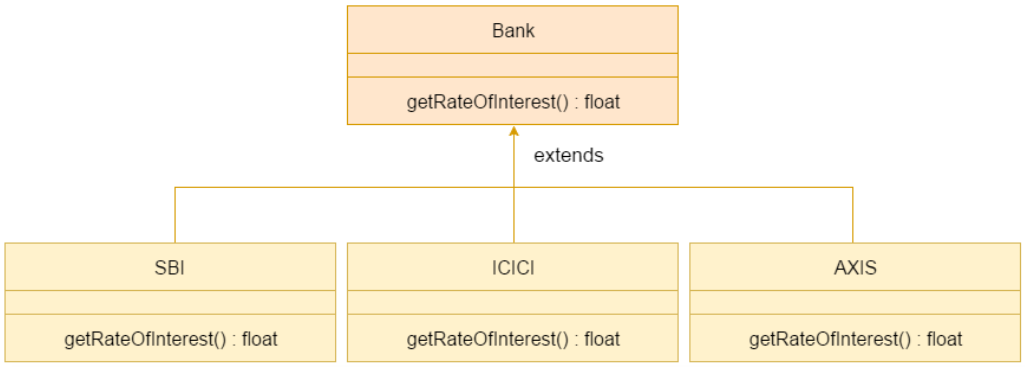
```
class X
{
    void methodOfX()
    {
        System.out.println("Class X");
    }
}

class Y extends X
{
    protected void methodOfX()
    {
        System.out.println("Class Y");
    }
}

class Z extends Y
{
    public void methodOfX()
    {
        System.out.println("Class Z");
    }
}
```

  6. Instance methods can be overridden only if they are inherited by the subclass.
  7. A method declared final cannot be overridden.
  8. A method declared static cannot be overridden but can be re-declared. Even main() method cannot be overridden.
  9. If a method cannot be inherited, then it cannot be overridden.
  10. A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.
  11. A subclass in a different package can only override the non-final methods declared public or protected.
  12. Constructors cannot be overridden.

**Real Life Example:**




```
class Bank
{
	int getRateOfInterest(){return 0;}
}

class SBI extends Bank
{
	int getRateOfInterest(){return 8;}
}

class ICICI extends Bank
{
	int getRateOfInterest(){return 7;}
}
class AXIS extends Bank
{
	int getRateOfInterest(){return 9;}
}

class Test2
{
	public static void main(String args[])
	{
		SBI s=new SBI();
		ICICI i=new ICICI();
		AXIS a=new AXIS();
	System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
	System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
	System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
	}
}
```

**Output:**

SBI Rate of Interest: 8
ICICI Rate of Interest: 7
AXIS Rate of Interest: 9

**Difference between method overloading and method overriding**

| No. | Method Overloading | Method Overriding |
|---|---|---|
| 1. | Method overloading is used *to increase the readability* of the program. | Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class. |

| | | |
|---|---|---|
| 2. | Method overloading is performed *within class*. | Method overriding occurs *in two classes* that have IS-A (inheritance) relationship. |
| 3. | In case of method overloading, *parameter must be different*. | In case of method overriding, *parameter must be same*. |
| 4. | Method overloading is the example of *compile time polymorphism*. | Method overriding is the example of *run time polymorphism*. |
| 5. | In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter. | *Return type must be same or covariant* in method overriding. |

**2.5 Dynamic Method Despatch**:

- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- Through Dynamic method dispatch 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 runtime.
- When an overridden method is called through a super class 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.
- The type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed. Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

**Example:**

```
class A
{
	void callme()
	{
		System.out.println("Inside A's callme method");
	}
}
class B extends A
{
	// override callme()
	void callme()
	{
		System.out.println("Inside B's callme method");
	}
}
class C extends A
{
	// override callme()
	void callme()
	{
		System.out.println("Inside C's callme method");
	}
}
```

```
class Dispatch
{
	public static void main(String args[])
	{
		A a = new A();
		B b = new B();
		C c = new C();

		A r;
		r = a;
		r.callme(); // calls A's version of callme
		r = b;
		r.callme(); // calls B's version of callme
		r = c;
		r.callme(); // calls C's version of callme
	}
}
```

**Output:**

Inside A's callme method
Inside B's callme method
Inside C's callme method

- Overridden methods are another way that Java implements the "*one interface, multiple methods*" aspect of polymorphism.
- By combining inheritance with overridden methods, a super class can define the general form of the methods that will be used by all of its subclasses.
- Dynamic, run-time polymorphism is one of the most powerful mechanisms that object oriented design brings to bear on code reuse and robustness.

**More Practical Example:**

```
class Figure
{
	double dim1, 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 FindAreas
    {
        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());
        }
    }
```

**Output:**

Inside Area for Rectangle.
Area is 45.0
Inside Area for Triangle.
Area is 40.0
Area for Figure is undefined.
Area is 0.0

**2.5.1 Upcasting & Downcasting**:

- Substituting a subclass instance for its superclass is called "*upcasting*".
- Upcasting is *always safe* because a subclass instance possesses all the properties of its superclass and can do whatever its superclass can do.
- The compiler checks for valid upcasting and issues error "incompatible types" otherwise.

```
Circle c1 = new Cylinder(1.1, 2.2);
// Compiler checks to ensure that R-value is a subclass of L-value.
Circle c2 = new String();    // Compilation error: incompatible types
```

- We can revert a substituted instance back to a subclass reference. This is called "*downcasting*".

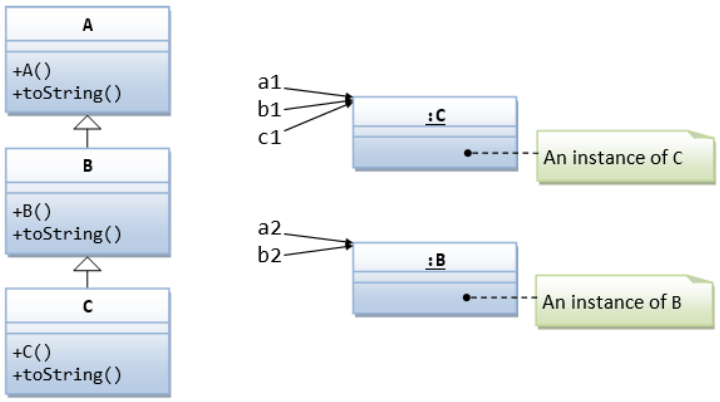
```
Circle c1 = new Cylinder(1.1, 2.2);  // upcast is safe
Cylinder cy1 = (Cylinder) c1;        // downcast needs the casting operator
```

- Downcasting requires *explicit type casting operator* in the form of prefix operator (*new-type*). Downcasting is not always safe, and throws a runtime ClassCastException if the instance to be downcasted does not belong to the correct subclass. A subclass object can be substituted for its superclass, but the reverse is not true.

**Another Example:**

```
class A
{
      A() { System.out.println("Constructed A"); }
      public String toString() { return "This is A"; }
}
class B extends A
{
   B()
   {
      super();
      System.out.println("Constructed B");
   }
public String toString() { return "This is B"; }
}
class C extends B
{
   C()
   {
      super();
      System.out.println("Constructed C");
   }
public String toString() { return "This is C"; }
}
public class Test
{
   public static void main(String[] args)
   {
      A a1 = new C();   // upcast
      System.out.println(a1);  // run C's toString()
      B b1 = (B)a1;     // downcast okay
      C c1 = (C)b1;     // downcast okay

      A a2 = new B();   // upcast
      System.out.println(a2);  // run B's toString()
      B b2 = (B)a2;     // downcast okay
      C c2 = (C)a2;     // compilation okay, but runtime error
   }
}
```




**2.6 final keyword in Java**:

- The ***final*** keyword is used in different contexts. First of all, final is a non-access modifier applicable only to a variable, a method or a class.Following are different contexts where final is used:

- final variable: Used to create constants
- final methods: Prevent method overriding
- final classes: Prevent Inheritance

### 2.6.1. final variables:

- If you make any variable as final, you cannot change the value of final variable (It will be constant).
- Hence final variables must be used only for the values that we want to remain constant throughout the execution of program.
- It is good practice to represent final variables in all uppercase, using underscore to separate words.
- We must initialize a final variable, otherwise compiler will throw compile-time error.A final variable can only be initialized once, either via an initializer or an assignment statement. There are three ways to initialize a final variable:
  - You can initialize a final variable when it is declared.This approach is the most common. A final variable is called blank final variable, if it is not initialized while declaration. Below are the two ways to initialize a blank final variable.
    - A blank final variable can be initialized inside instance-initializer block or inside constructor. If you have more than one constructor in your class then it must be initialized in all of them, otherwise compile time error will be thrown.
    - A blank final static variable can be initialized inside static block.

**Example:**

```
class FinalDemo
{
	// a final variable with direct initialize
	final int THRESHOLD = 5;

	// a blank final variable
	final int CAPACITY;

	// another blank final variable
	final int  MINIMUM;

	// a final static variable with direct initialize
	static final double PI = 3.141592653589793;

	// a  blank final static  variable
	static final double EULERCONST;

	//instance initializer block forinitializing CAPACITY
	{
		CAPACITY = 25;
	}
	//staticinitializer block for initializing EULERCONST
	static
	{
		EULERCONST = 2.3;
	}

	// constructor for initializing MINIMUM
	public FinalDemo()
	{
		MINIMUM = -1;
	}
}
```

- **Reference final variable:** When a final variable is a reference to an object, then this final variable is called reference final variable. In case of a reference final variable, internal

state of the object pointed by that reference variable can be changed. Note that this is not re-assigning. This property of *final* is called ***non-transitivity***.

