



# BCS306A – Object Oriented programming with JAVA

## MODULE - 3

### Inheritance & Interfaces

# Inheritance & Interfaces

- Inheritance: Inheritance Basics
- Using super
- Creating a Multilevel Hierarchy
- When Constructors Are Executed
- Method Overriding
- Dynamic Method Dispatch
- Using Abstract Classes
- Using final with Inheritance
- Local Variable Type Inference and Inheritance
- The Object Class.

# Interfaces

- Interfaces
- Default Interface Methods,
- Use static Methods in an Interface,
- Private Interface Methods.



**Inheritance:** inheritance basics, using super, creating multi level hierarchy, method overriding.

# Inheritance in Java

- **Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object.
- It is an important part of **OOPs** (Object Oriented programming system).
- The idea behind inheritance in Java is that you can create new **classes** that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class.
- Moreover, 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.

## Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

## Terms used in Inheritance

**Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.

**Sub Class/Child 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:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.

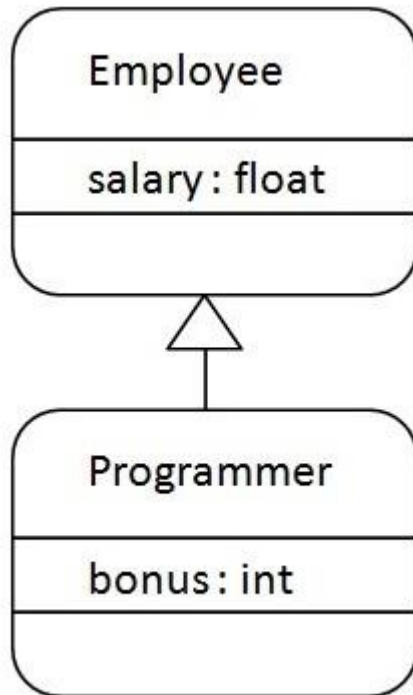
**Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

## The syntax of Java Inheritance

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

- The **extends keyword** indicates that you are making a new class that derives from an existing class.
- The meaning of "extends" is to increase the functionality.
- In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

# Java Inheritance Example



- Programmer is the subclass and Employee is the superclass.
- The relationship between the two classes is **Programmer IS-A Employee**.
- It means that Programmer is a type of Employee.
- [Programmer.java](#)

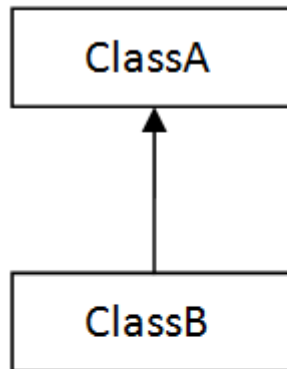
Programmer object can access the field of own class as well as of Employee class i.e. code reusability.



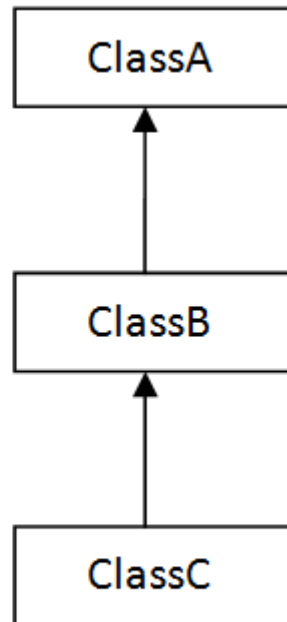
# Types of inheritance in java

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

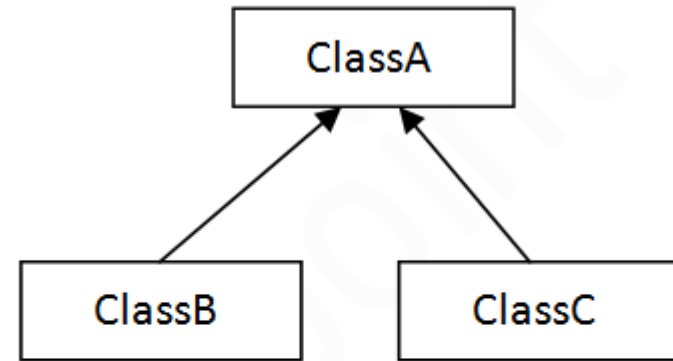
In java programming, multiple and hybrid inheritance is supported through interface only.



