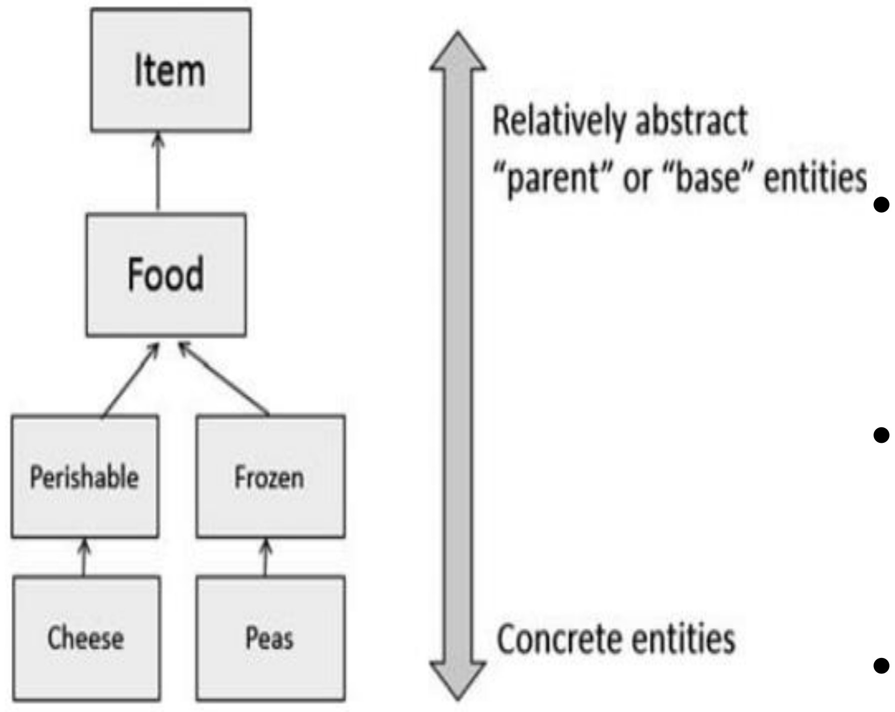
CHAPTER 5

Inheritance and Polymorphism

Objectives

- Study concepts: superclass, subclass (Base and derived classes)
  - Understand common relationships
  - Functions in inheritance
  - Using an “instanceof” operator
- Polymorphism
  - Overloading
  - Overriding
- Interface
- Abstract class
- Anonymous Classes
- Case study

Derived and Super Classes



- Object-oriented languages implement reusability of coding structure through inheritance
- It refers to the relationship between classes where one class inherits the entire structure of another class
- The root of our design is a relatively abstract entity, and we build upon that entity to produce progressively more concrete entities
- the higher-level entities are **"parent", "base" or "super"** classes
- the lower-level ones built from them are **"child", "derived" or "sub"** classes.

Object-Oriented Relationships

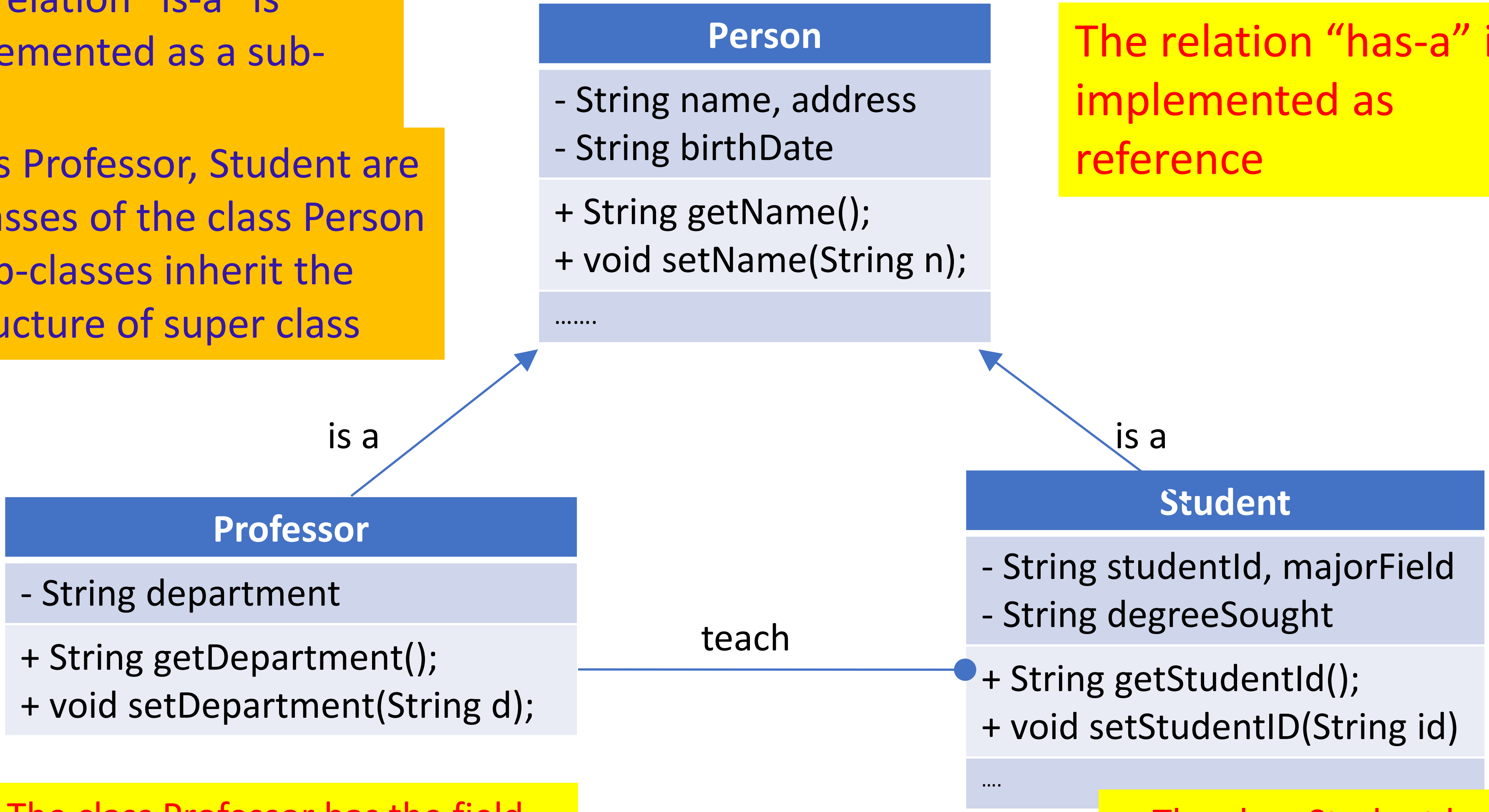
- common relationships in classes:
  - “is-a/ a kind of”
  - “has-a”
- Examples:
  - Student is a person
  - “A home is a house that has a family and a pet.”
  - An invoice contains some products and a product can be contained in some invoices

Object-Oriented Relationships…

The relation "is-a" is implemented as a sub-class

Classes Professor, Student are sub-classes of the class Person
Sub-classes inherit the structure of super class

The relation "has-a" is implemented as reference

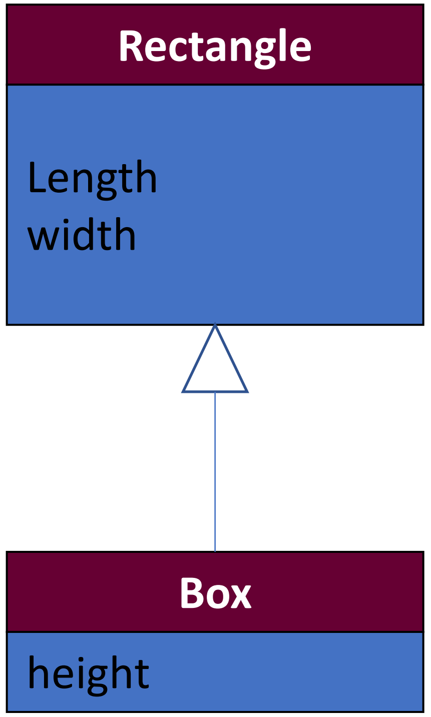


The class Professor has the field Student[] students

The class Student has the field Professor pr

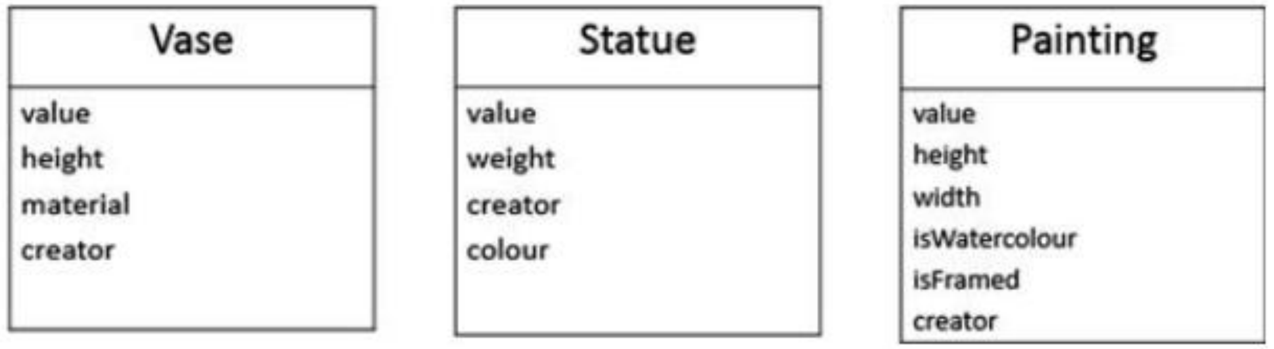
Inheritance

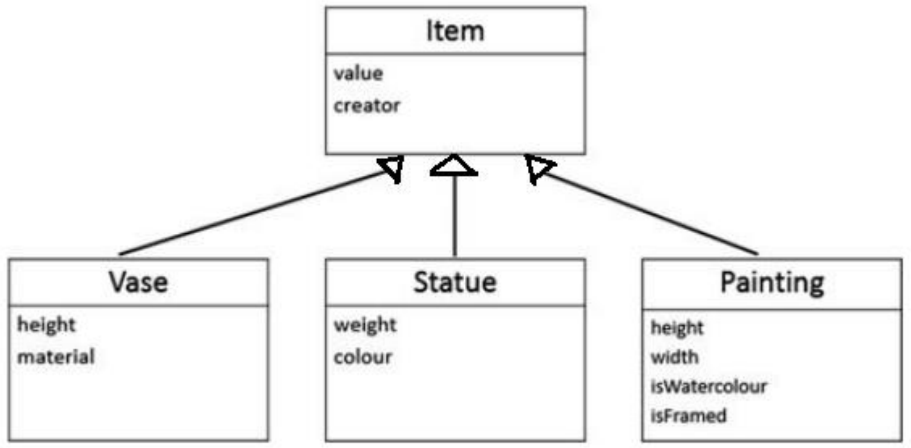
- **How to construct a class hierarchy? → Intersection**
  - Rectangle< length, width>
  - Box < length, width, height>



Inheritance…

- Consider a shop that sells antiques items, namely **vases, statues and paintings**





