CS6308- Java Programming

V P Jayachitra
Assistant Professor
Department of Computer Technology
MIT Campus
Anna University

MODULE II JAVA OBJECTS -1	L	T	P	EL
	3	0	4	3
Classes and Objects, Constructor, Destructor, Static instances, this, constants, Thinking in Objects, String class, Text I/O				
SUGGESTED ACTIVITIES : • Flipped classroom • Practical - Implementation of Java programs – using String class, Creating Classes and objects • EL – Thinking in Objects				
SUGGESTED EVALUATION METHODS: • Assignment problems • Quizzes				

Static

- **Why static member?**
  - A class member must be accessed only in conjunction with an object of its class.
  - Is it possible to create a member that can be used by itself, without reference to a specific instance?
  - Yes, static class member will be used independently of any object of that class.

What is static member?

- A member that is declared static can be accessed before any objects of its class are created, and without reference to any object.
- To create such a member, precede its declaration with the keyword static.
- Both methods and variables can be declared to be static.
- Example : main()method
  - main( ) is declared as static
  - Main() should be called before any objects exist.

Static variables

- Instance variables declared as static are, essentially, **global variables**.
- When objects of its class are declared, all instances of the class share the same static variable.
- Static variables are used independently of any object using class name followed by the dot operator.
  - classname.staticVariable

Static methods

- Methods declared as static are, essentially, **global methods**.
- Methods declared as static have several restrictions:
  - They can only directly call other static methods of their class.
  - They can only directly access static variables of their class.
  - They cannot refer to this or super in any way. (super relates to inheritance).
  - They are used independently of any object using class name followed by the dot operator.
    - classname.method( )

Static block

- Static block is used to initialize your static variables.

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

```
class StaticDemo {
  static int a = 42;
  static int b = 99;

  static void callme() {
    System.out.println("a = " + a);
  }
}

class StaticByName {
  public static void main(String args[]) {
    StaticDemo.callme();
    System.out.println("b = " + StaticDemo.b);
  }
}
```

Here is the output of this program:

```
a = 42
b = 99
```

Introducing final

- A field can be declared as constant using final.
- A final field must to initialized when it is declared, it can’t be altered.
  - Do final field initialization in one of two ways:
    - First, give it a value when it is declared.
    - Second, assign it a value within a constructor.
    - final float PI= 3.14;
    - common coding convention is to choose all uppercase identifiers for final fields.

```java
public class Final {
    public void final_varaible_method(){
        final int a=1;
        System.out.println(a++);
    }

    public static void main(String []argh){
        Final f=new Final();
        f.final_varaible_method();
    }
}
```

java: cannot assign a value to final variable a

```java
// 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);
  }
}
```

Output:
Static block initialized.
x = 42
a = 3
b = 12

```
class StaticDemo {
  static int a = 42;
  static int b = 99;

  static void callme() {
    System.out.println("a = " + a);
  }
}

class StaticByName {
  public static void main(String args[]) {
    StaticDemo.callme();
    System.out.println("b = " + StaticDemo.b);
  }
}
```

Here is the output of this program:

```
a = 42
b = 99
```

Introducing Nested and Inner Classes

- Define a class within another class and such classes are known as nested classes.
- The scope of a nested class is bounded by the scope of its enclosing class.
  - class B defined within class A does not exist independently of A.
- A nested class has access to the members, including private members, of the class in which it is nested.
- The enclosing class does not have access to the members of the nested class.
- A nested class that is declared directly within its enclosing class scope is a member of its enclosing class.
- A nested class also be declared that is local to a block.

```
public class EnclosingClass {
  . . .
  public class NestedClass {
  . . .  }
}
```

Introducing Nested and Inner Classes

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

- A nested class is a member of its enclosing class.
- Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared private.
- Static nested classes do not have access to other members of the enclosing class.
- As a member of the OuterClass, a nested class can be declared private, public, protected

Why nested class?

- A nested class is a class that’s tightly coupled with the class in which it’s defined.

- A nested class has access to the private data within its enclosing class

Why Use Nested Classes?

- Compelling reasons for using nested classes include the following:
  - 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:
    - Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private.
    - By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.
  - 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.

Nested classes-static nested class

- There are two types of nested classes:
  - Static
  - non-static.

- A static nested class is one that has the static modifier applied.
- Static class must access the non-static members of its enclosing class through an object.
- Static class cannot refer to non-static members of its enclosing class directly.
  - Because of this restriction, static nested classes are seldom used
- Note:Top level class can not be declared as static. Only nested classes can be declared as static.

