ORACLE PRESS

Also Covers **Java SE 11** Developer Exam

Java™

OCP

ORACLE CERTIFIED PROFESSIONAL

JAVA SE 17 Developer Exam 1Z0-829

Programmer's Guide

Volume I

Khalid A. Mughal
Vasily A. Strelnikov

Foreword by Simon Roberts

ORACLE

OCP

Oracle Certified Professional

Java SE 17 Developer
(Exam 1Z0-829)
Programmer’s Guide

```
    // (4) Mandatory non-zero argument constructor:
    SubInnerA(OuterA outerRef) {
      outerRef.super();                           // (5) Explicit super() call
    }
  }
  //______________________________________________________________________________
  class OuterB extends OuterA {                   // (6) Extends class at (1)
    class InnerB extends OuterA.InnerA { }        // (7) Extends NSMC at (2)
  }
  //______________________________________________________________________________
  public class Extending {
    public static void main(String[] args) {

      // (8) Outer instance passed explicitly in constructor call:
      SubInnerA obj1 = new SubInnerA(new OuterA());
      System.out.println(obj1.getClass());

      // (9) No outer instance passed explicitly in constructor call to InnerB:
      OuterB.InnerB obj2 = new OuterB().new InnerB();
      System.out.println(obj2.getClass());
    }
  }
```

Output from the program:

```
class SubInnerA
class OuterB$InnerB
```

In Example 9.8, the non-static member class `InnerA`, declared at (2) in the class `OuterA` at (1), is extended by `SubInnerA` at (3). Note that `SubInnerA` and the class `OuterA` are not related in any way, and that the subclass `OuterB` inherits the class `InnerA` from its superclass `OuterA`. An instance of `SubInnerA` is created at (8). An instance of the class `OuterA` is explicitly passed as an argument in the constructor call to `SubInnerA`. The constructor at (4) for `SubInnerA` has a special `super()` call in its body at (5), called a *qualified superclass constructor invocation*. This call ensures that the constructor of the superclass `InnerA` has an outer object (denoted by the reference `outerRef`) to bind to. Using the standard `super()` call in the subclass constructor is not adequate because it does not provide an outer instance for the superclass constructor to bind to. The non-zero argument constructor at (4) and the `outerRef.super()` expression at (5) are mandatory to set up the relationships correctly between the objects involved.

The outer object problem mentioned above does not arise if the subclass that extends an inner class is also declared within an outer class that extends the outer class of the superclass. This situation is illustrated at (6) and (7): The classes `InnerB` and `OuterB` extend the classes `InnerA` and `OuterA`, respectively. The member class `InnerA` is inherited by the class `OuterB` from its superclass `OuterA`—and can be regarded as being nested in the class `OuterB`. Thus an object of class `OuterB` can act as an outer object for both an instance of class `InnerA` and that of class `InnerB`. The object creation expression `new OuterB().new InnerB()` at (9) creates an `OuterB` object and implicitly passes its reference to the default constructor of class `InnerB`. The default constructor of class `InnerB` invokes the default constructor of its superclass `InnerA` by calling `super()` and passing it the reference of the `OuterB` object, which the constructor of class `InnerA` can readily bind to.

It goes without saying that such convoluted inheritance and nesting relationships as those in Example 9.8 hardly qualify as best coding practices.

Review Questions

9.1 Which statement is true about the following program?

```
public class MyClass {
  public static void main(String[] args) {
    Outer objRef = new Outer();
    System.out.println(objRef.createInner().getSecret());
  }
}

class Outer {
  private int secret;
  Outer() { secret = 123; }

  class Inner {
    int getSecret() { return secret; }
  }

  Inner createInner() { return new Inner(); }
}
```

Select the one correct answer.

(a) The program will fail to compile because the class `Inner` cannot be declared within the class `Outer`.
(b) The program will fail to compile because the method `createInner()` cannot be allowed to pass objects of the class `Inner` to methods outside the class `Outer`.
(c) The program will fail to compile because the field `secret` is not accessible from the method `getSecret()`.
(d) The program will fail to compile because the method `getSecret()` is not accessible from the `main()` method in the class `MyClass`.
(e) The code will compile and print `123` at runtime.