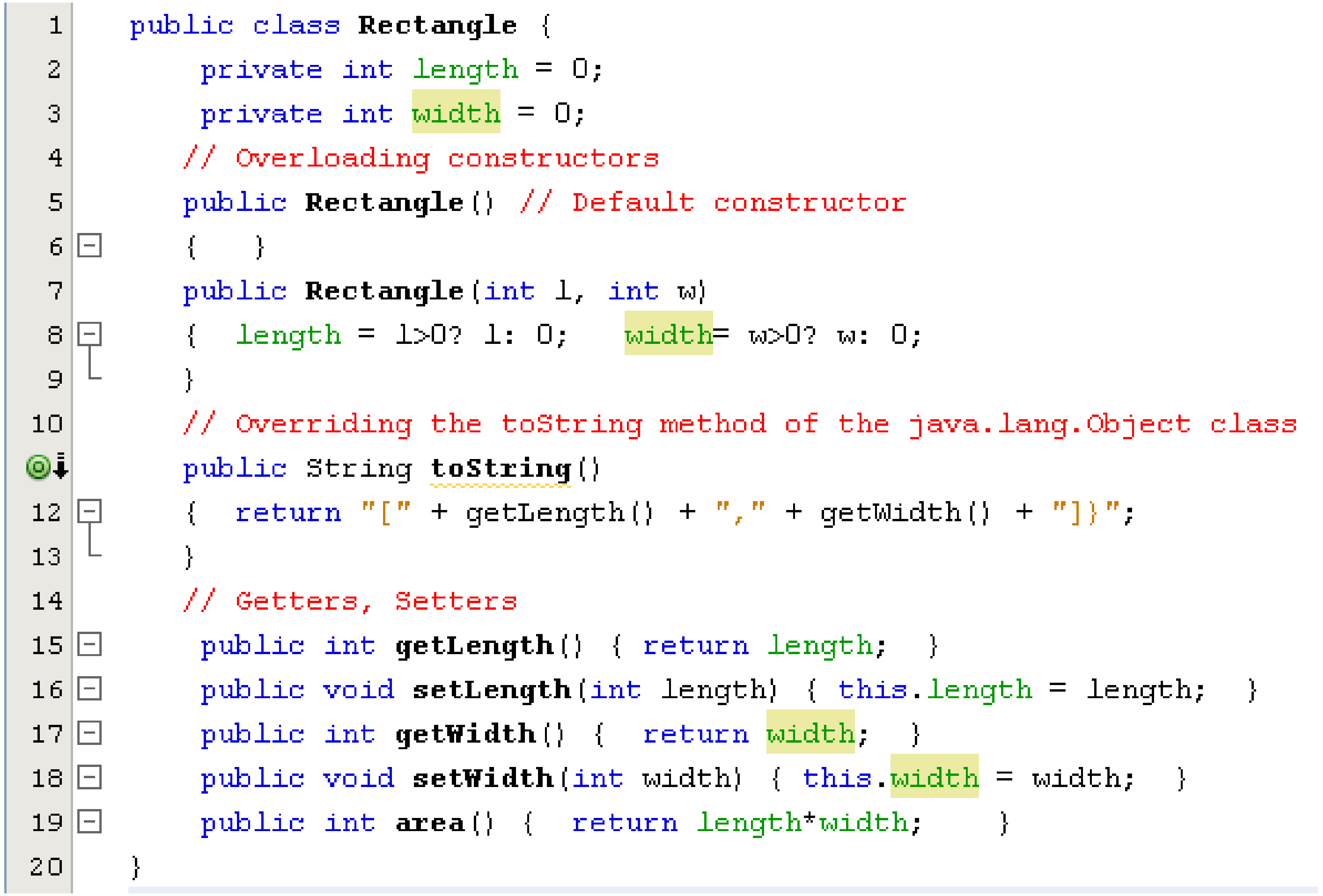
Inheritance…

- There are some sub-classes from one super class → An inheritance is a relationship where objects **share a common structure**: the structure of one object is a sub-structure of another object.
- The **extends** keyword is used to create sub-class.
- A class can be directly derived from only one class *(Java is a single-inherited OOP language)*.
- If a class does not have any superclass, then it is implicitly derived from Object class.
- Unlike other members, constructor **cannot be inherited** *(constructor of super class can not initialize sub-class objects)*

“super” Keyword

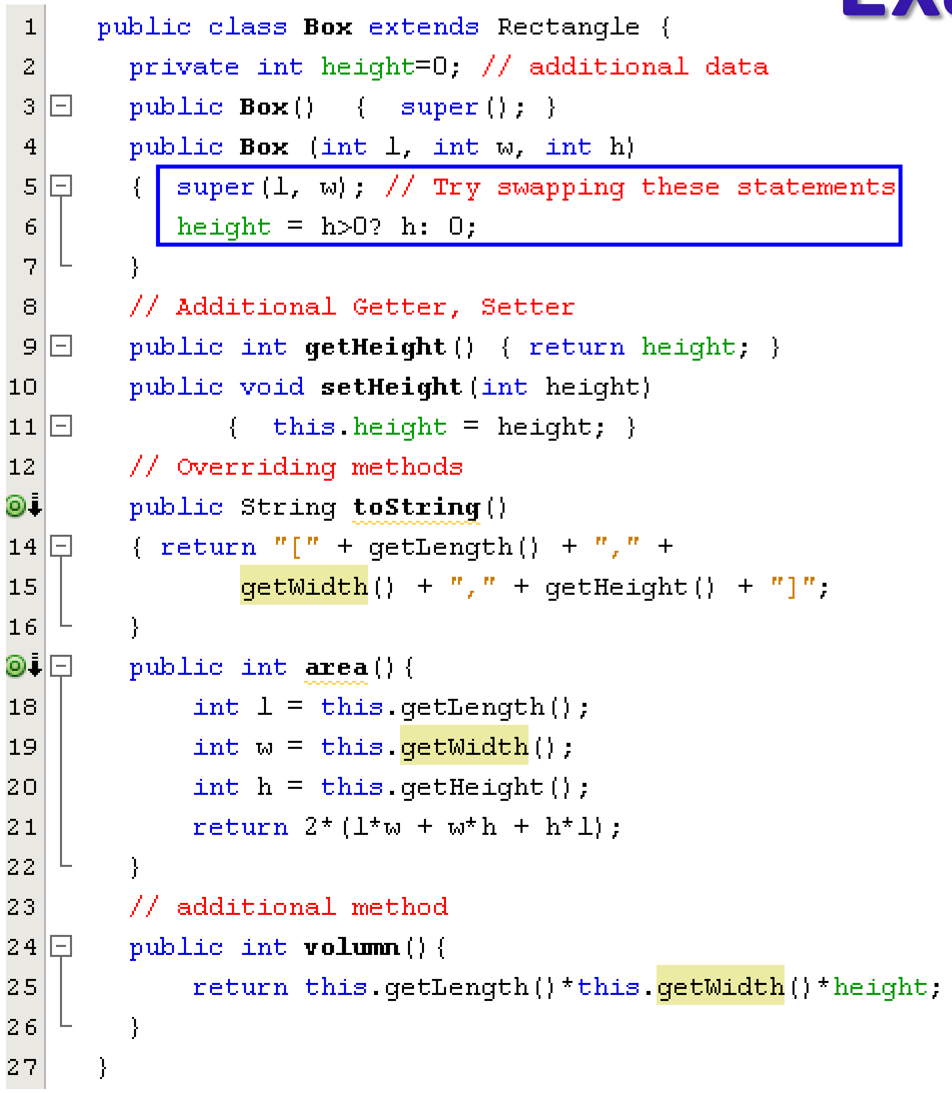
- Constructors Are **Not** Inherited
- super(...) for Constructor Reuse
  - super(arguments); *//invoke a superclass constructor*
  - Subclass constructor **must invoke super class constructor**
  - The call ***must* be the *first* statement in the subclass constructor**
- **Note**: If a constructor *does not explicitly invoke a superclass constructor*, the Java compiler *automatically inserts a call to the no-argument constructor of the superclass*. If the super class does not have a no-argument constructor, you will get a compile-time error.

Example

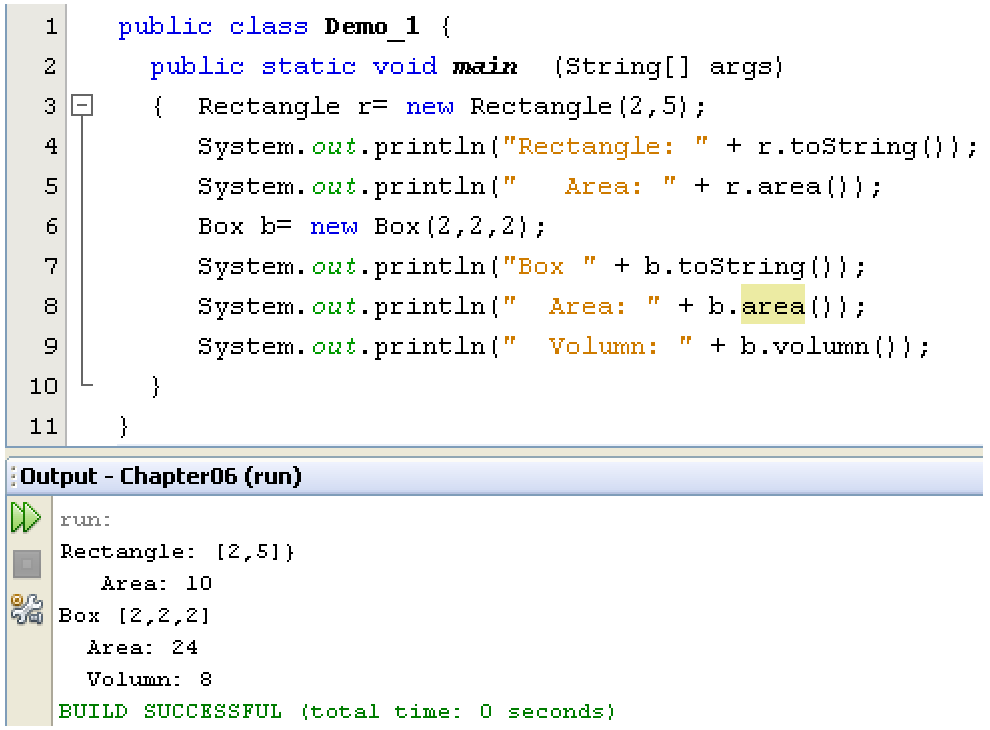


```java
public class Rectangle {
    private int length = 0;
    private int width = 0;
   // Overloading constructors
   public Rectangle() // Default constructor
   {   }
   public Rectangle(int l, int w)
   {  length = l>0? l: 0;    width= w>0? w: 0;
   }
   // Overriding the toString method of the java.lang.Object class
   public String toString()
   {  return "[" + getLength() + "," + getWidth() + "])";
   }
   // Getters, Setters
    public int getLength() { return length;  }
    public void setLength(int length) { this.length = length;  }
    public int getWidth() {  return width;  }
    public void setWidth(int width) { this.width = width;  }
    public int area() {  return length*width;     }
}
```

Example…



```
public class Box extends Rectangle {
  private int height=0; // additional data
  public Box()   {  super(); }
  public Box (int l, int w, int h)
  { super(l, w); // Try swapping these statements
    height = h>0? h: 0;
  }
  // Additional Getter, Setter
  public int getHeight() { return height; }
  public void setHeight(int height)
         {  this.height = height; }
  // Overriding methods
  public String toString()
  { return "[" + getLength() + "," +
         getWidth() + "," + getHeight() + "]";
  }
  public int area(){
      int l = this.getLength();
      int w = this.getWidth();
      int h = this.getHeight();
      return 2*(l*w + w*h + h*l);
  }
  // additional method
  public int volumn(){
      return this.getLength()*this.getWidth()*height;
  }
}
```



```
public class Demo_1 {
  public static void main  (String[] args)
  {  Rectangle r= new Rectangle(2,5);
     System.out.println("Rectangle: " + r.toString());
     System.out.println("   Area: " + r.area());
     Box b= new Box(2,2,2);
     System.out.println("Box " + b.toString());
     System.out.println("  Area: " + b.area());
     System.out.println("  Volumn: " + b.volumn());
  }
}
```

Output - Chapter06 (run)

```
run:
Rectangle: [2,5])
   Area: 10
Box [2,2,2]
  Area: 24
  Volumn: 8
BUILD SUCCESSFUL (total time: 0 seconds)
```

Example…

```
public class Point {
    private int x,y;
    public Point(int x,int y){
        this.x=x;  this.y=y;}
    …..//getter & setter }
public class Polygon {
    protected Point d1, d2, d3, d4;
    public void setD1(Point d1) {this.d1=d1;}
   public Point getD1(){return d1;}
    @Override
    public String toString(){
      return
 d1.getX()+"\t"+d1.getY()+"\t"+d2.getX()+"\t"+d2.getY()+
 "\t"+d3.getX()+"\t"+d3.getY()+"\t"+d4.getX()+"\t"+d4.getY(); }}
```

```
public class  Square extends Polygon{
  public Square(){
      d1 = new Point(0,0); d2 = new Point(0,1);
      d3 = new Point (1,0);d4 = new Point(1,1);
     }
}
public class Main {
    public static void main(String args[]){
        Square sq = new Square();
        System.out.println(sq.toString());
    }
}
```

```
public class Person {
    private String name;
    private String bithday;
    public Person() {
    }  // getter & setter }
public class Employee extends Person {
    private double salary;
    public double getSalary() { return salary; }
    public void setSalary(double salary){
        this.salary = salary;    }
    @Override
    public String toString() {
 //return name + "\t" + birthday + "\t" + salary;
      return super.getName() + "\t" + super.getBithday() +
 "\t" + salary;
    }
}
```

```
public class Main {
    public static void main(String
  args[]){
        Employee e = new Employee();
        e.setName("To Ngoc Van");
        e.setBithday("3/4/1994");
        e.setSalary(4.4);
  System.out.println(e.toString());
    }
}
```

Output???

