

Chapter 7

Classes



Declaration

- These slides are made for UIT, BU students only. I am not holding any copy write of it as I had collected these study materials from different books and websites etc. I have not mentioned those to avoid complexity.



Topics

- Method Overloading
- Using objects as Parameters
- Argument passing to methods
- Call by value
 - Call by reference
 - Returning Object
- Recursion
- Visibility Control
 - Visibility of fields in a class
- Static
- final
- Nested and inner classes
- String class



Method Overloading

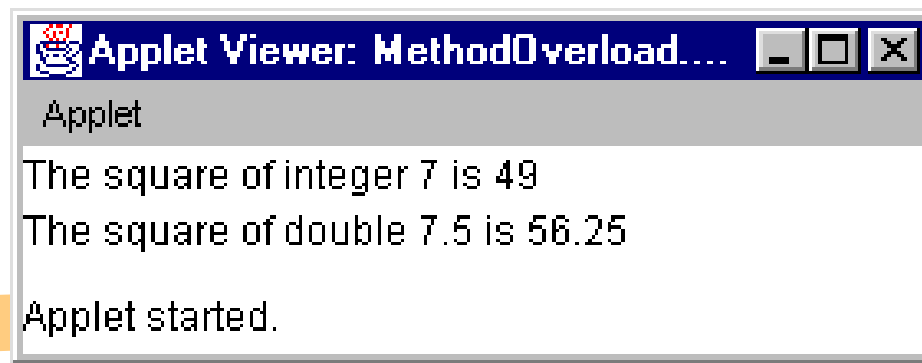
■ Method overloading

- Methods with same name and different parameters
- Overloaded methods should perform similar tasks
 - ▶ Method to square `ints` and method to square `doubles`

```
public int square( int x ) { return x * x; }  
public double square( double x ) { return x * x; }
```

- Program calls method by signature
 - ▶ Signature determined by method name and parameter types
 - ▶ Overloaded methods must have different parameters
 - Return type cannot distinguish method

```
1// MethodOverload.java
2// Using overloaded methods
3import java.awt.Container;
4import javax.swing.*;
5
6public class MethodOverload extends JApplet {
7    JTextArea outputArea;
8
9    public void init()
10    {
11        outputArea = new JTextArea( 2, 20 );
12        Container c = getContentPane();
13        c.add( outputArea );
14
15        outputArea.setText(
16            "The square of integer 7 is " + square( 7 ) +
17            "\nThe square of double 7.5 is " + square( 7.5 ) );
18    }
19
20    public int square( int x )
21    {
22        return x * x;
23    }
24
25    public double square( double y )
26    {
27        return y * y;
28    }
29}
```



Program Output



Method Overloading

- When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters.
- However this match need not always be exact. In some case Java's **automatic type conversions** can play a role in overload resolution.

```
void test(double a) {  
    System.out.println("Inside test(double) a: " + a);  
}
```

- Input

```
ob.test(88);
```

```
ob.test(123.2);
```

- Output

```
Inside test(double) a: 88.0
```

```
Inside test(double) a: 123.2
```



Using objects as Parameters

```
class Test {  
    int a, b;  
    Test(int i, int j) {  
        a = i;  
        b = j;  
    }  
    // return true if o is equal to the invoking object  
    boolean equals(Test o) {  
        if(o.a == this.a && o.b == this.b) return true;  
        else return false;  
    }  
}
```




Using objects as Parameters

```
class PassOb {  
    public static void main(String args[]) {  
        Test ob1 = new Test(100, 22);  
        Test ob2 = new Test(100, 22);  
        Test ob3 = new Test(-1, -1);  
        System.out.println("ob1 == ob2: " + ob1.equals(ob2));  
        System.out.println("ob1 == ob3: " + ob1.equals(ob3));  
    }  
}
```

■ Output

ob1 == ob2:true

ob1 == ob3:false

Argument passing to methods



- A computer language can pass argument to a subroutine in two ways
 - Call by value
 - Call by reference



Call by value

- This method **copies** the value of an argument into the formal parameter of the subroutine.
- Changes made to the parameter of the subroutine have **no effect** on the argument used to call it.
- **Simple types** are passed by value.

