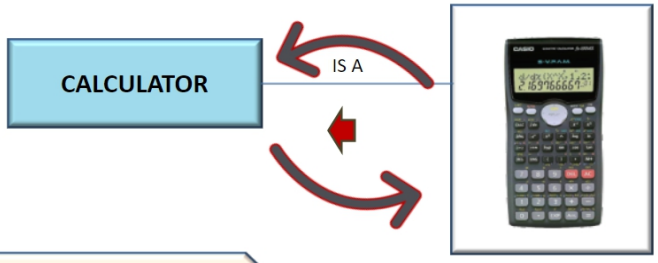
TEKNOTURF

INHERITANCE

Inheritance - Overview

TEKNOTURF



If the relationship between two classes is "IS A" or "IS A TYPE OF" then that relationship is termed as "Inheritance"

Inheritance

TEKNOTURF

- Inheritance is one of the major features of OO Concept.
- Inheritance is the process in which one object can acquire the properties and behaviour of the parent object.
- Using inheritance new classes can be built on already existing classes, so that the new class can reuse methods and fields present in the parent class.
- Inheritance represents an ***IS-A*** relationship, also known as parent-child relationship.

Inheritance

- Different types of objects have certain features in common
- They also possess certain special features and behaviors that make them different

Example

Mountain Bike

Road Bike

Tandem Bike

Common Features : Speed, pedal cadence, gear

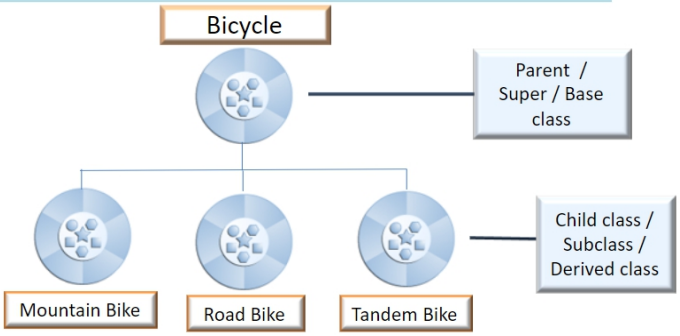
Special Features:

- Additional chain ring, Thick tyre
- Drop handlebars, Shock absorbers
- Two seats and two handlebars

Inheritance

Bicycle becomes the superclass of the classes Mountain Bike, Road Bike and Tandem Bike.

By inheriting from an already existing class, the methods of the parent class can be reused and can also add new fields and methods for the child class



Inheritance

Benefits of Inheritance

- Code Reusability
- Easy maintenance

Attributes and methods defined in a super class are automatically inherited by all its sub classes and reused.

If any changes are to be incorporated, it can be done in the super class, which in turn gets reflected to all the sub classes, making maintenance easy.

SimpleCalculator
Add
Subtract
Multiply
Divide

ScientificCalculator
calculateSine
calculateCos
calculateLog
calculateSqrt

Inheritance

Example – Without inheritance

```
public class Employee {
    private int empId;
    private String name;

    public Employee()
    {
    System.out.println("In Employee Constructor");
    }
   //Assume Getters and setters for all attributes

}
```

```
public class PermanentEmployee
{
  private int empId;
  private String name;
  private int basicPay;

  public PermanentEmployee()
  {
    System.out.println("In
      PermanentEmployee  constructor " );
  }
  // Assume Getters and setters for all
                        attributes
}
```

Inheritance

If there is an "IS A" relationship between two classes then an inheritance relationship can exist between them

Example - with Inheritance

- Permanent Employee is a type of Employee. So there is an inheritance relationship.

```
public class Employee {
  private int empId;
  private String name;
  public Employee()  //constructor
  {
     System.out.println("In Employee
                        Constructor");
  }
}
```

```
public class PermanentEmployee extends Employee
{
    private int basicPay;
    public PermanentEmployee()    //constructor
    {
        System.out.println("In PermanentEmployee
                                    constructor");
    }
}
```

Inheritance

- Example (contd.)

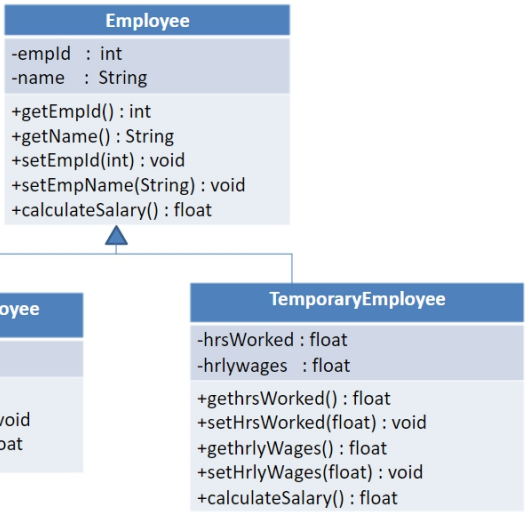
```
public class EmployeeMain {
        public static void main(String a[]) {
           PermanentEmployee per_empObj=new PermanentEmployee();
        }
}
```

Output :

In Employee Constructor
In PermanentEmployee Constructor

Subclassing

Inheritance leads to reusability of code, thereby making maintenance easier



Inheritance

- Private members of the superclass are not inherited by the subclass; instead they can only be indirectly accessed using public methods.
- Members with default access specifier in superclass cannot be inherited by subclasses in other packages. They can be accessed directly using their names in subclasses within the same package as the superclass.
- The protected modifier specifies that the member can only be accessed within its own package and, in addition, by a subclass of its class in another package.
- Constructors are not members of a class. So they are not inherited by a subclass, but the constructor of the superclass can be invoked from the subclass.

Inheritance

Example : when members are not protected

```
public class Employee {
  private  int empId;
  private  String name;
  public Employee()  //constructor
  {
     System.out.println("In Employee
                         Constructor");
  }
  //Assume Getters and Setters
}
```

```
public class PermanentEmployee extends Employee
{    private int basicPay;
    public PermanentEmployee()    //constructor
    {
        System.out.println("In PermanentEmployee
                                constructor");
    }
    public void display()
    {
      System.out.println("ID "+getEmpID()+"  Name "
                  +getName()+"  Bpay  "+basicPay);
 // As fields are private use public methods to access
    }
}
```

Inheritance

Example using protected

```
public class Employee {
  protected  int empId;
  protected String name;
  public Employee()  //constructor
  {
     System.out.println("In Employee
                         Constructor");
  }
}
```

```
public class PermanentEmployee extends Employee
{
    private int basicPay;
    public PermanentEmployee()    //constructor
    {
        System.out.println("In PermanentEmployee
                                constructor");
    }
    public void  display()
    {
       System.out.println("ID "+empID+"  Name "
                  +name+"  Bpay  "+basicPay);
    }
}
```

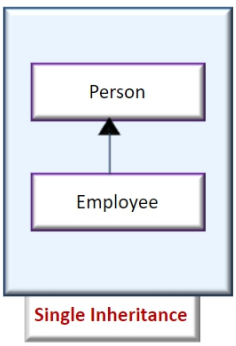
Access Control

Access modifiers on class member declarations are listed here:

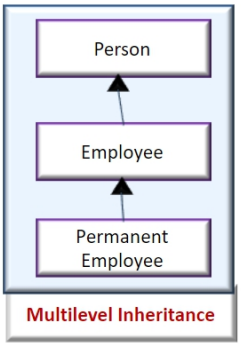
Modifier	Same Class	Same Package	Subclass	Universe
private	Yes			
default	Yes	Yes		
protected	Yes	Yes	Yes	
public	Yes	Yes	Yes	Yes

Types of Inheritance

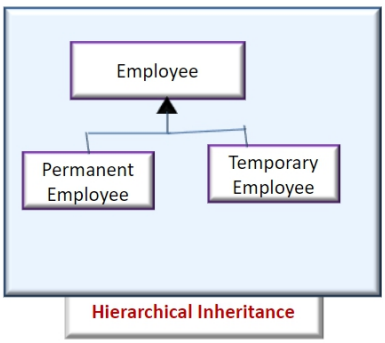
Types of Inheritance
- Single
- Multilevel
- Hierarchical



Single Inheritance



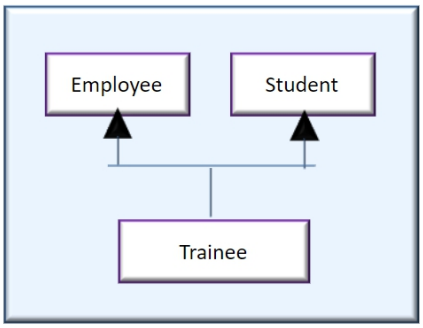
Multilevel Inheritance



Hierarchical Inheritance

Types of Inheritance

- When a class extends more than one class we call that as multiple inheritance
- Multiple inheritance is not supported in java.



