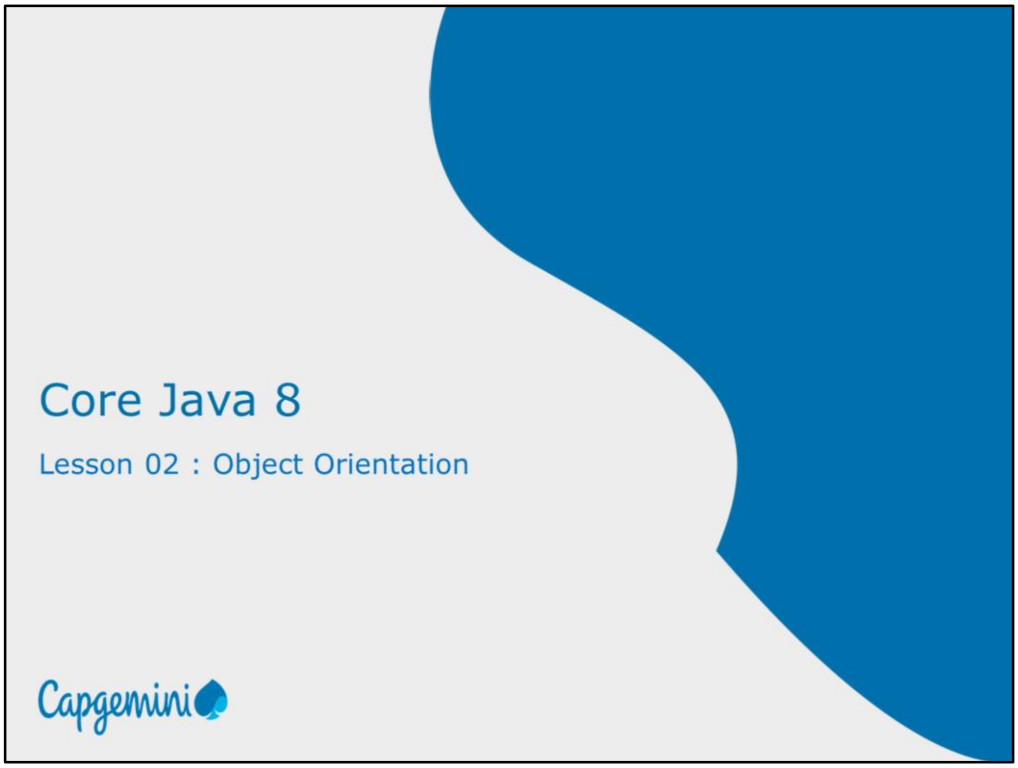
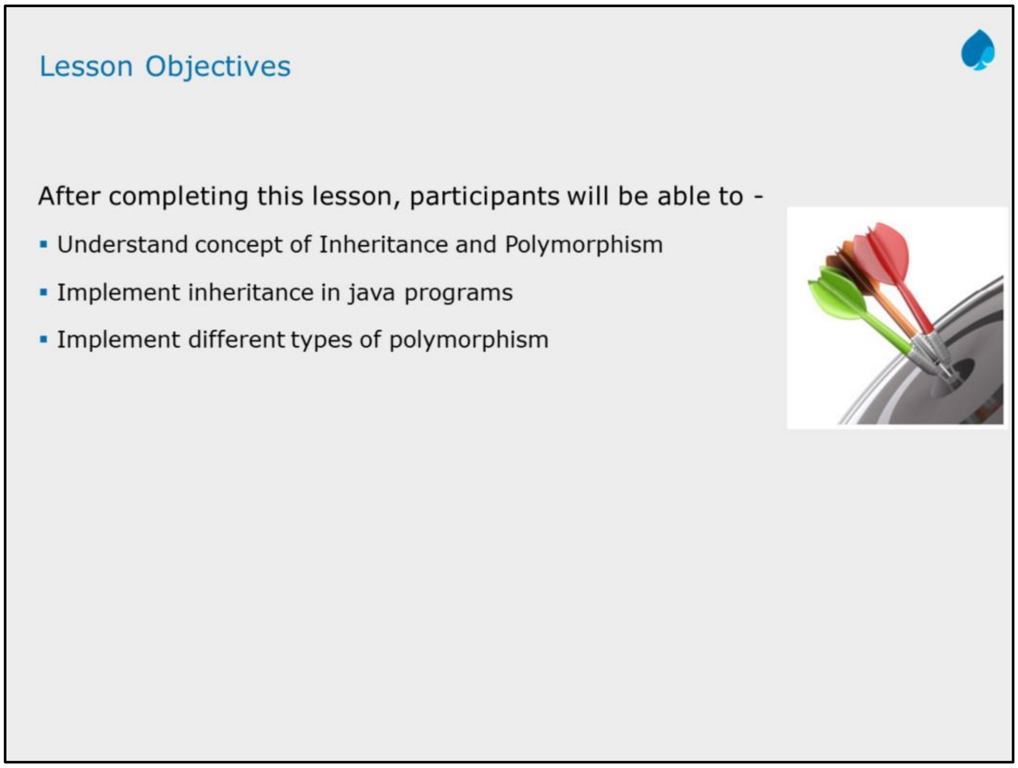


Core Java 8

Lesson 02 : Object Orientation

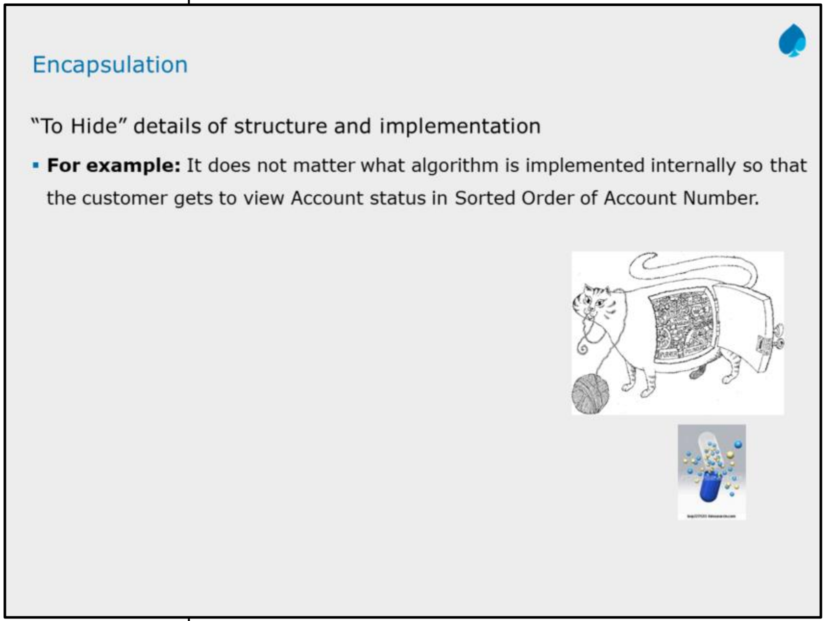
Capgemini



Lesson Objectives

After completing this lesson, participants will be able to -

- Understand concept of Inheritance and Polymorphism
- Implement inheritance in java programs
- Implement different types of polymorphism

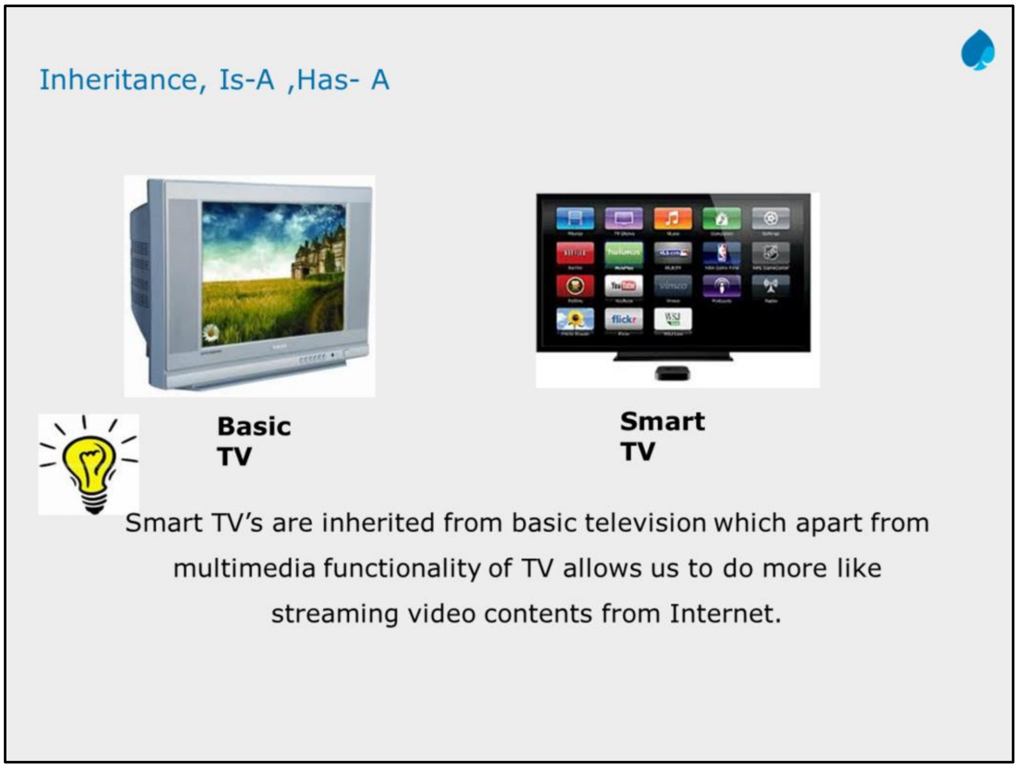


Encapsulation

"To Hide" details of structure and implementation

- **For example:** It does not matter what algorithm is implemented internally so that the customer gets to view Account status in Sorted Order of Account Number.