// Simple Types are passed by value.

```
class Test {  
    void meth(int i, int j) {  
        i *= 2;  
        j /= 2;  
    }  
}
```



Call by value

```
class CallByValue {  
    public static void main(String args[]) {  
        Test ob = new Test();  
        int a = 15, b = 20;  
        System.out.println("a and b before call: " + a + " " + b);  
        ob.meth(a, b);  
        System.out.println("a and b after call: " + a + " " + b);  
    }  
}
```

■ Output

a and b before call: 15 20

a and b after call: 15 20



Call by reference

- A reference to an argument (not the value of the argument) is passed to the parameter.
- Inside the subroutine, this reference is used to access the argument specified in the call.
- Changes made to the parameter **will affect** the argument used to call the subroutine.
- **Objects are passed by reference.**



Call by reference

// Objects are passed by reference.

```
class Test {  
    int a, b;  
    Test(int i, int j) {  
        a = i;  
        b = j;  
    }  
    // pass an object  
    void meth(Test o) {  
        o.a *= 2;  
        o.b /= 2;  
    }  
}
```



Call by reference

```
class CallByRef {  
    public static void main(String args[]) {  
        Test ob = new Test(15, 20);  
        System.out.println("ob.a and ob.b before call: " + ob.a + " " +  
            ob.b);  
        ob.meth(ob);  
        System.out.println("ob.a and ob.b after call: " + ob.a + " " +  
            ob.b);  
    }  
}
```