1) Single



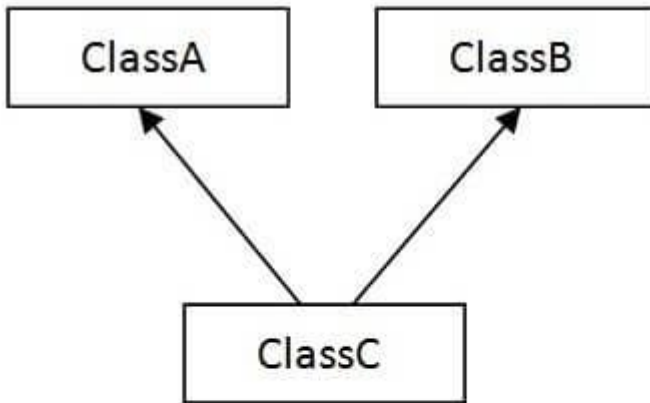
2) Multilevel



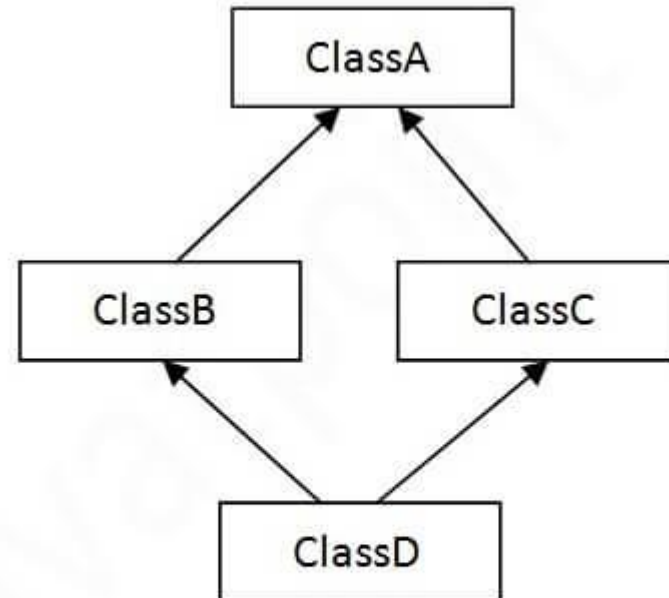
3) Hierarchical

# Types of inheritance in java

When one class inherits multiple classes, it is known as multiple inheritance. For Example:



4) Multiple



5) Hybrid

# Single Inheritance Example

- When a class inherits another class, it is known as a *single inheritance*.
- In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

```
class Animal{  
void eat(){System.out.println("eating...");}  
}  
class Dog extends Animal{  
void bark(){System.out.println("barking...");}  
}
```

```
class TestInheritance{  
public static void main(String args[]){  
Dog d=new Dog();  
d.bark();  
d.eat();  
}}
```

# Multilevel Inheritance Example

- When there is a chain of inheritance, it is known as *multilevel inheritance*.
- As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

# Hierarchical Inheritance Example

- When two or more classes inherits a single class, it is known as *hierarchical inheritance*.
- In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

[TestInheritance3.java](#)

# Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

- Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.
- Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

```
class A{
void msg(){System.out.println("Hello");}
}
class B{
void msg(){System.out.println("Welcome");}
}
class C extends A,B{//suppose if it were

public static void main(String args[]){
    C obj=new C();
    obj.msg();//Now which msg() method would be invoked?
}
}
```

# Super keyword in JAVA

- The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.
- Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

## Usage of Super Keyword

1

Super can be used to refer immediate parent class instance variable.

2

Super can be used to invoke immediate parent class method.

