GUJARAT TECHNOLOGICAL UNIVERSITY

MASTER OF COMPUTER APPLICATIONS

SEMESTER: III

Subject : **4639302**

Programming in JAVA

UNIT - 2

- Objects and Classes,
- Inheritance,
- Interface

Chapter-4

Object and Classes

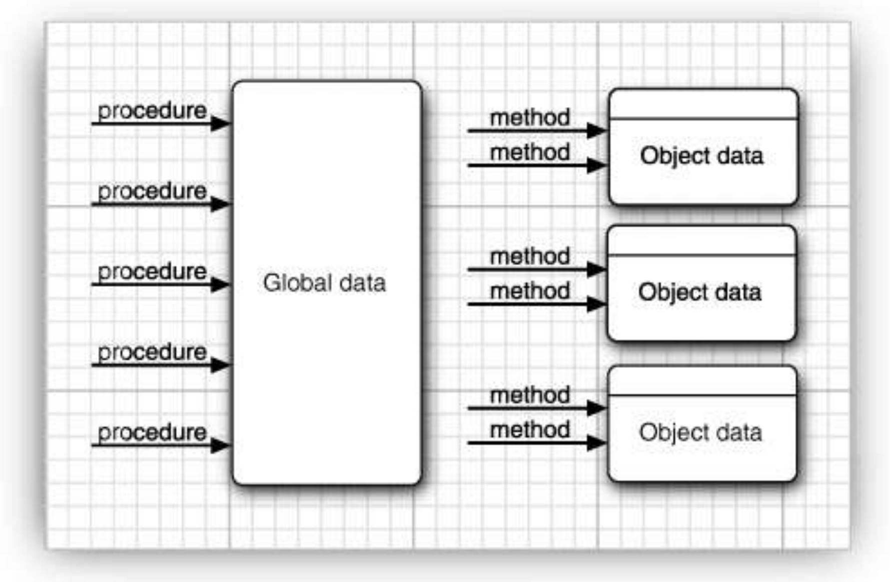
Why OOP?

1970s: “Structured” programming.

- Algorithms + Data Structures = Programs.
- Procedures operate on shared data.

1980s: Object-oriented programming.

- Each object has data and methods.
- Encapsulation: Only methods can access object data.



Java is thoroughly object-oriented.

- Everything other than a primitive type value is an object.

Classes and Objects

- **Class:**
  - ❑ Class is a blueprint of objects.
  - ❑ Describes object data and method behavior.
- **Object:**
  - ❑ Object is instance of class.
  - ❑ Object has:
    - Behavior
    - State
    - Identity
- OOP starts with identifying classes:
- Nouns are often classes: Item, Order, and so on.
- Verbs are often methods: add an item to an order, ship an order.

Object Variables

- An object variable holds a reference to an object.
- Copying a variable makes a copy of the reference:

```
deadline = birthday;
```

- A null reference refers to no object:

```
deadline = null;
```

- Caution: Don't call a method on null.

```
if (deadline != null)
    s = deadline.toString();
```

LocalDate class

- A Date is a point in time, measured in UTC.
- A LocalDate is a date (day, month, year) in a particular location.
- **Use factory methods to create instances:**

```
LocalDate rightNow = LocalDate.now();
LocalDate newYearsEve = LocalDate.of(1999, 12, 31);
```

- **LocalDate methods:**

```
LocalDate aThousandDaysLater = newYearsEve.plusDays(1000);
year = aThousandDaysLater.getYear(); // 2002
month = aThousandDaysLater.getMonthValue(); // 09
day = aThousandDaysLater.getDayOfMonth(); // 26
```

- How is a LocalDate stored? How do these methods do their job? You don't know, and you don't care. That's encapsulation.
- **import java.time.*;**

Accessor and Mutator Methods

- Accessor method doesn't modify object state.
- All LocalDate methods are accessors.
- Older version of calendar date class has mutator methods:

```
GregorianCalendar someDay = new GregorianCalendar(1999, 11, 31);
someDay.add(Calendar.DAY_OF_MONTH, 1000);
   // someDay has been mutated
int year = someDay.get(Calendar.YEAR); // 2002
```

- Tip: Minimize mutator methods. They make it more difficult to share objects in concurrent programs.

Defining Your Own Classes

```
class Employee {
   // Fields
   private String name;
   private double salary;
   private LocalDate hireDay;

   // Constructors
   public Employee(String n, double s, int year, int month, int day) {
      name = n;
      salary = s;
      hireDay = LocalDate.of(year, month, day);
   }

   // Methods
   public String getName() { return name; }
   . . .
}
```

Constructor

There are three types of constructors.

- **Default Constructor**
  - ❑ Compiler automatically creates one constructor if we don’t create
- **Parameterized Constructor**
  - ❑ Constructor with parameter which make possible to initialize objects with different set of values at time of their creation
- **Copy Constructor**
  - ❑ Constructor which creates a new object using an existing object of the same class and initializes each instance variable of newly created object with corresponding instance variables of the existing object passed as argument

Constructors

- In general, instance fields are private.
- **Initialized in constructor:**

```
public Employee(String n, double s, int year, int month, int day) {
   name = n;
   salary = s;
   hireDay = LocalDate.of(year, month, day);
}
```

- **The call**
  - new Employee("James Bond", 100000, 1950, 1, 1)
- **sets the fields as follows:**
  - name = "James Bond";
  - salary = 100000;
  - hireDay = LocalDate.of(1950, 1, 1);
- Name of constructor = class name.
- Constructor only works with new.

Implicit and Explicit Parameters

- Methods access and modify fields:

```
public void raiseSalary(double byPercent) {
  double raise = salary * byPercent / 100;
  salary += raise;
}
```

- In the call **number007**.raiseSalary(**5**), these steps occur:

```
double raise = number007.salary * 5 / 100;
number007.salary += raise;
```

- The call depends on two parameters:
  - **byPercent is an explicit parameter.**
  - **The object on which the method is invoked is the implicit parameter.**
- Can optionally use this to denote implicit parameter:

```
public void raiseSalary(double byPercent) {
  double raise = this.salary * byPercent / 100;
  this.salary += raise;
}
```

Benefits of Encapsulation

- Note the private field and public method:
  - `private String name;`
  - `public String getName() { return name; }`
- **Benefit 1:** The field is “read-only”.
- **Benefit 2:** The internal representation can evolve:

```
private String firstName;
private String lastName;
public string getName() {
        return firstName + " " + lastName;
}
```

