This covers basic java concepts and syntax + some portions of "Advanced Java Programming" –Unit: 1 (BSc. CSIT, TU)

Inheritance in Java

In OOP, computer programs are designed in such a way where everything is an object that interact with one another. Inheritance is one such concept where the properties of one class can be inherited by the other. It helps to reuse the code and establish a relationship between different classes.

- Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance the information is made manageable in a hierarchical order.
- Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications.
- Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it.
- The class which inherits the properties of other is known as **subclass (derived class, child class)** and the class whose properties are inherited is known as **superclass (base class, parent class).**

- Therefore, a subclass is a specialized version of a superclass. It inherits all of the members defined by the superclass and adds its own, unique elements

Inheritance Basics

- To inherit a class, you simply incorporate the definition of one class into another by using the **extends** keyword.
- **extends** is the keyword used to inherit the properties of a class. Following is the syntax of **extends** keyword.

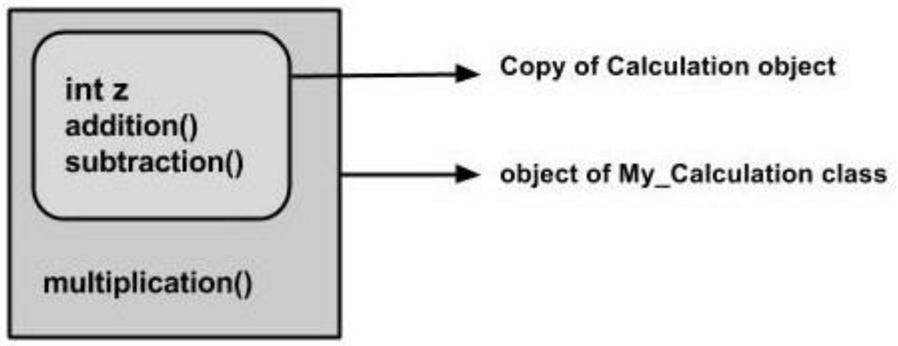
  Syntax

  ```
  class subclass-name extends superclass-name {
     // body of class
  }
  ```
- A major advantage 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 Calculation {
   int z;
   public void addition(int x, int y) {
      z = x + y;
      System.out.println("The sum of the given numbers:"+z);
   }
   public void subtraction(int x, int y) {
      z = x - y;
      System.out.println("The difference between the given 
numbers:"+z);
   }
}

public class My_Calculation extends Calculation {
   public void multiplication(int x, int y) {
      z = x * y;
      System.out.println("The product of the given numbers:"+z);
   }
   public static void main(String args[]) {
      int a = 20, b = 10;
      My_Calculation demo = new My_Calculation();
      demo.addition(a, b);
      demo.subtraction(a, b);
      demo.multiplication(a, b);
   }
}
```

In the given program, when an object to **My_Calculation** class is created, a copy of the contents of the superclass is made within it. That is why, using the object of the subclass you can access the members of a superclass.



The Superclass reference variable can hold the subclass object, but using that variable you can access only the members of the superclass, so to access the members of both classes it is recommended to always create reference variable to the subclass.

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**. For example, consider the following simple class hierarchy:
- Subclasses are allowed to access the members which have been declared **protected in** super class

```
/* In a class hierarchy, private members remain
   private to their class.

   This program contains an error and will not
   compile.
*/

// Create a superclass.
class A {
  int i; // public by default
  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 class member that has been declared as private will remain private to its class. It is not accessible by any code outside its class, including subclasses

Using super

- **super** is a keyword
- Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **supe**r. **super** has two general forms. The first calls the superclass' constructor. The second is used to access a member of the superclass that has been hidden by a member of a subclass.

Using super to Call Superclass Constructors

- A subclass can call a constructor defined by its superclass by use of the following form of super:

  `super(arg-list);`

  Here, **arg-list** specifies any arguments needed by the constructor in the superclass. **super( )** must always be the first statement executed inside a subclass' constructor.

```
class Superclass {

   int age;

   Superclass(int age) {

      this.age = age;

   }

   public void getAge() {

      System.out.println("The value of the variable named age in
super class is: " +age);

   }

}

public class Subclass extends Superclass {

   Subclass(int age) {

      super(age);

   }

   public static void main(String argd[]) {

      Subclass s = new Subclass(24);

      s.getAge();

   }}
```

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

A Second Use for super

- The second form of super acts somewhat like this, except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form:

  `super.member`

  Here, **member** can be either a method or an instance variable.

- This second form of super is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.
- *It is used to differentiate the members of superclass from the members of subclass, if they have same names.*