■ Output

ob.a and ob.b before call: 15 20

ob.a and ob.b after call: 30 10

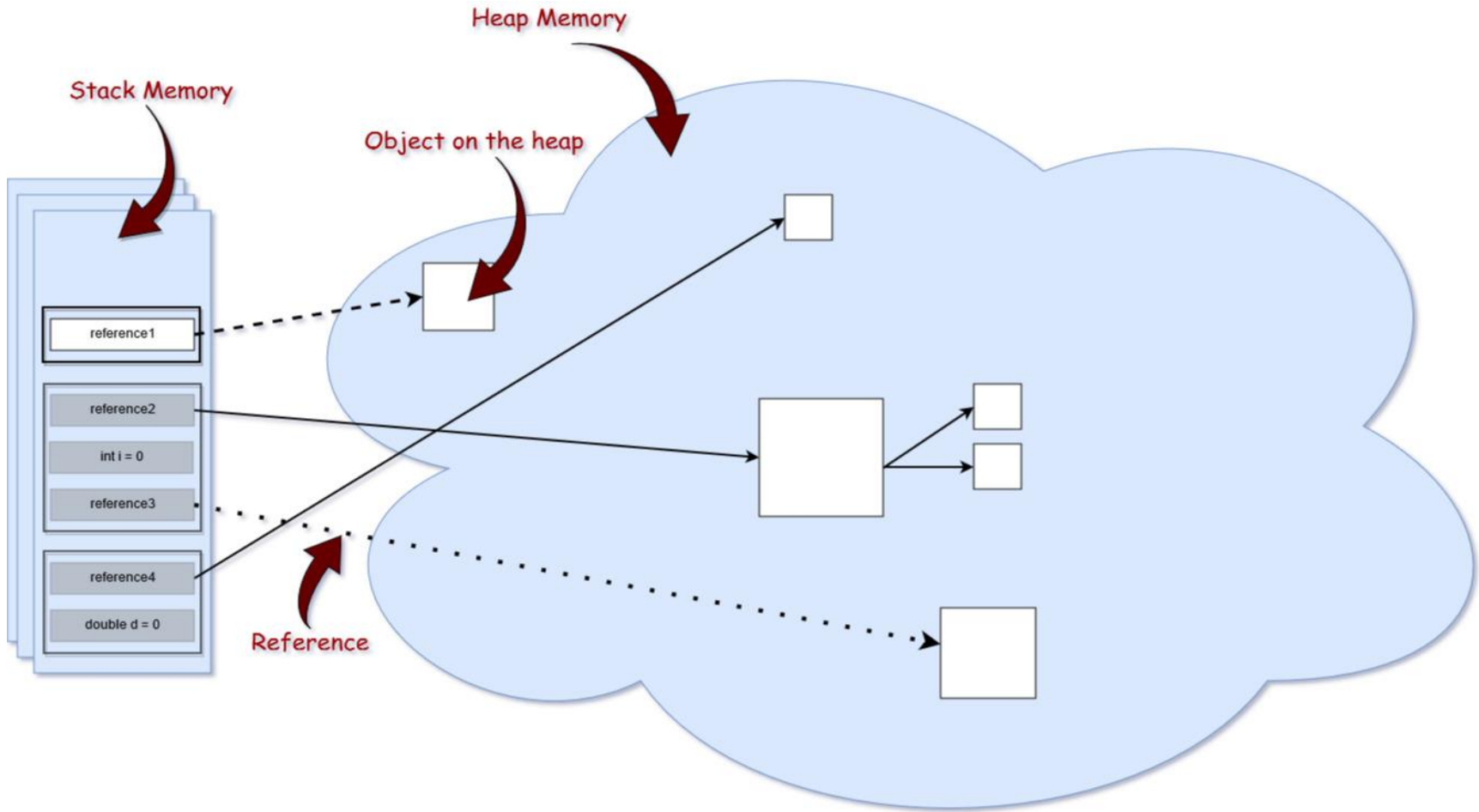


Memory Management

- Generally, memory is divided into two big parts: the **stack** and the **heap**.
- The heap is a huge amount of memory compared to the stack.



Memory Management





Memory Management

- Stack memory is responsible for holding references to heap objects and for storing value types (also known in Java as primitive types), which hold the value itself rather than a reference to an object from the heap.
- In addition, variables on the stack have a certain visibility, also called **scope**. Only objects from the active scope are used. For example, assuming that we do not have any global scope variables (fields), and only local variables, if the compiler executes a method's body, it can access only objects from the stack that are within the method's body. It cannot access other local variables, as those are out of scope. Once the method completes and returns, the top of the stack pops out, and the active scope changes.



Memory Management

- Maybe you noticed that in the picture above, there are multiple stack memories displayed. This is due to the fact that the **stack memory in Java is allocated per Thread**. Therefore, each time a Thread is created and started, it has its own stack memory — and cannot access another thread's stack memory.
- Heap of memory stores the actual object in memory. Those are referenced by the variables from the stack.
- There exists only one heap memory for each running JVM process. Therefore, this is a shared part of memory regardless of how many threads are running. Actually, the heap structure is a bit different than it is shown in the picture above. The heap itself is divided into a few parts, which facilitates the process of garbage collection.

Returning Object



// Returning an object.

```
class Test {
```

```
    int a;
```

```
    Test(int i) {
```

```
        a = i;
```

```
    }
```

```
    Test incrByTen() {
```

```
        Test temp = new Test(a+10);
```

```
        return temp;
```

```
    }
```

```
}
```



Returning Object

```
class RetOb {  
    public static void main(String args[]) {  
        Test ob1 = new Test(2);  
        Test ob2;  
        ob2 = ob1.incrByTen();  
        System.out.println("ob1.a: " + ob1.a);  
        System.out.println("ob2.a: " + ob2.a);  
        ob2 = ob2.incrByTen();  
        System.out.println("ob2.a after second increase: " + ob2.a);  
    }  
}
```

■ Output

```
ob1.a: 2  
ob2.a: 12  
ob2.a after second increase: 22
```



Recursion

■ Example: factorial

- $5! = 5 * 4 * 3 * 2 * 1$

Notice that

$$5! = 5 * 4!$$

$$4! = 4 * 3! \dots$$

$$n! = n * (n-1) !$$

- Can compute factorials recursively
- Solve base case ($1! = 0! = 1$) then plug in

