Polymorphism

Objectives

- Overloading and Overriding
- Interface
- Abstract class

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.

Polymorphism

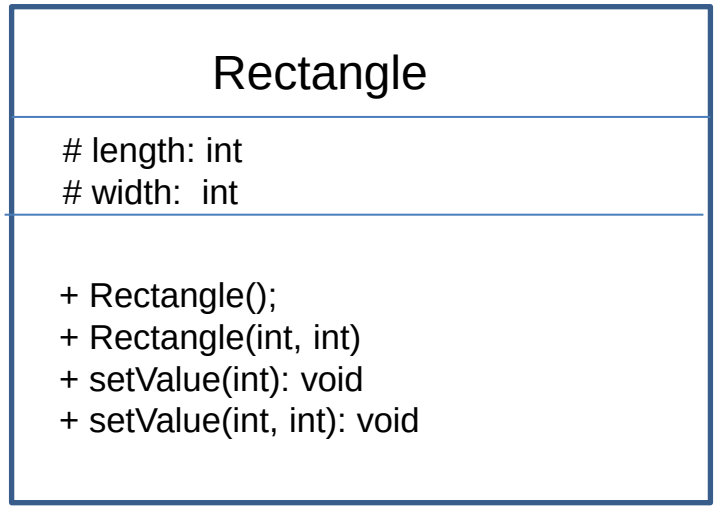
Overloading: A class can have some methods which have the same name but their parameter types are different.

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

- *Overloading* addresses variations in a function's signature.
- Overloading allows binding of function calls with the same identifier but different argument types
- The compiler binds the function call to the matching function definition .

Overloading



- overloading with constructors

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

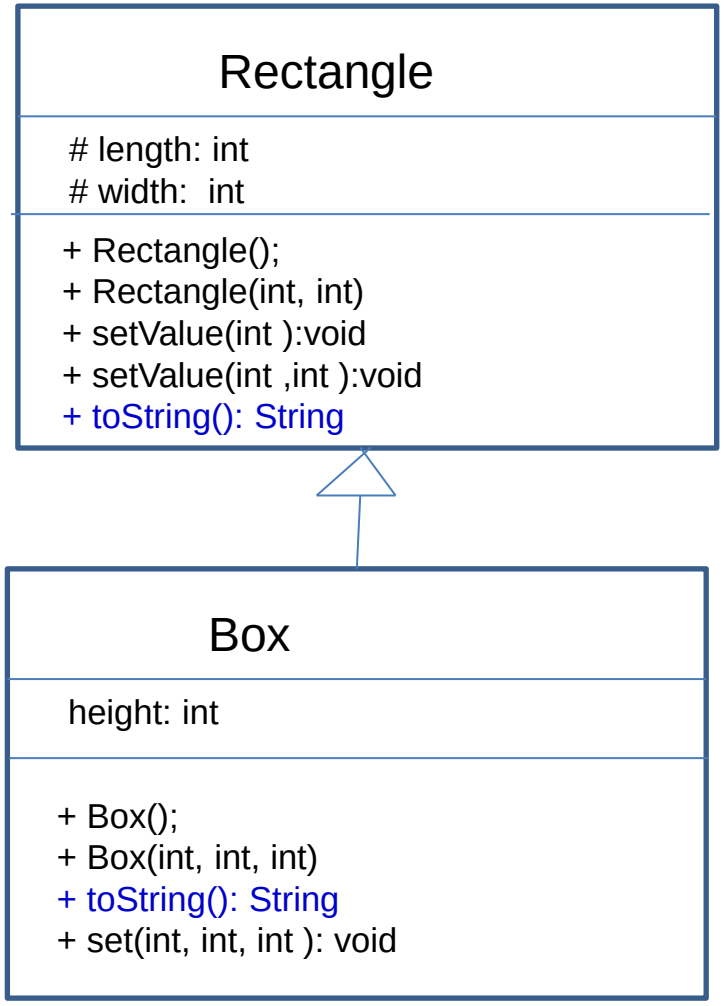
- Overloading also extends to general methods.

```
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.

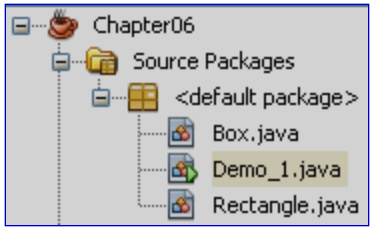
Overriding