Encapsulation:
Every object is encapsulated in such a way, that its data and implementations of behaviors are not visible to another object. Encapsulation allows restriction of access of internal data.
Encapsulation is often referred to as information hiding. However, although the two terms are often used interchangeably, information hiding is really the result of encapsulation, not a synonym for it.



Inheritance, Is-A ,Has- A

Basic TV

Smart TV

Smart TV's are inherited from basic television which apart from multimedia functionality of TV allows us to do more like streaming video contents from Internet.

<u>What is Inheritance</u>?

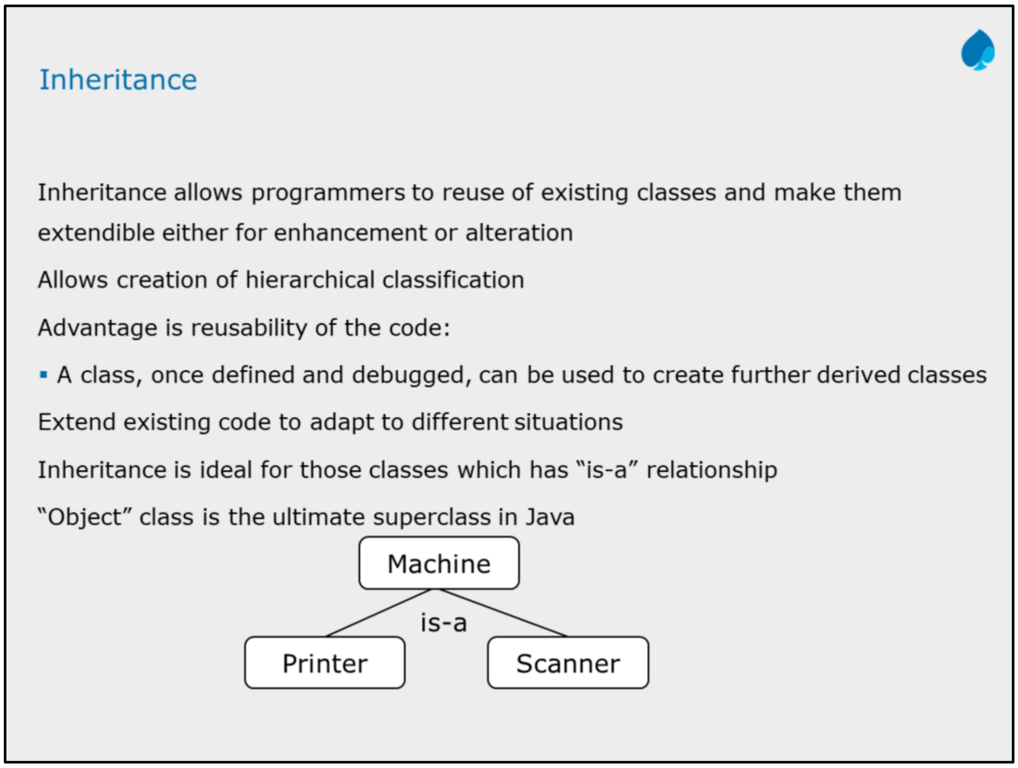
If you look at the slide example of how televisions are evolved in last decade, you will find two major differences between these two models. Basic TV's are used to watch programs typically they are streamed from set top box. On the other hand, the most advanced "smart TVs" are similar in function to "smart phones" and some tablets. These TVs go beyond providing access to web-based media, or streaming from content stored on your home computer. Smart TVs have the built in computing power that allows you to do many of the same things you can do with a smart phone or tablet such as web browsing, use of web-based services like Skype, and interactive access to social media sites.

Apart from the functional **enhancement**, there is slight **alteration** from hardware point of view. The position of sound boxes is altered and in smart TV's like to hide them from front view.

Smart TV's are **"inherited"** from Basic TV for two reasons:

1. To make enhancement and/or
2. To do alteration

This is basis of inheritance in OOPs where one class we can extend to either enhance its functionality or alter its behavior.



Inheritance

Inheritance allows programmers to reuse of existing classes and make them extendible either for enhancement or alteration

Allows creation of hierarchical classification

Advantage is reusability of the code:

- A class, once defined and debugged, can be used to create further derived classes

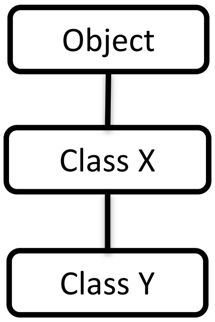
Extend existing code to adapt to different situations

Inheritance is ideal for those classes which has "is-a" relationship

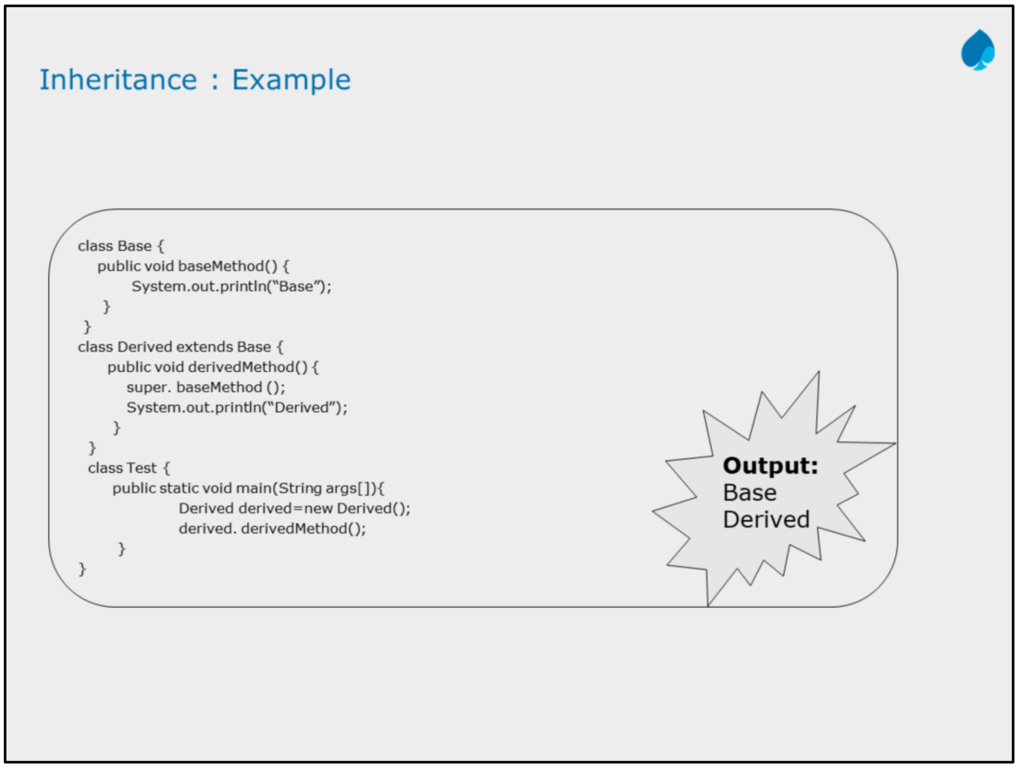
"Object" class is the ultimate superclass in Java

It is one of the fundamental mechanisms for code reuse in object-oriented programming. Inheritance allows new classes to be derived from existing classes. In Java, inheritance specifies how different is the sub class from its parent class. Thus, we can add new variables, methods and also modify the inherited methods. To inherit a class, you simply **extend** the super class into the subclass. Only single level inheritance is possible in Java.

Superclass and Subclass:
Any class **preceding** a specific class in the class hierarchy is said to be super class. On the other hand Any class **following** specific class in the hierarchy is called as subclass. All classes in Java are by default extensible. All All Java classes that do not explicitly extend a parent class automatically extend the **java.lang.Object** class.



In the above example, class Y is subclass. Class X is superclass of Y but subclass of Object class. **Object** class is the ultimate superclass in Java.