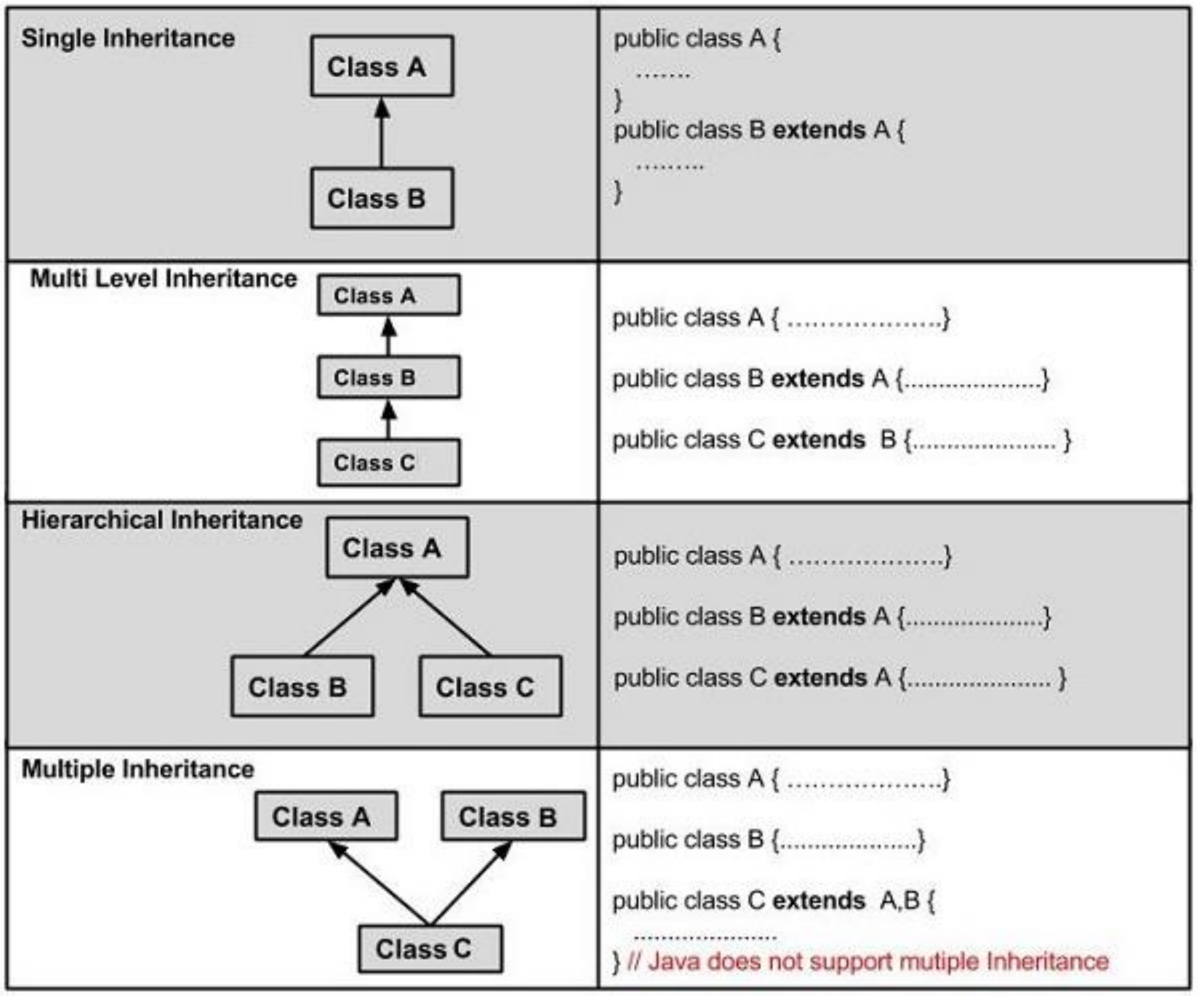
```
// Using super to overcome name hiding.
class A {
  int i;
}

// Create a subclass by extending class A.
class B extends A {
  int i; // this i hides the i in A

  B(int a, int b) {
    super.i = a; // i in A
    i = b; // i in B
  }

  void show() {
    System.out.println("i in superclass: " + super.i);
    System.out.println("i in subclass: " + i);
  }
}

class UseSuper {
  public static void main(String args[]) {
    B subOb = new B(1, 2);

    subOb.show();
  }
}
```

- Although the instance variable **i** in **B** hides the **i** in **A**, super allows access to the **i** defined in the superclass. As you will see, super can also be used to call methods that are hidden by a subclass.

Creating a Multilevel Hierarchy

Up to this point, we have been using simple class hierarchies that consist of only a superclass and a subclass. However, we can build hierarchies that contain as many layers of inheritance as we like. As mentioned, it is perfectly acceptable to use a subclass as a superclass of another. For example, given three classes called A, B, and C, C can be a subclass of B, which is a subclass of A. When this type of situation occurs, each subclass inherits all of the traits found in all of its super classes.

Types of inheritances


Class A
Class B
Class C

Single Inheritance	public class A { ……. } public class B **extends** A { ……… }
Multi Level Inheritance	public class A { ……………….} public class B **extends** A {………………..} public class C **extends** B {………………… }
Hierarchical Inheritance	public class A { ……………….} public class B **extends** A {………………..} public class C **extends** A {………………… }
Multiple Inheritance	public class A { ……………….} public class B {………………..} public class C **extends** A,B { ………………… } // Java does not support mutiple Inheritance

```
class Car{
   public Car()
   {
        System.out.println("Class Car");
   }
   public void vehicleType()
   {
        System.out.println("Vehicle Type: Car");
   }
}
class Maruti extends Car{
   public Maruti()
   {
        System.out.println("Class Maruti");
   }
   public void brand()
   {
        System.out.println("Brand: Maruti");
   }
   public void speed()
   {
        System.out.println("Max: 90Kmph");
   }
}
public class Maruti800 extends Maruti{

   public Maruti800()
   {
        System.out.println("Maruti Model: 800");
   }
   public void speed()
   {
        System.out.println("Max: 80Kmph");
   }
   public static void main(String args[])
   {
         Maruti800 obj=new Maruti800();
         obj.vehicleType();
         obj.brand();
         obj.speed();
   }
}
```

When Constructors Are Called

- When a class hierarchy is created, in what order are the constructors for the classes that make up the hierarchy executed? For example, given a subclass called B and a superclass called A, is A's constructor executed before B's, or vice versa?
- The answer is that **in a class hierarchy, constructors complete their execution in order of derivation, from superclass to subclass**.
- Further, since **super( )** must be the first statement executed in a subclass' constructor, this order is the same whether or not **super( )** is used. If **super( )** is not used, then the default

or parameterless constructor of each superclass will be executed. The following program illustrates when constructors are executed:

```
// Demonstrate when constructors are executed.

// Create a super class.
class A {
  A() {
    System.out.println("Inside A's constructor.");
  }
}
// Create a subclass by extending class A.
class B extends A {
  B() {
    System.out.println("Inside B's constructor.");
  }
}

// Create another subclass by extending B.
class C extends B {
  C() {
    System.out.println("Inside C's constructor.");
  }
}

class CallingCons {
  public static void main(String args[]) {
    C c = new C();
  }
}
```

OUTPUT:

```
Inside A's constructor
Inside B's constructor
Inside C's constructor
```