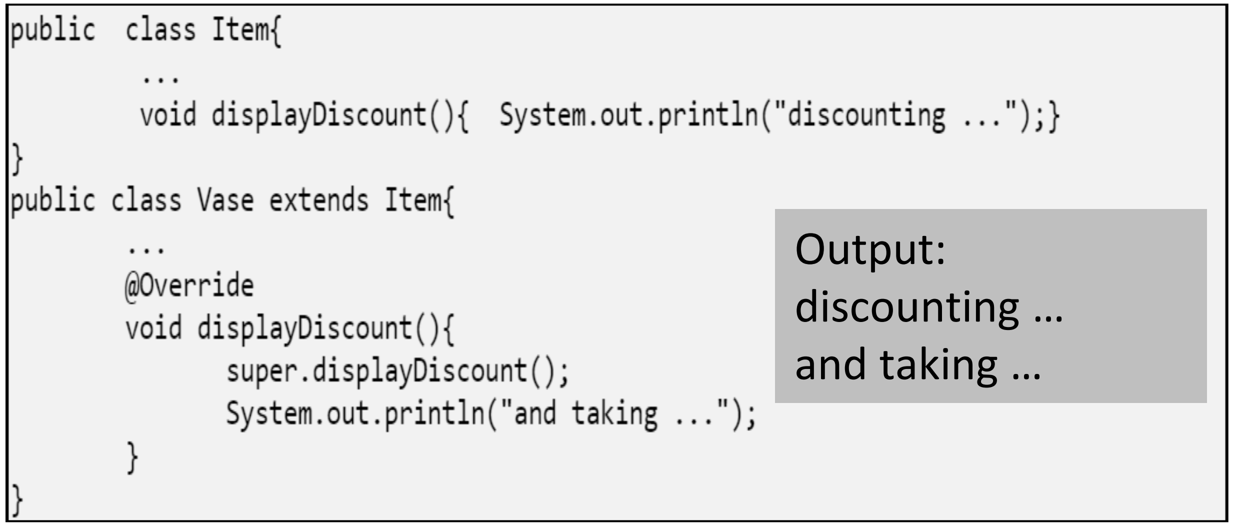
```
1. public class Polygon {
    protected Point d1, d2, d3, d4;
    public Polygon() {
     System.out.println(“Polygon class");
    }……}
2. public class Square extends Polygon{
  public Square(){
     System.out.println(“Square class");
    }
}
3. public class Main {
    public static void main(String args[]){
        Square hv = new Square();
    }
}
```

```
1. public class Polygon {
     protected Point d1, d2, d3, d4;
     //a constructor is not default contructor
     ……}
2. public class Square extends Polygon{
   public Square(){
      System.out.println("Square class");
     }
}
3. public class Main {
     public static void main(String args[]){
          Square hv = new Square();
     }
}
```

Why Error???

Functions in inheritance

- A derived class inherits from superclass is limited to the normal member functions of the superclass.
- We use the Java keyword **super** as the qualifier for calling a superclass ’s method:
  - ***super.methodName(arguments);***
  - To invoke the version of method methodName that was defined by our superclass.
- **Hiding a method**: Re-implementing a static method implemented in super class



```
public  class Item{
          ...
          void displayDiscount(){  System.out.println("discounting ...");}
}
public class Vase extends Item{
         ...
         @Override
         void displayDiscount(){
                 super.displayDiscount();
                 System.out.println("and taking ...");
         }
}
```

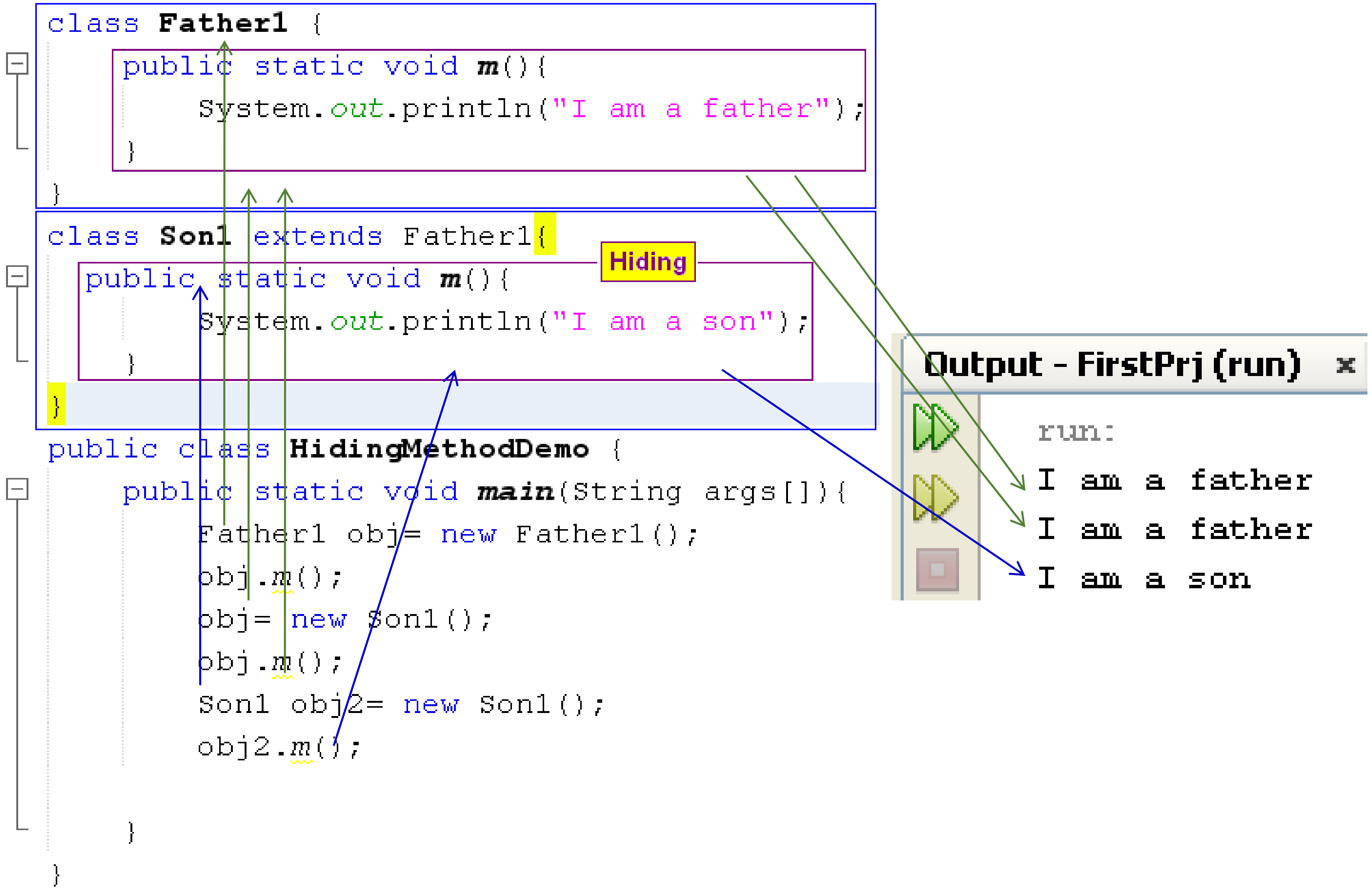
```
Output:
discounting …
and taking …
```

- The "displayDiscount" method has the same signature (name, plus the number and the type of its parameters) and return type as in the superclass. It is called overriding the superclass's method=> We will learn override method in the next topic
- The *"displayDiscount"*) that was defined by our superclass. We use the **"super"** keyword

Overriding and Hiding Methods (1)

- **Overriding a method**: An instance method in a subclass with the same signature (name, plus the number and the type of its parameters) and return type as an instance method in the superclass overrides the superclass's method.
  - Use the **@Override** annotation that instructs the compiler that you intend to override a method in the superclass (you may not use it because overriding is default in Java).
- **Hiding a method**: Re-implementing a static method implemented in super class

Hiding Method



```
class Father1 {
    public static void m(){
        System.out.println("I am a father");
    }
}
class Son1 extends Father1{
    public static void m(){
        System.out.println("I am a son");
    }
}
public class HidingMethodDemo {
    public static void main(String args[]){
        Father1 obj= new Father1();
        obj.m();
        obj= new Son1();
        obj.m();
        Son1 obj2= new Son1();
        obj2.m();

    }
}
```

```
run:
I am a father
I am a father
I am a son
```

Using an “instanceof” operator

- Dynamic and Static type
  - dynamic type: A reference variable that has the type of the superclass can store the address of the object of sub class. It is called to be *dynamic type*, the type that is has at runtime.
    Rectangle obj1 = new Box();
  - Static type: The type that it has when first declared. Static type checking is enforced by the compiler.
    Box obj2 = new Box();
- *“Instanceof” operator:* It checks whether the reference of an object belongs to the provided type or not, the instanceof operator will return true or false.

```
If ( obj1  instanceof  Box)
    System.out.println(“ obj1 is pointing to the Box object”);
```

Casting

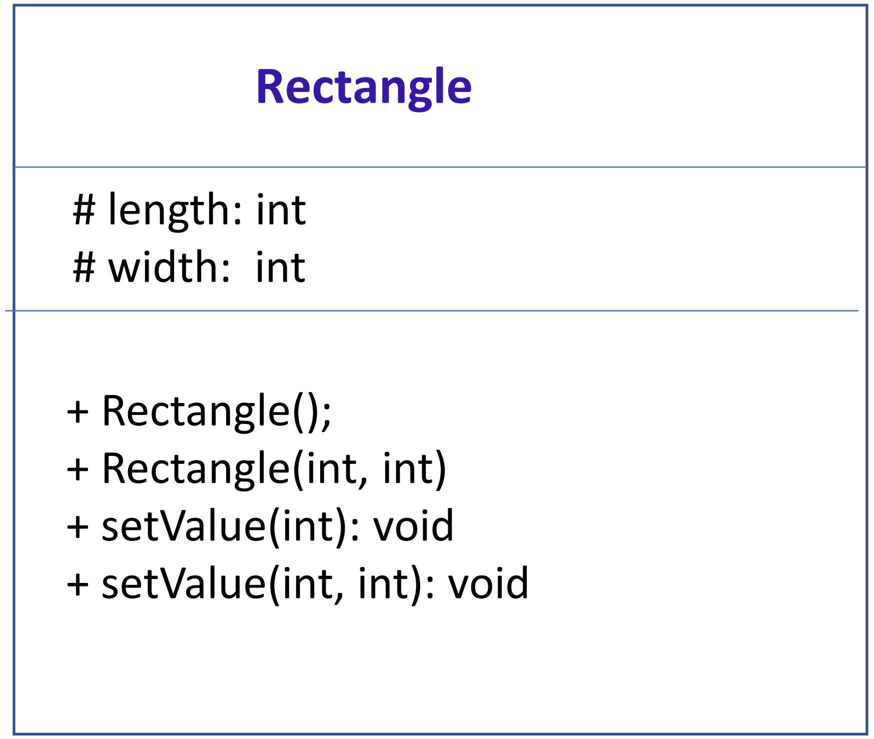
- A variable that has the type of the superclass only calls methods of the superclass. To call methods of the subclass we must *cast explicitly*
- *for example,*

```
Rectangle obj = new Box();
((Box)obj).setHeight(300);
```

Polymorphism

- The dictionary definition of *polymorphism* refers to a principle in biology in which an organism or species can have many different forms or stages.
- Polymorphism was perfected in object-oriented languages
- Ability allows many versions of a method based on **overloading** and **overriding** methods techniques.
- **Overloading**: A class can have some methods which have the same name but their parameter types are different.
- **Overriding**: A method in the father class can be overridden in its derived classes (body of a method can be replaced in derived classes).

Overloading



Rectangle

# length: int
# width: int

+ Rectangle();
+ Rectangle(int, int)
+ setValue(int): void
+ setValue(int, int): void

- overloading with constructors

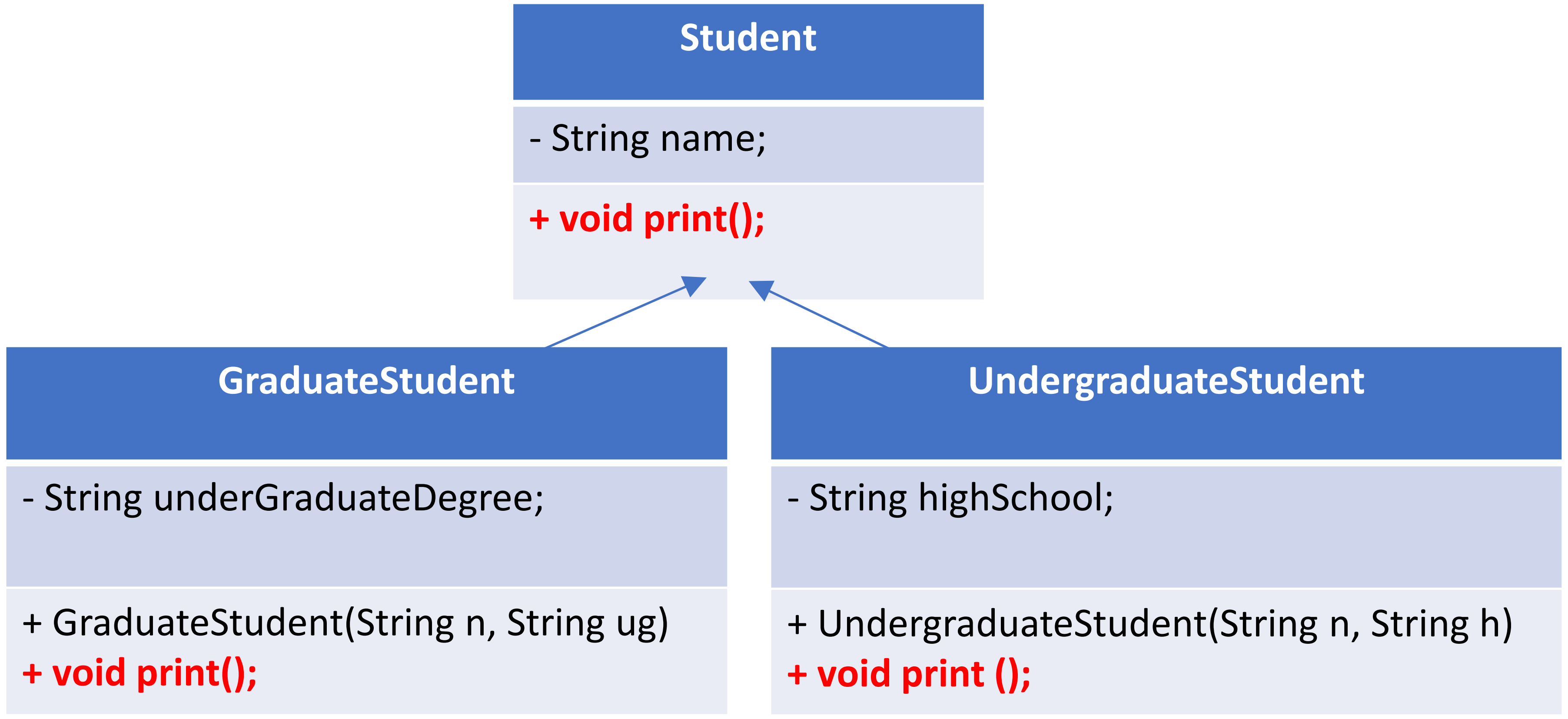
```
public Rectangle(){…}

    public Rectangle(int length,
int  width){… }
```

- Overloading also extends to general methods.