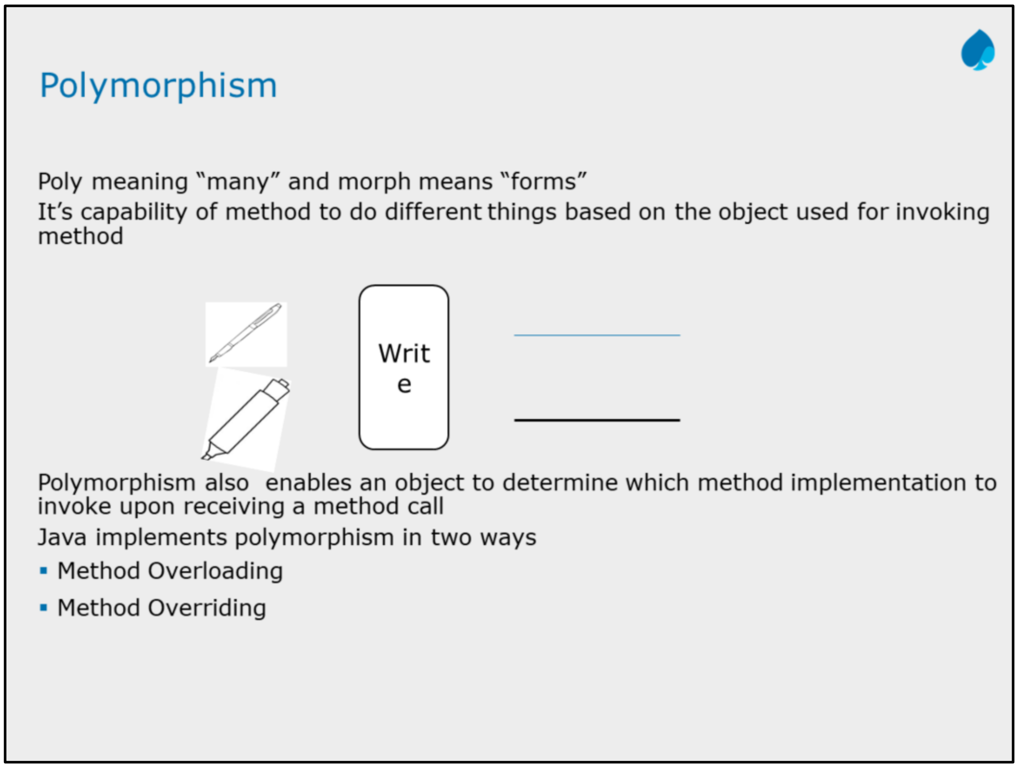


Inheritance : Example

```
class Base {
    public void baseMethod() {
           System.out.println("Base");
     }
 }
class Derived extends Base {
      public void derivedMethod() {
          super. baseMethod ();
          System.out.println("Derived");
       }
  }
  class Test {
       public static void main(String args[]){
                     Derived derived=new Derived();
                     derived. derivedMethod();
        }
}
```

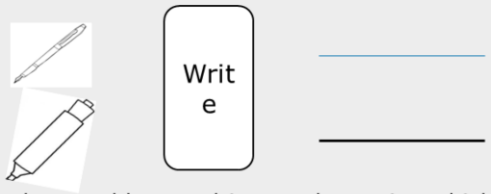
Output:

```
Base
Derived
```



Polymorphism

Poly meaning "many" and morph means "forms"
It's capability of method to do different things based on the object used for invoking method

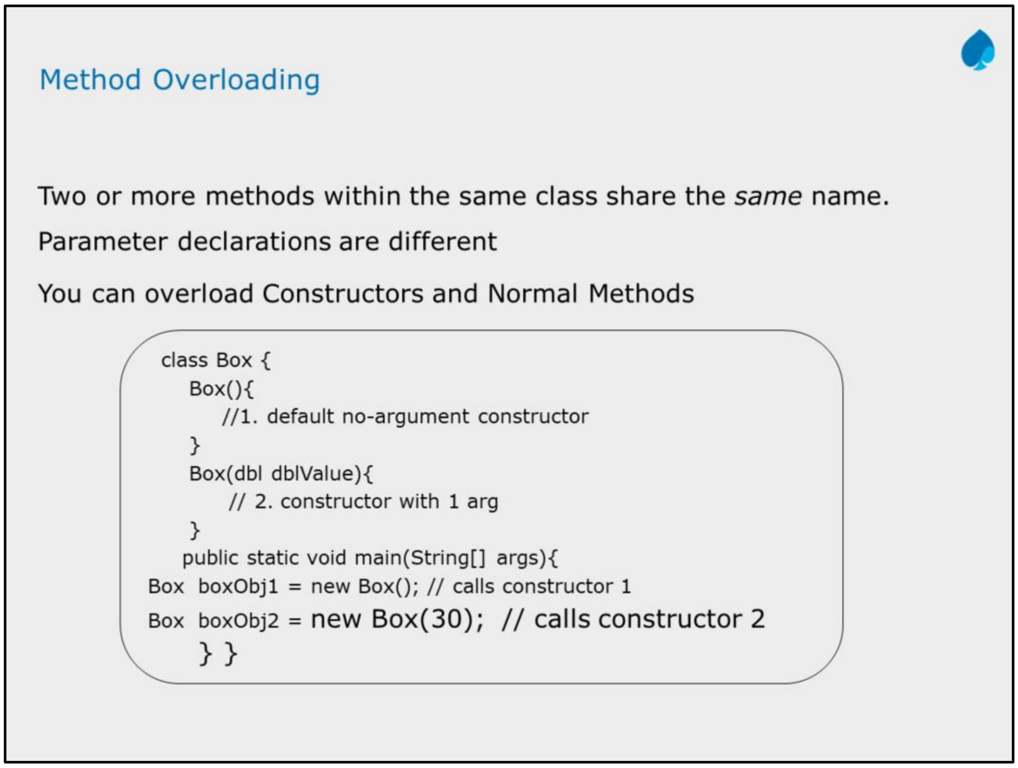


Polymorphism also enables an object to determine which method implementation to invoke upon receiving a method call
Java implements polymorphism in two ways
- Method Overloading
- Method Overriding

There are two ways in which polymorphism is implemented in Java.

1. Method Overloading
2. Method Overriding

Method Overloading is compile time polymorphism where the same method name has different meanings. Method overriding on other hand, is kind of runtime polymorphism where a subclass defines method with the same signature as defined by its superclass.



Method Overloading

Two or more methods within the same class share the *same* name.

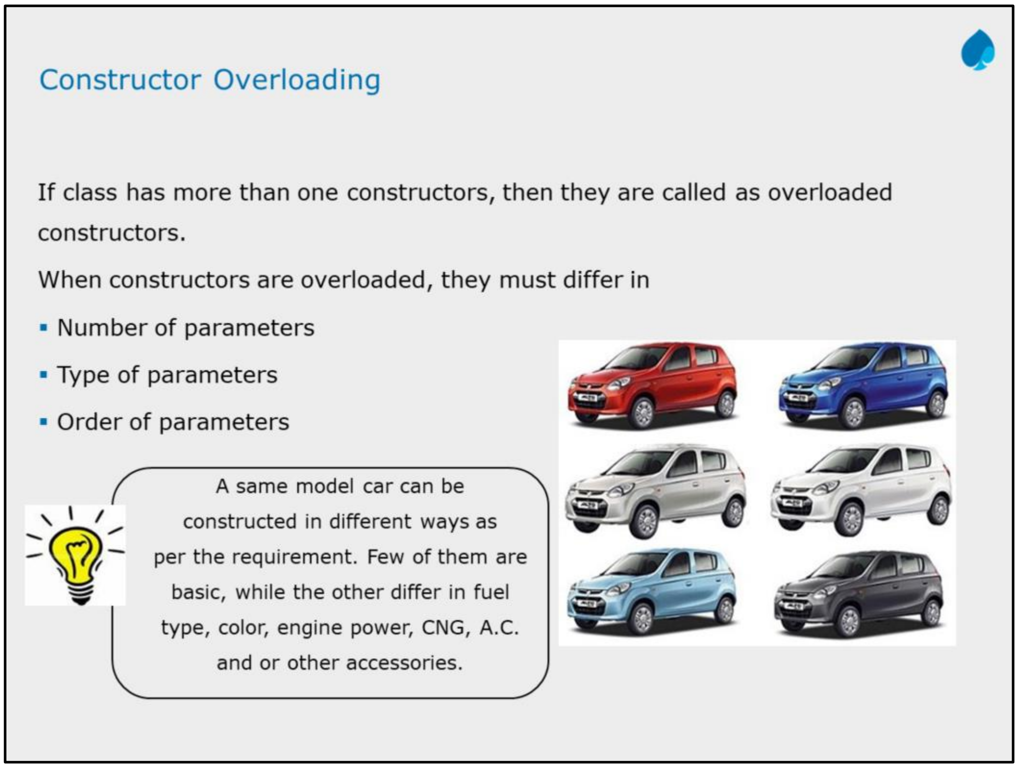
Parameter declarations are different

You can overload Constructors and Normal Methods

```
class Box {
    Box(){
        //1. default no-argument constructor
    }
    Box(dbl dblValue){
        // 2. constructor with 1 arg
    }
   public static void main(String[] args){
Box  boxObj1 = new Box(); // calls constructor 1
Box  boxObj2 = new Box(30);  // calls constructor 2
     } }
```

In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. Here, the methods are said to be overloaded. When an overloaded method is invoked, Java determines which method to actually call by using the type and/or number of operators as its guide.