Final Fields

- A final field cannot change:

```
private final String name;
```

- Caution: A final object can still be mutated:

```
private final StringBuilder evaluations;
public Employee() {
    evaluations = new StringBuilder(); . . . }
public void giveGoldStar() {
    evaluations.append("Gold star!\n"); }
```

Static Fields

- **A static field exists one copy per class:**

```
private static int nextId; // one field per class
private int id; // one field per object
public void setId() { id = nextId; nextId++; }
```

- **A static final field is a shared constant:**

```
public class Math
{
  public static final double PI = 3.14159265358979323846;
    // Accessible anywhere as Math.PI
  . . .
}
```

Static Methods

- A static method doesn't operate on objects.
- Example: Math.pow(a, b) computes a^b without using a Math object.
- In other words, a static method uses no **this.**
- Static methods can only access static fields:

```
public static int getNextId()
{
   return nextId; // returns static field
}
```

- Supply class name when calling the method:

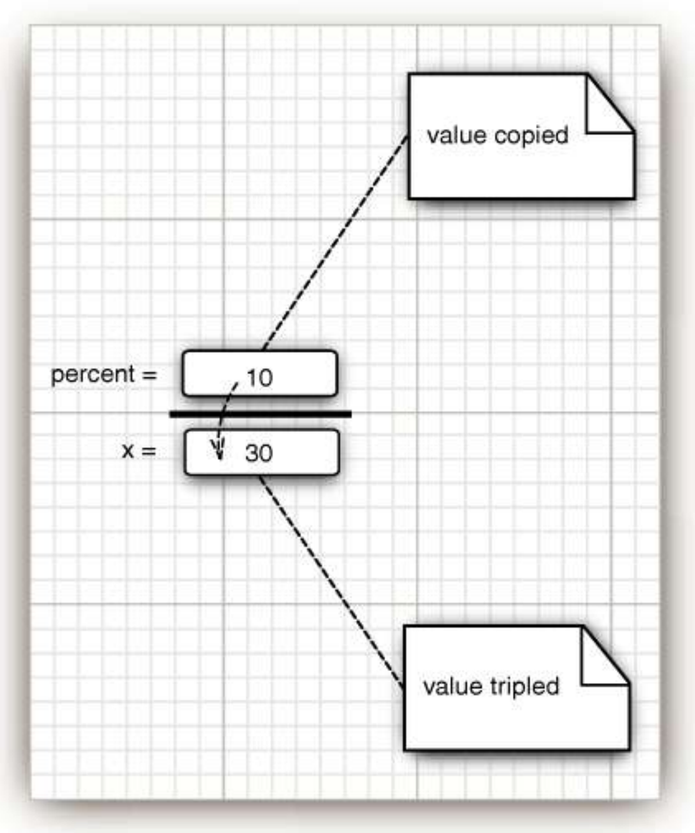
```
int n = Employee.getNextId();
```

- The main method is static because no objects have been constructed when the program starts.

method parameters

- Call by Value
- Call by Reference

Call by value:

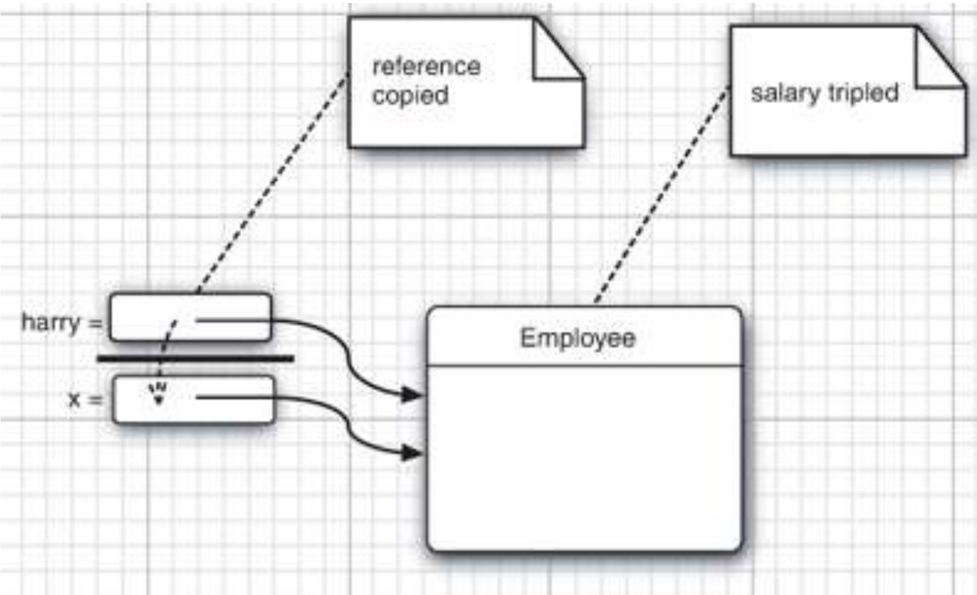


- The method gets copies of the argument values.
- A method cannot change the contents of variables passed to it.
- Example:

```
public static void
tripleValue(double x)
{
    x = 3 * x;
}
```

- In the following call, the percent variable is not changed:

Call with Object References



- A method can mutate objects:

```
public static void
tripleSalary(Employee x)
{
  x.raiseSalary(200);
}
```

- In the following call, the salary is changed:
- harry = new Employee(. . .);
- tripleSalary(harry);

Object References Are Passed by Value

- Some people say: “In Java, numbers are passed by value and objects are passed by reference.”
- That's nonsense. In Java, everything is passed by value.
- Object references are passed by value.
- If objects were passed by reference, you could swap them:

```
public static void swap(Employee x, Employee y) // doesn't work
{
  Employee temp = x;
  x = y;
  y = temp;
}
```

- But in the following call, a and b are not swapped:

```
Employee alice = new Employee("Alice", . . .);
Employee bob = new Employee("Bob", . . .);
swap(alice, bob);
```

object construction

- Overloading
- Default Field Initialization
- The Constructor with No Arguments
- Explicit Field Initialization
- Parameter Names
- Calling Another Constructor
- Initialization Blocks
- Object Destruction and the **finalize** Method

Overloading

- A class can have more than one constructor:
  - ❑ StringBuilder messages = new StringBuilder();
  - ❑ StringBuilder todoList = new StringBuilder("To do:\n");
- The constructor name is overloaded.
- Name + parameter types = Method signature.
- Overloading resolution: The compiler picks the appropriate version from the argument types.
- You can overload any method:
  - ❑ String.indexOf(int)
  - ❑ String.indexOf(int, int)
  - ❑ String.indexOf(String)
  - ❑ String.indexOf(String, int)
- The return type is not a part of the method signature.