**Example:**

```
class FinalRef
{
    public static void main(String[] args)
    {
        // a final reference variable sb
        final StringBuilder sb = new StringBuilder("India");

        System.out.println(sb);

        // changing internal state of object
        // reference by final reference variable sb
        sb.append(" is great");
        System.out.println(sb);
    }
}
```

**Output:**

```
India
India is great
```

- When a final variable is created inside a method/constructor/block, it is called local final variable, and it must initialize once where it is created.

**Example:**

```
class FD
{
    public static void main(String args[])
    {
        // local final variable
        final int i;
        i = 20;
        System.out.println(i);
    }
}
```

- **Difference between C++ const variables and Java final variables:** const variables in C++ must be assigned a value when it is declared whereas in Java, it is not necessary. A final variable can be assigned value later, but only once.

### 2.6.2. final methods:

- When a method is declared with *final* keyword, it is called a final method.
- A final method cannot be overridden.
- We must declare methods with final keyword for which we required to follow the same implementation throughout all the derived classes. The following program shows error.

**Example:**

```
class Bike
{
      final void run()
      {
            System.out.println("running");
      }
}

class Honda extends Bike
{
      void run()  // error since final methods can not be overridden
      {
            System.out.println("running safely with 100kmph");
      }

      public static void main(String args[])
      {
            Honda honda= new Honda();
            honda.run();
```

```
        }
}
```

- **NOTE:** When a small final method is called, often Java compiler copies the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call. Inlining is only an option with final methods. Normally, Java resolves calls to methods dynamically, at run time. This is called *late binding*. However, since final methods cannot be overridden, a call to one can be resolved at compile time. This is called *early binding*. In this way final provides a performance enhancement.

**2.6.3. final class:**

- When a class is declared as final then it cannot be subclassed i.e. no any other class can extend it.
- This is particularly useful for creating an immutable class like the predefined String class.

**Example:**

```
final class A
{
     // methods and fields
}

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

- Declaring a class as final implicitly declares all of its methods as final too.
- It is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

## 3. The Object CLASS

- Object is a special class defined by Java which is the super class of all the classes. All other classes are sub-classes of Object.
- This means that a reference variable of type Object can refer to an object of any other class.
- Since arrays are implemented as classes, a variable of type Object can also refer to any array.
- Object class defines the following methods, which means that they are available in every object:

| 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. Compares the contents of two objects. It returns true if the objects are equivalent and false otherwise. |
| **void finalize()** | Called before an unused object is recycled. |
| **Class getClass()** | Obtains the class of an object at run time. This is a final method. |
| **int hashCode()** | Returns the hash code associated with the invoking object. |
| **void notify()** | Reusmes execution of a thread waiting on the invoking object. This is a final method. |
| **void notifyAll()** | Reusmes execution of all threads waiting on the invoking object. This is a final method. |
| **String toString()** | Returns a string that describes the object. This method is automatically called when an object is output using println(). |
| **void wait()**<br>**void wait(long *milliseconds*)**<br>**void wait(long *milliseconds*, int *nanoseconds*)** | Wait on another thread of execution. This is a final method. |

**4. Wrapper Classes**:

- A Wrapper class is a class whose object wraps or contains a primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store a primitive data types. In other words, we can wrap a primitive value into a wrapper class object.
- Wrapper class provides the mechanism *to convert primitive data types into object and object into primitive data types*.
- **Need of Wrapper Classes:**
    1. They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
    2. The classes in java.util package handle only objects and hence wrapper classes help in this case also.
    3. Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.
    4. An object is needed to support synchronization in multithreading.
- **Primitive Data types and their Corresponding Wrapper class:**

| Primitive Type | Wrapper class |
|---|---|
| boolean | Boolean |
| char | Character |
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |

**Character Wrapper Class:**

- Character is a wrapper around a char. The constructor for Character is

  **Character(char ch):**ch specifies the character that will be wrapped by the Character object being created.
- To obtain the char value contained in a Character object, call charValue( ), shown here:

  **char charValue( ):** Returns the encapsulated character.

**Boolean Wrapper Class:**

- Boolean is a wrapper around boolean values. It defines these constructors:

  **Boolean(boolean boolValue):** boolValue must be either true or false.

  **Boolean(String boolString):** if boolString contains the string "true" (in uppercase or lowercase), then the new Boolean object will be true. Otherwise, it will be false.
- To obtain the boolean value contained in a Boolean object, call booleanValue( ), shown here:

  **Boolean booleanValue( ):** Returns the boolean equivalent of the invoking object.

**Numeric Type Wrapper Classes:**

- Byte, Short, Integer, Long, Float, and Double are numeric type wrapper classes.
- All of the numeric type wrappers inherit the abstract class Number.

- Number declares methods that return the value of an object in each of the different number formats. These methods are shown here:

  byte byteValue( )
  double doubleValue( )
  float floatValue( )
  int intValue( )
  long longValue( )
  short shortValue( )

- For example, doubleValue( ) returns the value of an object as a double, floatValue( ) returns the value as a float, and so on. These methods are implemented by each of the numeric type wrappers.
- All of the numeric type wrappers define constructors that allow an object to be constructed from a given value, or a string representation of that value. For example, here are the constructors defined for Integer:

  Integer(int num)
  Integer(String str)

- If str does not contain a valid numeric value, then a NumberFormatException is thrown.
- All of the type wrappers override toString( ). It returns the human-readable form of the value contained within the wrapper

**Autoboxing**: Automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing. For example – conversion of int to Integer, long to Long, double to Double etc.

**Example:**

```
import java.util.ArrayList;
public class WrapperExample1
{
   public static void main(String args[])
   {
         //Converting int into Integer
         int a=20;
         Integer i=Integer.valueOf(a);//converting int into Integer
         Integer    j=a;//autoboxing,    now    compiler    will    write
                                 Integer.valueOf(a) internally
         System.out.println(a+" "+i+" "+j);  //Prints 20 20 20
   }
}
```

**Unboxing**: It is just the reverse process of autoboxing. Automatically converting an object of a wrapper class to its corresponding primitive type is known as unboxing. For example – conversion of Integer to int, Long to long, Double to double etc.

**Example:**

```
public class WrapperExample2
{
         public static void main(String args[])
         {
```

```
                //Converting Integer to int
                Integer a=new Integer(3);
                int i=a.intValue();//converting Integer to int
                int j=a;//unboxing, now compiler will write a.intValue()
                        internally
                System.out.println(a+" "+i+" "+j); //Prints 3 3 3
        }
}
```

**Autoboxing/Unboxing in Expressions**:

- Autoboxing and unboxing take place whenever a conversion into an object or from an object is required. This applies to expressions. Within an expression, a numeric object is automatically unboxed. The outcome of the expression is reboxed, if necessary.For example, consider the following program:

**Example:**

```
class AutoBox3
{
   public static void main(String args[])
   {
         Integer iOb, iOb2;
         int i;
         iOb = 100;
         System.out.println("Original value of iOb: " + iOb);

         // The following automatically unboxes iOb, performs the
         increment, and then reboxes the result back into iOb.

         ++iOb;
         System.out.println("After ++iOb: " + iOb);

         // Here, iOb is unboxed, the expression is evaluated, and the
         result is reboxed and assigned to iOb2.

         iOb2 = iOb + (iOb/3);
         System.out.println("iOb2 after expression: " + iOb2);

         // The same expression is evaluated, but theresult is not
         reboxed.

         i = iOb + (iOb / 3);
         System.out.println("i after expression: " + i);
   }
}
```

**Output**:

```
Original value of iOb: 100
After ++iOb: 101
iOb2 after expression: 134
i after expression: 134
```

- Auto-unboxing also allows us to mix different types of numeric objects in an expression. Once the values are unboxed, the standard type promotions and conversions are applied:

**Example:**

```
class AutoBox4
{
	public static void main(String args[])
	{
		Integer iOb = 100;
		Double dOb = 98.6;
		dOb = dOb + iOb;
		System.out.println("dOb after expression: " + dOb);
	}
}
```

**Output**:

```
dOb after expression: 198.6
```

- Because of auto-unboxing, we can use integer numeric objects to control a switch statement.

**Example:**

```
Integer iOb = 2;
switch(iOb)
{
	case 1:
		System.out.println("one");
		break;
	case 2:
		System.out.println("two");
		break;
	default:
		System.out.println("error");
}
```

**Output**:

```
dOb after expression: 198.6
```

**Autoboxing/Unboxing Boolean and Character Values**:

- Autoboxing/unboxing applies to Boolean and Character wrappers also.

**Example:**

```
class AutoBox5
{
	public static void main(String args[])
	{
		// Autobox/unbox a boolean.
		Boolean b = true;
```

```
            // Below, b is auto-unboxed when used in a conditional
            expression, such as an if.

            if(b)
                  System.out.println("b is true");

            // Autobox/unbox a char.
            Character ch = 'x'; // box a char
            char ch2 = ch; // unbox a char
            System.out.println("ch2 is " + ch2);
      }
}
```

**Output**:

```
b is true
ch2 is x
```

- Because of auto-unboxing, a Boolean object can now also be used to control any loop statements. When a Boolean is used as the conditional expression of a loop, it is automatically unboxed into its boolean equivalent.

**Example:**

```
Boolean b;
// ...
while(b) { // ...
```

- Autoboxing/unboxing can also help prevent errors. For example, consider the following program.

**Example:**