Polymorphism in Java

Polymorphism in java is a concept by which we can perform a single action by different ways. Polymorphism is derived from 2 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 and runtime polymorphism.** We can perform polymorphism in java by **method overloading and method overriding**

Method Overloading in Java

Method Overloading is a feature that allows a class to have more than one method having the same name, if their argument lists are different. Method overloading is an example of **compile time polymorphism.**

- If two or more method in a class have same name but different parameters, it is known as method overloading.
- In Java, it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be overloaded, and the process is referred to as method overloading
- In order to overload a method, the parameter lists of the methods must differ in either of these:

Number of parameters:

```
add(int a, int b)
add(int a, int b, int c)
```

Data type of parameters:

```
add(int x, int y)
add(int x, float y)
```

Sequence of data type of parameters

```
add(int p, float q)
add(float m , int n)
```

- Method overloading is one of the ways that Java supports polymorphism.
- 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.
- When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.
- In practice, you should only overload closely related operations.

// Method overloading by changing data type of parameters

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

Overloading Constructors

- ✓ In addition to overloading normal methods, you can also overload constructors
- ✓ Constructor overloading is a concept of having more than one constructor with different parameters list, in such a way so that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.
- ✓ Constructor overloading is done to construct object in different ways.

```
//Java program to overload constructors in java
    class Student{
       int id;
       String name;
       int age;
       //creating two arg constructor
       Student(int i, String n){
       id = i;
       name = n;
       }
       //creating three arg constructor
       Student(int i,String n,int a){
       id = i;
       name = n;
       age=a;
       }
       void display(){
                System.out.println(id+" "+name+" "+age);
                }

       public static void main(String args[]){
       Student s1 = new Student(111,"Kamal");
       Student s2 = new Student(222,"Abiral",25);
       s1.display();
       s2.display();
      }
    }
```

Method Overriding in Java

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding** in java

In other words, If subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding.

- ✓ Method overriding is used to provide specific implementation of a method that is already provided by its super class.
- ✓ Method overriding is used for runtime polymorphism

- ▪ 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 its 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.
- ▪ The benefit of overriding is: ability to define a behavior that's specific to the subclass type, which means a subclass can implement a parent class method based on its requirement.
- ▪ In object-oriented terms, overriding means to override the functionality of an existing method.

Rules for Method Overriding

- ✓ The argument list should be exactly the same as that of the overridden method.
- ✓ The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.
- ✓ 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.
- ✓ Instance methods can be overridden only if they are inherited by the subclass.
- ✓ A method declared final cannot be overridden.
- ✓ A method declared static cannot be overridden but can be re-declared.
- ✓ If a method cannot be inherited, then it cannot be overridden.
- ✓ Constructors cannot be overridden.

```
// Method overriding.
class A {
  int i, j;
  A(int a, int b) {
    i = a;
    j = b;
  }

  // display i and j
  void show() {
    System.out.println("i and j: " + i + " " + j);
  }
}
class B extends A {
  int k;

  B(int a, int b, int c) {
    super(a, b);
    k = c;
  }

  // display k - this overrides show() in A
  void show() {
    System.out.println("k: " + k);
  }
}

class Override {
  public static void main(String args[]) {
    B subOb = new B(1, 2, 3);

     subOb.show(); // this calls show() in B
  }
}
```

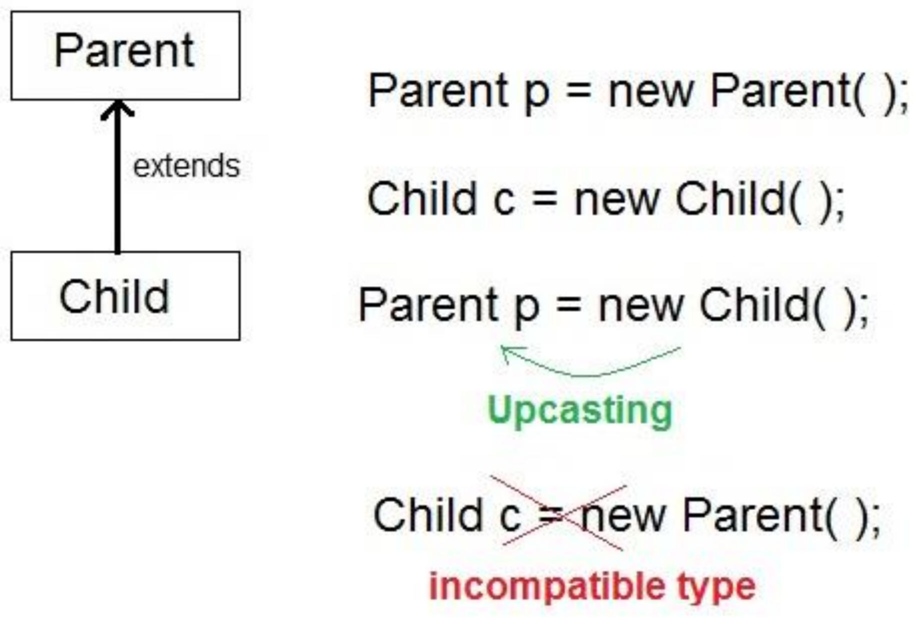
When **show( )** is invoked on an object of type **B**, the version of **show( )** defined within **B** is used. That is, the version of **show( )** inside **B** overrides the version declared in **A**.

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

Dynamic method dispatch

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

When an overridden method is called by a reference, java determines which version of that method to execute based on the type of object it refer to. In simple words the type of object which it referred determines which version of overridden method will be called.



- ✓ When Parent class reference variable refers to Child class object, it is known as **Upcasting**

Why Overridden Methods?

