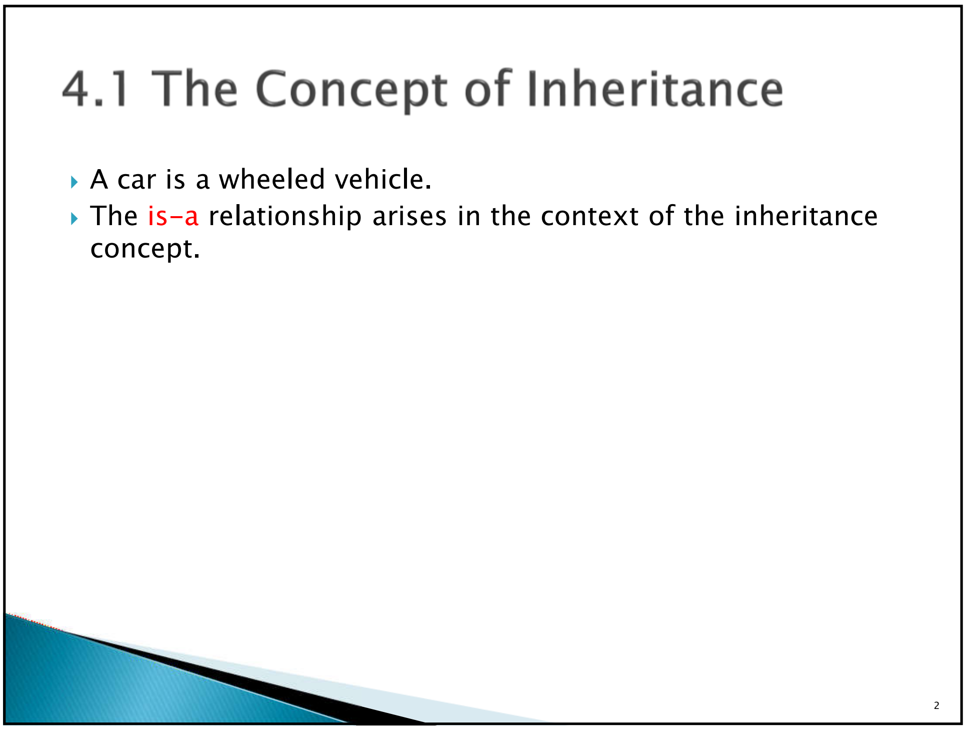
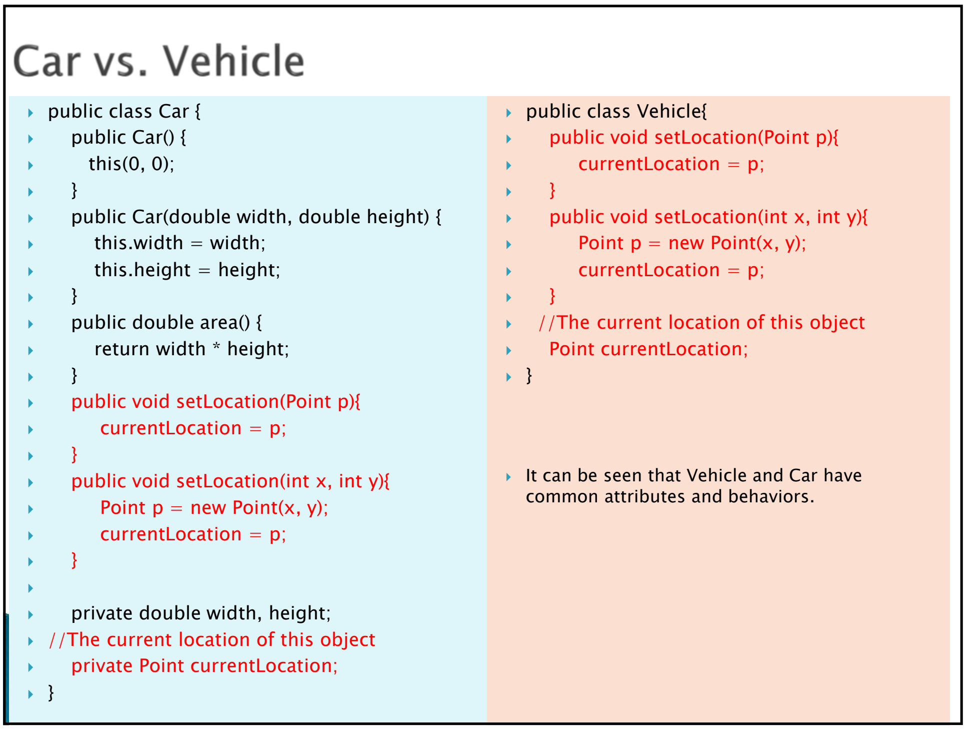
Inheritance, Interfaces and Polymorphism



4.1 The Concept of Inheritance

- A car is a wheeled vehicle.
- The is-a relationship arises in the context of the inheritance concept.

2

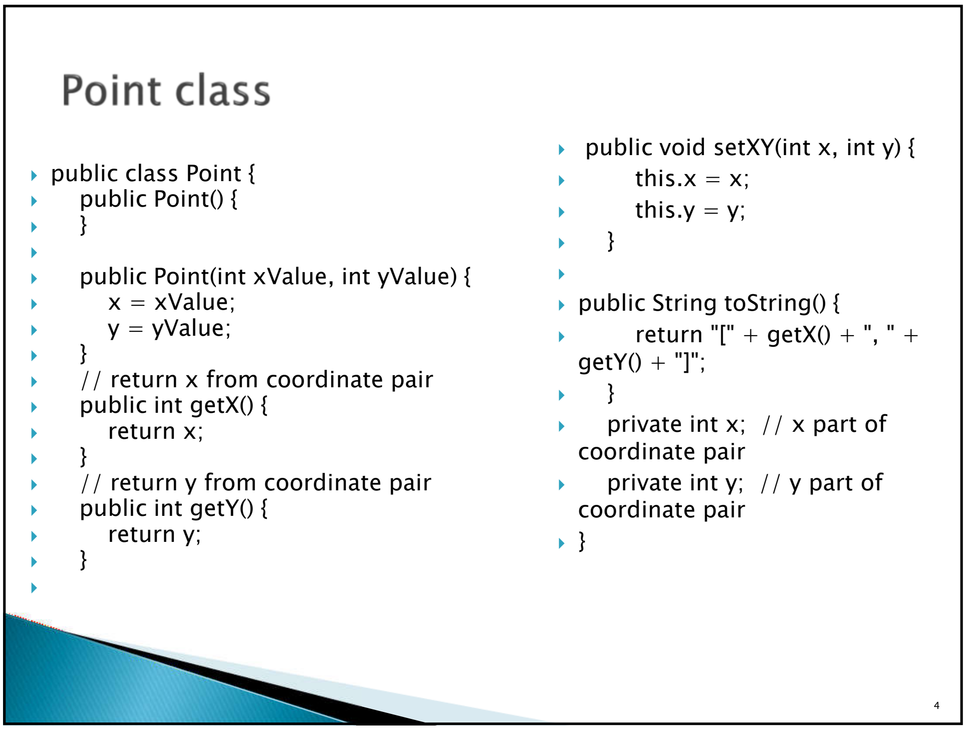


Car vs. Vehicle

```
public class Car {
    public Car() {
      this(0, 0);
    }
    public Car(double width, double height) {
       this.width = width;
       this.height = height;
    }
    public double area() {
       return width * height;
    }
    public void setLocation(Point p){
        currentLocation = p;
    }
    public void setLocation(int x, int y){
        Point p = new Point(x, y);
        currentLocation = p;
    }

    private double width, height;
//The current location of this object
    private Point currentLocation;
}
```

```
public class Vehicle{
    public void setLocation(Point p){
        currentLocation = p;
    }
    public void setLocation(int x, int y){
        Point p = new Point(x, y);
        currentLocation = p;
    }
  //The current location of this object
    Point currentLocation;
}
```

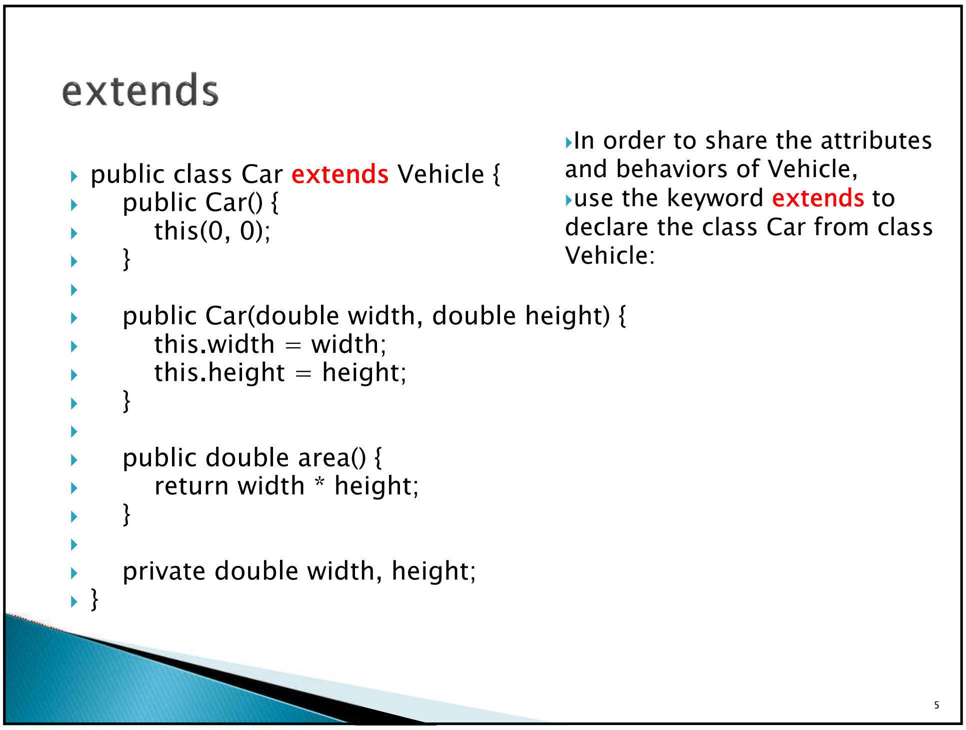
- It can be seen that Vehicle and Car have common attributes and behaviors.



Point class

```
public class Point {
    public Point() {
    }

    public Point(int xValue, int yValue) {
        x = xValue;
        y = yValue;
    }
    // return x from coordinate pair
    public int getX() {
        return x;
    }
    // return y from coordinate pair
    public int getY() {
        return y;
    }

 public void setXY(int x, int y) {
        this.x = x;
        this.y = y;
    }

public String toString() {
        return "[" + getX() + ", " +
getY() + "]";
    }
    private int x;  // x part of
coordinate pair
    private int y;  // y part of
coordinate pair
}
```

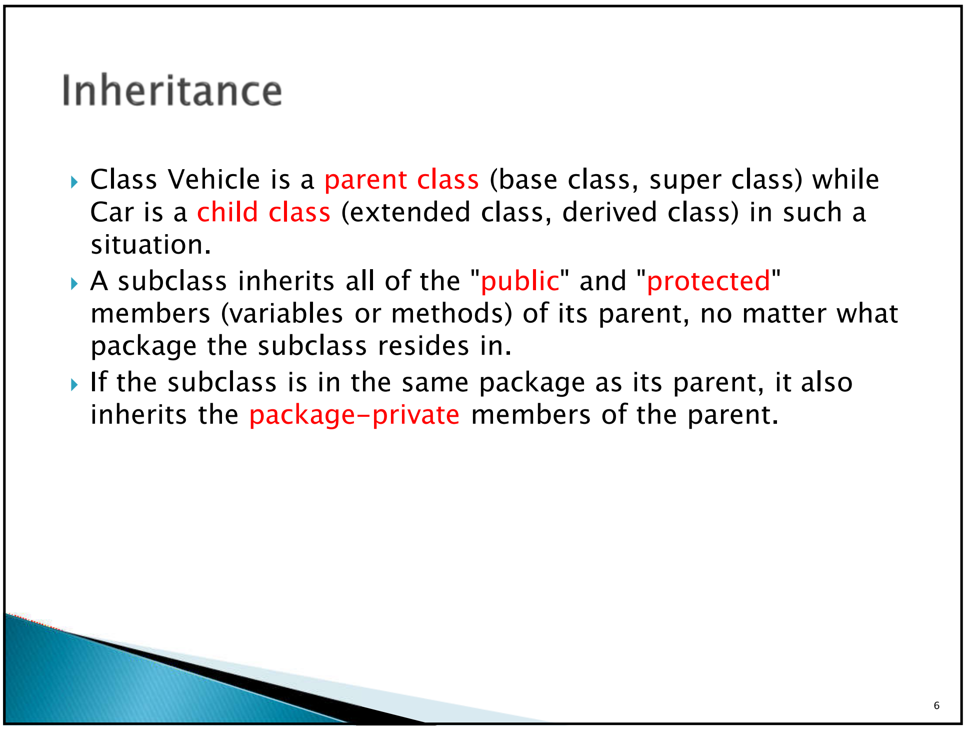
4



extends

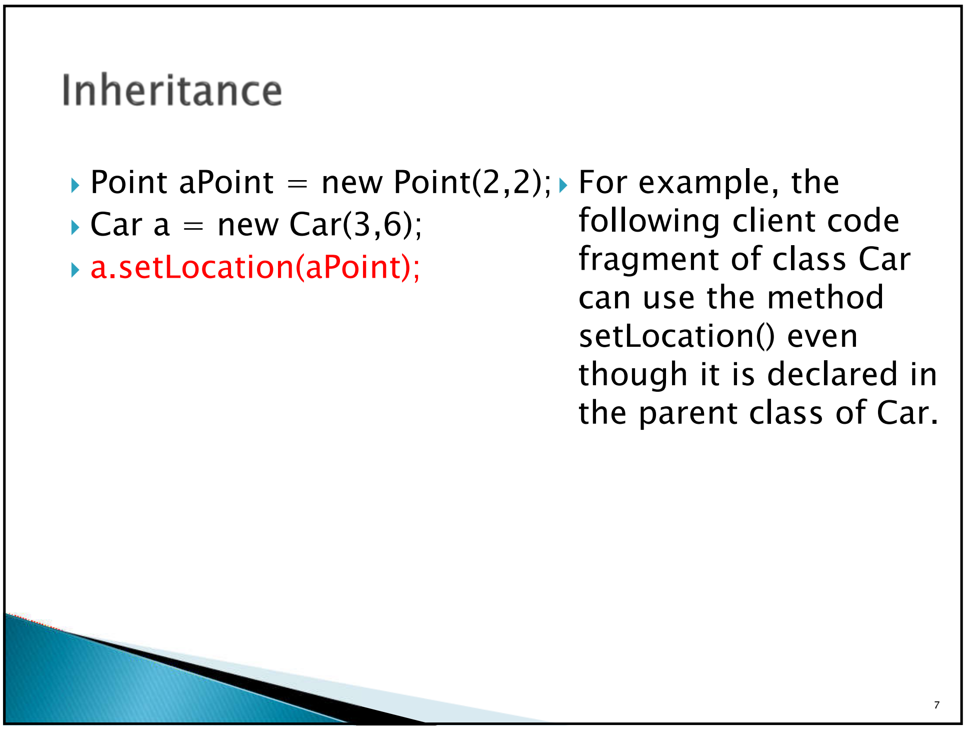
```
public class Car extends Vehicle {
    public Car() {
        this(0, 0);
    }

    public Car(double width, double height) {
        this.width = width;
        this.height = height;
    }

    public double area() {
        return width * height;
    }

    private double width, height;
}
```

- In order to share the attributes and behaviors of Vehicle,
- use the keyword **extends** to declare the class Car from class Vehicle:



Inheritance

- Class Vehicle is a parent class (base class, super class) while Car is a child class (extended class, derived class) in such a situation.
- A subclass inherits all of the "public" and "protected" members (variables or methods) of its parent, no matter what package the subclass resides in.
- If the subclass is in the same package as its parent, it also inherits the package-private members of the parent.

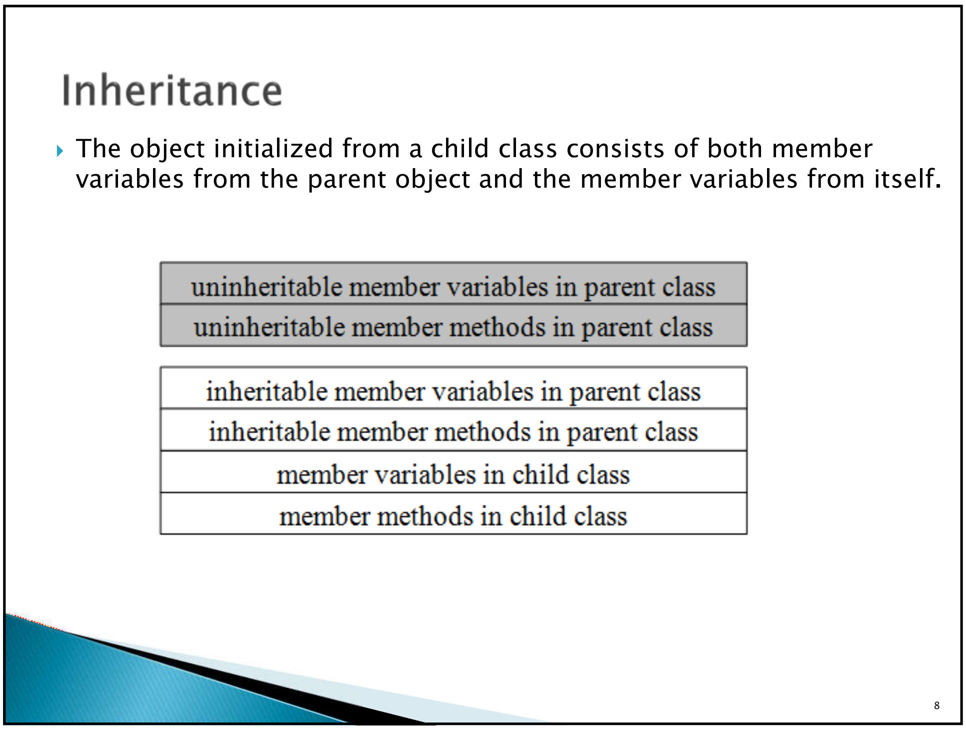


Inheritance

- Point aPoint = new Point(2,2);
- Car a = new Car(3,6);
- a.setLocation(aPoint);

- For example, the following client code fragment of class Car can use the method setLocation() even though it is declared in the parent class of Car.

7



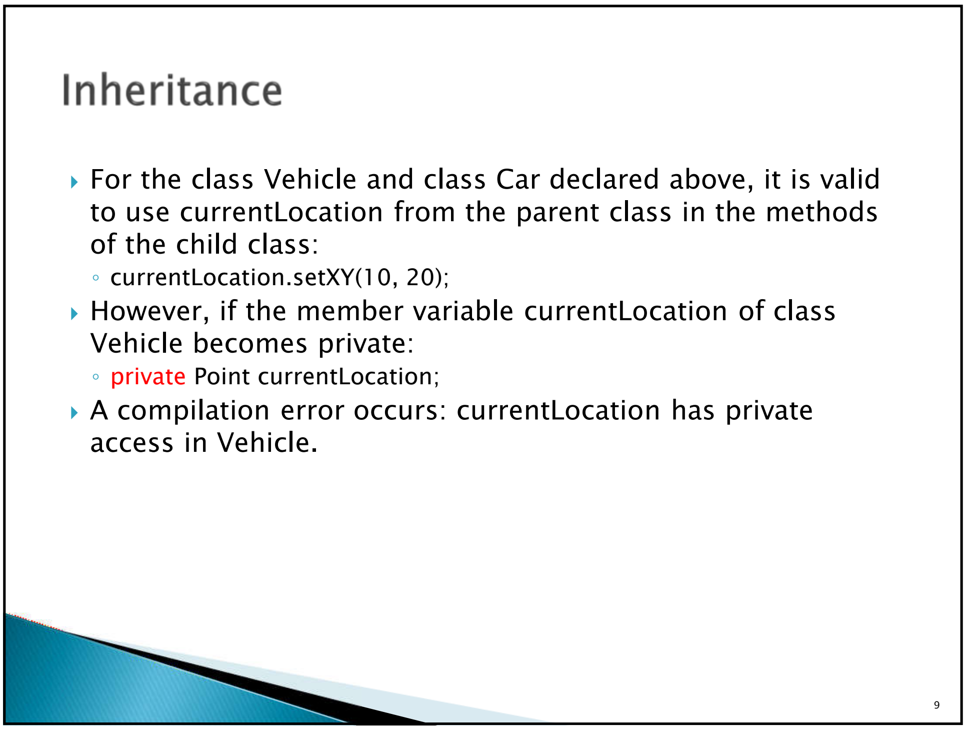
Inheritance

- The object initialized from a child class consists of both member variables from the parent object and the member variables from itself.

uninheritable member variables in parent class
uninheritable member methods in parent class

inheritable member variables in parent class
inheritable member methods in parent class
member variables in child class
member methods in child class

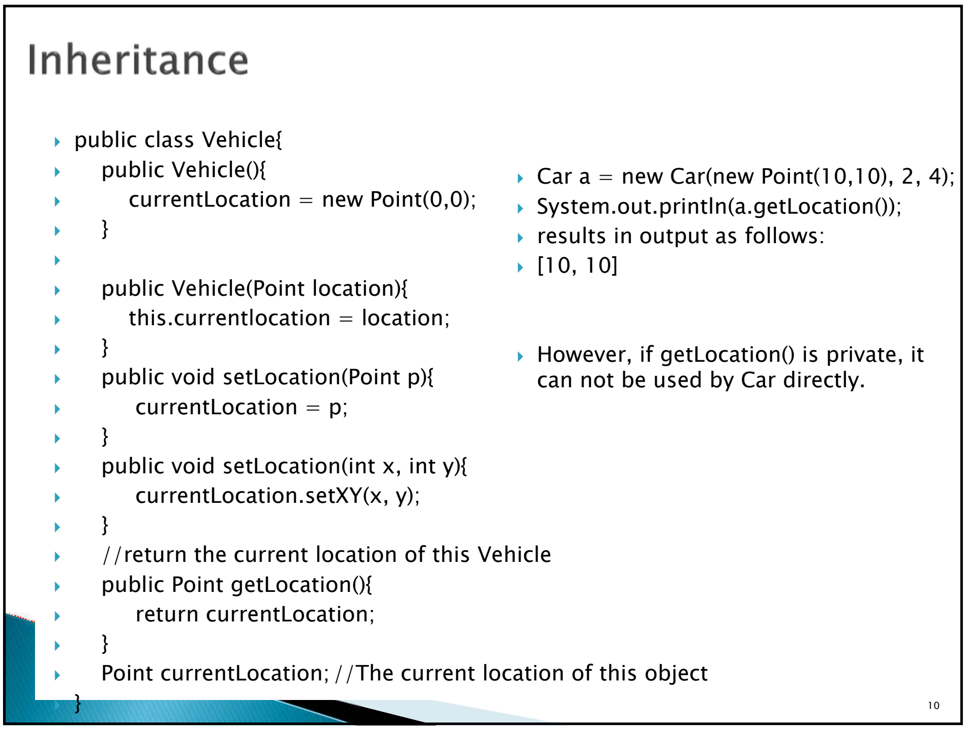
8



Inheritance

- For the class Vehicle and class Car declared above, it is valid to use currentLocation from the parent class in the methods of the child class:
  - currentLocation.setXY(10, 20);
- However, if the member variable currentLocation of class Vehicle becomes private:
  - private Point currentLocation;
- A compilation error occurs: currentLocation has private access in Vehicle.

9



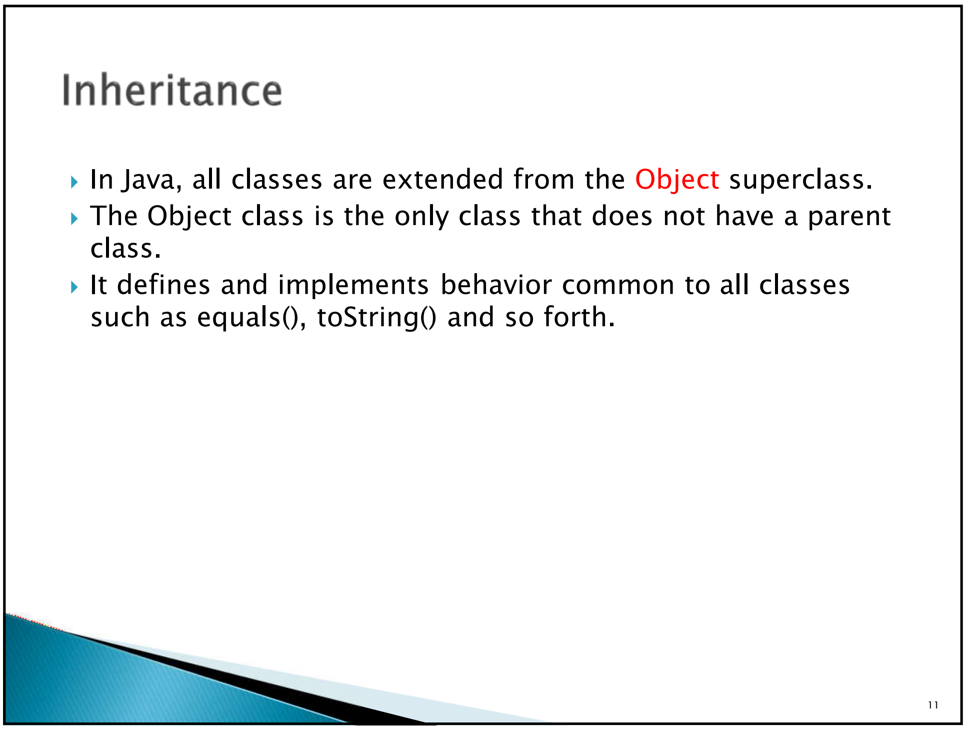
Inheritance

```
public class Vehicle{
    public Vehicle(){
        currentLocation = new Point(0,0);
    }

    public Vehicle(Point location){
        this.currentlocation = location;
    }
    public void setLocation(Point p){
         currentLocation = p;
    }
    public void setLocation(int x, int y){
         currentLocation.setXY(x, y);
    }
    //return the current location of this Vehicle
    public Point getLocation(){
         return currentLocation;
    }
    Point currentLocation; //The current location of this object
}
```

- Car a = new Car(new Point(10,10), 2, 4);
- System.out.println(a.getLocation());
- results in output as follows:
- [10, 10]

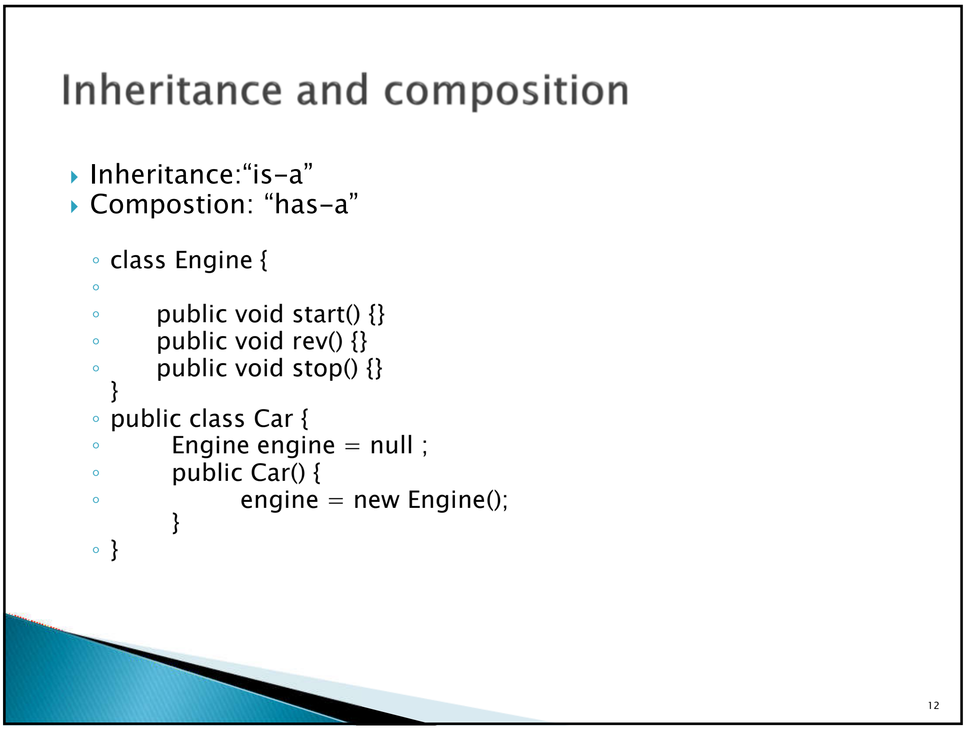
- However, if getLocation() is private, it can not be used by Car directly.

10



Inheritance

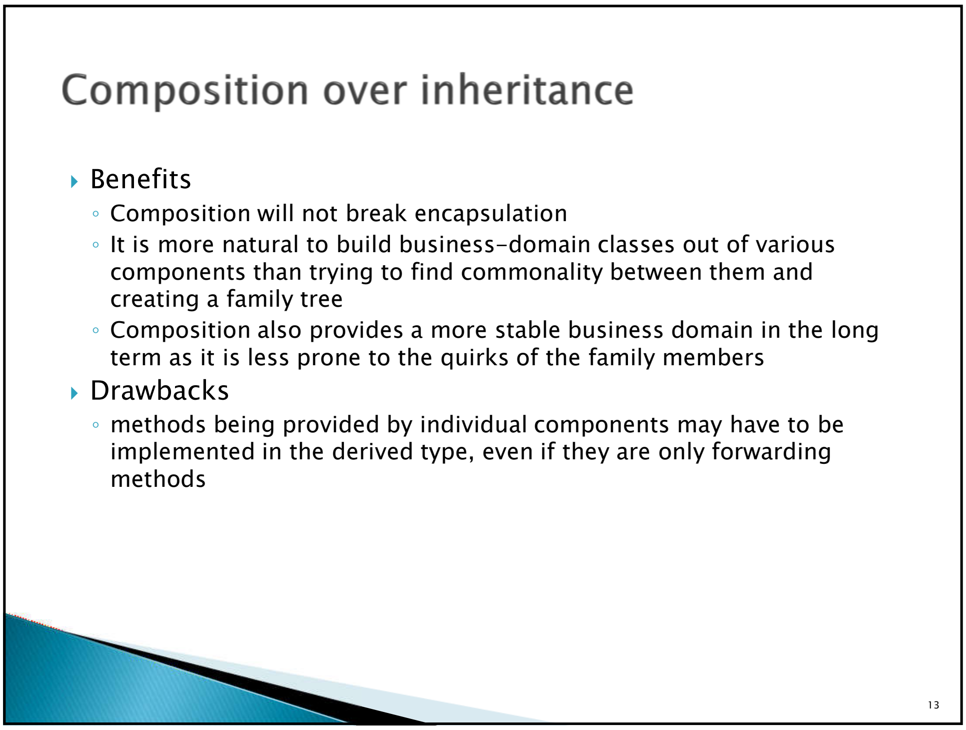
- In Java, all classes are extended from the Object superclass.
- The Object class is the only class that does not have a parent class.
- It defines and implements behavior common to all classes such as equals(), toString() and so forth.



Inheritance and composition

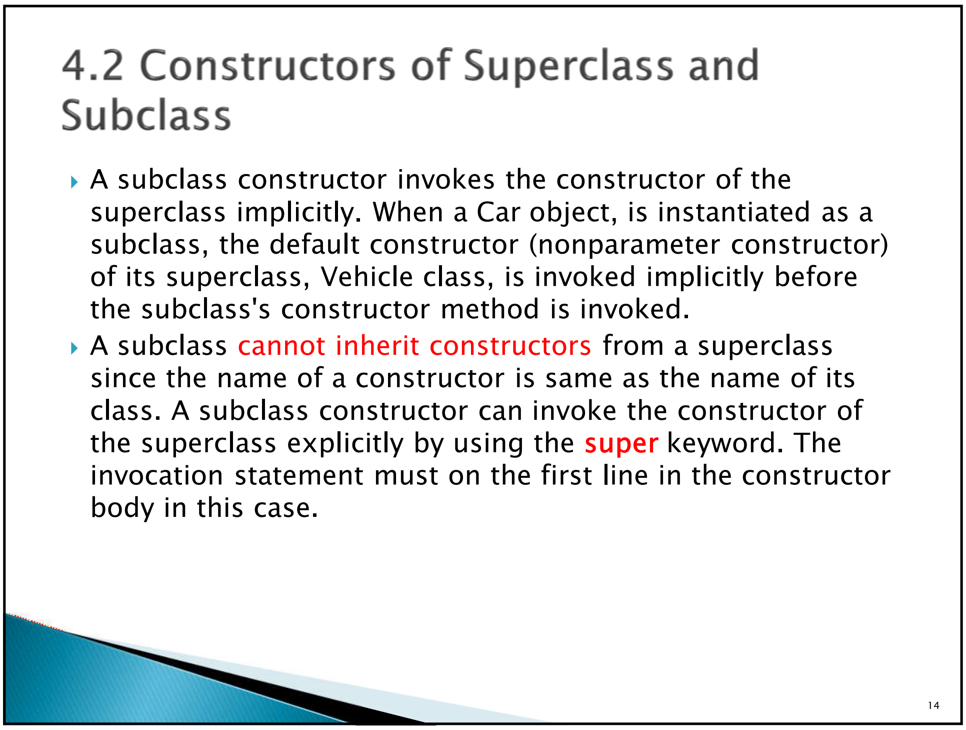
- Inheritance:“is-a”
- Compostion: “has-a”

```
◦ class Engine {
◦
◦       public void start() {}
◦       public void rev() {}
◦       public void stop() {}
  }
◦ public class Car {
◦         Engine engine = null ;
◦         public Car() {
◦                  engine = new Engine();
          }
◦ }
```



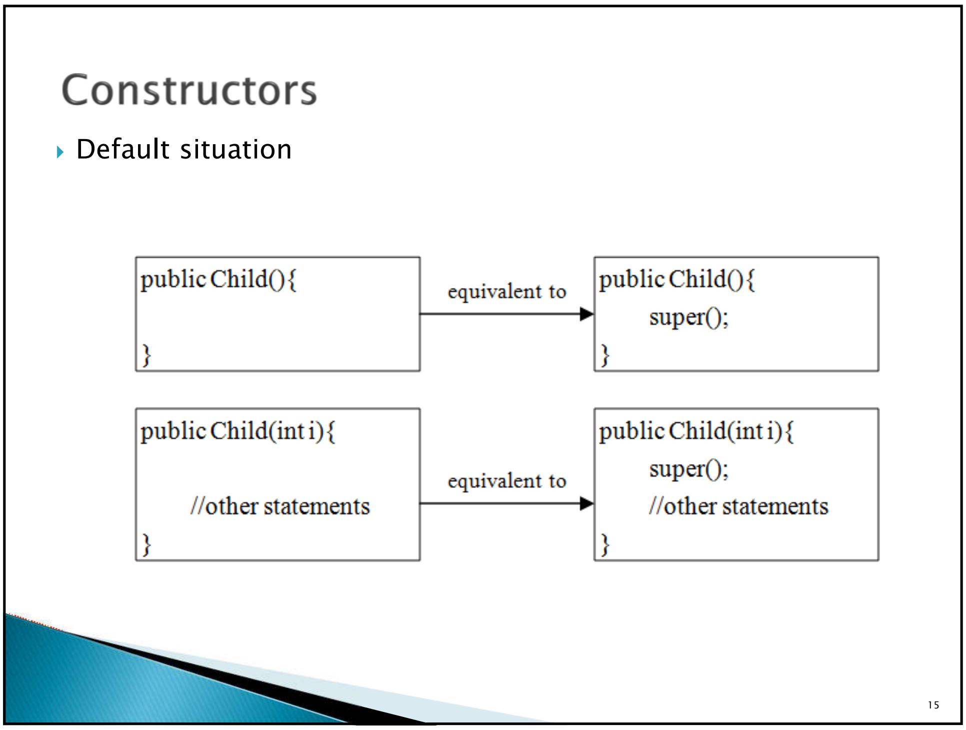
Composition over inheritance

- Benefits
  - Composition will not break encapsulation
  - It is more natural to build business-domain classes out of various components than trying to find commonality between them and creating a family tree
  - Composition also provides a more stable business domain in the long term as it is less prone to the quirks of the family members
- Drawbacks
  - methods being provided by individual components may have to be implemented in the derived type, even if they are only forwarding methods



4.2 Constructors of Superclass and Subclass

- A subclass constructor invokes the constructor of the superclass implicitly. When a Car object, is instantiated as a subclass, the default constructor (nonparameter constructor) of its superclass, Vehicle class, is invoked implicitly before the subclass's constructor method is invoked.
- A subclass cannot inherit constructors from a superclass since the name of a constructor is same as the name of its class. A subclass constructor can invoke the constructor of the superclass explicitly by using the super keyword. The invocation statement must on the first line in the constructor body in this case.



Constructors

- Default situation

```
public Child(){

}
```

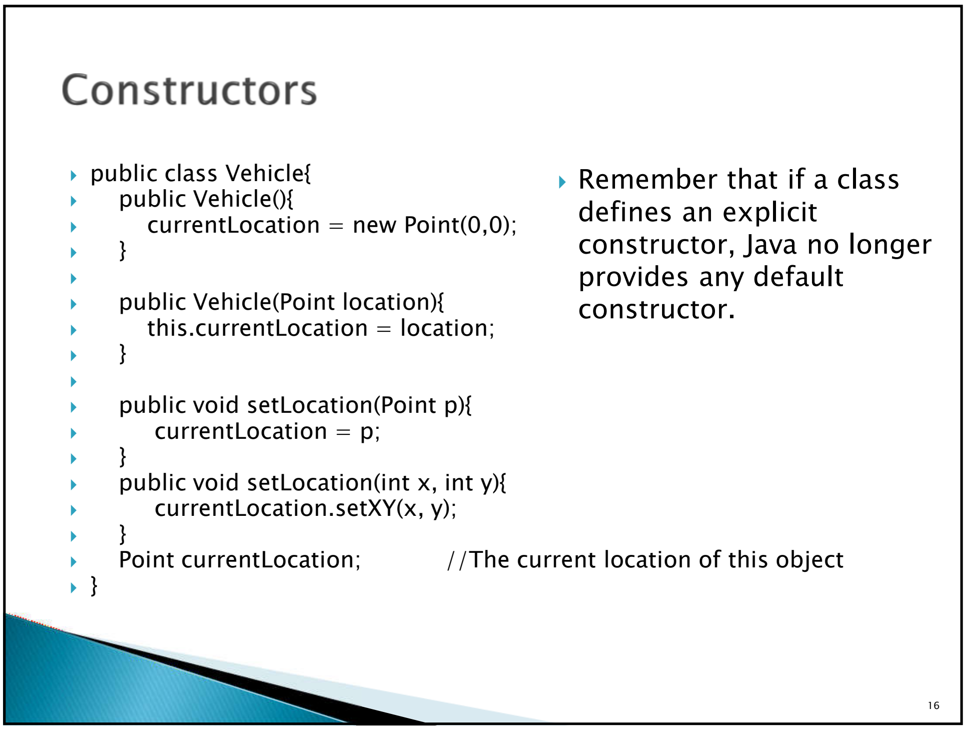
equivalent to

```
public Child(){
      super();
}
```

```
public Child(int i){

      //other statements
}
```

equivalent to

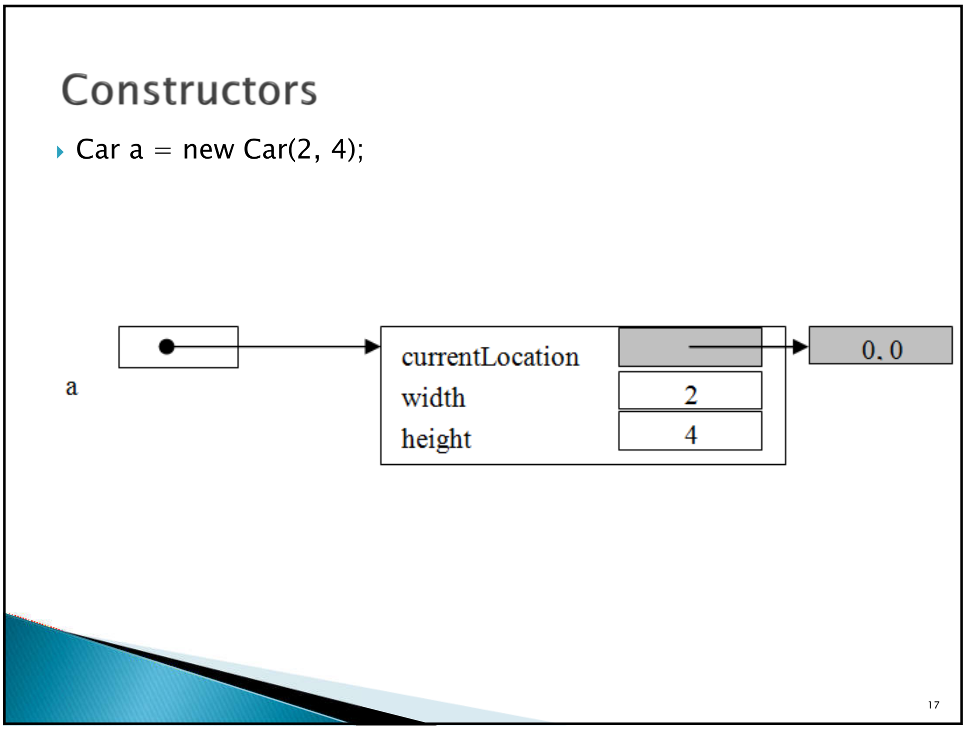
```
public Child(int i){
      super();
      //other statements
}
```



Constructors

```
public class Vehicle{
    public Vehicle(){
        currentLocation = new Point(0,0);
    }

    public Vehicle(Point location){
        this.currentLocation = location;
    }

    public void setLocation(Point p){
         currentLocation = p;
    }
    public void setLocation(int x, int y){
         currentLocation.setXY(x, y);
    }
    Point currentLocation;          //The current location of this object
}
```

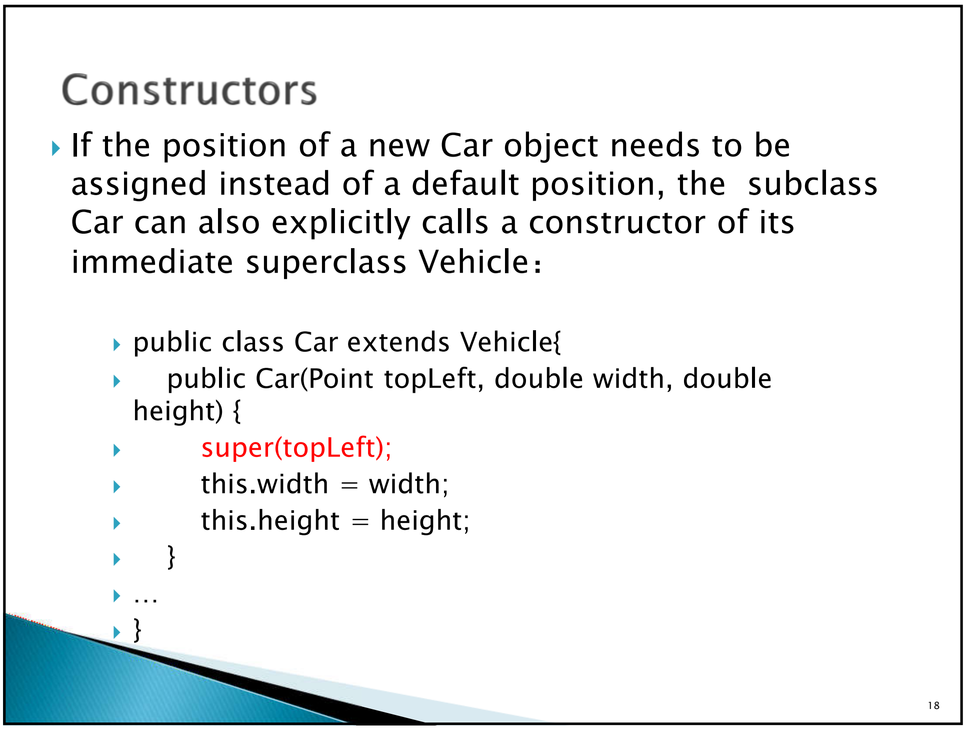
- Remember that if a class defines an explicit constructor, Java no longer provides any default constructor.