Refer the example above. Here, two constructors have been declared; one with no arguments, the second with a single argument
Note: In addition to overloading constructors, you can also overload normal methods.



Constructor Overloading

If class has more than one constructors, then they are called as overloaded constructors.

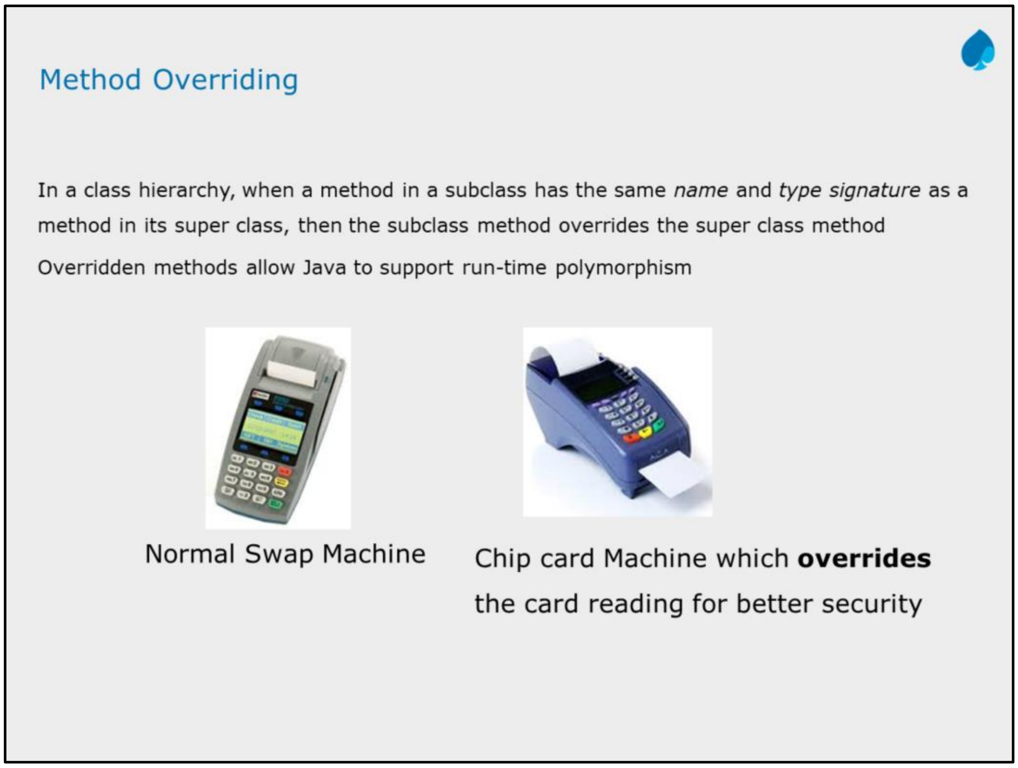
When constructors are overloaded, they must differ in

- Number of parameters
- Type of parameters
- Order of parameters

A same model car can be constructed in different ways as per the requirement. Few of them are basic, while the other differ in fuel type, color, engine power, CNG, A.C. and or other accessories.

Example:

```
class Car {
  int noOfCylinders;
  int noOfValves;
  int enginePower;
  boolean isPowerSteering;
  Car(){
   //1. default no-argument constructor
       noOfCylinders = 3;
       noOfValves = 4;
       enginePower = 48;              //48 ps
       isPowerSteering = false;
  }
  Car(boolean isPowerSteering){
   // 2. constructor with 1 arg
       this();
       this.isPowerSteering = isPowerSteering;
  }
  Car(int enginePower,int noOfCylinders, int noOfValves){
   // 3. constructor with // 3 args
       this.noOfCylinders = noOfCylinders;
       this.noOfValves = noOfValves ;
       this.enginePower = enginePower;
       this.isPowerSteering = true;
  }
}
```



Method Overriding

In a class hierarchy, when a method in a subclass has the same *name* and *type signature* as a method in its super class, then the subclass method overrides the super class method

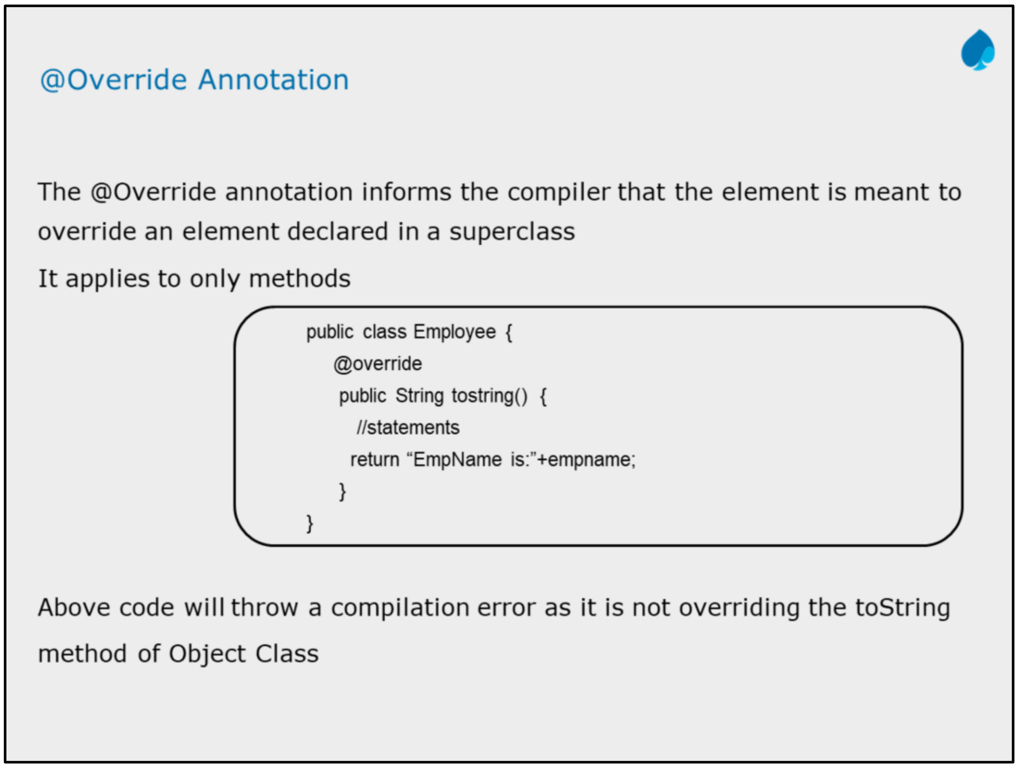
Overridden methods allow Java to support run-time polymorphism

Normal Swap Machine

Chip card Machine which **overrides** the card reading for better security

Example:

```
class SwipeMachine{
    void readCard (){
           // functionality to read normal cards
     }
}
class ChipCardMachine extends SwipeMachine{
    void readCard(){
           //functionality to read chip card
     }
}
public static void main(String args[]){
      SwipeMachine normal = new SwipeMachine();
      normal.readCard();        //reading normal swipe card
      normal = new ChipCardMachine();
      normal.readCard();        //reading chip based swipe card
} }
```



@Override Annotation

The @Override annotation informs the compiler that the element is meant to override an element declared in a superclass

It applies to only methods

```
public class Employee {
   @override
    public String tostring() {
      //statements
     return “EmpName is:”+empname;
    }
}
```

Above code will throw a compilation error as it is not overriding the toString method of Object Class

@Override:
The Compiler checks that a method with this annotation really overrides a method from the Super class or not