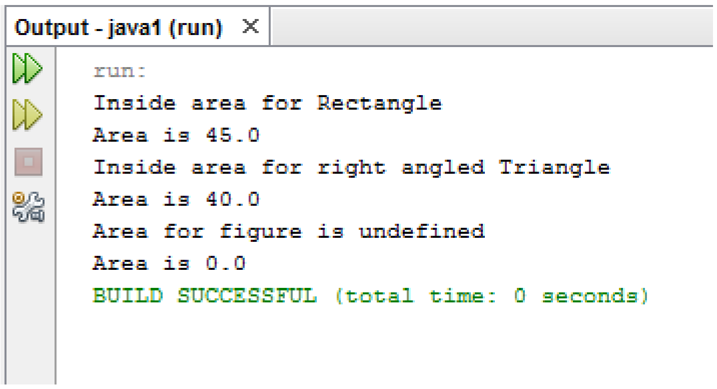
- As stated earlier, overridden methods allow Java to support run-time polymorphism.
- Polymorphism is essential to object-oriented programming for one reason: it allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.
- Overridden methods are another way that Java implements the “one interface, multiple methods” aspect of polymorphism.
- Part of the key to successfully applying polymorphism is understanding that the super classes and sub classes form a hierarchy which moves from lesser to greater specialization.
- Used correctly, the superclass provides all elements that a subclass can use directly. It also defines those methods that the derived class must implement on its own. This allows the subclass the flexibility to define its own methods, yet still enforces a consistent interface.
- Thus, by combining inheritance with overridden methods, a superclass 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. The ability of existing code libraries to call methods on instances of new classes without recompiling while maintaining a clean abstract interface is a profoundly powerful tool.

Applying Method Overriding

Let’s look at a more practical example that uses method overriding.

- The following program creates a superclass called **Figure** that stores the dimensions of a two-dimensional object. It also defines a method called **area( )** that computes the area of an object.
- The program derives two subclasses from **Figure**. The first is **Rectangle** and the second is **Triangle**. Each of these subclasses overrides **area( )** so that it returns the area of a rectangle and a triangle, respectively.

```
//Using run time polymorphism
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);
    }
    //overriding 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);
    }
    //overriding area for right angled Triangle
    double area(){
        System.out.println("Inside area for right angled
Triangle");
        return (dim1*dim2)/2;
    }
}
public 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());
    }
}
```



```
run:
Inside area for Rectangle
Area is 45.0
Inside area for right angled Triangle
Area is 40.0
Area for figure is undefined
Area is 0.0
BUILD SUCCESSFUL (total time: 0 seconds)
```

Through the dual mechanisms of inheritance and run-time polymorphism, it is possible to define one consistent interface that is used by several different, yet related, types of objects. In this case, if an object is derived from **Figure**, then its area can be obtained by calling **area( ).** The interface to this operation is the same no matter what type of figure is being used.

Compile time Polymorphism (or Static polymorphism)

Polymorphism that is resolved during compiler time is known as static polymorphism. Method overloading is an example of compile time polymorphism.

Runtime Polymorphism (or Dynamic polymorphism)

It is also known as Dynamic Method Dispatch. Dynamic polymorphism is a process in which a call to an overridden method is resolved at runtime, that's why it is called runtime polymorphism.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Upcasting: *When reference variable of Parent class refers to the object of Child class, it is known as upcasting.*

Static and Dynamic Binding in Java

Association of method call to the method body is known as binding. There are two types of binding: Static Binding that happens at compile time and Dynamic Binding that happens at runtime.

- ❖ The binding which can be resolved at compile time by compiler is known as **static or early binding**. The binding of static, private and final methods is compile-time. Why? The reason is that these method cannot be overridden and the type of the class is determined at the compile time.

  Static binding example

  Here we have two classes Human and Boy. Both the classes have same method walk() but the method is static, which means it cannot be overridden so even though I have used the object of Boy class while creating object obj, the parent class method is called by it. Because the reference is of Human type (parent class). So whenever a binding of static, private and final methods happen, type of the class is determined by the compiler at compile time and the binding happens then and there

```
class Human{
   public static void walk()
   {
       System.out.println("Human walks");
   }
}
class Boy extends Human{
   public static void walk(){
       System.out.println("Boy walks");
   }
   public static void main( String args[]) {
       /* Reference is of Human type and object is
        * Boy type
        */
       Human obj = new Boy();
       /* Reference is of HUman type and object is
        * of Human type.
        */
       Human obj2 = new Human();
       obj.walk();
       obj2.walk();
   }
}
```

❖ When compiler is not able to resolve the call/binding at compile time, such binding is known as **Dynamic or late Binding**. Method Overriding is a perfect example of dynamic binding as in overriding both parent and child classes have same method and in this case the type of the object determines which method is to be executed. The type of object is determined at the run time so this is known as dynamic binding.

Dynamic binding example

This is the same example that we have seen above. The only difference here is that in this example, overriding is actually happening since these methods are not static, private and final. In case of overriding the call to the overriden method is determined at runtime by the type of object thus late binding happens. Lets see an example to understand this:

```
class Human{
   //Overridden Method
   public void walk()
   {
       System.out.println("Human walks");
   }
}
class Demo extends Human{
   //Overriding Method
   public void walk(){
       System.out.println("Boy walks");
   }
   public static void main( String args[]) {
       /* Reference is of Human type and object is
        * Boy type
        */
       Human obj = new Demo();
       /* Reference is of HUman type and object is
        * of Human type.
        */
       Human obj2 = new Human();
       obj.walk();
       obj2.walk();
   }
}
```

As you can see that the output is different than what we saw in the static binding example, because in this case while creation of object obj the type of the object is determined as a Boy type so method of Boy class is called. Remember the type of the object is determined at the runtime.