↓

$$2! = 2 * 1! = 2 * 1 = 2;$$

↘

$$3! = 3 * 2! = 3 * 2 = 6;$$

↘

$$4! = 4 * 3! = 4 * 6 = 24$$

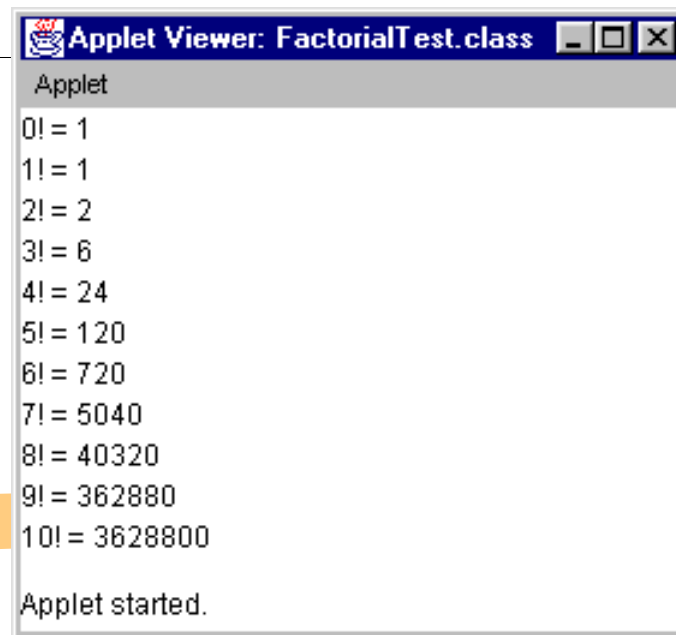
```

1// Fig. 6.12: FactorialTest.java
2// Recursive factorial method
3import java.awt.*;
4import javax.swing.*;
5
6public class FactorialTest extends JApplet {
7    JTextArea outputArea;
8
9    public void init()
10    {
11        outputArea = new JTextArea();
12
13        Container c = getContentPane();
14        c.add( outputArea );
15
16        // calculate the factorials of 0 through 10
17        for ( long i = 0; i <= 10; i++ )
18            outputArea.append(
19                i + "! = " + factorial( i ) + "\n" );
20    }
21
22    // Recursive definition of method factorial
23    public long factorial( long number )
24    {
25        if ( number <= 1 ) // base case
26            return 1;
27        else
28            return number * factorial( number - 1 );
29    }
30}

```

Method **factorial** keeps calling itself until the base case is solved.
Uses formula:

$$n! = n * (n-1)!$$



```
Applet Viewer: FactorialTest.class
Applet
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
Applet started.
```

Program Output



Visibility Control

- Encapsulation provides another important attribute: **access control**.
- How a member can be accessed is determined by the **access specifier**.
- Some aspect of visibility control are related mostly to inheritance or **packages (a grouping of classes)**.
- Java access specifies are **public, private and protected**.
- Java also defines a default access level.
- Protected applies only when **inheritance** is involved.



Visibility Control

- When a member of a class is modified by the **public** specifier, then that member can be accessed by any other code in your program.
- When a member of a class is specified as **private** then that member can only be accessed by other members of its class.
- When no access specifier is used, then by **default** the member of a class is public within its own package, but cannot be accessed outside of its package.
- The difference between the public access specifier and default access specifier i.e. when we are not mentioning any access specifier, the public access specifier makes fields visible in all classes, regardless of their packages while the default access specifier makes fields visible only in the same package, but not in the other packages.



Visibility Control

- The **protected** modifier makes the fields visible not only to all classes and subclasses in the same package but also to the subclasses in other packages.
- **Private protected** modifier marks the fields visible in all subclasses regardless of what package they are in. Remember these fields are not accessible by other classes in the same package.
- The 1.0 release of the Java language supported five access levels: the four listed above plus private protected. **The private protected access level is not supported in versions of Java higher than 1.0; you should no longer be using it in your Java programs.**



Visibility Control

`/* This program demonstrates the difference between public and private.*/`

```
class Test {  
    int a; // default access  
    public int b; // public access  
    private int c; // private access  
    // methods to access c  
    void setc(int i) { // set c's value  
        c = i;  
    }  
    int getc() { // get c's value  
        return c;  
    }  
}
```