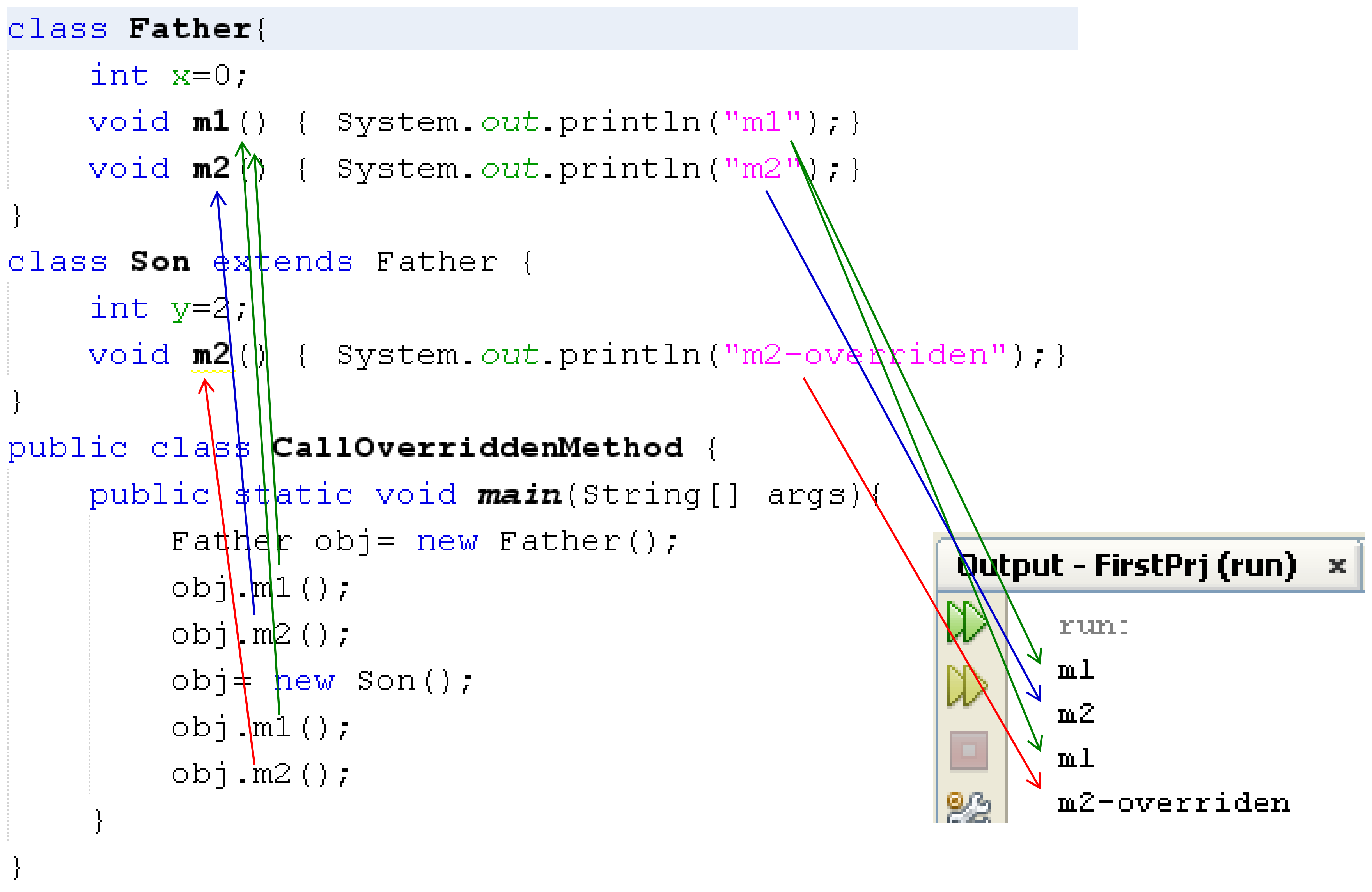
```
public void setValue(int len){
    length= (len>0)?1:0;
}
public void setValue (int  len,
int wi){
      length= (len>0)? 1: 0;
      width= (wi>0)? wi:0;
}
```

Overriding

- A subclass provides the specific implementation of the method that has been declared by one of its pare
- Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class.



How Can Overridden Method be Determined?



```
class Father{
    int x=0;
    void m1() { System.out.println("m1");}
    void m2() { System.out.println("m2");}
}
class Son extends Father {
    int y=2;
    void m2() { System.out.println("m2-overriden");}
}
public class CallOverriddenMethod {
    public static void main(String[] args){
        Father obj= new Father();
        obj.m1();
        obj.m2();
        obj= new Son();
        obj.m1();
        obj.m2();
    }
}
```

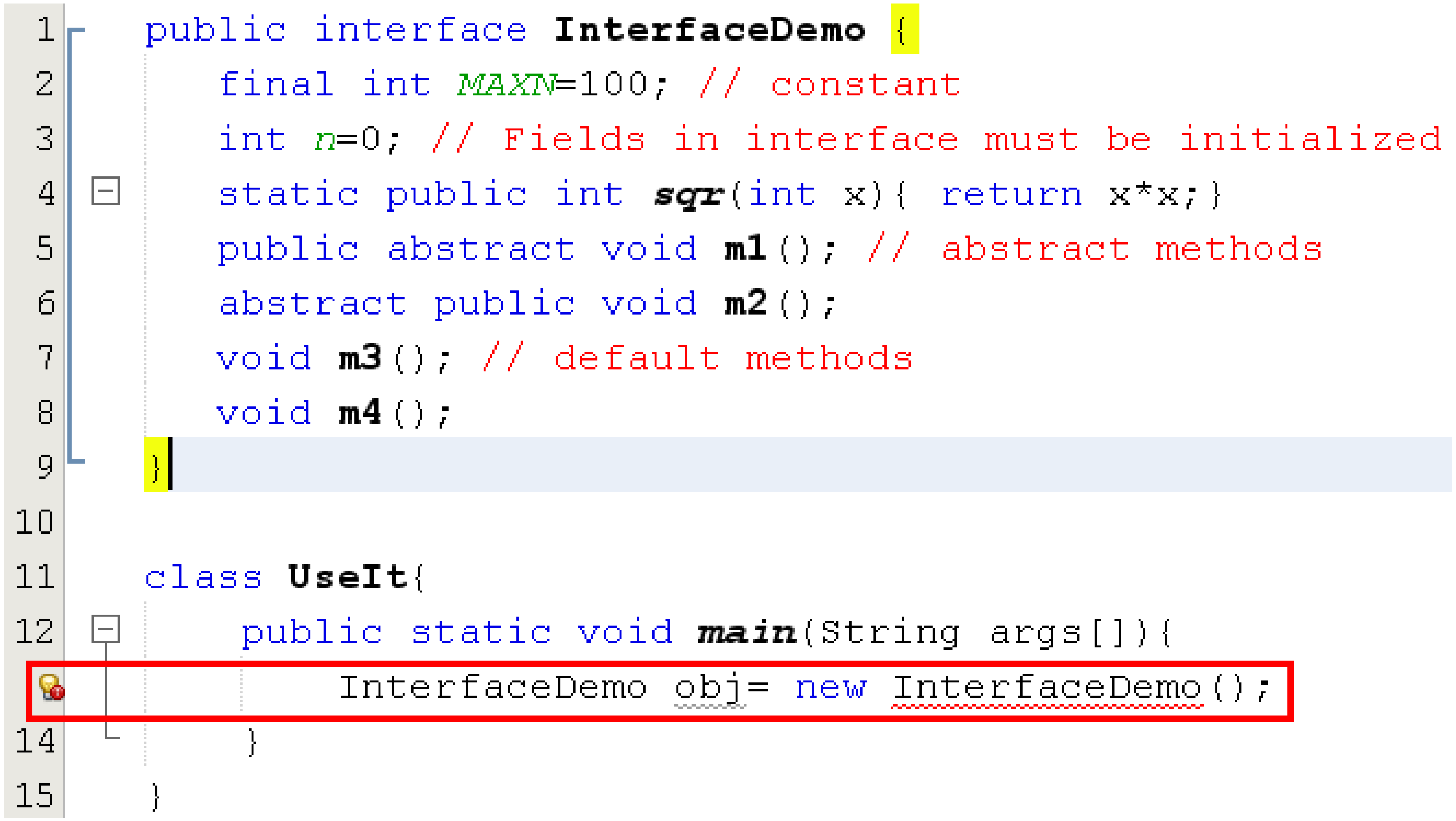
Output - FirstPrj (run)

```
run:
m1
m2
m1
m2-overriden
```

Interfaces

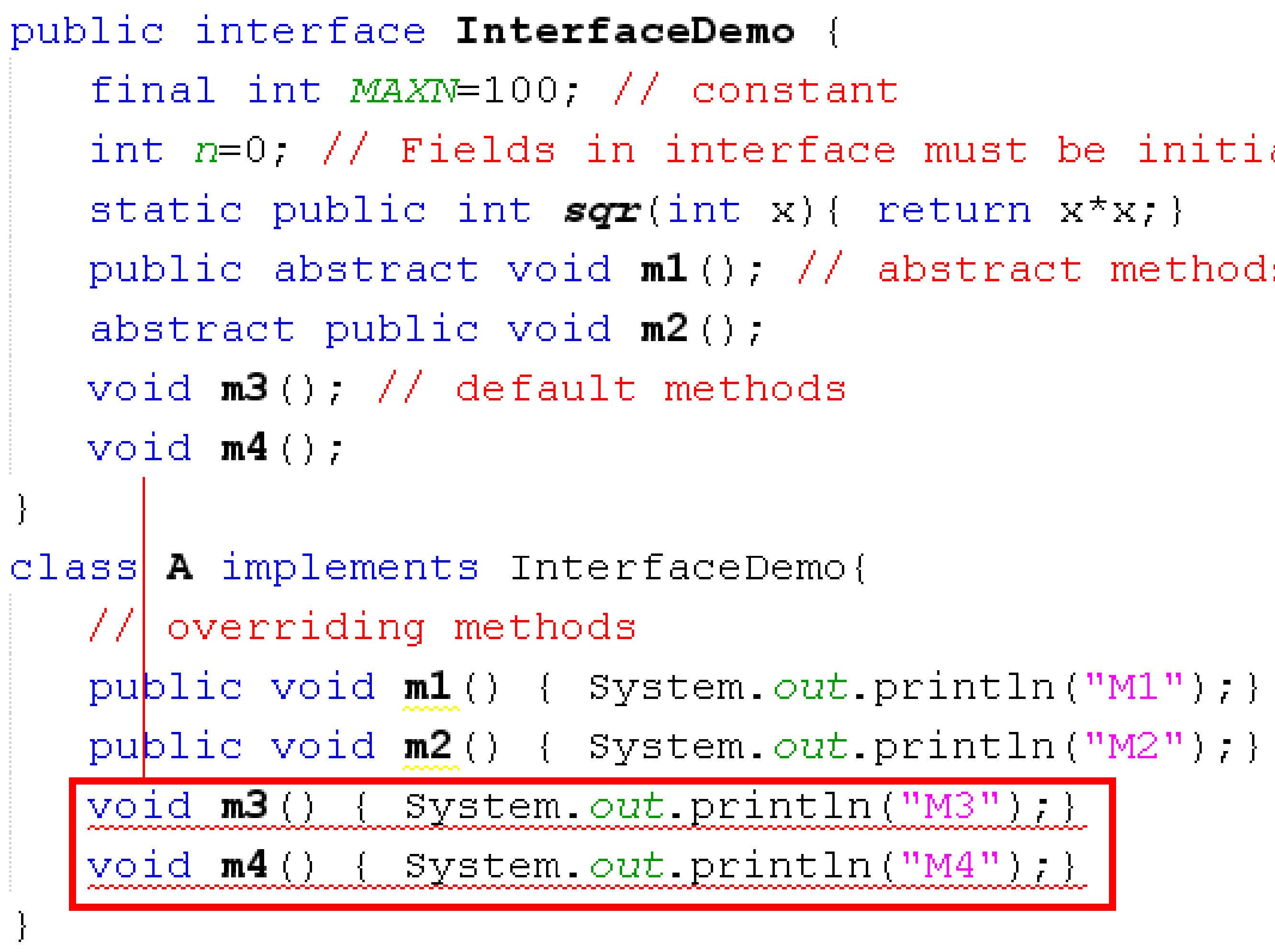
- An *interface* is a reference type, similar to a class, that can contain *only* **constants**, initialized fields, static methods, prototypes (abstract methods, default methods), static methods, and nested types.
- It will be the **core** of some classes
- Interfaces cannot be instantiated because they have **no-body** methods.
- Interfaces can only be *implemented* by classes or *extended* by other interfaces.
- WHY AND WHEN TO USE INTERFACES?
  - Objects define their interaction with the outside world through the methods that they expose
  - Java does not support "multiple inheritance" (a class can only inherit from one superclass). However, it can be achieved with interfaces, because the class can implement multiple interfaces