Constructors

- Car a = new Car(2, 4);

17

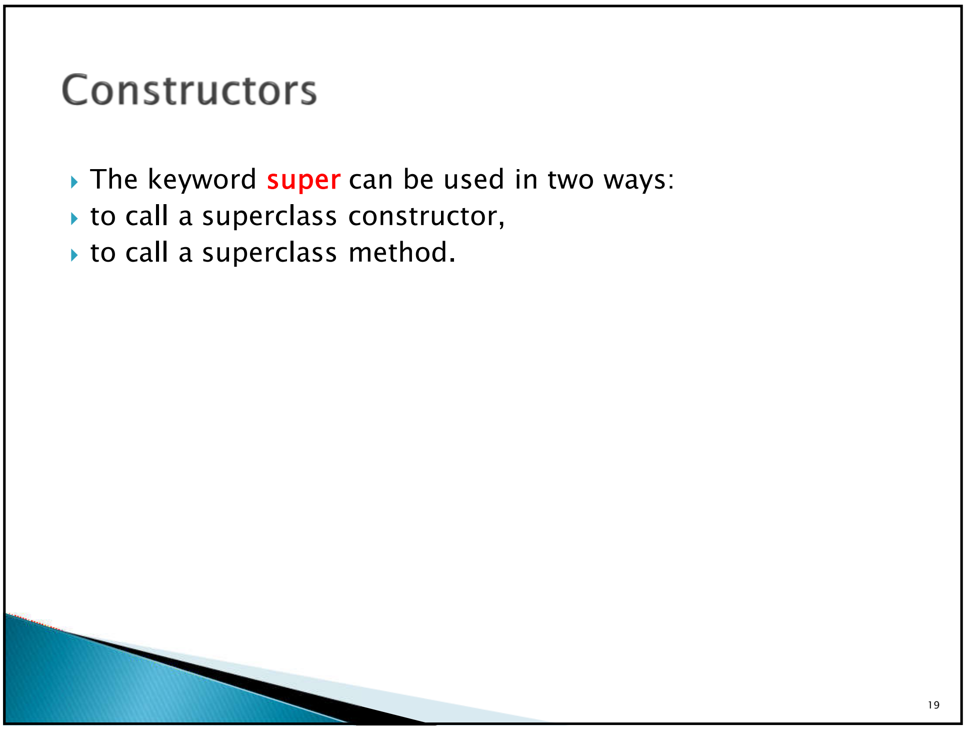


Constructors

- If the position of a new Car object needs to be assigned instead of a default position, the subclass Car can also explicitly calls a constructor of its immediate superclass Vehicle：

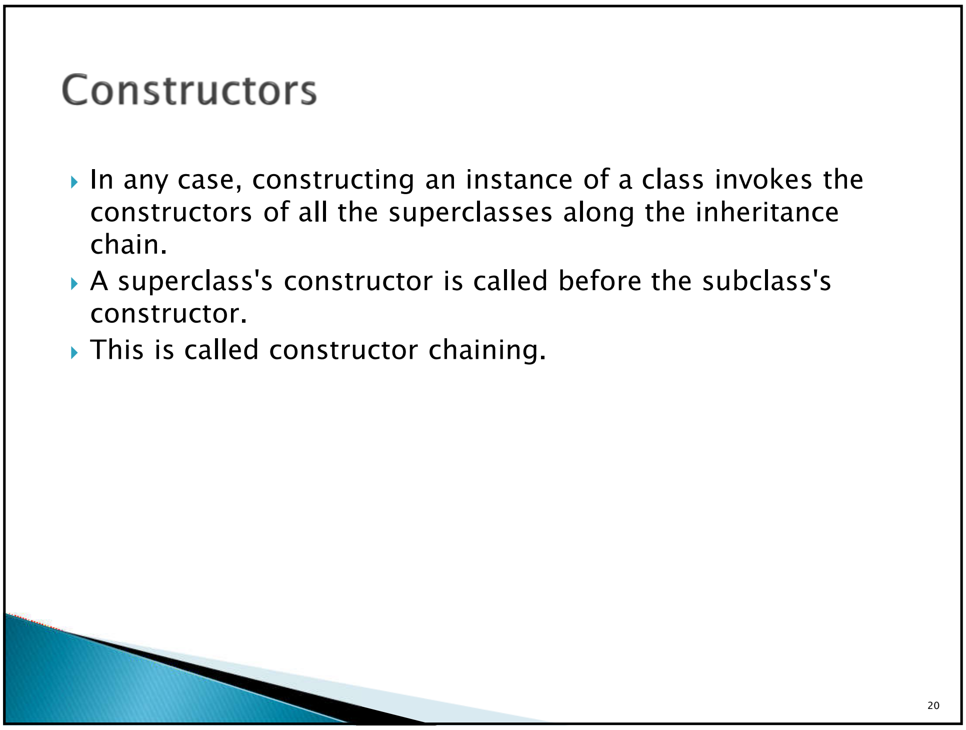
```
public class Car extends Vehicle{
    public Car(Point topLeft, double width, double height) {
        super(topLeft);
        this.width = width;
        this.height = height;
    }
…
}
```

18



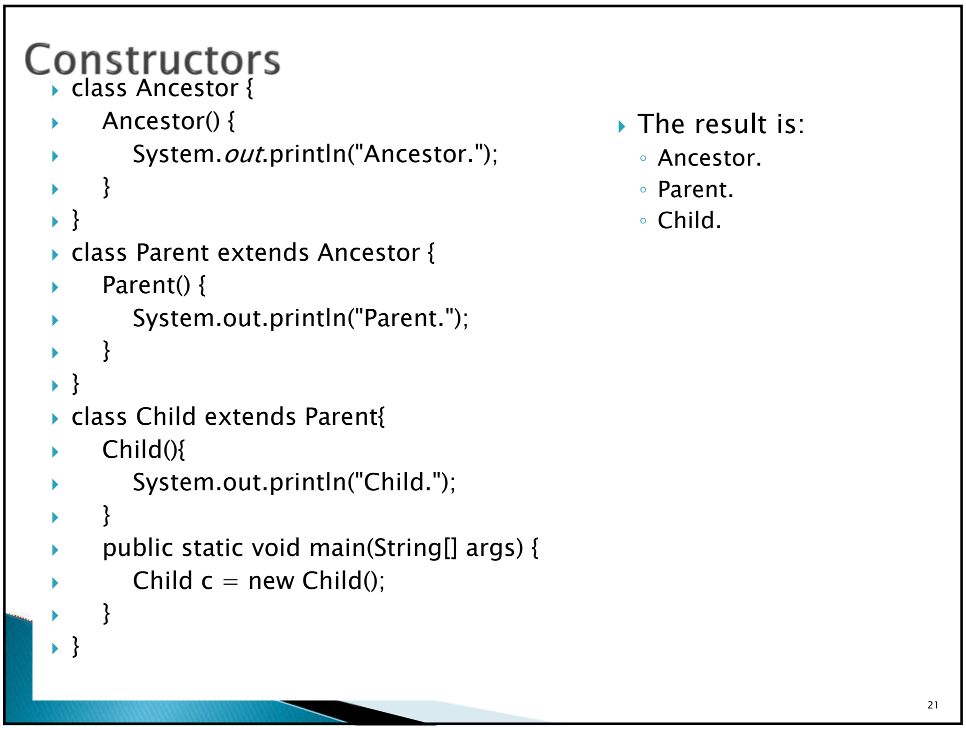
Constructors

- The keyword super can be used in two ways:
- to call a superclass constructor,
- to call a superclass method.



Constructors

- In any case, constructing an instance of a class invokes the constructors of all the superclasses along the inheritance chain.
- A superclass's constructor is called before the subclass's constructor.
- This is called constructor chaining.

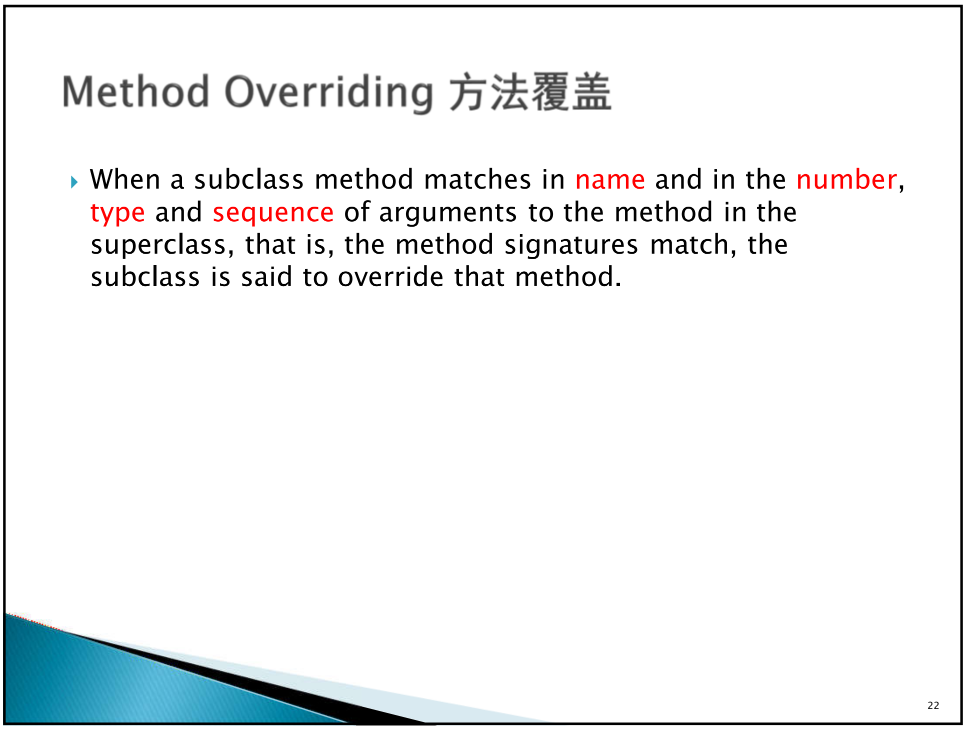


Constructors

```
class Ancestor {
    Ancestor() {
        System.out.println("Ancestor.");
    }
}
class Parent extends Ancestor {
    Parent() {
        System.out.println("Parent.");
    }
}
class Child extends Parent{
    Child(){
        System.out.println("Child.");
    }
    public static void main(String[] args) {
        Child c = new Child();
    }
}
```

- The result is:
  - Ancestor.
  - Parent.
  - Child.

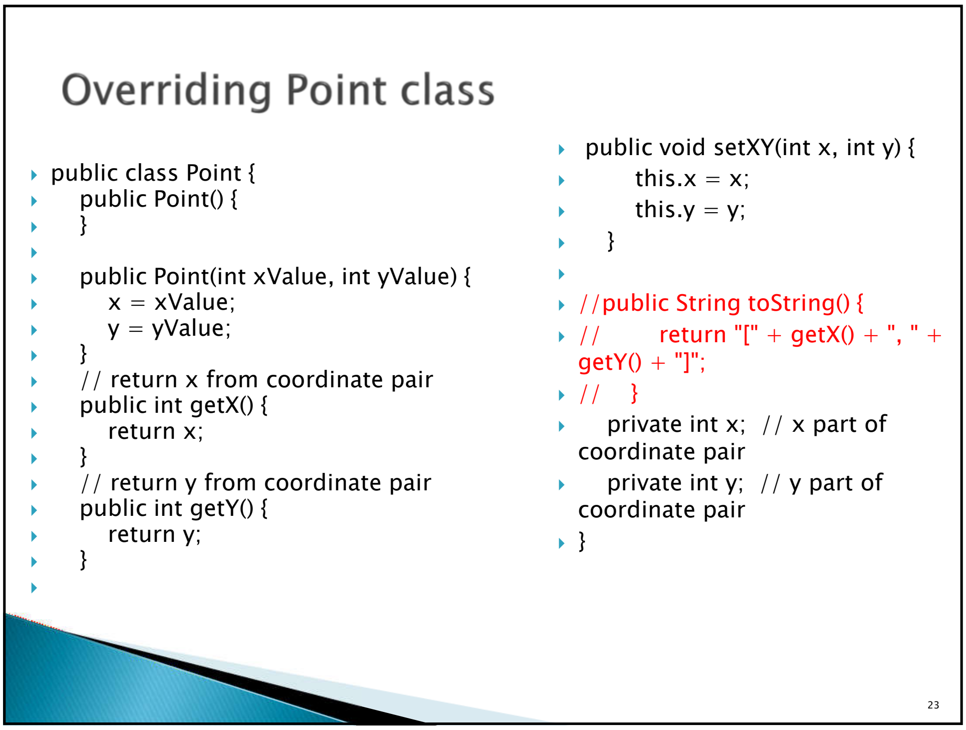
21



Method Overriding 方法覆盖

- When a subclass method matches in name and in the number, type and sequence of arguments to the method in the superclass, that is, the method signatures match, the subclass is said to override that method.

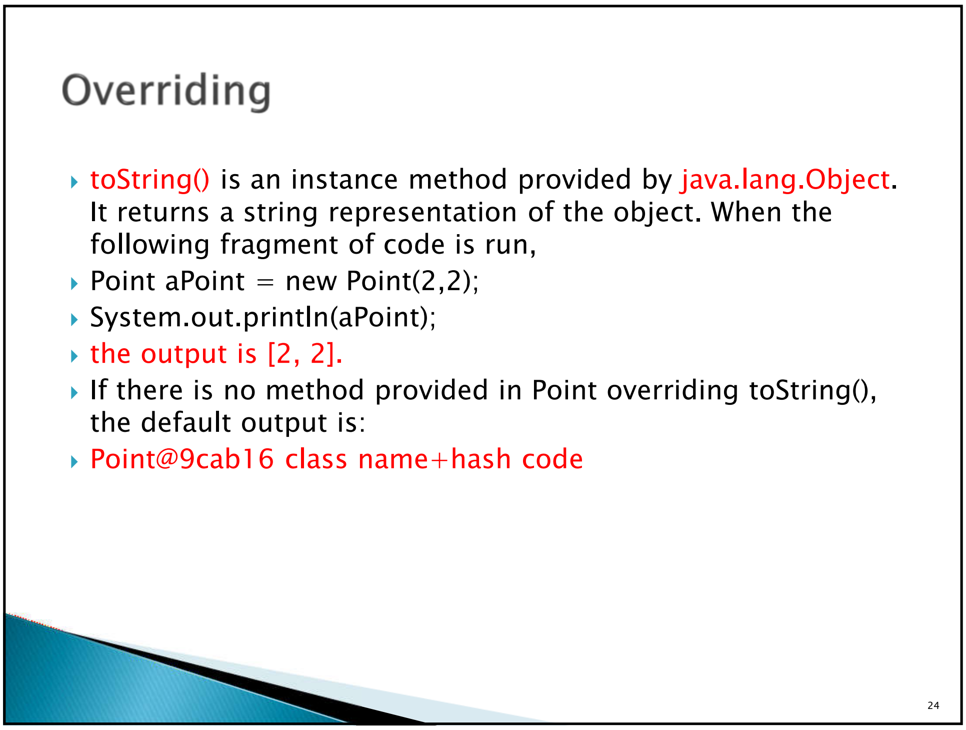
22



Overriding Point class

```
public class Point {
    public Point() {
    }

    public Point(int xValue, int yValue) {
        x = xValue;
        y = yValue;
    }
    // return x from coordinate pair
    public int getX() {
        return x;
    }
    // return y from coordinate pair
    public int getY() {
        return y;
    }

public void setXY(int x, int y) {
        this.x = x;
        this.y = y;
    }

//public String toString() {
//        return "[" + getX() + ", " + getY() + "]";
//    }
    private int x;  // x part of coordinate pair
    private int y;  // y part of coordinate pair
}
```

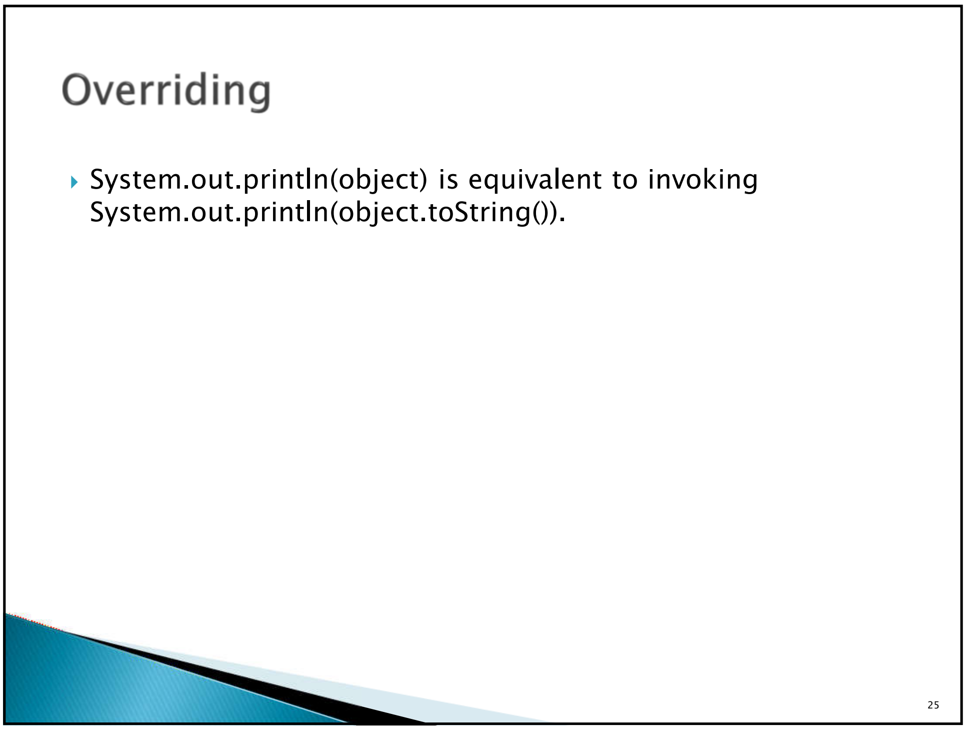
23



Overriding

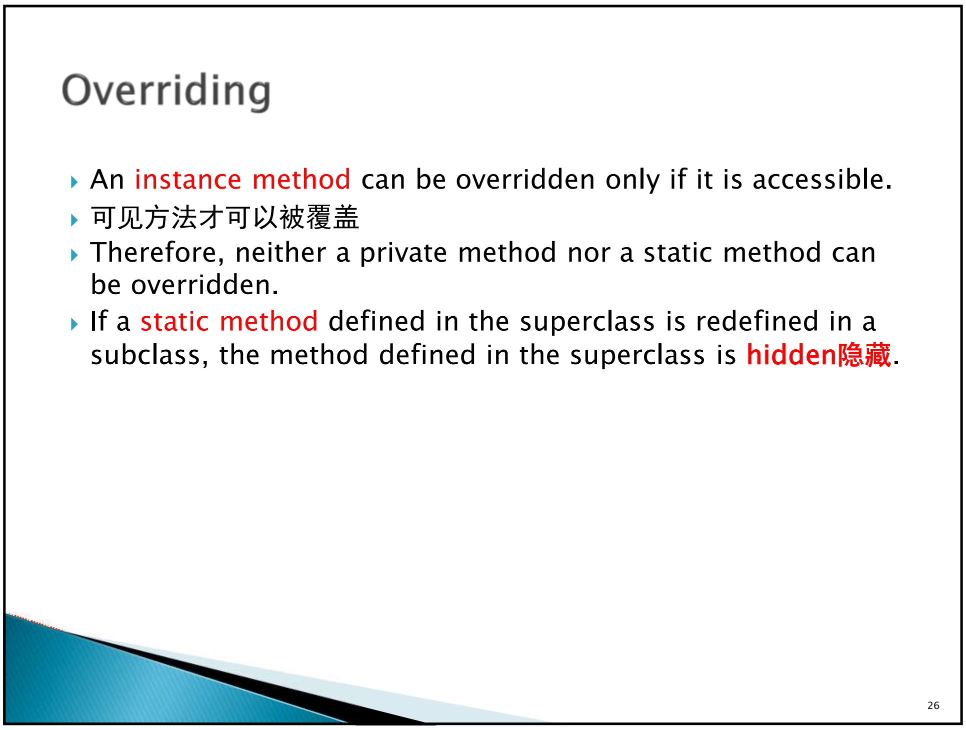
- toString() is an instance method provided by java.lang.Object. It returns a string representation of the object. When the following fragment of code is run,
- Point aPoint = new Point(2,2);
- System.out.println(aPoint);
- the output is [2, 2].
- If there is no method provided in Point overriding toString(), the default output is:
- Point@9cab16 class name+hash code

24



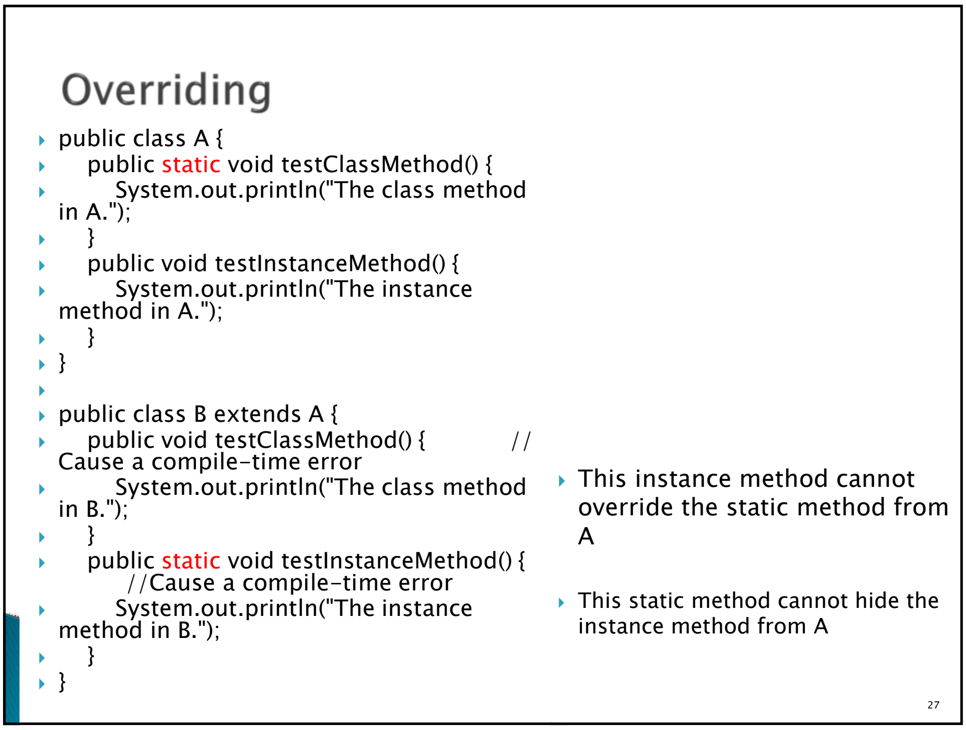
Overriding

- System.out.println(object) is equivalent to invoking System.out.println(object.toString()).



Overriding

- An instance method can be overridden only if it is accessible.
- 可见方法才可以被覆盖
- Therefore, neither a private method nor a static method can be overridden.
- If a static method defined in the superclass is redefined in a subclass, the method defined in the superclass is **hidden隐藏**.

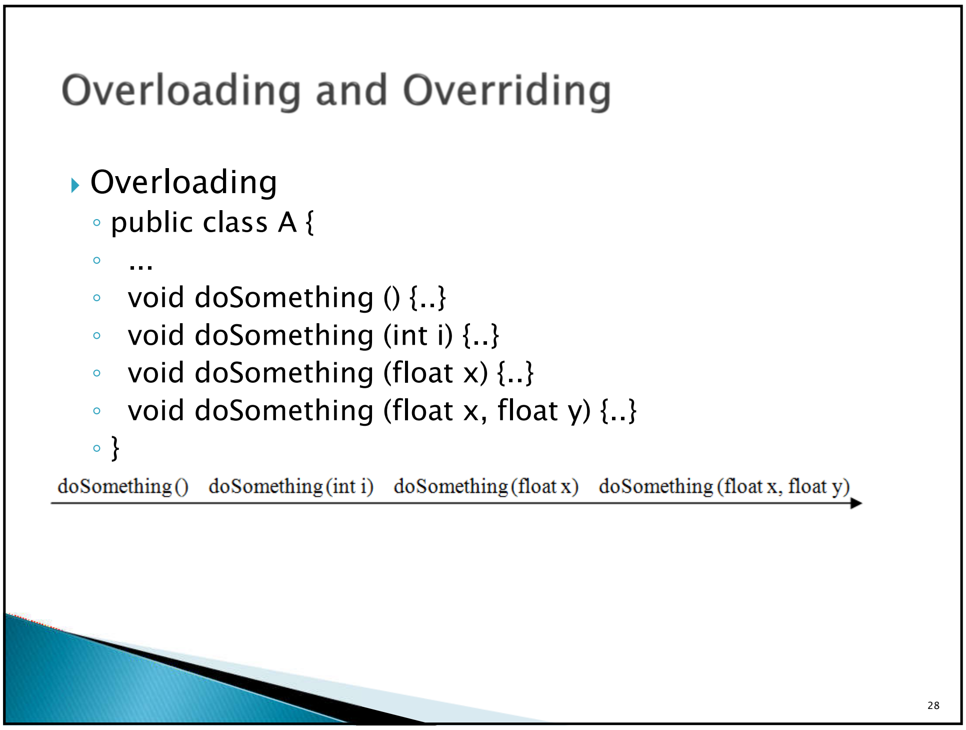


Overriding

```
public class A {
    public static void testClassMethod() {
        System.out.println("The class method in A.");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in A.");
    }
}

public class B extends A {
    public void testClassMethod() {             // Cause a compile-time error
        System.out.println("The class method in B.");
    }
    public static void testInstanceMethod() {
        //Cause a compile-time error
        System.out.println("The instance method in B.");
    }
}
```

- This instance method cannot override the static method from A
- This static method cannot hide the instance method from A

27

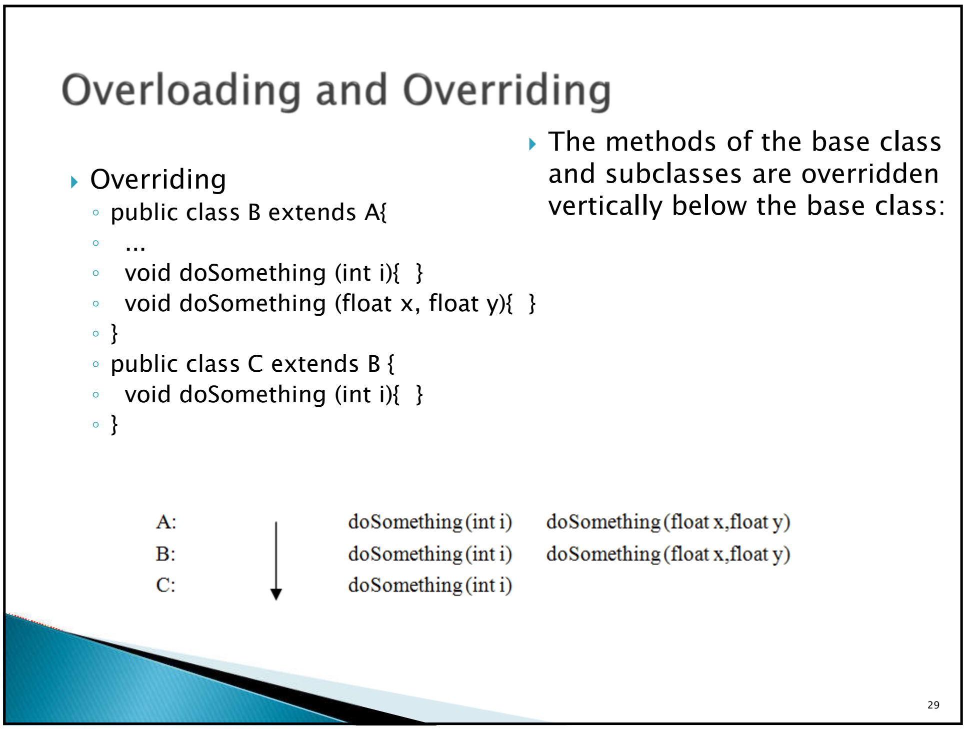


Overloading and Overriding

- Overloading
  - public class A {
  - ...
  - void doSomething () {..}
  - void doSomething (int i) {..}
  - void doSomething (float x) {..}
  - void doSomething (float x, float y) {..}
  - }

doSomething() doSomething(int i) doSomething(float x) doSomething(float x, float y)

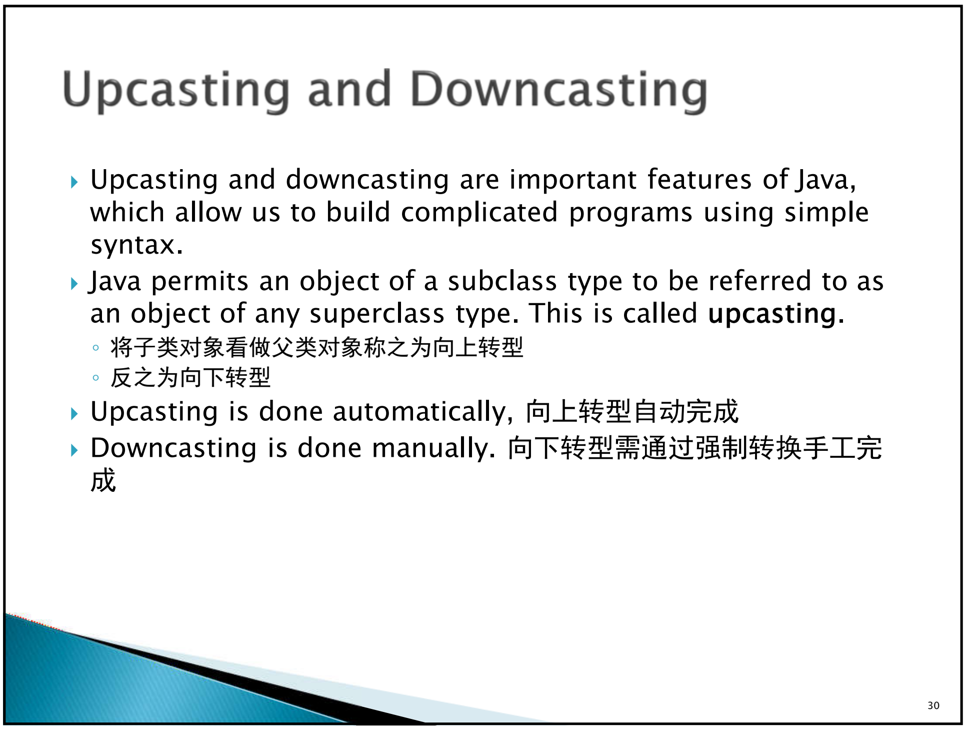
28



Overloading and Overriding

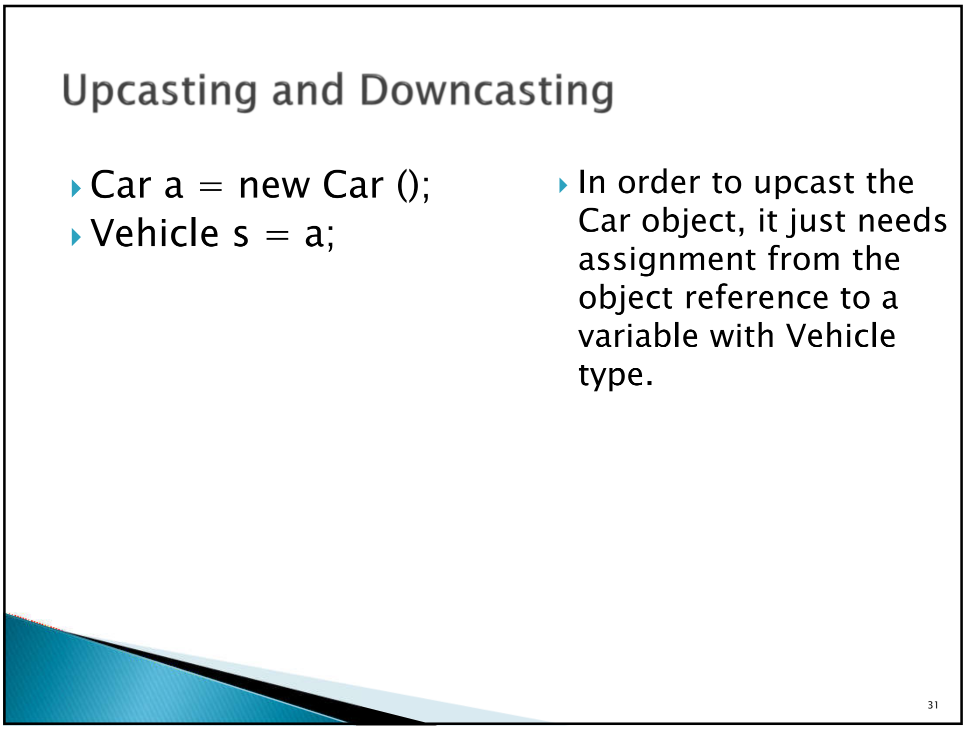
- Overriding
  - public class B extends A{
  - …
  - void doSomething (int i){ }
  - void doSomething (float x, float y){ }
  - }
  - public class C extends B {
  - void doSomething (int i){ }
  - }
- The methods of the base class and subclasses are overridden vertically below the base class:

A:	doSomething(int i)	doSomething(float x,float y)
B:	doSomething(int i)	doSomething(float x,float y)
C:	doSomething(int i)	



Upcasting and Downcasting

- Upcasting and downcasting are important features of Java, which allow us to build complicated programs using simple syntax.
- Java permits an object of a subclass type to be referred to as an object of any superclass type. This is called **upcasting**.
  - 将子类对象看做父类对象称之为向上转型
  - 反之为向下转型
- Upcasting is done automatically, 向上转型自动完成
- Downcasting is done manually. 向下转型需通过强制转换手工完成

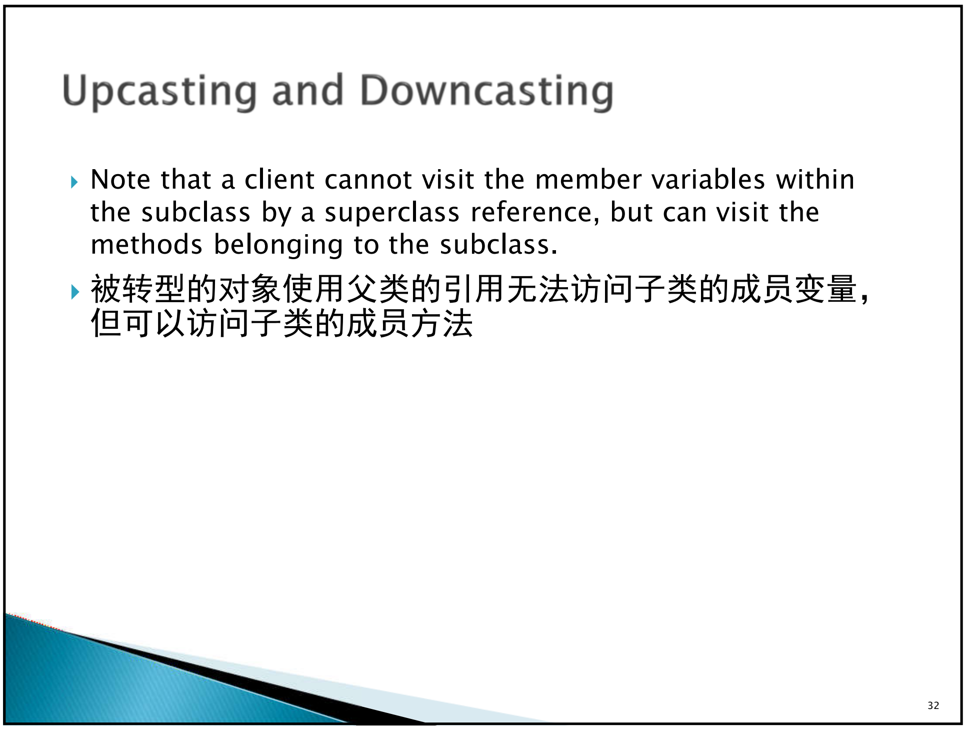


Upcasting and Downcasting

- Car a = new Car ();
- Vehicle s = a;

- In order to upcast the Car object, it just needs assignment from the object reference to a variable with Vehicle type.