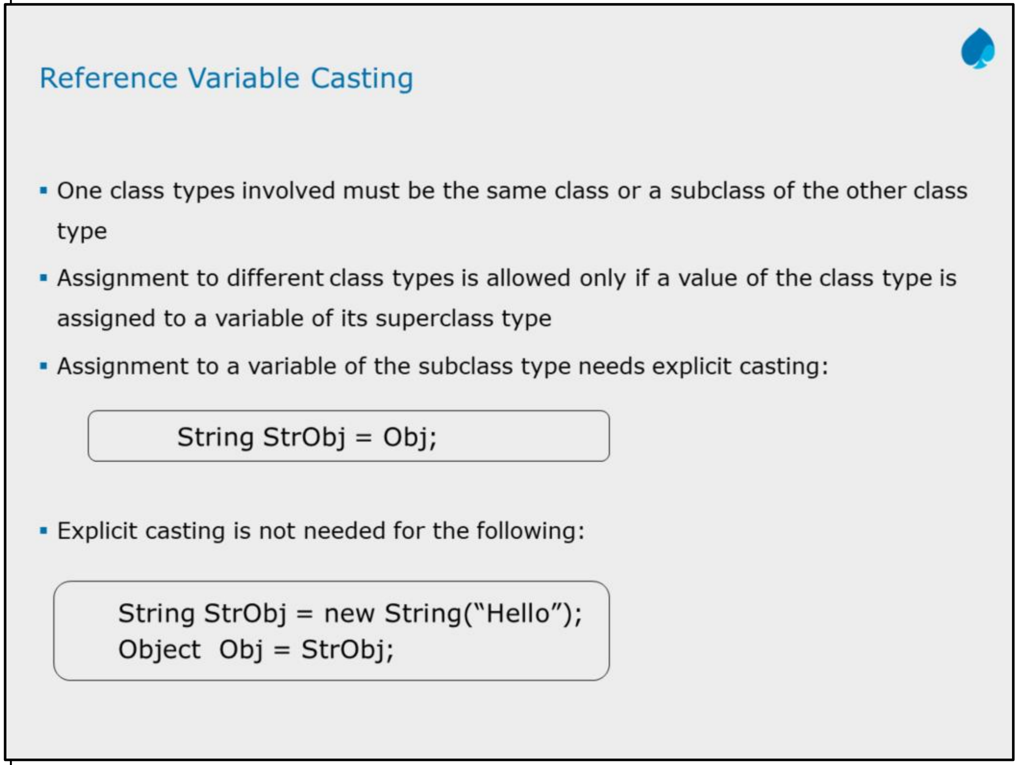
While it's not required to use this annotation while overriding a method, it helps to prevent errors. If a method marked with **@Override** fails to correctly override a method in one of its superclasses, the compiler generates an error.

Most commonly, it is useful when a method in the base class is changed to have a different parameter list. A method in a subclass that used to override the superclass method no longer does so due to the changed method signature. This can sometimes cause strange and unexpected behavior, especially while dealing with complex inheritance structures. The **@Override** annotation safeguards against this. **@Override** is useful in detecting changes in parent classes which has not been reported down the hierarchy. Without it, a method signature can be changed and altering its overrides can be forgotten. With **@Override**, the compiler catches it for you.

Other annotations supported in Java SE:

The **@SuppressWarnings** annotation instructs the compiler to suppress the warning messages it normally shows during compilation time.

@Deprecated marks an old method as deprecated. Which says this method must not be used anymore because in the future versions, this old method may not be supported.



Reference Variable Casting

- One class types involved must be the same class or a subclass of the other class type
- Assignment to different class types is allowed only if a value of the class type is assigned to a variable of its superclass type
- Assignment to a variable of the subclass type needs explicit casting:

```
String StrObj = Obj;
```

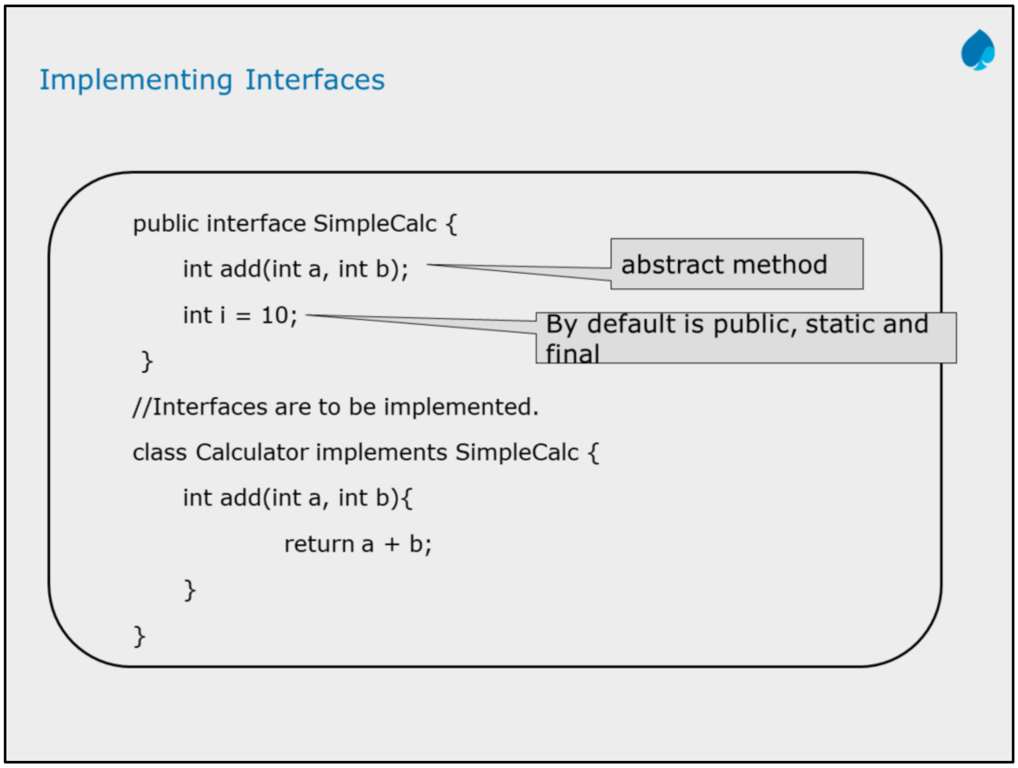
- Explicit casting is not needed for the following:

```
String StrObj = new String("Hello");
Object  Obj = StrObj;
```

The first rule of casting between reference types is that one of the class types involved must be the same class as, or a subclass of, the other class type. Assignment to different class type is allowed only if a value of the class type is assigned to a variable of its superclass type. Assignment to a variable of the subclass type needs explicit casting.

```
Object Obj = new Object ( );
String StrObj = Obj;
/*    The second statement of the above code is not a legitimate one,
because class String is a subclass of class object.  So, an explicit casting is
needed. This is called as downcasting
*/
String StrObj = (String) Obj;
//The reverse statement will not require casting. It is upcasting
String StrObj = new String("Hello");
Object  Obj = StrObj;
```

Rule number one in casting is that you cannot cast a primitive type to a reference type, nor can you cast the other way around. If a casting violation is detected at runtime, the exception ClassCastException is thrown.



Implementing Interfaces

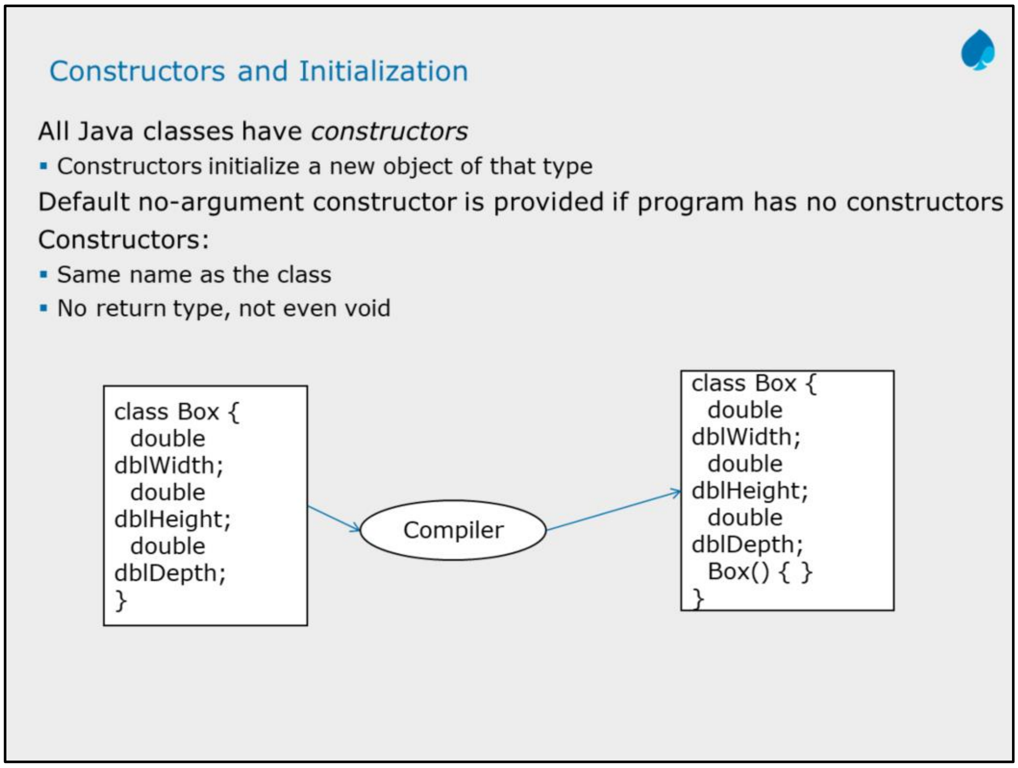
```
public interface SimpleCalc {
     int add(int a, int b);
     int i = 10;
 }
//Interfaces are to be implemented.
class Calculator implements SimpleCalc {
     int add(int a, int b){
              return a + b;
     }
}
```

abstract method
By default is public, static and final

An interface declaration can have two other components:
  The public access specifier
  The list of super interfaces

While a class can only extend one other class, an interface can extend any number of interfaces, and an interface cannot extend classes. An interface inherits all constants and methods from its super interface.
The interface body contains method declarations for the methods defined within the interface and constant declarations. All constant values defined in an interface are implicitly public, static and final. Similarly, all methods declared in an interface are implicitly public and abstract. One class can implement more than one interface at a time by separating them using commas.