Default Construction

- A field that isn't explicitly set in a constructor is 0, false, or null.
- Caution: Accidentally uninitialized variables can lead to null pointer errors.

```
public Employee() { name = ""; }
. . .
LocalDate h = harry.getHireDay();
int year = h.getYear();
```

- If a class has no constructor, a no-argument constructor is provided.
- It sets all fields to their default values.
- If a class has at least one constructor, the no-argument constructor is not provided.
- But you can provide it:

```
    public Employee() {}
```

Field Initialization

- You can override the 0/false/null default for fields:

```
class Employee
{
  private String name = "";
  . . .
}
```

- The initialization value can be computed:

```
class Employee
{
  private static int nextId;
  private int id = assignId();
  . . .
  private static int assignId()
  {
    int r = nextId;
    nextId++;
    return r;
  }
  . . .
}
```

Construction Parameter Names

```
public Employee(String name, double salary)
{
   this.name = name;
   this.salary = salary;
}
```

Calling Another Constructor

- A constructor can call another constructor in the first statement.
- Use this (and not the class name) for the call:

```
public Employee(double s)
{
  // calls Employee(String, double)
  this("Employee #" + nextId, s);
  nextId++;
}
```

- Allows you to factor out common construction code.
- Keyword reuse: Not related to using this for the implicit parameter.

Initialization Blocks

```
class Employee
{
   private static int nextId;
   private int id;
   // object initialization block
   {
      id = nextId;
      nextId++;
   }
   static
   {
      Random generator = new Random();
      nextId = generator.nextInt(10000);
   }
   public Employee(. . .) { . . . } // constructor
   . . .
}
```

- Class declarations can contain arbitrary blocks of code.
- Executed whenever an object is constructed:
- Static initialization block is executed when class is loaded:

Packages

- Related classes are organized into packages:
  - ❑ java.lang
  - ❑ java.util
  - ❑ java.time
- Avoids name conflict:
  - ❑ java.util.Date ≠ java.sql.Date
- Use reverse domain name for your own packages: com.bipinrupadiya.corejava

Imports

- **Can access classes from any package with fully qualified name:**
  - java.time.LocalDate today = java.time.LocalDate.now();
- **Import statements remove the tedious repetition:**
  - import java.time.*;
  - LocalDate today = LocalDate.now();
- **Can import single class:**
  - import java.time.LocalDate;
- **Cannot have multiple wildcards (import java.*.*).**
- **If two packages import the same class, you still need fully qualified names:**
  - import java.util.*;
  - import java.sql.*;
  - Date today; // Error--java.util.Date or java.sql.Date?

Static Imports

- **Imports static fields and methods:**

```
import static java.lang.System.*;
```

- **Now you can refer to System.out and System.exit without the class name:**

```
out.println("Goodbye, World!"); // i.e., System.out
exit(0); // i.e., System.exit
```

- **Can import a specific method or field:**

```
import static java.lang.System.out;
```

- **Can be handy for mathematical functions:**

```
import static java.lang.Math.*;
r = sqrt(pow(x, 2) + pow(y, 2));
```

Adding a Class to a Package

- Put a package declaration at the top of the file:

```
package corejava;
public class Employee
{
   . . .
}
```

- A class without a package declaration is in the default package.
- Place the source file into a subdirectory that matches the package name.
- Compile from the base directory:

```
javac corejava/Employee.java
```

The Class Path

- Class path=list of directories and JAR files.
- JAR file=zip file containing class files.
- Directories are base directories, containing package directories (such as com/horstmann/corejava).
- Class path elements are separated by : (Unix) or ; (Windows).
- Can include current directory as .
- Can specify all JAR files in a directory as directory/*

  Caution: In Unix, must escape * as '*' or *
- Pass to javac/java with -classpath option:

  ```
  java -classpath /home/user/classdir:.:/home/user/archives/archive.jar MyProg
  ```
- Or set CLASSPATH environment variable:

  ```
  export CLASSPATH=/home/user/classdir:.:/home/user/archives/archive.jar
  ```

Chapter-5

Inheritance

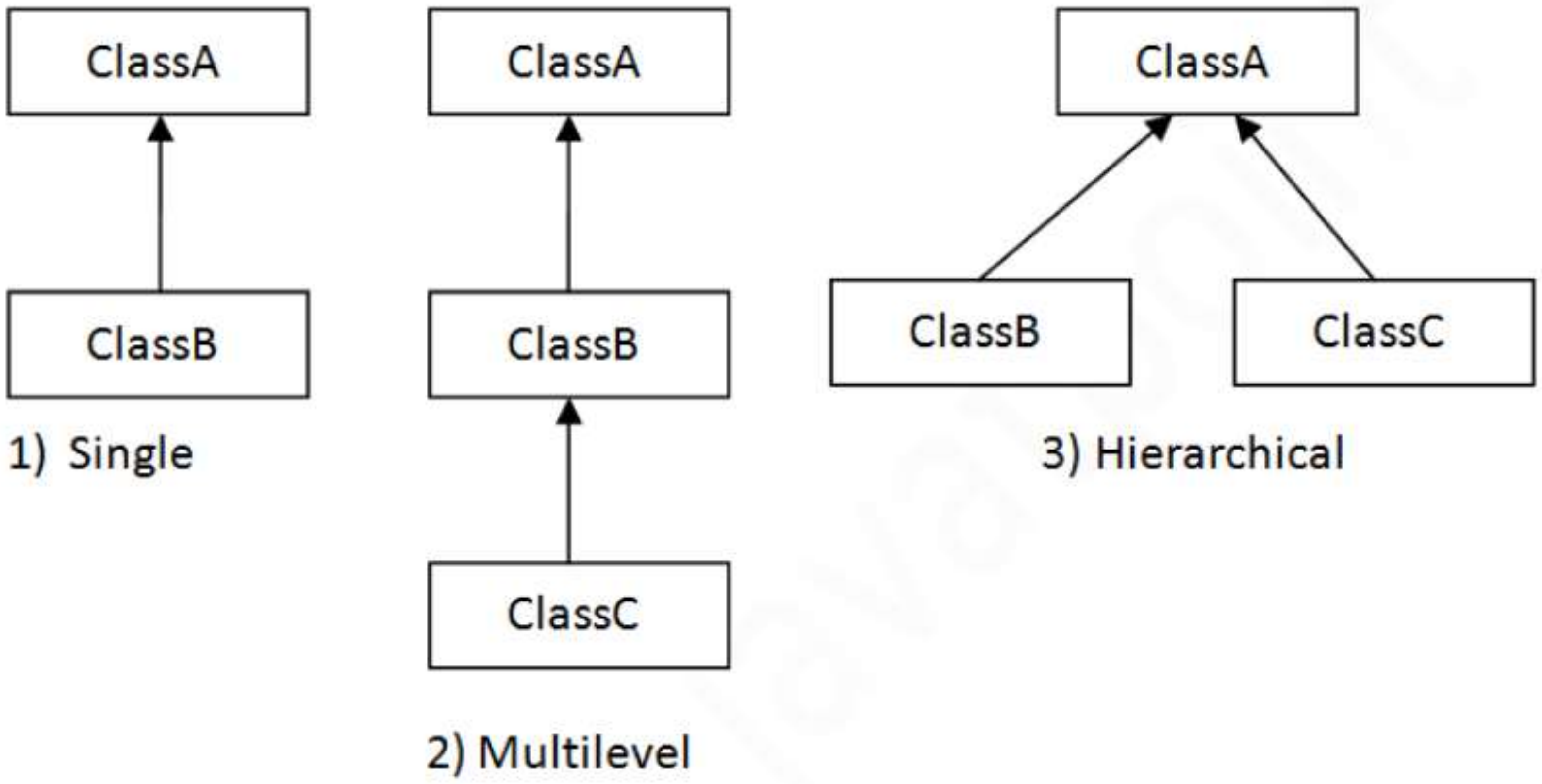
Inheritance:

- **Inheritance** is a mechanism in which one object acquires all the properties and behaviors of a parent object.
- It is an important part of OOPs.
- You can create new classes that are built upon existing classes.
- When you inherit from an existing class, you can reuse methods and fields of the parent class.
- You can add new methods and fields in your current class also.
- Inheritance represents the **IS-A relationship** which is also known as a ***parent-child*** **relationship.**

Classes, Superclasses, & Subclasses

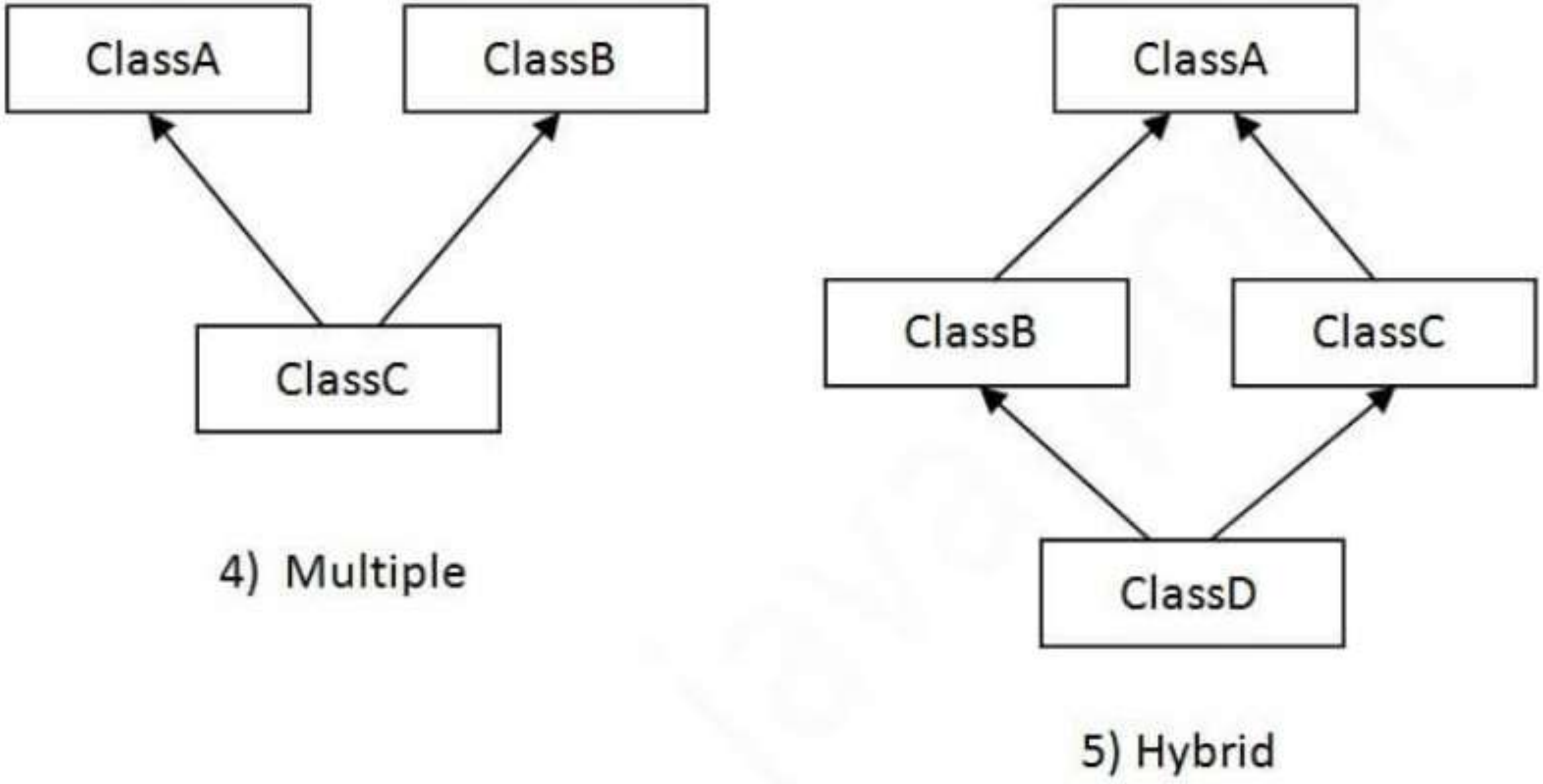
- **Class:**
  - ❑ A class is a group of properties and methods.
  - ❑ It is a template or blueprint from which objects are created.
- **Sub Class/Child Class/derived class:**
  - ❑ Subclass is a class which inherits the other class.
  - ❑ It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class/base class:**
  - ❑ Superclass is the class from where a subclass inherits the features.
  - ❑ It is also called a base class or a parent class.
- **Reusability:**
  - ❑ As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class.
  - ❑ You can use the same fields and methods already defined in the previous class.

Types of inheritance in java



On the ***basis of class***, there can be three types of inheritance in java: single, multilevel and hierarchical.

Types of inheritance in java



4) Multiple

5) Hybrid

multiple and hybrid inheritance is ***supported through interface*** only.