3

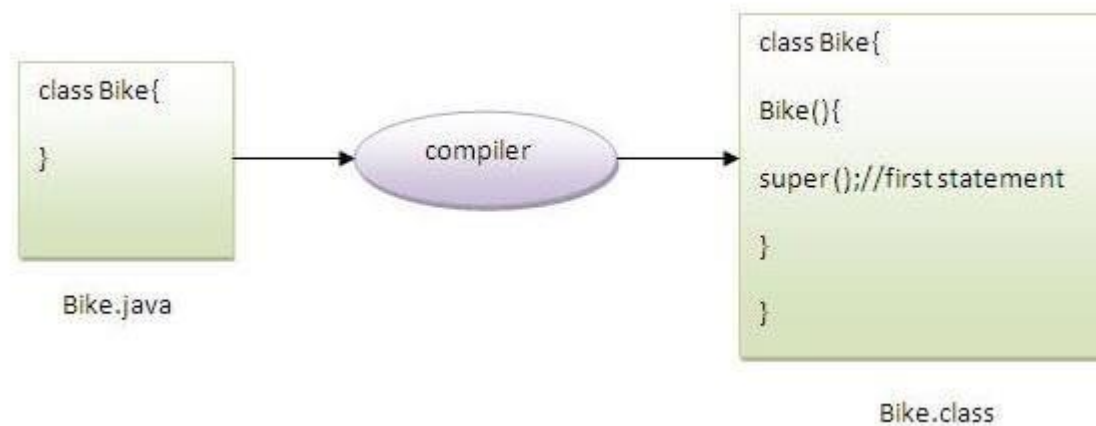
super() can be used to invoke immediate parent class constructor.

1. Super is used to refer immediate parent class instance variable

[TestSuper1.java](#)

2. Super can be used to invoke parent class method [TestSuper2.java](#)

3. Super is used to invoke parent class constructor [TestSuper3.java](#)



Note: `super()` is added in each class constructor automatically by compiler if there is no `super()` or `this()`.

**Another example of super keyword where `super()` is provided by the compiler implicitly.**

[TestSuper4.java](#)

super example: real use [TestSuper5.java](#)

# Constructors in Java

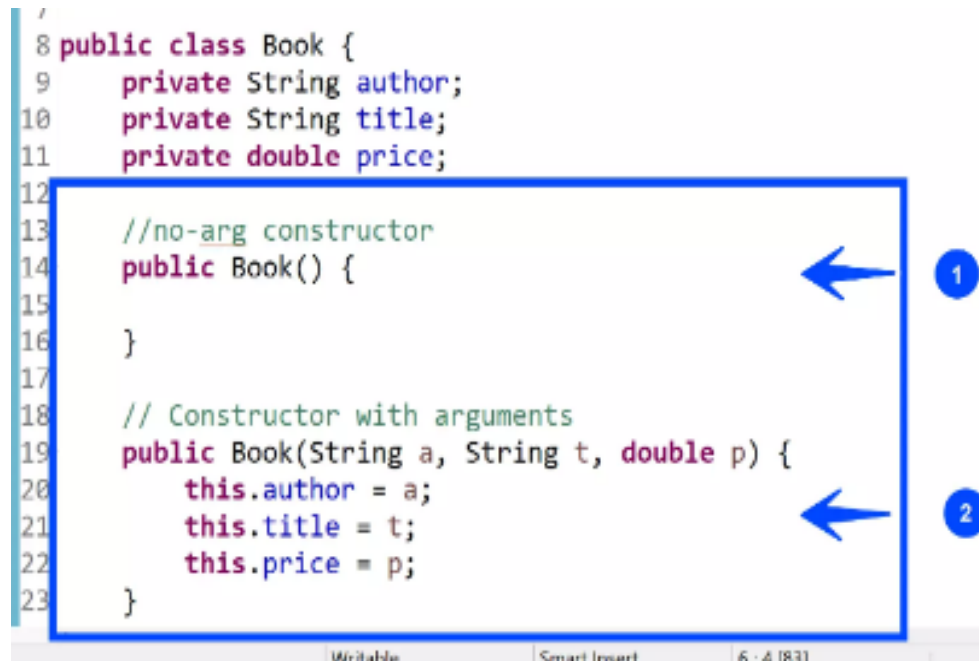
- A constructor in Java is similar to a method with a few differences. Constructor has the same name as the class name. A constructor doesn't have a return type.
- A Java program will automatically create a constructor if it is not already defined in the program. It is executed when an instance of the class is created.
- A constructor cannot be static, abstract, final or synchronized. It cannot be overridden