```
class UnboxingError
{
      public static void main(String args[])
      {
            Integer iOb = 1000; // autobox the value 1000
            int i = iOb.byteValue(); // manually unbox as byte !!!
            System.out.println(i);  // does not display 1000 !
      }
}
```

**Output**:

```
-24
```

**Reason**:

Manually unboxed as byte instead of int by mistake. Replace *`int i = iOb.byteValue()`* with *`int i = iOb.intValue()`* to get the expected output.

**CAUTION:**

We should restrict our use of the type wrappers to only those cases in which an object representation of a primitive type is required because autoboxing/unboxing adds overhead that is not present if the primitive type is used. Autoboxing/unboxing was not

added to Java as a "back door" way of eliminating the primitive types. The following is an example of bad use of autoboxing/unboxing:

```
Double a, b, c;
a = 10.0; b = 4.0;
c = Math.sqrt(a*a + b*b);
System.out.println("Hypotenuse is " + c);
```

## 5. DATA ABSTRACTION

### 5.1 Basics of Data Abstraction:

- Abstraction is a process of hiding the implementation details and showing only functionality to the user.
- It shows only important things to the user and hides the internal details.
- Example: sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.
- Abstraction lets you focus on what the object does instead of how it does it.
- There are two ways to achieve abstraction in java
  1. Abstract class
  2. Interface

### 5.2 Abstract Class:

- **Abstract Method:** An abstract method is a method with only signature (i.e., the method name, the list of arguments and the return type) without implementation (i.e., the method's body). You use the keyword abstract to declare an abstract method.

  **Example:** `abstract void printStatus();`//no body and abstract
- An abstract method cannot be declared final, as final method cannot be overridden. An abstract method, on the other hand, must be overridden in a descendant before it can be used.
- An abstract method cannot be private (which generates a compilation error). This is because private method are not visible to the subclass and thus cannot be overridden.
- Any class that contains one or more abstract methods must be declared abstract.
- A class that is declared as abstract is known as **abstract class**. To declare a class abstract, we simply use the abstract keyword in front of the class keyword at the beginning of the class declaration.

  **Example:** `abstract class A{}`
- It needs to be extended and its method implemented.
- It cannot be instantiated. There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the new operator. Such objects would be useless, because an abstract class is not fully defined.
- We cannot declare abstract constructors, 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.

**Example:**

```
abstract class A
{
    abstract void callme();
    void callmetoo()
    {
          System.out.println("This is a concrete method.");
    }
}
class B extends A
{
    void callme()
```

```
    {
        System.out.println("B's implementation of callme.");
    }
}
class AbstractDemo
{
    public static void main(String args[])
    {
        B b = new B();
        b.callme();
        b.callmetoo();
    }
}
```

**Output**:

```
B's implementation of callme.
This is a concrete method.
```

- An abstract class provides *a template for further development*. The purpose of an abstract class is to provide a common interface (or protocol, or contract, or understanding, or naming convention) to all its subclasses.
- In Java abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java's approach to run-time polymorphism is implemented through the use of superclass references. Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object.

**Example:**

```
abstract class Figure
{
    double dim1, dim2;
    Figure(double a, double b) { dim1 = a; dim2 = b; }
    // area is now an abstract method
    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 AbstractAreas
    {
        public static void main(String args[])
        {
            // Figure f = new Figure(10, 10); // illegal now
            Rectangle r = new Rectangle(9, 5);
            Triangle t = new Triangle(10, 8);
            Figure figref; // this is OK, no object is created
            figref = r;
            System.out.println("Area is " + figref.area());
            figref = t;
            System.out.println("Area is " + figref.area());
        }
    }
```

**Output**:

```
Inside Area for Rectangle.
Area is 45.0
Inside Area for Triangle.
Area is 40.0
```

- An abstract class can contain constructors in Java. And a constructor of abstract class is called when an instance of an inherited class is created.

**Example:**

```
abstract class Base
{
    Base() { System.out.println("Base Constructor Called"); }
    abstract void fun();
}
class Derived extends Base
{
    Derived() { System.out.println("Derived Constructor Called"); }
    void fun() { System.out.println("Derived fun() called"); }
}
class Main
{
    public static void main(String args[])
    {
        Derived d = new Derived();
    }
}
```

**Output**:

```
Base Constructor Called
Derived Constructor Called
```

- We can have an abstract class without any abstract method. This allows us to create classes that cannot be instantiated, but can only be inherited.

**Example:**

```
abstract class Base
{
    void fun() { System.out.println("Base fun called"); }
}
class Derived extends Base { }
class Main
{
   public static void main(String args[])
   {
        Derived d = new Derived();
        d.fun();
    }
}
```

**Output**:

```
Base fun called
```

- Abstract classes can also have final methods (methods that cannot be overridden). For example, the following program compiles and runs fine.

**Example:**

```
abstract class Base
{
    final void fun() { System.out.println("Derived fun called"); }
}
class Derived extends Base {}
class Main
{
   public static void main(String args[])
   {
       Base b = new Derived();
       b.fun();
   }
}
```

**Output**:

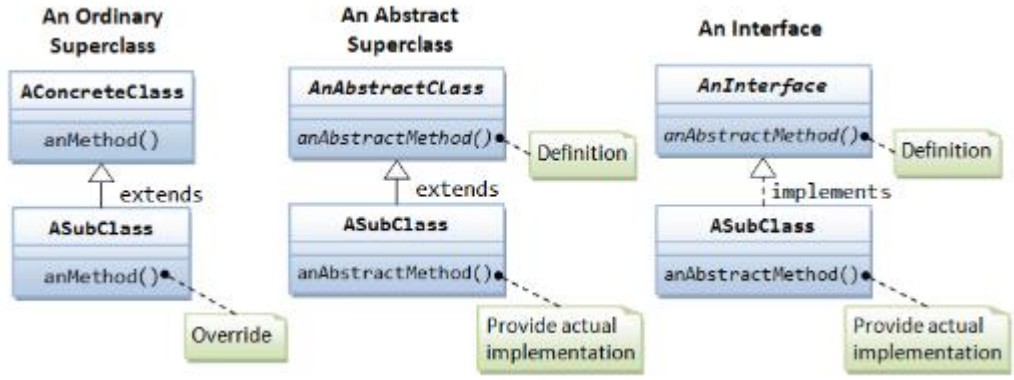
```
Derived fun called
```

## 6. INTERFACES

### 6.1 Understanding Interfaces:

- A Java interface is a 100% abstract superclass which define a set of methods its subclasses must support.
- An interface contains only public abstract methods (methods with signature and no implementation) and possibly constants (public static final variables).
- The interface in java is a mechanism to achieve abstraction. There can be only abstract methods in the java interface not method body.
- Interfaces specify what a class must do and not how. An interface is a contract for what the classes can do. It, however, does not specify how the classes should do it.
- An interface provides a form, a protocol, a standard, a contract, a specification, a set of rules, an interface, for all objects that implement it.

- It is used to achieve abstraction and multiple inheritances in Java.
- Java Interface also represents IS-A relationship.
- It cannot be instantiated just like abstract class.
- You have to use the keyword "interface" to define an interface (instead of keyword "class" for normal classes).
- All methods in an interface shall be public and abstract (default). You cannot use other access modifier such as private, protected and default, or modifiers such as static, final.
- All fields shall be public, static and final (default).




- Interface Naming Convention: Use an adjective (typically ends with "able") consisting of one or more words. Each word shall be initial capitalized (camel-case). For example, Serializable, Extenalizable, Movable, Clonable, Runnable, etc.

**6.2 Why do we use interface?**

- It is used to achieve total abstraction.
- Since java does not support multiple inheritance in case of class, but by using interface it can achieve multiple inheritance.
  - Java supports only single inheritance. Java does not support multiple inheritance to avoid inheriting conflicting properties from multiple superclasses.
  - A subclass, however, can implement more than one interfaces. This is permitted in Java as an interface merely defines the abstract methods without the actual implementations and less likely leads to inheriting conflicting properties from multiple interfaces.
- An interface is a contract (or a protocol, or a common understanding) of what the classes can do. When a class implements a certain interface, it promises to provide implementation to all the abstract methods declared in the interface. Interface defines a set of common behaviors. The classes implement the interface agree to these behaviors and provide their own implementation to the behaviors. This allows you to program at the interface, instead of the actual implementation. One of the main usage of interface is provide a communication contract between two objects. If you know a class implements an interface, then you know that class contains concrete implementations of the methods declared in that interface, and you are guaranteed to be able to invoke these methods safely. In other words, two objects can communicate based on the contract defined in the interface, instead of their specific implementation.
- It is also used to achieve loose coupling.
- Interfaces are used to implement abstraction. So the question is why do we use interfaces when we have abstract classes?
  - The reason is, abstract classes may contain non-final variables, whereas variables in interface are final, public and static.
- Interfaces are designed to support dynamic method resolution at run time.

**6.3 Defining an Interface?**

- The general form of an interface:

```
access-specifier interface <interface_name>
{
```

```
        return-type method-name1(parameter-list);
        return-type method-name2(parameter-list);
        type final-varname1 = value;
        type final-varname2 = value;
        // ...
        return-type method-nameN(parameter-list);
        type final-varnameN = value;
    }
```

- When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared.
- When it is declared as public, the interface can be used by any other code. In this case, the interface must be the only public interface declared in the file, and the file must have the same name as the interface.
- Each class that includes an interface must implement all of the methods.
- Variables can be declared inside of interface declarations.They are implicitly final and static, meaning they cannot be changed by the implementing class. They must also be initialized. All methods and variables are implicitly public.

## 6.4 Implementing Interfaces

- 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 looks like this:

```
class classname [extends superclass] [implements interface [,interface...]]
{
        // class-body
}
```

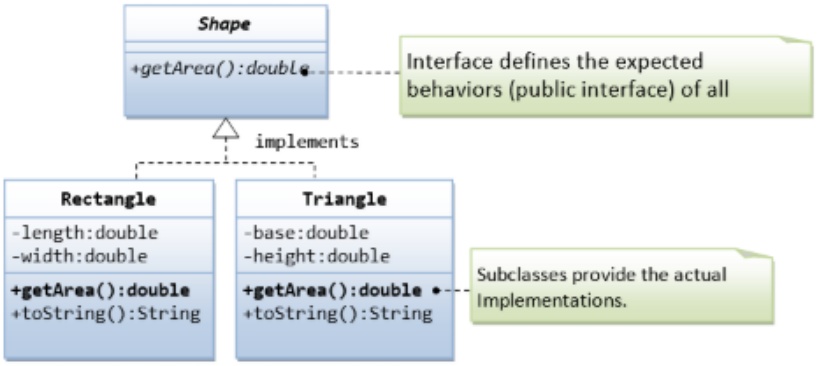
- If a class implements more than one interface, the interfaces are separated with a comma.
- If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface.
- The methods that implement an interface must be declared public. Also, the type signature of the implementing method must match exactly the type signature specified in the interface definition.

**Example 1:**

```
interface Callback
{
        void callback(int param);
}
class Client implements Callback
{
        // Implement Callback's interface
        public void callback(int p)
        {
                System.out.println("callback called with " + p);
        }
}
```

**NOTE:** When you implement an interface method, it must be declared as public.

**Example 2:**




```
public interface Shape
{
	double getArea();
}
public class Rectangle implements Shape
{
	private int length;
	private int width;
	public Rectangle(int length, int width)
	{
		this.length = length;
		this.width = width;
	}
	public String toString()
	{
	return "Rectangle[length=" + length + ",width=" + width + "]";
	}
	public double getArea()
	{
			return length * width;
	}
}
public class Triangle implements Shape
{
	private int base;
	private int height;

	public Triangle(int base, int height)
	{
			this.base = base;
			this.height = height;
	}
	public String toString()
	{
		return "Triangle[base=" + base + ",height=" + height + "]";
	}
	public double getArea()
	{
			return 0.5 * base * height;
	}
}
public class TestShape
{
	public static void main(String[] args)
	{
		Shape s;
		s = new Rectangle(1, 2);  // upcast
		System.out.println(s);
		System.out.println("Area is " + s.getArea());
		s = new Triangle(3, 4);  // upcast
		System.out.println(s);
		System.out.println("Area is " + s.getArea());
		//Shape s3 = new Shape("green");   // Compilation Error!!
   }
}
```

**Output:**

```
	Rectangle[length=1,width=2]
```

```
Area is 2.0
Triangle[base=3,height=4]
Area is 6.0
```

- It is both permissible and common for classes that implement interfaces to define additional members of their own.

**6.5 Accessing Implementations through Interface References**

- To implement an interface, include the *implements* clause in a class definition, and then create the methods defined by the interface.
- We can declare variables as object references that use an interface rather than a class type. Any instance of any class that implements the declared interface can be referred to by such a variable. When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to.
- The method to be executed is looked up dynamically at run time, allowing classes to be created later than the code which calls methods on them.

**Example:**

```
interface Callback
{
	void callback(int param);
}
class Client implements Callback
{
	// Implement Callback's interface
	public void callback(int p)
	{
		System.out.println("callback called with " + p);
	}
}
class TestIface
{
	public static void main(String args[])
	{
		Callback c = new Client();
		c.callback(42);
	}
}
```

**OUTPUT:**

callback called with 42

**NOTE:**

- Dynamic lookup of a method at run time incurs a significant overhead when compared with the normal method invocation in Java, we should be careful not to use interfaces casually in performance-critical code.
- In the above program although **'c'** is used to access the callback( ) method, it cannot access any other members of the Client class. An interface reference variable only has knowledge of the methods declared by its interface declaration.

- The following class demonstrate the polymorphic power of interface reference varibale.

**Example:**

```
class AnotherClient implements Callback
{
	// Implement Callback's interface
```

```
        public void callback(int p)
        {
            System.out.println("Another version of callback");
            System.out.println("p squared is " + (p*p));
        }
    }
    class TestIface2
    {
        public static void main(String args[])
        {
            Callback c = new Client();
            AnotherClient ob = new AnotherClient();
            c.callback(42);
            c = ob; // c now refers to AnotherClient object
            c.callback(42);
        }
    }
```

**OUTPUT**:

```
callback called with 42
Another version of callback
p squared is 1764
```

**NOTE:**

As you can see, the version of callback( ) that is called is determined by the type of object that c refers to at run time.

**6.6 Partial Implementations:** If a class includes an interface but does not fully implement the methods defined by that interface, then that class must be declared as abstract.

**Example:**

```
abstract class Incomplete implements Callback
{
    int a, b;
    void show()
    {
        System.out.println(a + " " + b);
    }
    // ...
}
```

**NOTE:**

Here, the class *Incomplete* does not implement callback( ) and must be declared as abstract. Any class that inherits the class *Incomplete* must implement callback( ) or be declared abstract itself.

**6.7 Nested Interfaces:**

- An interface can be declared a member of a class or another interface. Such an interface is called a member interface or a nested interface.
- A nested interface can be declared as public, private, or protected. This differs from a top-level interface, which must either be declared as public or use the default access level.
- When a nested interface is used outside of its enclosing scope, it must be qualified by the name of the class or interface of which it is a member.

**Example:**

```
class A
{
	public interface NestedIF
	{
		boolean isNotNegative(int x);
	}
}
// B implements the nested interface.
class B implements A.NestedIF
{
	public boolean isNotNegative(int x)
	{
		return x < 0 ? false : true;
	}
}
class NestedIFDemo
{
	public static void main(String args[])
	{
		// use a nested interface reference
		A.NestedIF nif = new B();
		if(nif.isNotNegative(10))
			System.out.println("10 is not negative");
		if(nif.isNotNegative(-12))
			System.out.println("this won't be displayed");
	}
}
```

**6.8 Applying Interfaces:** The following program shows the real power of interfaces. It implements stack both as fixed-size and growable.

**Example:**

```
// 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)
                 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--];
    }
}

class IFTest
{
     public static void main(String args[])
     {
            FixedStack mystack1 = new FixedStack(5);
            FixedStack mystack2 = new FixedStack(8);
            // push some numbers onto the stack
            for(int i=0; i<5; i++)
                  mystack1.push(i);
            for(int i=0; i<8; i++)
                  mystack2.push(i);
            // pop those numbers off the stack
            System.out.println("Stack in mystack1:");
            for(int i=0; i<5; i++)
                  System.out.println(mystack1.pop());

            System.out.println("Stack in mystack2:");
            for(int i=0; i<8; i++)
                  System.out.println(mystack2.pop());
      }
}
```

Following is another implementation of IntStack that creates a dynamic stack by use of the same interface definition. In this implementation, each stack is constructed with an initial length. If this initial length is exceeded, then the stack is increased in size. Each time more room is needed, the size of the stack is doubled.

```
class DynStack implements IntStack
{
      private int stck[];
      private int tos;
      // allocate and initialize stack
      DynStack(int size)
      {
            stck = new int[size];
            tos = -1;
      }

     // Push an item onto the stack
     public void push(int item)
     {
            // if stack is full, allocate a larger stack
            if(tos==stck.length-1)
            {
                  int temp[] = new int[stck.length * 2];
                  // double size
                  for(int i=0; i<stck.length; i++)
                        temp[i] = stck[i];
                  stck = temp;
                  stck[++tos] = item;
            }
            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--];
      }
}
class IFTest2
{
     public static void main(String args[])
     {
            DynStack mystack1 = new DynStack(5);
            DynStack mystack2 = new DynStack(8);

            // these loops cause each stack to grow
            for(int i=0; i<12; i++)
```

```
                mystack1.push(i);
            for(int i=0; i<20; i++)
                mystack2.push(i);

            System.out.println("Stack in mystack1:");
            for(int i=0; i<12; i++)
                System.out.println(mystack1.pop());

            System.out.println("Stack in mystack2:");
            for(int i=0; i<20; i++)
                System.out.println(mystack2.pop());
        }
}
```

The following class uses both the FixedStack and DynStack implementations. It does so through an interface reference. This means that calls to push( ) and pop( ) are resolved at run time rather than at compile time.