Method Overriding

Daughter inherits the properties of Mom

Mom is the super class and daughter is the subclass.

Method Overriding

Mom is the Super Class and daughter is the Sub Class.

Mom likes to listen to music. Daughter also likes to listen to music.

Mom likes melody, whereas

daughter loves pop…

*Here **daughter overrides the behavior of mom***

Overriding Methods

- A subclass can modify behavior inherited from a parent class
- A subclass can have a method with same name, argument list and return type (except co-variant type) as declared in the parent class, but with a different implementation. This concept is called Method overriding.
- Rules for Method overriding
  Method name should be same as in the parent class
  Methods argument list should be the same as in parent class
  Same return type (or its sub type)
  The subclass overridden method cannot have weaker access than super class method.
  Example : If parent class method is declared as public, then the overridden method in sub class cannot be private or protected or default.
- When the overridden method is invoked using a sub class object, it will override the method in the parent class and execute the method in the child class. Hence the concept Method overriding

Method Overriding

- Rules on access specifier in method overriding
  - *When overriding a method, the access level cannot be more restrictive than the overridden methods access level.*
  - *If not, it results in compilation error*
  - *The order of access level is private, default, protected and public (from more restrictive to less)*

Access Level in parent class	Access Level permissible in child class overridden method
Default	default , protected, public
Protected	protected, public
public	public

Method Overriding

Example :

```
public class Employee {
  protected  int empId;
  protected String name;
  public Employee()   //constructor
  {
     System.out.println("In Employee Constructor");
  }

  //Assume you have written getters and setters for empId  and name

  public float calculateSalary()
  {
     System.out.println("In Super class calculateSalary Method");
     return 1;
  }
}
```

Method Overriding

Example Contd.

```
public class PermanentEmployee extends Employee {
    private int basicPay;
    public PermanentEmployee()    //constructor
    {
        System.out.println("In PermanentEmployee  constructor");
    }
    // Assume you have written getters and setters for basicPay
    public  float  calculateSalary()   //  Overriding the method in super class
    {
       float  da = basicPay * 0.10f;
       float  pf =  basicPay *  0.12f;
       float salary = basicPay + da – pf;
       return salary;
    }
}
```

Overridden Methods

Example Contd.

```
public class EmployeeUtility {
    public static void main(String a[])
    {
        Employee empObj=new Employee();
        empObj.setEmpId(101);
        empObj.setName("Rohith");
        empObj.calcuateSalary();

        PermanentEmployee  perEmpObj=new
                            PermanentEmployee();
        perEmpObj.setEmpId(102);
        perEmpObj.setName("Tom");
        perEmpObj.setBasicPay(35000);
        System.out.println("Salary is "+
                    perEmpObj.calculateSalary());
    }
}
```

Output :

In Employee Constructor
In Super class calculateSalary Method

In Employee Constructor
In PermanentEmployee constructor
Salary is 34300.0

super

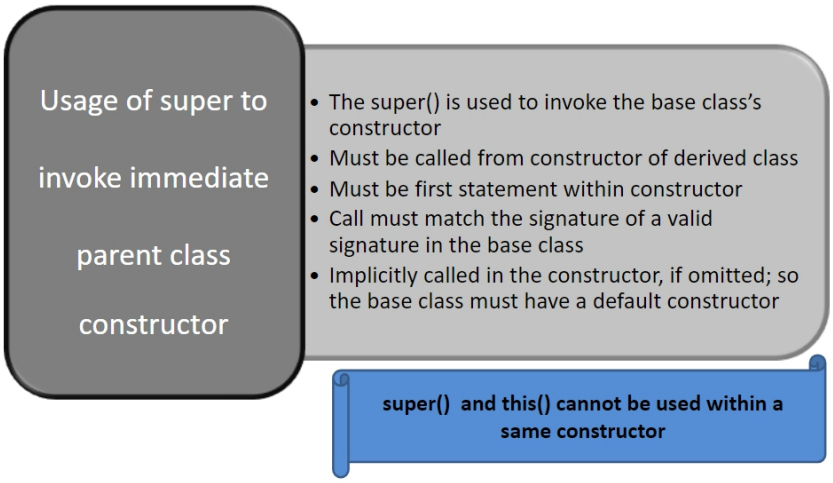
super keyword is a reference variable that refers to the immediate parent of a class

Creation of subclass object implicitly creates a parent class object, which can be referenced using the super keyword

Usage of super keyword

- super is used to invoke immediate parent class constructor
- super is used to invoke immediate parent class method

Use of super()



Usage of super

Example

```
class Employee {
   public Employee()
   {
      System.out.println("In Employee
                          Constructor");
   }
}
class PermanentEmployee extends
                          Employee {
    public PermanentEmployee()
    {
      //Compiler adds super()  implicitly
        System.out.println("In
         PermanentEmployee  constructor");
    }
}
```

```
public class Main {
   public static void main(String a[])
   {
      PermanentEmployee pObj=new
           PermanentEmployee();
   }
}
```

Output :

In Employee Constructor
In PermanentEmployee constructor

Usage of super

Example

```
class Employee {
   public Employee()
   {
      System.out.println("In Employee
                          Constructor");
   }
}
class PermanentEmployee extends
                          Employee {
    public PermanentEmployee()
    {
      super();   //can also be called explicitly
      System.out.println("In
PermanentEmployee  constructor");
    }
}
```

```
public class Main {
   public static void main(String a[])
   {
      PermanentEmployee pObj=new
           PermanentEmployee();
   }
}
```

Output :

In Employee Constructor
In PermanentEmployee constructor

Usage of Super

- super must be the first statement in constructor
- Parametrized constructor of parent class can be invoked by passing the required parameters while calling super.

Example:

```
public class  Employee{
   protected  int empId;
   protected  String name;

   public Employee(int empId,String name)
   {
      this.empId = empId;
      this.name = name;
   }
```

```
   public  void display()
   {
      System.out.println("ID  : "+id+"
                        Name : "+name);
   }
}
```

Usage of super

```
public class PermanentEmployee extends Employee{
    private int basicPay;
    public    PermanentEmployee(
            int empId,String name,int basicPay )
    {
        super(empId,name) ;
        this.basicPay=basicPay;
    }
    public void display()
    {
        System.out.println("ID  : "+id+"Name
                                  : "+name );
        System.out.println("Basic Pay  : "+basicPay);
    }
}
```

```
public class Main {
   public static void main(String a[])
   {
      PermanentEmployee pObj=new
PermanentEmployee(102,"Tom",35000);
      pObj.display();  //calls the
                       //overridden method
   }
}
```

Output :

ID: 102 Name : Tom
Basic Pay : 35000

Usage of super

Usage of super keyword to invoke super class method

```
public class PermanentEmployee extends Employee{
    private int basicPay;
    public    PermanentEmployee(
            int empId,String name,int basicPay )
    {
        super(empId,name) ;
        this.basicPay=basicPay;
    }
    public void display()
    {
       super.display();  //invokes the overridden method
       System.out.println("Basic Pay  : "+basicPay);
    }
}
```

```
public class Main {
   public static void main(String a[])
   {
      PermanentEmployee pObj=new
PermanentEmployee(102,"Tom",35000);
      pObj.display();  //calls the
                       //overridden method
   }
}
```

Both Employee and PermanentEmployee have display() method. Call to display() method of Employee class from PermanentEmployee class is possible using "super" keyword