9.2 Which of the following statements are true about nested classes?
Select the two correct answers.

(a) An instance of a static member class has an implicit outer instance.
(b) A static member class can contain non-`static` fields.
(c) A static member interface can contain non-`static` fields.
(d) A static member interface has an implicit outer instance.
(e) An instance of the outer class can be associated with many instances of a non-static member class.

9.3 Which statement is true about the following program?

```
public class Nesting {
  public static void main(String[] args) {
    B.C obj = new B().new C();
  }
}

class A {
  int val;
  A(int v) { val = v; }
}

class B extends A {
  int val = 1;
  B() { super(2); }

  class C extends A {
    int val = 3;
    C() {
      super(4);
      System.out.println(B.this.val);
      System.out.println(C.this.val);
      System.out.println(super.val);
    }
  }
}
```

Select the one correct answer.

(a) The program will fail to compile.
(b) The program will compile and print `2`, `3`, and `4`, in that order at runtime.
(c) The program will compile and print `1`, `4`, and `2`, in that order at runtime.
(d) The program will compile and print `1`, `3`, and `4`, in that order at runtime.
(e) The program will compile and print `3`, `2`, and `1`, in that order at runtime.

9.4 Which of the following statements are true about the following program?

```
public class Outer {
  public void doIt() {}
  public class Inner {
    public void doIt() {}
  }

  public static void main(String[] args) {
    new Outer().new Inner().doIt();
  }
}
```

Select the two correct answers.

(a) The `doIt()` method in the `Inner` class overrides the `doIt()` method in the `Outer` class.
(b) The `doIt()` method in the `Inner` class overloads the `doIt()` method in the `Outer` class.
(c) The `doIt()` method in the `Inner` class hides the `doIt()` method in the `Outer` class.

(d) The qualified name of the `Inner` class is `Outer.Inner`.
(e) The program will fail to compile.

9.4 Local Classes

Declaring Local Classes

A local class is an inner class that is defined in a block. This can be essentially any context where a local block or block body is allowed: a method, a constructor, an initializer block, a `try-catch-finally` construct, loop bodies, or an `if-else` statement. Example 9.9 shows declaration of the local class `StaticLocal` at (5) that is defined in the static context of the method `staticMethod()` at (1).

A local class cannot have any access modifier and cannot be declared `static`, as shown at (4) in Example 9.9. However, it can be declared `abstract` or `final`, as shown at (5). The declaration of the class is only accessible in the context of the block in which it is defined, subject to the same scope rules as for local variable declarations. In particular, it must be declared before use in the block. In Example 9.9, an attempt to create an object of class `StaticLocal` at (2) and use the class `StaticLocal` at (3) fails, as the class has not been defined before use, but this is not a problem at (11), (12), (13), and (14).

A local class can declare members and constructors, shown from (6) to (10), as in a normal class. The members of the local class can have any access level, and are accessible in the enclosing block regardless of their access level. Even though the field `if1` at (7) is `private`, it is accessible in the enclosing method at (12).

Blocks in non-static context have a `this` reference available, which refers to an instance of the class containing the block. An instance of a local class, which is declared in such a non-`static` block, has an instance of the enclosing class associated with it. This gives such a non-`static` local class much of the same capability as a non-static member class.

However, if the block containing a local class declaration is defined in static context (i.e., a `static` method or a static initializer block), the local class is implicitly `static` in the sense that its instantiation does not require any outer object. This aspect of local classes is reminiscent of static member classes. However, note that a local class cannot be specified with the keyword `static`. The `static` method at (1) is called at (15). The local class `StaticLocal` can only be instantiated, as shown at (11), in the enclosing method `staticMethod()` and does not require any outer object of the enclosing class. Analogous to the value of a local variable, the object of the local class is not available to the caller of the method after the method completes execution, unless measures are taken to store it externally or if its reference value is returned by the call.