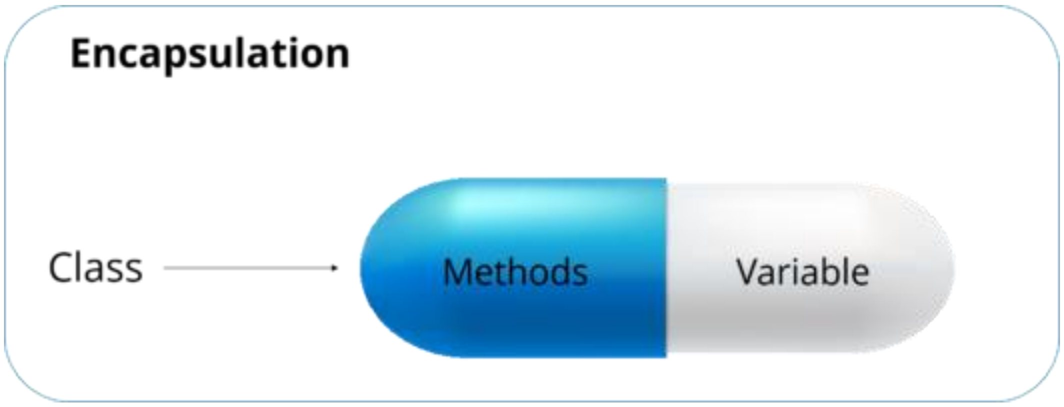
Static Binding vs Dynamic Binding

Let’s discuss the difference between static and dynamic binding in Java.

- Static binding happens at compile-time while dynamic binding happens at runtime.
- Binding of private, static and final methods always happen at compile time since these methods cannot be overridden. When the method overriding is actually happening and the reference of parent type is assigned to the object of child class type then such binding is resolved during runtime.
- The binding of overloaded methods is static and the binding of overridden methods is dynamic.

Encapsulation in Java

Encapsulation is a mechanism where you bind your data and code together as a single unit. It also means to hide your data in order to make it safe from any modification. What does this mean? The best way to understand encapsulation is to look at the example of a medical capsule, where the drug is always safe inside the capsule. Similarly, through encapsulation the methods and variables of a class are well hidden and safe.



In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as **data hiding.**

To achieve encapsulation in Java −

- Declare the variables of a class as private.
- Provide public setter and getter methods to modify and view the variables values.

Example

```java
/* File name : EncapTest.java */

public class EncapTest {

   private String name;

   private String idNum;

   private int age;

   public int getAge() {

      return age;

   }

   public String getName() {

      return name;

   }

   public String getIdNum() {

      return idNum;

   }

   public void setAge( int newAge) {

      age = newAge;

   }

   public void setName(String newName) {

      name = newName;

   }

   public void setIdNum( String newId) {

      idNum = newId;

   }

}
```

The public **setXXX()** and **getXXX()** methods are the access points of the instance variables of the EncapTest class. Normally, these methods are referred as **getter**s and **setters**. Therefore, any class that wants to access the variables should access them through these getters and setters**.**

The variables of the EncapTest class can be accessed using the following program –

```
/* File name : RunEncap.java */

public class RunEncap {


   public static void main(String args[]) {

      EncapTest encap = new EncapTest();

      encap.setName("James");

      encap.setAge(20);

      encap.setIdNum("12343ms");

      System.out.print("Name : " + encap.getName() + " Age : " + encap.getAge());

   }

}
```

This will produce the following result –

Output

```
Name : James Age : 20
```

Benefits of Encapsulation

- The fields of a class can be made read-only or write-only.
- A class can have total control over what is stored in its fields.

Abstraction in Java

As per dictionary, abstraction is the quality of dealing with ideas rather than events. For example, when you consider the case of e-mail, complex details such as what happens as soon as you send an e-mail, the protocol your e-mail server uses are hidden from the user. Therefore, to send an e-mail you just need to type the content, mention the address of the receiver, and click send.

Likewise in Object-oriented programming, abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user. In other words, the user will have the information on what the object does instead of how it does it. It basically deals with hiding the details and showing the essential things to the user.

In Java, abstraction is achieved using **Abstract classes** and **Interfaces**.

Abstract Classes

- There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method.
- That is, sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement.
- One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method. This is the case with the class Figure used in the preceding example. The definition of area( ) is simply a placeholder. It will not compute and display the area of any type of object.
- As you will see as you create your own class libraries, it is not uncommon for a method to have no meaningful definition in the context of its superclass. You can handle this situation two ways. One way, as shown in the previous example, is to simply have it report a warning message. While this approach can be useful in certain situations—such as debugging—it is not usually appropriate. You may have methods that must be overridden by the subclass in order for the subclass to have any meaning. Consider the class **Triangle**. It has no meaning if **area( )** is not defined. In this case, you want some way to ensure that a subclass does, indeed, override all necessary methods. Java's solution to this problem is the abstract method.
- You can require that certain methods be overridden by subclasses by specifying the **abstract** type modifier. These methods are sometimes referred to as subclasser responsibility because they have no implementation specified in the superclass. Thus, a subclass must override them—it cannot simply use the version defined in the superclass.

  Abstract Method??

A method without body (no implementation) is known as abstract method

- To declare an abstract method, use this general form:

```
abstract return_type method_name(parameter-list);
```

As you can see, no method body is present.