Constructor chaining

In a derived class constructor, the base class constructor is invoked, either explicitly or implicitly, there will be a whole chain of constructors called, all the way back to the constructor of Object (parent of all classes). This is called constructor chaining.

```
class Person  {
    public Person()     {
        System.out.println( "In Person  constructor");
    }
}
class Employee extends Person {
    public Employee()   {
        System.out.println("In Employee  constructor");
    }
}

class PermanentEmployee extends Employee {
    public PermanentEmployee() {
        System.out.println("In PermanentEmployee constructor");
    }
}
public class Main{
    public static void main(String a[])
    {
        PermanentEmployee e=new      PermanentEmployee();
    }
}
```

Output :
In Person constructor
In Employee constructor
In PermanentEmployee constructor

final

"final" keyword can be applied with

- variable - to stop value change
- method - to stop Method Overriding
- class - to stop inheriting

Using "final" with a variable

- An instance variable, class variable(static) or a local variable can be made **constant** by declaring it as final.
- A final variable can be assigned a value only once. It will remain unchanged.
- If a final variable holds reference to an object, the state of the object can change, but this variable will always refer to the same object.

final variable

Example:

```
public class  Employee{
   int empId;
   String name;
   final  String  panNo;
   public Employee(int empId,String name,String panNo)
   {
      this.empId = empId;
      this.name = name;
      this.panNo=panNo;
   }
}
```

A variable declared as final and not initialized is called a **blank final variable.**
A blank final variable forces the constructors to initialize it.

final Method

TEKNOTURF

Using “final” with method

- Methods declared as final cannot be overridden

Example:

```
class  Employee{
  final void display()  {
    System.out.println(“In Employee display
                                method”);
  }
}
class PermanentEmployee extends Employee{
  void display()  {
    System.out.println(“In PermanentEmployee
                          display method”);
  }
}
```

```
public class Main {
   public static void main(String a[])
   {
      PermanentEmployee pObj=new
                  PermanentEmployee();
      pObj.display();
   }
}
```

Output :
Compile Time Error

final class

TEKNOTURF

Using “final” with class

- A class declared as final cannot be inherited (extended)

Example:

```
final   class  Employee
{
}

class PermanentEmployee extends Employee
{
}
public class Main {
   public static void main(String a[])  {
      PermanentEmployee pObj=new
PermanentEmployee();
  }}
```

Output :
Compile Time Error

Inbuilt String class is final

Object class

TEKNOTURF

Object class is the cosmic class in java. It is called so because it is the parent of all classes by default.

It is available in java.lang package

There is no super class for Object class

It has some methods common to all the classes. Few methods are:

- getClass()
- equals()
- toString()

"Has A" Relationship - Aggregation

- When there is a "Has A" relationship between two classes, we term that relationship as Aggregation
- Simple Example is Car "has a " Carburetor. Or Carburetor "is a part of " Car.

- Also termed as "whole – part " relationship. The diamond notation points to the whole.
- In Aggregation, when the whole ceases to exist, the part can exist on its own

Aggregation Relationship

```
public class Carburetor  {

        //Attributes, methods
}
```

```
public class  Car {
    private String regnNumber;
    private String model;
    private Carburetor carburetor;

    public Car(String reg,String modl,
                        Carburetor c ) {
        carburetor = c;
    }
    //some code
}
```

```
public class  Main {
    public static void main(String args[])  {
        Carburetor  c = new Carburetor();
        Car car = new Car("S101","Ford",c);

    }
    //some code
}
```

Here even if car is destroyed, that is it is assigned a null value, the Carburetor object still exists.

Composition

- Composition is a stronger form of Aggregation.
- In composition, the part will live and die with the whole.
- Simple Example is Building "has " Floors.

Composition Relationship


```
public class Floor {

        //Attributes, methods
}
```

```
public class  Building {
    private String name;
    private Floor floor;

    public Building(String name) {
        floor = new Floor();
    }

    //some code
}
```

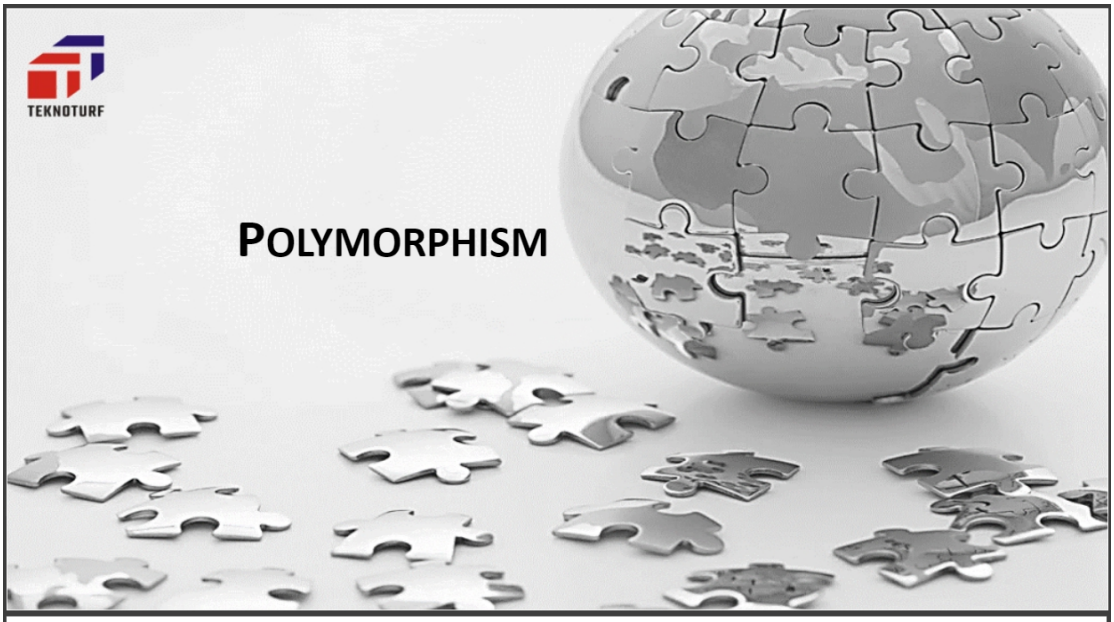
```
public class  Main {
    public static void main(String args[]) {
        Building b  = new Building("MayFlower");

    }
    //some code
}
```

Here building is the owner for creating the floor. Hence if Building is destroyed, that is, it is assigned a null value, the Floor object also gets destroyed

Composition Relationship


```
class Customer
{
//Attributes and methods
}

class Account
{
String accNo;
double balance;
Customer customer;
//Other attributes and methods
}
```

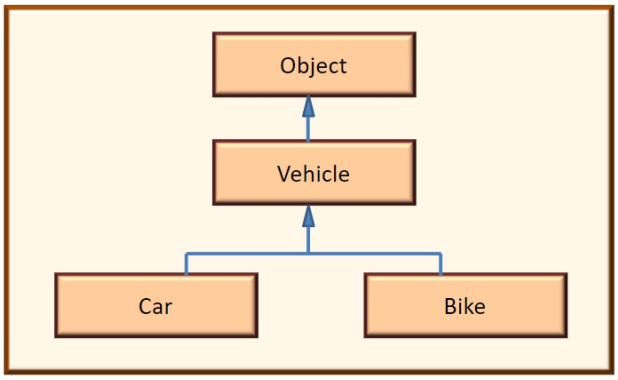
```
class Bank  {
    List<Customer> custList = new
                        ArrayList<Customer>();
    List<Account> accountList = new
                          ArrayList<Account>();

     public void addCustomer(Customer c){
                //some code
     }
      public void  openAccount(String accNo,
                    double balance, Customer c)  {
       Account a = new Account(accNo,balance,c);
      }
}
```



Upcasting and Downcasting

TEKNOTURF

Consider a parent class and a child class. For eg., Vehicle is the parent class and Car and Bike are child classes.



Upcasting and Downcasting

TEKNOTURF

When reference variable of Parent class refers to an object of Child class, it is known as Upcasting

Casting a reference of Parent class to one of its child class is called Downcasting

Upcasting

Objects for these classes can be created as:

```
Vehicle v=new Vehicle();        // Vehicle reference pointing to Vehicle Object

Car c=new Car();                // Car reference pointing to Car  Object

Vehicle v1=new Car();           // Vehicle reference  pointing to Car Object
```

Note that a **parent class reference** can hold **a child class object.**
This we call as ***Upcasting***

By casting, the object is not changed, but it is labeled differently.