Note: Once, a class implements an interface it has to override all the methods in that interface; otherwise, a class has to be declared as an abstract class.



Constructors and Initialization

All Java classes have *constructors*

- Constructors initialize a new object of that type

Default no-argument constructor is provided if program has no constructors

Constructors:

- Same name as the class
- No return type, not even void

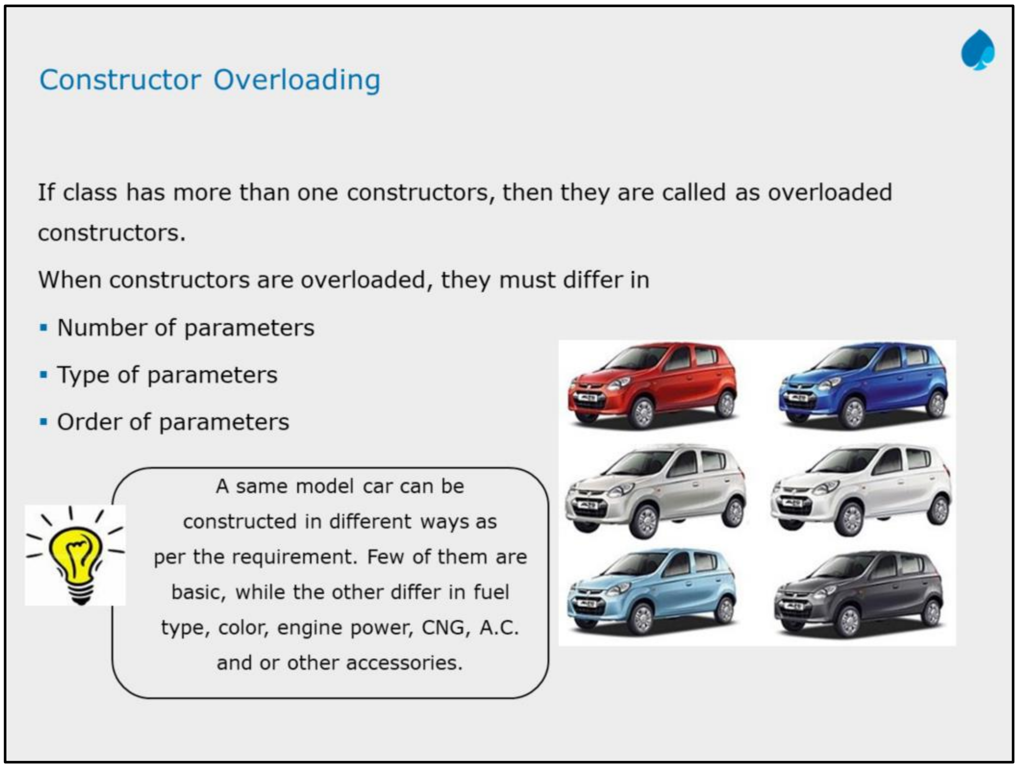
```
class Box {
  double
dblWidth;
  double
dblHeight;
  double
dblDepth;
}
```

```
class Box {
  double
dblWidth;
  double
dblHeight;
  double
dblDepth;
  Box() { }
}
```

Class basically holds blueprint of Objects. In order to create and instantiate objects of class and also to provide initial values for the object, constructors are used. The structure of a constructor looks similar to a method. Constructors are used to create objects of a class. A Java class must have at least one constructor.

A class can have many constructors in order to facilitate the object creation in different ways.

Default Constructor: If class doesn't declare any constructor, compiler adds a constructor to the class. This constructor is called as default constructor. The default constructor accepts no arguments. Above slide shows an example of default constructor.



Constructor Overloading

If class has more than one constructors, then they are called as overloaded constructors.

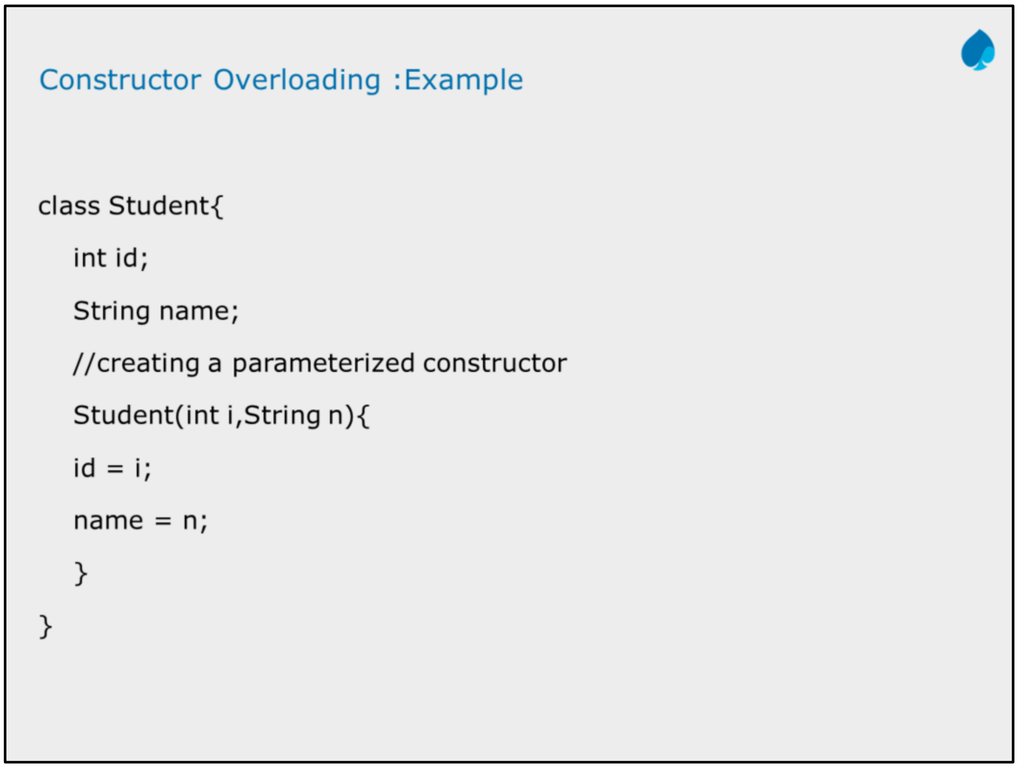
When constructors are overloaded, they must differ in

- Number of parameters
- Type of parameters
- Order of parameters

A same model car can be constructed in different ways as per the requirement. Few of them are basic, while the other differ in fuel type, color, engine power, CNG, A.C. and or other accessories.

Example:

```
class Car {
  int noOfCylinders;
  int noOfValves;
  int enginePower;
  boolean isPowerSteering;
  Car(){
   //1. default no-argument constructor
       noOfCylinders = 3;
       noOfValves = 4;
       enginePower = 48;               //48 ps
       isPowerSteering = false;
  }
  Car(boolean isPowerSteering){
   // 2. constructor with 1 arg
       this();
       this.isPowerSteering = isPowerSteering;
  }
  Car(int enginePower,int noOfCylinders, int noOfValves){
   // 3. constructor with // 3 args
       this.noOfCylinders = noOfCylinders;
       this.noOfValves = noOfValves ;
       this.enginePower = enginePower;
       this.isPowerSteering = true;
  }
}
```