Nested classes-Inner class

- A non-static nested class is an inner class.
- Inner class has access to all of the variables and methods of its outer class.
- Inner class refer variables and methods directly
- in the same way that other non-static members of the outer class do.

```
class OuterClass {
...
 class InnerClass {
 ...
 }
}
```

To instantiate an inner class, you must first instantiate the outer class. Then, create the inner object within the outer object with this syntax:

OuterClass outerObject = new OuterClass();
OuterClass.InnerClass innerObject = outerObject.new InnerClass();

```
public class Manager extends Employee {
  private DirectReports directReports;
  public Manager() {
    this.directReports = new DirectReports();
  }
  . . .
  private class DirectReports {
  . . .
  }
}
```

Creating DirectReports instances: Second attempt

```
public class Manager extends Employee {
  public Manager() {
  }
  . . .
  public class DirectReports {
  . . .
  }
  }
public static void main(String[] args) {
  Manager manager = new Manager();
  Manager.DirectReports dr = manager.new DirectReports();
}
```

Creating DirectReports instances: First attempt

```
public class Manager extends Employee {
  public Manager() {
  }
  . . .
  public class DirectReports {
  . . .
  }
}
public static void main(String[] args) {
  Manager.DirectReports dr = new Manager.DirectReports();// This won't
work!
```

```
public class OuterClass {

    String outerField = "Outer field";
    static String staticOuterField = "Static outer field";

    class InnerClass {
        void accessMembers() {
            System.out.println(outerField);
            System.out.println(staticOuterField);
        }
    }

    static class StaticNestedClass {
        void accessMembers(OuterClass outer) {
 // Compiler error: Cannot make a static reference to the non-
static
            //     field outerField
            // System.out.println(outerField);
            System.out.println(outer.outerField);
            System.out.println(staticOuterField);
        }
    }
```

```
 public static void main(String[] args) {
        System.out.println("Inner class:");
        System.out.println("------------");
        OuterClass outerObject = new OuterClass();
        OuterClass.InnerClass innerObject = outerObject.new
InnerClass();
        innerObject.accessMembers();

        System.out.println("\nStatic nested class:");
        System.out.println("--------------------");
        StaticNestedClass staticNestedObject = new
StaticNestedClass();
        staticNestedObject.accessMembers(outerObject);

        System.out.println("\nTop-level class:");
        System.out.println("--------------------");
        TopLevelClass topLevelObject = new TopLevelClass();
        topLevelObject.accessMembers(outerObject);
    }
```

```
Inner class:
------------
Outer field
Static outer field

Static nested class:
--------------------
Outer field
Static outer field
```

Shadowing

```
public class ShadowTest {

    public int x = 0;

    class FirstLevel {

        public int x = 1;

        void methodInFirstLevel(int x) {
            System.out.println("x = " + x);
            System.out.println("this.x = " + this.x);
            System.out.println("ShadowTest.this.x = " + ShadowTest.this.x);
        }
    }

    public static void main(String... args) {
        ShadowTest st = new ShadowTest();
        ShadowTest.FirstLevel fl = st.new FirstLevel();
        fl.methodInFirstLevel(23);
    }
}
```

```
x = 23
this.x = 1
ShadowTest.this.x = 0
```

1.What is a nested class?

1. A class that provides some utility to other classes in the application
2. A class that is defined within another class
3. A class where one or more interface references are passed into its constructor
4. None of the above

1.What is a nested class?

1. A class that provides some utility to other classes in the application
2. **A class that is defined within another class**
3. A class where one or more interface references are passed into its constructor
4. None of the above

1. Why might you use a nested class?
    1. When you need to define a class that is tightly coupled (functionally) with another class
    2. When you want to write a class that makes use of a large number of interfaces from the JDK
    3. When one class needs to access to another class private data, but you have exceeded the maximum number of allowable classes for your application
    4. For a class has more than 20 methods defined on it and should be refactored

1.Why might you use a nested class?

1. **When you need to define a class that is tightly coupled (functionally) with another class**
2. When you want to write a class that makes use of a large number of interfaces from the JDK
3. When one class needs to access to another class private data, but you have exceeded the maximum number of allowable classes for your application
4. For a class has more than 20 methods defined on it and should be refactored

```
public class Outer {

 private static final Logger log = Logger.getLogger(Outer.class.getName());

 public void setInner(Inner inner) {
 this.inner = inner;
 }
 private Inner inner;

 public Inner getInner() {
 return inner;
 }

 private class Inner {
 }
  public static void main(String[] args) {
 Outer outer = new Outer();
 Inner inner = new Outer.Inner();
 outer.setInner(inner);
 log.info("Outer/Inner: " + outer.hashCode() + "/" + outer.getInner().hashCode());
 }

}
```