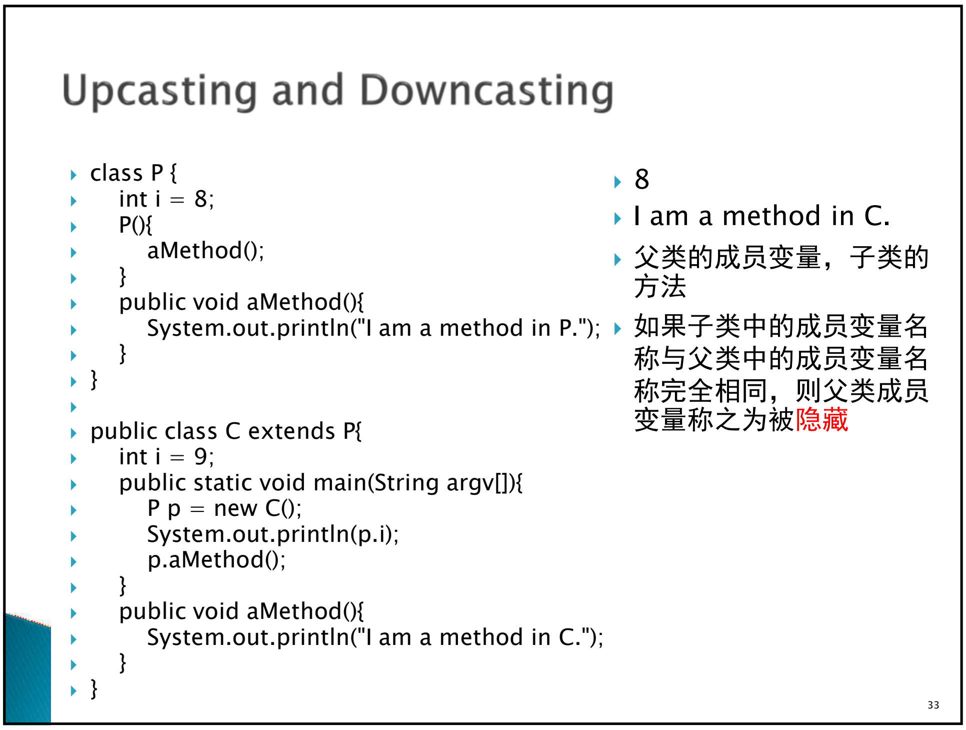
31



Upcasting and Downcasting

- Note that a client cannot visit the member variables within the subclass by a superclass reference, but can visit the methods belonging to the subclass.
- 被转型的对象使用父类的引用无法访问子类的成员变量，但可以访问子类的成员方法

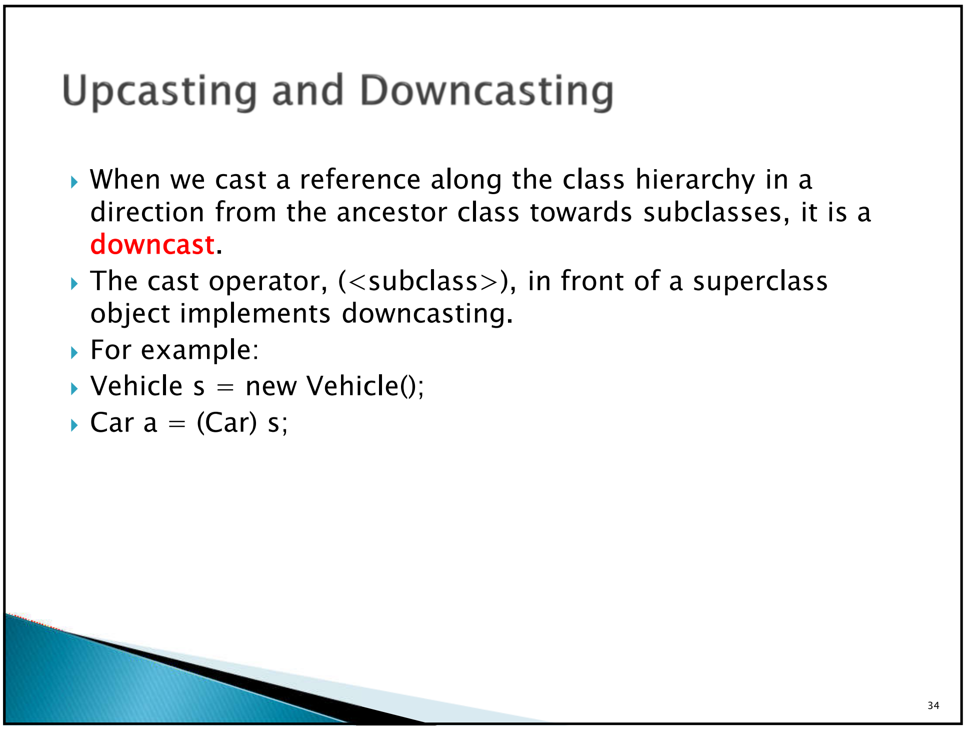
32



Upcasting and Downcasting

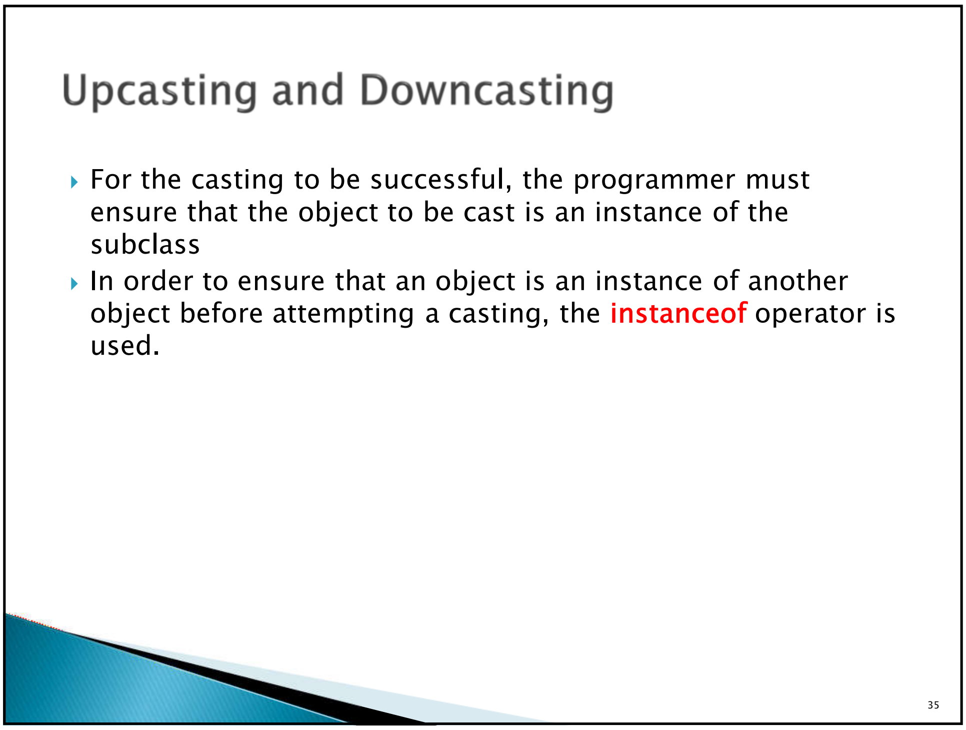
```
class P {
    int i = 8;
    P(){
        aMethod();
    }
    public void aMethod(){
        System.out.println("I am a method in P.");
    }
}

public class C extends P{
    int i = 9;
    public static void main(String argv[]){
        P p = new C();
        System.out.println(p.i);
        p.aMethod();
    }
    public void aMethod(){
        System.out.println("I am a method in C.");
    }
}
```

- 8
- I am a method in C.
- 父类的成员变量，子类的方法
- 如果子类中的成员变量名称与父类中的成员变量名称完全相同，则父类成员变量称之为被隐藏



Upcasting and Downcasting

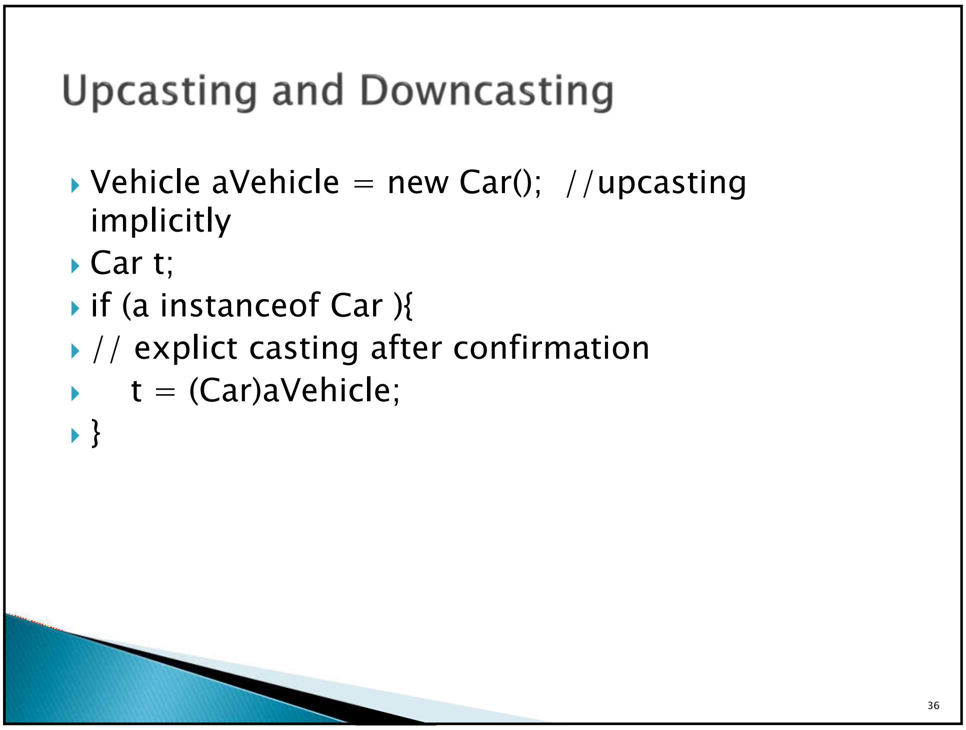
- When we cast a reference along the class hierarchy in a direction from the ancestor class towards subclasses, it is a downcast.
- The cast operator, (<subclass>), in front of a superclass object implements downcasting.
- For example:
- Vehicle s = new Vehicle();
- Car a = (Car) s;



Upcasting and Downcasting

- For the casting to be successful, the programmer must ensure that the object to be cast is an instance of the subclass
- In order to ensure that an object is an instance of another object before attempting a casting, the instanceof operator is used.

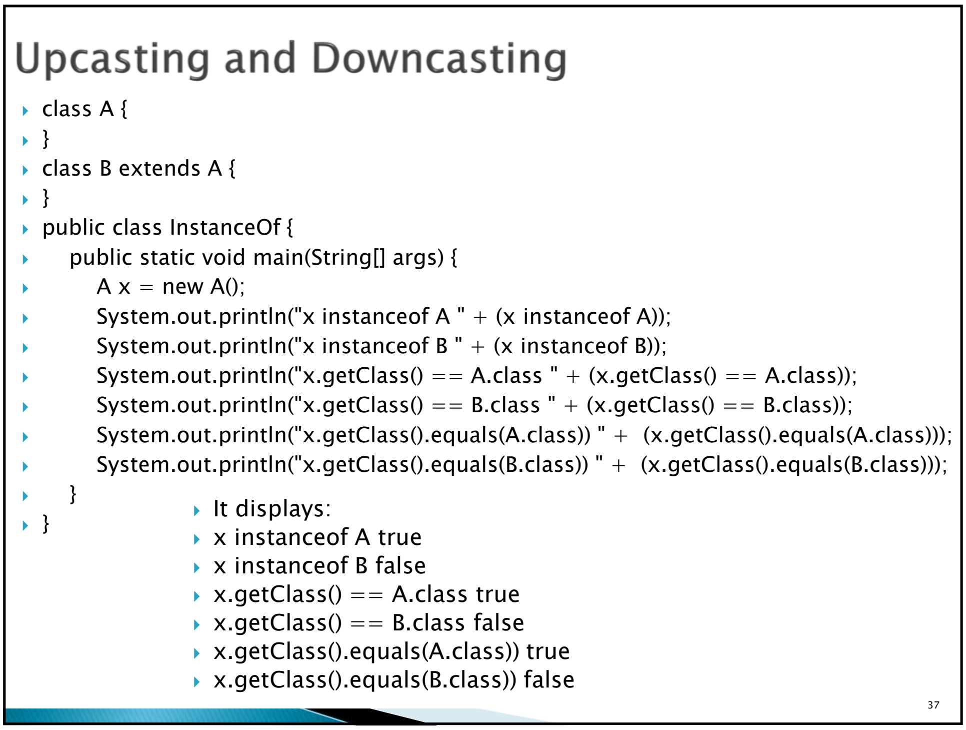
35



Upcasting and Downcasting

- Vehicle aVehicle = new Car(); //upcasting implicitly
- Car t;
- if (a instanceof Car){
- // explict casting after confirmation
-     t = (Car)aVehicle;
- }

36



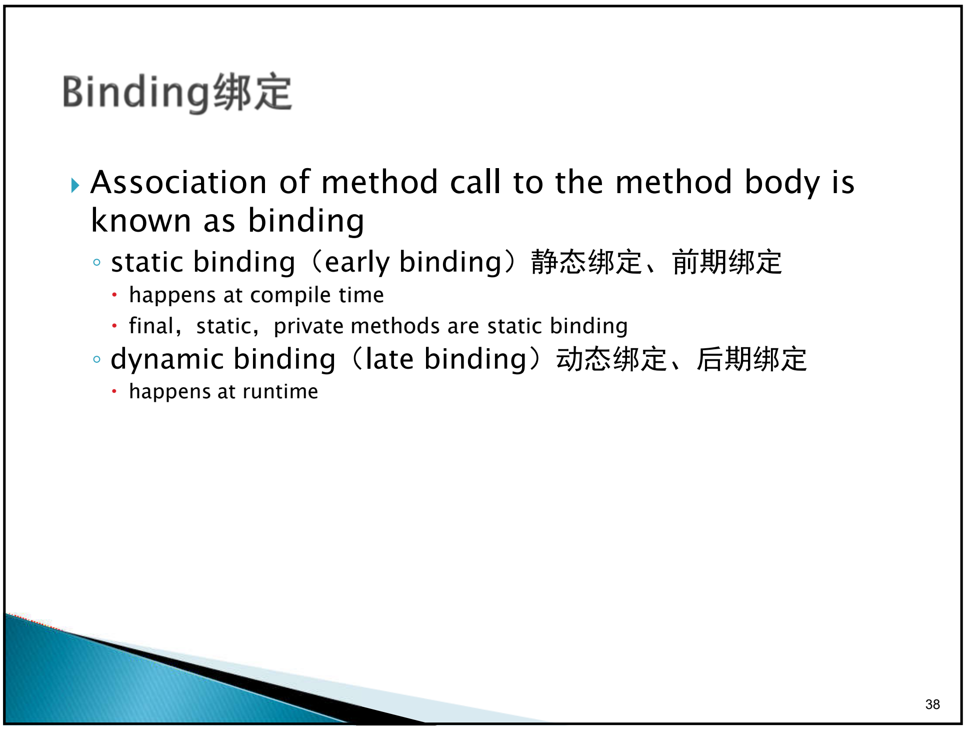
Upcasting and Downcasting

```
class A {
}
class B extends A {
}
public class InstanceOf {
    public static void main(String[] args) {
        A x = new A();
        System.out.println("x instanceof A " + (x instanceof A));
        System.out.println("x instanceof B " + (x instanceof B));
        System.out.println("x.getClass() == A.class " + (x.getClass() == A.class));
        System.out.println("x.getClass() == B.class " + (x.getClass() == B.class));
        System.out.println("x.getClass().equals(A.class)) " +  (x.getClass().equals(A.class)));
        System.out.println("x.getClass().equals(B.class)) " +  (x.getClass().equals(B.class)));
    }
}
```

It displays:

```
x instanceof A true
x instanceof B false
x.getClass() == A.class true
x.getClass() == B.class false
x.getClass().equals(A.class)) true
x.getClass().equals(B.class)) false
```

37

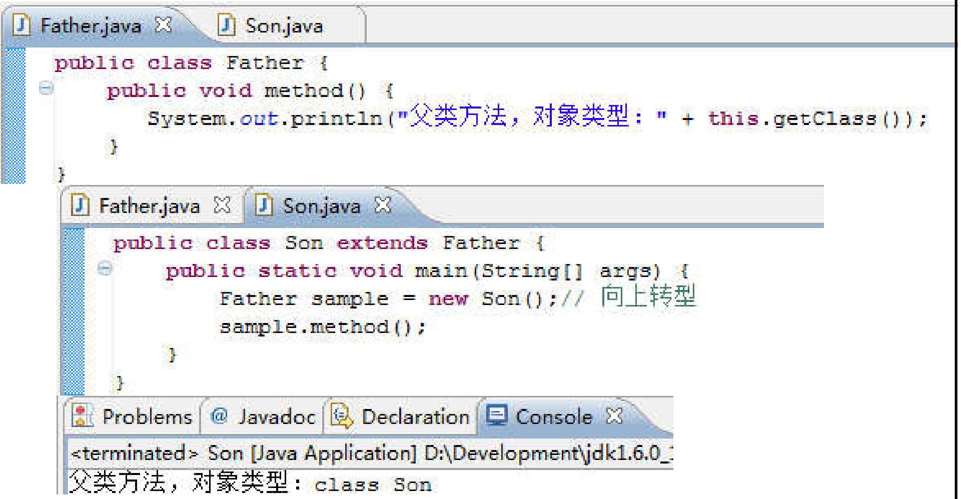


Binding绑定

- Association of method call to the method body is known as binding
  - static binding（early binding）静态绑定、前期绑定
    - happens at compile time
    - final，static，private methods are static binding
  - dynamic binding（late binding）动态绑定、后期绑定
    - happens at runtime

38

Dynamic binding



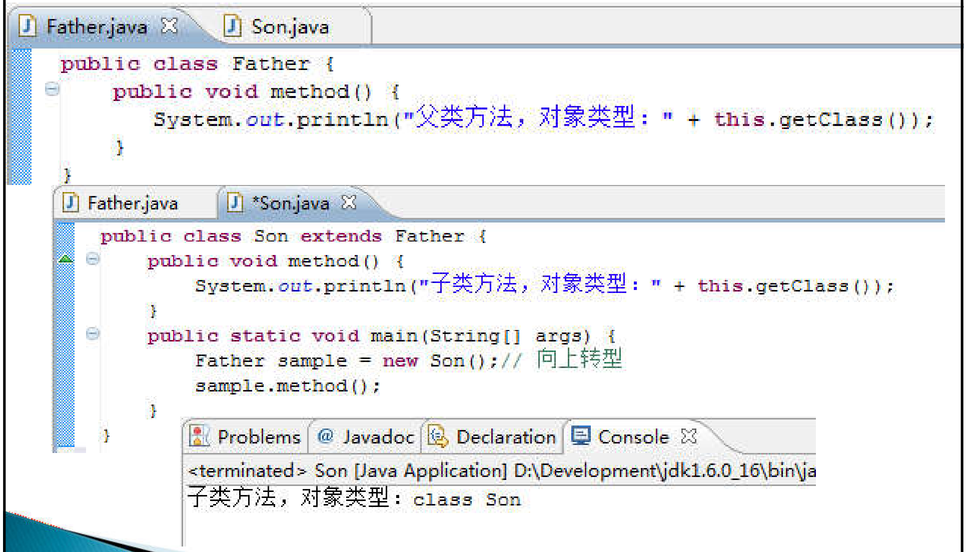
```java
public class Father {
    public void method() {
        System.out.println("父类方法，对象类型：" + this.getClass());
    }
}
```

```java
public class Son extends Father {
    public static void main(String[] args) {
        Father sample = new Son();// 向上转型
        sample.method();
    }
}
```

```
<terminated> Son [Java Application] D:\Development\jdk1.6.0_
父类方法，对象类型：class Son
```

Declared reference of super class, and object of sub-class

Dynamic binding

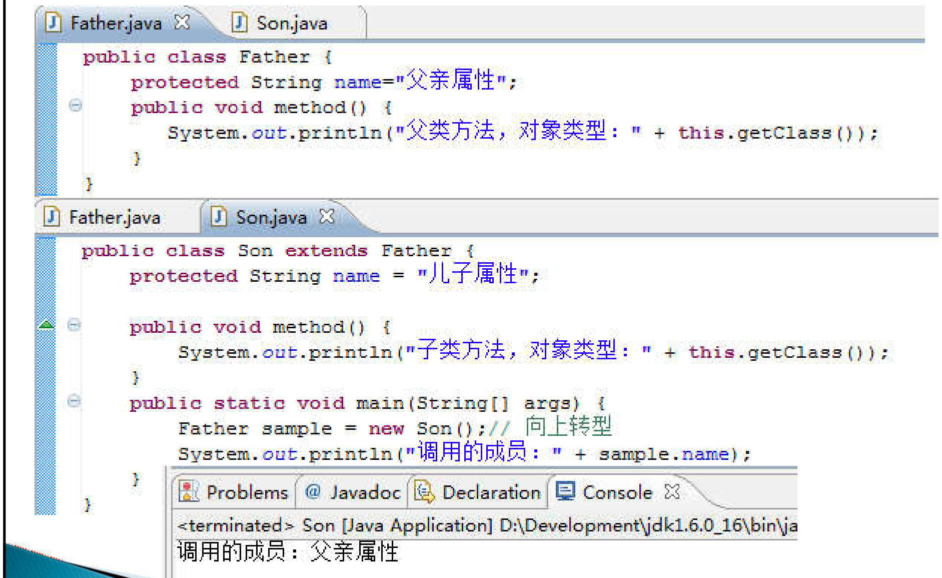


```java
public class Father {
    public void method() {
        System.out.println("父类方法，对象类型：" + this.getClass());
    }
}
```

```java
public class Son extends Father {
    public void method() {
        System.out.println("子类方法，对象类型：" + this.getClass());
    }
    public static void main(String[] args) {
        Father sample = new Son();// 向上转型
        sample.method();
    }
}
```

```
<terminated> Son [Java Application] D:\Development\jdk1.6.0_16\bin\ja
子类方法，对象类型：class Son
```

Dynamic binding

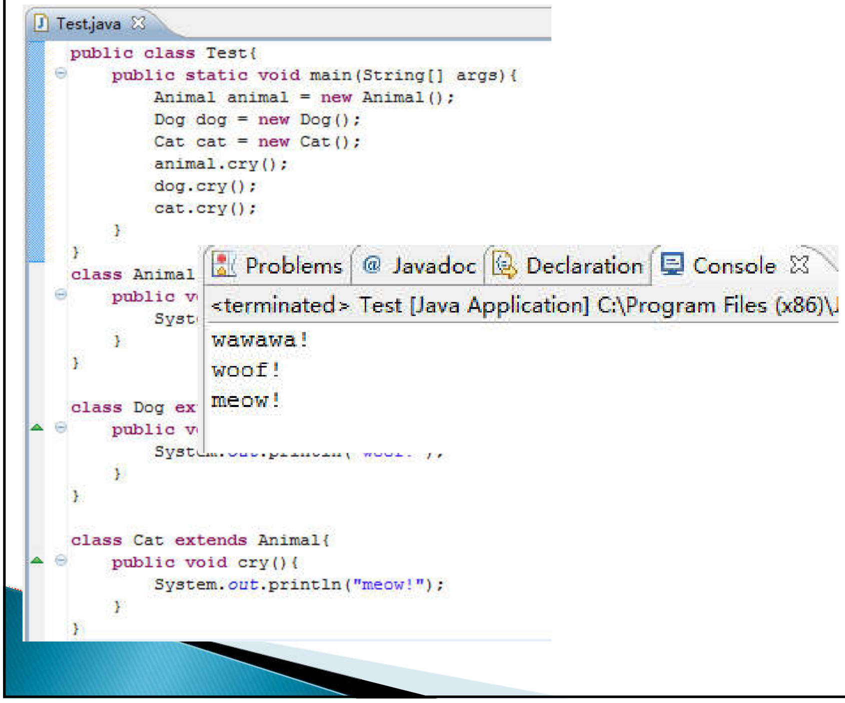


```
public class Father {
    protected String name="父亲属性";
    public void method() {
       System.out.println("父类方法，对象类型：" + this.getClass());
    }
}
```

```
public class Son extends Father {
    protected String name = "儿子属性";

    public void method() {
        System.out.println("子类方法，对象类型：" + this.getClass());
    }
    public static void main(String[] args) {
        Father sample = new Son();// 向上转型
        System.out.println("调用的成员：" + sample.name);
    }
}
```

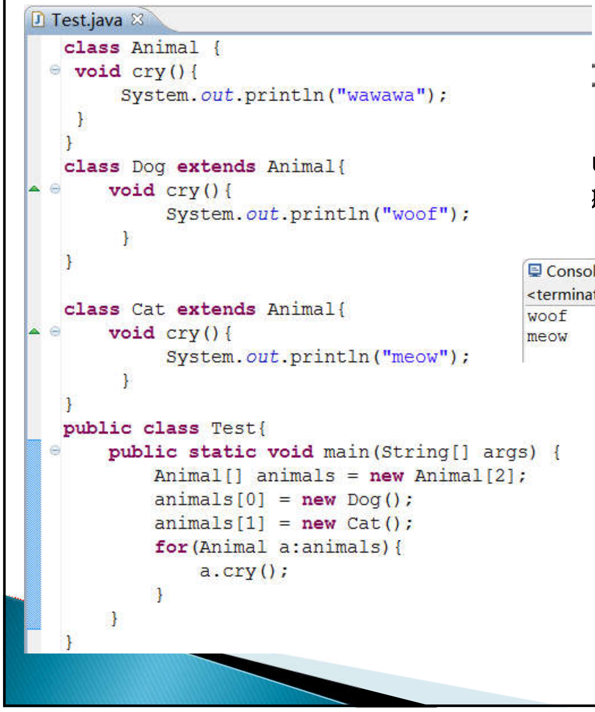
```
<terminated> Son [Java Application] D:\Development\jdk1.6.0_16\bin\ja
调用的成员：父亲属性
```

由父类的引用的子类的对象调用到的是父类的成员变量。动态绑定只针对对象的方法。



```
public class Test{
    public static void main(String[] args){
        Animal animal = new Animal();
        Dog dog = new Dog();
        Cat cat = new Cat();
        animal.cry();
        dog.cry();
        cat.cry();
    }
}
class Animal
    public v
        Syst
    }
}

class Dog ex
    public v
        System.out.println("woof!");
    }
}

class Cat extends Animal{
    public void cry(){
        System.out.println("meow!");
    }
}
```

```
<terminated> Test [Java Application] C:\Program Files (x86)\J
wawawa!
woof!
meow!
```



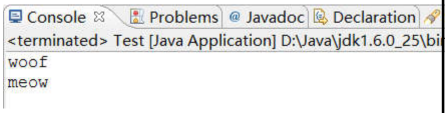
Test.java

```
class Animal {
 void cry(){
     System.out.println("wawawa");
 }
}
class Dog extends Animal{
     void cry(){
         System.out.println("woof");
     }
}

class Cat extends Animal{
     void cry(){
         System.out.println("meow");
     }
}
public class Test{
     public static void main(String[] args) {
         Animal[] animals = new Animal[2];
         animals[0] = new Dog();
         animals[1] = new Cat();
         for(Animal a:animals){
             a.cry();
         }
     }
}
```

ding

ub classes' objects,

g

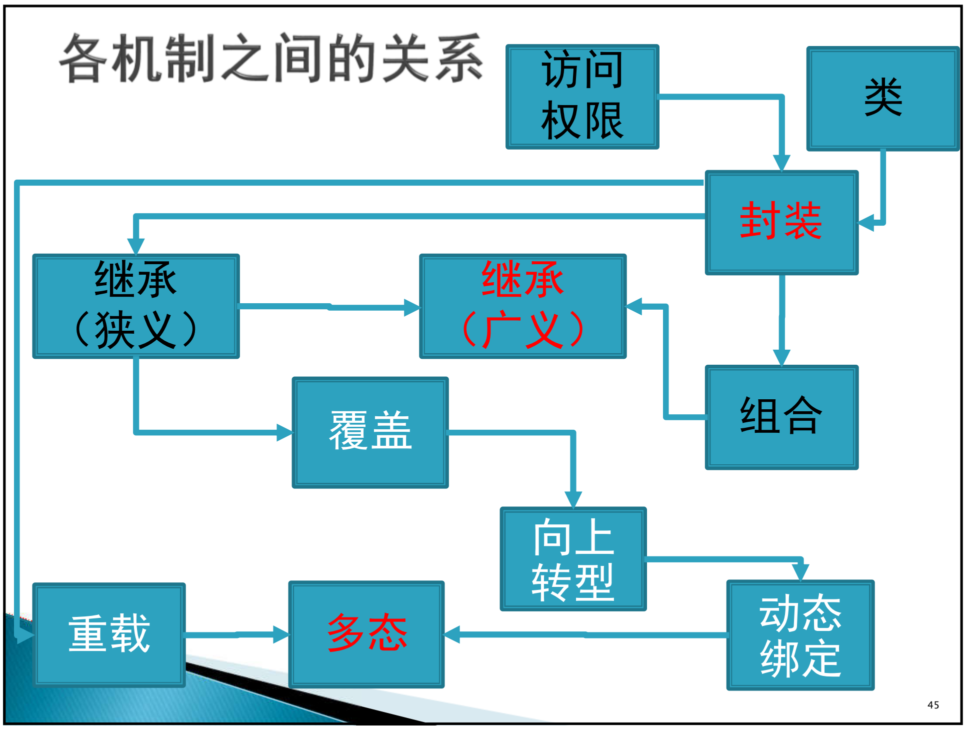


<terminated> Test [Java Application] D:\Java\jdk1.6.0_25\bi

```
woof
meow
```

Polymorphism多态

- Polymorphism in Java
  - Overloading
  - Overriding



各机制之间的关系

访问权限

类

封装

继承（狭义）

继承（广义）

组合

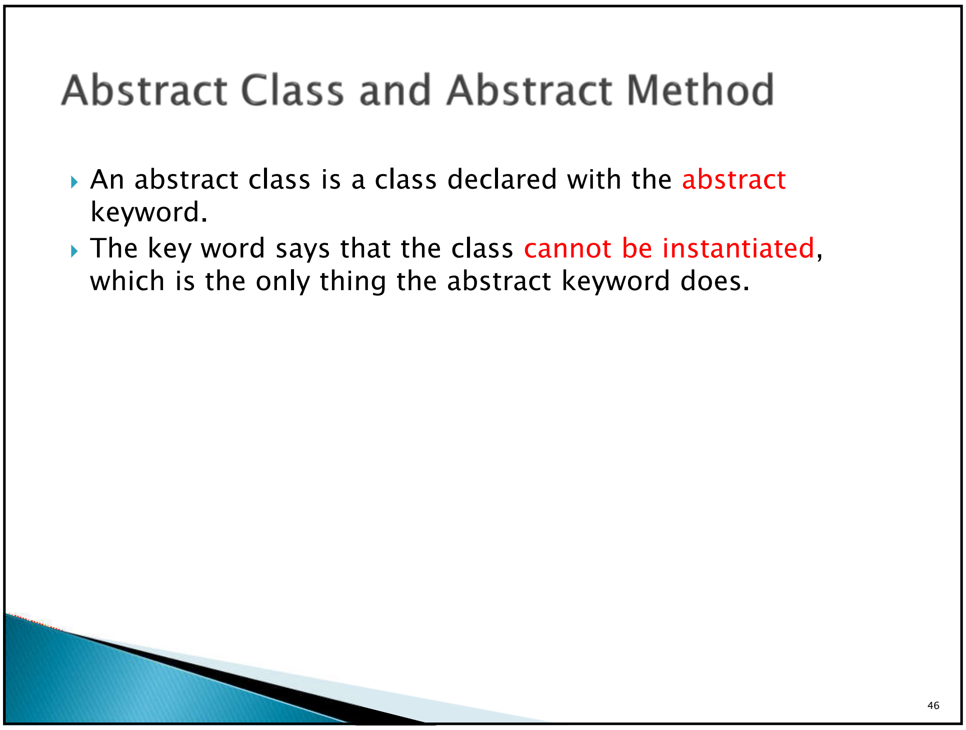
覆盖

向上转型

重载

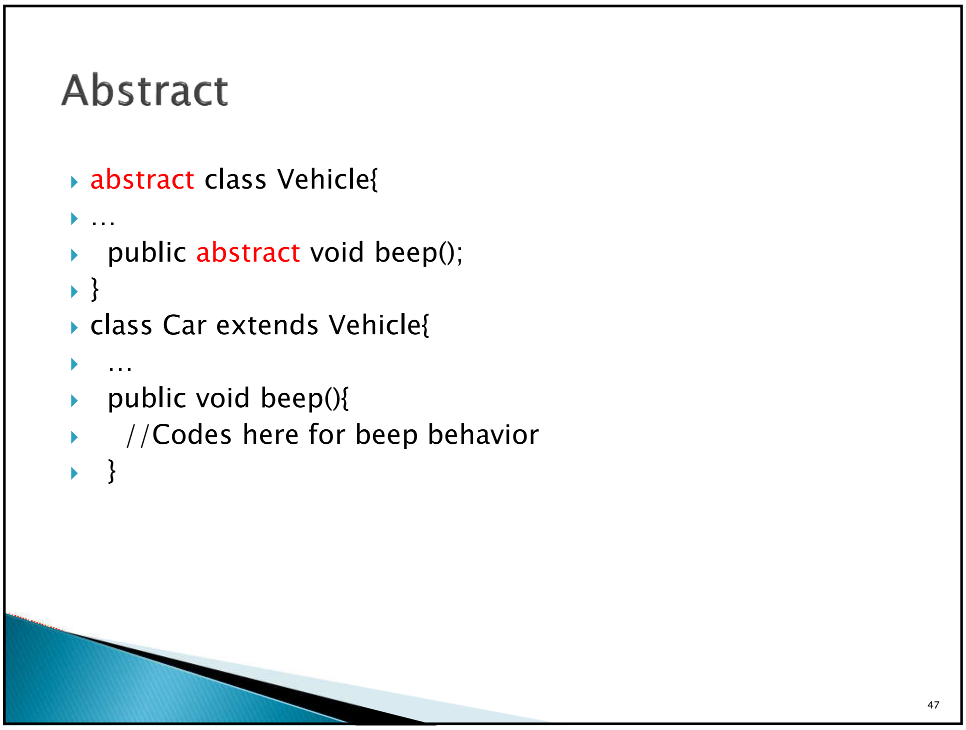
多态

动态绑定



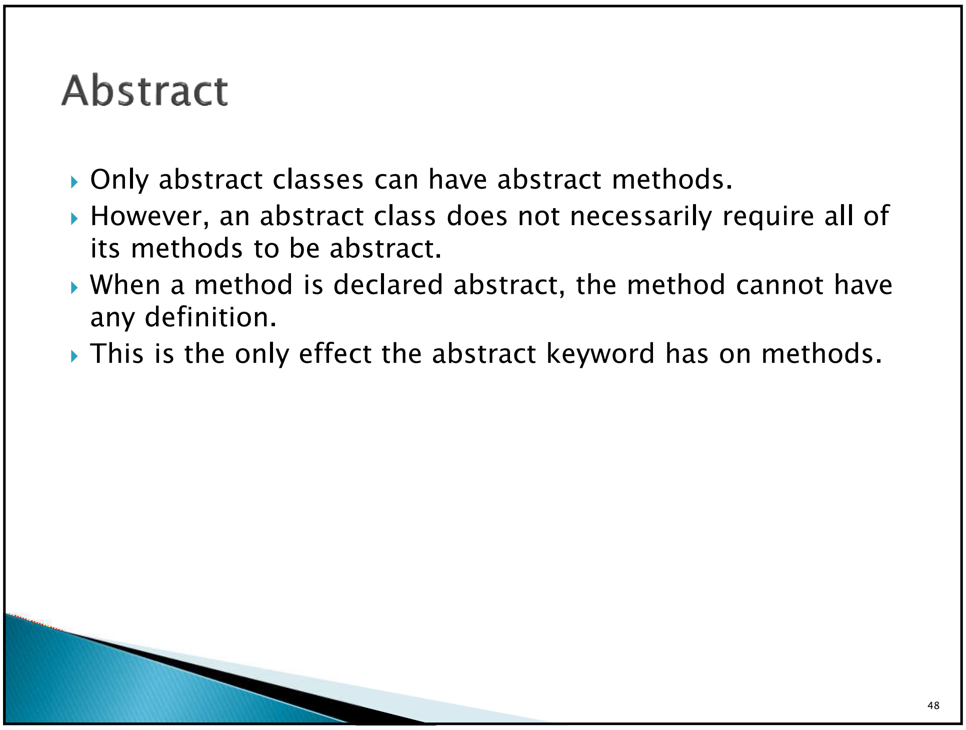
Abstract Class and Abstract Method

- An abstract class is a class declared with the abstract keyword.
- The key word says that the class cannot be instantiated, which is the only thing the abstract keyword does.



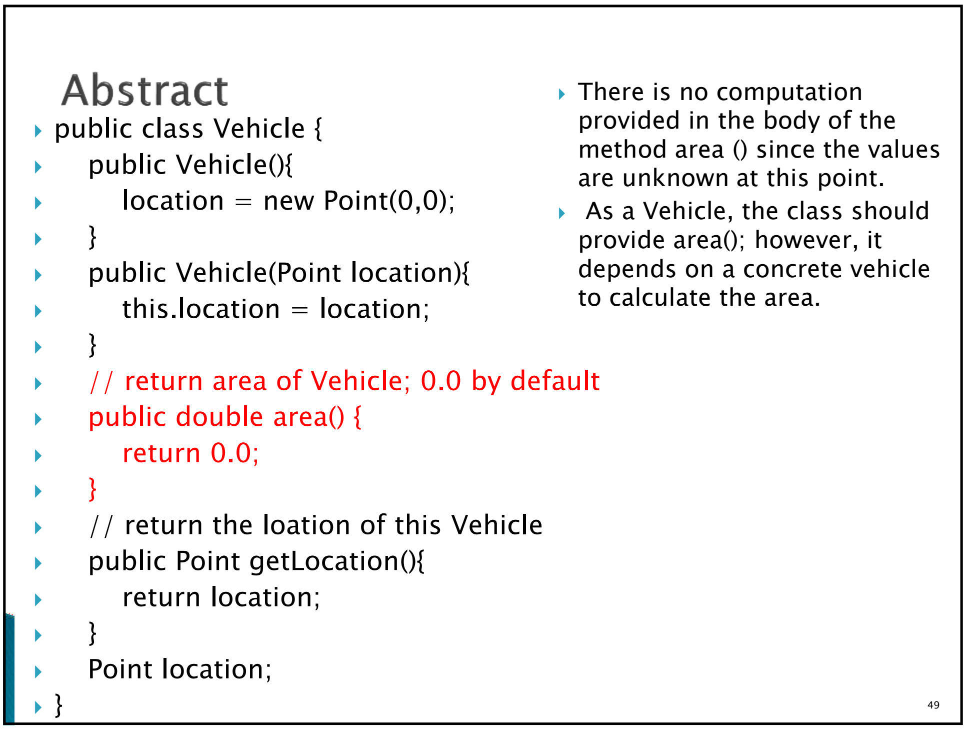
Abstract

```
abstract class Vehicle{
…
  public abstract void beep();
}
class Car extends Vehicle{
  …
  public void beep(){
    //Codes here for beep behavior
  }
```



Abstract

- Only abstract classes can have abstract methods.
- However, an abstract class does not necessarily require all of its methods to be abstract.
- When a method is declared abstract, the method cannot have any definition.
- This is the only effect the abstract keyword has on methods.

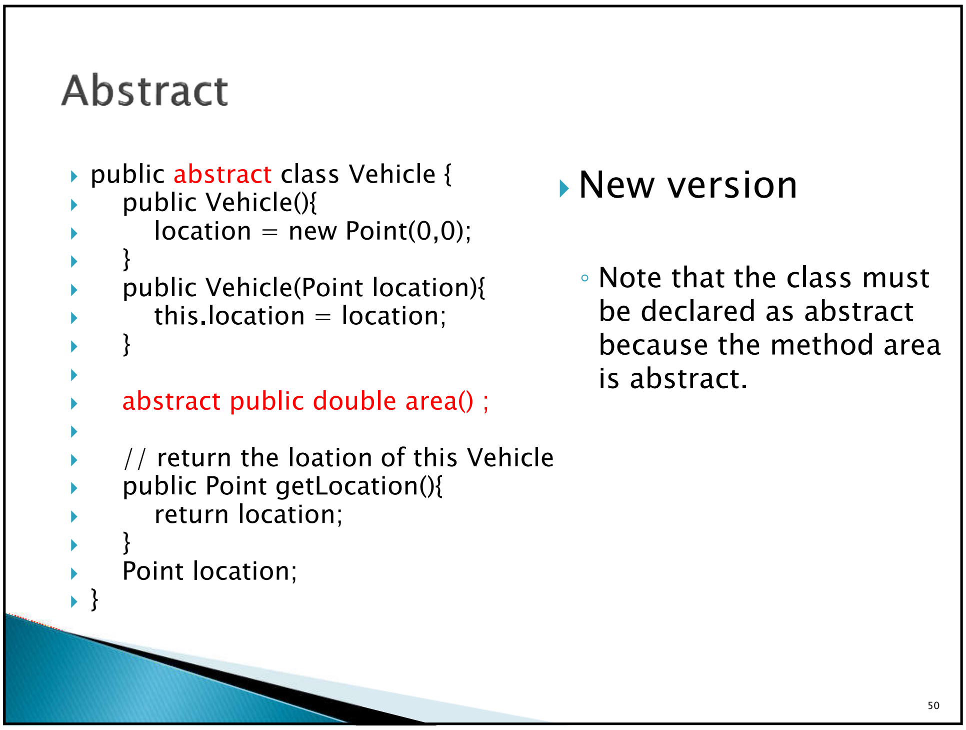


Abstract

```
public class Vehicle {
    public Vehicle(){
        location = new Point(0,0);
    }
    public Vehicle(Point location){
        this.location = location;
    }
    // return area of Vehicle; 0.0 by default
    public double area() {
        return 0.0;
    }
    // return the loation of this Vehicle
    public Point getLocation(){
        return location;
    }
    Point location;
}
```

- There is no computation provided in the body of the method area () since the values are unknown at this point.
- As a Vehicle, the class should provide area(); however, it depends on a concrete vehicle to calculate the area.

49

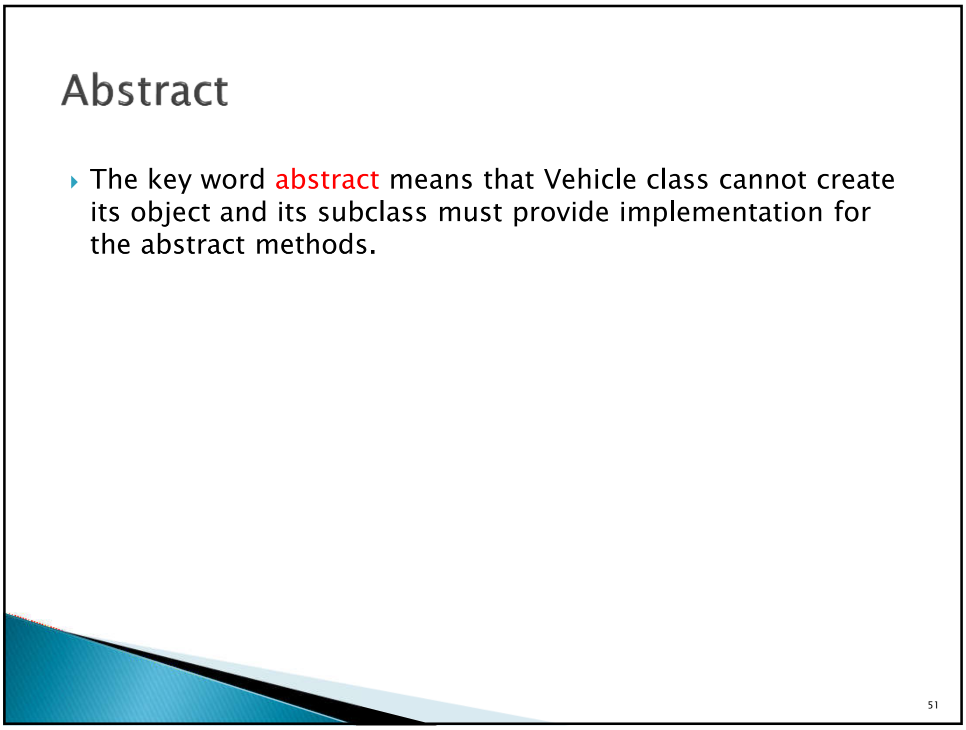


Abstract

```
public abstract class Vehicle {
    public Vehicle(){
        location = new Point(0,0);
    }
    public Vehicle(Point location){
        this.location = location;
    }

    abstract public double area() ;

    // return the loation of this Vehicle
    public Point getLocation(){
        return location;
    }
    Point location;
}
```

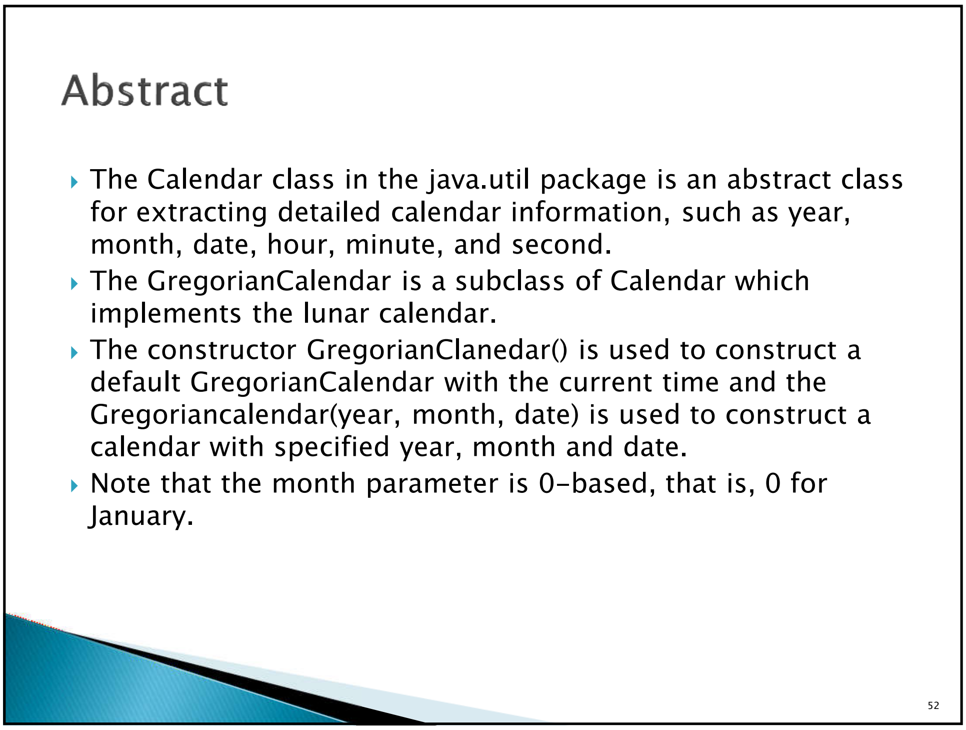
- New version
  - Note that the class must be declared as abstract because the method area is abstract.