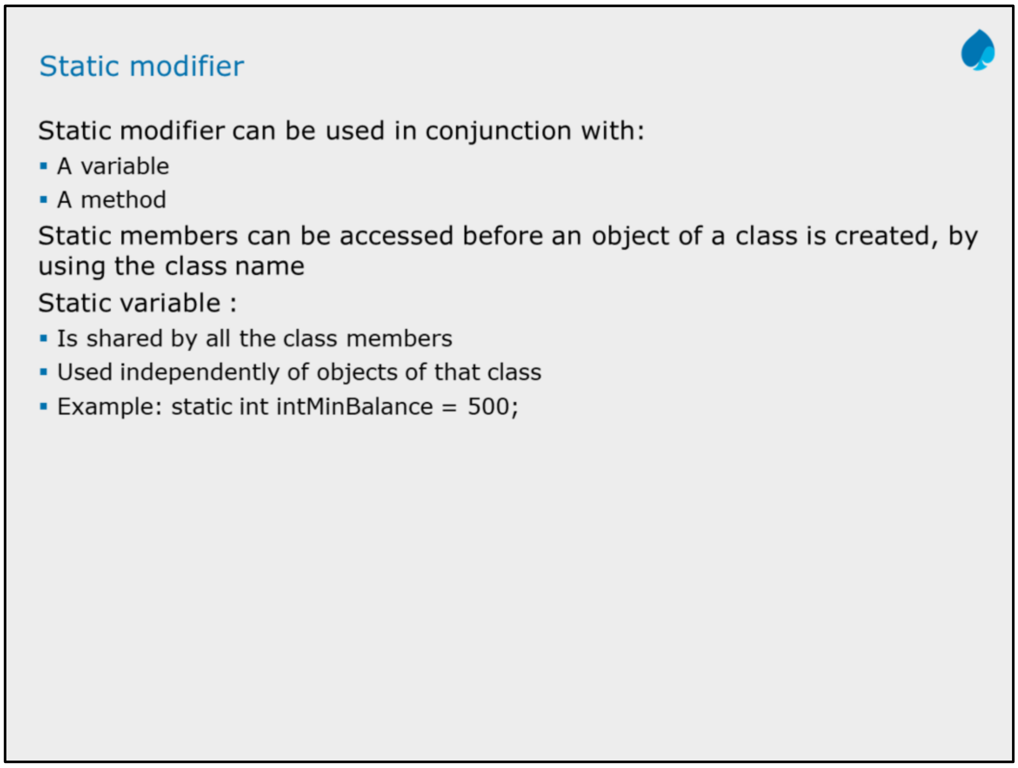


Constructor Overloading :Example

```
class Student{
   int id;
   String name;
   //creating a parameterized constructor
   Student(int i,String n){
   id = i;
   name = n;
   }
}
```

Example:

```
class Car {
  int noOfCylinders;
  int noOfValves;
  int enginePower;
  boolean isPowerSteering;
  Car(){
   //1. default no-argument constructor
      noOfCylinders = 3;
      noOfValves = 4;
      enginePower = 48;             //48 ps
      isPowerSteering = false;
  }
  Car(boolean isPowerSteering){
   // 2. constructor with 1 arg
      this();
      this.isPowerSteering = isPowerSteering;
  }
  Car(int enginePower,int noOfCylinders, int noOfValves){
   // 3. constructor with // 3 args
      this.noOfCylinders = noOfCylinders;
      this.noOfValves = noOfValves ;
      this.enginePower = enginePower;
      this.isPowerSteering = true;
  }
}
```



Static modifier

Static modifier can be used in conjunction with:

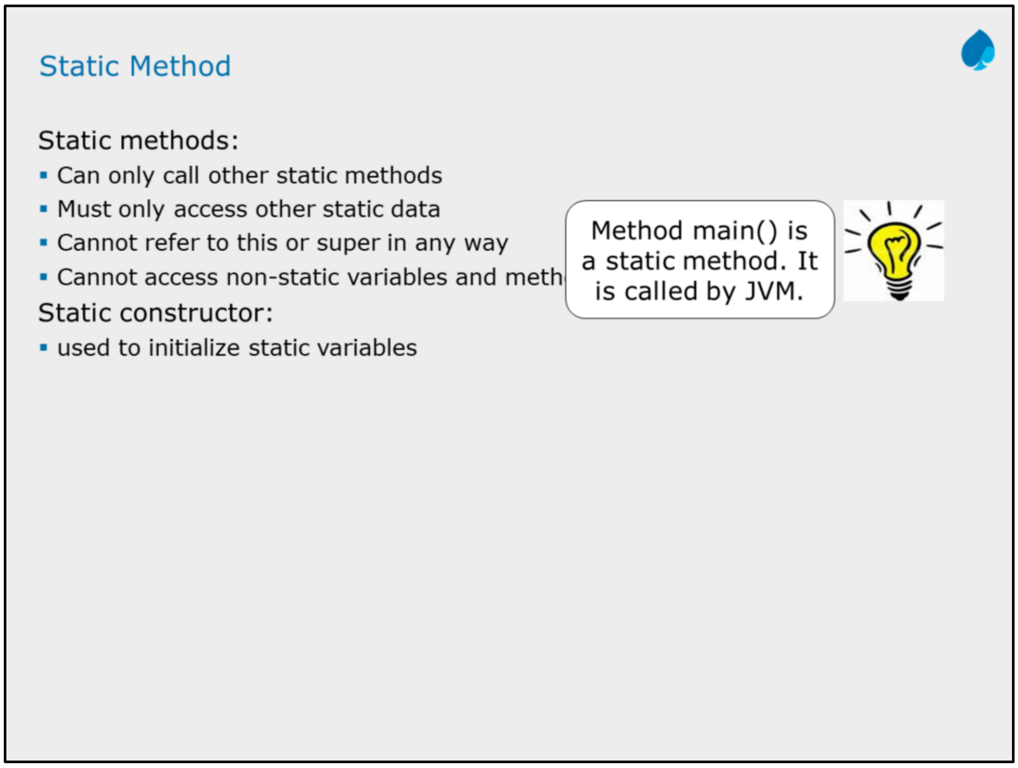
- A variable
- A method

Static members can be accessed before an object of a class is created, by using the class name

Static variable :

- Is shared by all the class members
- Used independently of objects of that class
- Example: static int intMinBalance = 500;

Variables and methods marked with static modifier belong to the class rather than any particular instance. Ie, you do not have to instantiate the class to invoke a static method or access a static variable.



Static Method

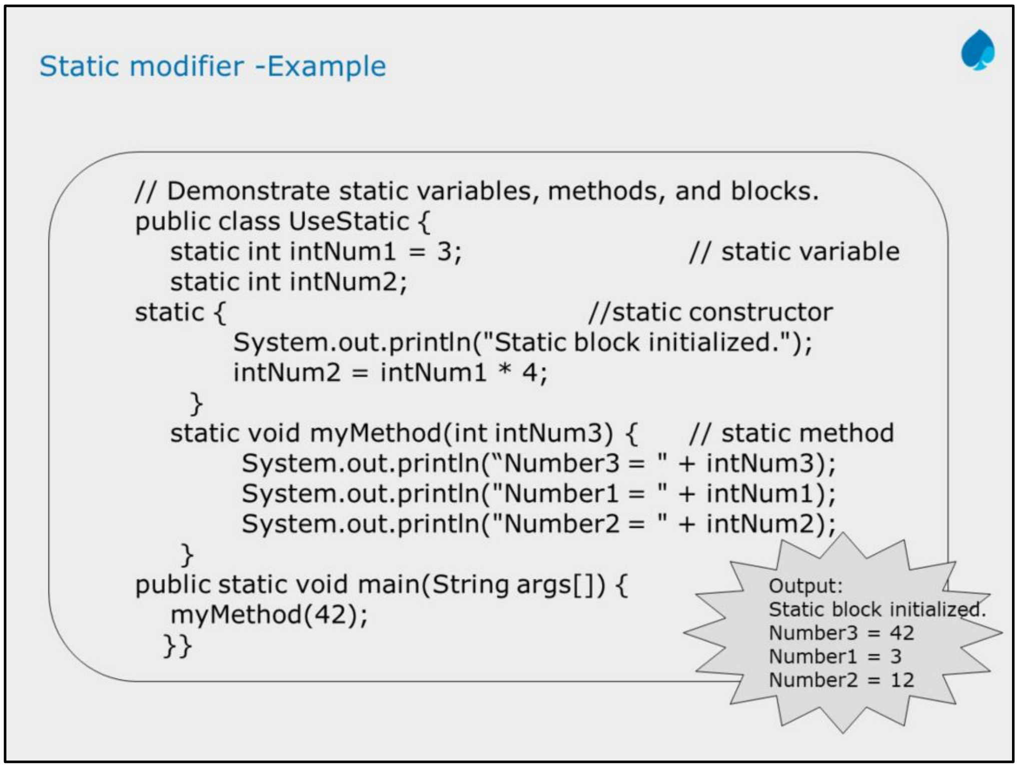
Static methods:
- Can only call other static methods
- Must only access other static data
- Cannot refer to this or super in any way
- Cannot access non-static variables and meth

Static constructor:
- used to initialize static variables

Method main() is a static method. It is called by JVM.

main() is called by JVM. Making this method static means the JVM does not have to create an instance of your class to start running code.