<u>Points to remember about abstract method</u>

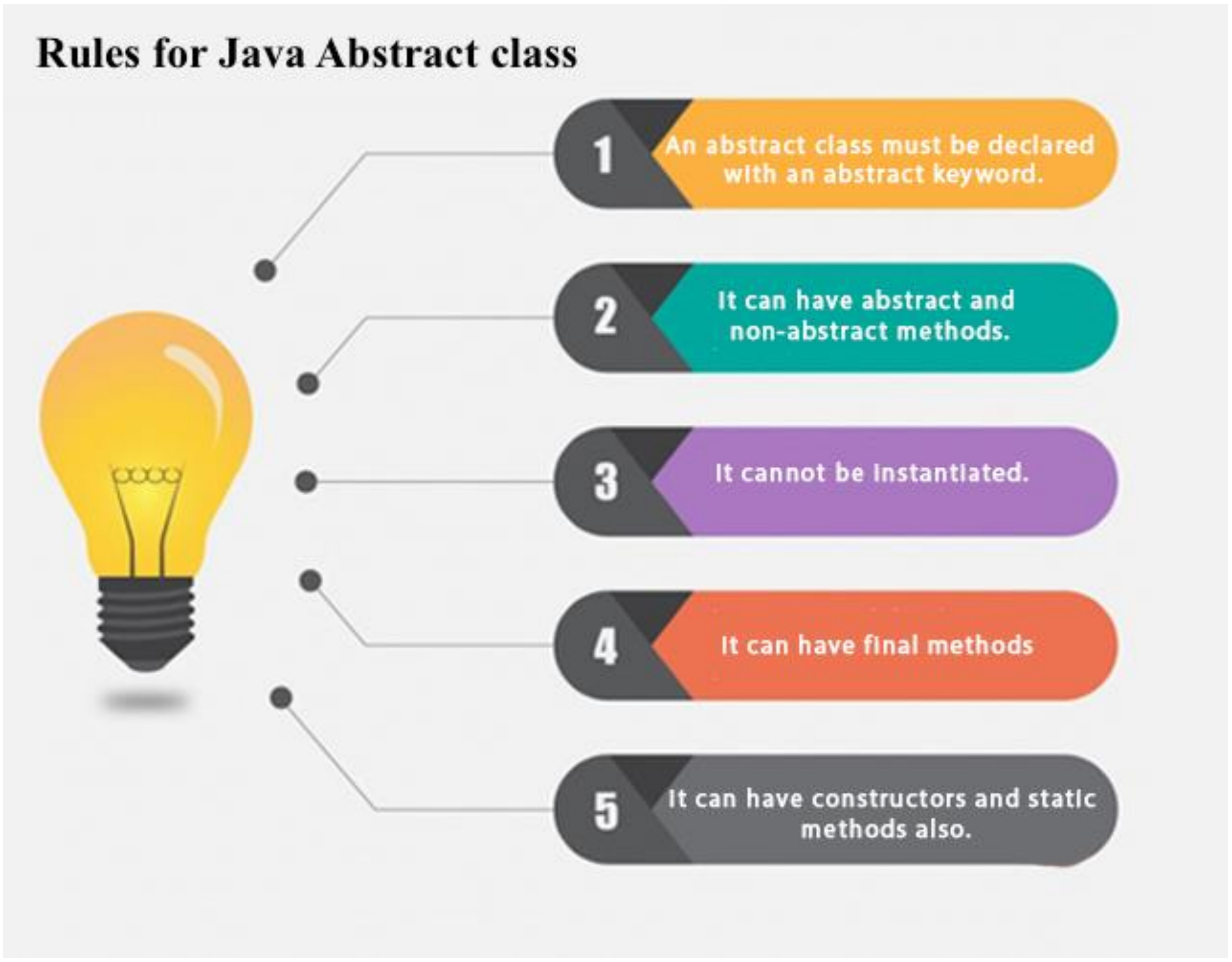
- ✓ Abstract methods don't have body, they just have method signature as shown above.
- ✓ If a class has an abstract method it should be declared abstract, the vice versa is not true, which means an abstract class doesn't need to have an abstract method compulsory.
- ✓ If a regular class extends an abstract class, then the class must have to implement all the abstract methods of abstract parent class or it has to be declared abstract as well.

Abstract class??

A class that is declared using "abstract" keyword is known as abstract class. It can have abstract methods (methods without body) as well as concrete methods (regular methods with body). Unlike normal class, it cannot be instantiated (i.e. we cannot create object of abstract class directly) . A normal class (non-abstract class / concrete class) cannot have abstract methods.

<u>Points to remember about abstract class</u>

- ♣ Any class that contains one or more abstract methods must also be declared abstract.
- ♣ To declare a class abstract, you simply use the **abstract** keyword in front of the class keyword at the beginning of the class declaration.
- ♣ 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.
- ♣ Also, you 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 declared abstract itself.


Rules for Java Abstract class
1 An abstract class must be declared with an abstract keyword.
2 It can have abstract and non-abstract methods.
3 It cannot be instantiated.
4 It can have final methods
5 It can have constructors and static methods also.

Example:

```
abstract class Shape{
void meth(){
System.out.println("I am from normal method");
}
abstract void draw();
}
class Rectangle extends Shape{
void draw(){
System.out.println("drawing rectangle");}
}
class Circle extends Shape{
```

```
void draw(){
System.out.println("drawing circle");
}
}
class TestAbstraction{
public static void main(String args[]){
Shape s=new Circle();
s.draw();
}
}
```

```
//abstract parent class
abstract class Animal{
   //abstract method
   public abstract void sound();
}
//Dog class extends Animal class
public class Dog extends Animal{

   public void sound(){
        System.out.println("Woof");
   }
   public static void main(String args[]){
        Animal obj = new Dog();
        obj.sound();
   }
}
```

Why we need an abstract class?

- Let’s say we have a class Animal that has a method sound() and the subclasses(see inheritance) of it like Dog, Lion, Horse, Cat etc. Since the animal sound differs from one animal to another, there is no point to implement this method in parent class. This is because every child class must override this method to give its own implementation details, like Lion class will say “Roar” in this method and Dog class will say “Woof”.

  So when we know that all the animal child classes will and should override this method, then there is no point to implement this method in parent class. Thus, making this method abstract would be the good choice as by making this method abstract we force all the sub classes to implement this method(otherwise you will get compilation error), also we need not to give any implementation to this method in parent class.

  Since the Animal class has an abstract method, you must need to declare this class abstract.

  Now each animal must have a sound, by making this method abstract we made it compulsory to the child class to give implementation details to this method. This way we ensures that every animal has a sound.

Why can't we create the object of an abstract class?

Because these classes are incomplete, they have abstract methods that have no body. So if java allows you to create object of this class then if someone calls the abstract method using that object then what would happen? There would be no actual implementation of the method to invoke.
Also because an object is concrete. An abstract class is like a template, so you have to extend it and build on it before you can use it.

Abstract class vs Concrete class

A class which is not abstract is referred as Concrete class.