Visibility Control

```
class AccessTest {  
    public static void main(String args[]) {  
        Test ob = new Test();  
  
        // These are OK, a and b may be accessed directly  
        ob.a = 10;  
        ob.b = 20;  
  
        // This is not OK and will cause an error  
        // ob.c = 100; // Error!  
  
        // You must access c through its methods  
        ob.setc(100); // OK  
  
        System.out.println("a, b, and c: " + ob.a + " " +  
                           ob.b + " " + ob.getc());  
    }  
}
```



Visibility of fields in a class

Access Modifier/ Access location	public	protected	default	Private protected	private
Same class	Yes	Yes	Yes	Yes	Yes
Subclass in same package	Yes	Yes	Yes	Yes	No
Other classes in same package	Yes	Yes	Yes	No	No
Subclass in other packages	Yes	Yes	No	Yes	No
Non-subclasses in other packages	Yes	No	No	No	No



Static

- When a member is declared as static, it can be **accessed before any objects of its class are created**, and without reference to any object.
- Both methods and variables can be declared as static.
- Most common one is **main()**.
- Instance variables declared as static are **global/class variables**.
- When objects of its class are declared, **no copy** of a static variables are made.
- All instance of the class **share** the same static variable.
- Methods declared as static have several **restrictions**
 - They can only call other static methods.
 - They can only access static data.
 - They cannot refer to **this** or **super** in any way.



Static block

- Static block gets executed exactly one, when the class is first loaded.

// Demonstrate static variables, methods, and blocks.

```
class UseStatic {  
    static int a = 3;  
    static int b;  
    static void meth(int x) {  
        System.out.println("x = " + x);  
        System.out.println("a = " + a);  
        System.out.println("b = " + b);}  
    static {  
        System.out.println("Static block initialized.");  
        b = a * 4;}  
    public static void main(String args[]) {  
        meth(42);}}
```




Static

- If you want to call static method from outside of the class you can do this by

classname.method();

- In case of static variables

Classname.staticVariable;



final

- final is similar to const in C/C++;
- The keyword final can be used to methods.

final int FILE_NEW = 1;

final int FILE_OPEN = 2;

final int FILE_SAVE = 3;

final int FILE_SAVEAS = 4;

final int FILE_QUIT = 5;



Nested and inner classes

- When a class is defined within another class, that class is known as nested classes.
- The scope of the nested class is bounded by the scope of its enclosing class.
- A nested class has access to the members, including private members, of the class in which it is nested.
- However, the enclosing class does not have access to the members of the nested class.



Nested and inner classes

- Two types of nested class
 - static
 - Non-static
- In static nested class static modifier applied.
- It must access the members of its enclosing class directly.
- This means that you can access the static nested class without creating an instance of the outer class.
- Static nested classes are often used to group related utility methods or constants together within a class.



Nested and inner classes

```
public class OuterClass
{ // Static nested class
    public static class StaticNestedClass
    {
        public void displayMessage()
        {
            System.out.println("This is a static nested class.");
        }
    }
    public static void main(String[] args)
    { // You can create an instance of the static nested class
        OuterClass.StaticNestedClass nestedObject = new
        OuterClass.StaticNestedClass(); // Call the method of the static
        nested class
        nestedObject.displayMessage();
    }
}
```



Nested and inner classes

- Inner classes are non-static nested class.
- Inner classes are particularly helpful when handling events in an applet.
- When inner class don't have name, that type of nested class is called **anonymous inner classes**.



String class

- String is probably most commonly used class in java class library.
- String are **immutable**; once a String object is created, its contents cannot be altered.
- This is not a serious restriction due to two reasons:
 - If you need to change a string, you can always create a new one that contains the modifications.
 - Java defines a peer class of String, called StringBuffer, which allows strings to be altered.
- + operator used to concatenate two strings.
- String class contains several methods,
 - equals()
 - length()
 - charAt()

End of Chapter 7

Questions?