```
public class Rectangle{
    …
    @override
     public String toString(){
        return length+”@” +width;
     }
}
public class Box extends Rectangle{
     …
     @override
     public String toString(){
        return super.toString()+”@”+ height;
     }
}
```

Overriding Inherited Methods

Overridden method: An inherited method is re-written




```
1   public class Box extends Rectangle {
2     int height=0;
3     public void set (int l, int w, int h)
4     {  super.setValue(l, w);
5        height = h>0? h: 0;
6     }
7     // Overriding the toString method
8     // of the Rectangle class
      public String toString()
10    { return "[" + length + "," + width +
11                  "," + height + "]";
12    }
13  }
```

```
1   public class Rectangle {
2      protected int length=0, width=0;
3      // Overloading methods
4      public void setValue(int l)
5      {  length = l>0?l:0; }
6      public void setValue(int l, int w)
7      {  length = l>0? l: 0;
8         width= w>0? w: 0;
9      }
10     // Overriding the toString method of the java.lang.Object class
       public String toString()
12     {  return "[" + length + "," + width + "]}";
13     }
14  }
15
```

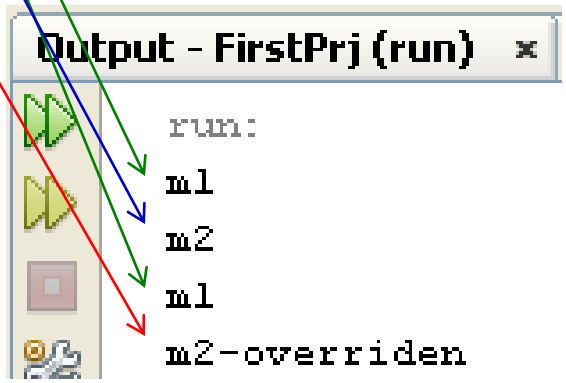
```
Output - Chapter06 (run)
run:
[5,0]}
[10,20]}
[5,10,15]
```

```
public class Demo_1 {
  public static void main  (String[] args)
  {  Rectangle r= new Rectangle();
     r.setValue(5);
     System.out.println(r.toString());
     r.setValue(10,20);
     System.out.println(r.toString());
     Box b= new Box();
     b.set(5,10,15);
     System.out.println(b.toString());
  }
}
```

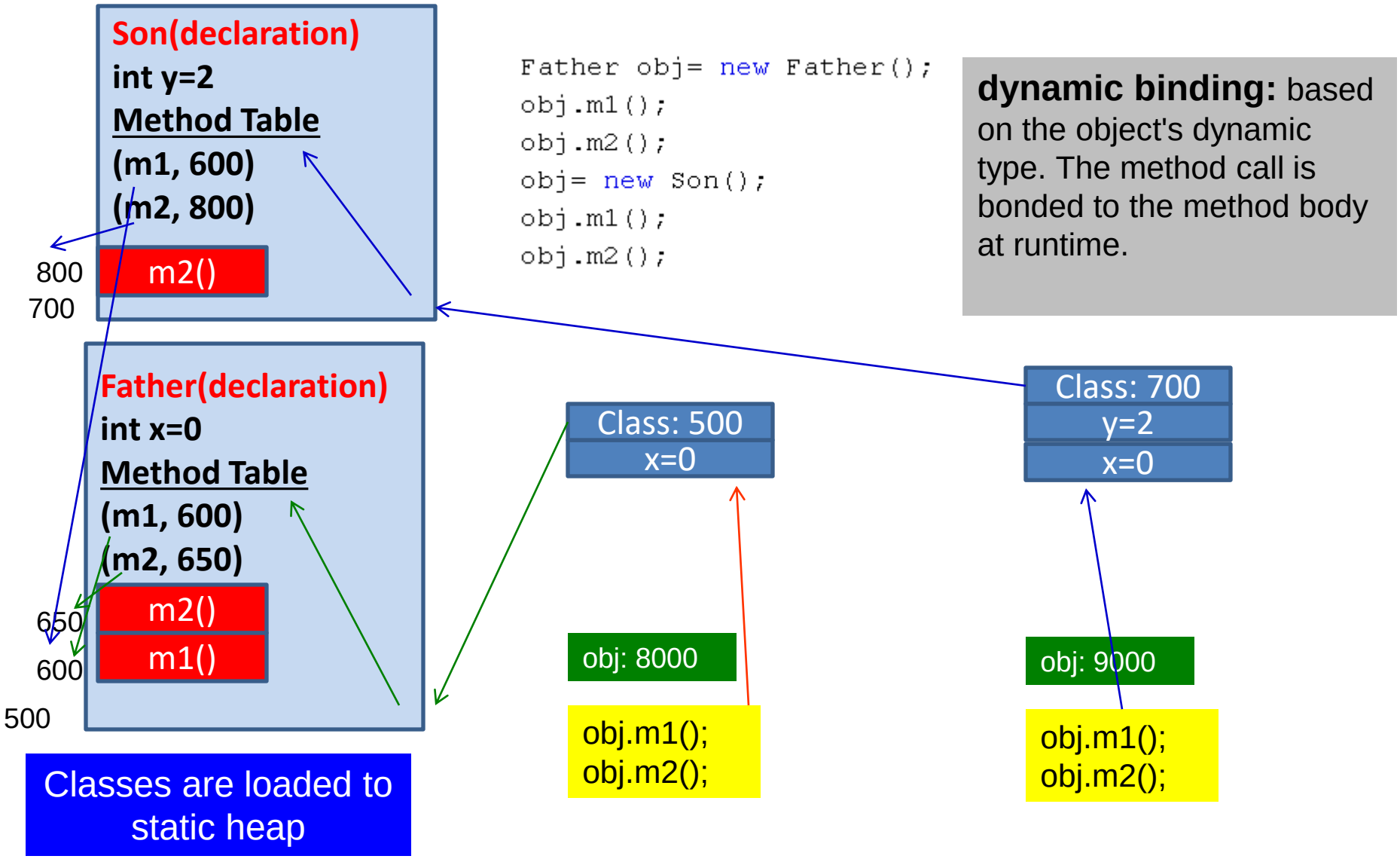
Overloaded methods: Methods have the same name but their parameters are different in a class

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();
    }
}
```