Interfaces...



```
public interface InterfaceDemo {
    final int MAXN=100; // constant
    int n=0; // Fields in interface must be initialized
    static public int sqr(int x){ return x*x;}
    public abstract void m1(); // abstract methods
    abstract public void m2();
    void m3(); // default methods
    void m4();
}

class UseIt{
    public static void main(String args[]){
        InterfaceDemo obj= new InterfaceDemo();
    }
}
```

Interfaces...

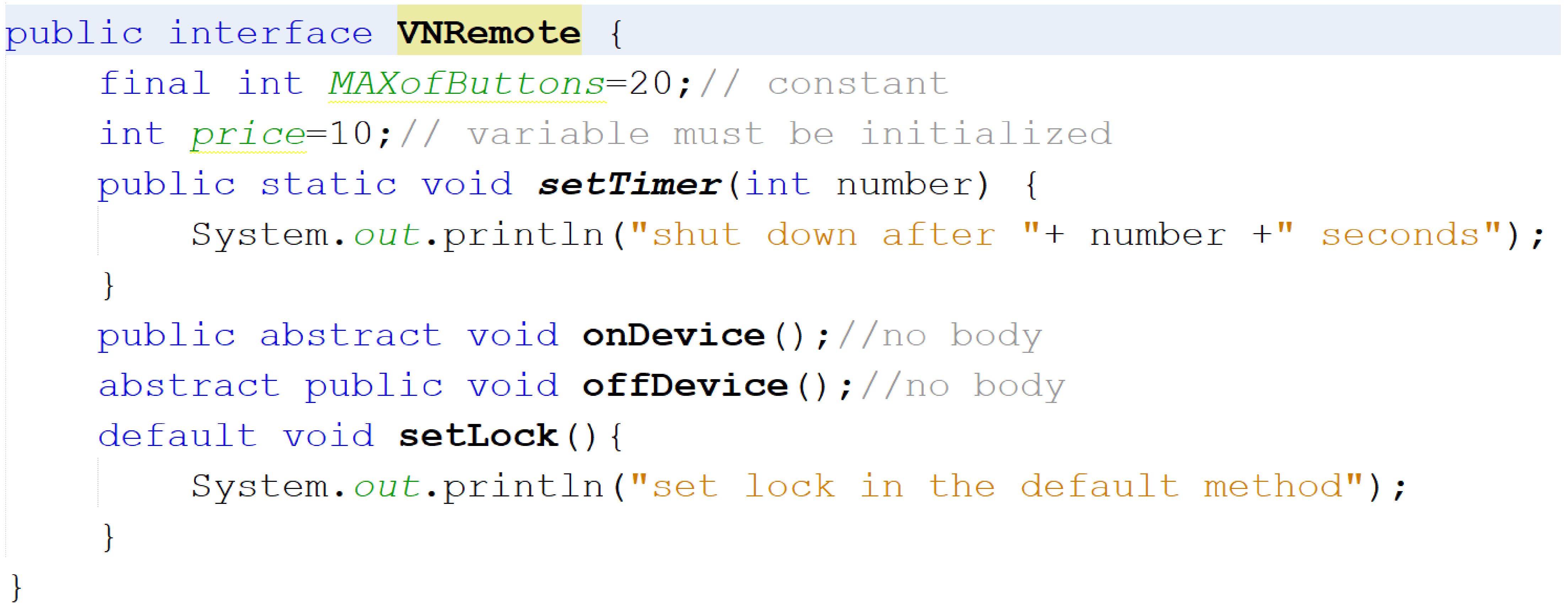


```
public interface InterfaceDemo {
    final int MAXN=100; // constant
    int n=0; // Fields in interface must be initialized
    static public int sqr(int x){ return x*x;}
    public abstract void m1(); // abstract methods
    abstract public void m2();
    void m3(); // default methods
    void m4();
}
class A implements InterfaceDemo{
    // overriding methods
    public void m1() { System.out.println("M1");}
    public void m2() { System.out.println("M2");}
    void m3() { System.out.println("M3");}
    void m4() { System.out.println("M4");}
}
```

m3(), m4() in A cannot implement m3(), m4() in InterfaceDemo, attempting to assign weaker access privileges, were public

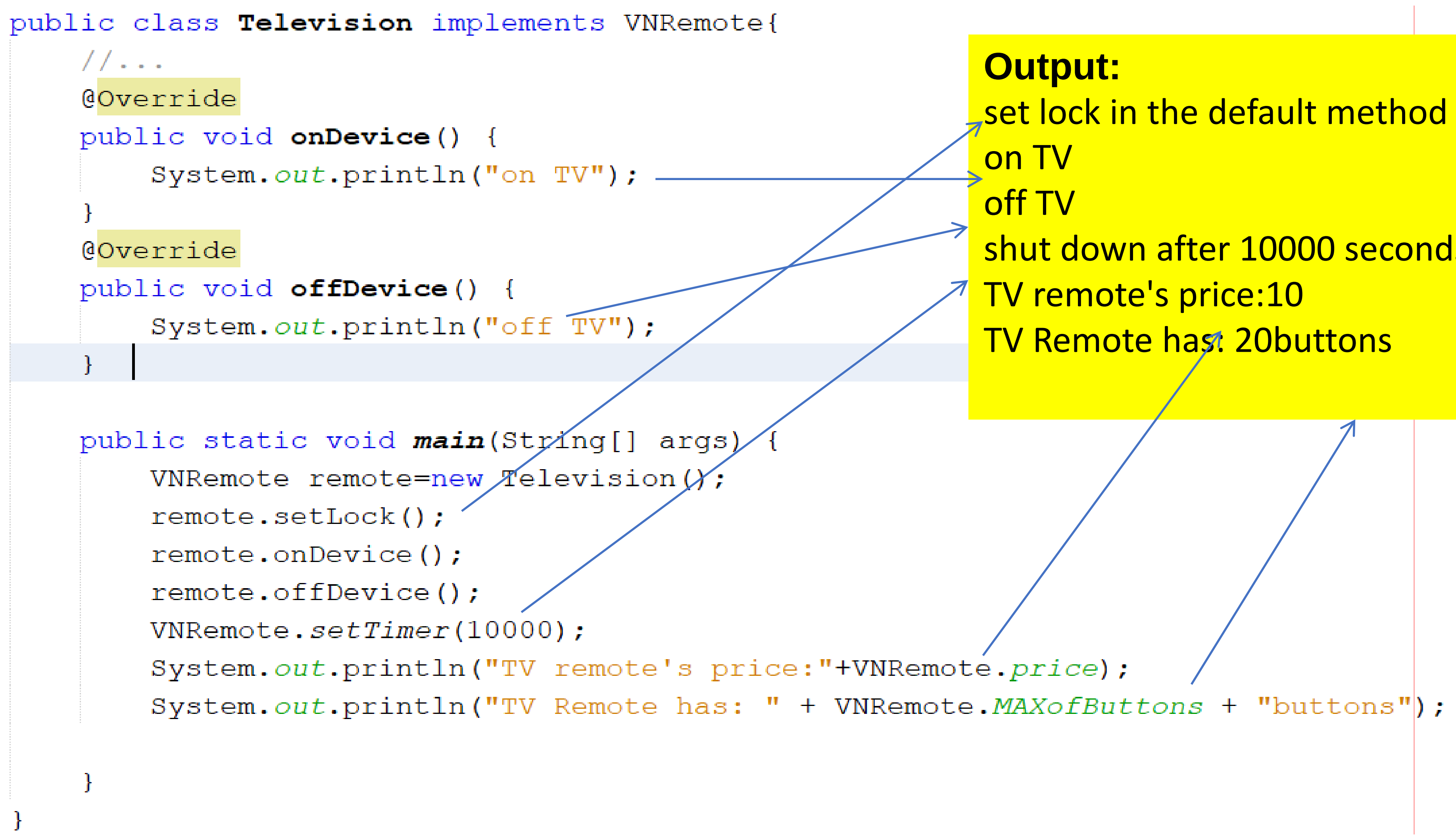
Default methods of an interface must be overridden as public methods in concrete classes.

Example: how to create an interface



```java
public interface VNRemote {
    final int MAXofButtons=20;// constant
    int price=10;// variable must be initialized
    public static void setTimer(int number) {
        System.out.println("shut down after "+ number +" seconds");
    }
    public abstract void onDevice();//no body
    abstract public void offDevice();//no body
    default void setLock(){
        System.out.println("set lock in the default method");
    }
}
```

Example:implement an interface

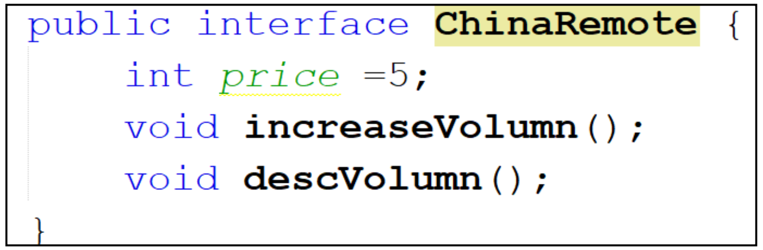


```
public class Television implements VNRemote{
    //...
    @Override
    public void onDevice() {
        System.out.println("on TV");
    }
    @Override
    public void offDevice() {
        System.out.println("off TV");
    }

    public static void main(String[] args) {
        VNRemote remote=new Television();
        remote.setLock();
        remote.onDevice();
        remote.offDevice();
        VNRemote.setTimer(10000);
        System.out.println("TV remote's price:"+VNRemote.price);
        System.out.println("TV Remote has: " + VNRemote.MAXofButtons + "buttons");

    }
}
```

Output:

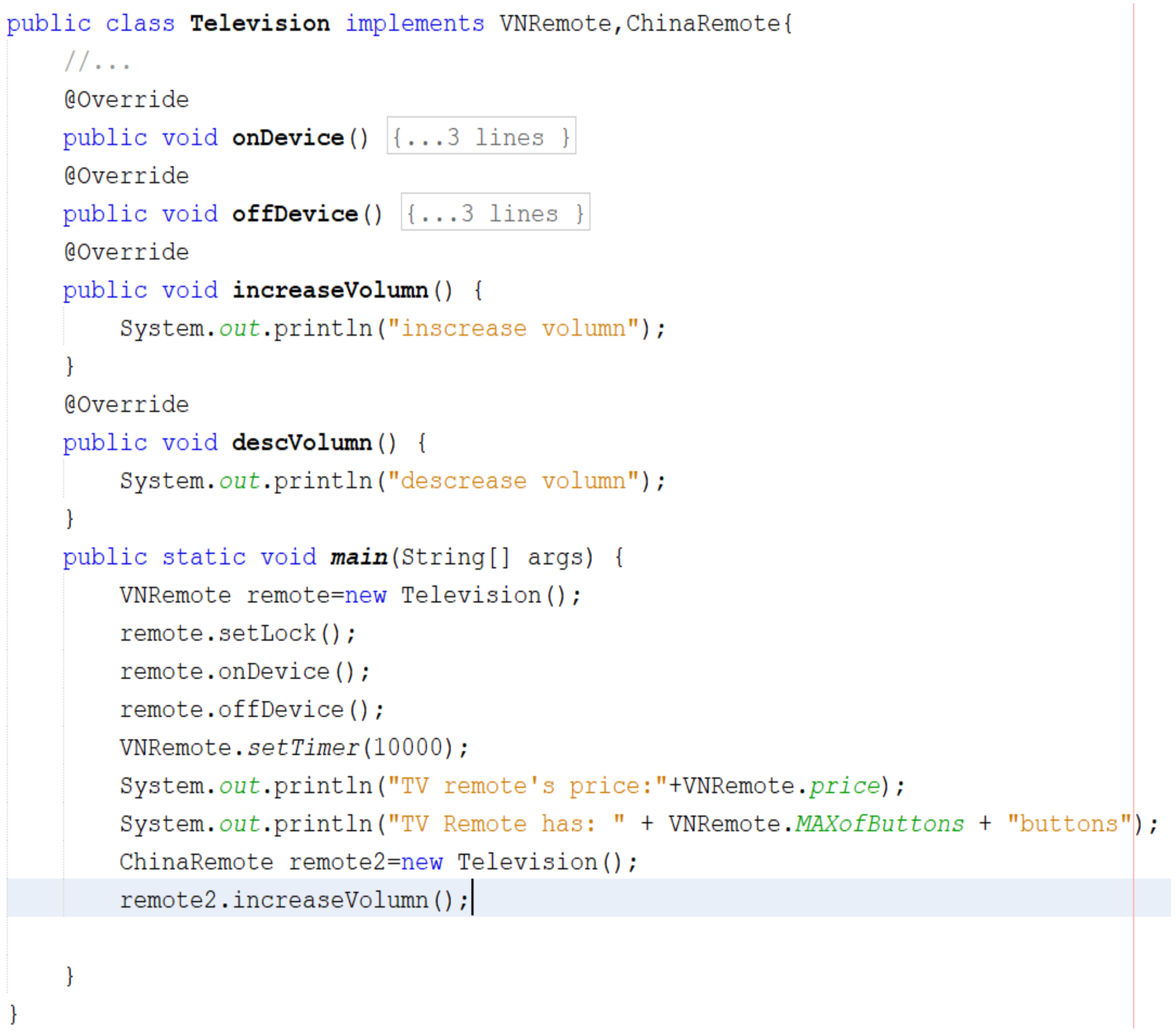
```
set lock in the default method
on TV
off TV
shut down after 10000 seconds
TV remote's price:10
TV Remote has: 20buttons
```

Example: multiple interfaces



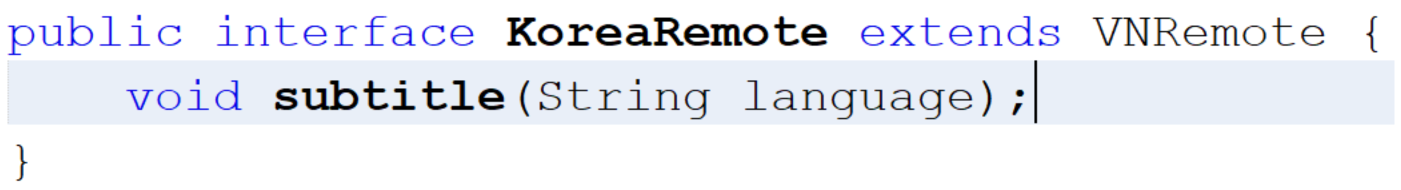
```java
public interface ChinaRemote {
    int price =5;
    void increaseVolumn();
    void descVolumn();
}
```

Output:
set lock in the default method
on TV
off TV
shut down after 10000 seconds
TV remote's price:10
TV Remote has: 20buttons
increase volumn

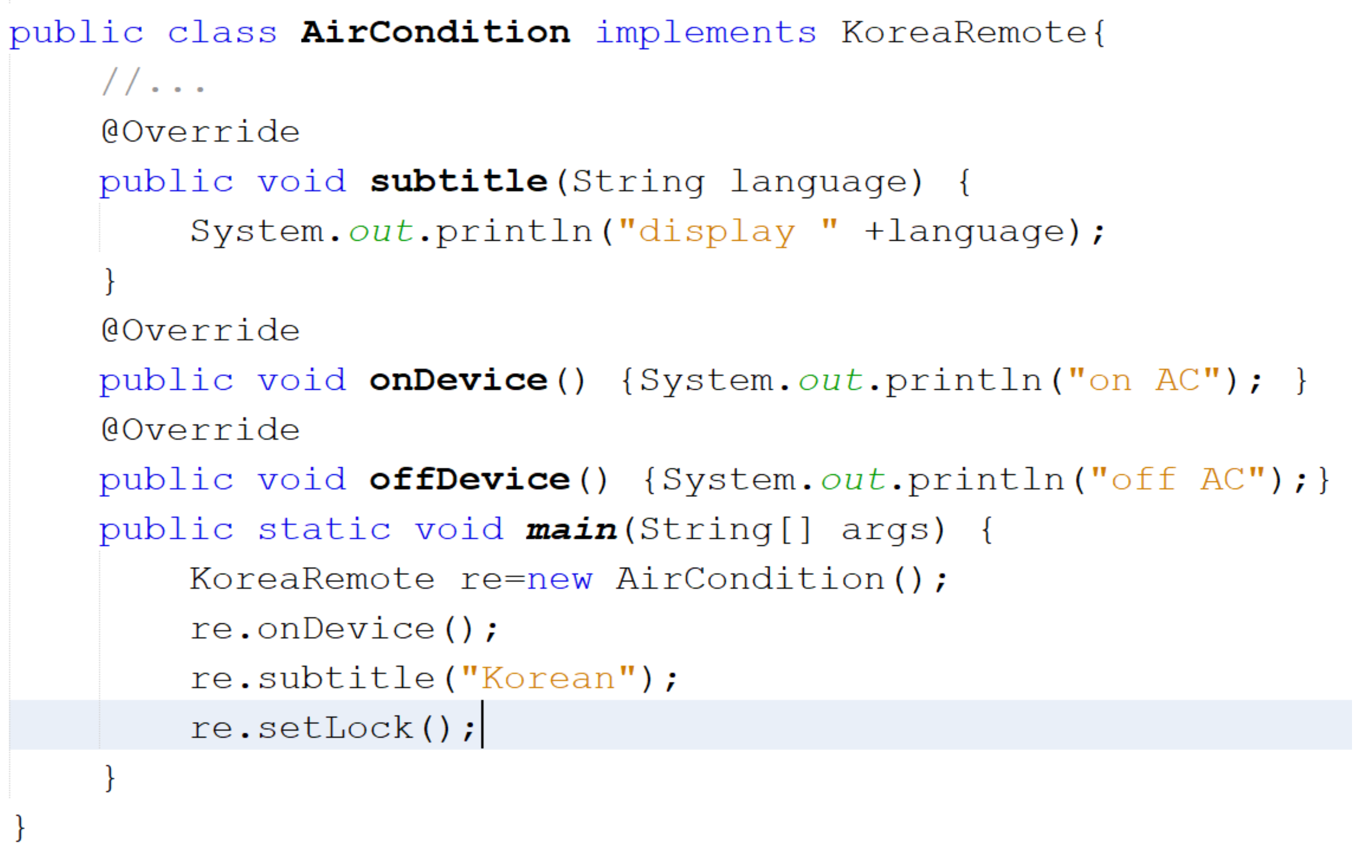


```java
public class Television implements VNRemote,ChinaRemote{
    //...
    @Override
    public void onDevice() {...3 lines }
    @Override
    public void offDevice() {...3 lines }
    @Override
    public void increaseVolumn() {
        System.out.println("inscrease volumn");
    }
    @Override
    public void descVolumn() {
        System.out.println("descrease volumn");
    }
    public static void main(String[] args) {
        VNRemote remote=new Television();
        remote.setLock();
        remote.onDevice();
        remote.offDevice();
        VNRemote.setTimer(10000);
        System.out.println("TV remote's price:"+VNRemote.price);
        System.out.println("TV Remote has: " + VNRemote.MAXofButtons + "buttons");
        ChinaRemote remote2=new Television();
        remote2.increaseVolumn();

    }
}
```

Example: how to extend interfaces



```java
public interface KoreaRemote extends VNRemote {
    void subtitle(String language);
}
```



```java
public class AirCondition implements KoreaRemote{
    //...
    @Override
    public void subtitle(String language) {
        System.out.println("display " +language);
    }
    @Override
    public void onDevice() {System.out.println("on AC"); }
    @Override
    public void offDevice() {System.out.println("off AC");}
    public static void main(String[] args) {
        KoreaRemote re=new AirCondition();
        re.onDevice();
        re.subtitle("Korean");
        re.setLock();
    }
}
```

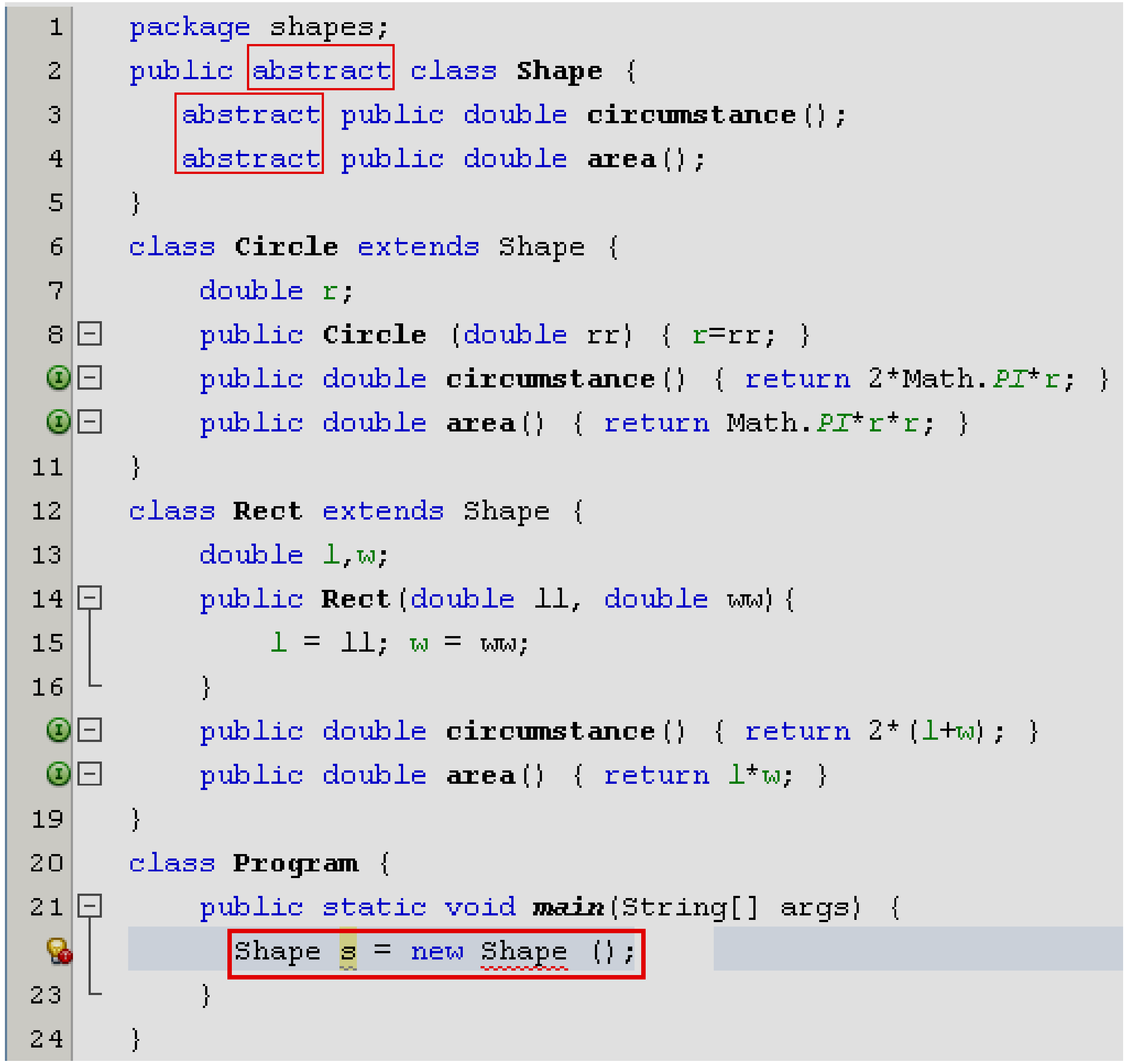
Output:

```
on AC
display Korean
set lock in the
default method
```

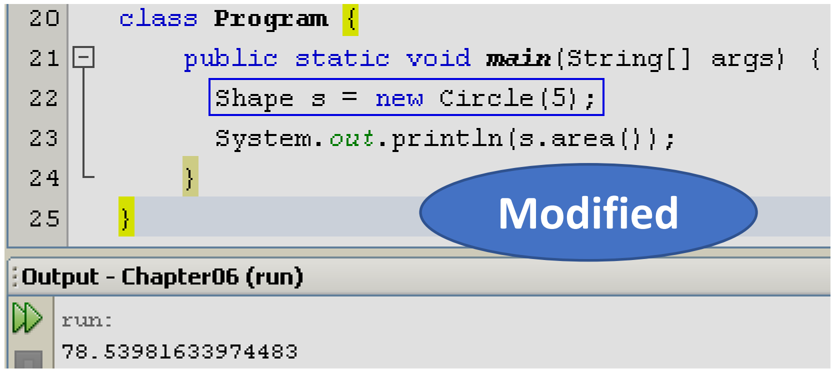
Abstract Classes

- Used to define ***what* behaviors a class is required to perform without having to**
  provide an explicit implementation.
- **It is the result of so-high generalization**
- **Syntax to define a abstract class**
  - *public abstract class className{ ... }*
- It isn't necessary for all of the methods in an abstract class to be abstract.
- An abstract class can also declare implemented methods.