# Constructors in Java

- Java has two types of constructors:
- Default constructor
- Parameterized constructor

```
8 public class Book {  
9     private String author;  
10    private String title;  
11    private double price;  
12  
13    //no-arg constructor  
14    public Book() {  
15  
16    }  
17  
18    // Constructor with arguments  
19    public Book(String a, String t, double p) {  
20        this.author = a;  
21        this.title = t;  
22        this.price = p;  
23    }
```



# Order of execution of constructor in Single inheritance

- In single level inheritance, the constructor of the base class is executed first.
- `/* Parent Class */`
- **class** ParentClass
- {
- `/* Constructor */`
- ParentClass()
- {
- System.out.println("ParentClass constructor executed.");
- }
- }

```
/* Child Class */  
class ChildClass extends ParentClass  
{  
    /* Constructor */  
    ChildClass()  
    {  
        System.out.println("ChildClass constructor executed.");  
    }  
}
```

```
public class OrderofExecution1
```

```
{  /* Driver Code */
```

```
    public static void main(String ar[])
```

```
{
```

```
    /* Create instance of ChildClass */
```

```
        System.out.println("Order of constructor execution...");
```

```
        new ChildClass();
```

```
}
```

```
}
```

**Output:**

Order of constructor execution...

Parent Class constructor executed.

Child Class constructor executed.

# Method Overriding

- If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.
- In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

# Rules for Java Method Overriding

- The method must have the same name as in the parent class
- The method must have the same parameter as in the parent class.
- There must be an IS-A relationship (inheritance).

```
class Vehicle{  
    void run()  
{  
    System.out.println("Vehicle is running");  
}  
//Creating a child class  
class Bike extends Vehicle{  
    public static void main(String args[]){  
        //creating an instance of child class  
        Bike obj = new Bike();  
        //calling the method with child class instance  
        obj.run();  
    }  
}
```

Output

Vehicle is running

# Dynamic Method Dispatch

- Method overriding is one of the ways in which Java supports Runtime Polymorphism. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- When an overridden method is called through a superclass reference, Java determines which version(superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs.