50



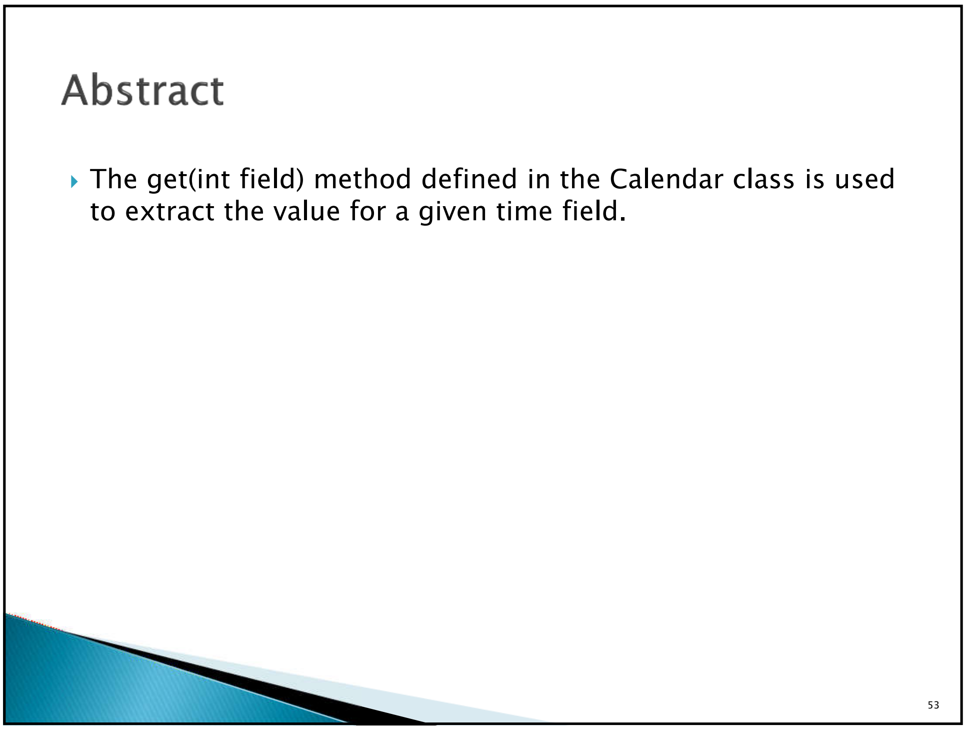
Abstract

- The key word abstract means that Vehicle class cannot create its object and its subclass must provide implementation for the abstract methods.



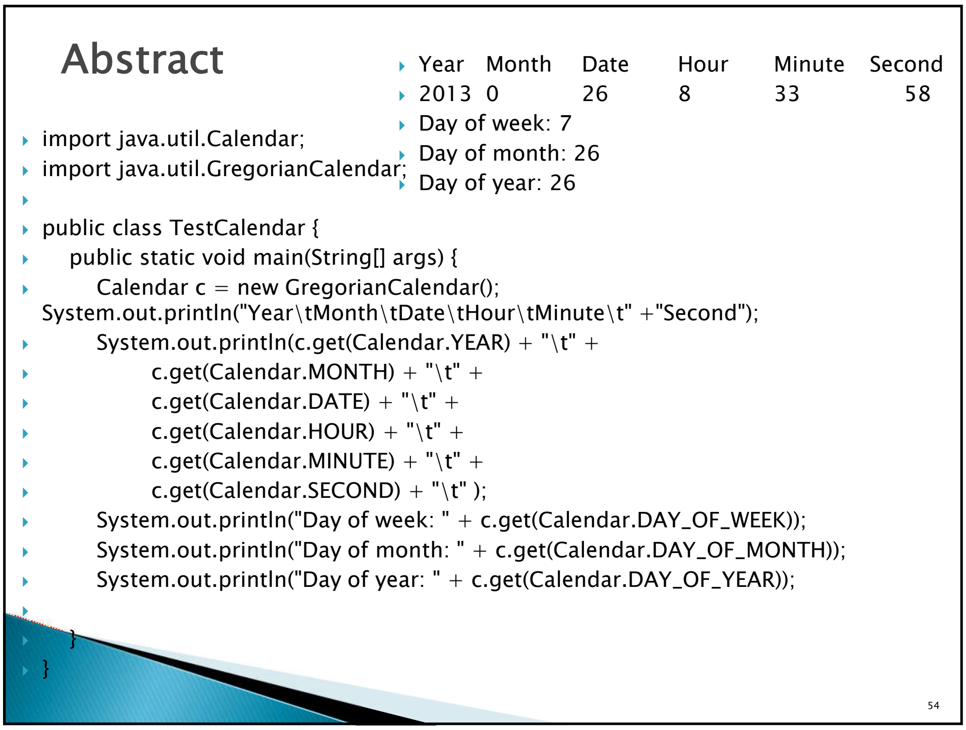
Abstract

- The Calendar class in the java.util package is an abstract class for extracting detailed calendar information, such as year, month, date, hour, minute, and second.
- The GregorianCalendar is a subclass of Calendar which implements the lunar calendar.
- The constructor GregorianClanedar() is used to construct a default GregorianCalendar with the current time and the Gregoriancalendar(year, month, date) is used to construct a calendar with specified year, month and date.
- Note that the month parameter is 0-based, that is, 0 for January.



Abstract

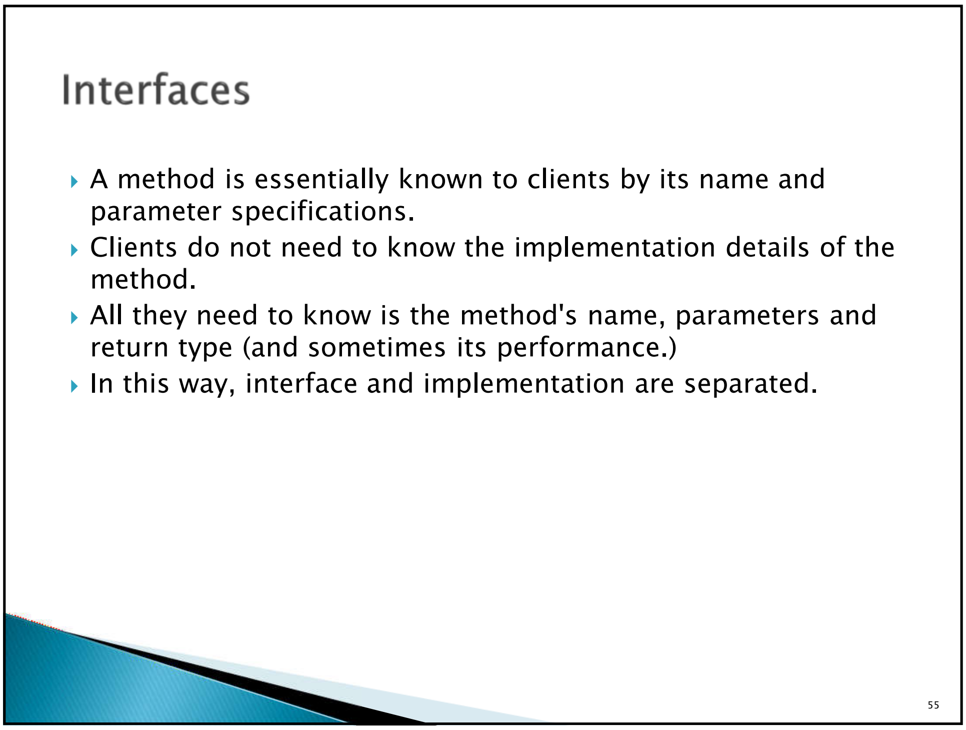
- The get(int field) method defined in the Calendar class is used to extract the value for a given time field.



Abstract

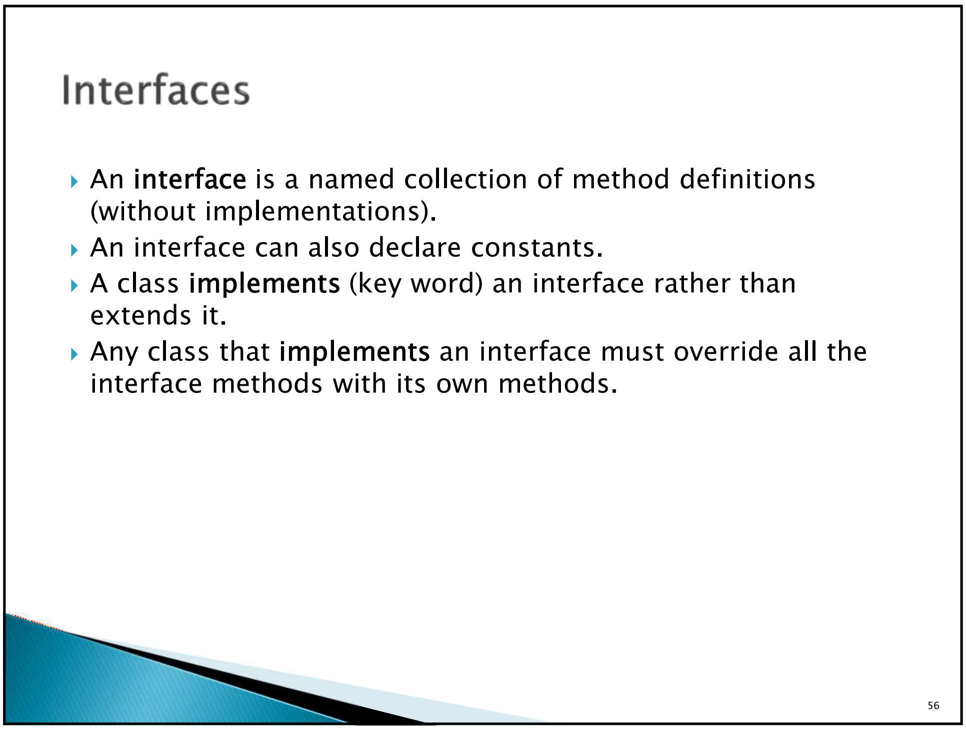
```
Year   Month   Date   Hour   Minute   Second
2013   0       26     8      33       58
Day of week: 7
Day of month: 26
Day of year: 26
```

```
import java.util.Calendar;
import java.util.GregorianCalendar;

public class TestCalendar {
  public static void main(String[] args) {
    Calendar c = new GregorianCalendar();
System.out.println("Year\tMonth\tDate\tHour\tMinute\t" +"Second");
    System.out.println(c.get(Calendar.YEAR) + "\t" +
        c.get(Calendar.MONTH) + "\t" +
        c.get(Calendar.DATE) + "\t" +
        c.get(Calendar.HOUR) + "\t" +
        c.get(Calendar.MINUTE) + "\t" +
        c.get(Calendar.SECOND) + "\t" );
    System.out.println("Day of week: " + c.get(Calendar.DAY_OF_WEEK));
    System.out.println("Day of month: " + c.get(Calendar.DAY_OF_MONTH));
    System.out.println("Day of year: " + c.get(Calendar.DAY_OF_YEAR));

  }
}
```



Interfaces

- A method is essentially known to clients by its name and parameter specifications.
- Clients do not need to know the implementation details of the method.
- All they need to know is the method's name, parameters and return type (and sometimes its performance.)
- In this way, interface and implementation are separated.



Interfaces

- An **interface** is a named collection of method definitions (without implementations).
- An interface can also declare constants.
- A class **implements** (key word) an interface rather than extends it.
- Any class that **implements** an interface must override all the interface methods with its own methods.

Interfaces

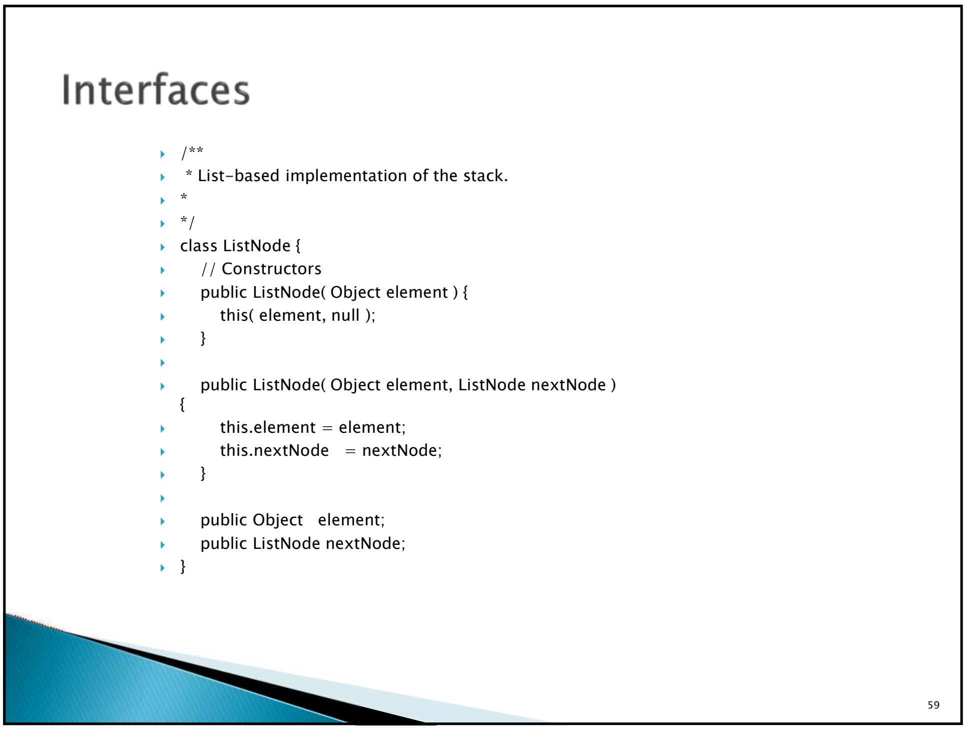
- By default, all the methods in an interface are public and abstract.
- Fields in the interface are public, final, and static by default. 所有字段都是静态常量
- Note that interface methods are public, so the overriding methods in the classes that implement the interface must also be public.所有方法都是共有抽象

57

Interfaces

```
public interface Stack {
          /**
           * Insert a new item into the stack.
           * @param e  the item to insert.
           * @return false if the stack is full, true otherwise.
           */
          boolean push(Object e);

          /**
           * Remove the most recently inserted item from the stack.
           * @return false if the stack is empty, true otherwise.
           */
          boolean pop();

          /**
           * Get the most recently inserted item in the stack. The stack keep unchanged.
           * @return the most recently inserted item on the top of the stack, null in case of empty stack.
           */
          Object top();

          /**
           * Test if the stack is logically empty.
           * @return true if empty, false otherwise.
           */
          boolean isEmpty();

          /**
           * Make the stack logically empty.
           */
          void clear();
}
```

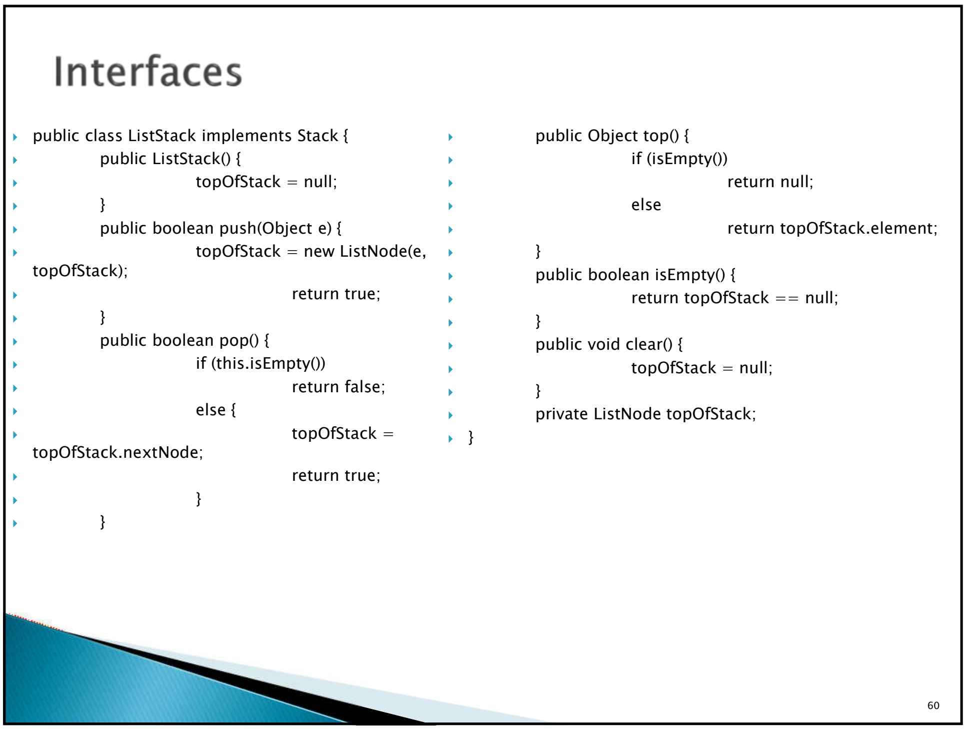
58



Interfaces

```
/**
 * List-based implementation of the stack.
 *
 */
class ListNode {
    // Constructors
    public ListNode( Object element ) {
        this( element, null );
    }

    public ListNode( Object element, ListNode nextNode ) {
        this.element = element;
        this.nextNode   = nextNode;
    }

    public Object   element;
    public ListNode nextNode;
}
```

59

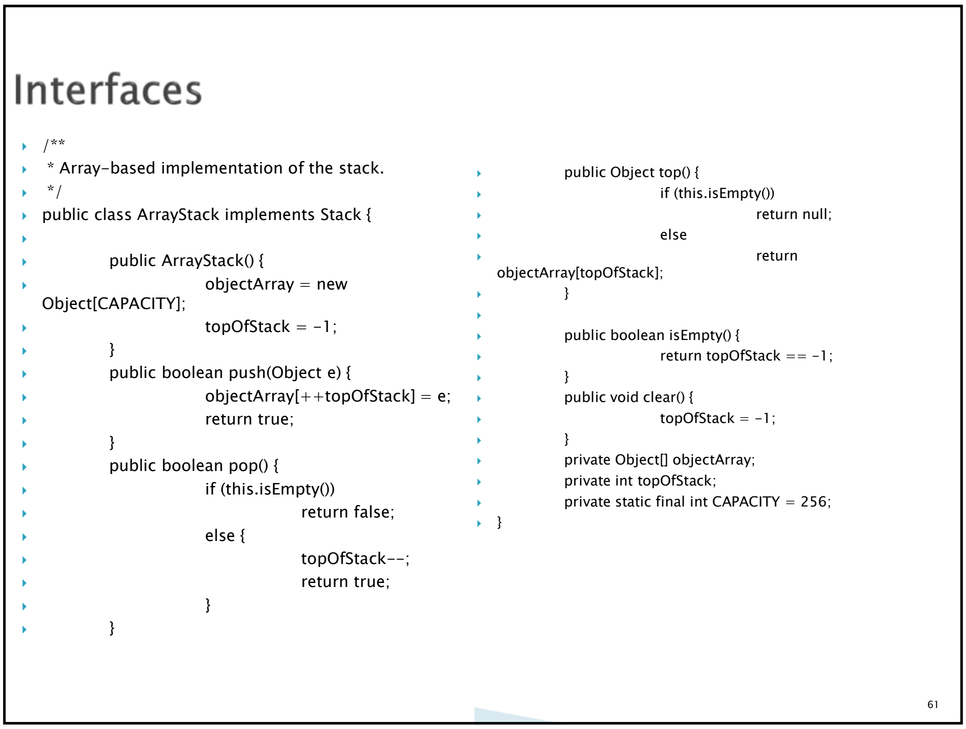


Interfaces

```
public class ListStack implements Stack {
        public ListStack() {
                topOfStack = null;
        }
        public boolean push(Object e) {
                topOfStack = new ListNode(e, topOfStack);
                        return true;
        }
        public boolean pop() {
                if (this.isEmpty())
                        return false;
                else {
                        topOfStack = topOfStack.nextNode;
                        return true;
                }
        }
```

```
        public Object top() {
                if (isEmpty())
                        return null;
                else
                        return topOfStack.element;
        }
        public boolean isEmpty() {
                return topOfStack == null;
        }
        public void clear() {
                topOfStack = null;
        }
        private ListNode topOfStack;
}
```

60

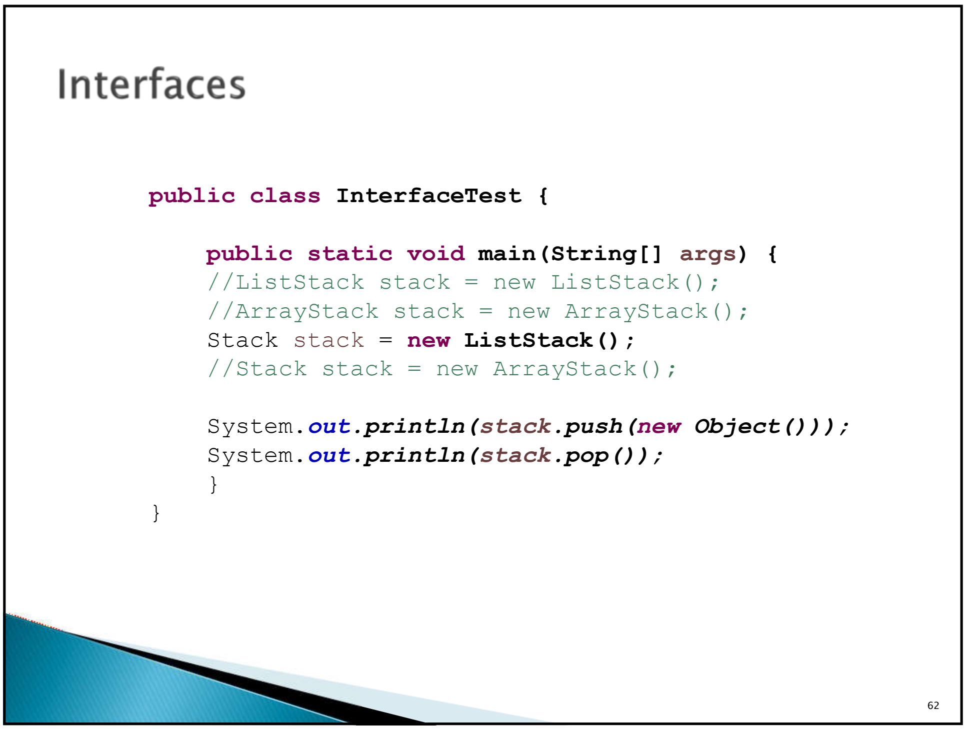


Interfaces

```
/**
 * Array-based implementation of the stack.
 */
public class ArrayStack implements Stack {

        public ArrayStack() {
                objectArray = new
Object[CAPACITY];
                topOfStack = -1;
        }
        public boolean push(Object e) {
                objectArray[++topOfStack] = e;
                return true;
        }
        public boolean pop() {
                if (this.isEmpty())
                        return false;
                else {
                        topOfStack--;
                        return true;
                }
        }
```

```
        public Object top() {
                if (this.isEmpty())
                        return null;
                else
                        return
objectArray[topOfStack];
        }


        public boolean isEmpty() {
                return topOfStack == -1;
        }
        public void clear() {
                topOfStack = -1;
        }
        private Object[] objectArray;
        private int topOfStack;
        private static final int CAPACITY = 256;
}
```

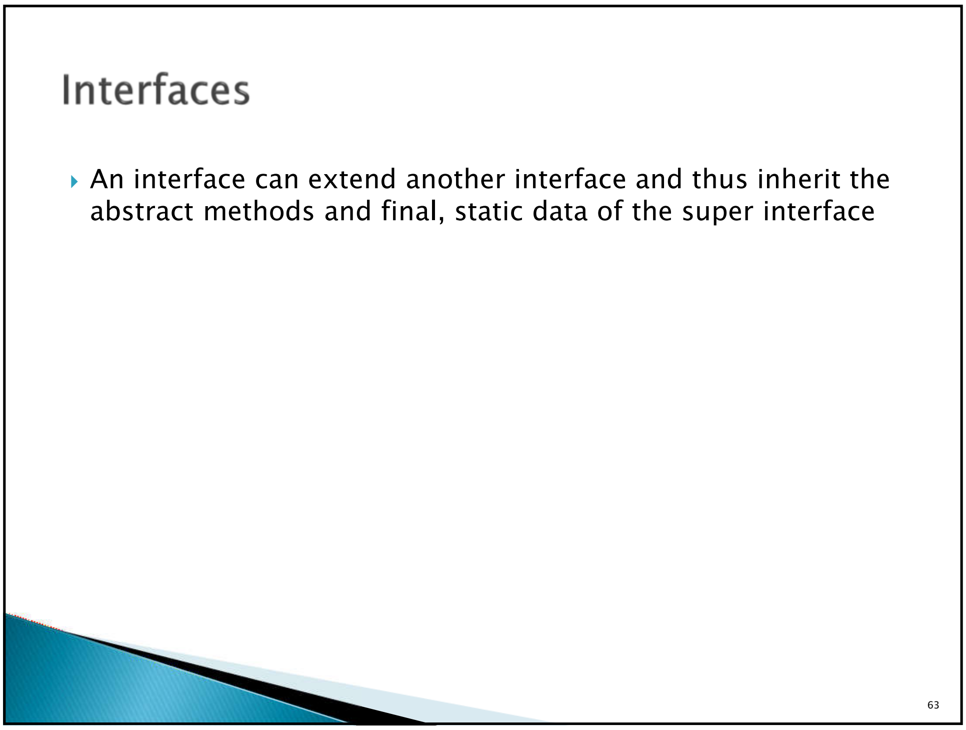
61



Interfaces

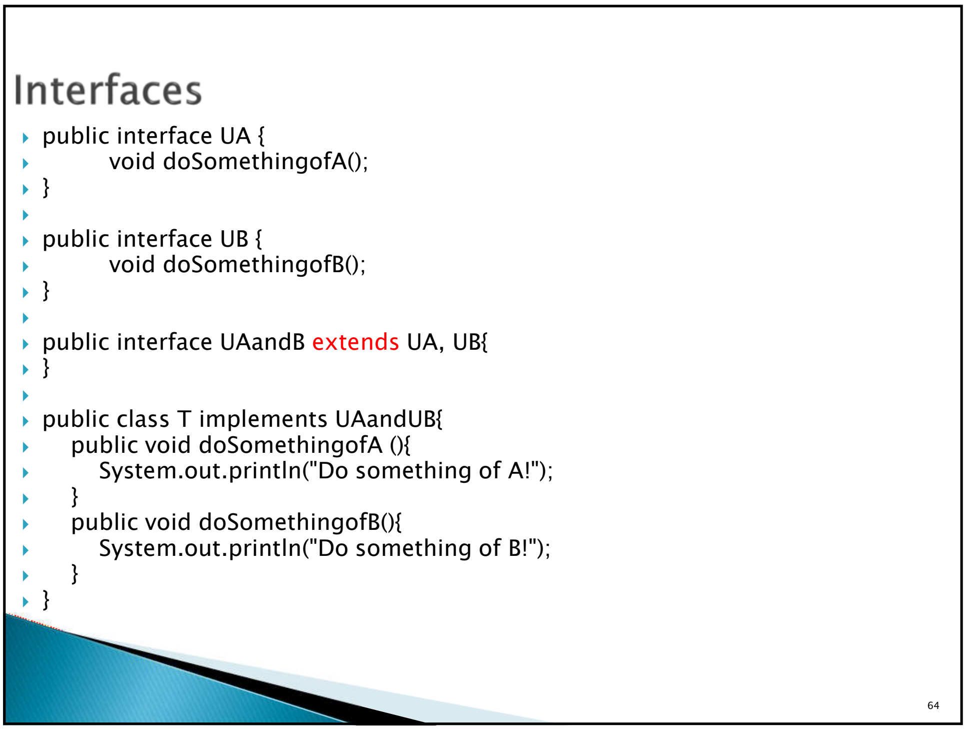
```
public class InterfaceTest {

    public static void main(String[] args) {
    //ListStack stack = new ListStack();
    //ArrayStack stack = new ArrayStack();
    Stack stack = new ListStack();
    //Stack stack = new ArrayStack();

    System.out.println(stack.push(new Object()));
    System.out.println(stack.pop());
    }
}
```

62



Interfaces

- An interface can extend another interface and thus inherit the abstract methods and final, static data of the super interface



Interfaces

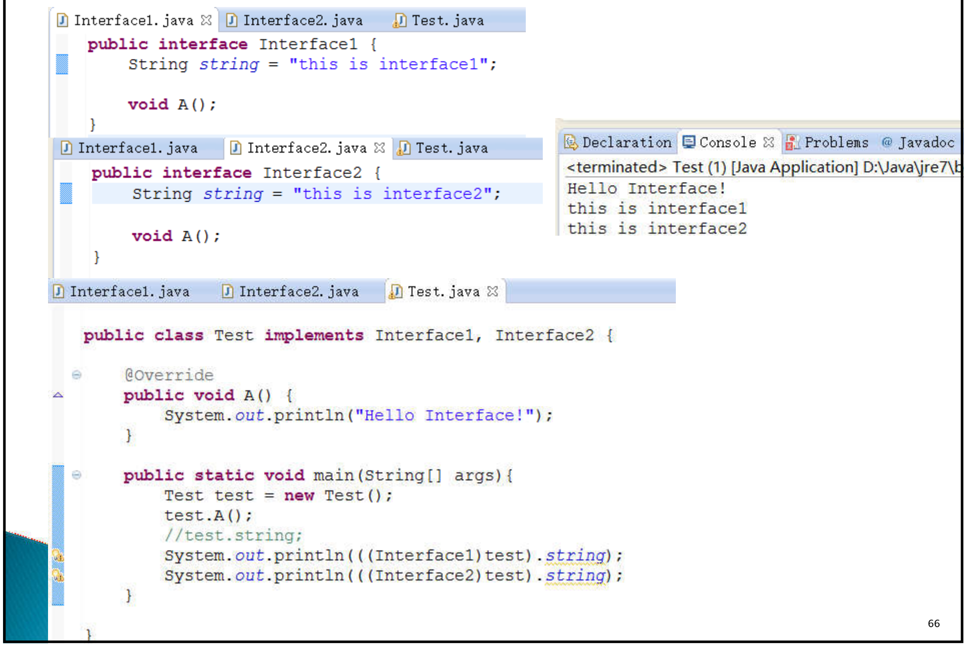
```
public interface UA {
        void doSomethingofA();
}

public interface UB {
        void doSomethingofB();
}

public interface UAandB extends UA, UB{
}

public class T implements UAandUB{
    public void doSomethingofA (){
        System.out.println("Do something of A!");
    }
    public void doSomethingofB(){
        System.out.println("Do something of B!");
    }
}
```

Interfaces

- A class can implement more than one interface (or none):

```
◦ public class S implements UA, UB{
◦     public void doSomethingofA (){
◦         System.out.println("Do something of A!");
◦     }
◦     public void doSomethingofB(){
◦         System.out.println("Do something of B!");
◦     }
◦ }
```

Multi-Interface

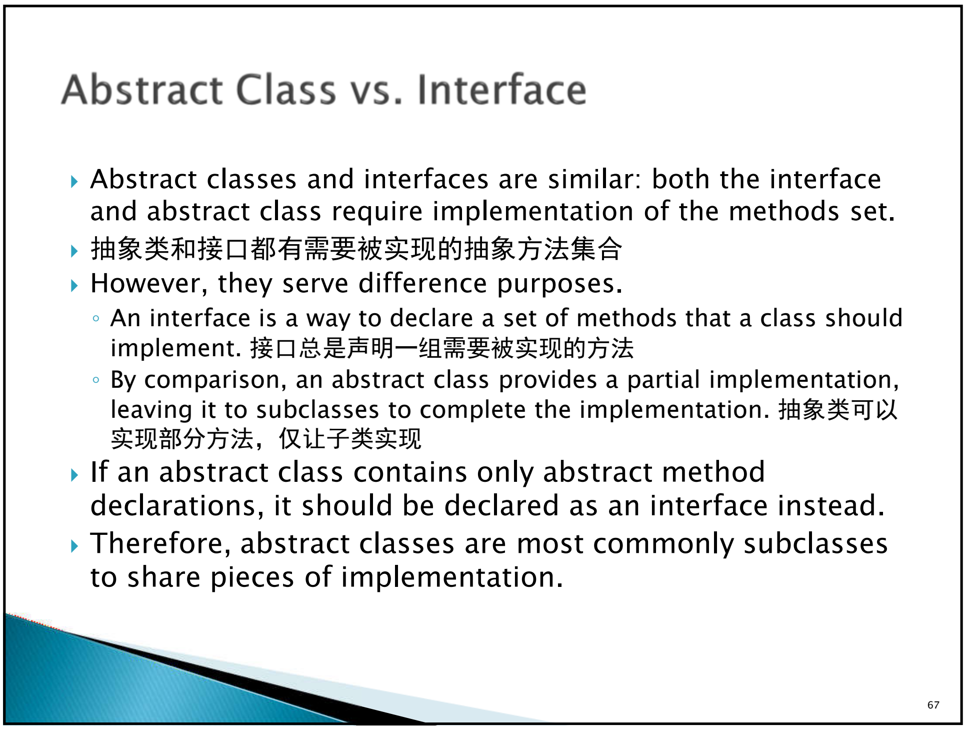


```
public interface Interface1 {
    String string = "this is interface1";

    void A();
}
```

```
public interface Interface2 {
    String string = "this is interface2";

    void A();
}
```

```
<terminated> Test (1) [Java Application] D:\Java\jre7\b
Hello Interface!
this is interface1
this is interface2
```

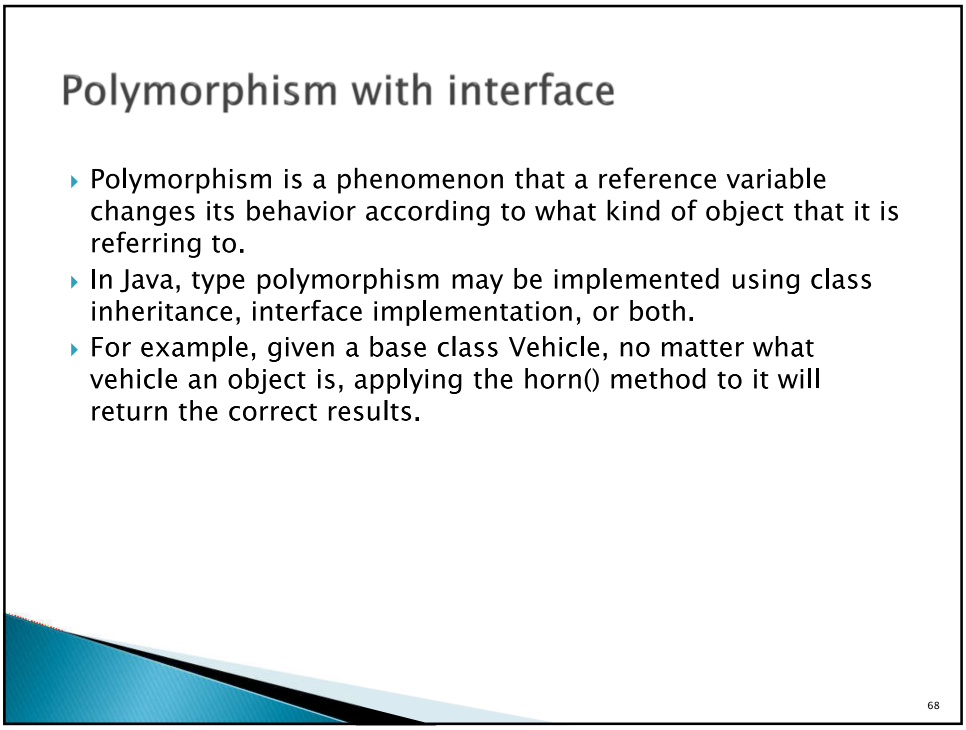
```
public class Test implements Interface1, Interface2 {

    @Override
    public void A() {
        System.out.println("Hello Interface!");
    }

    public static void main(String[] args){
        Test test = new Test();
        test.A();
        //test.string;
        System.out.println(((Interface1)test).string);
        System.out.println(((Interface2)test).string);
    }
}
```



Abstract Class vs. Interface

- Abstract classes and interfaces are similar: both the interface and abstract class require implementation of the methods set.
- 抽象类和接口都有需要被实现的抽象方法集合
- However, they serve difference purposes.
  - An interface is a way to declare a set of methods that a class should implement. 接口总是声明一组需要被实现的方法
  - By comparison, an abstract class provides a partial implementation, leaving it to subclasses to complete the implementation. 抽象类可以实现部分方法，仅让子类实现
- If an abstract class contains only abstract method declarations, it should be declared as an interface instead.
- Therefore, abstract classes are most commonly subclasses to share pieces of implementation.

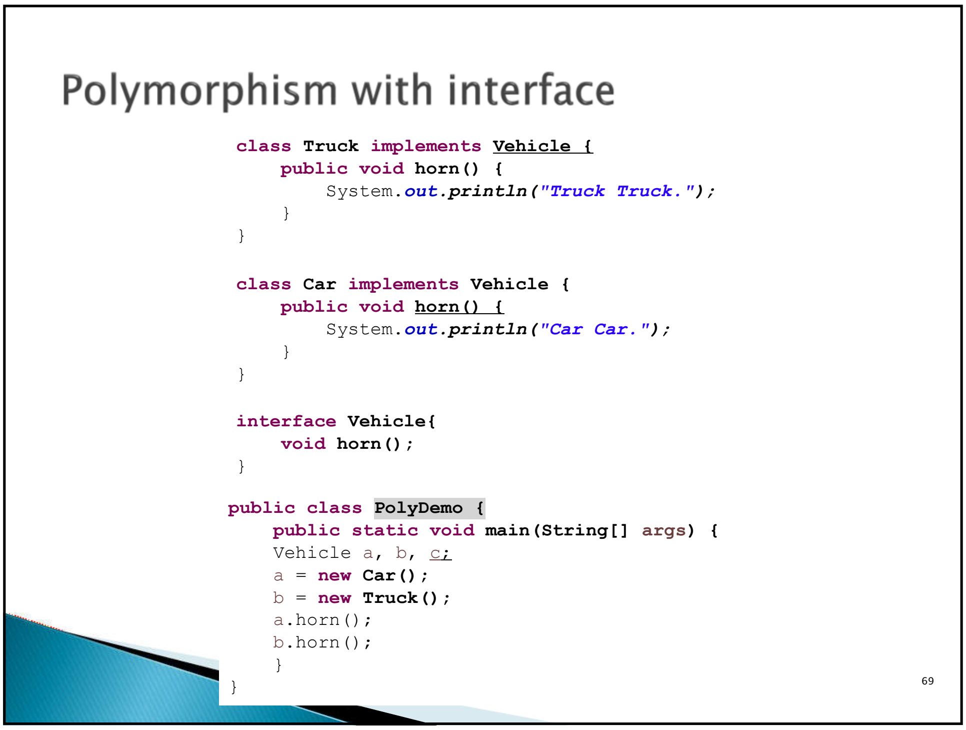
67



Polymorphism with interface

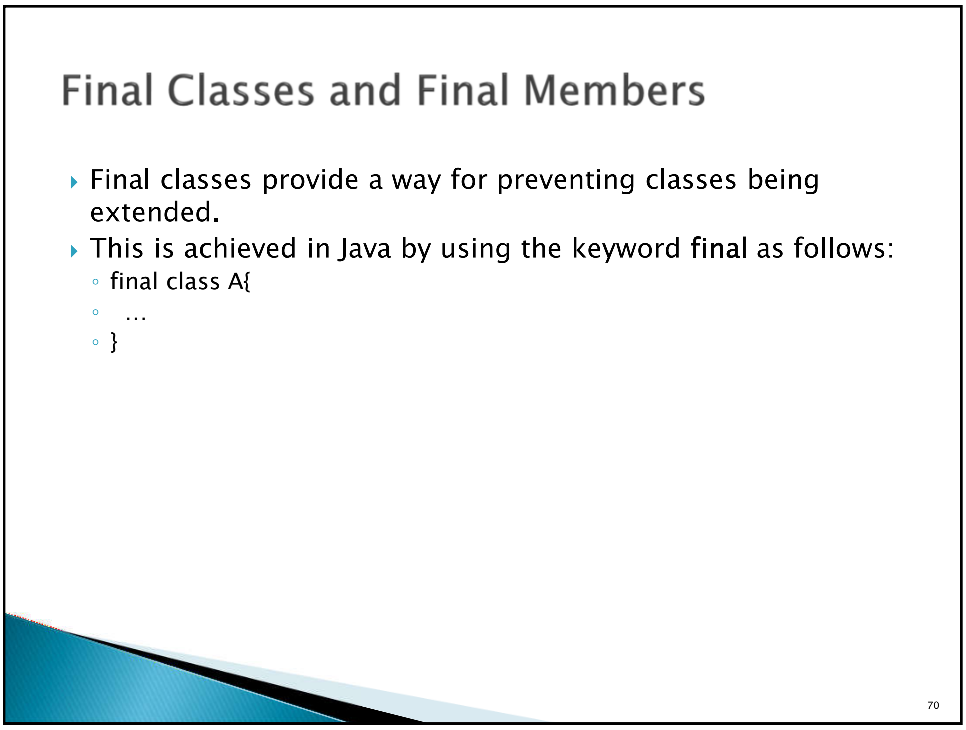
- Polymorphism is a phenomenon that a reference variable changes its behavior according to what kind of object that it is referring to.
- In Java, type polymorphism may be implemented using class inheritance, interface implementation, or both.
- For example, given a base class Vehicle, no matter what vehicle an object is, applying the horn() method to it will return the correct results.