Abstract Classes…



```
package shapes;
public abstract class Shape {
    abstract public double circumstance();
    abstract public double area();
}
class Circle extends Shape {
    double r;
    public Circle (double rr) { r=rr; }
    public double circumstance() { return 2*Math.PI*r; }
    public double area() { return Math.PI*r*r; }
}
class Rect extends Shape {
    double l,w;
    public Rect(double ll, double ww){
        l = ll; w = ww;
    }
    public double circumstance() { return 2*(l+w); }
    public double area() { return l*w; }
}
class Program {
    public static void main(String[] args) {
        Shape s = new Shape ();
    }
}
```



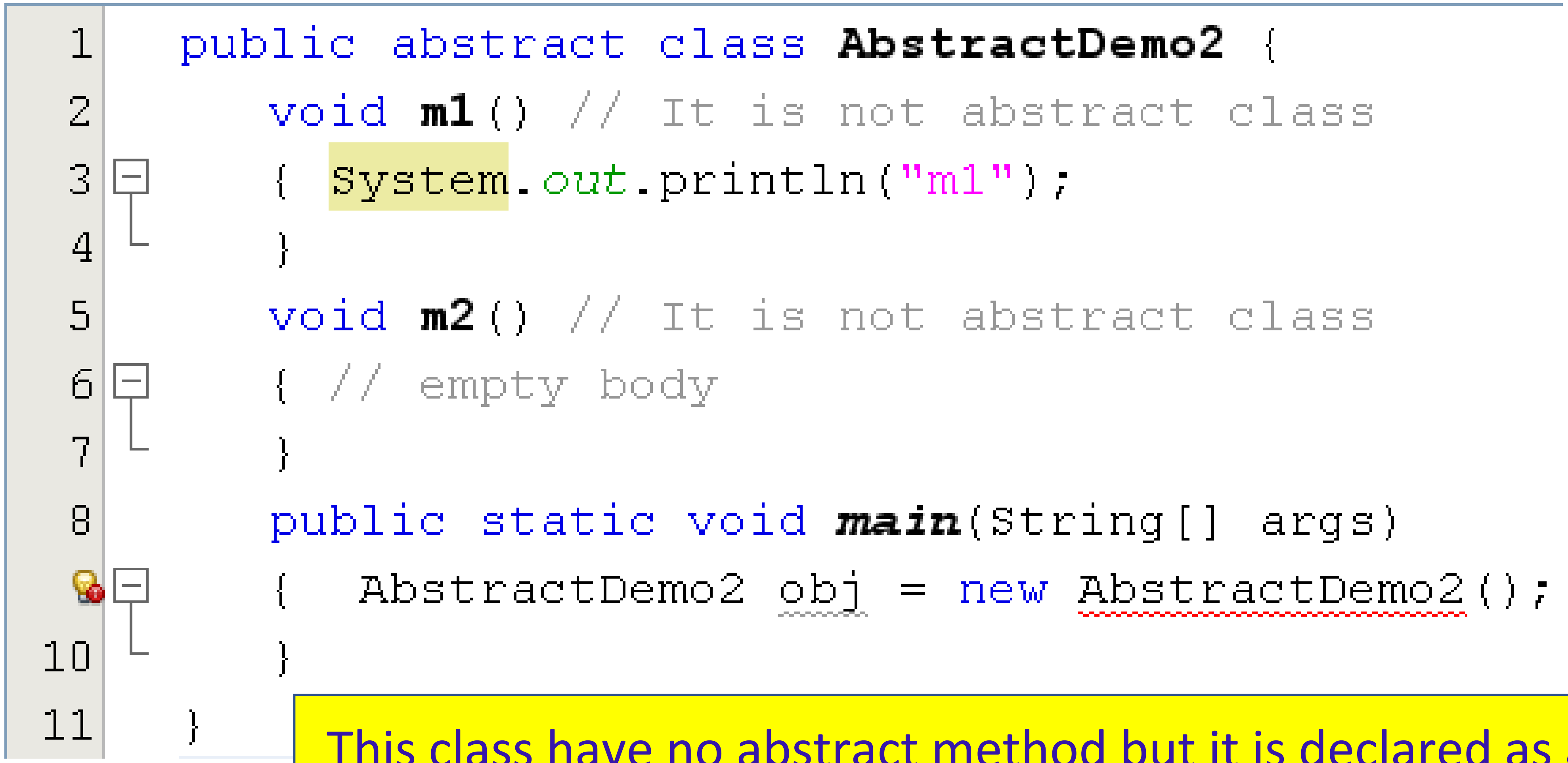
```
class Program {
    public static void main(String[] args) {
        Shape s = new Circle(5);
        System.out.println(s.area());
    }
}
```

Modified

Output - Chapter06 (run)

```
run:
78.53981633974483
```

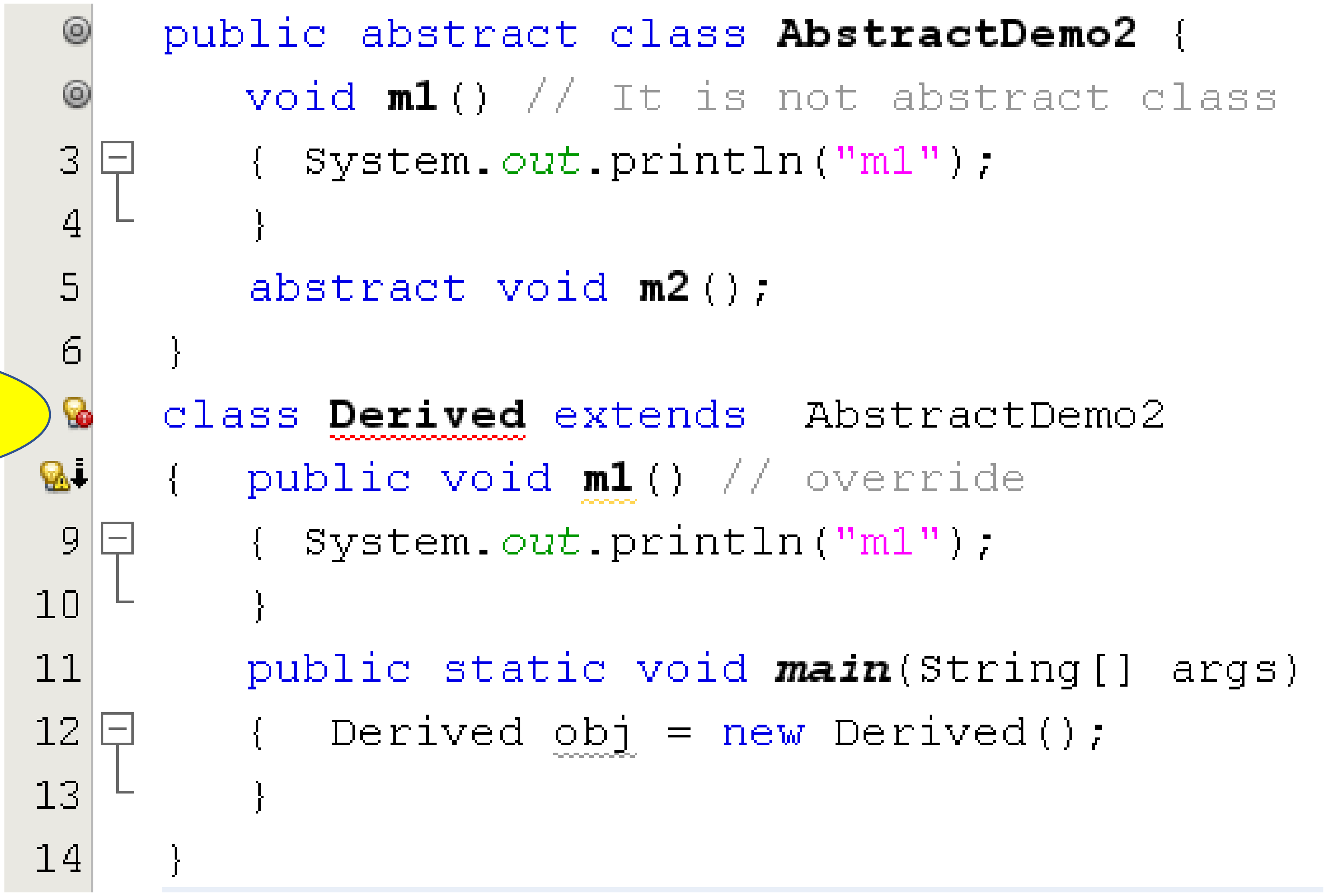
Abstract Classes...



```
public abstract class AbstractDemo2 {
    void m1() // It is not abstract class
    { System.out.println("m1");
    }
    void m2() // It is not abstract class
    { // empty body
    }
    public static void main(String[] args)
    {  AbstractDemo2 obj = new AbstractDemo2();
    }
}
```

This class have no abstract method but it is declared as an abstract class. So, we can not initiate an object of this class.

Abstract Classes...



```
public abstract class AbstractDemo2 {
    void m1() // It is not abstract class
    { System.out.println("m1");
    }
    abstract void m2();
}
class Derived extends  AbstractDemo2
{   public void m1() // override
    { System.out.println("m1");
    }
    public static void main(String[] args)
    {   Derived obj = new Derived();
    }
}
```

Error. Why?

Implementing Abstract Methods

- Derive a class from an abstract superclass, the subclass will inherit all of the superclass's
  features, all of ***abstract* methods** included**.**
- To replace an inherited abstract method with
  a concrete version, the subclass need merely override it.
- Abstract classes ***cannot be instantiated***

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

Anonymous Classes

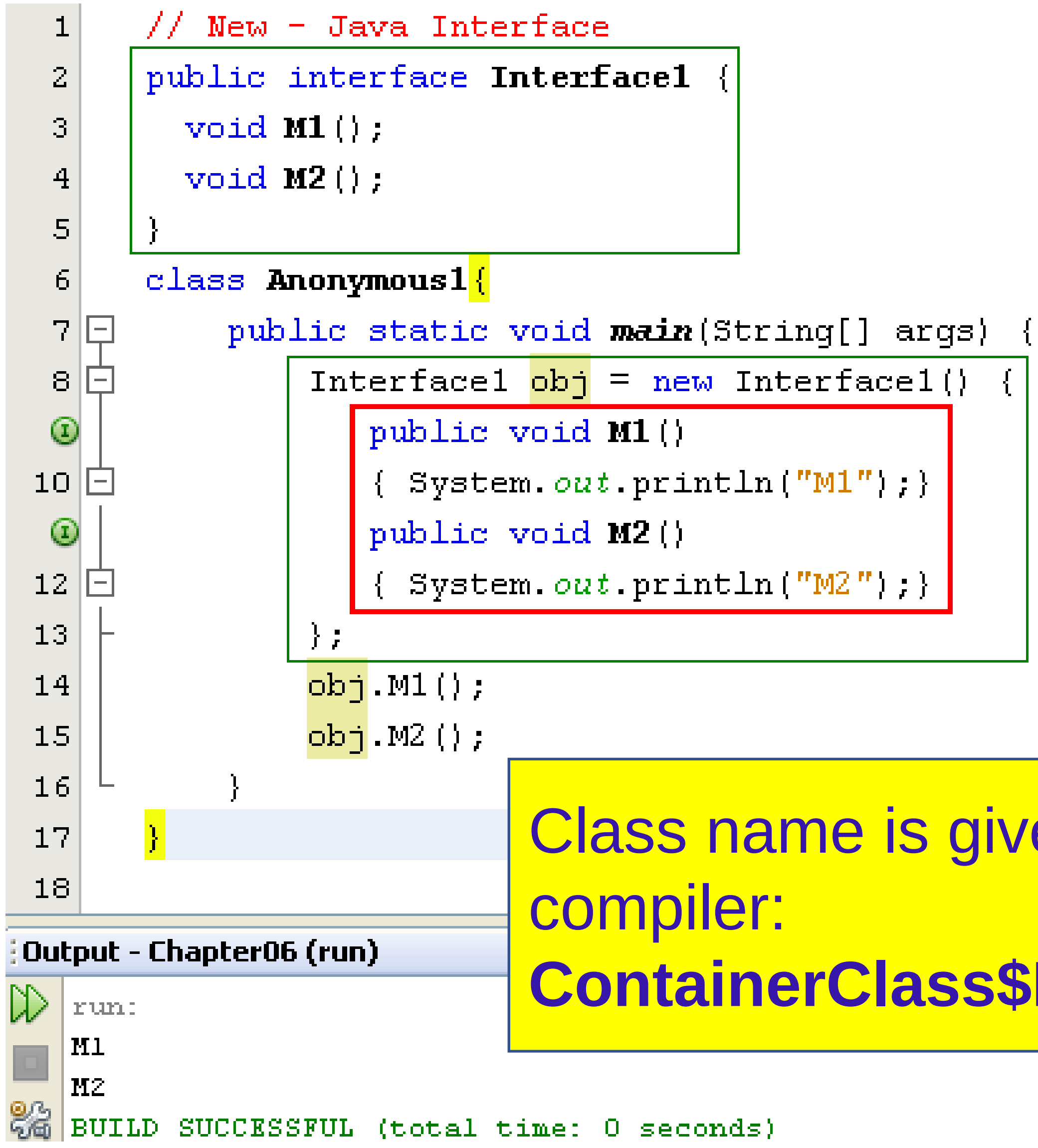
Anonymous classes are classes which are not named but they are identified automatically by Java compiler.