Defining Subclasses

```
public class Manager extends Employee
{
        private double bonus;
        . . .
        public void setBonus(double bonus)
        {
          this.bonus = bonus;
        }
}
```

- Manager inherits methods from superclass: getName, getHireday, getSalary, raiseSalary
- Superclass fields name, salary are present in all Manager objects.

overriding methods,

- If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in java**.
- **Usage of Java 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
- **Rules for Java Method Overriding**
  - ❑ method must have same name as in the parent class
  - ❑ method must have same parameter as in the parent class.
  - ❑ must be IS-A relationship (inheritance).

Example

```
class Vehicle {
        void run() {
                System.out.println("Vehicle is running");
        }
}
class Bike2 extends Vehicle {
        void run() {
                System.out.println("Bike is running safely");
        }
        public static void main(String args[]) {
        Bike2 obj = new Bike2();
        obj.run();
}
```

Subclass Construction

- Subclass constructor can invoke superclass constructor:

```
public Manager(String name, double salary, int year, int month, int day)
{
  super(name, salary, year, month, day);
  bonus = 0;
}
```

1. Call using super must be the first statement.
2. It call parameterized constructor of super class with supplied arguments.
3. If no explicit call to superclass constructor, no-arg/default constructor of superclass is invoked.
4. If the superclass does not have a no-arg/default constructor, the compiler reports an error.

Polymorphism

- **Polymorphism** is a concept by which we can perform a *single action in different ways*.
- Polymorphism is derived from 2 Greek words:
  - ❑ poly and morphs
  - ❑ "poly" means many and
  - ❑ "morphs" means forms.
- So polymorphism means many forms.

Types of Polymorphism

- There are two types of polymorphism in Java:
  1. **compile-time polymorphism and**
  2. **runtime polymorphism.**
- We can perform polymorphism in java by method
  - ❑ overloading and
  - ❑ method overriding.
- If you overload a static method in Java, it is known as **compile time polymorphism**.
- **Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

More about Method Calls

- Suppose **x** is declared to be of type/class C.
  - ❑ Consider a method call:
  - ❑ x.myMehod(args)
- The compiler finds all accessible methods called myMehod() in C and its superclasses.
- The compiler selects the method whose parameter types match the argument types (overloading resolution).
- If the method is private, static, or final, then the compiler knows exactly which method to call (static binding).
- Otherwise, the exact method is found at runtime (dynamic binding).

Example

- **Consider a mix of employees and managers:**

```
staff[0] = boss;
staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
```

- **Print out everyone's salaries:**

```
for (Employee e : staff)
  System.out.println(e.getName() + " " + e.getSalary());
```

- **Which getName() called?**
  - There is only one: Employee.getName
- **Which getSalary() called?**
  - Employee.getSalary or Manager.getSalary?
  - It depends on the actual type of e!

Final Classes

- A final class cannot be extended:

```
public final class Executive
{
    . . .
}
                    /*                    or                    */
public final class Executive extends Manager
{
    . . .
}
```

Final Methods

- A final method cannot be overridden:

```
public class Employee
{
        . . .
        public final String getName()
        {
                return name;
        }
}
```

Casting

- Sometimes, you know more than the compiler about the actual type of a value.
- Suppose you know that staff[0] is a Manager.
- To call Manager methods, you need to cast:

```
Manager boss = (Manager) staff[0];
boss.setBonus(...);
```

- If staff[0] wasn't actually a Manager, a **ClassCastException** occurs.

instanceOf operator:

```
if (staff[1] instanceof Manager)
{
        boss = (Manager) staff[1];
        . . .
}
```

Abstraction

- **Abstraction** is a process of **hiding the implementation** details and showing only functionality to the user.
- It shows only essential things to the user and hides the internal details,
- **for example,**
  - ❑ sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.
- **Abstraction lets you focus on what the object does instead of how it does it.**
- There are **two ways** to achieve abstraction in java
  1. abstract class (0 to 100%)
  2. interface (100%)

Abstract Method

- **A method which is declared as abstract and does not have implementation/body is known as an abstract method.**
- When factoring out common classes, it can become difficult to implement methods in the most general classes.
- **Example:**
  - ❑ Classes **Employee** and **Student** with common superclass **Person**.
  - ❑ Each class defines a **getDescription()**, returning a string:
    - *an employee with a salary of $50,000.00*
    - *a student majoring in computer science*
  - ❑ What is the description of a Person?
- Declare method as abstract and don't provide implementation:

```
public abstract String getDescription();
```

Abstract Classes

- A class which is declared as abstract is known as an **abstract class**.
- It can have abstract and non-abstract methods.
- It needs to be extended and its method implemented.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

Abstract Classes

- **Class with abstract methods must be declared abstract:**
  - ❑ public **abstract** class Person
- **Ok for abstract classes to have fields, constructors, and concrete methods:**

```
public abstract class Person
{
   private String name;
   public Person(String n) { name = n; }
   public String getName() { return name; }
   public abstract String getDescription();
}
```

- **Abstract classes cannot be instantiated:**
  - ❑ Person p1 = new Person(“Bipin"); // Error!
  - ❑ Person p2 = new Student(“Bipin", “Core Java"); // Ok
- **A class can be declared abstract even if it has no abstract methods.**

Access Modifiers

- The access modifiers in java specifies accessibility (scope) of a data member, method, constructor or class.
- There are 4 types of java access modifiers:

1. **private**
2. **default**
3. **protected**
4. **public**

Access Modifiers

- **private**
  - ❑ The private access modifier is accessible only **within class**.
  - ❑ If you make any class constructor private, you cannot create the instance of that class from outside the class
- **default**
  - ❑ If you don't use any modifier, it is treated as default bydefault.
  - ❑ The default modifier is accessible only **within package**.
- **protected**
  - ❑ The protected access modifier is accessible within package and outside the package **but through inheritance only**.
  - ❑ The protected access modifier can be applied on the data member, method and constructor.
  - ❑ It can't be applied on the class.
- **public**
  - ❑ The public access modifier is **accessible everywhere**.
  - ❑ It has the widest scope among all other modifiers.

