CERTIFICATION SUMMARY

After absorbing the material in this chapter, you should be familiar with some of the nuances of the Java language. You may also be experiencing confusion around why you ever wanted to take this exam in the first place. That's normal at this point. If you hear yourself asking, "What was I thinking?" just lie down until it passes. We would like to tell you that it gets easier…that this was the toughest chapter and it's all downhill from here.

Let's briefly review what you'll need to know for the exam:

There will be many questions dealing with keywords indirectly, so be sure you can identify which are keywords and which aren't.

You need to understand the rules associated with creating legal identifiers and the rules associated with source code declarations, including the use of `package` and `import` statements.

You learned the basic syntax for the `java` and `javac` command-line programs.

You learned about when `main()` has superpowers and when it doesn't.

We covered the basics of `import` and `import static` statements. It's tempting to think that there's more to them than saving a bit of typing, but there isn't.

You now have a good understanding of access control as it relates to classes, methods, and variables. You've looked at how access modifiers (`public`, `protected`, and `private`) define the access control of a class or member.

You learned that `abstract` classes can contain both `abstract` and nonabstract methods, but that if even a single method is marked `abstract`, the class must be marked `abstract`. Don't forget that a concrete (nonabstract) subclass of an `abstract` class must provide implementations for all the `abstract` methods of the superclass, but that an `abstract` class does not have to implement the `abstract` methods from its superclass. An `abstract` subclass can "pass the buck" to the first concrete subclass.

We covered interface implementation. Remember that interfaces can extend another interface (even multiple interfaces), and that any class that implements an interface must implement all methods from all the interfaces in the inheritance tree of the interface the class is implementing.

You've also looked at the other modifiers, including `static`, `final`, `abstract`, `synchronized`, and so on. You've learned how some modifiers can never be combined in a declaration, such as mixing `abstract` with either `final` or `private`.

Keep in mind that there are no `final` objects in Java. A reference variable marked `final` can never be changed, but the object it refers to can be modified. You've seen that `final` applied to methods means a subclass can't override them, and when applied to a class, the `final` class can't be subclassed.

Methods can be declared with a var-arg parameter (which can take from zero to many arguments of the declared type), but that you can have only one var-arg per method, and it must be the method's last parameter.

Make sure you're familiar with the relative sizes of the numeric primitives. Remember that while the values of nonfinal variables can change, a reference variable's type can never change.

You also learned that arrays are objects that contain many variables of the same type. Arrays can also contain other arrays.

Remember what you've learned about `static` variables and methods, especially that `static` members are per-class as opposed to per-instance. Don't forget that a `static` method can't directly access an instance variable from the class it's in because it doesn't have an explicit reference to any particular instance of the class.

Finally, we covered `enums`. An `enum` is a safe and flexible way to implement constants. Because they are a special kind of class, `enums` can be declared very simply, or they can be quite complex—including such attributes as methods, variables, constructors, and a special type of inner class called a constant specific class body.

Before you hurl yourself at the practice test, spend some time with the following optimistically named "Two-Minute Drill." Come back to this particular drill often as you work through this book and especially when you're doing that last-minute cramming. Because—and here's the advice you wished your mother had given you before you left for college—it's not what you know, it's when you know it.

For the exam, knowing what you can't do with the Java language is just as important as knowing what you can do. Give the sample questions a try! They're very similar to the difficulty and structure of the real exam questions and should be an eye opener for how difficult the exam can be. Don't worry if you get a programmers need two or three serious passes through a chapter (or an individual objective) before they can answer the questions confidently.

CERTIFICATION SUMMARY

We started the chapter by discussing the importance of encapsulation in good OO design, and then we talked about how good encapsulation is implemented: with private instance variables and public getters and setters.

Next, we covered the importance of inheritance, so that you can grasp overriding, overloading, polymorphism, reference casting, return types, and constructors.

We covered IS-A and HAS-A. IS-A is implemented using inheritance, and HAS-A is implemented by using instance variables that refer to other objects.

Polymorphism was next. Although a reference variable's type can't be changed, it can be used to refer to an object whose type is a subtype of its own. We learned how to determine what methods are invocable for a given reference variable.

We looked at the difference between overridden and overloaded methods, learning that an overridden method occurs when a subtype inherits a method from a supertype and then reimplements the method to add more specialized behavior. We learned that, at runtime, the JVM will invoke the subtype version on an instance of a subtype and the supertype version on an instance of the supertype. `Abstract` methods must be "overridden" (technically, `abstract` methods must be implemented, as opposed to overridden, since there really isn't anything to override).