Where are they? They are identified at initializations of interface/abstract class object but abstract methods are implemented as attachments.

Why are they used?

- Enable you to make your code more concise.
- Enable you to declare and instantiate a class at the same time.
- They are like local classes except that they do not have a name.
- Use them if you need to use a local class only once.

Anonymous Class



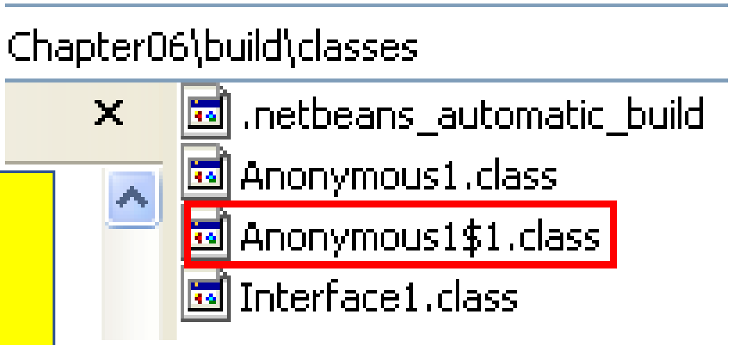
```
// New - Java Interface
public interface Interface1 {
  void M1();
  void M2();
}
class Anonymous1{
    public static void main(String[] args) {
        Interface1 obj = new Interface1() {
          public void M1()
          { System.out.println("M1");}
          public void M2()
          { System.out.println("M2");}
        };
        obj.M1();
        obj.M2();
    }
}
```

Output - Chapter06 (run)

```
run:
M1
M2
BUILD SUCCESSFUL (total time: 0 seconds)
```

Anonymous class.

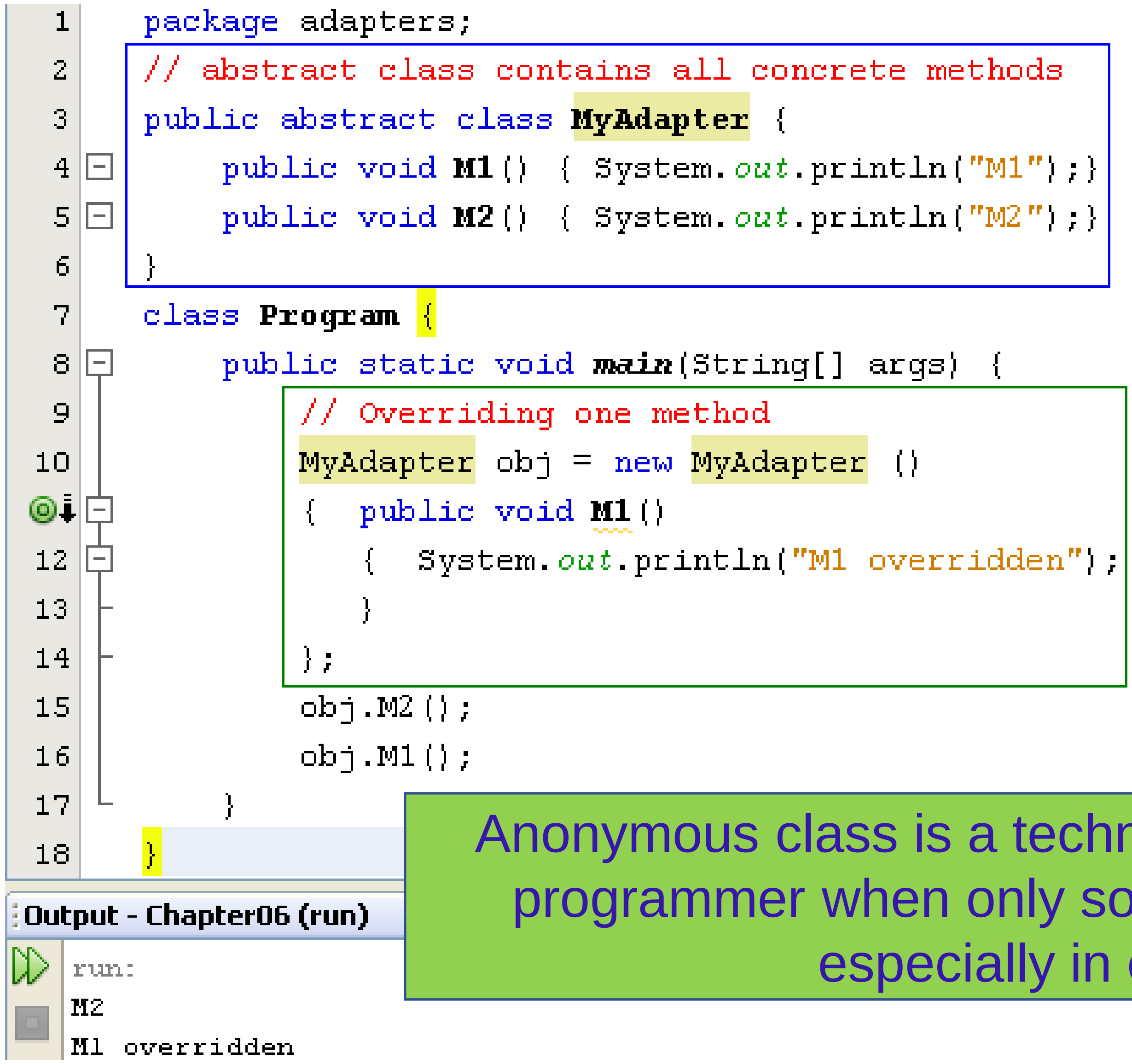
Class name is given by the compiler:
ContainerClass$Number



Chapter06\build\classes

```
.netbeans_automatic_build
Anonymous1.class
Anonymous1$1.class
Interface1.class
```

Anonymous Class…



```java
package adapters;
// abstract class contains all concrete methods
public abstract class MyAdapter {
    public void M1() { System.out.println("M1");}
    public void M2() { System.out.println("M2");}
}
class Program {
    public static void main(String[] args) {
        // Overriding one method
        MyAdapter obj = new MyAdapter ()
        { public void M1()
            { System.out.println("M1 overridden");
            }
        };
        obj.M2();
        obj.M1();
    }
}
```

Output - Chapter06 (run)

```
run:
M2
M1 overridden
```

Concrete methods but they can not be used because the class is declared as abstract one.

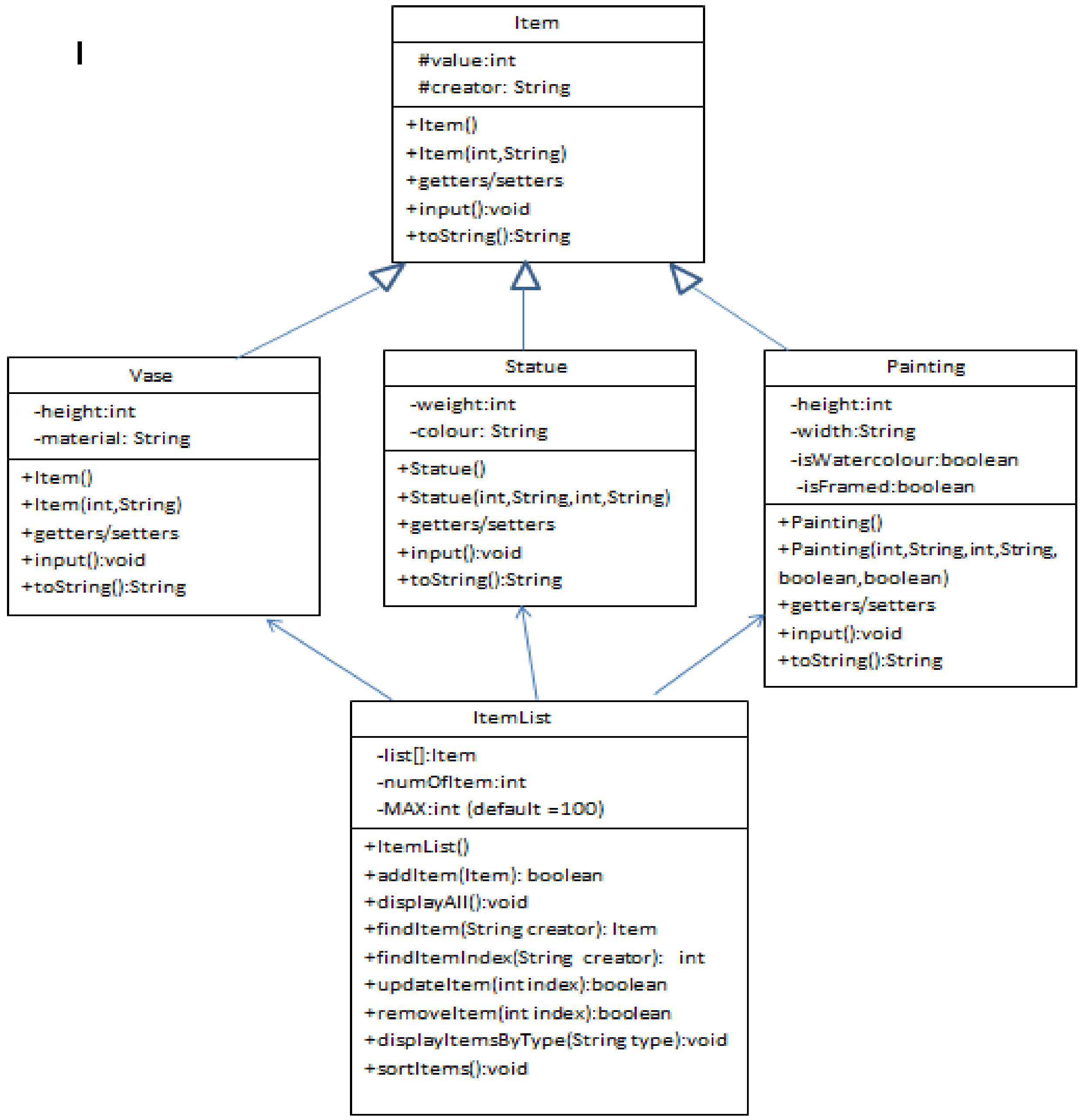
The abstract class can be used only when at least one of it’s methods is overridden

Anonymous class is a technique is commonly used to support programmer when only some methods are overridden only especially in event programming.

Case study

- A antique shop that sells antique items, namely vases, statues, and paintings. The owner can add item to inventory. The shop will keep items in the list. The owner can add a new item to it, he search also the item,….
- =>For now, we want to manage the list of objects such as vases, statues, paintings in an array or a list.

I


Item
#value:int
#creator: String
+Item()
+Item(int,String)
+getters/setters
+input():void
+toString():String
Vase
-height:int
-material: String
+Item()
+Item(int,String)
+getters/setters
+input():void
+toString():String
Statue
-weight:int
-colour: String
+Statue()
+Statue(int,String,int,String)
+getters/setters
+input():void
+toString():String
Painting
-height:int
-width:String
-isWatercolour:boolean
-isFramed:boolean
+Painting()
+Painting(int,String,int,String,
boolean,boolean)
+getters/setters
+input():void
+toString():String
ItemList
-list[]:Item
-numOfItem:int
-MAX:int (default =100)
+ItemList()
+addItem(Item): boolean
+displayAll():void
+findItem(String creator): Item
+findItemIndex(String creator): int
+updateItem(int index):boolean
+removeItem(int index):boolean
+displayItemsByType(String type):void
+sortItems():void

Summary

- Object-oriented languages implement reusability of coding structure through inheritance
- A derived class does not by default inherit the constructor of a super class
- Constructors in an inheritance hierarchy execute in order from the super class to the derived class
- Using the instanceof keyword if we need to check the type of the reference variable.
- Check the type of the reference variable before casting it explicitly.

- Polymorphism is a concept of object-oriented programming
- Polymorphism is the ability of an object to take on many forms
- Overloading and overriding are a technology to implement polymorphism feature.
- In OOP occurs when a parent class/ interface reference is used to refer to a child class object