- ✓ An abstract class has no use until unless it is extended by some other class.
- ✓ If you declare an abstract method in a class then you must declare the class abstract as well.
- ✓ You can't have abstract method in a concrete class. It's vice versa is not always true: If a class is not having any abstract method then also it can be marked as abstract.
- ✓ Abstract class can have non-abstract method (concrete) as well.

Interface in Java

- An interface is a reference type in Java. It is similar to class
- An interface in java is a blueprint of a class. It has static constants and abstract methods.
- Using the keyword **interface**, you can fully abstract a class' interface from its implementation. That is, using interface, you can specify what a class must do, but not how it does it.
- Interfaces specify what a class must do and not how.
- Interfaces are syntactically similar to classes, but they lack instance variables, and, as a general rule, their methods are declared without any body. In practice, this means that you can define interfaces that don't make assumptions about how they are implemented.
- Once it is defined, any number of classes can implement an interface. Also, one class can implement any number of interfaces.
- Writing an interface is similar to writing a class. But a class describes the attributes and behaviors of an object. And an interface contains behaviors that a class implements.
- Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

- Syntax:

  ```
  [modifiers] interface InterfaceName {

  // declaring methods
  [public abstract] returnType methodName1(arguments);

  // defining constants
  [public static final] type fieldName = value;
  }
  ```

  The access level for the entire interface is usually public. It may be omitted, in which case the interface is only available to other classes in the same package (i.e. default access).Note, for the sake of completeness, and there are situations where the interface definition could be protected or private; these involve what are called inner classes.

- Beginning with JDK 8, it is possible to add a default implementation to an interface method. Also, static methods are supported in interface.

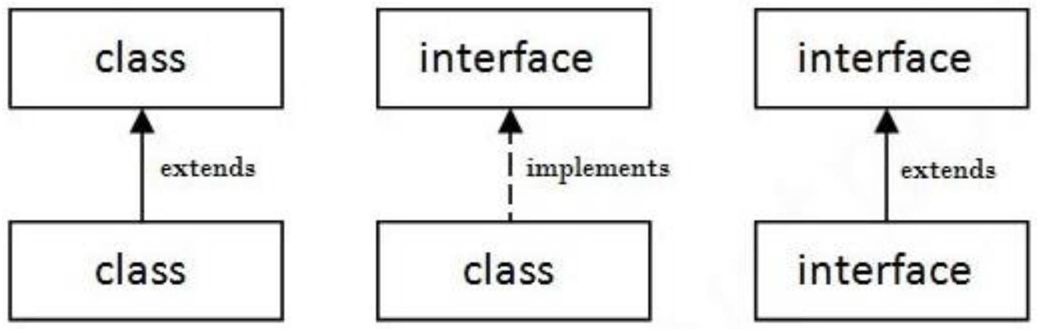
An interface is similar to a class in the following ways −

- ✓ An interface can contain any number of methods.
- ✓ An interface is written in a file with a .java extension, with the name of the interface matching the name of the file.
- ✓ The byte code of an interface appears in a .class file.
- ✓ Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

However, an interface is different from a class in several ways, including −

- ✓ We cannot instantiate an interface (i.e. object cannot be created).
- ✓ An interface does not contain any constructors.
- ✓ All of the methods in an interface are abstract.
- ✓ An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both **static** and **final**.
- ✓ An interface is not extended by a class; it is **implemented** by a class.
- ✓ An interface can extend multiple interfaces.

Note : As shown in the figure given below, a class extends another class, an interface extends another interface, but a class implements an interface.



Interfaces have the following properties −

- An interface is implicitly abstract. You do not need to use the abstract keyword while declaring an interface.
- Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
- Methods in an interface are implicitly public.

Interface fields are public, static and final by default, and the methods are public and abstract.

Implementing Interfaces

- Once an interface has been defined, one or more classes can implement that interface. To implement an interface, include the implements clause in a class definition, and then create the methods required by the interface.
- When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.
- A class uses the implements keyword to implement an interface
- Syntax:

```
[modifiers] class ClassName implements InterfaceName {
//any desired fields


// implement required methods
[modifiers] returnType methodName1(arguments) {
     //executable code
}


//any other desired methods


}
```

- If a class implements more than one interface, the interfaces are separated with a comma.

  If a class **Tyre** is implementing two interfaces **Moveable** and **Rollable** then

```
class Tyre implements Moveable, Rollable
{
. . . .
}
```

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

```
//Interface declaration: by first user
interface Drawable{
void draw();
}
//Implementation: by second user
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class Circle implements Drawable{
public void draw(){System.out.println("drawing circle");}
}
//Using interface: by third user
class TestInterface1{
public static void main(String args[]){
Drawable d=new Circle();
d.draw();
     }
}
```

Another Example:

```
interface Bank{
float rateOfInterest();
}
class SBI implements Bank{
public float rateOfInterest(){return 9.15f;}
}
class NMB implements Bank{
public float rateOfInterest(){return 9.7f;}
}
class TestInterface{
public static void main(String[] args){
Bank b=new SBI();
System.out.println("ROI: "+b.rateOfInterest());
}
}
```

Extending Interfaces

One interface can inherit another by use of 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 required by the interface inheritance chain.

Following is an example:

```
// One interface can extend another.
interface A {
  void meth1();
  void meth2();
}

// B now includes meth1() and meth2() -- it adds meth3().
interface B extends A {
  void meth3();
}

// This class must implement all of A and B
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();
  }
}
```

Extending Multiple Interfaces

A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface.

The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.

Multiple inheritance is not supported through class in java, but it can be achieved using interfaces

```
interface Printable{
void print();
}
interface Showable{
void show();
}
class Test implements Printable,Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
Test obj = new Test();
obj.print();
obj.show();
 }
}
```

Advantages of interface in java:

Advantages of using interfaces are as follows:

- ✓ Without bothering about the implementation part, we can achieve the security of implementation
- ✓ In java, **multiple inheritance** is not allowed, however you can use interface to make use of it as you can implement more than one interface.

Abstract Class vs Interface

Both abstract class and interface are used for abstraction.

