CS205 Object Oriented Programming in Java

Module 2 - **Core Java Fundamentals (Part 8)**

Prepared by

Renetha J.B.

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

Topics

- Core Java Fundamentals:

  ✓**Final Variables**

  ✓**Inner Classes**

  ✓**Command-Line Arguments**

  ✓**Variable Length Arguments**

Final Variables

- A variable can be declared as **final** by prefixing **final** keyword**.**
- The contents of final variables **cannot be modified**.
- We must **initialize a final variable** when it is declared.

E.g.

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
```

- It is a convention to choose **uppercase identifiers(CAPITAL LETTERS)** for **final variables.** E.g. TOTAL
- We can use **final variables** as if they were **constants,** without fear that a value has been changed.
- Variables declared as **final do not occupy memory on a per-instance basis.**

Nested Classes

- It is possible to define a class within another class; such classes are known as ***nested classes.***
- The scope of a nested class is bounded by the scope of its enclosing class(**outer**).
  - Thus, if class B is defined within class A, then B does not exist independently of A.

Nested Classes(contd.)

- A **nested class** has access to the members, including private members, of the **enclosing(outer) class.**
- The **enclosing class** does not have access to the members of the nested class.

Inner Classes(contd)

- A nested class, that is **declared** directly within its enclosing class scope, is a member of its enclosing class.

```
class Outer
{
//variables and methods
    class Inner
    {
//variables and methods
    }
}
```

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

Inner Classes(contd)

- **Static** nested class
  - A static nested class is one that has the **static modifier** applied**.**
  - It must access the members of its enclosing class through an object.
  - It **cannot refer** to members of its enclosing class **directly**.

- // Demonstrate a STATIC inner class.

```
class Outer
 {
int outer_x = 100;
void test() {
           Nested nested= new Nested ();
           nested.display();
           }
    static class Nested {           //static nested class
           void display() {
           Outer obj = new Outer();
           System.out.println("display: outer_x = " + obj.outer_x);
                               }
                     }
}
class NestedClassDemo {
    public static void main(String args[]) {
           Outer outer = new Outer();
           outer.test();
    } }
```

OUTPUT

```
display: outer_x = 100
```

Inner Class

- **Non static class**
  - A non-static nested class is called **inner class**.
  - An **inner class** has access to all of the variables and **methods of its outer class.**
  - It may **refer** to members of its enclosing class **directly** in the same way that other non-static members of the outer class do.

- // Demonstrate a NONSTATIC inner class.

```
class Outer
 {
int outer_x = 100;
void test() {
        Inner inner = new Inner();
        inner.display();
        }
   class Inner {
        void display() {
                System.out.println("display: outer_x = " + outer_x);
                        }
                }
}
class InnerClassDemo {
   public static void main(String args[]) {
        Outer outer = new Outer();
        outer.test();
    }
}
```

```
OUTPUT
display: outer_x = 100
```

- In the program, an inner class named **Inner** is defined within the scope of **class Outer.**
- Therefore, any code in **class Inner can directly access the** variable outer_x in Outer class**.**
- An instance method named display( ) is defined inside Inner.
  - This method displays outer_x on the standard output stream.
- The main( ) method of InnerClassDemo creates an instance of class Outer and invokes its test( ) method.
- That method creates an instance of class Inner and the display() method is called.

Inner class(contd.)

- An **instance(object) of Inner can be created only within the scope of class Outer.**

- We can create an **instance** of Inner class **outside of Outer class** by qualifying its name with Outer classname, as in

  Outer.Inner ob=outerobject.new Inner();

Inner class(contd.)

- An inner class can **access** all of the **members of its enclosing class**, but the reverse is not true.
- Members of the inner class are known only within the scope of the inner class and may not be used by the outer class.

We can define a nested class within the block defined by a method or even within the body of a **for loop**

```
// Define an inner class within a for loop.
class Outer {
int outer_x = 100;
void test() {
    for(int i=0; i<5; i++)
                    {          class Inner {
                               void display() {
                               System.out.println("display: outer_x = " + outer_x);
                                           }
                                         }
          Inner inner = new Inner();
          inner.display();
                        }
               }      }
class InnerClassDemo {
public static void main(String args[]) {
Outer outer = new Outer();
outer.test();  } }
```

OUTPUT

```
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
```

Command-Line Arguments

- If we want to pass information into a program when you run it, then you can do this by passing *command-line arguments to* ***main( ).***
- A command-line argument is the information that follows program's name on the command line when it is executed.
- Command-line arguments are stored as strings in a **String** array passed to the args parameter of main( ).
  - The first command-line argument is stored **at args[0]**
  - the second at args[1]
  - so on.