Example 9.9 *Declaring Local Classes*

```
// File: LocalClient1.java
class TLCWithSLClass {                                 // Top-level Class

  static void staticMethod(final int fp) {            // (1) Static Method
//  StaticLocal slRef = new StaticLocal(10);          // (2) Class cannot be resolved
//  System.out.println(StaticLocal.staticValue()); // (3) Class cannot be resolved


//  public static class StaticLocal { // (4) Not OK. Cannot be static,
                                      //             and no access modifier
    final class StaticLocal {                          // (5) Static local class
      public static final int sf1 = 10;                // (6) Static field
      private int if1;                                 // (7) Instance field
      public StaticLocal(int val) {                    // (8) Constructor
        this.if1 = val;
      }
      public int getValue() { return if1; }            // (9) Instance method
      public static int staticValue() { return sf1; }// (10) Static method
    } // end StaticLocal

    StaticLocal slRef2 = new StaticLocal(100);                           // (11)
    System.out.println("Instance field: " + slRef2.if1);                 // (12)
    System.out.println("Instance method call: " + slRef2.getValue());    // (13)
    System.out.println("Static method call: " + StaticLocal.staticValue());// (14)
  } // end staticMethod
}

public class LocalClient1 {
  public static void main(String[] args) {
    TLCWithSLClass.staticMethod(100);                                    // (15)
  }
}
```

Output from the program:

```
Instance field: 100
Instance method call: 100
Static method call: 10
```

Accessing Declarations in Enclosing Context

Declaring a local class in a static or a non-static block influences what the class can access in the enclosing context.

Accessing Local Declarations in the Enclosing Block

Example 9.10 illustrates how a local class can access declarations in its enclosing block. Example 9.10 shows declaration of the local class `NonStaticLocal` at (7) that is defined in the non-static context of the method `nonStaticMethod()` at (1).

A local class can access variables (local variables, method parameters, and catch-block parameters) that are declared final or *effectively* final in the scope of its local context. A variable whose value does not change after it is initialized is said to be *effectively* final. This situation is shown at (8) and (9) in the NonStaticLocal class, where the final parameter fp and the effectively final local variable flv of the method nonStaticMethod() are accessed. Access to local variables that are not final or effectively final is not permitted from local classes. The local variable nfv1 at (4) is accessed at (10) in the local class, but this local variable is not effectively final as it is reassigned a new value at (6).

Accessing a local variable from the local context that has not been declared or has not been *definitely assigned* (§5.5, p. 232) results in a compile-time error, as shown at (11) and (12). The local variable nflv2 accessed at (11) is not declared before use, as it is declared at (16). The local variable nflv3 accessed at (12) is not initialized before use, as it is initialized at (17)—which means it is not definitely assigned at (12).

Declarations in the local class can *shadow* declarations in the enclosing block. The field hlv at (13) shadows the local variable by the same name at (3) in the enclosing method. There is no way for the local class to refer to shadowed declarations in the enclosing block.

The non-static method at (1) is called at (19) on an instance of its enclosing class. When the constructor at (15) in the non-static method is executed, the reference to this instance is passed implicitly to the constructor, thus this instance acts as the enclosing object of the local class instance.

Example 9.10 *Accessing Local Declarations in the Enclosing Block (Local Classes)*

```
// File: LocalClient2.java
class TLCWithNSLClass {                // Top-level Class

  void nonStaticMethod(final int fp) { // (1) Non-static Method
    // Local variables:
    int flv = 10;              // (2) Effectively final local variable
    final int hlv = 20;        // (3) Final local variable (constant variable)
    int nflv1 = 30;            // (4) Non-final local variable
    int nflv3;                 // (5) Non-final local variable declaration

    nflv1 = 40;                // (6) Not effectively final local variable

    // Non-static local class
    class NonStaticLocal {// (7)
      int f1 = fp;        // (8) Final param from enclosing method
      int f2 = flv;       // (9) Effectively final variable from enclosing method
//    int f3 = nflv1;     // (10) Not effectively final from enclosing method
//    int f4 = nflv2;     // (11) Name nflv2 cannot be resolved: use-before-decl
//    int f5 = nflv3;     // (12) Not definitely assigned
      int hlv;            // (13) Shadows local variable at (3)
      NonStaticLocal (int value) {
        hlv = value;
       System.out.println("Instance field: " + hlv);// (14) Prints value from (13)
```