<u>Abstract class vs Interface</u>

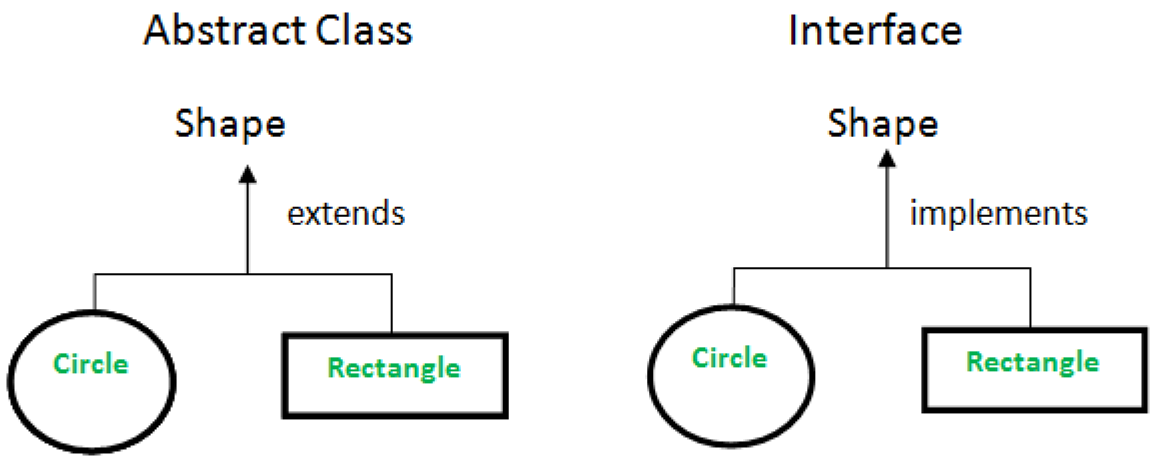


Figure 1: Abstract Class vs Interface

1. **Type of methods:** Interface can have only abstract methods. Abstract class can have abstract and non-abstract methods. From Java 8, it can have default and static methods also.
2. **Final Variables:** Variables declared in a Java interface are by default final. An abstract class may contain non-final variables.
3. **Type of variables**: Abstract class can have final, non-final, static and non-static variables. Interface has only static and final variables.
4. **Implementation:** Abstract class can provide the implementation of interface. Interface can't provide the implementation of abstract class.
5. **Inheritance vs Abstraction:** A Java interface can be implemented using keyword "implements" and abstract class can be extended using keyword "extends".
6. **Multiple implementation**: An interface can extend another Java interface only, an abstract class can extend another Java class and implement multiple Java interfaces.
7. **Accessibility of Data Members:** Members of a Java interface are public by default. A Java abstract class can have class members like private, protected, etc.

final keyword in Java

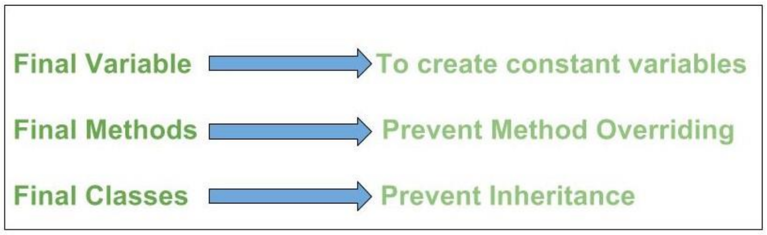
Introducing final [final keyword]

- **final** keyword is a **non-access modifier**
- The final keyword in java is used to restrict the user. The java final keyword can be used in many context. Final can be:
    - variable (field)
    - method
    - class

A field can be declared as **final**. Doing so prevents its contents from being modified, making it, essentially, a **constant**. This means that you must initialize a final field when it is declared. You can do this in one of two ways: First, you can give it a value when it is declared. Second, you can assign it a value within a constructor. The first approach is the most common.

In addition to fields, both method parameters and local variables can be declared final. Declaring a parameter final prevents it from being changed within the method. Declaring a local variable final prevents it from being assigned a value more than once. The keyword final can also be applied to methods and class.

- If we make any **variable as final**, we cannot change the value of final variable (It **will be constant**) [**final** variable once assigned a value can never be changed.]
- If we make any **method as final**, we **cannot override** it.
- If we make any class as final, we cannot extend it (i.e. we cannot create subclass)



- **NOTE: final variables are declared in upper case (convention)**

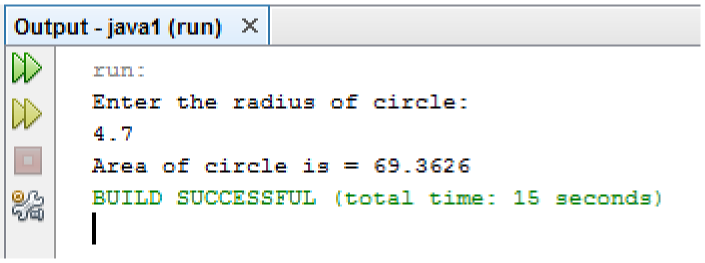
The following program will produce compile time error

```
 class Bike{

final int SPEED_LIMIT=50;//final variable
void run(){
SPEED_LIMIT =110;  // We cannot reassign the value to final variable . ERROR occurs !
}
public static void main(String args[]){
Bike mybike=new  Bike9();
```

```
    obj.run();
    }
   }//end of class
```

//Program to show the use of final variable

```
package examples;
import java.util.Scanner;
public class MyCircle {
    final double PI = 3.14;
    double r;
    void getRad(){
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the radius of circle:");
        r = sc.nextDouble();
    }
    void circleArea(){
        double a = PI*r*r;
        System.out.println("Area of circle is = "+a);
    }
    public static void main(String[] args) {
        MyCircle c1 = new MyCircle();
        c1.getRad();
        c1.circleArea();
    }
}
```



```
run:
Enter the radius of circle:
4.7
Area of circle is = 69.3626
BUILD SUCCESSFUL (total time: 15 seconds)
```