Casting of an object of child class to its parent class is called Upcasting. It is done implicitly i.e., done automatically.

Upcasting can be done whenever there is an IS-A relationship.

Upcasting


```
public class Vehicle {

private int power;

public Vehicle(int power) {
    this.power = power;
}
public int getPower() {
    return power;
}
public void setPower(int power) {
     this.power = power;
}
public void display() {
     System.out.println("Vehicle Object");
}
}
```

```
public class Car extends Vehicle {
private int noOfSeats=1;

public Car(int power, int noOfSeats) {
   super(power);
   this.noOfSeats = noOfSeats;
}

public int getNoOfSeats() {
     return noOfSeats;
}
public void setNoOfSeats(int noOfSeats) {
this.noOfSeats = noOfSeats;
}
public void display() {
     System.out.println("Car Object");
}
}
```

Usage of instanceof operator


```
public class Truck extends Vehicle {
   private float load;

public Truck(int power, float load) {
    super(power);
    this.load = load;
}
public float getLoad() {
    return load;
}
public void setLoad(float load) {
    this.load = load;
}
public void display() {
     System.out.println("Truck Object");
}

}
```

```
public class Road
{
public void ride(Vehicle v)
{
   if(v instanceof Car)  {
      System.out.println("Car Object in
                                  ride");
   }
   else if(v instanceof Truck)  {
      System.out.println("Truck Object in
                                  ride");
   }
   else  {
      System.out.println("Vehicle Object
                          in ride");
   }
}
}
```

Upcasting


```
public class MainVehicle {

public static void main(String a[]) {
        Vehicle v1=new Vehicle(100);
        Vehicle v2=new Car(100,5);
        Vehicle v3=new Truck(200,1000);

        v1.display();    //invokes Vehicle's display
        v2.display();    //invokes Car's display
        v3.display();    //invokes Truck's display

        Road r=new Road();
        r.ride(v1);
        r.ride(v2);
        r.ride(v3);
}
}
```

Output

Vehicle Object
Car Object
Truck Object

Vehicle Object in ride
Car Object in ride
Truck Object in ride

Upcasting

When an overridden method is called through a superclass reference, Java determines which version of the method to call, based upon the type of the object being referred to at the time the call occurs.

```
v1.display();
v2.display();
v3.display();
```

When a reference of Vehicle invokes the overridden method display, the method invoked depends on the object invoking that method and not on the reference.

Hence we call method overriding as **dynamic binding.**

Downcasting

To invoke a method that is specific to child class, we need

- a reference of that class or
- Cast the reference of the Parent to the Child class

Casting of parent class reference to that specific child class is known as Downcasting

```
Vehicle v=new Car(100,5);  //Upcasting
```

To invoke the child class specific method, say getNoOfSeats(),

```
Car c= (Car) v;          //Downcasting
c.getNoOfSeats();
```

Usage of instanceof operator

- Downcasting has to be performed manually
- Using instanceof in downcasting:

```
public class Road
{
    public void  displayDetails (Vehicle v)
    {
        if(v instanceof Car)  {
            Car c=(Car)v;
            System.out.println("No Of Seats"  +c.getNoOfSeats());
        }
        else if(v instanceof Truck)        {
            Truck t =(Truck)v;
            System.out.println(" Max  Load "  +t.getLoad());
        }
}
```

OBJECT Class – equals method

- To compare if two objects are equal use the equals() method in Object class:

  public boolean equals(Object o)

- As this method is used by all classes, it takes a reference of Object class as parameter.
- Parent class reference can hold any child class object.

  Object class is the parent of all class. So the parent class reference can hold any child class object.

OBJECT Class – equals method

- Example

```
public class Employee {

    protected  int employeeId;
    protected String name;

    public Employee(int employeeId, String name) {
        this.employeeId = employeeId;
        this.name = name;
    }

    public static void main(String a[]) {
        Employee empObj1 = new Employee(101,"Peter");
        Employee empObj2 = new Employee(102,"Tom");
        Employee empObj3 = new Employee(101,"Peter");
        Employee empObj4 = empObj1;
        System.out.println(empObj1.equals(empObj2)); //false
        System.out.println(empObj1.equals(empObj3));//false
        System.out.println(empObj1.equals(empObj4));//true

    }
}
```

empObj1 and empObj2 hold different objects.

So that comparison returns false.

empObj1 and empObj3 hold different objects that have same values.

But as they hold different address this comparison returns false.

empObj1 and empObj4 both hold the same object. So that comparison returns true.

OBJECT Class – equals method

- When using equals method, if the comparison is to be done for the values held in object and not reference, then override the equals method.
- Override the equals method in the Employee class
- Invoke the equals method for the employee objects.

OBJECT Class – equals method

- Example

```
public class Employee {

    protected  int employeeId;
    protected String name;

    public Employee(int employeeId, String name) {
        this.employeeId = employeeId;
        this.name = name;
    }

    public boolean equals(Object o)
    {
        Employee e = (Employee) o;
        if(e.employeeId == this.employeeId &&
                e.name.equals(this.name))
            return true;
        else
            return false;
    }
    public static void main(String a[]) {
        Employee empObj1 = new Employee(101,"Peter");
        Employee empObj2 = new Employee(102,"Tom");
        Employee empObj3 = new Employee(101,"Peter");
        Employee empObj4 = empObj1;
        System.out.println(empObj1.equals(empObj2)); //false
        System.out.println(empObj1.equals(empObj3));//true
        System.out.println(empObj1.equals(empObj4));//true

    }
}
```

empObj1.equals(empObj3) invokes the overridden equals method in Employee class.

empObj1 is the current object, 'this' and empObj3 is passed as o.

To compare this.employeeId with parameter o.employeeId,

employeeId is specific to child Employee. So, here it needs to do downcasting.

Then perform the necessary comparison and return true or false accordingly.

OBJECT Class – toString method

- Create an object for Employee and print the reference:

  Employee empObj1 = new Employee(101,"Peter");
  System.*out.println(empObj1);*

- This print statement invokes the toString method in Object class.
  The toString method invokes the hashCode method in Object class, which returns an int.
  This toString method converts the hashCode to hexa decimal and prints that value.

- To make this print the employee details, override the toString method
  What needs to be printed needs to be returned as String.

OBJECT Class – toString method

- Example

```
public class Employee {

    protected  int employeeId;
    protected String name;

    public Employee(int employeeId, String name) {
        this.employeeId = employeeId;
        this.name = name;
    }

    public String toString() {
        return "ID "+employeeId+" Name "+name;
    }

    public static void main(String a[]) {
        Employee empObj1 = new Employee(101,"Peter");
        System.out.println(empObj1);
    }
}
```

Output :
ID 101 Name Peter

Polymorphism Overview

Mobile behaves differently in different places. It behaves as Camera, Music Player, Phone, Send Message, Chat, Play games etc.,

One object takes many forms. This we call as polymorphism

In OOP, Polymorphism is the concept in which a variable or a function can take many forms.

Polymorphism

Polymorhpism is a Latin word which means many (poly) forms (morph)

Polymorphism is the capability of a method to do different things based on the object that it is acting upon

Move()

Trotting

Swimming

Flying

Polymorphism

Two types of Polymorphism

- Compile Time Polymorphism / Static Binding
- Run Time Polymorphism / Dynamic Binding

Static Binding is achieved through Method Overloading

Method overloading

- In a class, there is more than one method with the same name, but different parameters.
- Depending on the parameters passed, the corresponding method is invoked. This is decided at Compile time. Hence we call method overloading as compile time polymorphism or static binding.

Polymorphism

Dynamic Binding Is achieved through Method Overriding

Method overriding

- When a method in a sub class has the same name and signature as a method in the super class, then the method in sub class overrides the method in the super class.
- When an overridden method is called through a superclass reference, Java determines which version of the method to call, based upon the type of the object being referred to at the time the call occurs.
- As method overriding depends on the object invoking it, we call this concept as run time polymorphism or dynamic binding.