1.The class name `Outer` is not a legal Java class name.
2.The class name `Inner` is confusing.
3.The main method defined in class `Outer` has the wrong method signature.
4.The `log.info()` call in main() has too many parameters.
5.The class body for class `Inner` is empty.
6.None of the above. The line Inner inner = new Outer.Inner(); is in error. This is not the correct syntax for instantiating a nested class using an enclosing class reference. Rather, it should be: Inner inner = outer.new Inner();

```
public class Outer {

 private static final Logger log = Logger.getLogger(Outer.class.getName());

 public void setInner(Inner inner) {
 this.inner = inner;
 }
 private Inner inner;

 public Inner getInner() {
 return inner;
 }

 private class Inner {
 }
  public static void main(String[] args) {
 Outer outer = new Outer();
 Inner inner = new Outer.Inner();
 outer.setInner(inner);
 log.info("Outer/Inner: " + outer.hashCode() + "/" + outer.getInner().hashCode());
 }

}
```

1. The class name `Outer` is not a legal Java class name.
2. The class name `Inner` is confusing.
3. The main method defined in class `Outer` has the wrong method signature.
4. The `log.info()` call in main() has too many parameters.
5. The class body for class `Inner` is empty.
6. None of the above. The line Inner inner = new Outer.Inner(); is in error. This is not the correct syntax for instantiating a nested class using an enclosing class reference. Rather, it should be: Inner inner = outer.new Inner();

1.Is it possible to write an inner class that can be instantiated by any class in your application (regardless of what package in which it resides) without an enclosing instance of the outer class? Explain your answer.

1.Is it possible to write an inner class that can be instantiated by any class in your application (regardless of what package in which it resides) without an enclosing instance of the outer class? Explain your answer. **Yes, it is. The inner class should be declared with the static modifier, along with public visibility. Then any class in your application can instantiate the inner class without an enclosing outer instance of the class.**

1.What’s the difference between a nested class an an inner class?

1. An inner class is one that is defined using private access only, whereas a nested class is declared static.
2. The two terms are synonymous and can be used interchangeably.
3. A nested class must reside inside the enclosing class and be declared before any of the enclosing class’s variables.
4. A nested class is defined in main(), whereas an inner class can be defined anywhere.
5. There is no such thing as a nested class.
6. None of the above.

1.What’s the difference between a nested class an an inner class?

1. An inner class is one that is defined using private access only, whereas a nested class is declared static.
2. **The two terms are synonymous and can be used interchangeably.**
3. A nested class must reside inside the enclosing class and be declared before any of the enclosing class’s variables.
4. A nested class is defined in main(), whereas an inner class can be defined anywhere.
5. There is no such thing as a nested class.
6. None of the above.

1. Which of these statements is true regarding inner classes?
    1. An inner class can access any private data variables of its enclosing class unless it is declared `static`.
    2. An inner class must be declared `public` to be instantiated by any other class than its enclosing class.
    3. A static inner class is not allowed except under special circumstances.
    4. An inner class is completely invisible to its enclosing class.
    5. None of the above.

1. Which of these statements is true regarding inner classes?
    1. **An inner class can access any private data variables of its enclosing class unless it is declared `static`.**
    2. An inner class must be declared `public` to be instantiated by any other class than its enclosing class.
    3. A static inner class is not allowed except under special circumstances.
    4. An inner class is completely invisible to its enclosing class.
    5. None of the above.

String Class

- Every string created is actually an object of type String.
- Even string constants are actually String objects.
  - System.out.println("This is a String, too");
  - the string "This is a String, too" is a String object.

String Class

- Objects of type String are immutable;
- Once a String object is created, its contents cannot be altered.
  - To change a string, create a new one that contains the modifications.
  - Java defines peer classes of String, called StringBuffer and StringBuilder, which allow strings to be altered.

String Class

- Strings can be constructed in a variety of ways.
- The easiest is to use a statement like this:
  - String myString = "this is a test";
  - System.out.println(myString);
- Java defines one operator for String objects: +.
  - It is used to concatenate two strings.
  - String myString = "I" + " like " + "Java.";
    - myString containing "I like Java."

String Class

```
// Demonstrating Strings.
class StringDemo {
  public static void main(String args[]) {
    String strOb1 = "First String";
    String strOb2 = "Second String";
    String strOb3 = strOb1 + " and " + strOb2;

    System.out.println(strOb1);
    System.out.println(strOb2);
    System.out.println(strOb3);
  }
}
```

The output produced by this program is shown here:

```
First String
Second String
First String and Second String
```