```
class IFTest3
{
    public static void main(String args[])
    {
            IntStack mystack;
            DynStack ds = new DynStack(5);
            FixedStack fs = new FixedStack(8);
            mystack = ds; // load dynamic stack
            // push some numbers onto the stack
            for(int i=0; i<12; i++)
                mystack.push(i);
            mystack = fs; // load fixed stack
            for(int i=0; i<8; i++)
                mystack.push(i);
            mystack = ds;
            System.out.println("Values in dynamic stack:");
            for(int i=0; i<12; i++)
                System.out.println(mystack.pop());
            mystack = fs;
            System.out.println("Values in fixed stack:");
            for(int i=0; i<8; i++)
                System.out.println(mystack.pop());
        }
}
```

**6.9 Variables in Interfaces:** We can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values. When you include that interface in a class, all of those variable names will be in scope as constants.

**Example:**

```
import java.util.Random;
interface SharedConstants
{
        int NO = 0;
```

```
        int YES = 1;
        int MAYBE = 2;
        int LATER = 3;
        int SOON = 4;
        int NEVER = 5;
}
class Question implements SharedConstants
{
        Random rand = new Random();
        int ask()
        {
                int prob = (int) (100 * rand.nextDouble());
                if (prob < 30)
                        return NO;           // 30%
                else if (prob < 60)
                        return YES;          // 30%
                else if (prob < 75)
                        return LATER;        // 15%
                else if (prob < 98)
                        return SOON;         // 13%
                else
                        return NEVER;        // 2%
        }
}
class AskMe implements SharedConstants
{
        static void answer(int result)
        {
                switch(result)
                {
                        case NO:
                                System.out.println("No");
                                break;
                        case YES:
                                System.out.println("Yes");
                                break;
                        case MAYBE:
                                System.out.println("Maybe");
                                break;
                        case LATER:
                                System.out.println("Later");
                                break;
                        case SOON:
                                System.out.println("Soon");
                                break;
                        case NEVER:
                                System.out.println("Never");
                                break;
                }
```

```
        }
        public static void main(String args[])
        {
                Question q = new Question();
                answer(q.ask());
                answer(q.ask());
                answer(q.ask());
                answer(q.ask());
        }
}
```

**6.10 Extending Interfaces:** One interface can inherit another by using the keyword extends. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

**Example:**

```
interface A
{
        void meth1();
        void meth2();
}
interface B extends A
{
        void meth3();
}

class MyClass implements B
{
        public void meth1()
        {
                System.out.println("Implement meth1().");
        }
        public void meth2()
        {
                System.out.println("Implement meth2().");
        }
        public void meth3()
        {
                System.out.println("Implement meth3().");
        }
}
class IFExtend
{
        public static void main(String arg[])
        {
                MyClass ob = new MyClass();
                ob.meth1();
                ob.meth2();
                ob.meth3();
        }
}
```

**6.11 Default Interface Methods:** Before Java 8, interfaces could have only abstract methods. The implementation of these methods has to be provided in a separate class. So, if a new method is to be added in an interface then its implementation code has to be provided in the class implementing the same interface. To overcome this issue, Java 8 has introduced the concept of default methods which allow the interfaces to have methods with implementation without affecting the classes that implement the interface.

**Example:**

```
interface TestInterface
{
    // abstract method
    public void square(int a);

    // default method
    default void show()
    {
      System.out.println("Default Method Executed");
    }
}

class TestClass implements TestInterface
{
    // implementation of square abstract method
    public void square(int a)
    {
        System.out.println(a*a);
    }

    public static void main(String args[])
    {
        TestClass d = new TestClass();
        d.square(4);
        d.show();// default method executed
    }
}
```

**6.12 Static methods in an Interface:** JDK 8 added another new capability to interface: the ability to define one or more static methods. Like static methods in a class, a static method defined by an interface can be called independently of an object.

**Syntax of calling a static method:** *InterfaceName.staticMethodName*

**Example:**

```
interface TestInterface
{
    // abstract method
    public void square (int a);

    // static method
    static void show()
    {
        System.out.println("Static Method Executed");
```

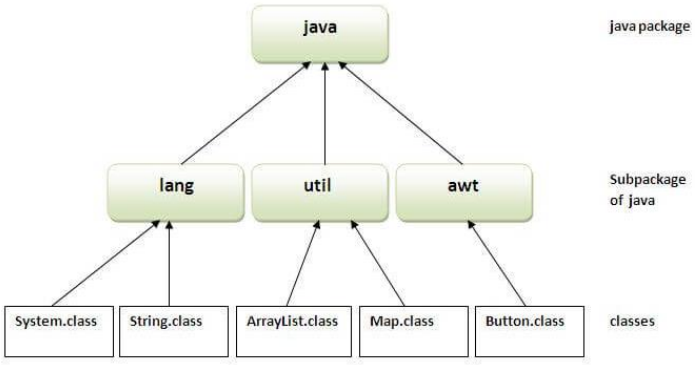
```
    }
}

class TestClass implements TestInterface
{
      // Implementation of square abstract method
      public void square (int a)
      {
            System.out.println(a*a);
      }
      public static void main(String args[])
      {
            TestClass d = new TestClass();
            d.square(4);
            // Static method executed
            TestInterface.show();
      }
}
```

## 7. PACKAGES

### 7.1 Introduction to Packages:

- A java package is a group of similar types of classes, interfaces and sub-packages.
- Package in Java is a mechanism to encapsulate a group of classes, sub packages and interfaces.
- Packages are used for:
  - Preventing naming conflicts. For example there can be two classes with same name in two packages.
  - Making searching/locating and usage of classes, interfaces will be easier
  - **Providing controlled access:** protected and default have package level access control. A protected member is accessible by classes in the same package and its subclasses. A default member (without any access specifier) is accessible by classes in the same package only.
  - Packages can be considered as data encapsulation (or data-hiding).




- We should put related classes into packages.
- A package is container of group of related classes where some of the classes are accessible are exposed and others are kept for internal purpose.

### 7.2 Defining Packages:

- To create a package, include a *package* command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package.
- The package statement defines a name space in which classes are stored. If you omit the package statement, the class names are put into the default package, which has no name.
- This is the general form of the package statement:
  **Syntax:** package *pkgName*;
  **Example:** package MyPackage;
- Java uses file system directories to store packages. For example, the .class files for any classes you declare to be part of MyPackage must be stored in a directory called MyPackage.

- More than one file can include the same package statement. The package statement simply specifies to which package the classes defined in a file belong.
- We can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here:

  package pkg1[.pkg2[.pkg3]];

**7.3 Finding Packages and CLASSPATH**:

- How does the Java run-time system know where to look for packages that you create?
- Answer:
  - First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found.
  - Second, you can specify a directory path or paths by setting the CLASSPATH environmental variable.
  - Third, you can use the -classpath option with java and javac to specify the path to your classes.
- **Easy Way to Run:** The easiest way to run a program having package is to simply create the package directories below your current development directory, put the .class files into the appropriate directories, and then execute the programs from the development directory.

**Example:**

```
// A simple package
package MyPack;
class Balance
{
   String name; double bal;
   Balance(String n, double b) { name = n; bal = b; }
   void show()
   {
         if(bal<0)
               System.out.print("--> ");
         System.out.println(name + ": $" + bal);
   }
}
class AccountBalance
{
   public static void main(String args[])
   {
         Balance current[] = new Balance[3];
         current[0] = new Balance("K. J. Fielding", 123.23);
         current[1] = new Balance("Will Tell", 157.02);
         current[2] = new Balance("Tom Jackson", -12.33);
         for(int i=0; i<3; i++)
               current[i].show();
   }
}
```

**NOTE:**

Compile the program under the current directory in normal way. Create a directory with the name MyPack. Keep the .class files (both Balance.class and AccountBalance.class) of this program under the folder MyPack. Run the

program using command java MyPack.AccountBalance by staying just above MyPack folder.

**Output**:

```
K. J. Fielding: $123.23
Will Tell: $157.02
--> Tom Jackson: $-12.33
```

### 7.4 Types of Packages:

- **Built-in Packages:** These packages consist of a large number of classes which are a part of Java **API**. Some of the commonly used built-in packages are:
  1) **java.lang:** Contains language support classes (e.g classed which defines primitive data types, math operations). This package is automatically imported.
  2) **java.io:** Contains classed for supporting input / output operations.
  3) **java.util:** Contains utility classes which implement data structures like Linked List, Dictionary and support for Date / Time operations.
  4) **java.applet:** Contains classes for creating Applets.
  5) **java.awt:** Contain classes for implementing the components for graphical user interfaces (like button, menus etc).
  6) **java.net:** Contain classes for supporting networking operations.
- **User-definedPackages:**
  - These are the packages that are defined by the user. First we create a directory **myPackage** (name should be same as the name of the package). Then create the **MyClass** inside the directory with the first statement being the **package name**.

**Example:**

```
// Name of the package must be same as the directory
// under which this file is saved
package myPackage;
public class MyClass
{
    public void getNames(String s)
    {
        System.out.println(s);
    }
}
Now we can use the MyClass class in our program.
/* import 'MyClass' class from 'names' myPackage */
import myPackage.MyClass;
public class PrintName
{
   public static void main(String args[])
   {
      // Initializing the String variable with a value
      String name = "GeeksforGeeks";
      // Creating an instance of class MyClass in the package.
      MyClass obj = new MyClass();
      obj.getNames(name);
   }
}
```

**NOTE: MyClass.java** must be saved inside the **myPackage** directory since it is a part of the package.

### 7.5 Access Protection:

- Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code.
- Java addresses four categories of visibility for class members:
  - Subclasses in the same package
  - Non-subclasses in the same package
  - Subclasses in different packages
  - Classes that are neither in the same package nor subclasses
- The three access specifiers, private, public, and protected, provide a variety of ways to produce the many levels of access required by these categories. The following table sums up the interactions:

| | Private | No modifies | Protected | Public |
|---|---|---|---|---|
| Same Class | **Yes** | **Yes** | **Yes** | **Yes** |
| Same Package Subclass | **No** | **Yes** | **Yes** | **Yes** |
| Same Package non-subclass | **No** | **Yes** | **Yes** | **Yes** |
| Different package subclass | **No** | **No** | **Yes** | **Yes** |
| Different package non-subclass | **No** | **No** | **No** | **Yes** |

- Anything declared public can be accessed from anywhere.
- Anything declared private cannot be seen outside of its class.
- When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the default access.
- If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element protected.

**Example:**