How Can Overridden Methods be Determined?



Interface

- An *interface* is a reference type, similar to a class, that can contain *only* constants, initialized fields, static methods, prototypes (abstract methods), default 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

Example: how to create an interface

```
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

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

```
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();

    }
}
```

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

Example: how to extend interfaces

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

```
1   package shapes;
2   public abstract class Shape {
3       abstract public double circumstance();
4       abstract public double area();
5   }
6   class Circle extends Shape {
7       double r;
8       public Circle (double rr) { r=rr; }
9       public double circumstance() { return 2*Math.PI*r; }
10      public double area() { return Math.PI*r*r; }
11  }
12  class Rect extends Shape {
13      double l,w;
14      public Rect(double ll, double ww){
15          l = ll; w = ww;
16      }
17      public double circumstance() { return 2*(l+w); }
18      public double area() { return l*w; }
19  }
20  class Program {
21      public static void main(String[] args) {
22          Shape s = new Shape ();
23      }
24  }
```

```
20  class Program {
21      public static void main(String[] args) {
22          Shape s = new Circle(5);
23          System.out.println(s.area());
24      }
25  }
```

Modified

```
Output - Chapter06 (run)
run:
78.53981633974483
```

Abstract Classes

```
1   public abstract class AbstractDemo2 {
2       void m1() // It is not abstract class
3       { System.out.println("m1");
4       }
5       void m2() // It is not abstract class
6       { // empty body
7       }
8       public static void main(String[] args)
9       {   AbstractDemo2 obj = new AbstractDemo2();
10      }
11  }
```

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…

Error. Why?