Access Modifiers

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Object: Cosmic superclass

- **Object is superclass of all Java classes**.
- Only **primitive types** int, double, etc. are **not objects**.
- Arrays are objects.
- Any object or array reference can be stored in a variable of type Object:
  - **Object obj1 = new Employee(...);**
  - **Object obj2 = new int[10];**
- Object class has useful methods:
  - equals(),
  - hashCode(),
  - toString()

The equals Method

- **Object.equals** tests whether the object references are identical.
- Override to test when two objects should be equal.
- Example: Consider two Employee objects equal if their fields are the same.

```
 public boolean equals(Object otherObject)  {
       if (this == otherObject) return true;
       if (otherObject == null) return false;
       if (getClass() != otherObject.getClass()) return false;
       Employee other = (Employee) otherObject;
       return name.equals(other.name)
                  && salary == other.salary
                  && hireDay.equals(other.hireDay);
 }
```

- **Static Objects.equals method is null safe:**

```
 return Objects.equals(name, other.name)    && salary == other.salary
            && Object.equals(hireDay, other.hireDay);
```

The equals Method in a Subclass

- Call **super.equals**, then compare subclass fields:

```
public class Manager extends Employee
{
  . . .
  public boolean equals(Object otherObject)
  {
    if (!super.equals(otherObject))  return false;
    Manager other = (Manager) otherObject;
    return bonus == other.bonus;
  }
}
```

- The superclass method checks that the classes match.

The hashCode Method

- A hash code is an integer that is derived from an object.
- The hashCode method is defined in the Object class. Therefore, every object has a default hash code.
- That hash code is derived from the object’s memory address.
- Hash codes should be scrambled—if x and y are two distinct objects, there should be a high probability that x.hashCode() and y.hashCode() are different.
- Override hashCode whenever you override equals!
- Combine the hash codes of the fields that the equals method compares:

Example

```
public class Employee
{
    . . .
    public int hashCode()
    {
        return Objects.hash(name, salary, hireDay);
    }
}
```

The toString Method

- **Yields a string representation of an object.**
- **When you concatenate a string and an object, the toString method is invoked on the object:**

```
"Center: " + p // Calls p.toString()
```

- **Object.toString yields class name and hash code.**
- **Override for a more meaningful result, such as:**

```
java.awt.Point[x=10,y=20]
```

- **Implementation:**

```
public class Point
{
      . . .
      public String toString()
      {
         return "java.awt.Point[x=" + x + ",y=" + y + "]";
      }
}
```

Inheritance and the toString()

- **In Employee class:**

```
public String toString()
{
   return getClass().getName()
      + "[name=" + name + ",salary=" + salary + ",hireDay=" + hireDay + "]";
}
```

- **In Manager subclass:**

```
public String toString()
{
   return super.toString() + "[bonus=" + bonus + "]";
}
```

- **Result format:**

```
Manager[name=...,salary=...,hireDay=...][bonus=...]
```

Generic Array Lists

- The length of an array is fixed—inconvenient when it is not known in advance.
- ArrayList class manages an Object[] array that grows and shrinks on demand.
- Generic class: Use a type parameter such as ArrayList<Employee> to specify element type.
- Can omit type parameter in the constructor (diamond syntax):

```
ArrayList<Employee> staff = new ArrayList<>();
```

- Use add method to add object to the end:

```
staff.add(new Employee("Harry Hacker", . . .));
```

- The call staff.size() yields the current size.
- Access and modify elements with the get and set methods:

```
Employee e = staff.get(i);
staff.set(i, tony);
```

- Can use “for each” loop to visit elements:

```
or (Employee e : staff) System.out.println(e);
```

Object Wrappers and Autoboxing

- **ArrayList can only hold objects, not int values.**
- **An object of the Integer wrapper class wraps an int value.**
- **Conversion between int and Integer is automatic:**

```
ArrayList<Integer> list = new ArrayList<>();
list.add(3); // same as list.add(Integer.valueOf(3));
int n = list.get(i); // same as int n = list.get(i).intValue();
```

- **Even works with increment:**

```
Integer n = 1000;
n++;
```

- **Caution: == doesn't work with wrappers.**

```
Integer a = n + 1;
Integer b = n + 1;
System.out.println(a == b); // May be false
```

- **Caution: Wrappers can be null.**

```
Integer n = null;
System.out.println(n + 1); // Null pointer exception
```

Enumeration classes

- **Enumeration class defines all instances:**

```
public enum Size { SMALL, MEDIUM, LARGE, EXTRA_LARGE };
```

- **Can add constructors, methods, and fields:**

```
public enum Size
{
  SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");
  private String abbreviation;
  private Size(String abbreviation) { this.abbreviation = abbreviation; }
  public String getAbbreviation() { return abbreviation; }
}
```

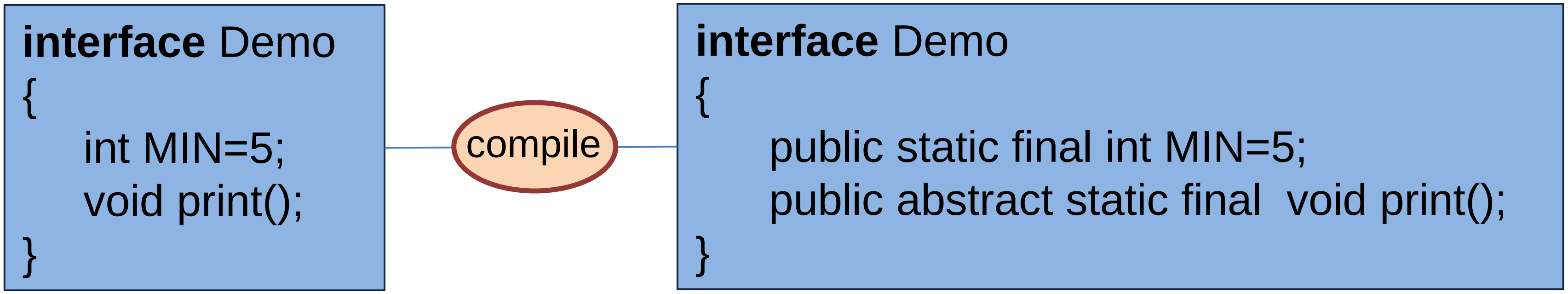
- **All enumeration classes are subclasses of Enum and inherit methods:**
  - toString—yields the name "SMALL", "MEDIUM", ...
  - ordinal—yields the position 0, 1, ...
- **Useful static methods:**
  - Enum.valueOf(Size.class, "SMALL") yields Size.SMALL
  - Size.values() yields all values in an array of type Size[]

Interface

- An **interface in java** is a blueprint of a class.
- It has static constants and abstract methods.
- There can be only abstract methods in the Java interface, not method body.
- It is used to achieve abstraction and multiple inheritance in Java.
- Java Interface also **represents the IS-A relationship**.
- It cannot be instantiated just like the abstract class.
- Since Java 8,
  - ❑ we can have **default and static methods** in an interface.
- Since Java 9,
  - ❑ we can have **private methods** in an interface.