```
//Save the following code in Protection.java
package p1;
public class Protection
{
   int n = 1;
   private int n_pri = 2;
   protected int n_pro = 3;
   public int n_pub = 4;
   public Protection()
   {
      System.out.println("base constructor");
      System.out.println("n = " + n);
      System.out.println("n_pri = " + n_pri);
      System.out.println("n_pro = " + n_pro);
      System.out.println("n_pub = " + n_pub); }
}
//Save the following code in Derived.java:
package p1;
class Derived extends Protection
```

```
{
   Derived()
   {
         System.out.println("derived constructor");
         System.out.println("n = " + n);
         //  class only
         //  System.out.println("n_pri = "4 + n_pri);
         System.out.println("n_pro = " + n_pro);
         System.out.println("n_pub = " + n_pub);
   }
}
//Save the following code in SamePackage.java:
package p1;
class SamePackage
{
   SamePackage()
   {
         Protection p = new Protection();
         System.out.println("same package constructor");
         System.out.println("n = " + p.n);
         //  class only
         //  System.out.println("n_pri = " + p.n_pri);
         System.out.println("n_pro = " + p.n_pro);
         System.out.println("n_pub = " + p.n_pub);
   }
}
//Save the following code in Protection2.java:
package p2;
class Protection2 extends p1.Protection
{
   Protection2()
   {
         System.out.println("derived other package constructor");
         //  class or package only
         //  System.out.println("n = " + n);
         //  class only
         //  System.out.println("n_pri = " + n_pri);
         System.out.println("n_pro = " + n_pro);
         System.out.println("n_pub = " + n_pub);
   }
}
Save the following code in OtherPackage.java:
package p2;
class OtherPackage
{
   OtherPackage()
   {
      p1.Protection p = new p1.Protection();
      System.out.println("other package constructor");
```

```
        //  class or package only
        //  System.out.println("n = " + p.n);
        //  class only
        //  System.out.println("n_pri = " + p.n_pri);
        //  class, subclass or package only
        //  System.out.println("n_pro = " + p.n_pro);
        System.out.println("n_pub = " + p.n_pub);
    }
}
```

To test the above two packages we use the following two programs for each package.

**Save the following code in Demo.java:**

```
// Demo package p1.
package p1;
// Instantiate the various classes in p1.
public class Demo
{
    public static void main(String args[])
    {
        Protection ob1 = new Protection();
        Derived ob2 = new Derived();
        SamePackage ob3 = new SamePackage();
    }
}
```

**Output**:

```
***************
```

**Save the following code in Demo.java:**

```
// Demo package p2.
package p2;
// Instantiate the various classes in p2.
public class Demo
{
    public static void main(String args[])
    {
        Protection2 ob1 = new Protection2();
        OtherPackage ob2 = new OtherPackage();
    }
}
```

**Output**:

```
****************
```

### 7.6 Importing Packages:

- To use a package in Java we need to import that by using ***import*** statement.
- In C/C++ if we include a header file in a program, during compilation the #include directive makes the compiler go to the standard library and copies the entire header file contents into program. As a result the program size increases unnecessarily wasting memory and processor's time. When a class name or a method name is used in a Java, the import statement makes the JVM goes to the Java standard library, executes the code there, comes back and substitutes the result in that place of the program. Thus, the Java program size is not increased.

- In a Java source file, import statements occur immediately following the package statement (if it exists and before any class definitions.
- Syntax:

  import pkg1[.pkg2].(classname|*);
- Example:

  import java.util.Date;
  import java.io.*;
- **CAUTION:** The star form may increase compilation time—especially if you import several large packages. However, the star form has absolutely no effect on the run-time performance or size of your classes.
- The import statement is optional. Any place you use a class name, you can use its fully qualified name, which includes its full package hierarchy:

  **Example:**

  import java.util.*;
  class MyDate extends Date { }
  **OR**
  class MyDate extends java.util.Date { }
- When a package is imported, only those items within the package declared as public will be available to non-subclasses in the importing code.

  **Example:**

```
package MyPack;
Public class Balance
{
   String name;
   double bal;
   public Balance(String n, double b) { name = n; bal = b; }
   public void show()
   {
         if(bal<0)
               System.out.print("--> ");
         System.out.println(name + ": $" + bal);
   }
}
```

*Here the Balance class is now public. Also, its constructor and its show() method are public, too. This means that they can be accessed by any type of code outside the MyPack package. For example, here TestBalance imports MyPack and is then able to make use of the Balance class.*

```
import MyPack.*;
class TestBalance
{
   public static void main(String args[])
   {
      Balance test = new Balance("J. J. Jaspers", 99.88);
      test.show(); // you may also call show()
   }
}
```

- **Subpackages:** Packages that are inside another package are the **subpackages**. These are not imported by default, they have to import explicitly. Also, members of a subpackage have no access privileges, i.e., they are considered as different package for protected and default access specifiers.

  `import package.*;`// All the classes and interfaces of this package will be accessible but not subpackages.

**NOTE:**

1. Every class is part of some package.
2. If no package is specified, the classes in the file goes into a special unnamed package (the same unnamed package for all files).
3. All classes/interfaces in a file are part of the same package. Multiple files can specify the same package name.
4. If package name is specified, the file must be in a subdirectory called name (i.e., the directory name must match the package name).
5. We can access public classes in another (named) package using: package-name.class-name.

## 8. EXCEPTION HANDLING

### 8.1 Exception Handling Fundamentals:

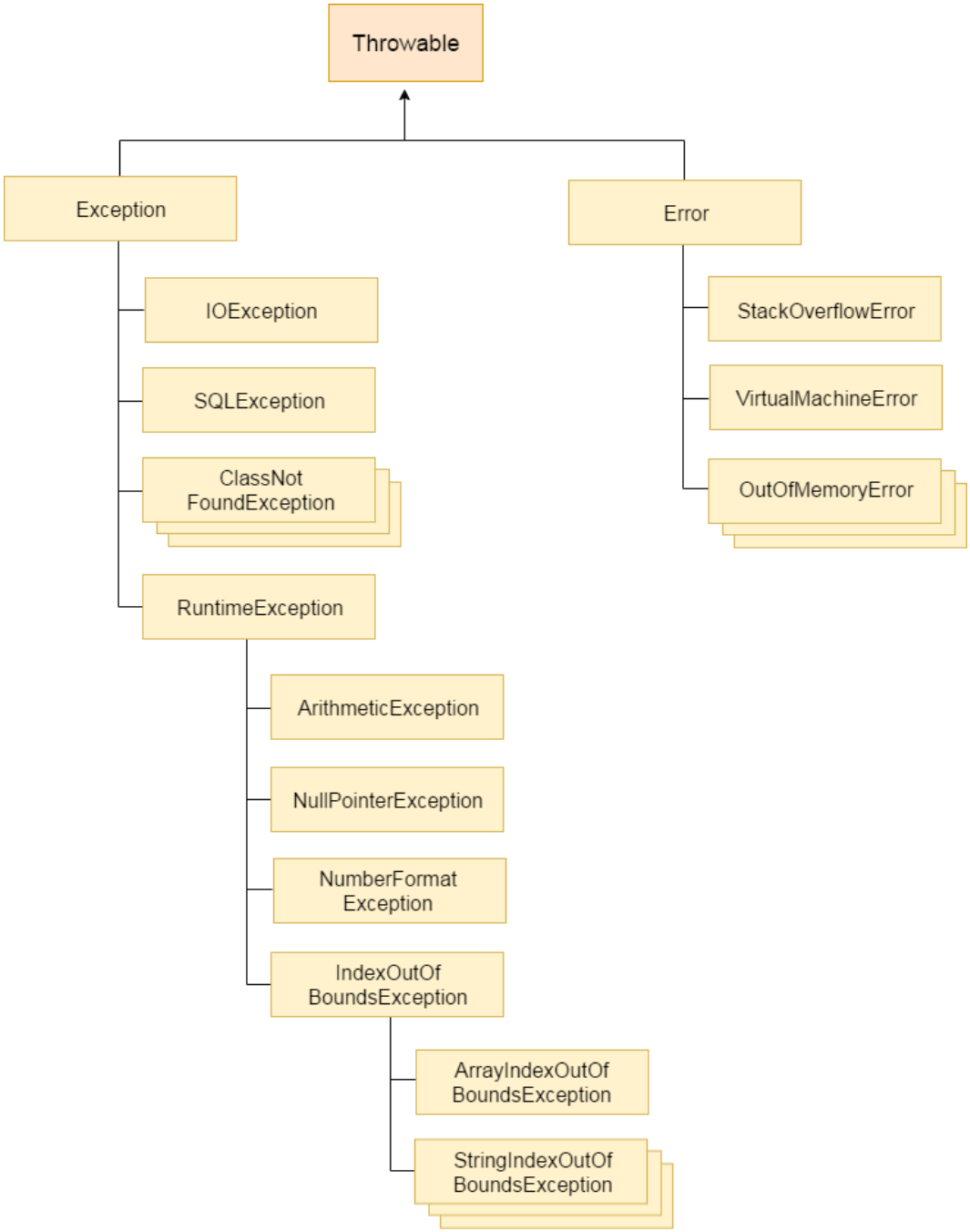
- An exception is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a run-time error.
- An exception is an unwanted or unexpected event, which occurs during the execution of a program i.e at run time, which disrupts the normal flow of the program's instructions.
- A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.
- In java, exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.
- Exception Handling is a mechanism to handle runtime errors such as ClassNotFound, IO, SQL, Remote etc.
- The core advantage of exception handling is **to maintain the normal flow of the application**. Exception normally disrupts the normal flow of the application that is why we use exception handling.
- **Example**: Suppose there are 5 statements in a program and there occurs an exception at statement number 3, rest of the code will not be executed i.e. statement 4, 5 will not run. If we perform exception handling, rest of the statement will be executed. That is why we use exception handling in java.
- Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**.
- Program statements that we want to monitor for exceptions are contained within a **try** block.
- If an exception occurs within the try block, it is thrown.
- Our code can catch this exception (using **catch**) and handle it in some rational manner.
- System-generated exceptions are automatically thrown by the Java run-time system.
- To manually throw an exception, use the keyword **throw**.
- Any exception that is thrown out of a method must be specified as such by a **throws** clause.
- Any code that absolutely must be executed after a try block completes is put in a **finally** block.
- The general form of an exception-handling block:

```
try
{
        // block of code to monitor for errors
}
catch (ExceptionType1 exOb)
{
        // exception handler for ExceptionType1
}
catch (ExceptionType2 exOb)
{
        // exception handler for ExceptionType2
}
// ...
finally
```

```
{
        // block of code to be executed after try block ends
}
```

**8.2 Exception Types**:

- All exception types are subclasses of the built-in class Throwable.
- Below Throwable there are two subclasses that partition exceptions into two distinct branches. One branch is headed by Exception.
- This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types.
- One of the important subclass of Exception, called RuntimeException. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing etc.
- The other branch (of Throwable) is topped by Error, which defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type Error are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error.
- In this chapter will not be dealing with exceptions of type Error.




**8.3 Uncaught Exception**:

- Let us understand the following code in which we will see what happens when we do not handle an exception.

```
class Exc0
{
      public static void main(String args[])
      {
            int d = 0;
            int a = 42 / d;
      }
}
```

- When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception.
- This causes the execution of Exc0 tostop, because once an exception has been thrown, it must be caught by an exception handler and dealt with immediately.
- As we have not supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system.

- Any exception that is not caught by your program will ultimately be processed by the default handler. The default handler displays a string describing the exception,prints a stack trace from the point at which the exception occurred, and terminates the program.
- Here is the exception generated when this example is executed:

```
java.lang.ArithmeticException: / by zero
      at Exc0.main(Exc0.java:4)
```

- The stack trace will always show the sequence of method invocations that led up to the error. Consider the following code.

```
class Exc1
{
      static void subroutine()
      {
            int d = 0;
            int a = 10 / d;
      }
      public static void main(String args[])
      {
            Exc1.subroutine();
      }
}
```

- The resulting stack trace from the default exception handler shows how the entire ***call stack*** is displayed. The call stack is useful for debugging. Let us mark the output of the above code:

```
java.lang.ArithmeticException: / by zero
at Exc1.subroutine(Exc1.java:4)
at Exc1.main(Exc1.java:7)
```

**8.4 Using try and catch**:

- The user should handle his/her exceptions because of two reasons:
  - it allows you to fix the error.
  - it prevents the program from automatically terminating.
- To understand how can we handle our exception let us see the following example:

```
class Exc2
{
 public static void main(String args[])
 {
      int d, a;
      try
      {     // monitor a block of code.
            d = 0;
            a = 42 / d;
            System.out.println("This will not be printed.");
      }
      catch (ArithmeticException e)
      {
            // catch divide-by-zero error
            System.out.println("Division by zero.");
      }
      System.out.println("After catch statement.");
 }
}
```

**OUTPUT**:

```
Division by zero.
After catch statement.
```

- Enclose the code that you want to monitor inside a try block.
- Immediately following the try block, include a catch clause that specifies the exception type that you wish to catch.
- Once an exception is thrown, program control transfers out of the try block into the catch block.
- Once the catch statement has executed, program control continues with the next line in the program following the entire try/catch mechanism.
- Atry and its catch statement form a unit.
- A catch statement cannot catch an exception thrown by another try statement (except in the case of nested try statements).

- The statements that are protected by try must be surrounded by curly braces.
- You cannot use try on a single statement.
- The goal of most well-constructed catch clauses should be to resolve the exceptionalcondition and then continue on as if the error had never happened.
- **Example:**

```
import java.util.Random;
class HandleError
{
      public static void main(String args[])
      {
            int a=0, b=0, c=0;
            Random r = new Random();
            for(int i=0; i<32000; i++)
            {
                  try
                  {
                        b = r.nextInt();
                        c = r.nextInt();
                        a = 12345 / (b/c);
                  }
                  catch (ArithmeticException e)
                  {
                        System.out.println("Division by zero.");
                        a = 0; // set a to zero and continue
                  }
                  System.out.println("a: " + a);
            }
      }
}
```

**8.5 Displaying a Description of an Exception:**

- Throwable overrides the toString( ) method (defined by Object) so that it returns a string containing a description of the exception.
- We can display this description in a println( ) statement by simply passing the exception as an argument.

```
catch (ArithmeticException e)
{
   System.out.println("Exception: " + e);
   a = 0; // set a to zero and continue
 }
```

- Display a description of an exception is valuable in the circumstances when we are experimenting with exceptions or when we are debugging.

**8.6 Multiple catch Clauses:**

- In some cases, more than one exception could be raised by a single piece of code.
- To handle this type of situation, we can specify two or more catch clauses, each catching a different type of exception.
- When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed.
- After one catch statement executes, the others are bypassed, and execution continues after the try/catch block.

```
// Multiple catch statements.
Class MultiCatch
{
 public static void main(String args[])
 {
      try
      {
            int a = args.length;
            System.out.println("a = " + a);
            int b = 42 / a;
            int c[] = {1};
            c[42] = 99;
      }
      catch(ArithmeticException e)
      {
```

```
                System.out.println("Divide by 0: " + e);
            }
            catch(ArrayIndexOutOfBoundsException e)
            {
                System.out.println("Array index oob: " + e);
            }
            System.out.println("After try/catch blocks.");
        }
    }
```

**OUTPUT**:

(2) D:\>java MultiCatch

a = 0

Divide by 0: java.lang.ArithmeticException: / by zero

After try/catch blocks.

(2) D:\>java MultiCatch TestArg

a = 1

Array index oob: java.lang.ArrayIndexOutOfBoundsException:42

After try/catch blocks.

- When you use multiple catch statements, it is important to remember that exception subclasses must come before any of their superclasses. Otherwise a catch statement that uses a superclass will catch exceptions of that type plus any of its subclasses.
- Thus, a subclass would never be reached if it came after its superclass. In that case Java will show error since unreachable code is an error in Java.

```
Class SuperSubCatch
{
    public static void main(String args[])
    {
        try
        {
            int a = 0;
            int b = 42/a;
        }
        catch(Exception e)
        {
            System.out.println("Generic Exception catch.");
        }
        /* This catch is never reached becauseArithmeticException
            is a subclass of Exception. */
        catch(ArithmeticException e) // ERROR - unreachable
        {
            System.out.println("This is never reached.");
        }
    }
}
```

**8.7 Nested try Statements**:

- A try statement can be nested inside the block of another try.
- Each time a try statement is entered, the context of that exception is pushed on the stack. If an inner try statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement's catch handlers are inspected for a match. This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted.
- If no catch statement matches, then the Java run-time system will handle the exception.

```
Class NestTry
{
  public static void main(String args[])
  {
        try
        {
            int a = args.length;
            int b = 42 / a;
            System.out.println("a = " + a);
            try
            {
                if(a==1)
```

```
                        a = a/(a-a);
                    if(a==2)
                    {
                        int c[] = { 1 };
                        c[42] = 99;
                    }
                }
                catch(ArrayIndexOutOfBoundsException e)
                {
                    System.out.println("Array index out-of-bounds:"+ e);
                }
            }
            catch(ArithmeticException e)
            {
                System.out.println("Divide by 0: " + e);
            }
        }
}
```

**OUTPUT**:

(3) D:\>java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero

(2) D:\>java NestTry One
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero

(4) D:\>java NestTry One Two
a = 2
Array index out-of-bounds:
java.lang.ArrayIndexOutOfBoundsException:42

- We can enclose a call to a method within a try block. Inside that method is another try statement. In this case, the try within the method is still nested inside the outer try block, which calls the method.
- We can write the previous program in the following way:

```
class MethNestTry
{
    static void nesttry(int a)
    {
        try
        {
            if(a==1)
                a = a/(a-a);
            if(a==2)
            {
                int c[] = {1};
                c[42] = 99;
            }
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("Array index out-of-bounds:"+ e);
        }
    }
    public static void main(String args[])
    {
        try
        {
            int a = args.length;
            int b = 42 / a;
            System.out.println("a = " + a);
            nesttry(a);
        }
        catch(ArithmeticException e)
        {
            System.out.println("Divide by 0: " + e);
        }
    }
}
```

**8.8 throw**:

- We can throw an exception explicitly, using the throw statement.
- The general form of throw is shown here:

```
throw ThrowableInstance;
```

- Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable.
- Primitive types, such as int or char, as well as non-Throwable classes, such as String and Object, cannot be used as exceptions.
- There are two ways you can obtain a Throwable object:
  - using a parameter in a catch clause, or
  - creating one with the new operator
- The flow of execution stops immediately after the throw statement; any subsequent statements are not executed.
- The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of exception.
- If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on.
- If no matching catch is found, then the default exception handler halts the program and prints the stack trace.