String Class

- The String class contains several methods.
  - Test two strings for equality by using equals( ).
  - Obtain the length of a string by calling the length( ) method.
  - Obtain the character at a specified index within a string by calling charAt( ).
  - The general forms of these three methods are :
    - **boolean equals(secondStr)**
    - **int length( )**
    - **char charAt(index)**

```
// Demonstrating some String methods.
class StringDemo2 {
  public static void main(String args[]) {
    String strOb1 = "First String";
    String strOb2 = "Second String";
    String strOb3 = strOb1;

    System.out.println("Length of strOb1: " +
                       strOb1.length());

    System.out.println("Char at index 3 in strOb1: " +
                       strOb1.charAt(3));

    if(strOb1.equals(strOb2))
      System.out.println("strOb1 == strOb2");
    else
      System.out.println("strOb1 != strOb2");

    if(strOb1.equals(strOb3))
      System.out.println("strOb1 == strOb3");
    else
      System.out.println("strOb1 != strOb3");
  }
}
```

This program generates the following output:

```
Length of strOb1: 12
Char at index 3 in strOb1: s
strOb1 != strOb2
strOb1 == strOb3
```

```
// Demonstrate String arrays.
class StringDemo3 {
  public static void main(String args[]) {
    String str[] = { "one", "two", "three" };

    for(int i=0; i<str.length; i++)
      System.out.println("str[" + i + "]: " +
                         str[i]);
  }
}
```

Here is the output from this program:

```
str[0]: one
str[1]: two
str[2]: three
```

```
// Display all command-line arguments.
class CommandLine {
  public static void main(String args[]) {
    for(int i=0; i<args.length; i++)
      System.out.println("args[" + i + "]: " +
                         args[i]);
  }
}
```

```
java CommandLine this is a test 100 -1
```

output:

```
args[0]: this
args[1]: is
args[2]: a
args[3]: test
args[4]: 100
args[5]: -1
```

Varargs: Variable-Length Arguments

- A method that takes a variable number of arguments is called a variable-arity method, or simply a varargs method.
- Used in cases where the maximum number of potential arguments was larger, or unknowable.
- A second approach was used in which the arguments were put into an array, and then the array was passed to the method.

```
// Use an array to pass a variable number of
// arguments to a method. This is the old-style
// approach to variable-length arguments.
class PassArray {
  static void vaTest(int v[]) {
    System.out.print("Number of args: " + v.length +
                     " Contents: ");

    for(int x : v)
      System.out.print(x + " ");
    System.out.println();
  }

  public static void main(String args[])
  {
    // Notice how an array must be created to
    // hold the arguments.
    int n1[] = { 10 };
    int n2[] = { 1, 2, 3 };
    int n3[] = { };

    vaTest(n1); // 1 arg
    vaTest(n2); // 3 args
    vaTest(n3); // no args
  }
}
```

The output from the program is shown here:

```
Number of args: 1 Contents: 10
Number of args: 3 Contents: 1 2 3
Number of args: 0 Contents:
```

Varargs: Variable-Length Arguments

- A variable-length argument is specified by three periods (…).
- For example, vaTest( ) is written using a vararg:
  - static void vaTest(int ... v) {
    - This syntax tells the compiler that vaTest( ) can be called with zero or more arguments.
    - As a result, v is implicitly declared as an array of type int[ ].
      - Thus, inside vaTest( ), v is accessed using the normal array syntax.

```
// Demonstrate variable-length arguments.
class VarArgs {

  // vaTest() now uses a vararg.
  static void vaTest(int ... v) {
    System.out.print("Number of args: " + v.length +
                     " Contents: ");

    for(int x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  public static void main(String args[])
  {
    // Notice how vaTest() can be called with a
    // variable number of arguments.
    vaTest(10);       // 1 arg
    vaTest(1, 2, 3); // 3 args
    vaTest();         // no args
  }
}
```

Varargs: Variable-Length Arguments

- A method can have “normal” parameters along with a variable-length parameter.
- However, the variable-length parameter must be the last parameter declared by the method.
- For example, this method declaration is perfectly acceptable:
  - int doIt(int a, int b, double c, int ... vals) {
  - In this case, the first three arguments used in a call to doIt( ) are matched to the first three parameters.
  - Then, any remaining arguments are assumed to belong to vals.
- For example, the following declaration is incorrect:
  - int doIt(int a, int b, double c, int ... vals, boolean stopFlag) { // Error!
- For example, this declaration is also invalid:
  - int doIt(int a, int b, double c, int ... vals, double ... morevals) { // Error!

```
// Use varargs with standard arguments.
class VarArgs2 {

  // Here, msg is a normal parameter and v is a
  // varargs parameter.
  static void vaTest(String msg, int ... v) {
    System.out.print(msg + v.length +
                     " Contents: ");

    for(int x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  public static void main(String args[])
  {
    vaTest("One vararg: ", 10);
    vaTest("Three varargs: ", 1, 2, 3);
    vaTest("No varargs: ");
  }
}
```

The output from this program is shown here:

```
One vararg: 1 Contents: 10
Three varargs: 3 Contents: 1 2 3
No varargs: 0 Contents:
```