How to declare an interface?

```
interface <interface_name>
{
    // declare constant fields
    // declare methods that abstract
    // by default.
}
```



```
interface Demo
{
    int MIN=5;
    void print();
}
```

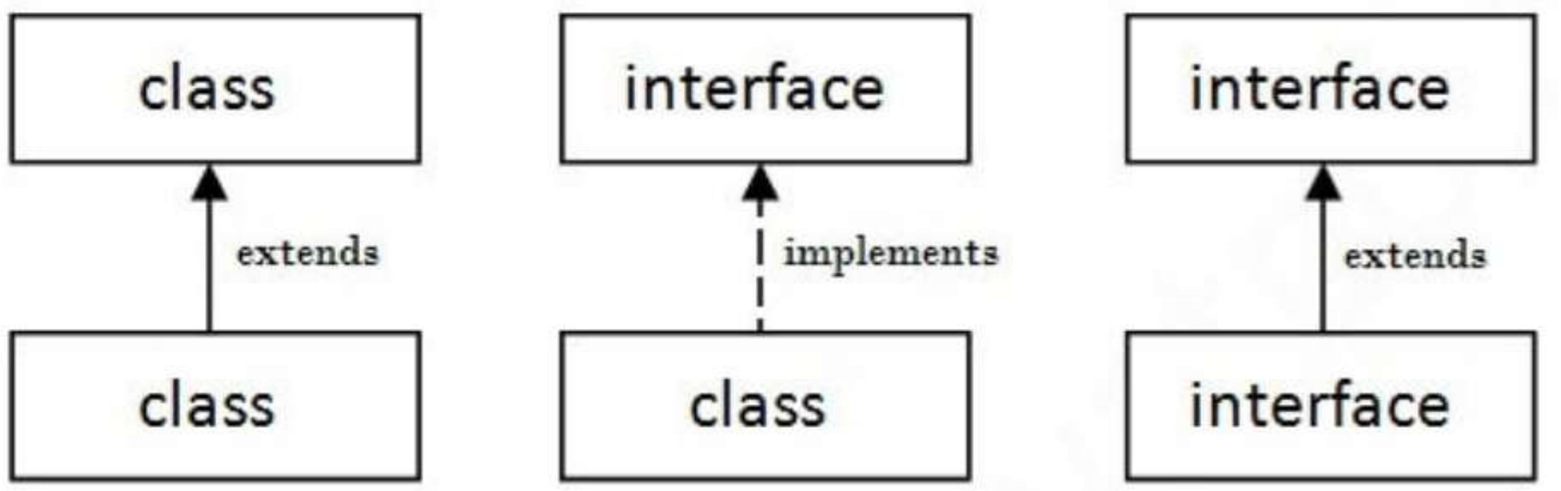
compile

```
interface Demo
{
      public static final int MIN=5;
      public abstract static final  void print();
}
```

The Java compiler adds public and abstract keywords before the interface method.

Relationship between classes & interfaces

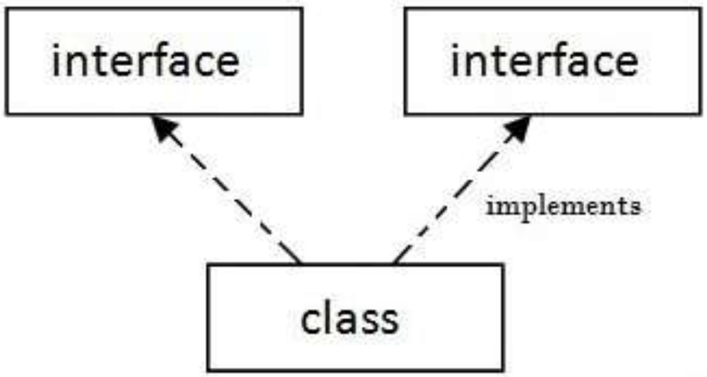
- Java Interface **represents the IS-A relationship**.
- As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.

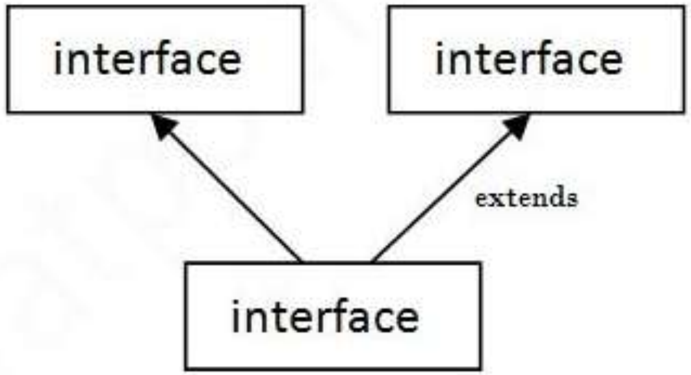


Example

```
interface Drawable{
    void draw();
}
class Rectangle implements Drawable{
     public void draw(){   System.out.println("drawing rectangle");   }
}
class Circle implements Drawable{
    public void draw(){   System.out.println("drawing circle");   }
}
class InterfaceDemo{
    public static void main(String args[]){
        Drawable d=new Circle();
        d.draw();
    }
}
```

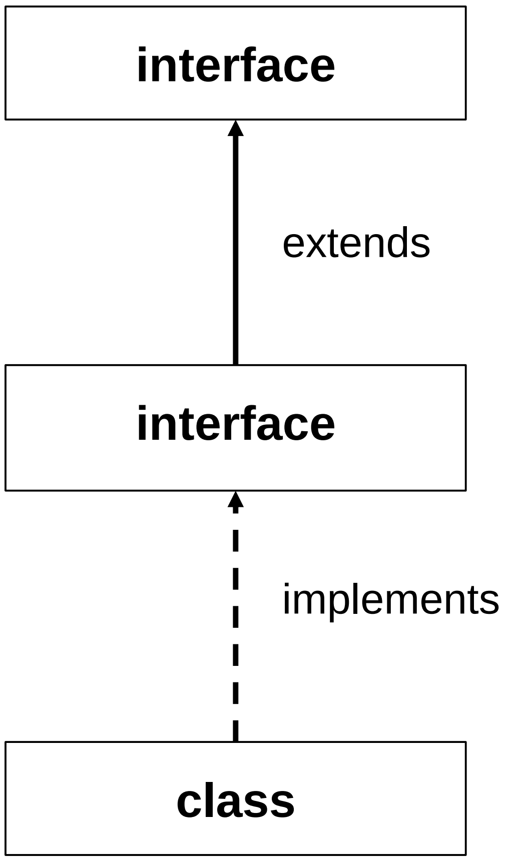
Multiple inheritance by interface





```
interface A{
                void print();
}
interface B{
                void show();
}
class Demo implements A, B  {
                public void print()  {
                  System.out.println("Hello");
                }
                public void show()  {
                  System.out.println("Welcome");
                }
                public static void main(String args[]) {
                                Demo obj = new Demo();
                                obj.print();
                                obj.show();
                }
}
```

