**Switch Statements Revisited**

When we discussed Switch statements in the Java Syntax section, I mentioned that each case statement had to use a value that was understood at compile time (variables and return values from methods cannot be used).  At that point in the course, the only value that fit that description was a literal value.  For example, the following code uses char literals for each case statement:

char studentGrade = 'A';

switch (studentGrade) {

case 'A':

System.out.println("Student received an A");

break;

case 'B':

System.out.println("Student received a B");

break;

default:

System.out.println("Student received a C or lower");

}

Now that we've learned about "final," we can add constants to our short list of valid case values.  Constants works because their value is known at compile time.  So the following is legal:

final char A\_GRADE = 'A';

final char B\_GRADE = 'B';

char studentGrade = 'A';

switch (studentGrade) {

case A\_GRADE:

System.out.println("Student received an A");

break;

case B\_GRADE:

System.out.println("Student received a B");

break;

default:

System.out.println("Student received a C or lower");

}

**Know This For The Exam: Class/Object Invocation Order**

When an object in an inheritance chain is instantiated for the first time, this is the order that the code is executed:

1. All of the static variables are defined in the class (with default values).  For example, the code public static String firstName = "Jason";  would result in the creation of the variable firstName  but in this step, it is assigned the default value of null (rather than the explicit value of "Jason").  This happens for all of the classes in the hierarchy (moving up the inheritance chain).
2. All of the static initialization blocks and explicit values assigned to static variables (such as "Jason" above), are executed in the base class, in the order they are written (from top to bottom).
3. All of the static initialization blocks and explicit values assigned to static variables are executed for the immediate child of the base class, in the order they are written (from top to bottom).
4. Step 3 is repeated all the way down the object hierarchy until the instantiated object type is reached.
5. All of the instance variables are defined with default values, for all of the classes in the hierarchy (moving up the inheritance chain).
6. All of the instance initialization blocks and explicit values assigned to instance variables are executed for the base class, in the order they are written (from top to bottom).
7. The constructor for the base class is executed.
8. All of the instance initialization blocks and explicit values assigned to instance variables are executed for the immediate child of the base class, in the order they are written (from top to bottom),
9. The constructor for the immediate child of the base class is executed.
10. Steps 8 - 9 are repeated all the way down the object hierarchy until the instantiated object type is reached.

Note that the first steps that execute "static" code will only happen once during the entire life of the class, and may have been executed before an object has been instantiated (such as when a reference variable is created or a static member is accessed).

Here's an example to illustrate the order class and object instantiation is executed. The following classes are defined in three separate files (and note that their members are organized in a haphazard manner just to make sure you're paying attention!):

**Person**

public class Person {

{

System.out.println("Person: First Instance Initialization Block");

}

static {

System.out.println("Person: First Static Block");

}

{

System.out.println("Person: Second Instance Initialization Block");

}

static {

System.out.println("Person: Second Static Block");

}

public Person() {

System.out.println("Person()");

}

public void sayHello() {

System.out.println("Person: Hello!");

}

}

**Employee**

public class Employee extends Person {

{

System.out.println("Employee: First Instance Initialization Block");

}

static {

System.out.println("Employee: First Static Block");

}

{

System.out.println("Employee: Second Instance Initialization Block");

}

static {

System.out.println("Employee: Second Static Block");

}

public Employee() {

System.out.println("Employee()");

}

}

**Instructor**

public class Instructor extends Employee {

{

System.out.println("Instructor: First Instance Initialization Block");

str1 = "First Instance Initialization String";

}

static {

System.out.println("Instructor: First Static Block");

str1 = "First Static Initialization String";

}

public static String str1 = "Explicit Initialization String";

{

System.out.println("Instructor: Second Instance Initialization Block");

str1 = "Second Instance Initialization String";

}

static {

System.out.println("Instructor: Second Static Block");

str1 = "Second Static Initialization String";

}

public Instructor() {

System.out.println("Instructor()");

}

}

The following code...

Instructor i = new Instructor();

i.sayHello();

System.out.println(Instructor.str1);

...would generate this output :

(assuming the Person, Employee, and Instructor classes have not been previously referenced, otherwise the static blocks wouldn't be executed again)

Person: First Static Block

Person: Second Static Block

Employee: First Static Block

Employee: Second Static Block

Instructor: First Static Block

Instructor: Second Static Block

Person: First Instance Initialization Block

Person: Second Instance Initialization Block

Person()

Employee: First Instance Initialization Block

Employee: Second Instance Initialization Block

Employee()

Instructor: First Instance Initialization Block

Instructor: Second Instance Initialization Block

Instructor()

Person: Hello!

Second Instance Initialization String

NOTE: If another Instructor object is instantiated, the static blocks would be skipped, since that only happens once during the life of the class.