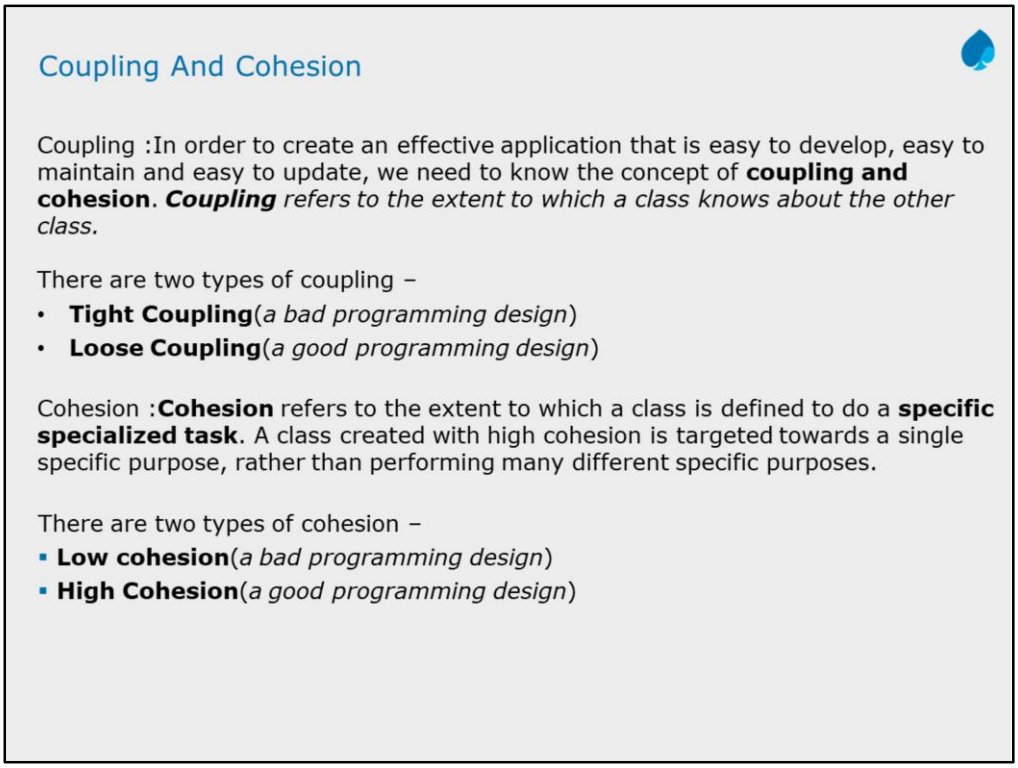
Static constructor is also known as static initialization block. This is a normal block of code enclosed in braces { } and preceded by the static keyword. This can appear anywhere in the class body. It is normally used to initialize static variables.



Static modifier -Example

```
// Demonstrate static variables, methods, and blocks.
public class UseStatic {
    static int intNum1 = 3;                        // static variable
    static int intNum2;
static {                                       //static constructor
           System.out.println("Static block initialized.");
           intNum2 = intNum1 * 4;
      }
    static void myMethod(int intNum3) {      // static method
            System.out.println("Number3 = " + intNum3);
            System.out.println("Number1 = " + intNum1);
            System.out.println("Number2 = " + intNum2);
     }
public static void main(String args[]) {
    myMethod(42);
   }}
```

```
Output:
Static block initialized.
Number3 = 42
Number1 = 3
Number2 = 12
```



Coupling And Cohesion

Coupling :In order to create an effective application that is easy to develop, easy to maintain and easy to update, we need to know the concept of **coupling and cohesion**. ***Coupling*** *refers to the extent to which a class knows about the other class.*

There are two types of coupling –

- **Tight Coupling**(*a bad programming design*)
- **Loose Coupling**(*a good programming design*)

Cohesion :**Cohesion** refers to the extent to which a class is defined to do a **specific specialized task**. A class created with high cohesion is targeted towards a single specific purpose, rather than performing many different specific purposes.

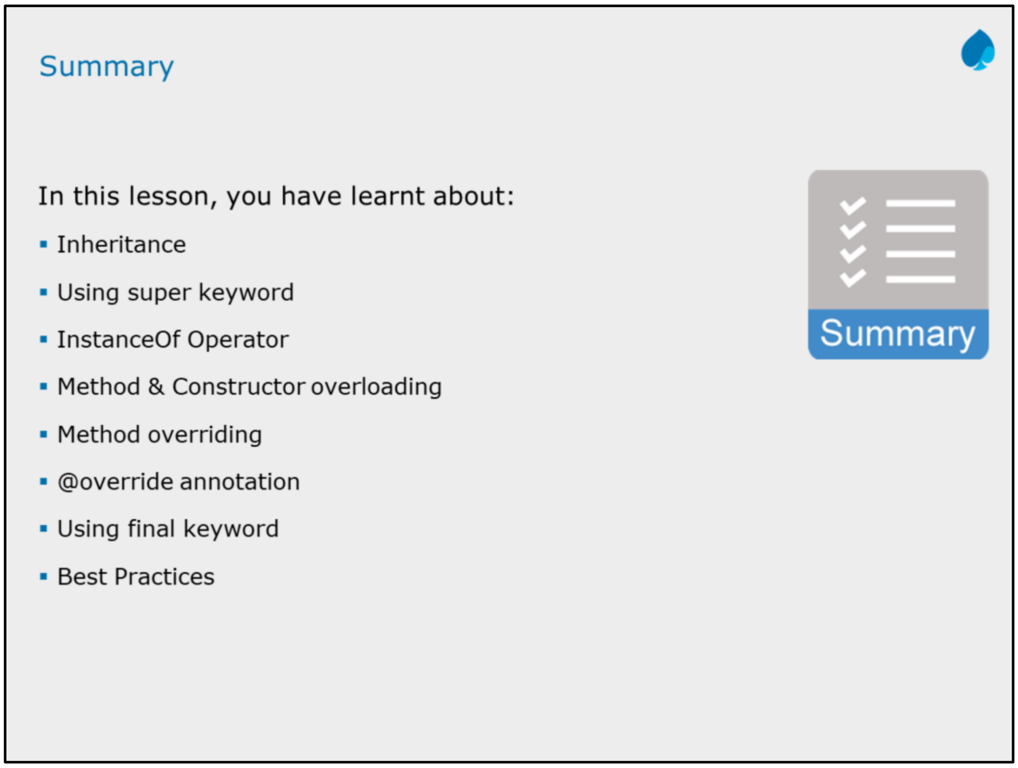
There are two types of cohesion –

- **Low cohesion**(*a bad programming design*)
- **High Cohesion**(*a good programming design*)

Demo

Polymorphism

- Method Overloading
- Method Overriding

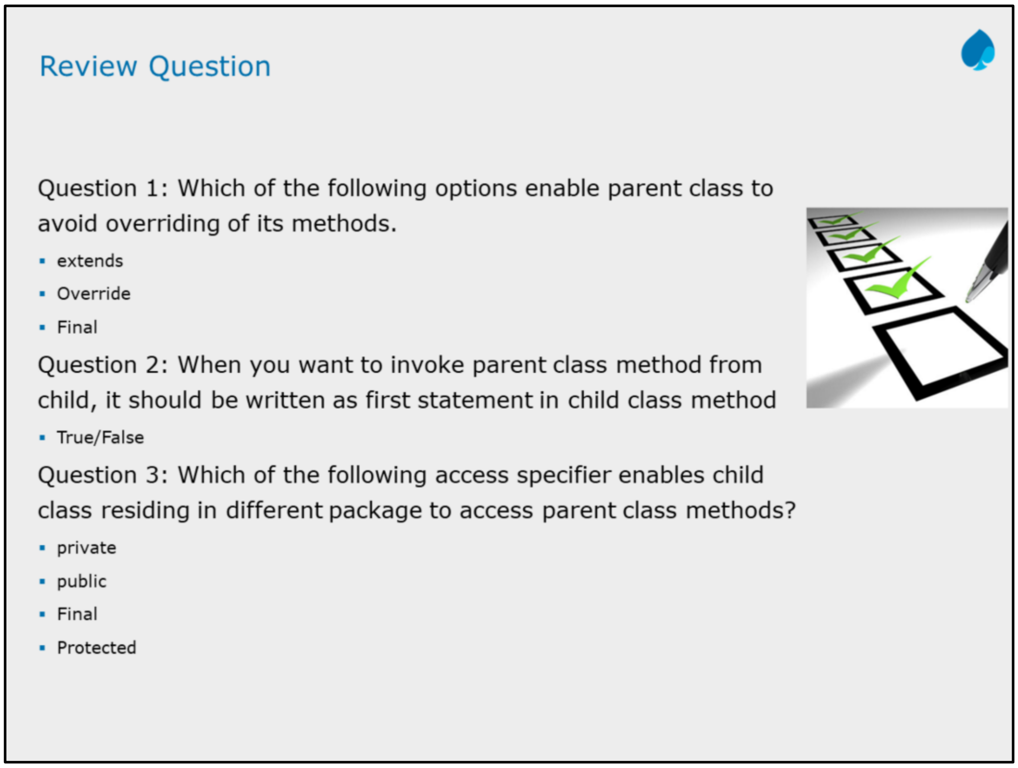


Summary

In this lesson, you have learnt about:

- Inheritance
- Using super keyword
- InstanceOf Operator
- Method & Constructor overloading
- Method overriding
- @override annotation
- Using final keyword
- Best Practices

Summary



Review Question

Question 1: Which of the following options enable parent class to avoid overriding of its methods.

- extends
- Override
- Final

Question 2: When you want to invoke parent class method from child, it should be written as first statement in child class method

- True/False

Question 3: Which of the following access specifier enables child class residing in different package to access parent class methods?

- private
- public
- Final
- Protected