Abstract class

TEKNOTURF

A class declared as abstract is an abstract class

If a class contains any abstract method it has to be declared as abstract

Syntax : **abstract class class_name**

What is an Abstract Method?

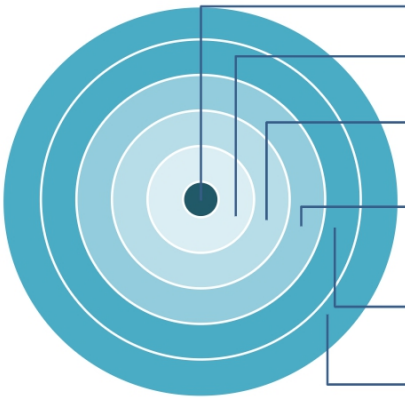
- A method in a class that is just declared without any implementation
- It just has the method signature

  Syntax : abstract return_type function_name(<parameterlist>);

Abstract class can also have attributes and normal methods with implementations along with abstract methods.

Abstract class

TEKNOTURF



Abstract class cannot be instantiated.

Abstract class can be subclassed

When a class inherits an abstract class with abstract methods, it has to provide implementation for all the abstract methods to make it a concrete class

If the subclass does not implement all the abstract methods, then the subclass also needs to be declared abstract

Abstract class can contain non abstract methods also

If an object need not be created for a class, it can be declared as abstract though it does not have any abstract methods.

Abstract classes contains zero or more abstract methods, which are later implemented by concrete classes.
A class cannot be both abstract and final.

Abstract Class

Example

```
abstract public class Employee {
  protected  int empId;
  protected String name;

  public Employee(int empId, String name) {
            this.empId = empId;
            this.name = name;
  }
  public int getEmpId() {
            return empId;
  }
  public void setEmpId(int empId) {
            this.empId = empId;
  }
```

```
public String getName() {
            return name;
  }
  public void setName(String name) {
            this.name = name;
  }
 abstract public float calculateSalary();
}
```

Abstract Class

Example Contd.

```
public class PermanentEmployee extends
                                  Employee {
   private int basicPay;
   public PermanentEmployee(int empId,
                  String name, int basicPay) {
            super(empId, name);
            this.basicPay = basicPay;
   }
   public int getBasicPay() {
            return basicPay;
   }
   public void setBasicPay(int basicPay) {
            this.basicPay = basicPay;
   }
```

```
// Implementation of abstract method
  public  float  calculateSalary() {
      float  da = basicPay * 0.10f;
      float  pf =  basicPay *  0.12f;
      float salary = basicPay+da-pf;
      return salary;
  }
}
```

Abstract Class

Example Contd.

```
public class TemporaryEmployee extends
                          Employee {
      private int noOfHrs;
      private int hrlywages;
      public TemporaryEmployee(int empId,
      String name, int noOfHrs,    int   hrlywages) {
            super(empId, name);
            this.noOfHrs = noOfHrs;
            this.hrlywages = hrlywages;
      }
      public int getNoOfHrs() {
            return noOfHrs;
      }
      public void setNoOfHrs(int noOfHrs) {
            this.noOfHrs = noOfHrs;
      }
```

```
public int getHrlywages() {
      return hrlywages;
}
public void setHrlywages(int hrlywages) {
      this.hrlywages = hrlywages;
}
//Implementation of abstract method
public  float  calculateSalary() {
      float salary = noOfHrs * hrlywages;
      return salary;
   }
}
```

Abstract Class

Example Contd.

```
public class EmployeeUtility {
    public static void main(String a[]) {
        Employee permanentEmployeeObj=new PermanentEmployee(101,"Rohith",20000);
        System.out.println("Salary of "+permanentEmployeeObj.getName()+" is " +
                                        permanentEmployeeObj.calculateSalary());
        Employee tempEmpObj=new TemporaryEmployee(102,"Tom",60,200);
        System.out.println("Salary of "+tempEmpObj.getName()+" is "+
                                        tempEmpObj.calculateSalary());
    }
}
```

Output :
Salary of Rohith is 19600.0
Salary of Tom is 12000.0

Using the parent reference, we can invoke all methods in the parent and the overridden methods in the child. When overridden methods are invoked, which method is invoked depends on the object it holds leading to run time polymorphism.

Abstract Class

- Consider the below method in an abstract class

```
abstract void calculateInsuranceBonus();
```

VALID OVERRIDDEN METHODS

- ☑ `void calculateInsuranceBonus() { }`
- ☑ `protected void calculateInsuranceBonus() { }`
- ☑ `public void calculateInsuranceBonus() { }`

INVALID OVERRIDDEN METHIDS

- ☒ `private void calculateInsuranceBonus() { }`

```
abstract protected void calculateInsuranceBonus();
```

VALID OVERRIDDEN METHODS

- ☑ `protected void calculateInsuranceBonus() { }`
- ☑ `public void calculateInsuranceBonus() { }`

INVALID OVERRIDDEN METHODS

- ☒ `private void calculateInsuranceBonus() { }`
- ☒ `void calculateInsuranceBonus() { }`

Abstract Class

- Consider the below abstract class

```
abstract public class Account {
  int accoutNumber;
  double balance;

  public double getBalance() {
        return balance;
  }
  abstract public double calculateInterest();
  abstract public void displayInterest(float interest);
}
```

VALID OVERRIDDEN METHODS

☑
```
class SavingsAccout extends Account  {

      public double calculateInterest() {
            return balance * 0.02 ;
      }
      public void displayInterest(float interest) {

      }
}
```

INVALID OVERRIDDEN METHODS

☒
```
class SavingsAccout extends Account  {
      public double calculateInterest() {
            return balance * 0.02 ;
      }
}
```

☒
```
class SavingsAccout extends Account  {

      double calculateInterest() {
            return balance * 0.02 ;
      }
      void displayInterest(float interest) {

      }
}
```

OOP Concept - Abstraction In Java

TEKNOTURF

In Java, abstraction is achieved by Abstract class and Interface

An abstract class can have zero or more abstract methods

An instance cannot be created for an abstract class.

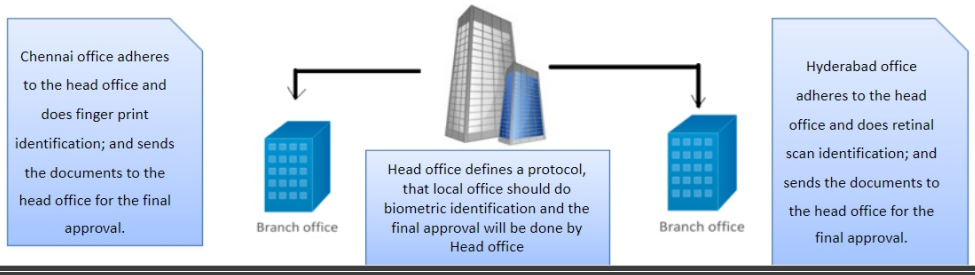
An abstract class can have both concrete methods and abstract methods.

Hence 0 – 100% abstraction is achieved through abstract class concept.

Abstraction- Real world scenario

TEKNOTURF

Let us consider the Aadhaar card issuing process. The head office for the final approval is present in Bangalore and we have two local offices one at Chennai and the other at Hyderabad. Aadhaar card issuing involves 2 steps. Biometric identification and the other is final approval and issuing the acknowledgement. Chennai does the biometric identification through the finger print and Hyderabad does it through retinal scan. So the Bangalore head office has decided to delegate the biometric identification to both the local offices and take care of the final approval and issuing the acknowledgement



OOP Concept - Abstraction In Java

TEKNOTURF

There will be information on what a method will do instead of how it does

```
abstract class BangaloreHeadOffice {

   abstract public void performBiometricIdentification();

   public void approveAadharCard() {
            //some code
   }
}
class ChennaiOffice extends BangaloreHeadOffice {
    public void performBiometricIdentification() {
        System.out.println("Use Finger print");
    }
}
```

```
class HyderabadOffice extends
            BangaloreHeadOffice {
    public void performBiometricIdentification ()
    {
        System.out.println("Use
                retinal scan");
    }
}
```

Here the Bangalore head office knows what is to be done - performBiometricIdentification .
But how it is done is not known. Hence here it is abstract.