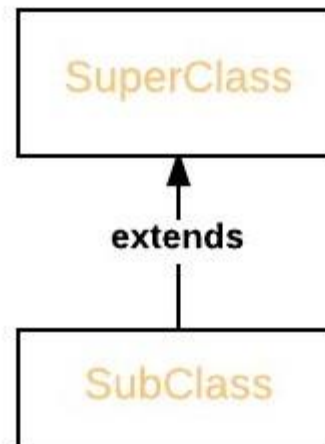


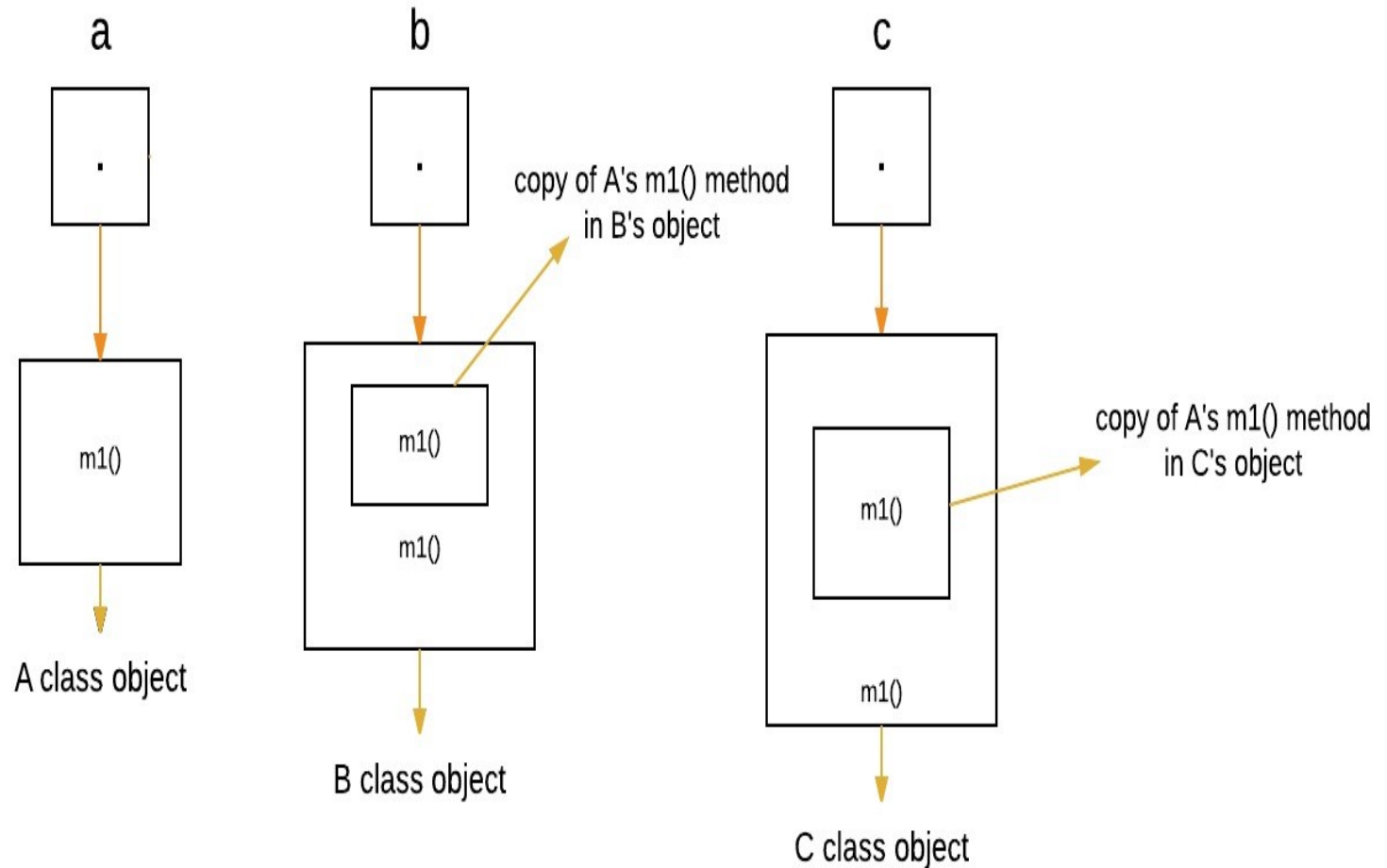
# Dynamic Method Dispatch

- At run-time, it depends on the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed
- A superclass reference variable can refer to a subclass object. This is also known as upcasting. Java uses this fact to resolve calls to overridden methods at run time

## Upcasting

SuperClass obj = new SubClass





# Abstract Classes

- Data **abstraction** is the process of hiding certain details and showing only essential information to the user.
- Abstraction can be achieved with either **abstract classes** or [interfaces](#)
- Java abstract class is a class that can not be initiated by itself, it needs to be subclassed by another class to use its properties. An abstract class is declared using the “abstract” keyword in its class definition.

```
abstract class Shape
{
    int color;
    // An abstract function
    abstract void draw();
}
```

```
abstract class Sunstar  
{  
    abstract void printInfo();  
}
```

```
// Abstraction performed using extends  
class Employee extends Sunstar {  
    void printInfo()  
    {  
        String name = "avinash";  
        int age = 21;  
        float salary = 222.2F;
```



```
System.out.println(name);  
    System.out.println(age);  
    System.out.println(salary);  
}  
}
```

Output  
avinash  
21  
222.2

```
// Base class  
class Base {  
    public static void main(String args[])  
    {  
        Sunstar s = new Employee();  
        s.printInfo();  
    }  
}
```

# Using Final with Inheritance

- During inheritance, the methods with the final keyword for which we are required to follow the same implementation throughout all the derived classes
- We can declare a final method in any subclass for which we want that if any other class extends this subclass, then it must follow the same implementation of the method as in that subclass.