68



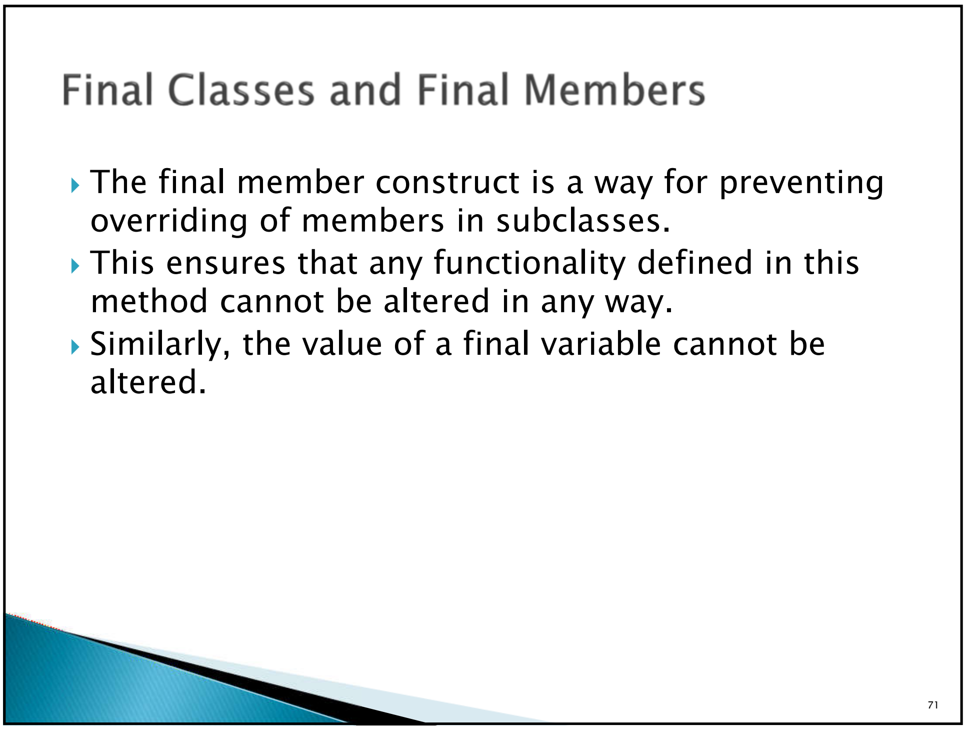
Polymorphism with interface

```
class Truck implements Vehicle {
    public void horn() {
        System.out.println("Truck Truck.");
    }
}

class Car implements Vehicle {
    public void horn() {
        System.out.println("Car Car.");
    }
}

interface Vehicle{
    void horn();
}

public class PolyDemo {
    public static void main(String[] args) {
    Vehicle a, b, c;
    a = new Car();
    b = new Truck();
    a.horn();
    b.horn();
    }
}
```



Final Classes and Final Members

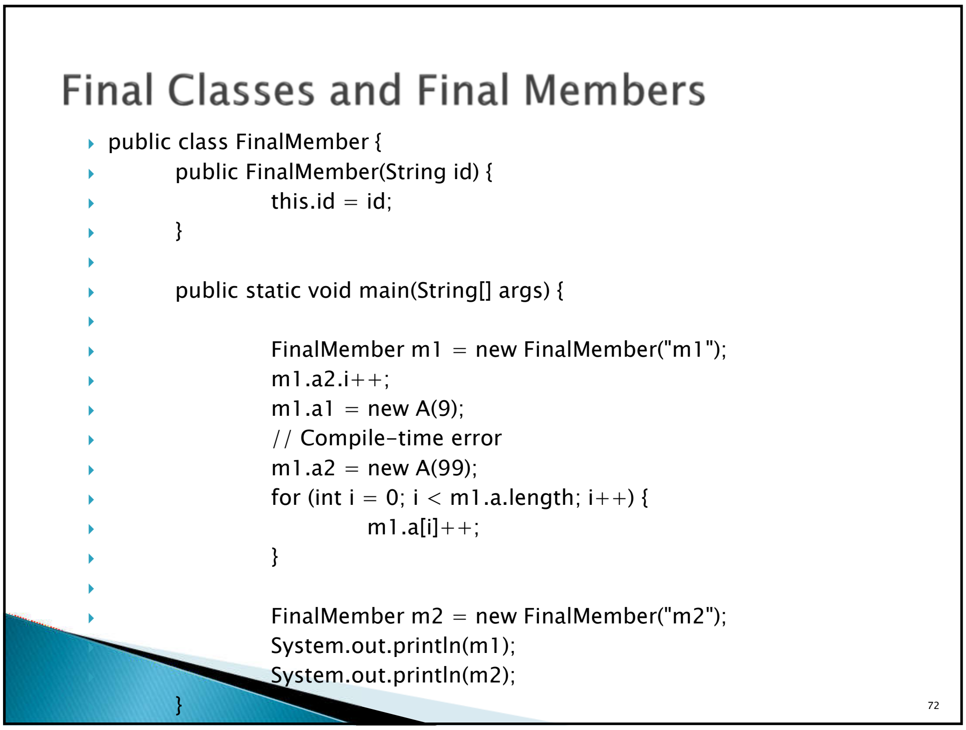
- Final classes provide a way for preventing classes being extended.
- This is achieved in Java by using the keyword **final** as follows:
  - final class A{
  - …
  - }



Final Classes and Final Members

- The final member construct is a way for preventing overriding of members in subclasses.
- This ensures that any functionality defined in this method cannot be altered in any way.
- Similarly, the value of a final variable cannot be altered.

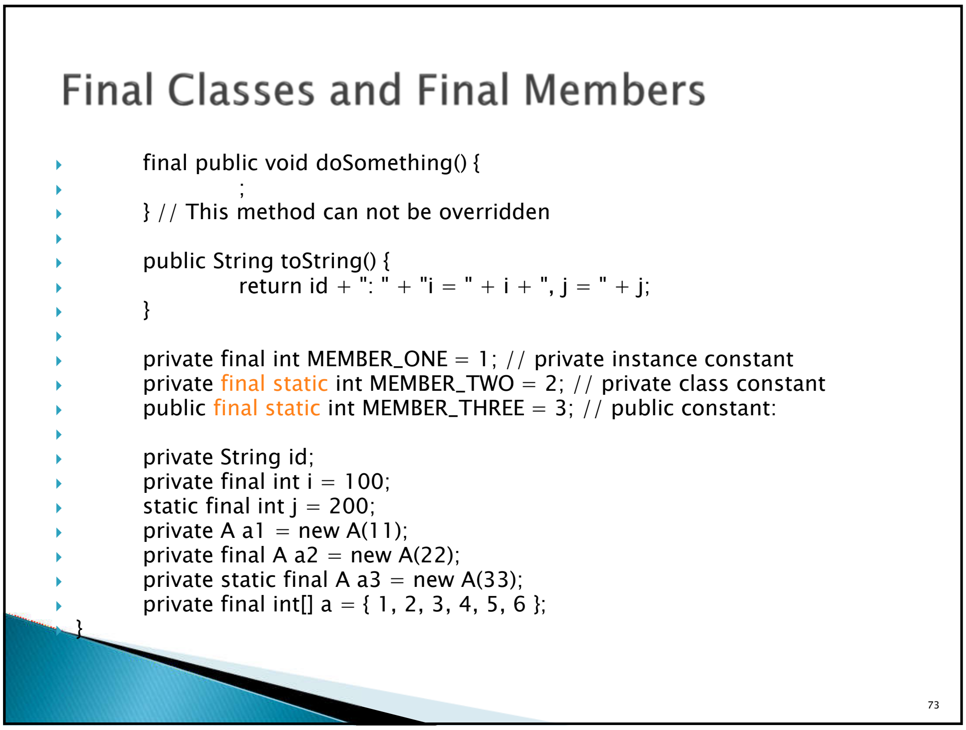
71



Final Classes and Final Members

```
public class FinalMember {
        public FinalMember(String id) {
                this.id = id;
        }

        public static void main(String[] args) {

                FinalMember m1 = new FinalMember("m1");
                m1.a2.i++;
                m1.a1 = new A(9);
                // Compile-time error
                m1.a2 = new A(99);
                for (int i = 0; i < m1.a.length; i++) {
                        m1.a[i]++;
                }

                FinalMember m2 = new FinalMember("m2");
                System.out.println(m1);
                System.out.println(m2);
        }
```

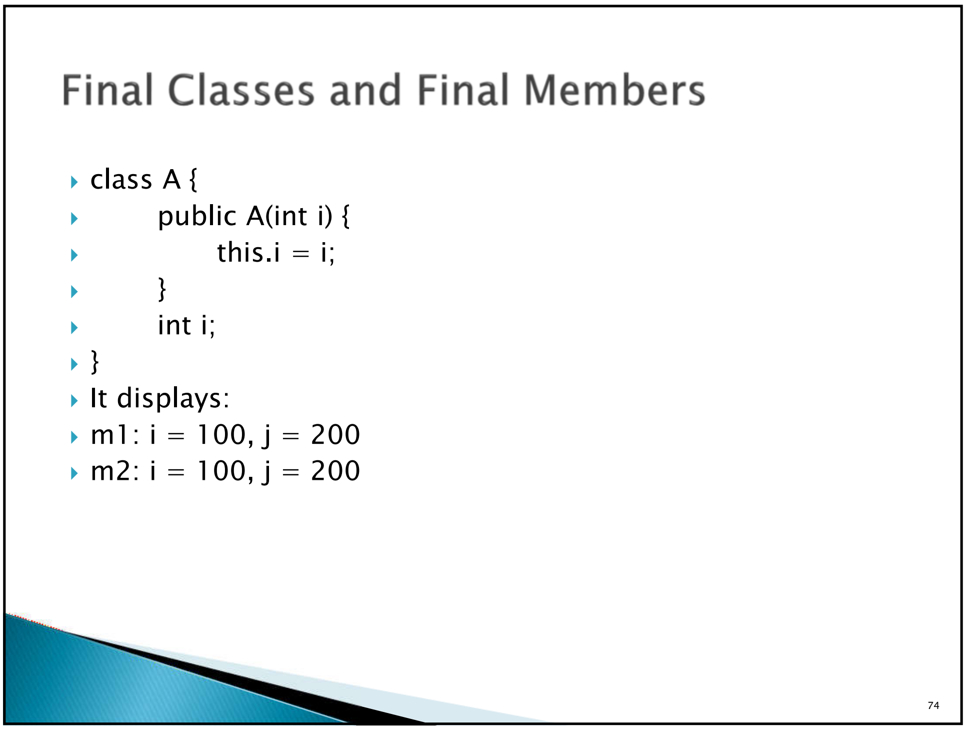
72



Final Classes and Final Members

```
        final public void doSomething() {
                ;
        } // This method can not be overridden

        public String toString() {
                return id + ": " + "i = " + i + ", j = " + j;
        }

        private final int MEMBER_ONE = 1; // private instance constant
        private final static int MEMBER_TWO = 2; // private class constant
        public final static int MEMBER_THREE = 3; // public constant:

        private String id;
        private final int i = 100;
        static final int j = 200;
        private A a1 = new A(11);
        private final A a2 = new A(22);
        private static final A a3 = new A(33);
        private final int[] a = { 1, 2, 3, 4, 5, 6 };
}
```

73

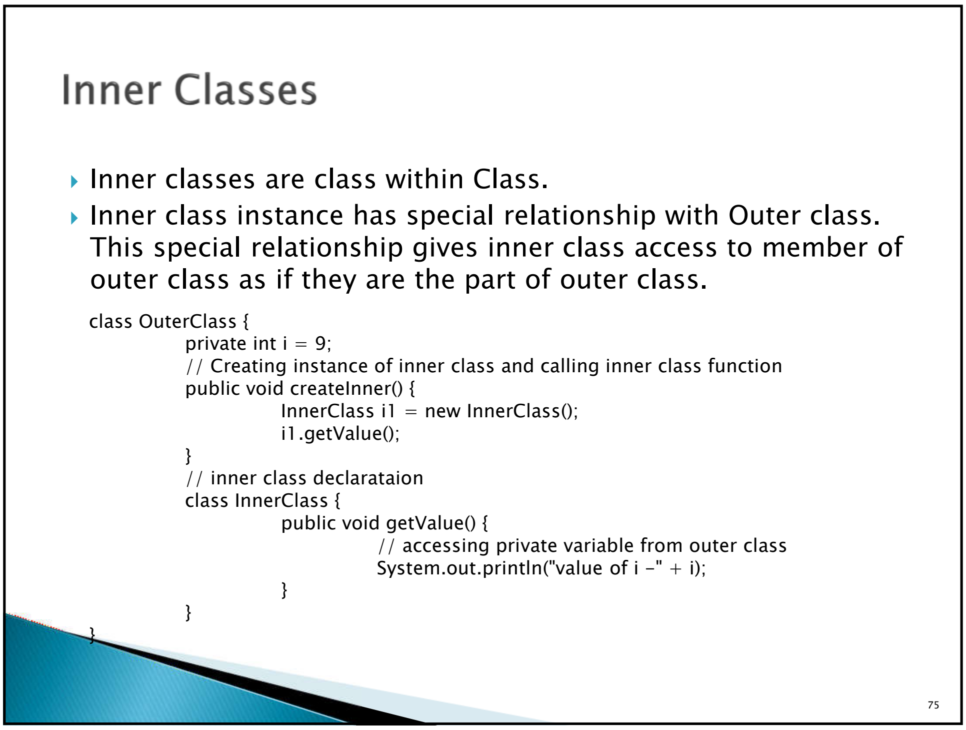


Final Classes and Final Members

```
class A {
        public A(int i) {
                this.i = i;
        }
        int i;
}
```

- It displays:
- m1: i = 100, j = 200
- m2: i = 100, j = 200

74

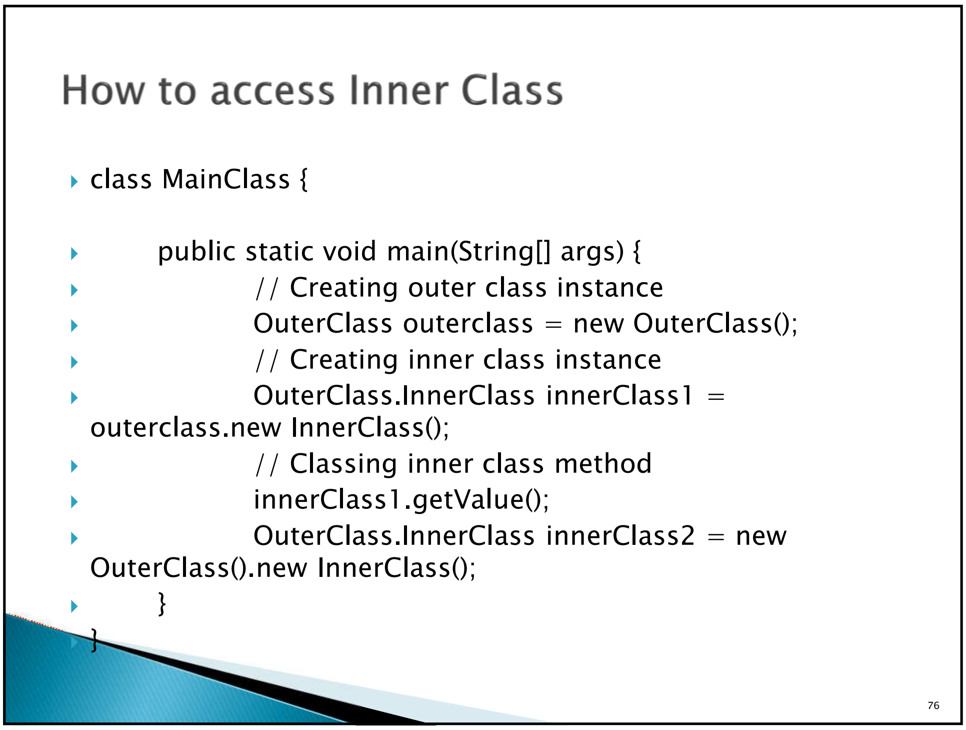


Inner Classes

- Inner classes are class within Class.
- Inner class instance has special relationship with Outer class. This special relationship gives inner class access to member of outer class as if they are the part of outer class.

```
class OuterClass {
        private int i = 9;
        // Creating instance of inner class and calling inner class function
        public void createInner() {
                InnerClass i1 = new InnerClass();
                i1.getValue();
        }
        // inner class declarataion
        class InnerClass {
                public void getValue() {
                        // accessing private variable from outer class
                        System.out.println("value of i -" + i);
                }
        }
}
```

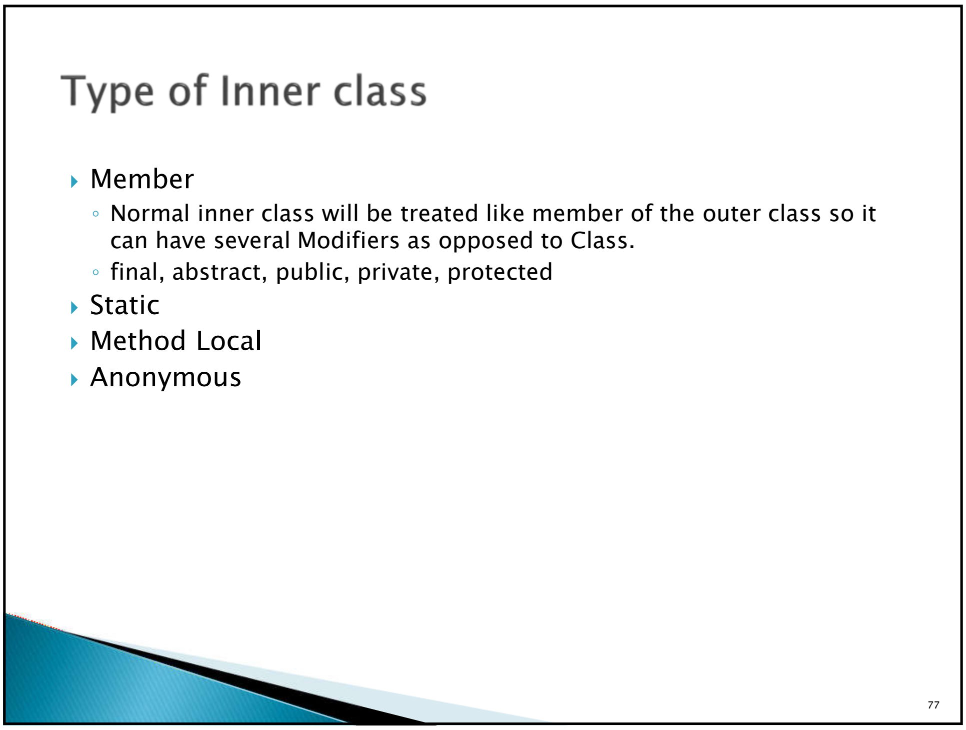
75



How to access Inner Class

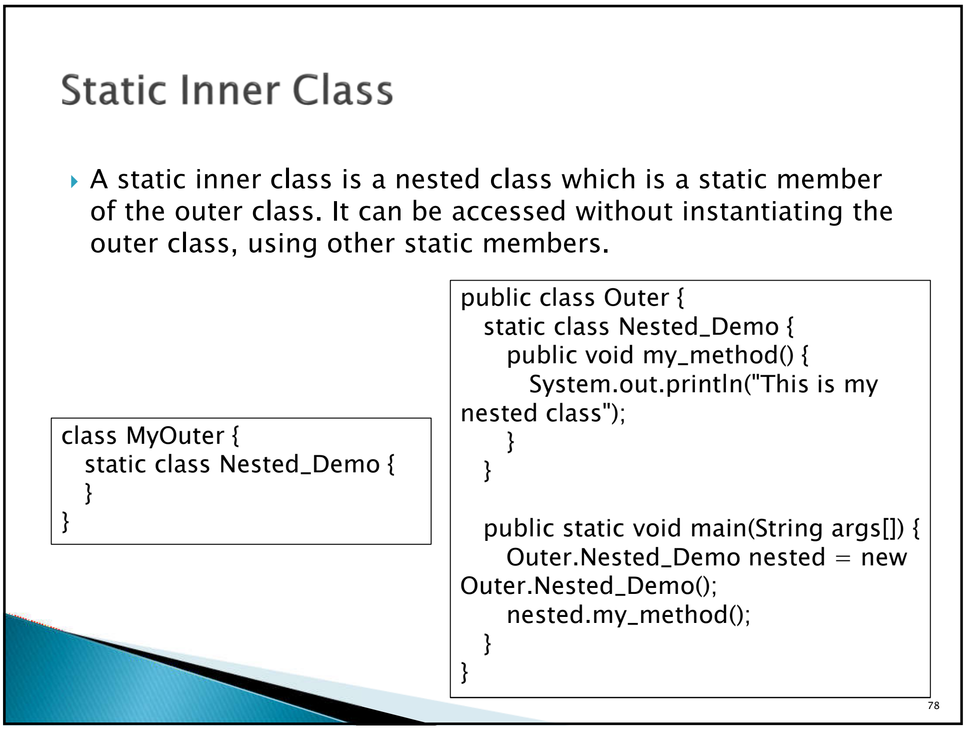
```
class MainClass {

      public static void main(String[] args) {
              // Creating outer class instance
              OuterClass outerclass = new OuterClass();
              // Creating inner class instance
              OuterClass.InnerClass innerClass1 =
outerclass.new InnerClass();
              // Classing inner class method
              innerClass1.getValue();
              OuterClass.InnerClass innerClass2 = new
OuterClass().new InnerClass();
      }
}
```

76



Type of Inner class

- Member
  - Normal inner class will be treated like member of the outer class so it can have several Modifiers as opposed to Class.
  - final, abstract, public, private, protected
- Static
- Method Local
- Anonymous

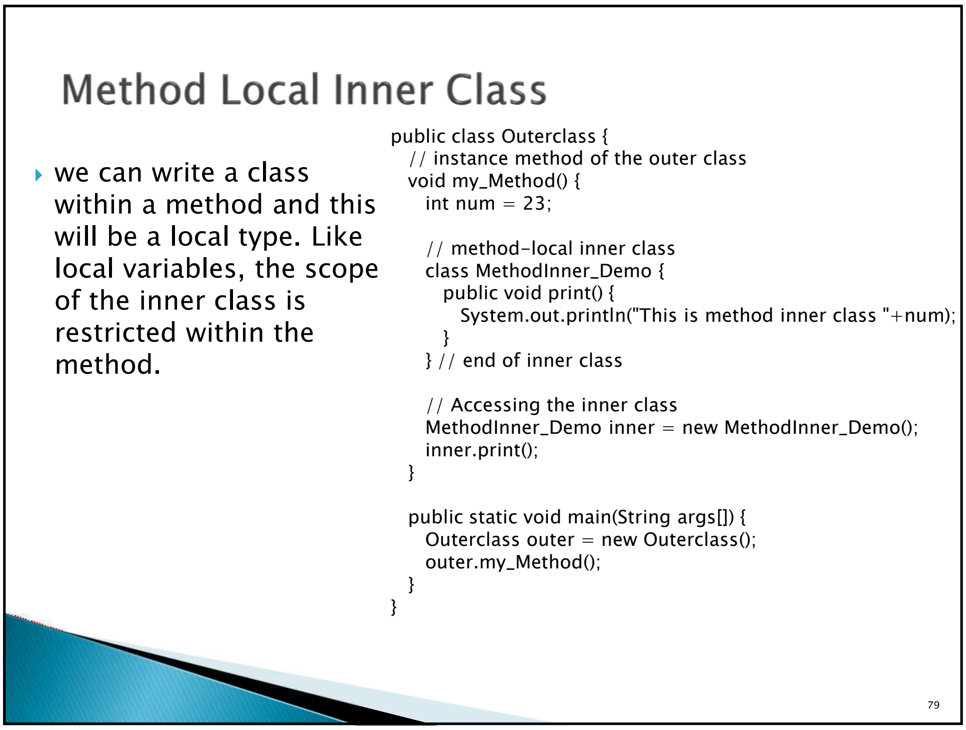


Static Inner Class

- A static inner class is a nested class which is a static member of the outer class. It can be accessed without instantiating the outer class, using other static members.

```
class MyOuter {
   static class Nested_Demo {
   }
}
```

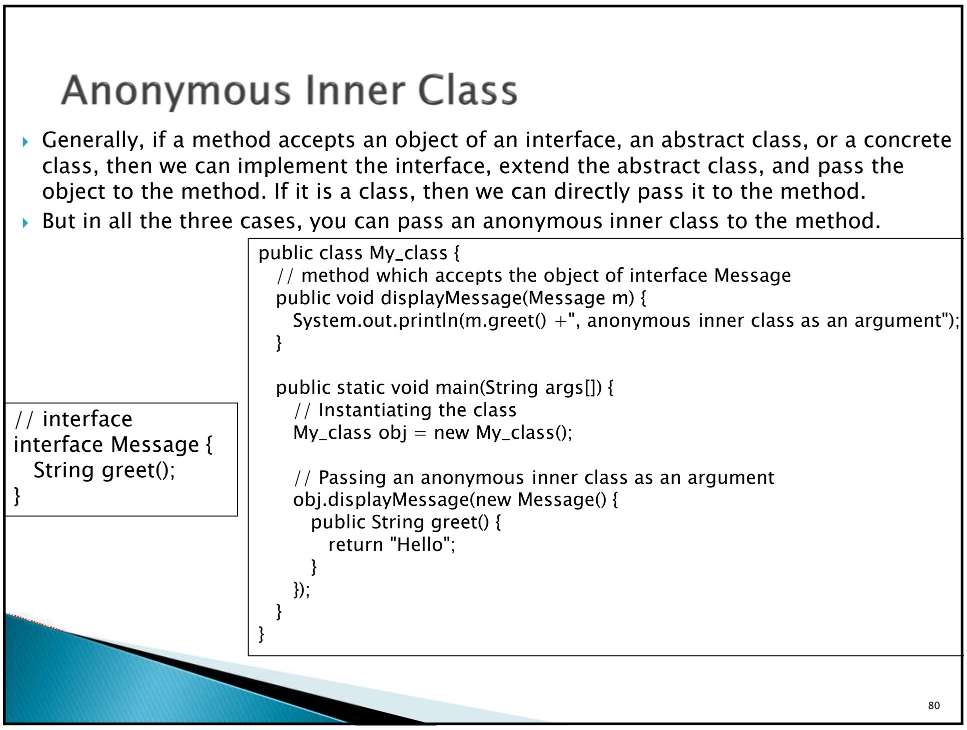
```
public class Outer {
   static class Nested_Demo {
      public void my_method() {
         System.out.println("This is my
nested class");
      }
   }

   public static void main(String args[]) {
      Outer.Nested_Demo nested = new
Outer.Nested_Demo();
      nested.my_method();
   }
}
```



Method Local Inner Class

- we can write a class within a method and this will be a local type. Like local variables, the scope of the inner class is restricted within the method.

```
public class Outerclass {
  // instance method of the outer class
  void my_Method() {
    int num = 23;

    // method-local inner class
    class MethodInner_Demo {
      public void print() {
        System.out.println("This is method inner class "+num);
      }
    } // end of inner class

    // Accessing the inner class
    MethodInner_Demo inner = new MethodInner_Demo();
    inner.print();
  }

  public static void main(String args[]) {
    Outerclass outer = new Outerclass();
    outer.my_Method();
  }
}
```

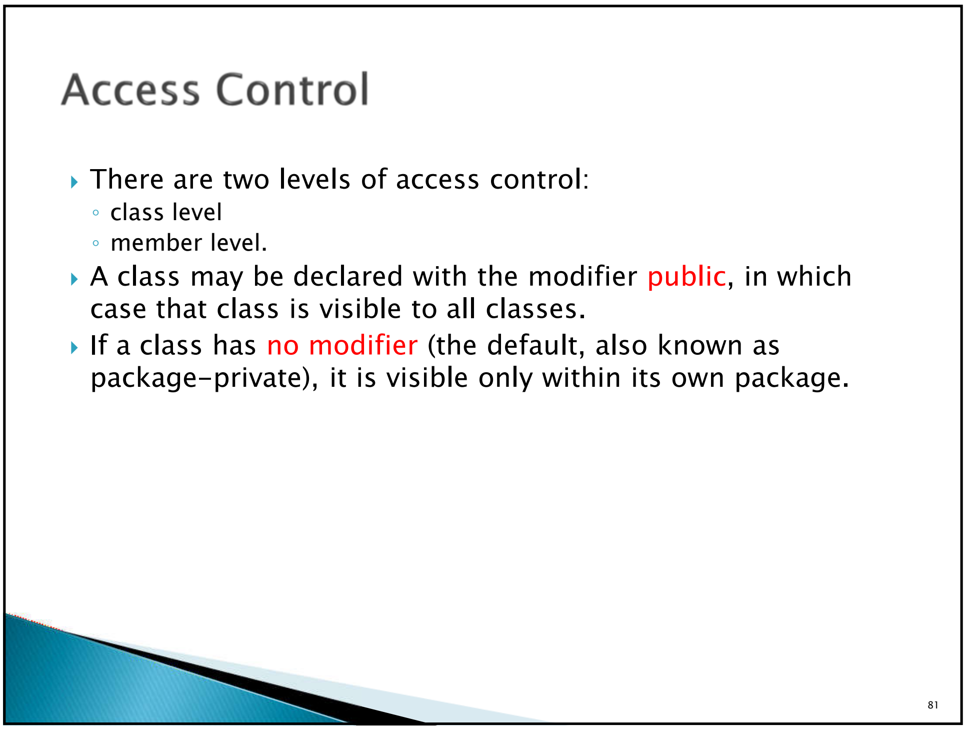


Anonymous Inner Class

- Generally, if a method accepts an object of an interface, an abstract class, or a concrete class, then we can implement the interface, extend the abstract class, and pass the object to the method. If it is a class, then we can directly pass it to the method.
- But in all the three cases, you can pass an anonymous inner class to the method.

```
// interface
interface Message {
  String greet();
}
```

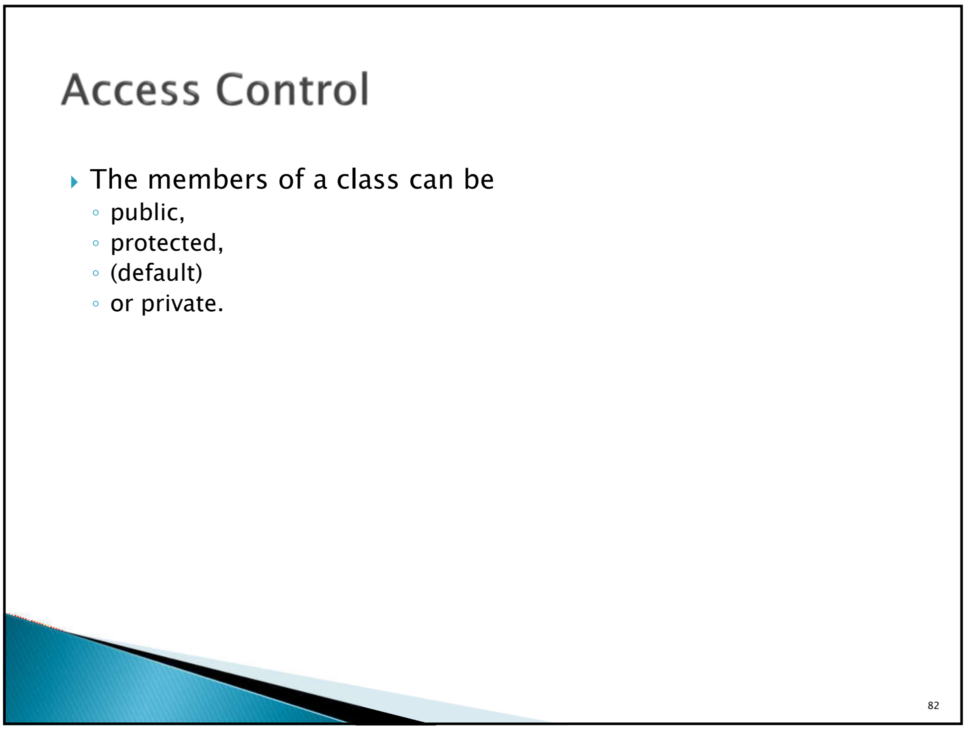
```
public class My_class {
  // method which accepts the object of interface Message
  public void displayMessage(Message m) {
    System.out.println(m.greet() +", anonymous inner class as an argument");
  }

  public static void main(String args[]) {
    // Instantiating the class
    My_class obj = new My_class();

    // Passing an anonymous inner class as an argument
    obj.displayMessage(new Message() {
      public String greet() {
        return "Hello";
      }
    });
  }
}
```



Access Control

- There are two levels of access control:
  - class level
  - member level.
- A class may be declared with the modifier public, in which case that class is visible to all classes.
- If a class has no modifier (the default, also known as package-private), it is visible only within its own package.

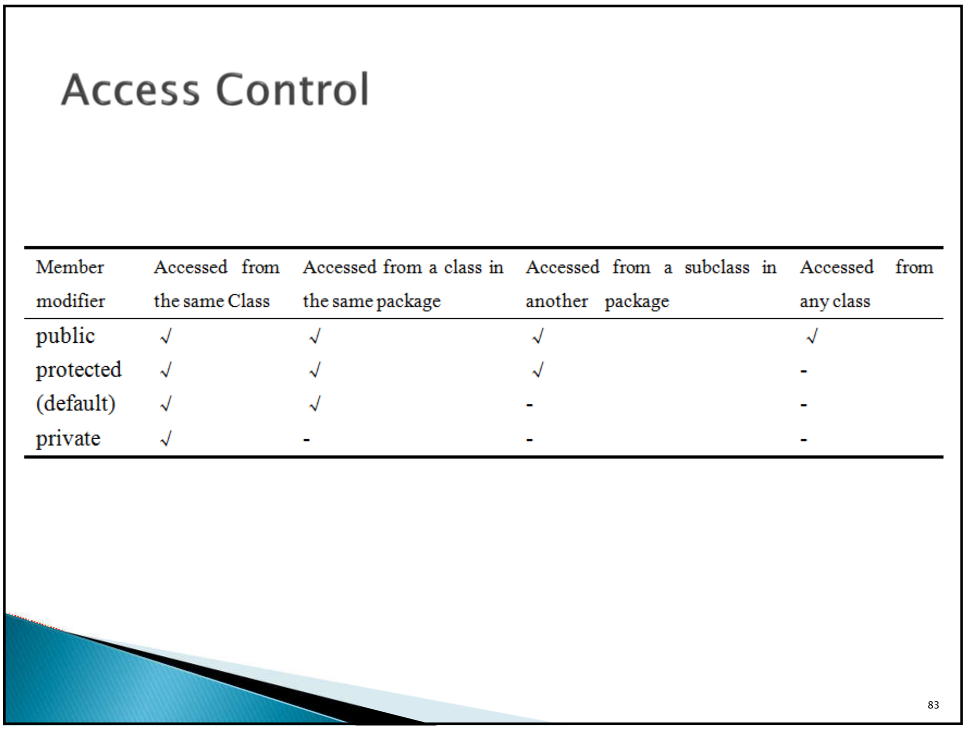
81



Access Control

- The members of a class can be
  - public,
  - protected,
  - (default)
  - or private.

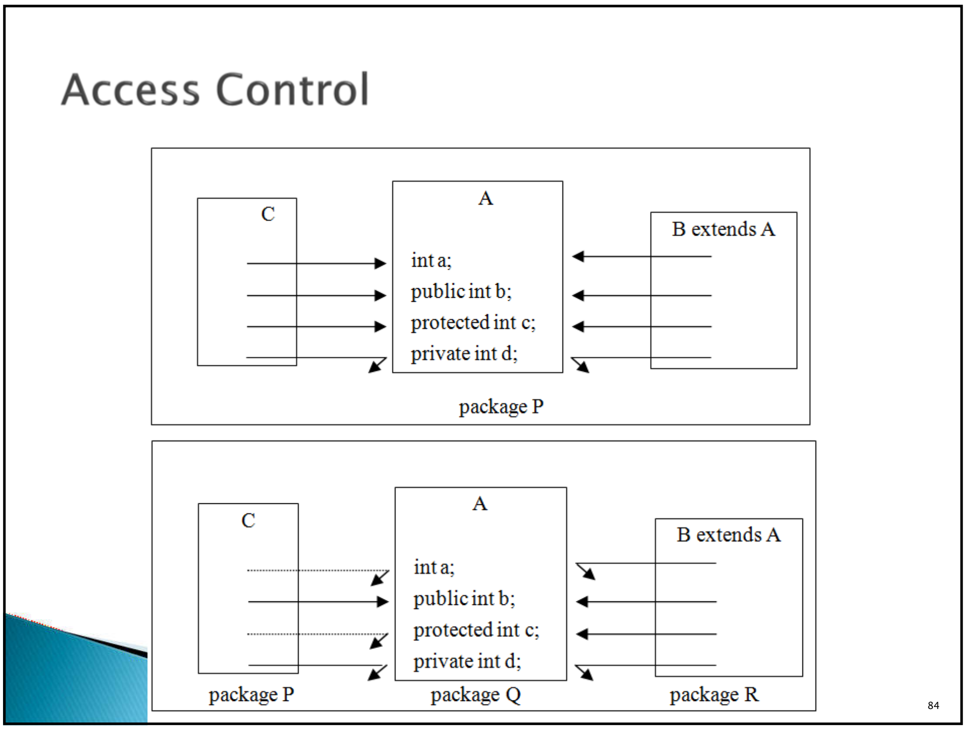
82



Access Control

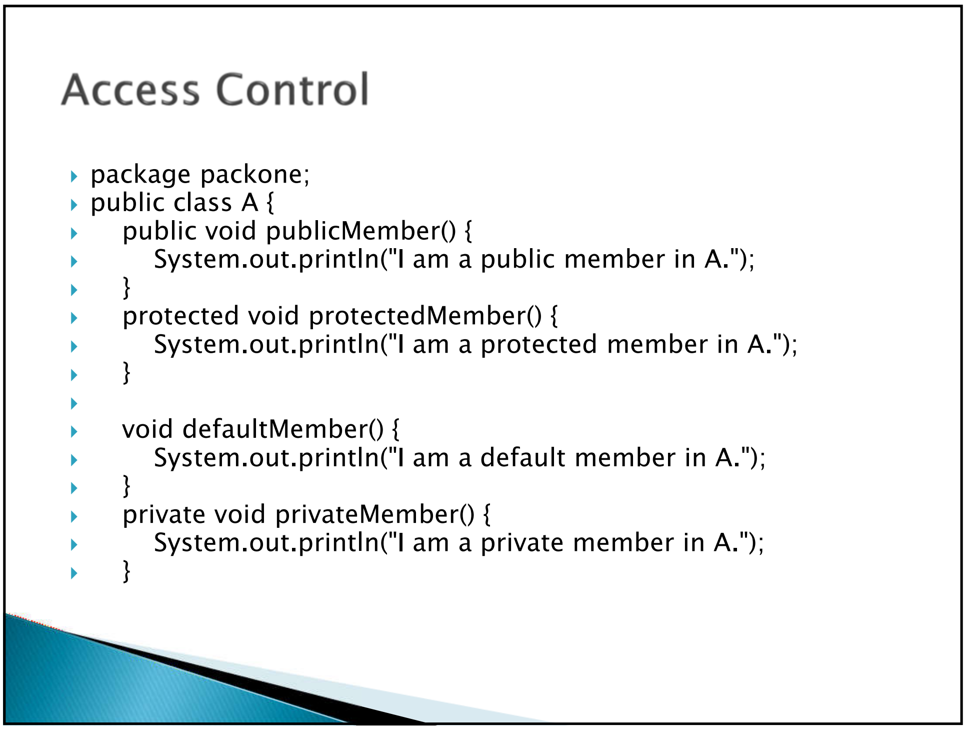
Member modifier	Accessed from the same Class	Accessed from a class in the same package	Accessed from a subclass in another package	Accessed from any class
public	√	√	√	√
protected	√	√	√	-
(default)	√	√	-	-
private	√	-	-	-