```
Class ThrowDemo
{
        static void demoproc()
        {
              try
              {
                    throw new NullPointerException("demo");
              }
              catch(NullPointerException e)
              {
                    System.out.println("Caught inside demoproc.");
                    throw e; // rethrow the exception
              }
        }
        public static void main(String args[])
        {
              try
              {
                    demoproc();
              }
              catch(NullPointerException e)
              {
                    System.out.println("Recaught: " + e);
              }
        }
}
```

**OUTPUT**:

Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo

- In the above program new is used to construct an instance of *NullPointerException*.
- While creating the instance it is calling the constructor by sending a string as parameter which describes the exception.
- This string is displayed when the object is used as an argument to print( ) or println( ). It can also be obtained by a call to getMessage( ), which is defined by Throwable.

**8.9 throws**:

- If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception.
- We can do this by including a *throws* clause in the method's declaration.
- A *throws* clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses. All

other exceptions that a method can throw must be declared in the throws clause. If they are not, a compile-time error will result.

- The general form of a method declaration that includes a throws clause:

```
type method-name(parameter-list) throws exception-list
{
        // body of method
}
```

- The following is an incorrect program that tries to throw an exception that it does not catch. Because the program does not specify a throws clause to declare this fact, the program will not compile.

```
class ThrowsDemo
{
 static void throwOne()
 {
        System.out.println("Inside throwOne.");
        throw new IllegalAccessException("demo");
 }
 public static void main(String args[])
 {
        throwOne();
 }
}
```

- To make this example compile, we need to make the following two changes:
  - declare that throwOne( ) throws IllegalAccessException.
  - main( ) must define a try/catch statement that catches this exception.
- The corrected code is as follows:

```
class ThrowsDemo
{
      static void throwOne() throws IllegalAccessException
      {
            System.out.println("Inside throwOne.");
            throw new IllegalAccessException("demo");
      }
      public static void main(String args[])
      {
            try
            {
                  throwOne();
            }
            catch (IllegalAccessException e)
            {
                  System.out.println("Caught " + e);
            }
      }
}
```

**OUTPUT**:

inside throwOne
caught java.lang.IllegalAccessException: demo

**8.10 Finally**:

- The keyword *finally* creates a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block.
- The finally block will execute whether or not an exception is thrown.
- If an exception is thrown, the finally block will execute even if no catch statement matches the exception.
- Any time a method is about to return to the caller from inside a try/catch block, via an uncaught exception or an explicit return statement, the finally clause is also executed just before the method returns.
- The finally clause is optional. However, each try statement requires at least one catch or a finally clause.
- If a finally block is associated with a try, the finally block will be executed upon conclusion of the try.

```
Class FinallyDemo
```

```
{
	static void procA()
	{
		try
		{
			System.out.println("inside procA");
			throw new RuntimeException("demo");
		}
		finally
		{
			System.out.println("procA's finally");
		}
}
static void procB()
{
	try
	{
		System.out.println("inside procB");
		return;
	}
	finally
	{
		System.out.println("procB's finally");
	}
}
static void procC()
{
	try
	{
		System.out.println("inside procC");
	}
	finally
	{
		System.out.println("procC's finally");
	}
}
public static void main(String args[])
{
	try
	{
		procA();
	}
	catch (Exception e)
	{
		System.out.println("Exception caught");
	}
	procB();
	procC();
}
}
```

**OUTPUT**:

inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally

**8.11Summary of Java Exception Keywords**:

| Keyword | Description |
|---|---|
| try | The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone. |

| | |
|---|---|
| Catch | The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later. |
| Finally | The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not. |
| Throw | The "throw" keyword is used to throw an exception. |
| Throws | The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature. |

**8.12Java's Built-in Exceptions**:

1. **Checked Exception**:
   - Checked exception are the exceptions that are checked at compile time.
   - The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc.
   - If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using throws keyword.
   - **Example**: The following program opens a file and prints first three lines of it. The program doesn't compile, because the function main() uses FileReader() method which throws a checked exception FileNotFoundException. It also uses readLine() and close() methods, and these methods also throw checked exception IOException.

```
Importjava.io.*;
class Main
{
      Public static void main(String[] args)
      {
        FileReader file = new FileReader("a.txt");
        BufferedReader fileInput = new BufferedReader(file);

        // Print first 3 lines of file "a.txt"
        for(int counter = 0; counter < 3; counter++)
            System.out.println(fileInput.readLine());

        fileInput.close();
      }
}
```

   - To fix the above program, we either need to specify list of exceptions using throws, or we need to use try-catch block.

```
Importjava.io.*;

class Main
{
      Public static void main(String[] args) throwsIOException
      {
        FileReader file = new FileReader("a.txt");
        BufferedReader fileInput = new BufferedReader(file);

        // Print first 3 lines of file "a.txt"
        for(int counter = 0; counter < 3; counter++)
            System.out.println(fileInput.readLine());

        fileInput.close();
    }
}
```

   - The checked exceptions defined in **java.lang** are listed in the following table:

| Exception | Meaning |
|---|---|
| ClassNotFoundException | Class not found. |

| | |
|---|---|
| CloneNotSupportedException | Attempt to clone an object that does not implement the Cloneable interface. |
| IllegalAccessException | Access to a class is denied. |
| InstantiationException | Attempt to create an object of an abstract class or interface. |
| InterruptedException | One thread has been interrupted by another thread. |
| NoSuchFieldException | A requested field does not exist. |
| NoSuchMethodException | A requested method does not exist. |

### 8.13 Unchecked Exception:

- These exceptions are not checked at compiled time but they are checked at runtime.
- The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.
- In Java exceptions under Error and RuntimeException classes are unchecked exceptions, everything else under Throwable is checked.
- Consider the following Java program. It compiles fine, but it throws ArithmeticException when run. The compiler allows it to compile, because ArithmeticException is an unchecked exception.

```
class Main
{
      public static void main(String args[])
      {
      int x = 0;
      int y = 10;
      int z = y/x;
      }
}
```

- The unchecked exceptions defined in **java.lang** are listed in the following table:

| Exception | Meaning |
|---|---|
| ArithmeticException | Arithmetic error, such as divide-by-zero. |
| ArrayIndexOutOfBoundsException | Array index is out-of-bounds. |
| ArrayStoreException | Assignment to an array element of an incompatible type. |
| ClassCastException | Invalid cast. |
| EnumConstantNotPresentException | An attempt is made to use an undefined enumeration value. |
| IllegalArgumentException | Illegal argument used to invoke a method. |
| IllegalMonitorStateException | Illegal monitor operation, such as waiting on an unlocked thread. |
| IllegalStateException | Environment or application is in incorrect state. |
| IllegalThreadStateException | Requested operation not compatible with current thread state. |
| IndexOutOfBoundsException | Some type of index is out-of-bounds. |
| NegativeArraySizeException | Array created with a negative size. |
| NullPointerException | Invalid use of a null reference. |
| NumberFormatException | Invalid conversion of a string to a numeric format. |
| SecurityException | Attempt to violate security. |
| StringIndexOutOfBounds | Attempt to index outside the bounds of a string. |
| TypeNotPresentException | Type not found. |
| UnsupportedOperationException | An unsupported operation was encountered. |

### 8.14 Creating Your own Exception Subclasses:

- If we want to create our own exception types to handle situations specific to our applications, we need to just define a subclass of Exception (which is, of course, a subclass of Throwable).
- All exceptions, including those that we create, have methods defined by Throwable available to them. They are shown in the following Table:

| Method | Description |
|---|---|
| Throwable fillInStackTrace( ) | Returns a **Throwable** object that contains a completed stack trace. This object can be rethrown. |
| Throwable getCause( ) | Returns the exception that underlies the current exception. If there is no underlying exception, **null** is returned. |
| String getLocalizedMessage( ) | Returns a localized description of the exception. |
| String getMessage( ) | Returns a description of the exception. |
| StackTraceElement[ ] getStackTrace( ) | Returns an array that contains the stack trace, one element at a time, as an array of **StackTraceElement**. The method at the top of the stack is the last method called before the exception was thrown. This method is found in the first element of the array. The **StackTraceElement** class gives your program access to information about each element in the trace, such as its method name. |
| Throwable initCause(Throwable *causeExc*) | Associates *causeExc* with the invoking exception as a cause of the invoking exception. Returns a reference to the exception. |
| void printStackTrace( ) | Displays the stack trace. |
| void printStackTrace(PrintStream *stream*) | Sends the stack trace to the specified stream. |
| void printStackTrace(PrintWriter *stream*) | Sends the stack trace to the specified stream. |
| void setStackTrace(StackTraceElement *elements*[ ]) | Sets the stack trace to the elements passed in *elements*. This method is for specialized applications, not normal use. |
| String toString( ) | Returns a **String** object containing a description of the exception. This method is called by **println( )** when outputting a **Throwable** object. |

- We may also override one or more of the methods in exception classes that we create.
- The following example declares a new subclass of Exception and then uses that subclass to signal an error condition in a method. It overrides the toString( ) method (defined by Throwable and inherited by Exception), allowing a improved description of the exception to be displayed.

```
class MyException extends Exception
{
	private int detail;
	MyException(int a)
	{
		detail = a;
	}
	public String toString()
	{
		return "MyException[" + detail + "]";
	}
}
class ExceptionDemo
{
	static void compute(int a) throws MyException
	{
		System.out.println("Called compute(" + a + ")");
		if(a > 10)
			throw new MyException(a);
		System.out.println("Normal exit");
```

```
        }
        public static void main(String args[])
        {
              try
              {
                    compute(1);
                    compute(20);
              }
              catch (MyException e)
              {
                    System.out.println("Caught " + e);
              }
        }
  }
```

**OUTPUT**:

Called compute(1)
Normal exit
Called compute(20)
Caught MyException[20]