# Local Variable Type Inference and Inheritance

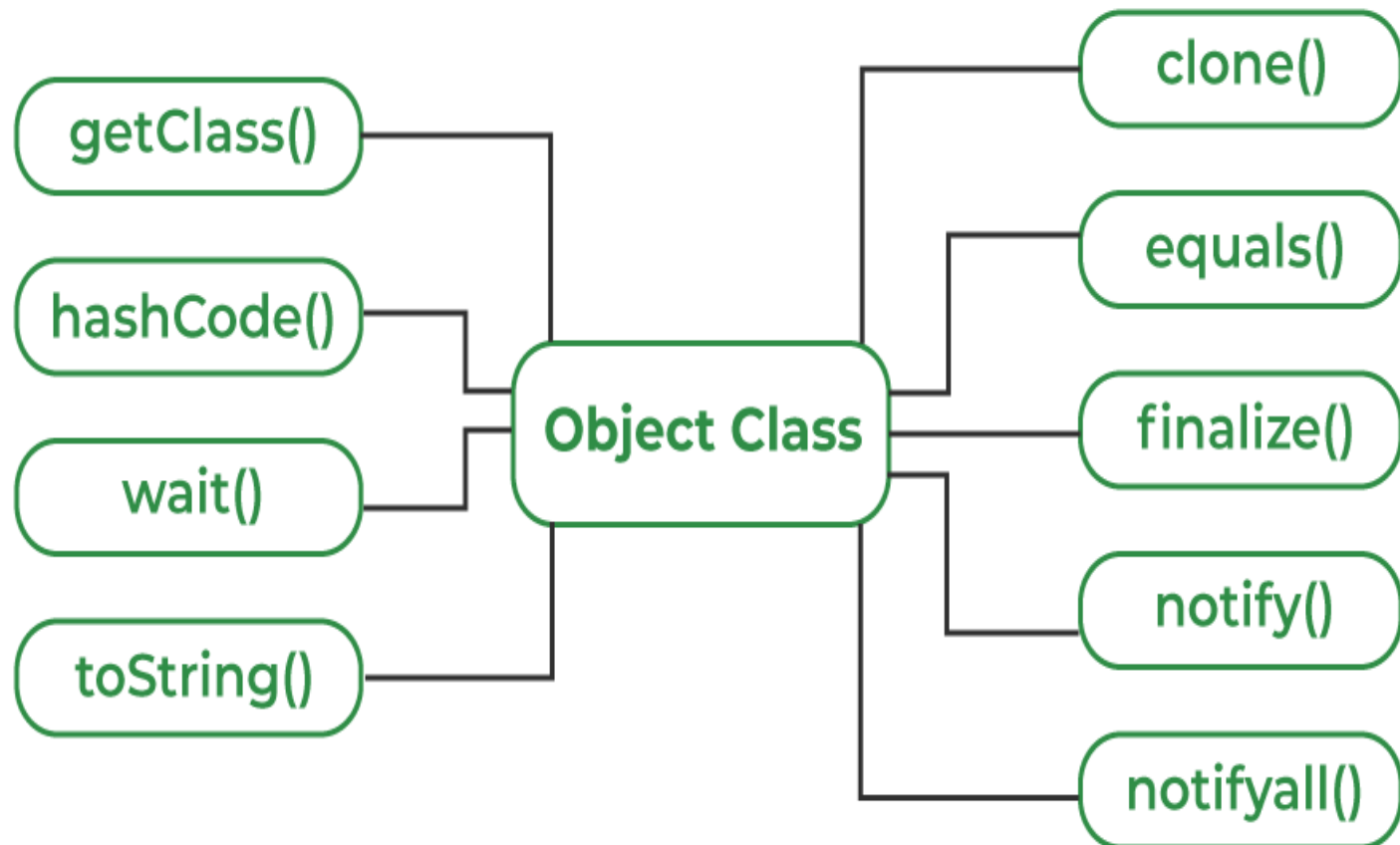
- Local Variable Type Inference is one of the most evident change to language available
- It allows to define a variable using var and without specifying the type of it.
- The compiler infers the type of the variable using the value provided.
- This type inference is restricted to local variables

```
import java.util.Arrays;
import java.util.HashMap;
public class UseOfVar
{
    public static void main(String[] args)
    {
        var intValue = 5;
        var stringValue = "JavaGoal.com";
        var floatValue = 5.5f;
        var longValue = 5.5555;
        var listOfIds = Arrays.asList(1, 2, 3, 4, 5);
        var hashMap = new HashMap<Integer, String>();
        System.out.println(intValue);
        System.out.println(stringValue);
        System.out.println(floatValue);
        System.out.println(longValue);
        System.out.println(listOfIds.toString());
        System.out.println(hashMap);
    }
}
```