```
// 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]);
}
}
```

- Compile this usig javac and execute this program as:-

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

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

Variable length arguments

- In Java methods can take a variable number of arguments.
  - This feature is called **varargs** or **variable-length arguments**.
- A method that takes a variable number of arguments is called a **variable-arity method**, or simply a **varargs** method.

Variable length arguments(contd.

- E.g. A method that opens an Internet connection might take a user name, password, filename, protocol, and so on, but supply defaults if some of this information is not provided. Here it is better to pass only the arguments to which the defaults did not apply.
- E.g. printf() method can have any number of arguments.

Handling variable length arguments

- If the *maximum number of arguments is* ***small*** and ***known***, then we can create **overloaded** versions of the method, one for each way the method could be called.
- If the *maximum number of potential arguments* is ***larger***, or ***unknowable***, then the arguments can be put into an **array**, and then the array can be passed to the method.

```
class PassArray {
    static void test(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[])
    {
      int n1[] = { 10 };
      int n2[] = { 1, 2, 3 };
      int n3[] = { };
      test(n1);          // 1 arg
      test(n2);          // 3 args
      test(n3);          // no args
      }
}
```

OUTPUT

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

This old method requires that these arguments be manually packaged into an array prior to calling the function test( ).

Handling variable length arguments(contd.)

- A variable-length argument is specified by three periods **(...).**
- **E.g.**

static void test(int ... v) { //statemenst }

- This syntax tells the compiler that **test( )** can be called with zero or more arguments.

```
class PassArray {
static void test(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[])
{
    test(10);       // 1 arg
    test(1,2,3);    // 3 args
    test();         // no args
}
}
```

OUTPUT

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

Handling variable length arguments(contd.)

- 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.
- E.g:

  int test(int a, int b, double c, **int ... vals**) { //statements }

VALID

- E.g.

int test(int a, int b, double c, **int ... vals**, boolean stopFlag) { // ERROR!

Overloading Vararg Methods

- We **can overload** a method that takes a variable-length argument.
- There can be many functions with same name and having different type of variable length arguments.

- // Varargs and overloading.

```
class VarArgs3
{
        static void test(int ... v)
        {
        System.out.print(“test(int ...): " + "Number of args: " + v.length);
        }
        static void test(boolean ... v)
        {
        System.out.print(“test(boolean ...) " +"Number of args: " + v.length);
        }
   public static void main(String args[])
   {
                test(1, 2, 3);
                test(true, false);
   }
}
```

OUTPUT

```
test(int ...): Number of args: 3
test(boolean  ...): Number of args: 2
```

Varargs and Ambiguity

- It is possible to create an ambiguous call to an overloaded varargs method.

```
class VarArgs3
{
        static void test(int ... v)
        {
        System.out.print(“test(int ...): " + "Number of args: " + v.length);
        }
        static void test(boolean ... v)
        {
        System.out.print(“test(boolean ...) " +"Number of args: " + v.length);
        }
   public static void main(String args[])
   {
                test(1, 2, 3);
                test();      // Error: Ambiguous!
   }
}
```

test() can call
test(int ...) or **test(boolean ...).**
Because both these functions
have varargs so they can accept
zero arguments .
System is confused which one to call
AMBIGUITY

Varargs and Ambiguity(contd.)

- Another e.g. of ambiguous functions

```
static void test(int ... v) { // ...  }
static void test(int n, int ... v) { // ... }
```

If a call **test(2);** comes, then this will create error (ambiguous)

Reference

- Herbert Schildt, Java: The Complete Reference, 8/e, Tata McGraw Hill, 2011.