83



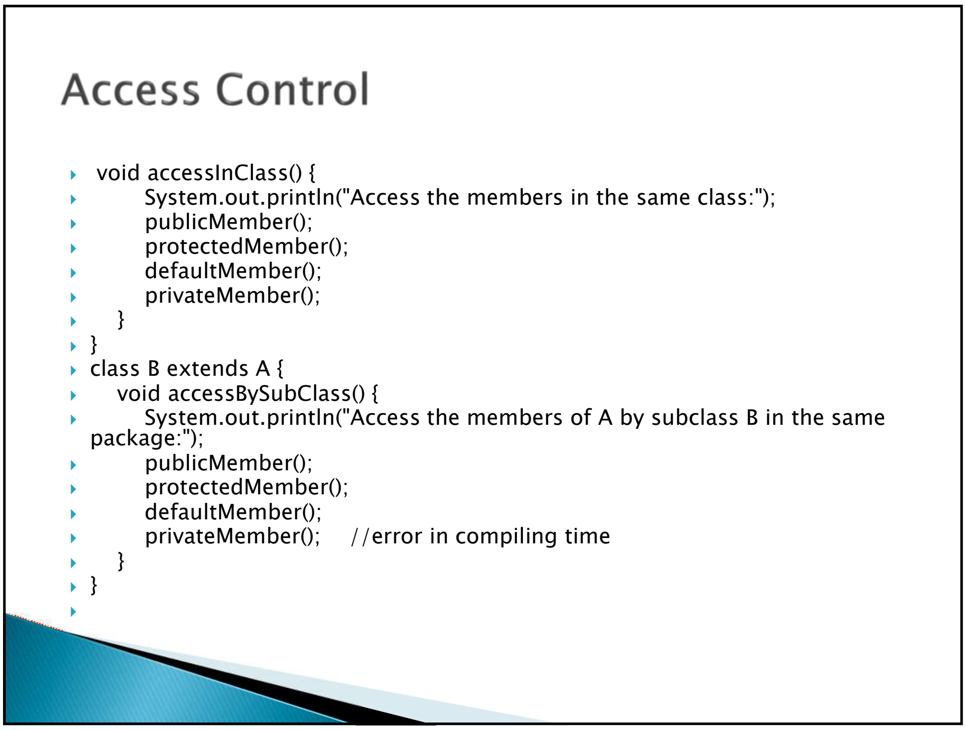
Access Control

84



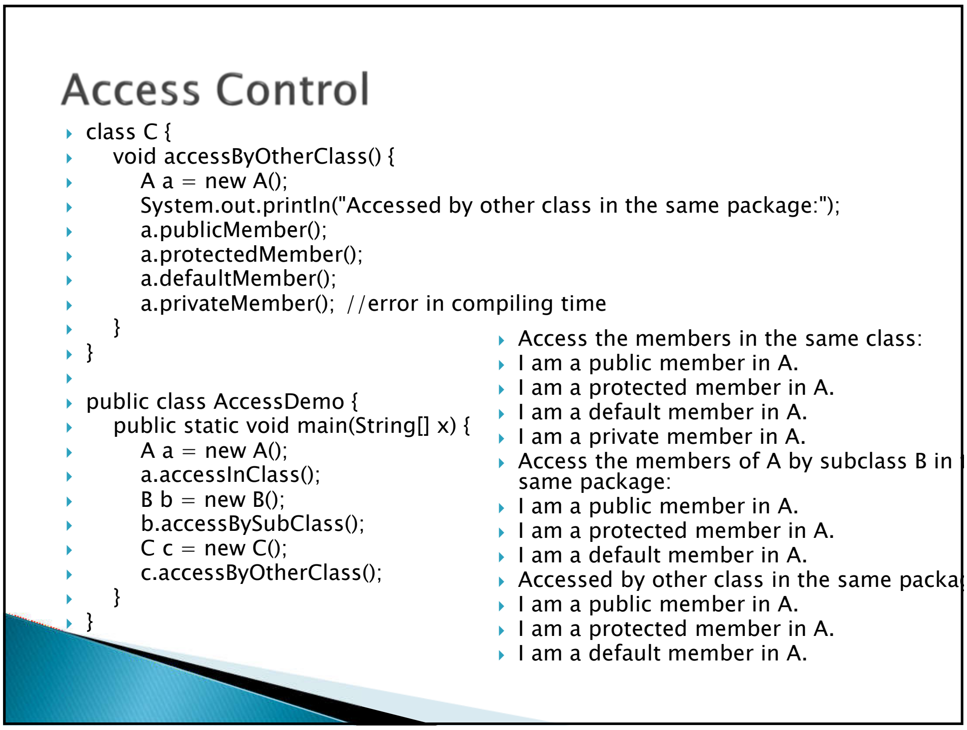
Access Control

```
package packone;
public class A {
    public void publicMember() {
        System.out.println("I am a public member in A.");
    }
    protected void protectedMember() {
        System.out.println("I am a protected member in A.");
    }

    void defaultMember() {
        System.out.println("I am a default member in A.");
    }
    private void privateMember() {
        System.out.println("I am a private member in A.");
    }
```



Access Control

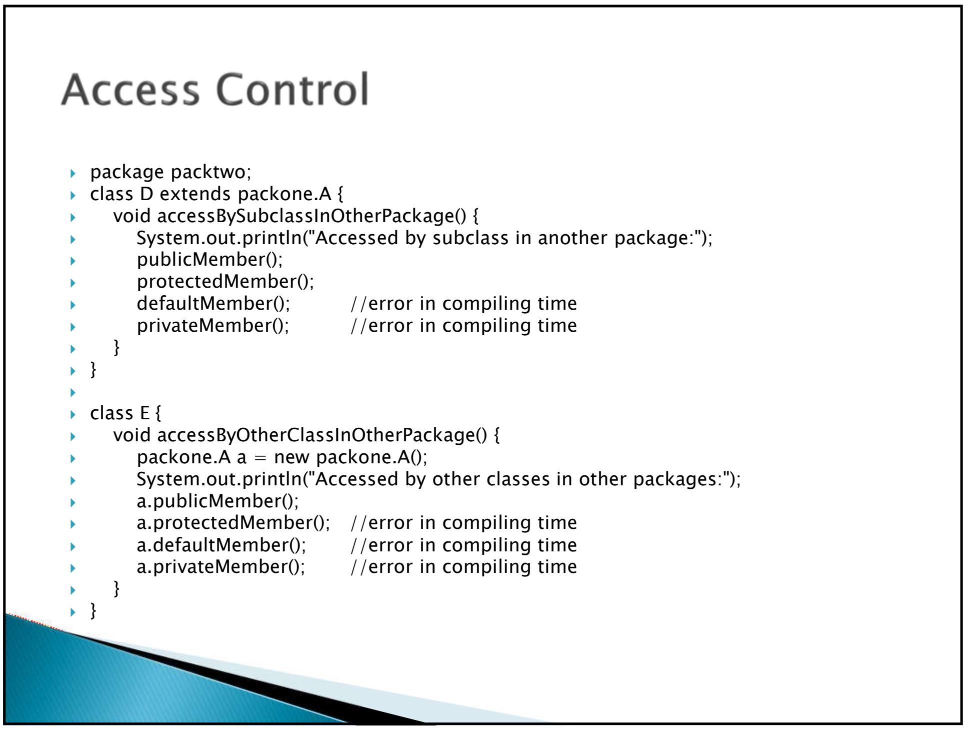
```
 void accessInClass() {
        System.out.println("Access the members in the same class:");
        publicMember();
        protectedMember();
        defaultMember();
        privateMember();
    }
}
class B extends A {
    void accessBySubClass() {
        System.out.println("Access the members of A by subclass B in the same
package:");
        publicMember();
        protectedMember();
        defaultMember();
        privateMember();    //error in compiling time
    }
}
```



Access Control

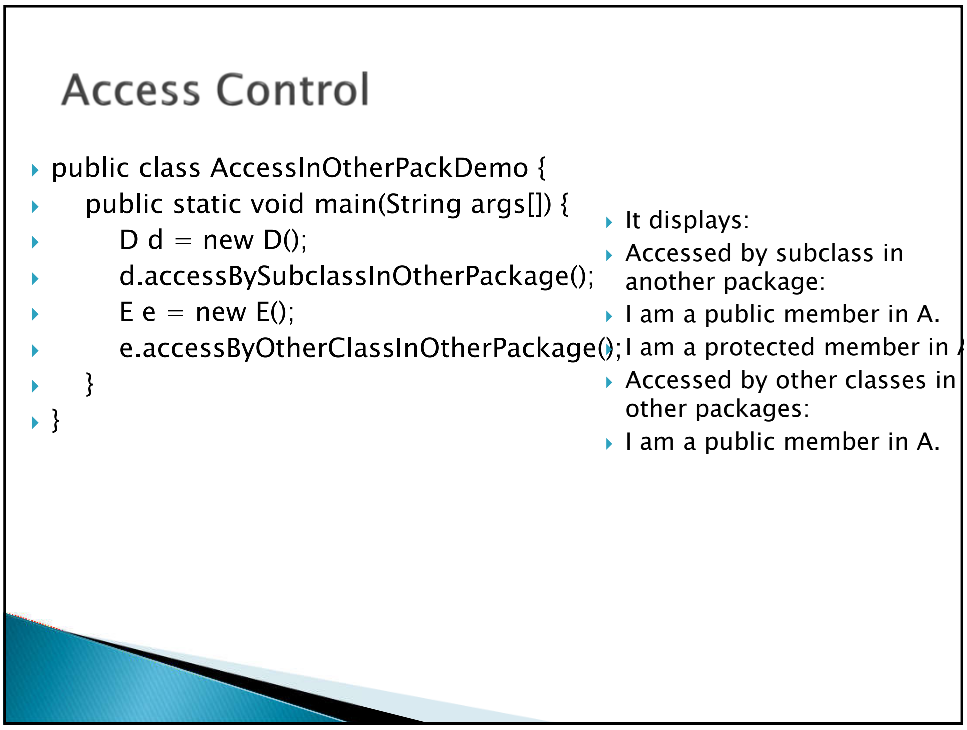
```
class C {
    void accessByOtherClass() {
        A a = new A();
        System.out.println("Accessed by other class in the same package:");
        a.publicMember();
        a.protectedMember();
        a.defaultMember();
        a.privateMember();  //error in compiling time
    }
}

public class AccessDemo {
    public static void main(String[] x) {
        A a = new A();
        a.accessInClass();
        B b = new B();
        b.accessBySubClass();
        C c = new C();
        c.accessByOtherClass();
    }
}
```

```
Access the members in the same class:
I am a public member in A.
I am a protected member in A.
I am a default member in A.
I am a private member in A.
Access the members of A by subclass B in t
same package:
I am a public member in A.
I am a protected member in A.
I am a default member in A.
Accessed by other class in the same packa
I am a public member in A.
I am a protected member in A.
I am a default member in A.
```



Access Control

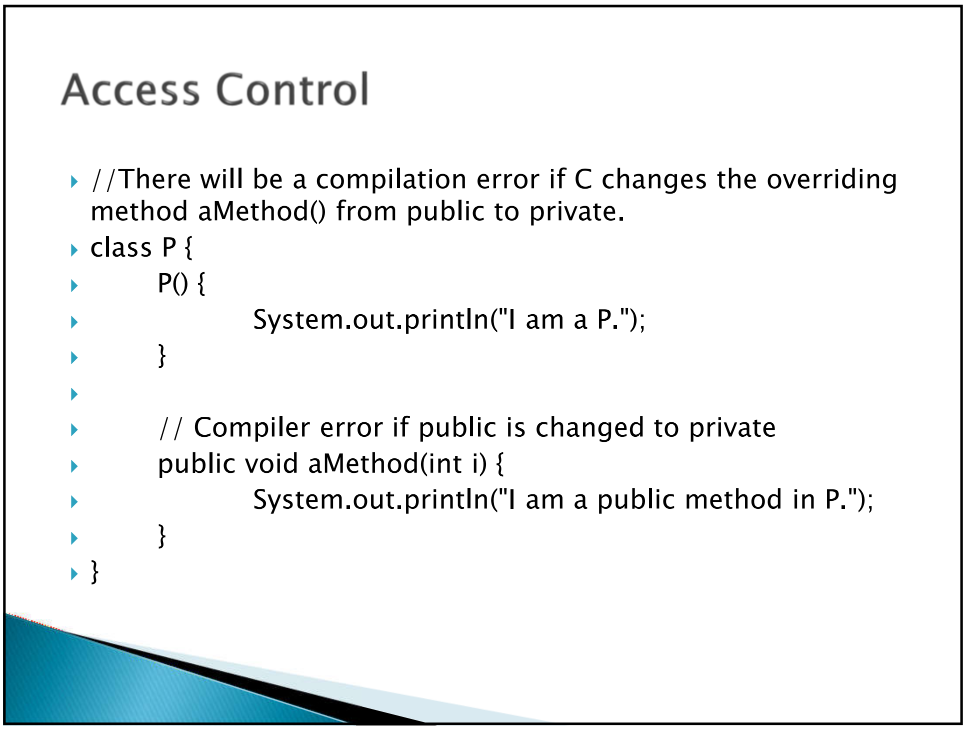
```
package packtwo;
class D extends packone.A {
    void accessBySubclassInOtherPackage() {
        System.out.println("Accessed by subclass in another package:");
        publicMember();
        protectedMember();
        defaultMember();        //error in compiling time
        privateMember();        //error in compiling time
    }
}

class E {
    void accessByOtherClassInOtherPackage() {
        packone.A a = new packone.A();
        System.out.println("Accessed by other classes in other packages:");
        a.publicMember();
        a.protectedMember();  //error in compiling time
        a.defaultMember();     //error in compiling time
        a.privateMember();     //error in compiling time
    }
}
```



Access Control

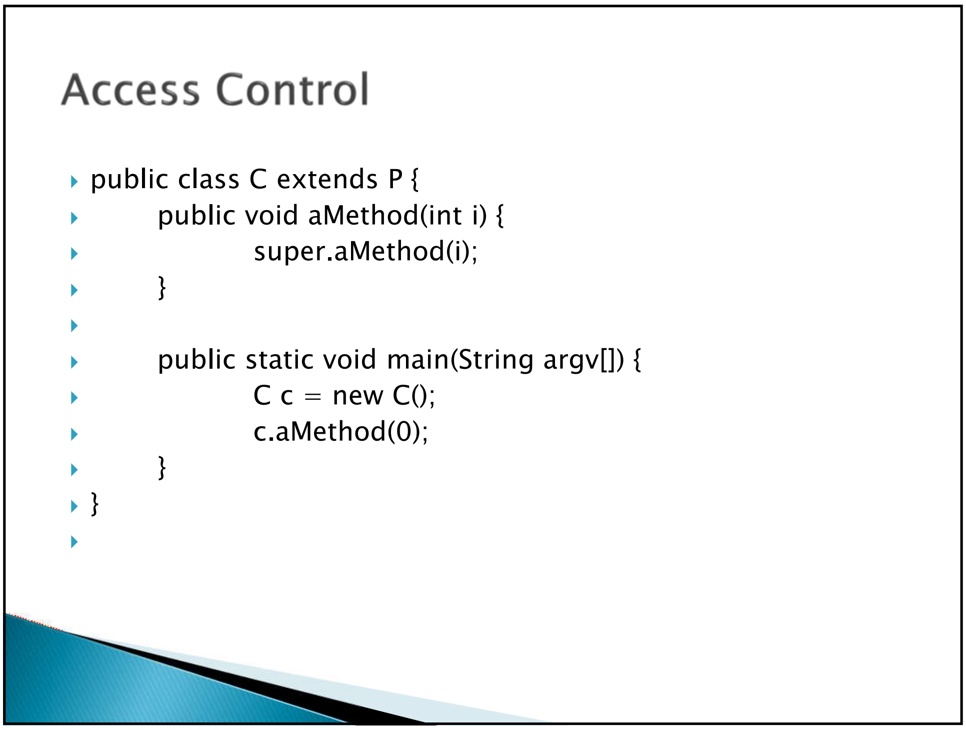
```
public class AccessInOtherPackDemo {
    public static void main(String args[]) {
        D d = new D();
        d.accessBySubclassInOtherPackage();
        E e = new E();
        e.accessByOtherClassInOtherPackage();
    }
}
```

- It displays:
- Accessed by subclass in another package:
- I am a public member in A.
- I am a protected member in
- Accessed by other classes in other packages:
- I am a public member in A.



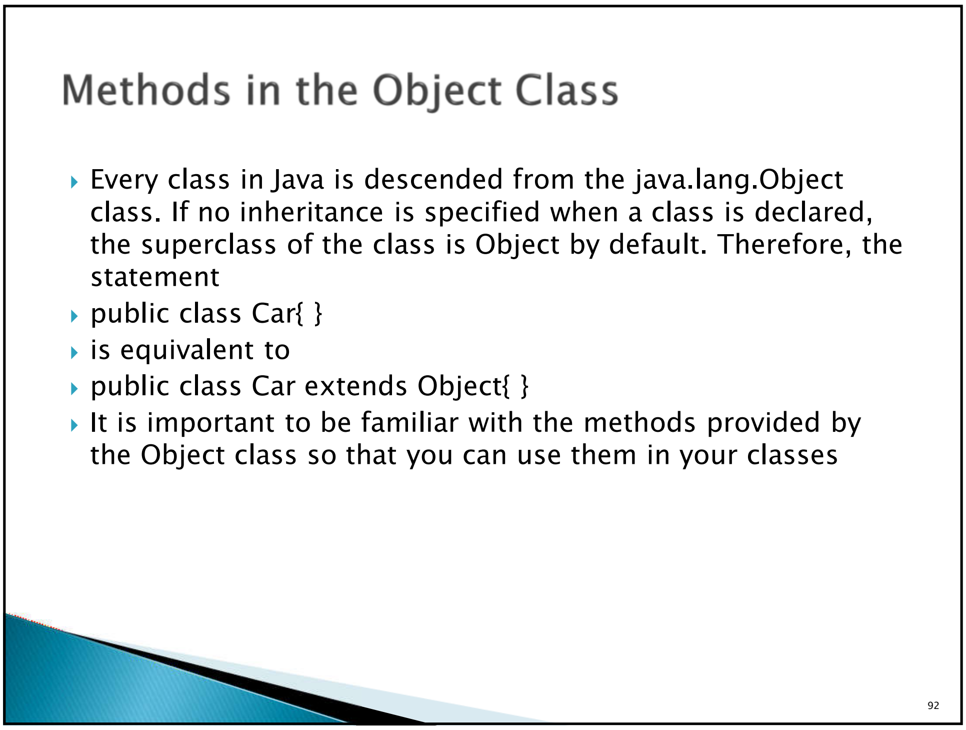
Access Control

```
//There will be a compilation error if C changes the overriding
method aMethod() from public to private.
class P {
        P() {
                System.out.println("I am a P.");
        }

        // Compiler error if public is changed to private
        public void aMethod(int i) {
                System.out.println("I am a public method in P.");
        }
}
```



Access Control

```
public class C extends P {
        public void aMethod(int i) {
                super.aMethod(i);
        }

        public static void main(String argv[]) {
                C c = new C();
                c.aMethod(0);
        }
}
```



Methods in the Object Class

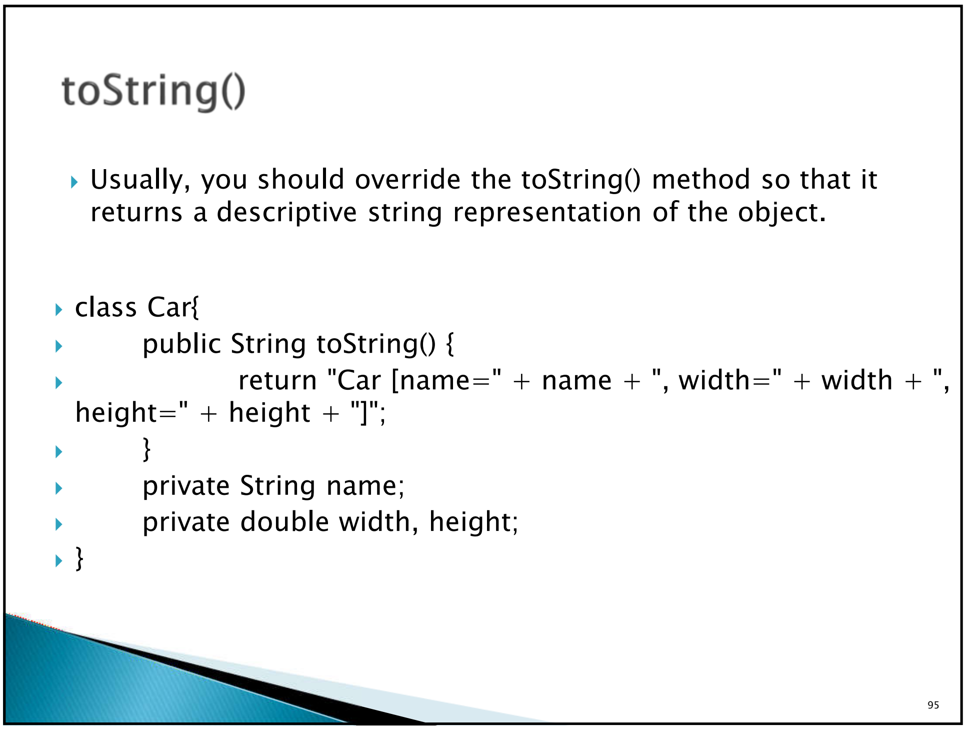
- Every class in Java is descended from the java.lang.Object class. If no inheritance is specified when a class is declared, the superclass of the class is Object by default. Therefore, the statement
- public class Car{ }
- is equivalent to
- public class Car extends Object{ }
- It is important to be familiar with the methods provided by the Object class so that you can use them in your classes

92

- toString()
- equals()
- hashCode()
- clone()

toString()

- Invoking toString() on an object returns a string that represents the object.
- By default, it returns a string consisting of a class name of which the object is an instance, an at sign(@) and the object's hash code in hexadecimal.

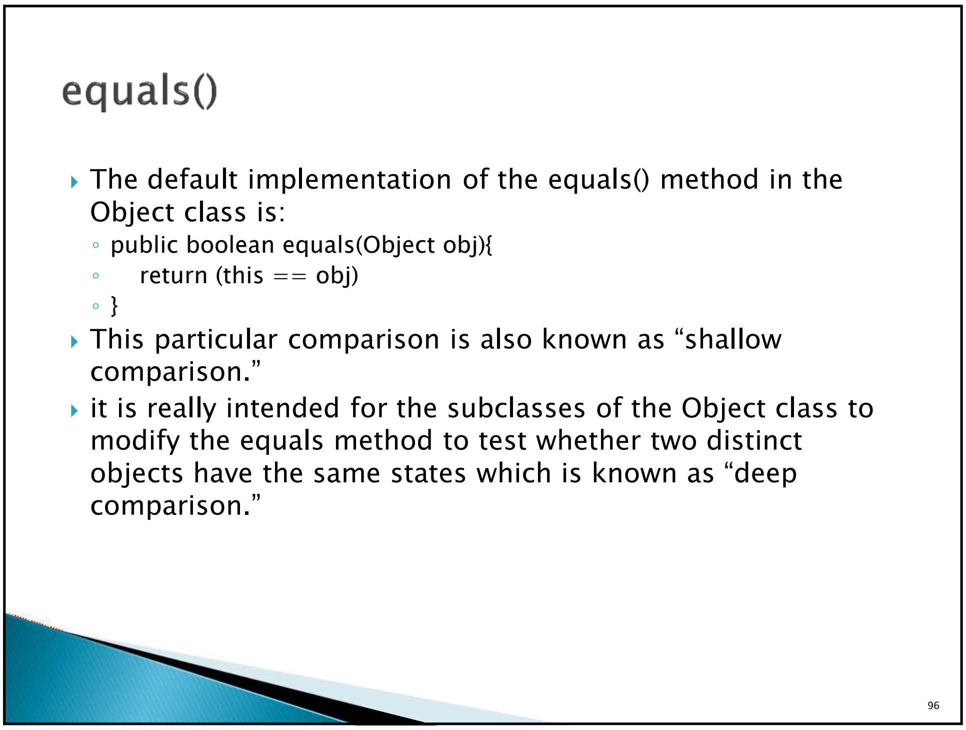


toString()

- Usually, you should override the toString() method so that it returns a descriptive string representation of the object.

```
class Car{
        public String toString() {
                return "Car [name=" + name + ", width=" + width + ", height=" + height + "]";
        }
        private String name;
        private double width, height;
}
```

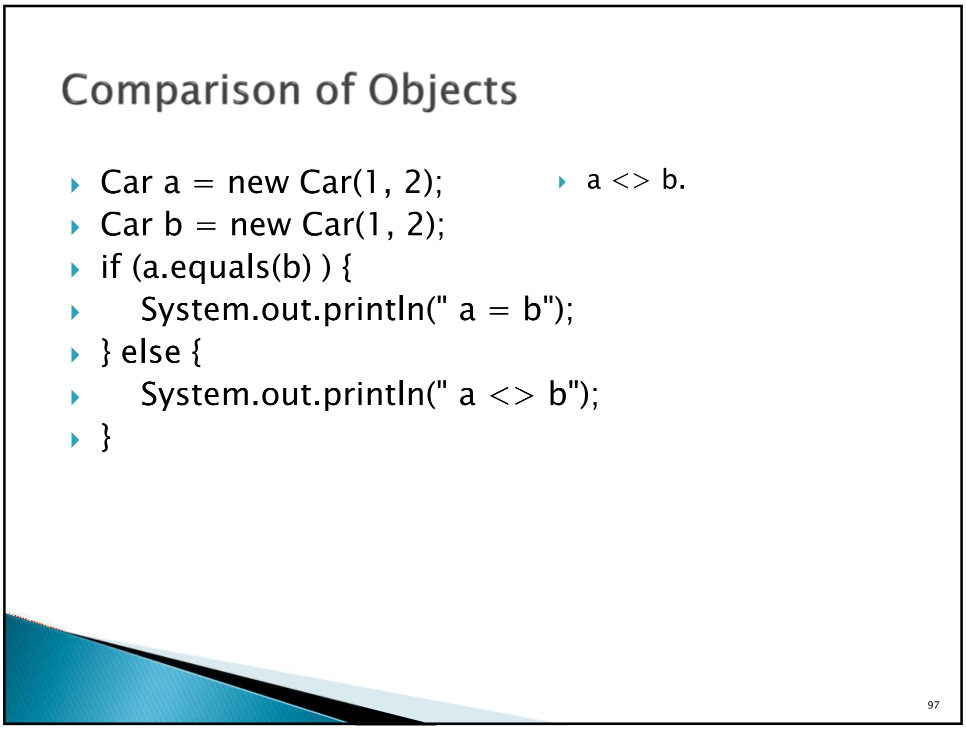
95



equals()

- The default implementation of the equals() method in the Object class is:
  - public boolean equals(Object obj){
  -     return (this == obj)
  - }
- This particular comparison is also known as “shallow comparison.”
- it is really intended for the subclasses of the Object class to modify the equals method to test whether two distinct objects have the same states which is known as “deep comparison.”

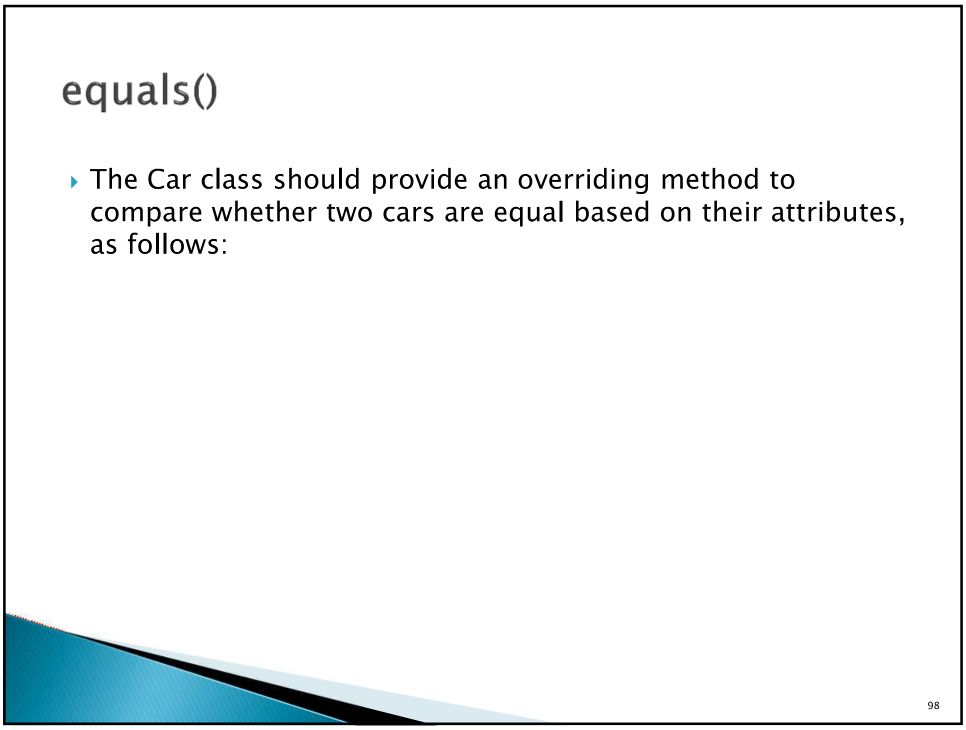
96



Comparison of Objects

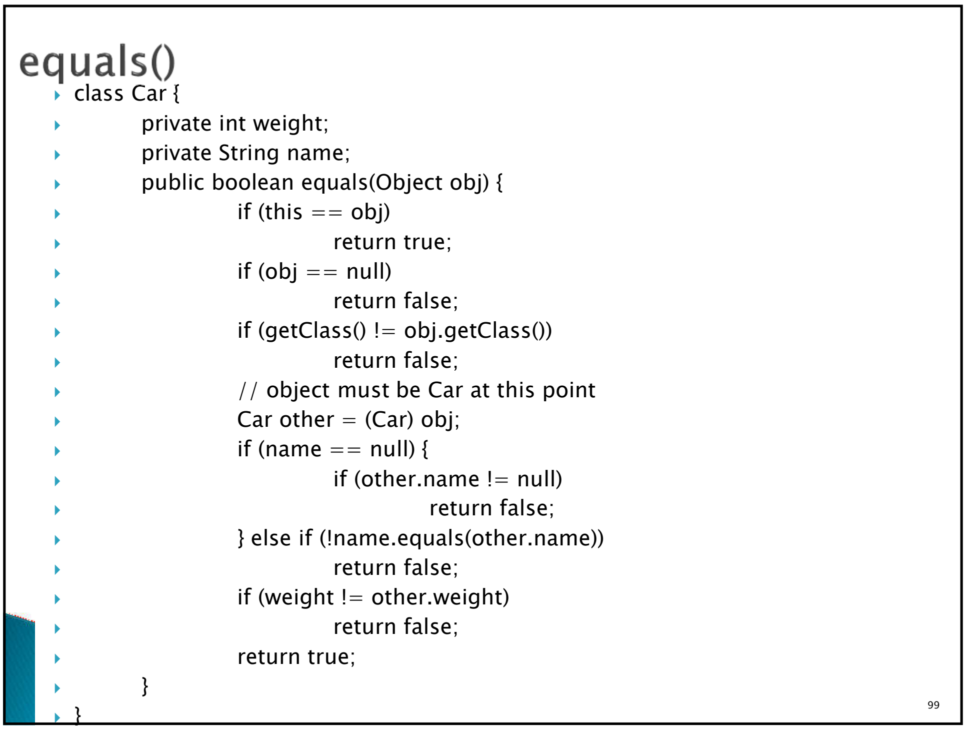
```
Car a = new Car(1, 2);
Car b = new Car(1, 2);
if (a.equals(b) ) {
    System.out.println(" a = b");
} else {
    System.out.println(" a <> b");
}
```

- a <> b.



equals()

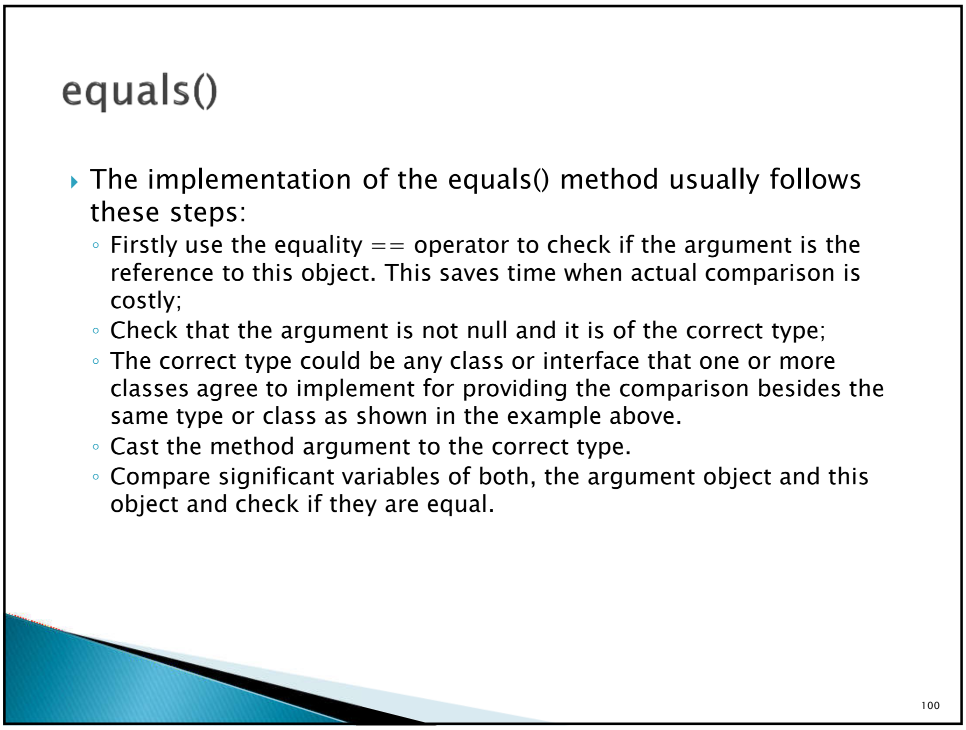
- The Car class should provide an overriding method to compare whether two cars are equal based on their attributes, as follows:



equals()

```
class Car {
        private int weight;
        private String name;
        public boolean equals(Object obj) {
                if (this == obj)
                        return true;
                if (obj == null)
                        return false;
                if (getClass() != obj.getClass())
                        return false;
                // object must be Car at this point
                Car other = (Car) obj;
                if (name == null) {
                        if (other.name != null)
                                return false;
                } else if (!name.equals(other.name))
                        return false;
                if (weight != other.weight)
                        return false;
                return true;
        }
}
```

99



equals()

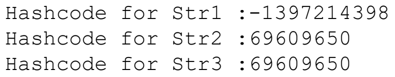
- The implementation of the equals() method usually follows these steps:
  - Firstly use the equality == operator to check if the argument is the reference to this object. This saves time when actual comparison is costly;
  - Check that the argument is not null and it is of the correct type;
  - The correct type could be any class or interface that one or more classes agree to implement for providing the comparison besides the same type or class as shown in the example above.
  - Cast the method argument to the correct type.
  - Compare significant variables of both, the argument object and this object and check if they are equal.

100

hashCode()

- Invoking hashCode() on an object returns the object's hash code.
- The hash code is an integer that can be used to store the object in a hash set so that it can be located quickly.
- The hashCode() implemented in the Object class returns the internal memory address of the object in hexadecimal.

```
public class Test {

  public static void main(String args[]) {
    String Str1 = new String("Welcome");
    String Str2 = new String("Hello");
    String Str3 = new String("Hello");
    System.out.println("Hashcode for Str1 :" + Str1.hashCode() );
    System.out.println("Hashcode for Str2 :" + Str2.hashCode() );
    System.out.println("Hashcode for Str3 :" + Str3.hashCode() );
  }
}
```

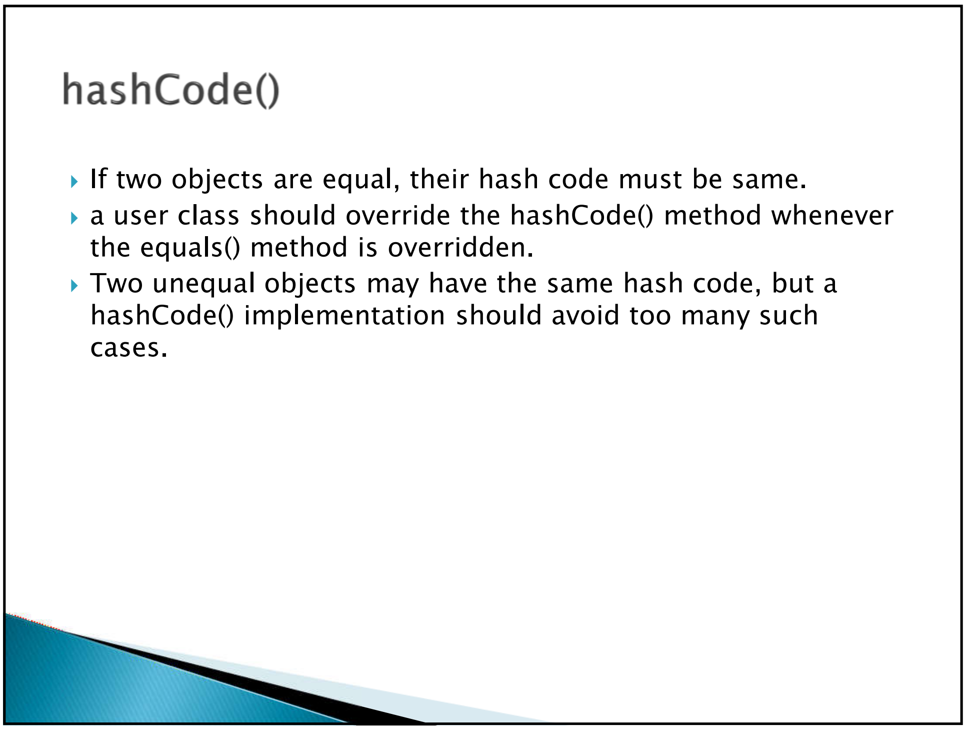


```
Hashcode for Str1 :-1397214398
Hashcode for Str2 :69609650
Hashcode for Str3 :69609650
```

hashCode()

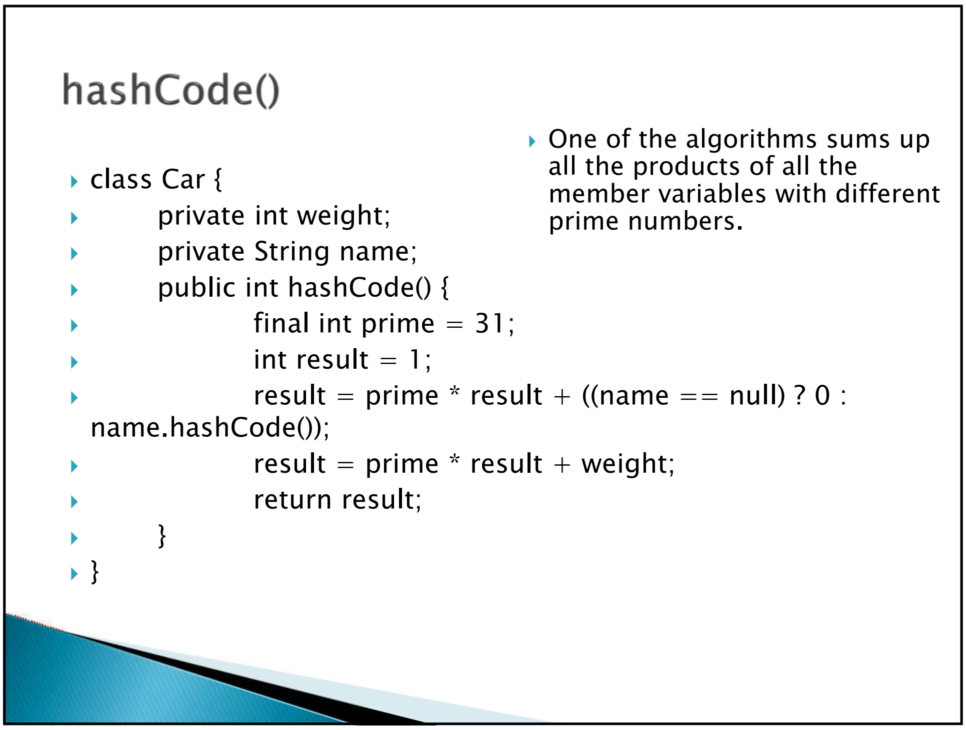
- The hash code for a String object is computed as

$$s_0 \times 31^{n-1} + s_1 \times 31^{n-2} + \cdots + s_{n-1}$$



hashCode()

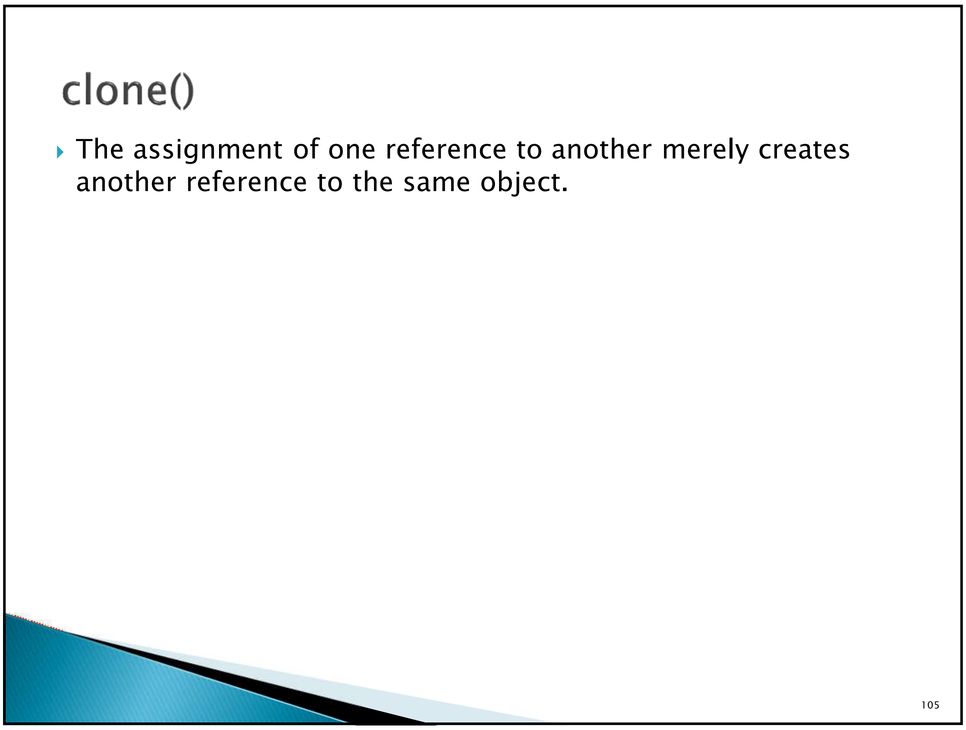
- If two objects are equal, their hash code must be same.
- a user class should override the hashCode() method whenever the equals() method is overridden.
- Two unequal objects may have the same hash code, but a hashCode() implementation should avoid too many such cases.



hashCode()