# The Object Class

- **Object** class is present in **java.lang** package.
- Every class in Java is directly or indirectly derived from the **Object** class.
- If a class does not extend any other class then it is a direct child class of **Object** and if extends another class then it is indirectly derived.
- Object class methods are available to all Java classes.
- Object class acts as a root of the inheritance hierarchy in any Java Program.



# Object class provides multiple methods

- toString() method
- hashCode() method
- equals(Object obj) method
- finalize() method
- getClass() method
- clone() method
- wait(), notify() notifyAll() methods

# Interfaces

- Using the keyword **interface**, **you can fully abstract a class' interface from its implementation.**
- That is, using **interface**, **you can specify what a class must do, but not how it does it.**
- Interface are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body.
- This means that you can define interfaces that don't make assumptions about how they are implemented. Once it is defined, any number of classes can implement an **interface**.

- Using interfaces, you can perform multiple inheritance.
- Using **interface**, you can fully abstract a class.
- Methods within interfaces are declared without any body.
- Any number of classes can implement an interface.
- One class can implement any number of interfaces.

# Defining an Interface

*Go, change the world*

```

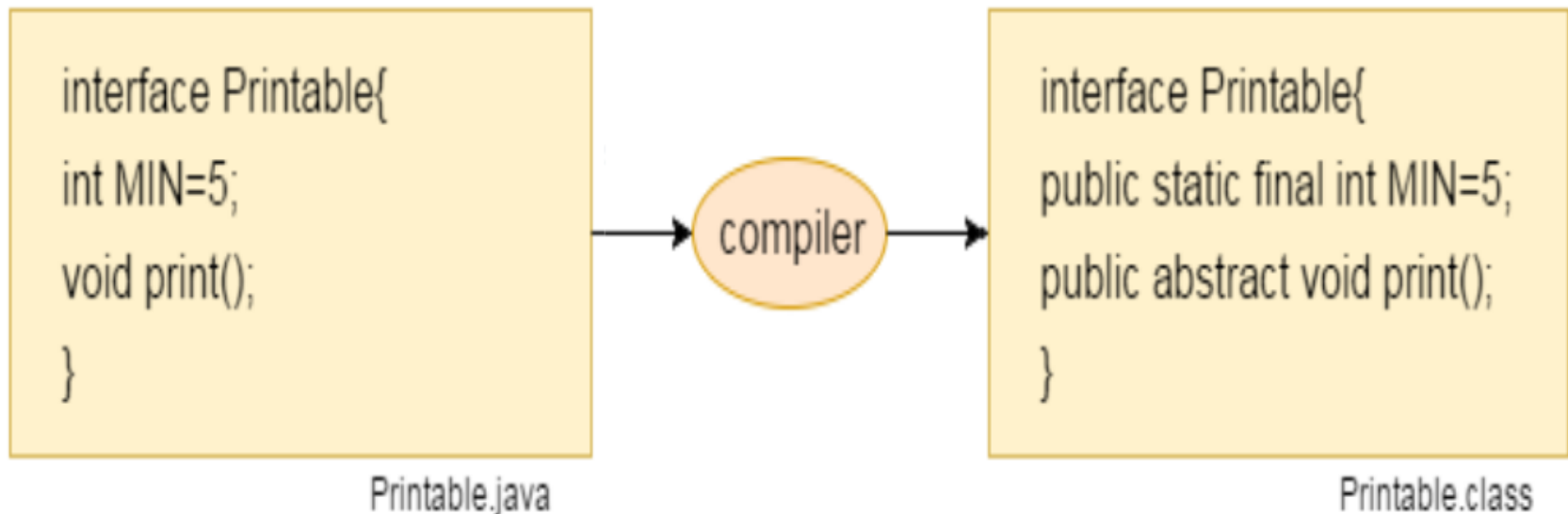
/ default/public
access interface name {
    return-type method-name1(parameter-list);
    return-type method-name2(parameter-list);
    type final-varname1 = value;
    type final-varname2 = value;
    // ...
    return-type method-nameN(parameter-list);
    type final-varnameN = value;
}
    
```