```
   public abstract class AbstractDemo2 {
      void m1() // It is not abstract class
3     { System.out.println("m1");
4     }
5     abstract void m2();
6  }
   class Derived extends  AbstractDemo2
   {  public void m1() // override
9     { System.out.println("m1");
10    }
11    public static void main(String[] args)
12    {  Derived obj = new Derived();
13    }
14 }
```

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***

Nested Class(optional)

- Why use nested class?
- Types:
  - Local class
  - Anonymous class
  - …
- Syntax of nested class
- Examples

Reference:

- Nested Classes (The Java™ Tutorials > Learning the Java Language > Classes and Objects) (oracle.com)
- Anonymous Classes (The Java™ Tutorials > Learning the Java Language > Classes and Objects) (oracle.com)

Why use nested class? (optional)

- It is a way of logically grouping classes that are only used in one place: If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together.
- It increases encapsulation
- It can lead to more readable and maintainable code: Nesting small classes within top-level classes places the code closer to where it is used.
- Syntax:

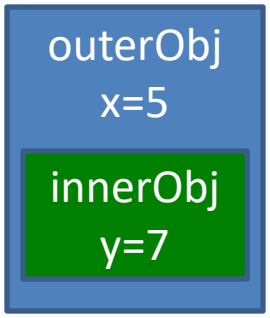
```
class OuterClass {
    ...
    class InnerClass {
        ...
    }
    static class StaticNestedClass {
        ...
    }
}
```

Nested classes are divided into two categories: non-static(inner class/local class) and static.

Inner/local class(optional)

- Local classes are classes that are defined in a *block*, which is a group of zero or more statements between balanced braces. You typically find local classes defined in the body of a method.
- [demo]

Inner class(optional)



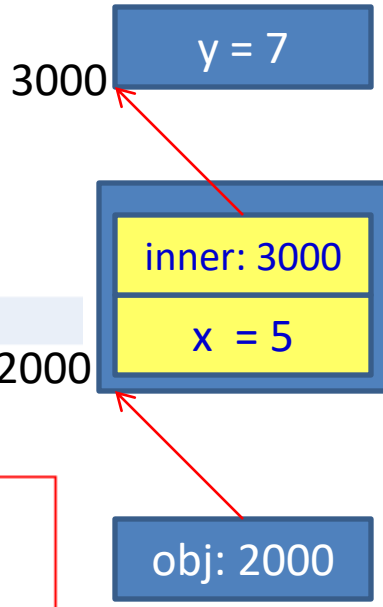
A method of nested class can access all members of its enclosing class

```
public class OuterClass {
    int x=5;
    class Inner1 {
        int y= 7;
        void M_Inner(){
            System.out.print(x+y);
        }
    }
    void M_Outer(){
        System.out.print(x+y);
    }
}
```

Error: inner obj do not created yet. Modified

```
1  public class OuterClass {
2      int x=5;
3      class Inner1 {
4          int y= 7;
5          Inner1(){
6              System.out.println(2*x+y);
7          }
8          void printInner(){
9              System.out.println(y);
10         }
11     }
12     Inner1 inner= new Inner1();
13     void M_Outer(){
14         System.out.println(x + inner.y);
15     }
16     public static void main(String[] args){
17         OuterClass obj= new OuterClass();
18         obj.M_Outer();
19     }
20 }
```

```
Output - Chapter06 (run)
run:
17
12
```



Anonymous Classes(optional)

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.

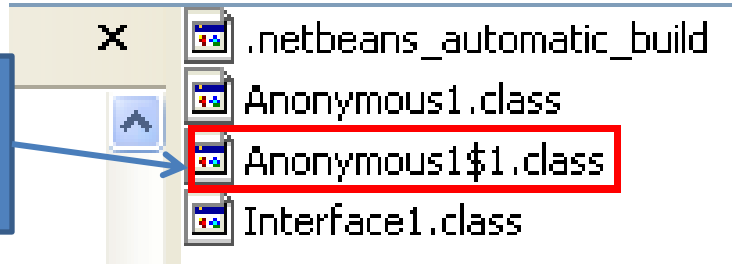
Anonymous Class(optional)

```
1   // New - Java Interface
2   public interface Interface1 {
3     void M1();
4     void M2();
5   }
6   class Anonymous1{
7       public static void main(String[] args) {
8           Interface1 obj = new Interface1() {
10              public void M1()
                { System.out.println("M1");}
12              public void M2()
                { System.out.println("M2");}
13          };
14          obj.M1();
15          obj.M2();
16      }
17  }
18
```

Anonymous class.

Class name is given by the compiler: **ContainerClass$Number**

Chapter06\build\classes




```
Output - Chapter06 (run)
run:
M1
M2
BUILD SUCCESSFUL (total time: 0 seconds)
```

Anonymous Class…(optional)

```
1  package adapters;
2  // abstract class contains all concrete methods
3  public abstract class MyAdapter {
4      public void M1() { System.out.println("M1");}
5      public void M2() { System.out.println("M2");}
6  }
7  class Program {
8      public static void main(String[] args) {
9          // Overriding one method
10         MyAdapter obj = new MyAdapter ()
           {  public void M1()
12             {  System.out.println("M1 overridden");
13             }
14         };
15         obj.M2();
16         obj.M1();
17     }
18 }
```

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

```
Output - Chapter06 (run)
run:
M2
M1 overridden
```

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

Summary

- 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