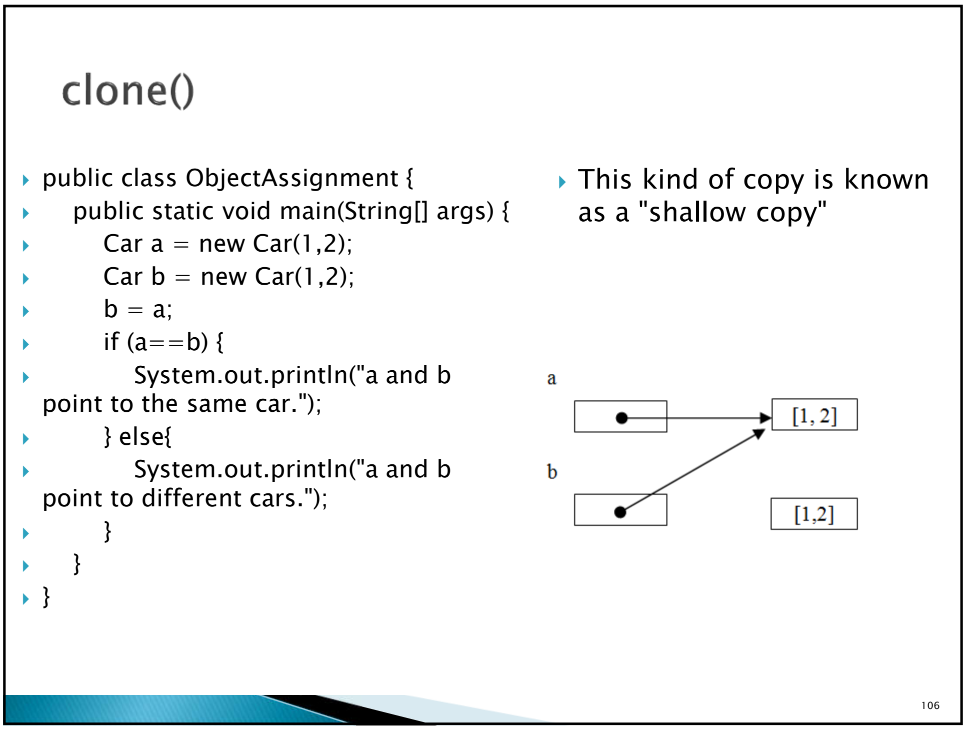
```
class Car {
      private int weight;
      private String name;
      public int hashCode() {
              final int prime = 31;
              int result = 1;
              result = prime * result + ((name == null) ? 0 : name.hashCode());
              result = prime * result + weight;
              return result;
      }
}
```

- One of the algorithms sums up all the products of all the member variables with different prime numbers.



clone()

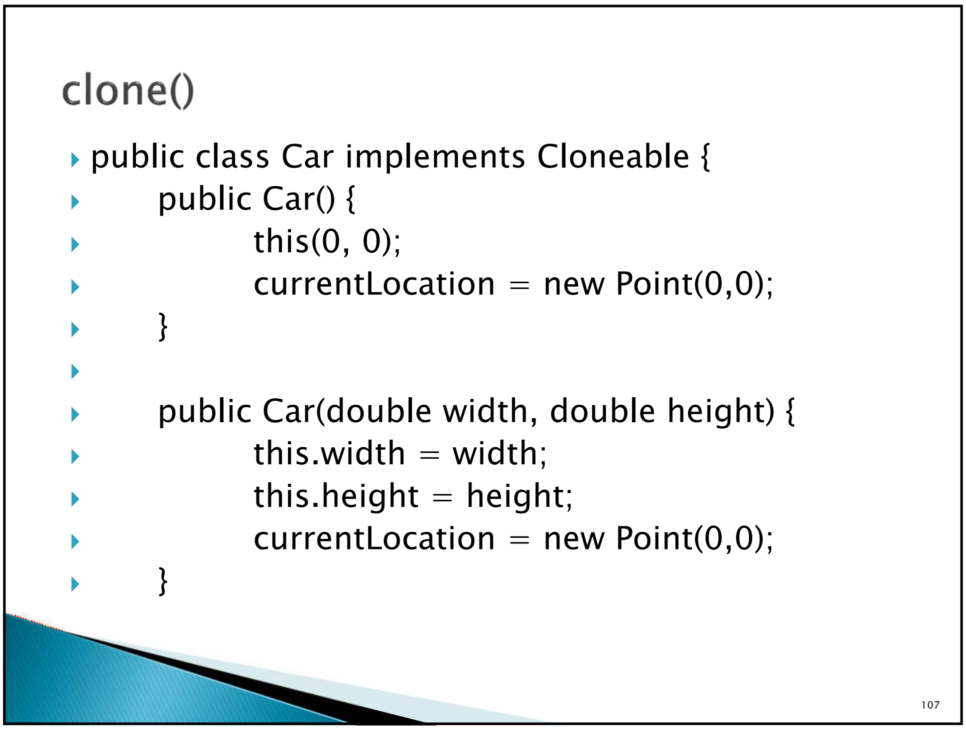
- The assignment of one reference to another merely creates another reference to the same object.



clone()

```
public class ObjectAssignment {
    public static void main(String[] args) {
        Car a = new Car(1,2);
        Car b = new Car(1,2);
        b = a;
        if (a==b) {
            System.out.println("a and b
point to the same car.");
        } else{
            System.out.println("a and b
point to different cars.");
        }
    }
}
```

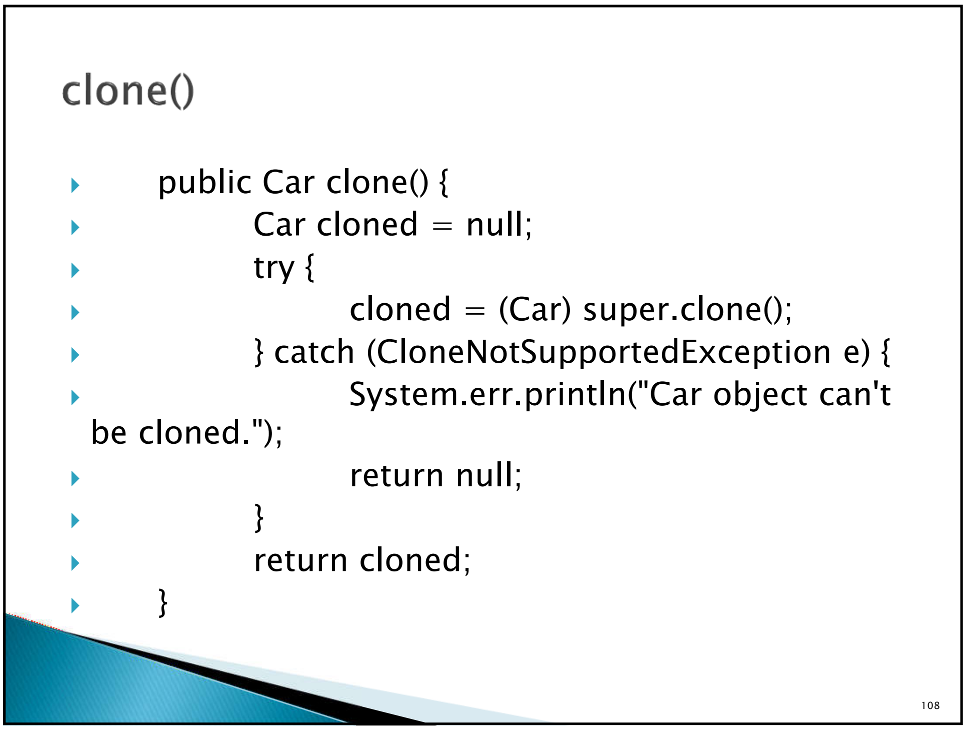
- This kind of copy is known as a "shallow copy"



clone()

```
public class Car implements Cloneable {
    public Car() {
        this(0, 0);
        currentLocation = new Point(0,0);
    }

    public Car(double width, double height) {
        this.width = width;
        this.height = height;
        currentLocation = new Point(0,0);
    }
```

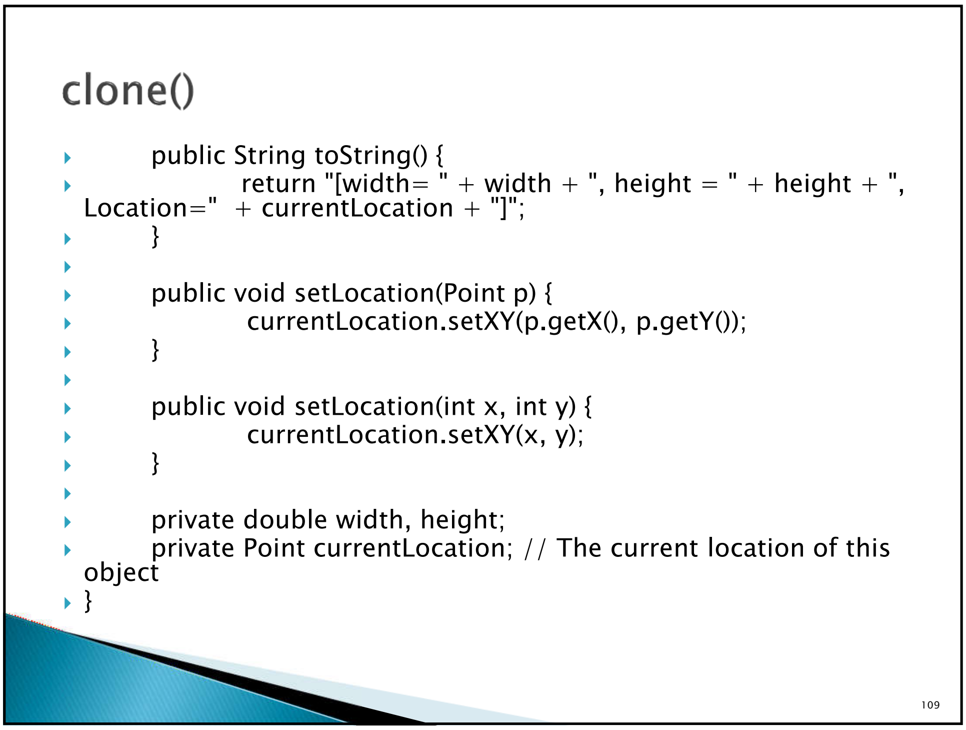
107



clone()

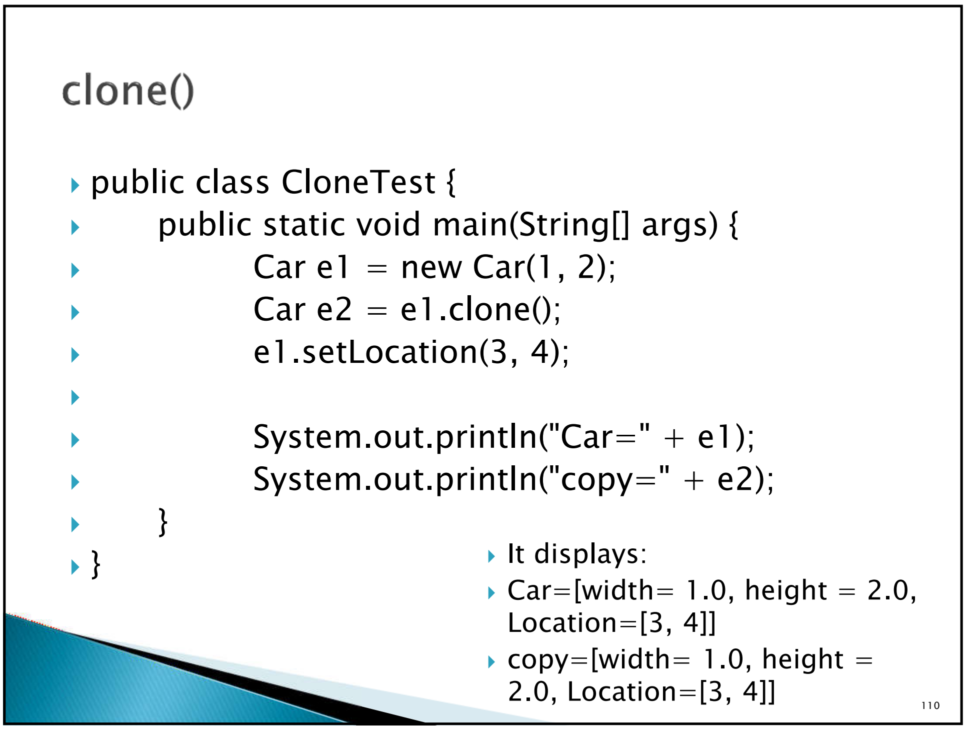
```
    public Car clone() {
        Car cloned = null;
        try {
            cloned = (Car) super.clone();
        } catch (CloneNotSupportedException e) {
            System.err.println("Car object can't
be cloned.");
            return null;
        }
        return cloned;
    }
```

108



clone()

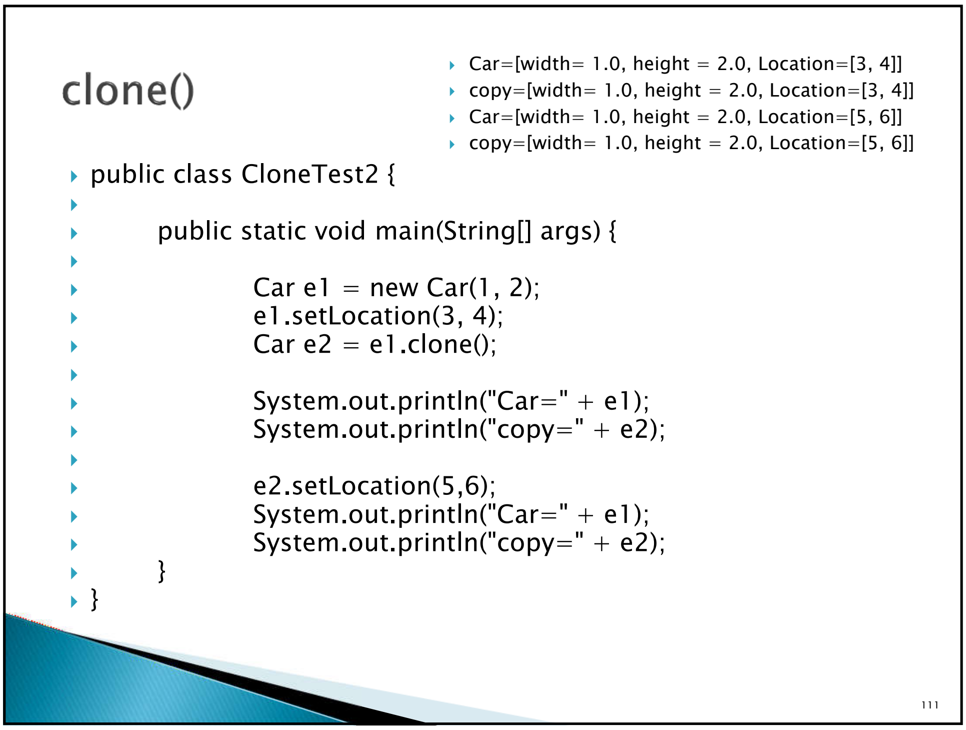
```
        public String toString() {
                return "[width= " + width + ", height = " + height + ",
Location=" + currentLocation + "]";
        }

        public void setLocation(Point p) {
                currentLocation.setXY(p.getX(), p.getY());
        }

        public void setLocation(int x, int y) {
                currentLocation.setXY(x, y);
        }

        private double width, height;
        private Point currentLocation; // The current location of this
object
}
```



clone()

```
public class CloneTest {
        public static void main(String[] args) {
                Car e1 = new Car(1, 2);
                Car e2 = e1.clone();
                e1.setLocation(3, 4);

                System.out.println("Car=" + e1);
                System.out.println("copy=" + e2);
        }
}
```

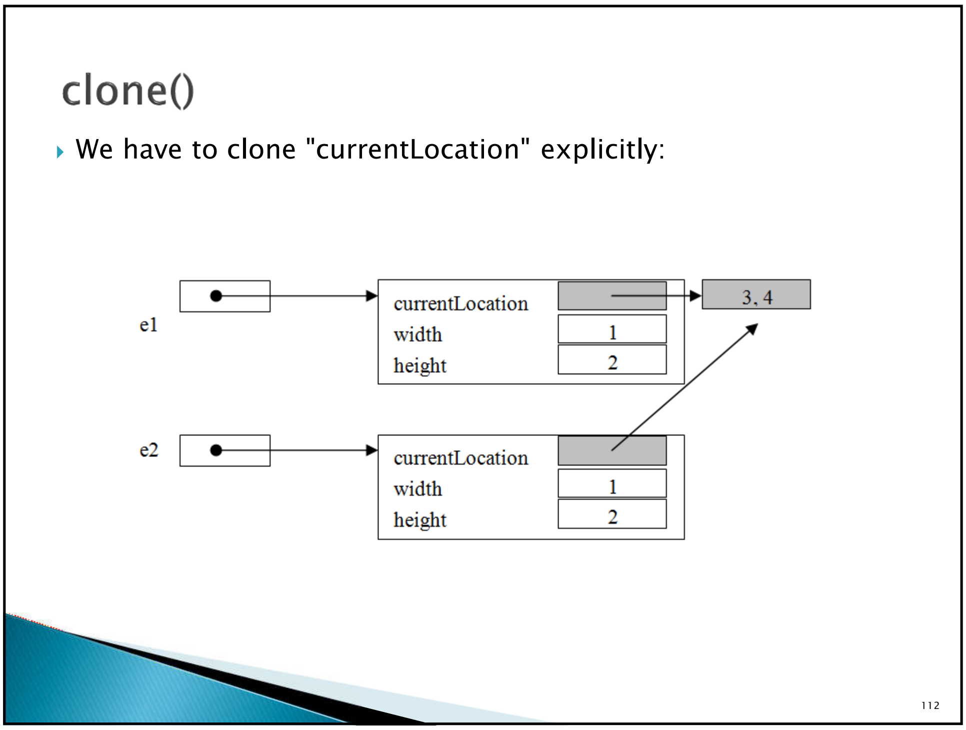
- It displays:
- Car=[width= 1.0, height = 2.0, Location=[3, 4]]
- copy=[width= 1.0, height = 2.0, Location=[3, 4]]



clone()

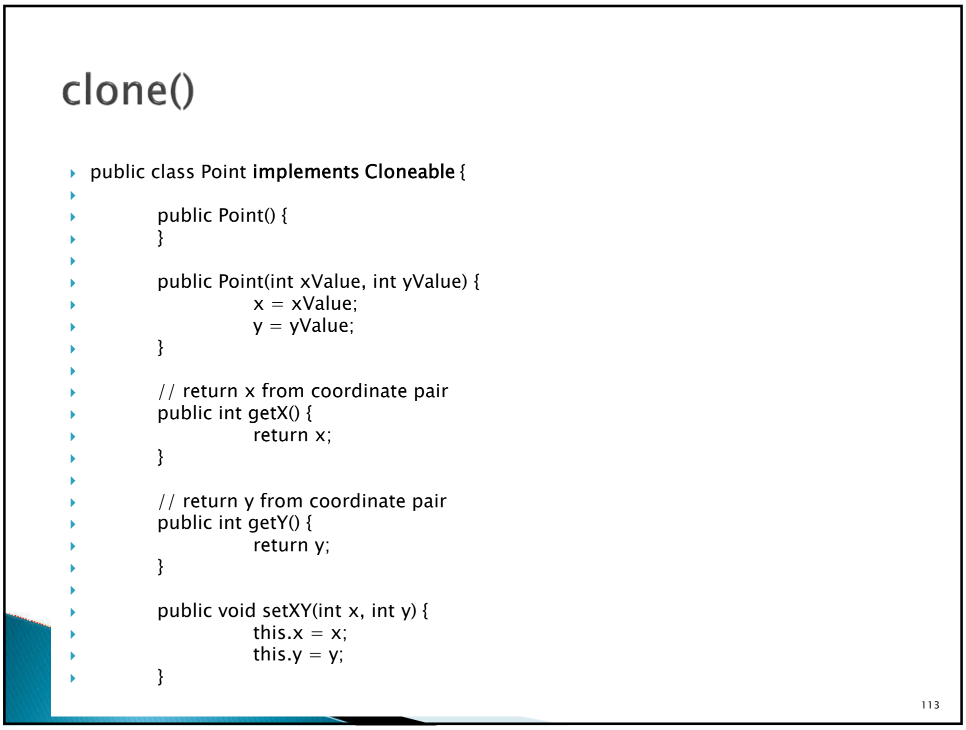
```
Car=[width= 1.0, height = 2.0, Location=[3, 4]]
copy=[width= 1.0, height = 2.0, Location=[3, 4]]
Car=[width= 1.0, height = 2.0, Location=[5, 6]]
copy=[width= 1.0, height = 2.0, Location=[5, 6]]
```

```
public class CloneTest2 {

	public static void main(String[] args) {

		Car e1 = new Car(1, 2);
		e1.setLocation(3, 4);
		Car e2 = e1.clone();

		System.out.println("Car=" + e1);
		System.out.println("copy=" + e2);

		e2.setLocation(5,6);
		System.out.println("Car=" + e1);
		System.out.println("copy=" + e2);
	}
}
```



clone()

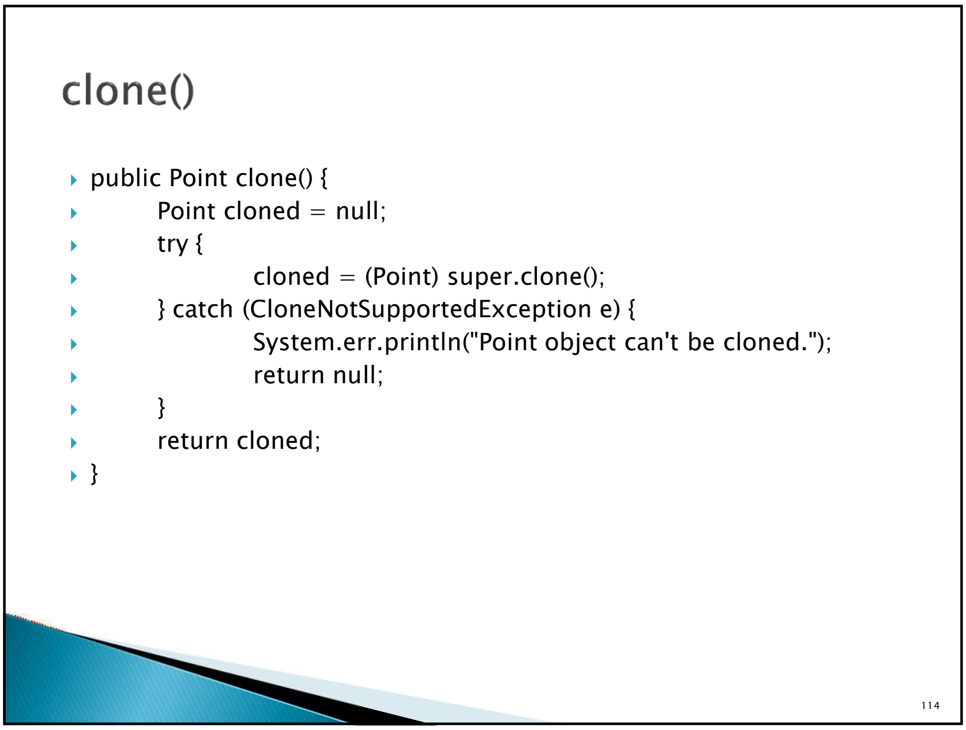
- We have to clone "currentLocation" explicitly:



clone()

```
public class Point implements Cloneable {

        public Point() {
        }

        public Point(int xValue, int yValue) {
                x = xValue;
                y = yValue;
        }

        // return x from coordinate pair
        public int getX() {
                return x;
        }

        // return y from coordinate pair
        public int getY() {
                return y;
        }

        public void setXY(int x, int y) {
                this.x = x;
                this.y = y;
        }
```

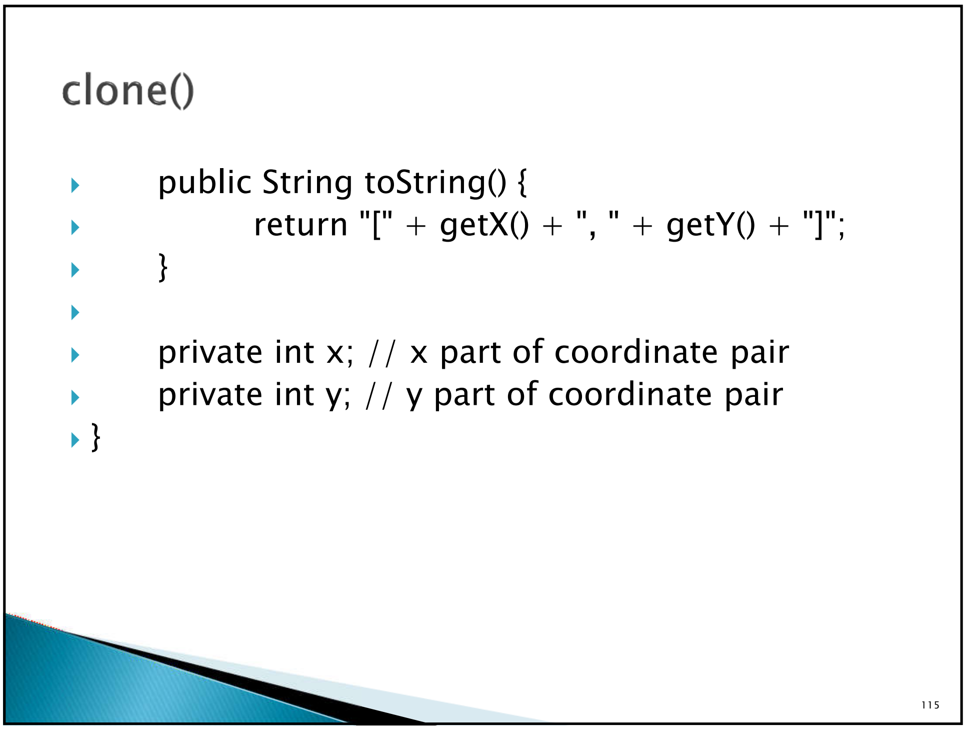
113



clone()

```
public Point clone() {
        Point cloned = null;
        try {
                cloned = (Point) super.clone();
        } catch (CloneNotSupportedException e) {
                System.err.println("Point object can't be cloned.");
                return null;
        }
        return cloned;
}
```

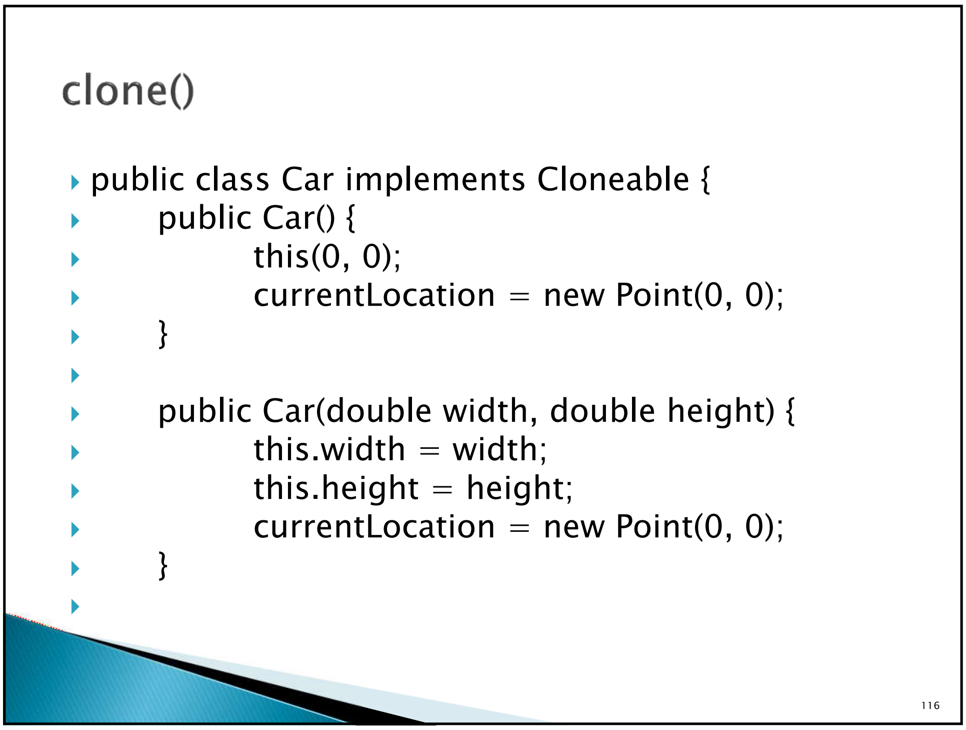
114



clone()

```
    public String toString() {
        return "[" + getX() + ", " + getY() + "]";
    }

    private int x; // x part of coordinate pair
    private int y; // y part of coordinate pair
}
```

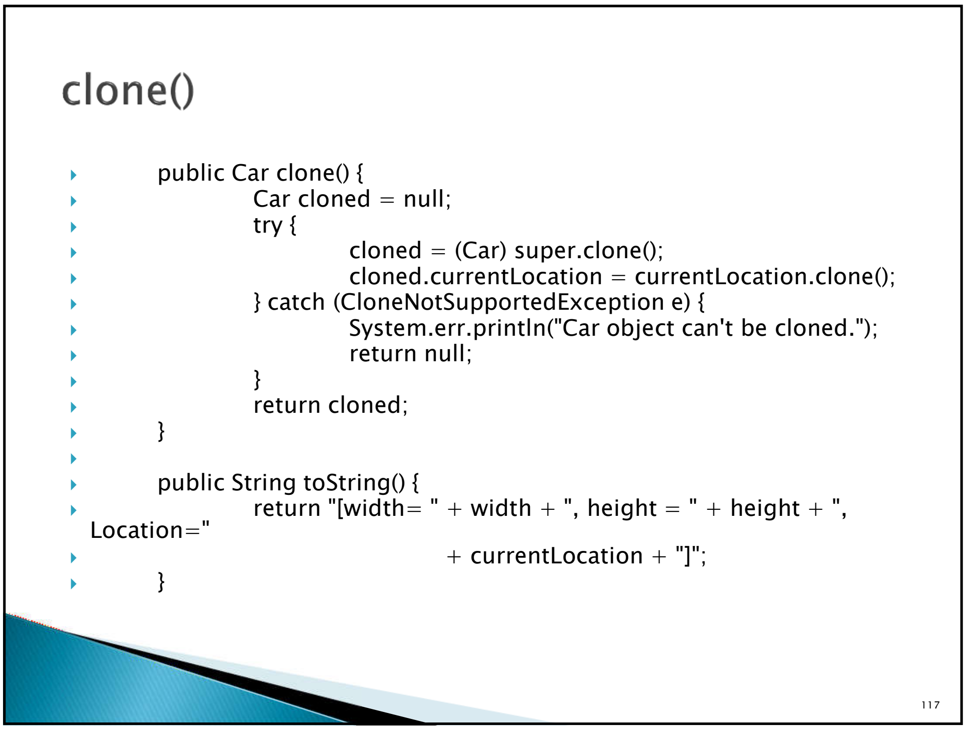
115



clone()

```
public class Car implements Cloneable {
    public Car() {
        this(0, 0);
        currentLocation = new Point(0, 0);
    }

    public Car(double width, double height) {
        this.width = width;
        this.height = height;
        currentLocation = new Point(0, 0);
    }

```

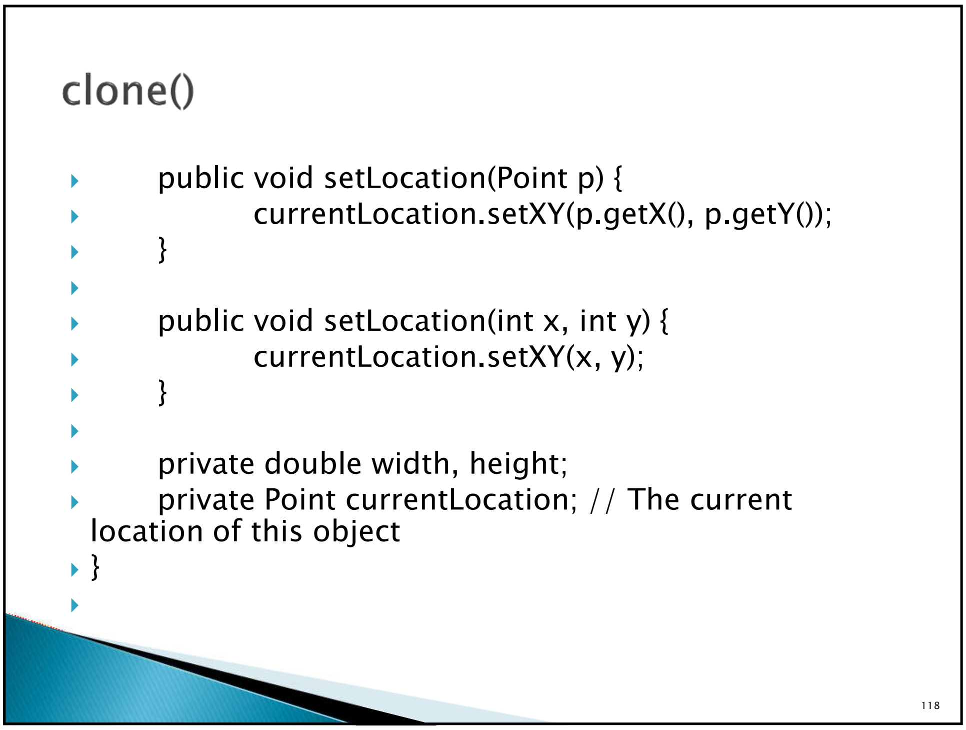
116



clone()

```
        public Car clone() {
                Car cloned = null;
                try {
                        cloned = (Car) super.clone();
                        cloned.currentLocation = currentLocation.clone();
                } catch (CloneNotSupportedException e) {
                        System.err.println("Car object can't be cloned.");
                        return null;
                }
                return cloned;
        }

        public String toString() {
                return "[width= " + width + ", height = " + height + ",
Location="
                                + currentLocation + "]";
        }
```

117



clone()

```
        public void setLocation(Point p) {
                currentLocation.setXY(p.getX(), p.getY());
        }

        public void setLocation(int x, int y) {
                currentLocation.setXY(x, y);
        }

        private double width, height;
        private Point currentLocation; // The current
location of this object
}
```

118

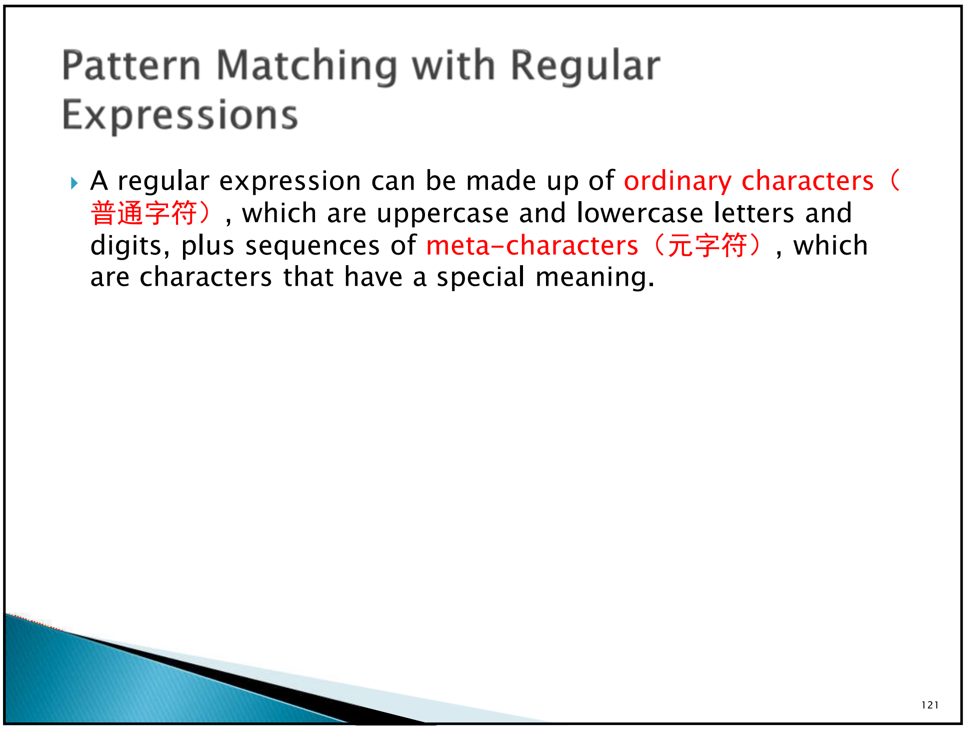
clone()

- Now if we run CloneTest2.java again, it displays:
- Car=[width= 1.0, height = 2.0, Location=[3, 4]]
- copy=[width= 1.0, height = 2.0, Location=[3, 4]]
- Car=[width= 1.0, height = 2.0, Location=[3, 4]]
- copy=[width= 1.0, height = 2.0, Location=[5, 6]]

Pattern Matching with Regular Expressions

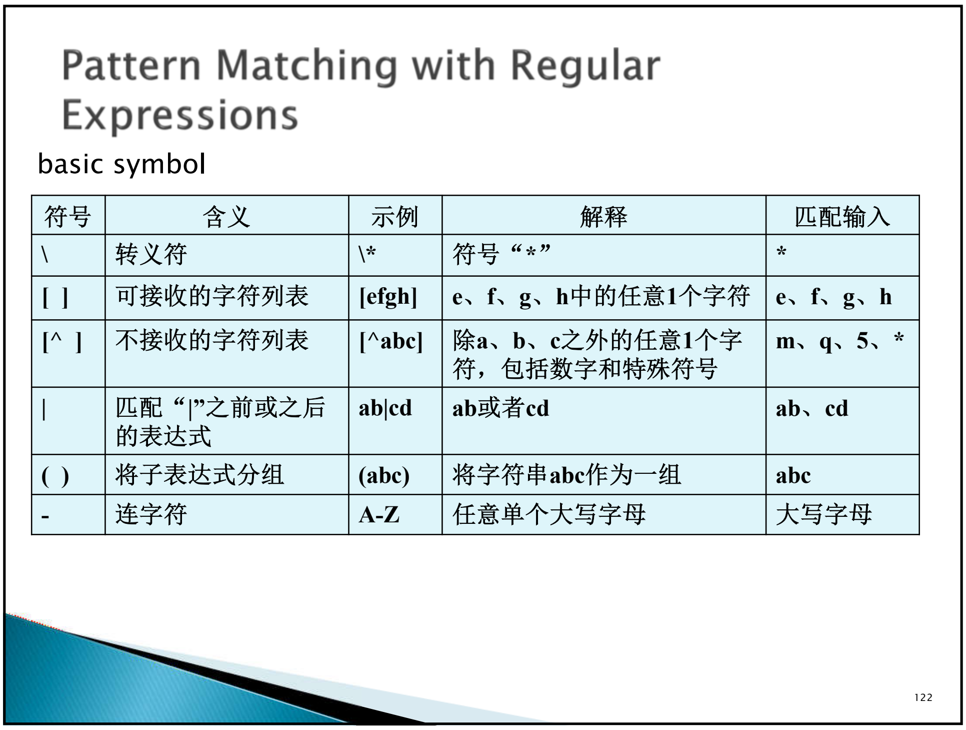
- A regular expression is a string that describes a pattern that is to be used to search for matches within some other string.
- For example, if you try to find out whether the input string "Donald is one of the top coders" contains "the", or "top", or "coder", you can use the regular expression "the|top|coder" to match the input string.
- The match results are:
  - Found the text "the" starting at index 17 and ending at index 20.
  - Found the text "top" starting at index 21 and ending at index 24.
  - Found the text "coder" starting at index 24 and ending at index 29.

/^\d{6}(18|19|20)?\d{2}(0[1-9]|1[012])(0[1-9]|[12]\d|3[01])\d{3}(\d|[xX])$/



Pattern Matching with Regular Expressions

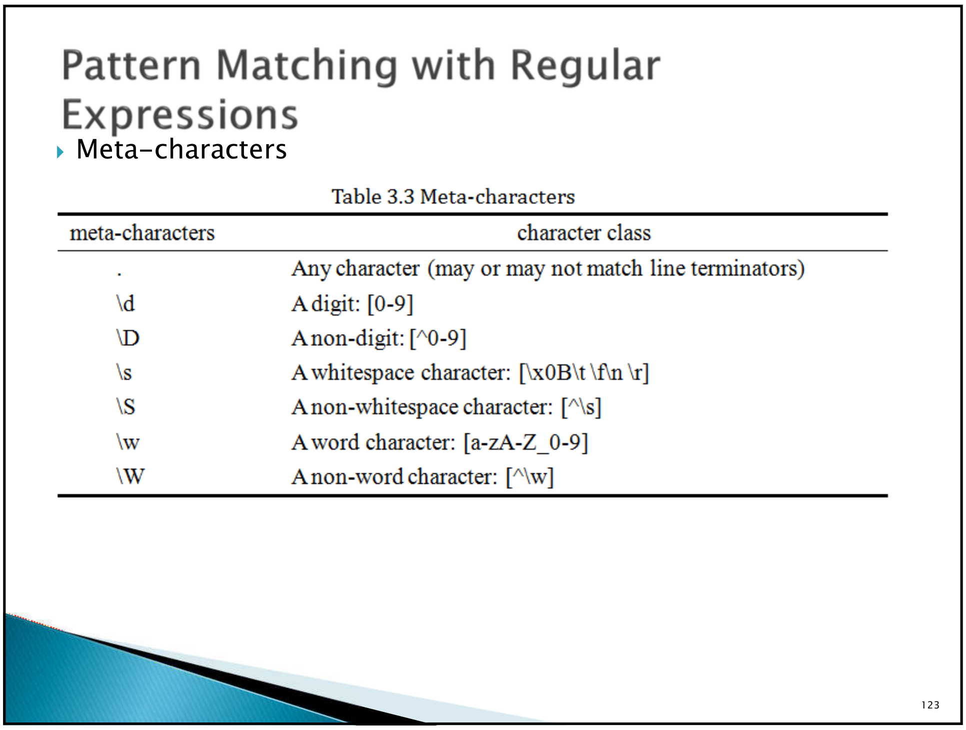
- A regular expression can be made up of ordinary characters（普通字符）, which are uppercase and lowercase letters and digits, plus sequences of meta-characters（元字符）, which are characters that have a special meaning.