OOP Concept - Abstraction In Java

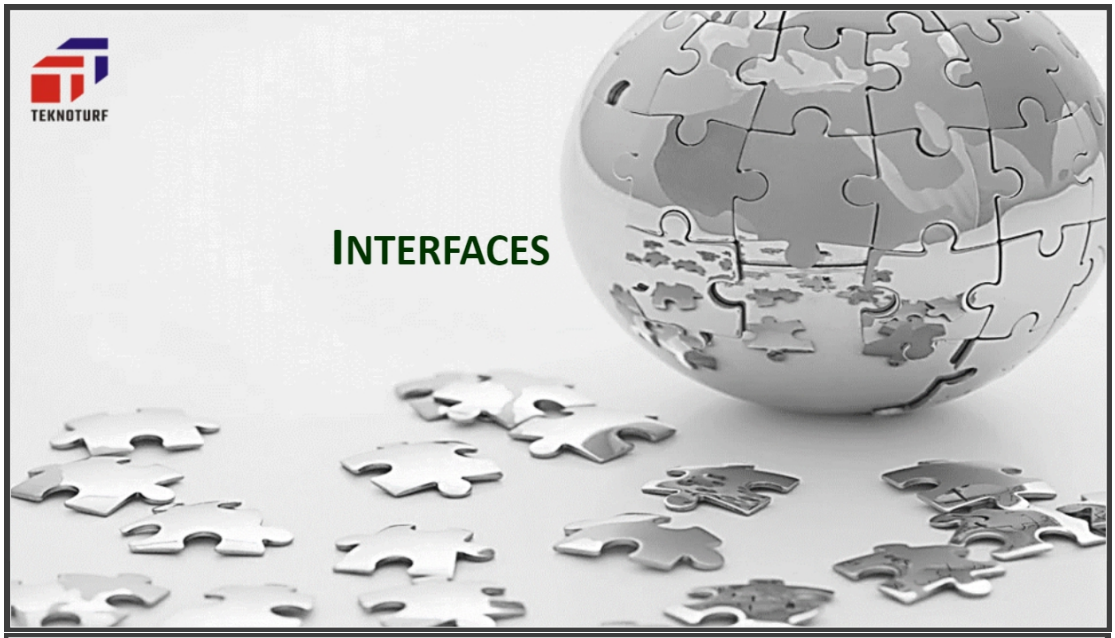
TEKNOTURF

Another Example

```
abstract  public class  Employee {

   //Atributes, Constructor, Getter, Setter

   abstract public float calculateSalary();

   public  void display()  {
                //some code
   }
}
```

```
public class  PermanentEmployee
                    extends  Employee  {
     private float salary;
     private float incentive;

      public float calculateSalary() {
          //sum of  salary and incentive
      }
}
```

Here the Employee class has both abstract and concrete methods.
This class knows what the calculateSalary method should do – return the total salary. But how it is done is unknown.



Interfaces

TEKNOTURF

In software engineering, there are scenarios where disparate groups of developers agree to a contract that describes what is expected from the product.

Each group writes their code to meet the contract not bothering how other groups code.

Interfaces are such contracts.

Interfaces

TEKNOTURF

RBI fixes the base interest for Home Loan.

All banks calculate the interest for home loan based on some specific factors of theirs.

Here RBI acts as an interface with the method calculateHomeLoanInterest as abstract

All banks implement this interface and give implementation for this abstract method

Interfaces in java

TEKNOTURF

An interface is a “contract” which an implementing class must adhere to

Interface is a 100% abstract class

Interface can contain variables and methods.

All methods that are just declared in an interface are by default public abstract
All variables in an interface are implicitly public static final

Interfaces cannot be instantiated—they can only be implemented by classes or extended by other interfaces

Defining an Interface

TEKNOTURF

An interface declaration consists of

- Modifiers
- The keyword interface
- The interface name
- A comma-separated list of parent interfaces
- The interface body

Defining an interface

TEKNOTURF

```
public interface GroupedInterface extends Interface1, Interface2, Interface3{

        //  Constant Declarations
        //base of natural logarithms
        double E = 2.718282;     //implicitly public static final

        //Method signatures  -  implicitly all methods that are just declared are public
        //and abstract
        void operation1(int x,double d);
         int operation2(String s);
}
```

Note :
One interface can extend many interfaces.

Implementing the interface

- To declare a class that implements an interface, include an implements clause in the class declaration.
- Class Implementing the interface need not have an "is-a " relationship.
- A class can implement more than one interface, so the implements keyword is followed by a comma-separated list of the interfaces implemented by the class.
- By convention, the implements clause follows the extends clause, if there is one.

Create an Interface

Use the keyword interface instead of class

```
public interface Shape {

      //Methods just declared without implementation,
      //are by default public abstract
            float calculateArea();
            float calculatePerimeter();
}
```

Implement the Interface

- An interface will be implemented by a class using the keyword "implements"
- This class should implement all the methods in the interface. Otherwise, that class becomes abstract
- Implementing class can have its own methods
- Implementing class can also extend only one super class or abstract class

Implement the Interface

Example:

```
public class Square implements Shape {
    private float side;
    public Square(float side) {
        this.side = side;
    }
    public float calculateArea() {
        float area=side * side;
        return area;
    }
    public float calculatePerimeter() {
        float perimeter=4*side;
        return perimeter;
    }
}
```

```
public class Rectangle implements Shape {
    private float length;
    private float breadth;
public Rectangle(float length, float breadth) {
    this.length = length;
    this.breadth = breadth;   }
    public float calculateArea() {
        float area=length * breadth;
        return area;          }
public float calculatePerimeter() {
    float perimeter=2*(length+breadth);
    return perimeter;
}
}
```

Using an interface as a Type

When a new interface is defined, a new reference data type is defined

Interface cannot be instantiated, but a reference of interface can be created

If you define a reference variable whose type is an interface, any object you assign to it must be an instance of a class that implements the interface.

```
public class ShapeUtility  {
   public static void main(String a[])  {
        Shape s=new Square(10);
        System.out.println("Area of square "+s.calculateArea());
        System.out.println("Perimeter of square "+s.calculatePerimeter());
        s=new Rectangle(2.5f,12);
        System.out.println("Area of Rectangle "+s.calculateArea());
        System.out.println("Perimeter of Rectangle "+s.calculatePerimeter());
  } }
```

Implement the Interface

- One class can implement multiple interfaces.
- If so, class should implement abstract methods in all the interface it implements.
- Example

```
interface BlueTooth  {
    void sendFile();
    void receiveFile();
}

interface Phone {
    void makeCall();
    void attendCall();
}

interface WhatsApp {
    void sendMessage();
    void receiveMessage();
}
```

```
class SmartPhone implements BlueTooth,
                        Phone, WhatsApp  {
   //Implement all methods in the interface
}
```

```
class iPhone extends SmartPhone
            implements BlueTooth, WhatsApp  {

}
```

Extending the Interface

Interfaces can be extended to include additional contracts

```
public interface Loan{

   void operation1(int x,double d);
   int operation2(String s);
}
```

```
public interface HomeLoan extends Loan{

   void operation3();
   float operation4(float f);
}
```

Extending the Interface

One interface can extend multiple interfaces

```
interface BlueTooth  {
    void sendFile();
    void receiveFile();
}

interface WhatsApp {
    void sendMessage();
    void receiveMessage();
}

interface Mobile extends BlueTooth, WhatsApp  {
    void makeCall();
    void attendCall();
}
```