Interface inheritance



interface

extends

interface

implements

class

```
interface A{
        void print();
}
interface B extends A {
        void show();
}
class Demo implements  B  {
        public void print()  {
          System.out.println("Hello");
        }
        public void show()  {
          System.out.println("Welcome");
        }
        public static void main(String args[]) {
                Demo obj = new Demo();
                obj.print();
                obj.show();
        }
}
```

Java 8 : Interface's Default & static Method in

```
interface Drawable {
        void draw();
        default void msg()   {  System.out.println("default method");      }
        static int cube(int x) {  return x*x*x;  }
}
class Rectangle implements Drawable {
        public void draw()   {  System.out.println("drawing rectangle");   }
}
class Demo {
        public static void main(String args[]) {
                Drawable d=new Rectangle();
                d.draw();
                System.out.println(Drawable.cube(3));
        }
}
```

Resolving Default Method Conflicts

What happens when the exact same method is defined as a default method in one interface and again as a method of a superclass or another interface?

- Two simple rules:

1. **Interfaces clash**
   - If an interface provides a default method and another interface provides the same one (default or not), you must resolve the conflict.
2. **Superclasses win**
   - Concrete superclass methods mask default methods.

1. The “Interfaces Clash” Rule

- Consider two interfaces:

```
interface Person {  default String getName() { return “hi, Person"; }; }
interface Named { default String getName() { return “hi, Named”; };}
```

- What happens if a class implements both?
- You need to implement the getName method.
- If you like, you can call one or the other interface method:

```
class Student implements Person, Named {
        public String getName()   {
                return Person.super.getName();
        }
}
```

- Even if Named.getName is abstract, you must provide Student.getName.
- If both methods are abstract, you can provide an implementation or declare the class abstract.

2. The “Superclasses Win” Rule

- Assume that Person is a superclass and Named is an interface:

```
class Student extends Person implements Named
{
      . . .
}
```

- Only the superclass method matters.
- The default method **Named.getName** is ignored.
- Ensures compatibility with Java 7:
  - If you add a default method to an interface, it has no impact on existing code.

Java 9 : Interface's private Method

- Interfaces can have concrete private and private static methods.
- Any interface method is abstract, default, static, private, or private static
- Private methods can only be called from default and static methods of the same interface.

Java 9 :
Interface’s private Method

```
public interface MyInterface
{
    default void defaultMethod(){
        privateMethod("Hello from the default method!");
    }
    private void privateMethod(final String string) {
        System.out.println(string);
    }
    void normalMethod();
}
public class PrivateMethodInterfaceDemo implements MyInterface
{
    public void normalMethod() {
        System.out.println("Hello from the implemented method!");
    }
}
```

Abstract class		Interface
1	Abstract class can have abstract and non-abstract methods.	Interface can have only abstract methods. Since Java 8, it can have default and static methods also.
2	Abstract class doesn't support multiple inheritance.	Interface supports multiple inheritance.
3	Abstract class can have final, non-final, static and non-static variables.	Interface has only static and final variables.
4	Abstract class can provide the implementation of interface.	Interface can't provide the implementation of abstract class.
5	The abstract keyword is used to declare abstract class.	The interface keyword is used to declare interface.
6	An abstract class can extend another Java class and implement multiple Java interfaces.	An interface can extend another Java interface only.
7	An abstract class can be extended using keyword extends	An interface class can be implemented using keyword implements
8	A Java abstract class can have class members like private, protected, etc.	Members of a Java interface are public by default.

9	Example:
	public abstract class Shape{
	public abstract void draw();
	}
		Example:
		public interface Drawable{
		void draw();
		}

What is

marker or tagged interface ?

- An interface which has no member is known as a **marker or tagged interface**, for example, Serializable, Cloneable, Remote, etc.
- They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

```
public interface Serializable
{
}
```

Some important interfaces

- ActionListener
- Comparators
- Cloneable

ActionListener interface

- It is used for callbacks
- Callback: **Action that should happen when an event occurs.**
  - ❑ Example: Timer makes callback whenever a time interval has elapsed.
- **The timer calls the actionPerformed method:**

```
class TimePrinter implements ActionListener
{
  public void actionPerformed(ActionEvent event)
  {
        System.out.println("At the tone, the time is " + new Date());
        Toolkit.getDefaultToolkit().beep();
  }
}
```

- **Construct and install the object:**

```
ActionListener listener = new TimePrinter();
Timer t = new Timer(10000, listener);
t.start();
```

Comparable Interface

- Arrays.sort sorts an array if the element class conforms to the Comparable interface.
- Interface definition:

```
public interface Comparable
{
  int compareTo(Object other); // automatically public
}
```

- Conforming class must provide compareTo method.

Comparators interface

- Comparator have two version of interface
  - ❑ Non – Generic
  - ❑ Generic
- It have compareTo() method
- The compareTo method returns:
  - ❑ A positive integer if otherObject should come before this object.
  - ❑ Zero if the two are indistinguishable.
  - ❑ A negative integer otherwise.

Non-Generic

Comparators interface

```
public class Employee implements Comparable
{
   public int compareTo(Object otherObject)
   {
      Employee other = (Employee) otherObject;
      return Double.compare(salary, other.salary);
   }
   . . .
}
```

Generic

Comparators interface

- Can avoid the cast with the generic version of the Comparable interface:

```
class Employee implements Comparable<Employee>
{
   public int compareTo(Employee other)
   {
      return Double.compare(salary, other.salary);
   }
   . . .
}
```

object cloning with Cloneable interface

- The **object cloning** is a way to create exact copy of an object.
- The **clone() method of Object class** is used to clone an object.
- The **java.lang.Cloneable interface** must be implemented by the class whose object clone we want to create.
- If we don't implement Cloneable interface, clone() method generates **CloneNotSupportedException**.

Cloneable interface

```
public class Employee implements Cloneable {
        . . .
        public Employee clone() throws
        CloneNotSupportedException  {
            // call Object.clone()
            Employee cloned = (Employee) super.clone();
            // clone mutable fields
            cloned.hireDay = (Date) hireDay.clone();
            return cloned;
        }
}
```

Contact:

Bipin S. Rupadiya

Assistant Professor, JVIMS

Mo. : +91-9228582425

Email: info@bipinrupadiya.com

Blog : www.BipinRupadiya.com