Pattern Matching with Regular Expressions

basic symbol

符号	含义	示例	解释	匹配输入
\	转义符	*	符号 “*”	*
[ ]	可接收的字符列表	[efgh]	e、f、g、h中的任意1个字符	e、f、g、h
[^]	不接收的字符列表	[^abc]	除a、b、c之外的任意1个字符，包括数字和特殊符号	m、q、5、*
|	匹配 “|”之前或之后的表达式	ab|cd	ab或者cd	ab、cd
( )	将子表达式分组	(abc)	将字符串abc作为一组	abc
-	连字符	A-Z	任意单个大写字母	大写字母

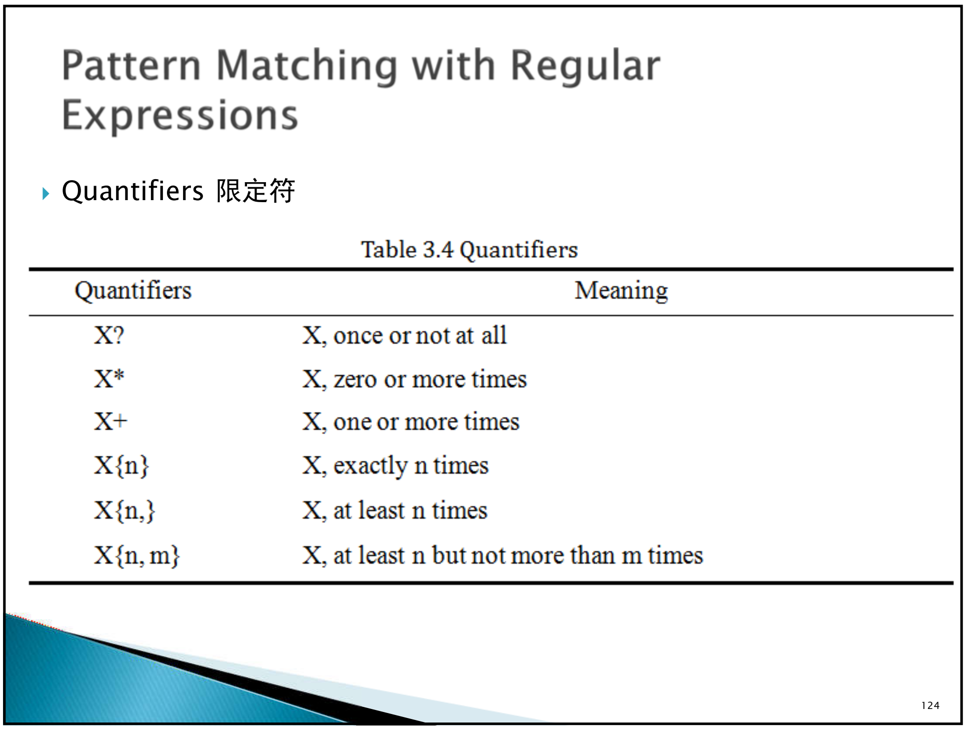


Pattern Matching with Regular Expressions

- Meta-characters

Table 3.3 Meta-characters

meta-characters	character class
.	Any character (may or may not match line terminators)
\d	A digit: [0-9]
\D	A non-digit: [^0-9]
\s	A whitespace character: [\x0B\t \f\n \r]
\S	A non-whitespace character: [^\s]
\w	A word character: [a-zA-Z_0-9]
\W	A non-word character: [^\w]

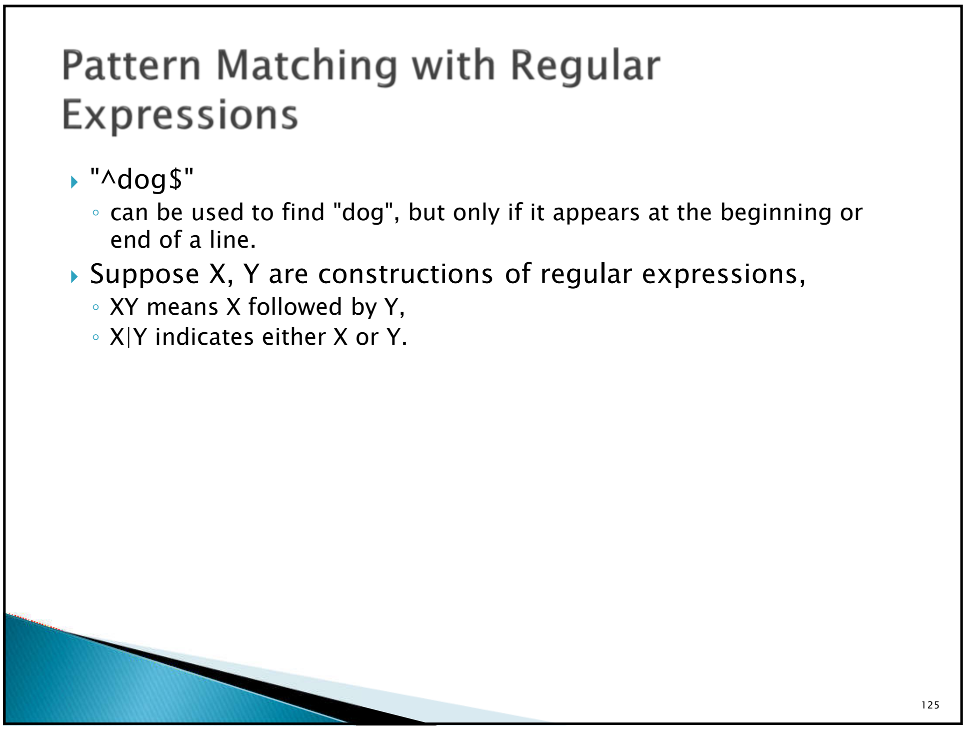


Pattern Matching with Regular Expressions

- Quantifiers 限定符

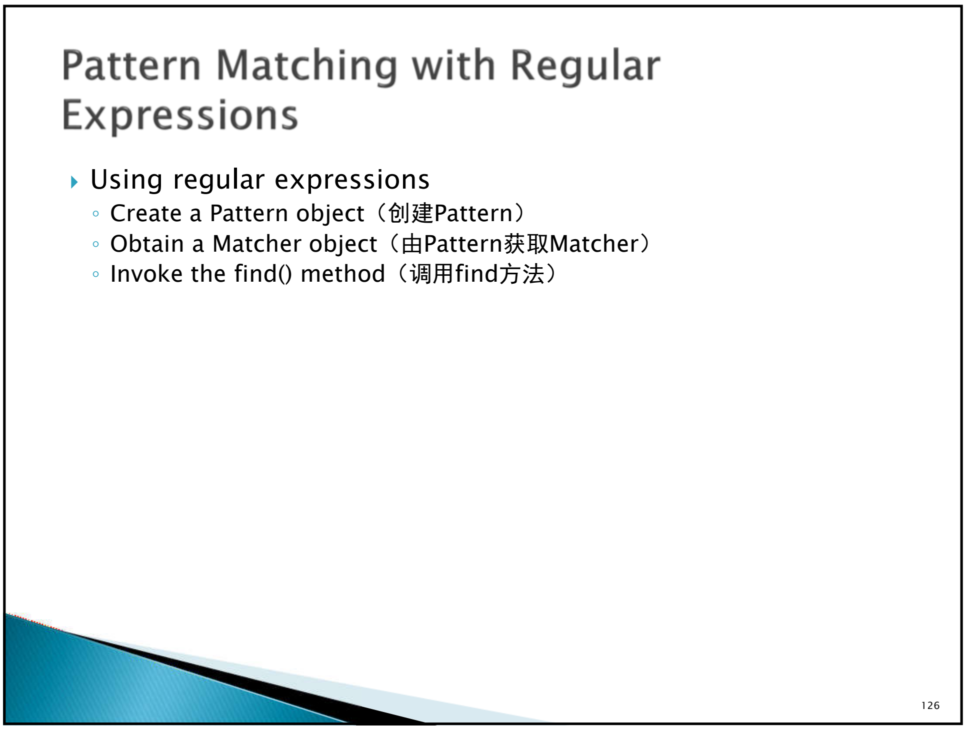
Table 3.4 Quantifiers

Quantifiers	Meaning
X?	X, once or not at all
X*	X, zero or more times
X+	X, one or more times
X{n}	X, exactly n times
X{n,}	X, at least n times
X{n, m}	X, at least n but not more than m times



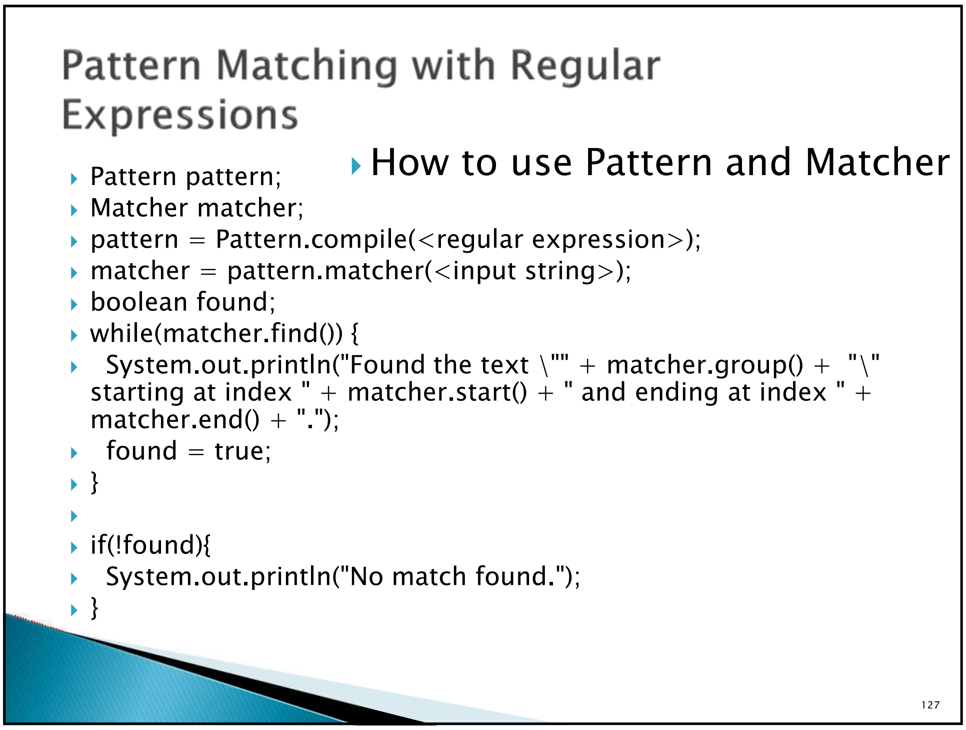
Pattern Matching with Regular Expressions

- "^dog$"
  - can be used to find "dog", but only if it appears at the beginning or end of a line.
- Suppose X, Y are constructions of regular expressions,
  - XY means X followed by Y,
  - X|Y indicates either X or Y.



Pattern Matching with Regular Expressions

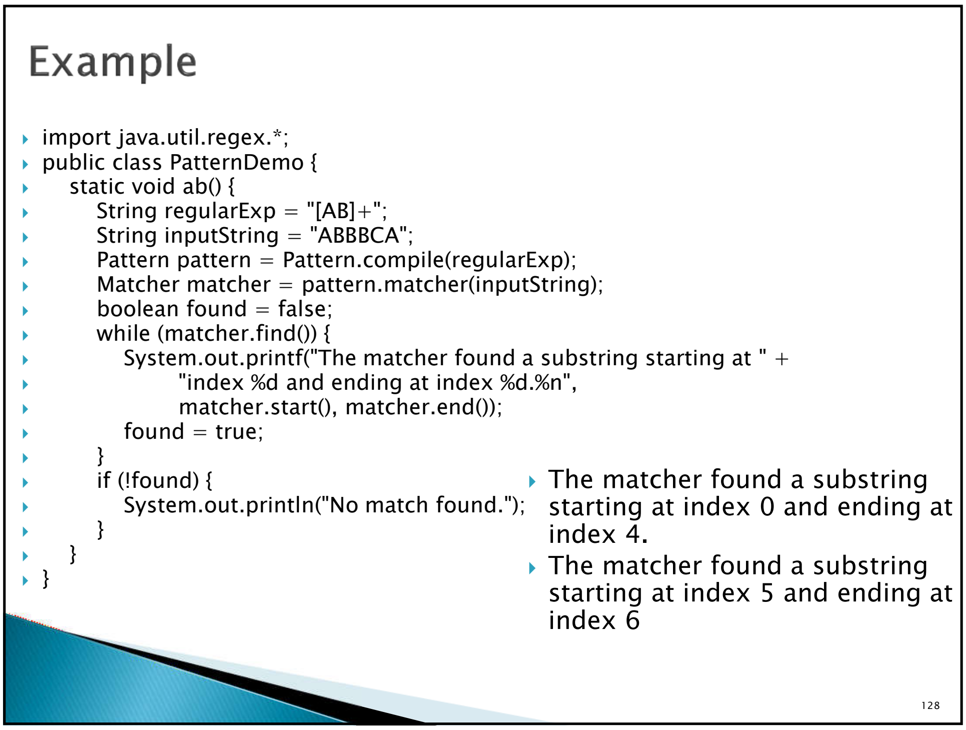
- Using regular expressions
  - Create a Pattern object（创建Pattern）
  - Obtain a Matcher object（由Pattern获取Matcher）
  - Invoke the find() method（调用find方法）



Pattern Matching with Regular Expressions

- How to use Pattern and Matcher

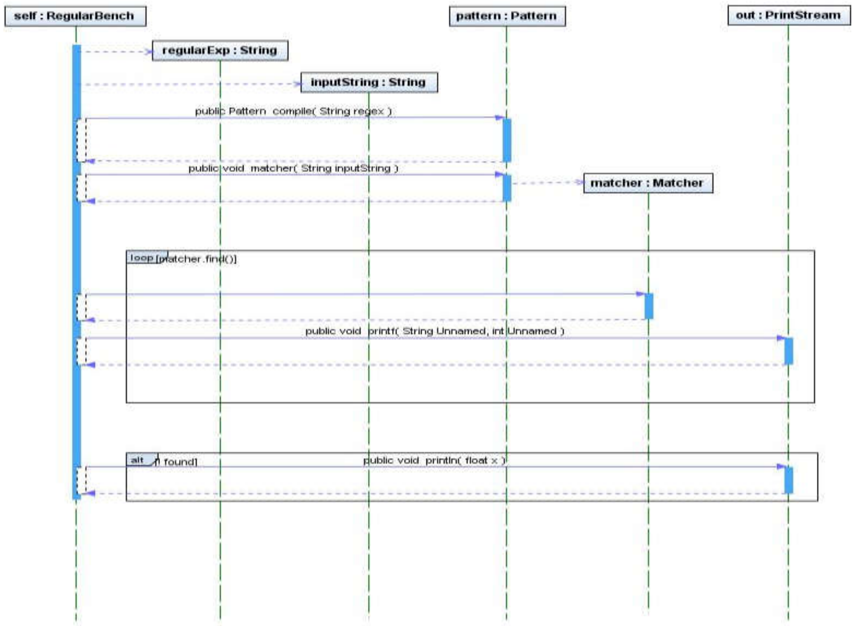
```
Pattern pattern;
Matcher matcher;
pattern = Pattern.compile(<regular expression>);
matcher = pattern.matcher(<input string>);
boolean found;
while(matcher.find()) {
  System.out.println("Found the text \"" + matcher.group() + "\" starting at index " + matcher.start() + " and ending at index " + matcher.end() + ".");
  found = true;
}

if(!found){
  System.out.println("No match found.");
}
```



Example

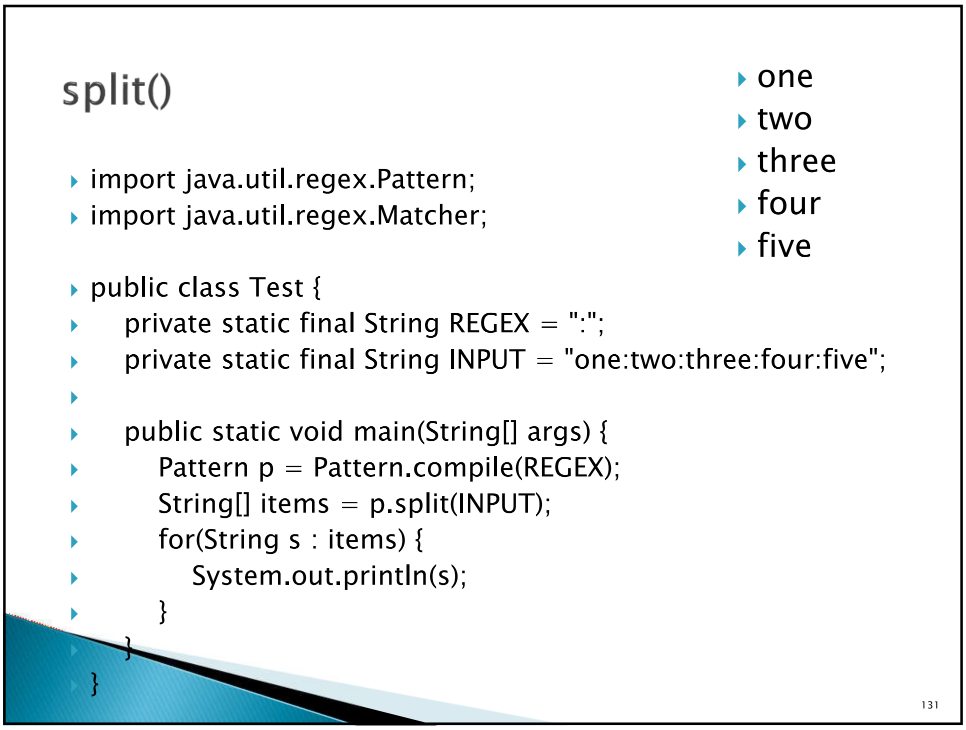
```
import java.util.regex.*;
public class PatternDemo {
    static void ab() {
        String regularExp = "[AB]+";
        String inputString = "ABBBCA";
        Pattern pattern = Pattern.compile(regularExp);
        Matcher matcher = pattern.matcher(inputString);
        boolean found = false;
        while (matcher.find()) {
            System.out.printf("The matcher found a substring starting at " +
                    "index %d and ending at index %d.%n",
                    matcher.start(), matcher.end());
            found = true;
        }
        if (!found) {
            System.out.println("No match found.");
        }
    }
}
```

- The matcher found a substring starting at index 0 and ending at index 4.
- The matcher found a substring starting at index 5 and ending at index 6


self : RegularBench
pattern : Pattern
out : PrintStream
regularExp : String
inputString : String
public Pattern compile(String regex)
public void matcher(String inputString)
matcher : Matcher
loop [matcher.find()]
public void printf(String Unnamed, int Unnamed)
alt [! found]
public void println(float x)

Pattern Matching with Regular Expressions

- The Pattern class defines a convenient matches method that allows you to quickly check if a pattern is present in a given input string.
- For example, Pattern.matches("\\d","1"); returns true, because the digit "1" matches the regular expression \d.

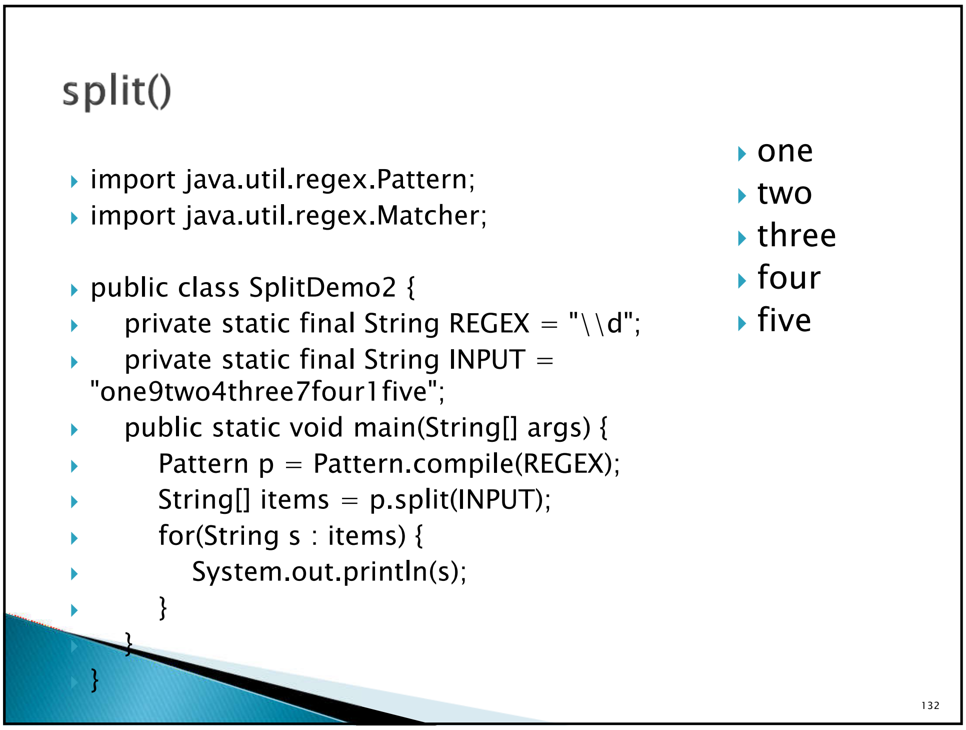


split()

- one
- two
- three
- four
- five

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class Test {
    private static final String REGEX = ":";
    private static final String INPUT = "one:two:three:four:five";

    public static void main(String[] args) {
        Pattern p = Pattern.compile(REGEX);
        String[] items = p.split(INPUT);
        for(String s : items) {
            System.out.println(s);
        }
    }
}
```

131

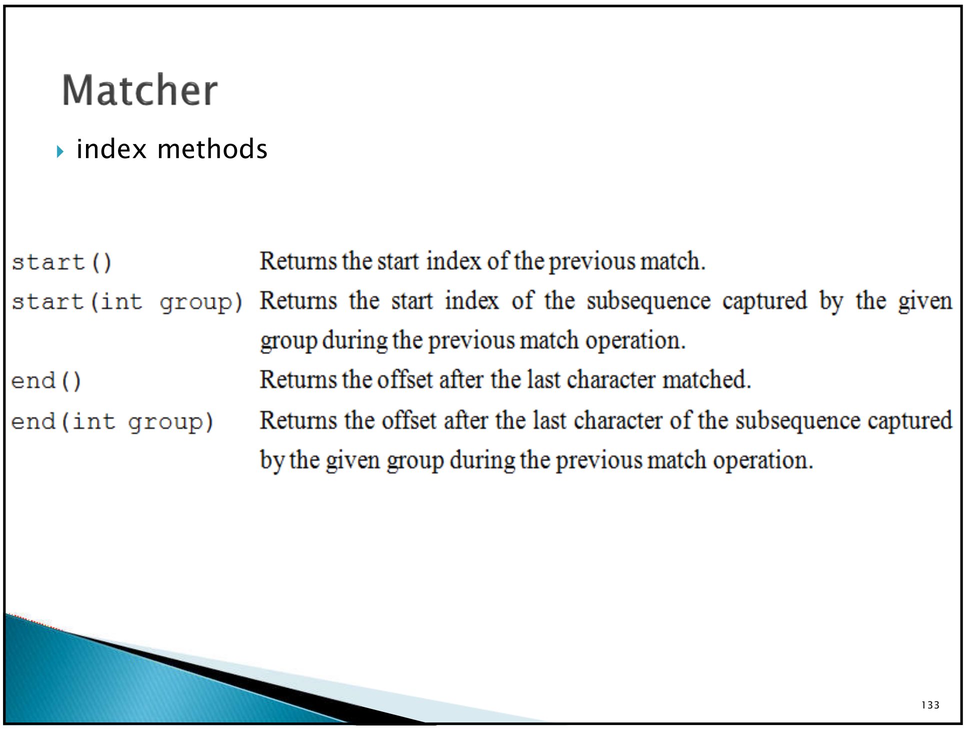


split()

- one
- two
- three
- four
- five

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class SplitDemo2 {
    private static final String REGEX = "\\d";
    private static final String INPUT =
"one9two4three7four1five";
    public static void main(String[] args) {
        Pattern p = Pattern.compile(REGEX);
        String[] items = p.split(INPUT);
        for(String s : items) {
            System.out.println(s);
        }
    }
}
```

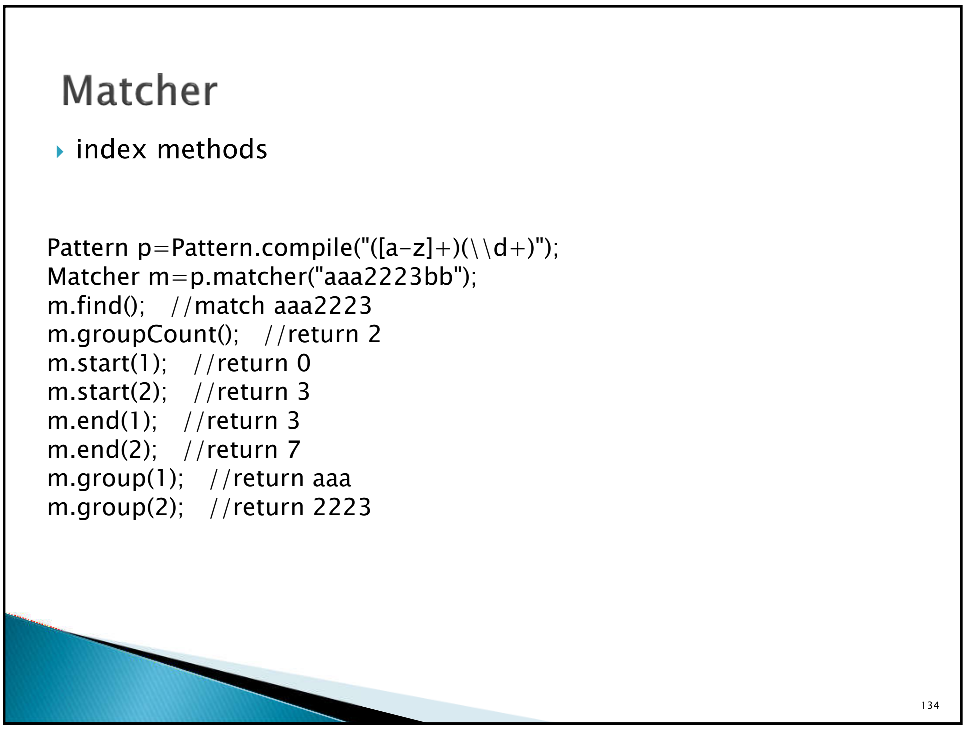
132



Matcher

- index methods

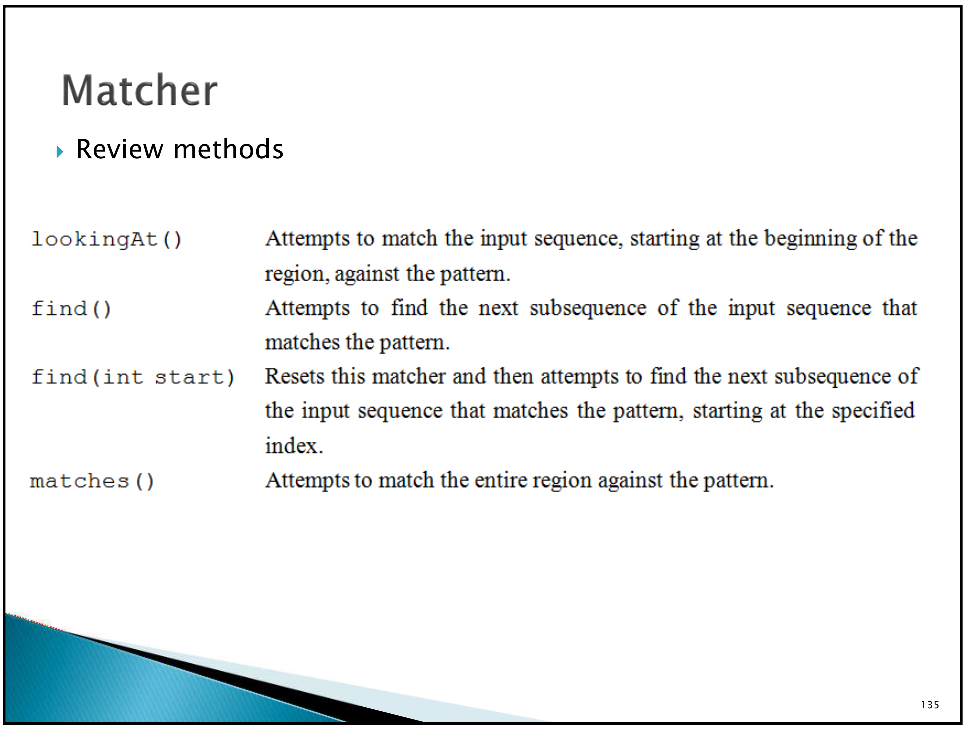
`start()`	Returns the start index of the previous match.
`start(int group)`	Returns the start index of the subsequence captured by the given group during the previous match operation.
`end()`	Returns the offset after the last character matched.
`end(int group)`	Returns the offset after the last character of the subsequence captured by the given group during the previous match operation.



Matcher

- index methods

```
Pattern p=Pattern.compile("([a-z]+)(\\d+)");
Matcher m=p.matcher("aaa2223bb");
m.find();   //match aaa2223
m.groupCount();   //return 2
m.start(1);   //return 0
m.start(2);   //return 3
m.end(1);   //return 3
m.end(2);   //return 7
m.group(1);   //return aaa
m.group(2);   //return 2223
```

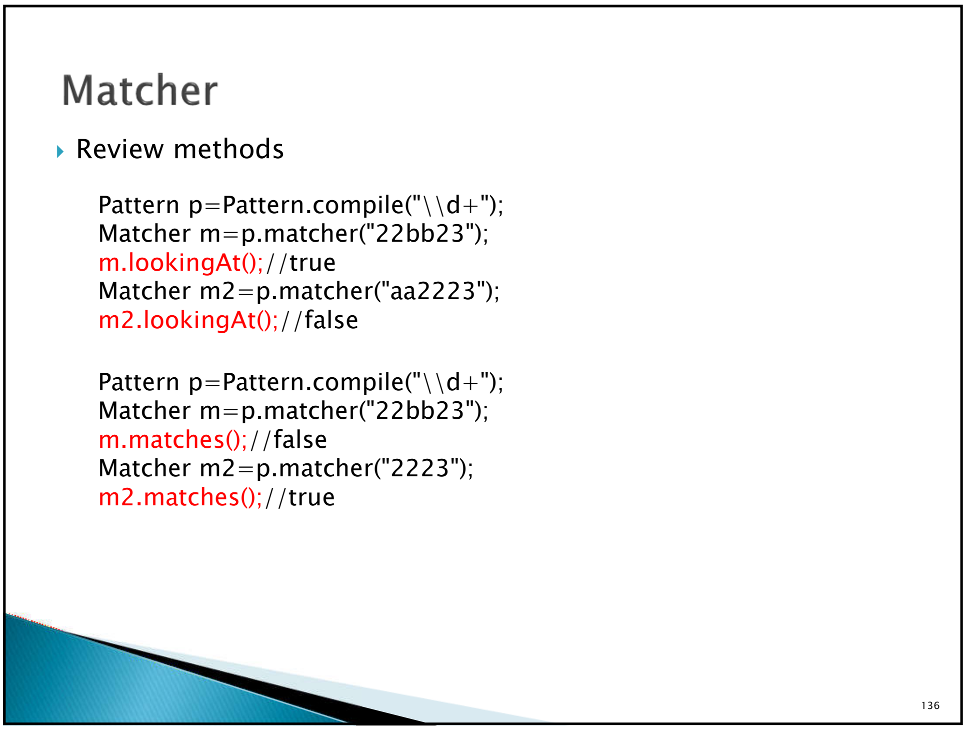


Matcher

- Review methods

`lookingAt()`	Attempts to match the input sequence, starting at the beginning of the region, against the pattern.
`find()`	Attempts to find the next subsequence of the input sequence that matches the pattern.
`find(int start)`	Resets this matcher and then attempts to find the next subsequence of the input sequence that matches the pattern, starting at the specified index.
`matches()`	Attempts to match the entire region against the pattern.

135

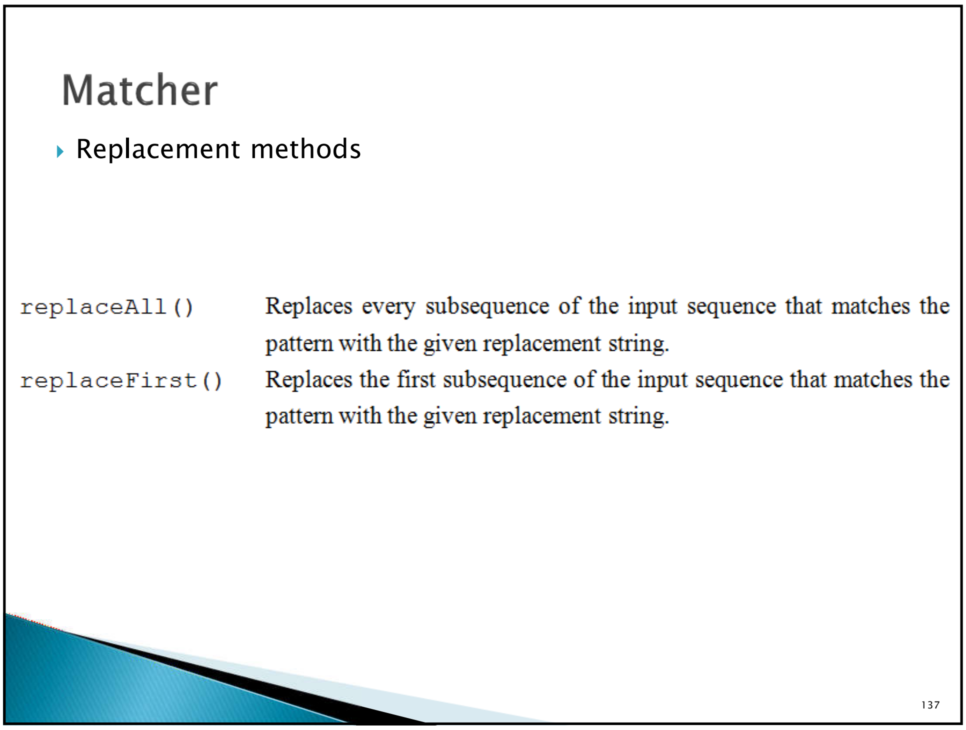


Matcher

- Review methods

```
Pattern p=Pattern.compile("\\d+");
Matcher m=p.matcher("22bb23");
m.lookingAt();//true
Matcher m2=p.matcher("aa2223");
m2.lookingAt();//false

Pattern p=Pattern.compile("\\d+");
Matcher m=p.matcher("22bb23");
m.matches();//false
Matcher m2=p.matcher("2223");
m2.matches();//true
```

136

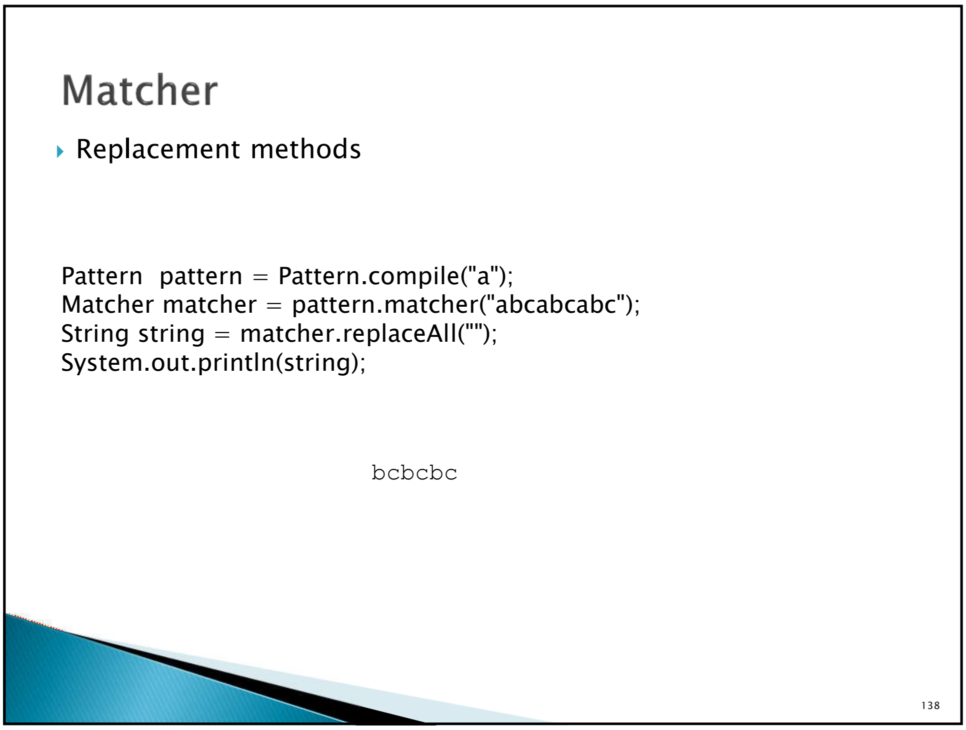


Matcher

- Replacement methods

`replaceAll()`	Replaces every subsequence of the input sequence that matches the pattern with the given replacement string.
`replaceFirst()`	Replaces the first subsequence of the input sequence that matches the pattern with the given replacement string.

137



Matcher

- Replacement methods

```
Pattern  pattern = Pattern.compile("a");
Matcher matcher = pattern.matcher("abcabcabc");
String string = matcher.replaceAll("");
System.out.println(string);
```

```
bcbcbc
```

138

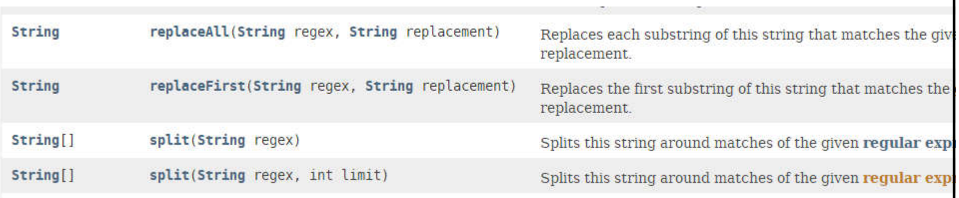
Example

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class MatcherDemo {

  private static final String REGEX = "\\bdog\\b";
  private static final String INPUT = "dog dog dog doggie dogg";

  public static void main(String[] args) {
    Pattern p = Pattern.compile(REGEX);
    Matcher m = p.matcher(INPUT); // get a matcher object
    int count = 0;
    while(m.find()) {
        count++;
        System.out.println("Match number "+count);
        System.out.println("start(): "+m.start());
        System.out.println("end(): "+m.end());
    }
  }
}
```

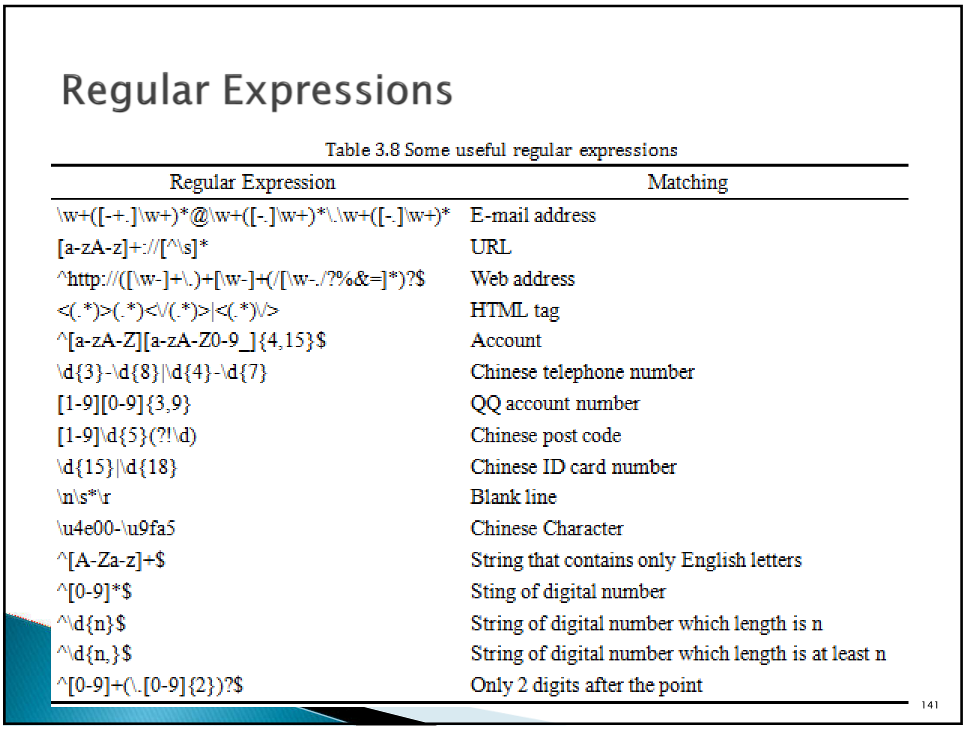
```
Match number 1
start(): 0
end(): 3
Match number 2
start(): 4
end(): 7
Match number 3
start(): 8
end(): 11
```

String Class

`matches()`	Tells whether or not this string matches the given regular expression. An invocation of this method of the form `str.matches(regex)` yields exactly the same result as the expression `Pattern.matches(regex, str)`.
`split()`	Splits this string around matches of the given regular expression.
`replaceFirst()`	This replaces the first substring of this string that matches the given regular expression with the given replacement.
`replaceAll()`	Replaces each substring of this string that matches the given regular expression with the given replacement.



String	replaceAll(String regex, String replacement)	Replaces each substring of this string that matches the giv replacement.
String	replaceFirst(String regex, String replacement)	Replaces the first substring of this string that matches the replacement.
String[]	split(String regex)	Splits this string around matches of the given **regular exp**
String[]	split(String regex, int limit)	Splits this string around matches of the given **regular exp**



Regular Expressions

Table 3.8 Some useful regular expressions

Regular Expression	Matching
\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*	E-mail address
[a-zA-z]+://[^\s]*	URL
^http://([\w-]+\.)+[\w-]+(/[\w-./?%&=]*)?$	Web address
<(.*)>(.*)<\/(.*)>|<(.*)\/>	HTML tag
^[a-zA-Z][a-zA-Z0-9_]{4,15}$	Account
\d{3}-\d{8}|\d{4}-\d{7}	Chinese telephone number
[1-9][0-9]{3,9}	QQ account number
[1-9]\d{5}(?!\d)	Chinese post code
\d{15}|\d{18}	Chinese ID card number
\n\s*\r	Blank line
\u4e00-\u9fa5	Chinese Character
^[A-Za-z]+$	String that contains only English letters
^[0-9]*$	Sting of digital number
^\d{n}$	String of digital number which length is n
^\d{n,}$	String of digital number which length is at least n
^[0-9]+(\.[0-9]{2})?$	Only 2 digits after the point