Any class that implenents Mobile should implement the methods in Mobile, BlueTooth and WhatsApp

default methods in Interface

Interfaces can have default method - Java 8 Feature

Default Methods are

- Methods defined in an interface with the default keyword
- Must be implemented

Classes that implement that interface can

- either override this default method
- or use the method as such

This leads to backward compatibility

default methods in interface


```
public interface  Insurance
{
    void applyInsurance();
    void approveInsurance();
    double claimInsurance();

    public default void display()
    {
       System.out.println("In default
                    method display");
    }

}
```

```
public class Car implements Insurance
{
   public  void applyInsurance()  { }
   public void approveInsurance()  { }
   public  double claimInsurance()  { }

 public static void main(String args[])
 {
      Car c=new Car();
      c.display(); //similar to invoking
                   // Insurance.display()
 }

}
```

default methods in interface


```
public interface  Insurance
{
    void applyInsurance();
    void approveInsurance();
    double claimInsurance();

    public default void display()
    {
       System.out.println("In    Insurances
                   default method display");
    }

}
```

```
public interface Vehicle
{
   public  void getColor() ;
   public void drive() ;

    public default void display()
    {
       System.out.println("In Vehicles
            default method   display");
    }
}
```

default methods in interface


```
public class Car implements Vehicle,   Insurance   //DOES NOT COMPILE
{
   public  void applyInsurance()  { }
   public void approveInsurance()  { }
   public  double claimInsurance()  { }
   public  void getColor() { }
   public void drive() { }

 public static void main(String args[])
 {
      Car c=new Car();
      c.display();
 }
}
```

default methods in interface

```
public class Car implements Vehicle,   Insurance   // COMPILES
{
   public  void applyInsurance()  { }
   public void approveInsurance()  { }
   public  double claimInsurance()  { }
   public  void getColor() { }
   public void drive() { }

    //Overrides
   public void display()    {
      System.out.println("In Car display method ");
    }
 public static void main(String args[]) {
       Car c=new Car();
       c.display();
 }
}
```

default methods in interface

Rules for default methods in an interface

- A default method can be declared only within an interface.
- A default method must be marked with the default keyword and should have a method body.
- A default method is not assumed to be static, final, or abstract.
- It may be used or overridden by a class that implements the interface.
- A default method is public by default.
- Code will not compile if it is marked as private or protected.

static methods in interface

- Interface can have static methods
- These static methods should be implemented

```
public interface StudentService{
        public Student getStudent(int studentId);

        static public int getStudentCount()
        {
                //logic to find the count
                return 0;
        }
}
```

static methods in interface

```
public class Hosteller implements StudentService{
        public void getStudentInfo(int studentId) {
                //search and return the student info

        }
        public static int getStudentCount() {
                // return count of student
                return 0;
        }
}
```

static methods in interface

A static method in an interface can be invoked only by using the interface name and not by using the reference of the interface

```
public class StudentTest {
    public static void main(String[] args) {
        // Legal - calls the static method in the CustomerService interface
            int custCount = StudentService.getStudentCount();

        // Legal - calls the static method in the MidwestCustomerService interface
            int custCount2 = Hosteller.getStudentCount();
        StudentService ss = new Hosteller();

        // Cannot access a static method defined in an interface using instance variable

            ss.getStudentCount(); //Compilation error.
            System.out.println(StudentService.getStudentCount());//Compiles
            System.out.println(Hosteller.getStudentCount());//Compiles
        }
}
```

OOP Concept - Abstraction In Java

In Java, 100% abstraction is achieved through Interface

Unlike abstract class, no attribute or constructor or instance methods are there in an interface. All methods declared in it are abstract and public.

An instance cannot be created for an interface.

Hence 100% abstraction is achieved through interface in java.

OOP Concept - Abstraction In Java

TEKNOTURF

Example

```
public interface Loan
{
    public double issueLoan();
    public  void repayLoan(double amt);
}
```

Loan interface knows what the methods issueLoan and repayLoan should do. But how it is done is implemented by the class that implements the interface.

As all methods are abstract, 100% abstraction is achieved by interface.

```
public class  Employee implements Loan {
    //Attributes, constructor, methods

     public double issueLoan()  {
          //some logic
     }

     public  void repayLoan(double amt)  {
          //some logic
     }
}
```

Functional Interfaces

TEKNOTURF

Functional interfaces are interfaces which have only one abstract method in it.

This is a Java 8 feature.

These interfaces are also called Single Abstract Method interfaces (SAM Interfaces)

These interfaces can be annotated as @FunctionalInterface, which is optional.

@FunctionalInterface can be used for compiler level errors.

A Functional interface

- can have only one abstract method
- but it can have default method and static method

Functional Interfaces

TEKNOTURF

In a functional interface, the single abstract method gets matched up to the lambda expression if it has a compatible method signature.

For parameters and return type, generics can also be used.

Few Functional interface before Java 8

```
public interface Comparator <T> {
    public int compare(T o1, T o2);
}

public interface Runnable {
    public void run();
}
```

Functional Interfaces - Example


```
@FunctionalInterface
interface MaxFinder
{
  public int maximum(int num1,int num1);
}
public class MaxFinderImpl implements MaxFinder
{
     public int maximum(int num1, int num1) {
          return num1 > num2 ? num1 : num2;
     }
     public static void main(String a[])   {
          MaxFinder max= new MaxFinderImpl();
          int res = max.maximum(20,30);
      }
}
```

Functional Interfaces –Example

Same Example Using Lambda Expression

```
@FunctionalInterface
interface MaxFinder
{
  public int maximum(int num1,int num2);
}
public class MaxFinderImpl
{
     public static void main(String a[])   {
          MaxFinder max= (num1,num2) ->  num1 > num2 ? num1 : num2;
          int res = max.maximum(20,30);
      }
}
```

Lambda Expression

Why Lambdas?

Enables functional programming. Lambda expressions are used to pass Code as Data.

Avoid boilerplate code - **boilerplate code** or **boilerplate** refers to sections of **code** that have to be included in many places with little or no alteration

Enables support for parallel processing

Lambda expression avoids unwanted anonymous inner class

Syntax

(argument-list) -> {body}

argument-list: It can be empty or non-empty as well.

arrow-token: It is used to link arguments-list and body of expression.

body: It contains expressions and statements for lambda expression.

Lambda Expression

- Using lambda expression we can assign a block of code to a variable and we can pass the variable around the application where the code is needed.
- For example

```
aBlockOfCode = public void greet() {
System.out.println("Hello World");
}
```

Lambda Expression

- ❖ After removing the unwanted things the code is left with

```
aBlockOfCode = ()-> {
System.out.println("Hello World"); }
```

- ❖ '{}' is optional if the block of code is just a line.

```
aBlockOfCode = ()-> System.out.println("Hello World");
```

- ❖ Note : What is the type of aBlockOfCode?

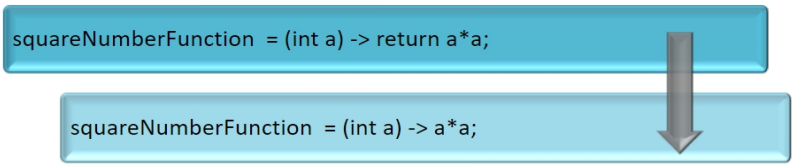
Lambda Expression

```
interface MyLambda
{
void display();
}
        MyLambda  aBlockOfCode = () ->
System.out.println("Hello World");
        aBlockOfCode.display(); // display 'Hello World'
```

- The variable assigned with Lambda expression can be passed around the application in order reuse the code. Or, we use inline lambda expression.
- Ways of calling a function ***firstLambda***
  - firstLambda(aBlockofCode);
  - firstLamda(()-> System.out.println("Hello World")); //using inline lambda expression.
- Signature of ***'firstLambda'*** method.

```
return_type firstLambda(MyLambda varibleName)
{
}
```

Example – Lambda expression




```
interface Mylambda2
{
int squareFunction(int a);
}
```

- Mylambda2 squareNumberFunction = (a) -> a*a;

Example – Lambda expression

```
Interface MyLambda3 // define functional interface
{
int divideNumber(int a, int b);
}

//defining lambda expression
MyLambda3 numberDivideFunction = ( x, y) -> {
if(y==0) return 0;
return x/y;
}

//calling method using lambda variable
numberDivideFunction.divideNumber(20,5);
```

Lambda Expression

Before Lambda

```
interface MathOperation{
    public int add(int a,int b);
}
    public class Interfacedemo {
        public static void main(String a[])   {
          MathOperation math=new MathOperation() {
          @Override
          public int add(int a, int b) {
                              return a+b;
          }
          };
        System.out.println("The result is "+math.add(5, 10));
    }
}
```