- If the interface is **public**, it must be the only public interface declared in the file, and the file must have the same name as the interface.

# Defining an Interface

## Internal addition by compiler

The java compiler adds **public** and **abstract** keywords before the interface method. More, it adds **public**, **static** and **final** keywords before data members.



# Implementing Interfaces

- Once an **interface** has been defined, one or more classes can **implement** that interface.
- **To** implement an interface, include the **implements** clause in a **class definition**, and then **create** the methods defined by the interface.
- The general form of a class that includes the **implements** clause looks like this:

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

# Implementing Interfaces

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



# Default Interface Methods

- Java provides a facility to create default methods inside the interface.

Methods which are defined inside the interface and tagged with default are known as default methods. These methods are non-abstract methods.

```
interface Sayable{  
    // Default method  
    default void say(){  
        System.out.println("Hello, this is default method");  
    }  
    // Abstract method  
    void sayMore(String msg);  
}  
  
public class DefaultMethods implements Sayable{  
    public void sayMore(String msg){    // implementing abstract method  
        System.out.println(msg);  
    }  
}
```

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

```
    DefaultMethods dm = new DefaultMethods();
```

```
    dm.say(); // calling default method
```

```
    dm.sayMore("Work is worship"); // calling abstract method
```

```
}
```

```
}
```

Output

Hello, this is default  
method

Work is worship

# Static Methods inside Java 8 Interface

- **interface** Sayable{
  - // default method
  - **default void** say(){
    - System.out.println("Hello, this is default method");
  - }
  - // Abstract method
  - **void** sayMore(String msg);
  - // static method
  - **static void** sayLouder(String msg){
    - System.out.println(msg);
  - }
- }

```
public class DefaultMethods implements Sayable
{
    public void sayMore(String msg){    // implementing abstract method
        System.out.println(msg);
    }

    public static void main(String[] args) {
        DefaultMethods dm = new DefaultMethods();
        dm.say();                // calling default method
        dm.sayMore("Work is worship");    // calling abstract method
        Sayable.sayLouder("Helloooo..."); // calling static method
    } }
```

Output

Hello there

Work is worship

Helloooo...

# Use of static Methods in an Interface

- Java interface static methods are good for providing utility methods

example null check, collection sorting etc. Java interface static method helps us in providing security by not allowing implementation classes to override them

- The static method in an interface can be defined in the interface, but cannot be overridden in Implementation Classes. To use a static method, Interface name should be instantiated with it, as it is a part of the Interface only.

```
interface NewInterface {  
    // static method  
    static void hello()  
    {  
        System.out.println("Hello, New Static Method Here");  
    }  
    // Public and abstract method of Interface  
    void overrideMethod(String str);  
}  
// Implementation Class  
public class InterfaceDemo implements NewInterface {  
  
    public static void main(String[] args)  
    {  
        InterfaceDemo interfaceDemo = new InterfaceDemo();  
    }  
}
```

```
// Calling the static method of interface
    NewInterface.hello();

    // Calling the abstract method of interface
    interfaceDemo.overrideMethod("Hello, Override Method here");
}
// Implementing interface method

@Override
public void overrideMethod(String str)
{
    System.out.println(str);
}
}
```

### **Output**

**Hello, New Static Method Here**

**Hello, Override Method here**



# Private Interface Methods

- A private interface method is a special type of Java method that is accessible inside the declaring interface only. This means that no class that extends the interface can access this method directly using an instance of the class. Interface methods are public by default
- Interface can have only four types:
  - Constant variables
  - Abstract methods
  - Default methods
  - Static methods



THANK YOU