After Lambda

```
interface MathOperation
{
    public int add(int a,int b);
}

public class Interfacedemo {
    public static void main(String arg[]){
        MathOperation sum = (a,b)->a+b;
        System.out.println(sum.add(20, 15));
    }
}
```

Functional Interface and Lambda

Functional Method	Lambda Expression
int fun(int a)	(int a) -> a+10;
int fun(int a,int b)	(int a,int b) -> a+b;

int fun(int a,int b)	(int a,int b) -> {
	int min = a>b ? a : b;
	return min; }

String fun()	() -> “Hello World”
void fun()	() -> System.out.println(“Hello World”);
int fun(String str)	(String s) -> s.length() Or S -> s.length()

Built-in Functional Interfaces

Interface Name	Method Name	Arguments	Returns	Example
Predicate<T>	test	T	boolean	Has this album been released yet ?
Consumer<T>	accept	T	void	Printing a value
Function<T,R>	apply	T	R	Get the value of an attribute from an Object
Supplier<T>	get	None	T	A factory method
UnaryOperator<T>	apply	T	T	Logical not (!)
BinaryOperator<T>	apply	T , T	T	Multiply two numbers

Built – in functional interfaces

TEKNOTURF

PREDICATE - boolean test(T t)

Predicate is a Functional Interface that can be used anywhere you need to evaluate a boolean condition. The test method evaluates a predicate on the given argument that results in true or false..

```
Predicate<String> panCardValid = str ->
                str.matches("[A-Z]{5}[0- 9]{4}[A-Z]{1}");
String panCardNo = "ANCZE1234Q";

boolean result   =   panCardValid.test(panCardNo);

System.out.println(result);

if("Coasta Ri9955".matches("[A-Za-z ]+")){valid = true;}
System.out.println(valid);
```

FUNCTION – R apply(T t)

Function is a Functional Interface defined with generic types T and R. The method apply that takes an argument of type T and returns a result of type R.

```
Function<Integer,Integer> function =  (x) -> x*10;

System.out.println( function.apply(15) ); //150
System.out.println(function.apply(20));  //200
```

Built – in functional interfaces

TEKNOTURF

CONSUMER : void accept(T t)

Consumer is a Functional Interface that has a method accept, that takes a single input as argument and returns no result. It performs some operation on the passed argument.

```
Consumer consumer  =  str -> System.out.println(str);
consumer.accept("Welcome");

Student studentObj = new Student(101,"Peter");
consumer.accept(studentObj);
```

SUPPLIER : T get()

Supplier is a Functional Interface that does the opposite of the Consumer. It takes no arguments but returns some value of type T.

```
Supplier<String> supplier = ()->  {
    Scanner sc=new Scanner(System.in);
     String name=sc.next();
      return name;
  } ;

System.out.println(supplier.get());
```

Supplier functional interfaces

TEKNOTURF

```
public class Student {

    private int id;
    private String name;
    private String gender;
    private int age;

    public Student(int id, String name, String gender, int age) {
            this.id = id;
            this.name = name;
            this.gender = gender;
            this.age = age;
    }
     //Assume 4 argument Constructor and Getters and Setters
     //are written

       @Override
        public String toString() {
            return "Student ID "+id+ ", Name "+name+" , Gender " +
                                                gender + ", Age "+age;
        }
    }
```

```
public class FunctionalInterfaceDemo {

    public static void main(String[] args) {

        Supplier<Student> supplier = () -> {
            Scanner sc = new Scanner(System.in);
            int id=sc.nextInt();
            String name=sc.next();
            String gender=sc.next();
            int age=sc.nextInt();

            return new Student(id,name,gender,age);
        };

        Student[] studentList = new Student[3];
        for(int i=0; i<3; i++){
            System.out.println("Enter Student "+(i+1)+" Details");

            studentList[i] = supplier.get();
        }

        for(Student student : studentList)
            System.out.println(student);
    }
}
```

Supplier functional interfaces

TEKNOTURF

```
public class FunctionalInterfaceDemo {

    public static void main(String[] args) {

        Supplier<Student> supplier = () -> {
            Scanner sc = new Scanner(System.in);
            int id=sc.nextInt();
            String name=sc.next();
            String gender=sc.next();
            int age=sc.nextInt();

            return new Student(id,name,gender,age);
        };

        Student[] studentList = new Student[3];
        for(int i=0; i<3; i++){
            System.out.println("Enter Student "+(i+1)+" Details");

            studentList[i] = supplier.get();
        }

        for(Student student : studentList)
            System.out.println(student);
    }
}
```

OUTPUT

```
Enter Student 1 Details
101
Peter
Male
25
Enter Student 2 Details
102
Pinky
Female
18
Enter Student 3 Details
103
Raghu
Male
19
Student ID 101, Name Peter , Gender Male, Age 25
Student ID 102, Name Pinky , Gender Female, Age 18
Student ID 103, Name Raghu , Gender Male, Age 19
```

Consumer functional interfaces

TEKNOTURF

```
public class FunctionalInterfaceDemo {

    public static void main(String[] args) {

      //Continuation of  previous main method

          System.out.println("Student Details using Consumer");

     //Using Consumer to print the Student object
        Consumer consumer = (value) -> System.out.println(value);

          for(Student studentObj : studentList)  {

                    consumer.accept(studentObj);

          }

    }
}
```

OUTPUT

```
Enter Student 1 Details
101
Peter
Male
25
Enter Student 2 Details
102
Pinky
Female
18
Enter Student 3 Details
103
Raghu
Male
19
Student Details using Consumer
Student ID 101, Name Peter , Gender Male, Age 25
Student ID 102, Name Pinky , Gender Female, Age 18
Student ID 103, Name Raghu , Gender Male, Age 19
```

Predicate functional interfaces

TEKNOTURF

```
public class FunctionalInterfaceDemo {

    public static void main(String[] args) {

    //Continuation of  previous main method

    System.out.println("Student of age below 21");

    Predicate<Student> predicate = (student)->student.getAge()<=20;

    for(Student studentObj : studentList){

            if(predicate.test(studentObj))
                    System.out.println(studentObj);
    }
  }
}
```

OUTPUT

```
Enter Student 1 Details
101
Peter
Male
25
Enter Student 2 Details
102
Pinky
Female
18
Enter Student 3 Details
103
Raghu
Male
19
Student of age below 21
Student ID 102, Name Pinky , Gender Female, Age 18
Student ID 103, Name Raghu , Gender Male, Age 19
```

Function Functional Interface

TEKNOTURF

```
public class FunctionalInterfaceDemo {

    public static void main(String[] args) {

     //Continuation of  previous main method

     System.out.println("Name of students whose age is less than "
                                              + "or equal to 20");
     //Using Function

     Function<Student,String> function = (student)->
                                          student.getName();

     for(Student student : studentList) {
            System.out.println(   function.apply(student)  );

     }
   }
}
```

OUTPUT

```
Enter Student 1 Details
101
Peter
Male
25
Enter Student 2 Details
102
Pinky
Female
18
Enter Student 3 Details
103
Raghu
Male
19

Name of students whose age is less than or equal to 20
Pinky
Raghu
```

Example - LAMBDA

TEKNOTURF

```
interface Calculator
{
        public float calculate(float a,float b);
}
```

```
public class Main
{
    public static Calculator performCalculation(char c){
        Calculator calc=null;
        switch(c)
        {
        case '+' : calc = (a,b)->a+b;
        case '-' : calc = (a,b)->a-b;
        case '*' : calc = (a,b)->a*b;
        case '/' : calc = (a,b)->a/b;
        }
        return calc;
    }
```

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

        Calculator c = (a,b) -> a+b; // Directly

        //By calling the function performCalculation

        Calculator add = performCalculation('+');
        Calculator diff = performCalculation('-');
        Calculator pdt = performCalculation('*');
        float f1 = c.calculate(5,6);

        float x=4.5f; float y=2.5f;
        float f2 = add.calculate(x,y);
        float f3 = c.calculate(x,y);
        float f4 = c.calculate(x,y);
        System.out.println(f1+" "+f2+" "+f3+" "+f4);
        //11.0 1.8 7.0 7.0
    }
}
```