We saw that overriding methods must declare the same argument list and return type or they can return a subtype of the declared return type of the supertype's overridden method), and that the access modifier can't be more restrictive. The overriding method also can't throw any new or broader checked exceptions that weren't declared in the overridden method. You also learned that the overridden method can be invoked using the syntax `super.doSomething();`.

Overloaded methods let you reuse the same method name in a class, but with different arguments (and, optionally, a different return type). Whereas overriding methods must not change the argument list, overloaded methods must. But unlike overriding methods, overloaded methods are free to vary the return type, access modifier, and declared exceptions any way they like.

We learned the mechanics of casting (mostly downcasting) reference variables and when it's necessary to do so.

Implementing interfaces came next. An interface describes a *contract* that the implementing class must follow. The rules for implementing an interface are similar to those for extending an `abstract` class. As of Java 8, interfaces can have concrete methods, which are labeled `default`. Also, remember that a class can implement more than one interface and that interfaces can extend another interface.

We also looked at method return types and saw that you can declare any return type you like (assuming you have access to a class for an object reference return type), unless you're overriding a method. Barring a covariant return, an overriding method must have the same return type as the overridden method of the superclass. We saw that, although overriding methods must not change the return type, overloaded methods can (as long as they also change the argument list).

Finally, you learned that it is legal to return any value or variable that can be implicitly converted to the declared return type. So, for example, a `short` can be returned when the return type is declared as an `int`. And (assuming `Horse` extends `Animal`), a `Horse` reference can be returned when the return type is declared an `Animal`.

We covered constructors in detail, learning that if you don't provide a constructor for your class, the compiler will insert one. The compiler-generated constructor is called the default constructor, and it is always a no-arg constructor with a no-arg call to `super()`. The default constructor will never be generated if even a single constructor exists in your class (regardless of the arguments of that constructor); so if you need more than one constructor in your class and you want a no-arg constructor, you'll have to write it yourself. We also saw that constructors are not inherited and that you can be confused by a method that has the same name as the class (which is legal). The return type is the giveaway that a method is not a constructor because constructors do not have return types.

We saw how all the constructors in an object's inheritance tree will always be invoked when the object is instantiated using `new`. We also saw that constructors can be overloaded, which means defining constructors with different argument lists. A constructor can invoke another constructor of the same class using the keyword `this()`, as though the constructor were a method named `this()`. We saw that every constructor must have either `this()` or `super()` as the first statement (although the compiler can insert it for you).

After constructors, we discussed the two kinds of initialization blocks and how and when their code runs.

We looked at `static` methods and variables. `static` members are tied to the class or interface, not an instance, so there is only one copy of any `static` member. A common mistake is to attempt to reference an instance variable from a `static` method. Use the respective class or interface name with the dot operator to access `static` members.

And, once again, you learned that the exam includes tricky questions designed largely to test your ability to recognize just how tricky the questions can be.

CERTIFICATION SUMMARY

This chapter covered a wide range of topics. Don't worry if you have to review some of these topics as you get into later chapters. This chapter includes a lot of foundational stuff that will come into play later.

We started the chapter by reviewing the stack and the heap; remember that local variables live on the stack and instance variables live with their objects on the heap.

We reviewed legal literals for primitives and `Strings`, and then we discussed the basics of assigning values to primitives and reference variables and the rules for casting primitives.

Next we discussed the concept of scope, or "How long will this variable live?" Remember the four basic scopes in order of lessening life span: static, instance, local, and block.

We covered the implications of using uninitialized variables and the importance of the fact that local variables MUST be assigned a value explicitly. We talked about some of the tricky aspects of assigning one reference variable to another and some of the finer points of passing variables into methods, including a discussion of "shadowing."

Finally, we dove into garbage collection, Java's automatic memory management feature. We learned that the heap is where objects live and where all the cool garbage collection activity takes place. We learned that in the end, the JVM will perform garbage collection whenever it wants to. You (the programmer) can request a garbage collection run, but you can't force it. We talked about garbage collection only applying to objects that are eligible, and that eligible means "inaccessible from any live thread." Finally, we discussed the rarely useful `finalize()` method and what you'll have to know about it for the exam. All in all, this was one fascinating chapter.

CERTIFICATION SUMMARY

If you’ve studied this chapter diligently, you should have a firm grasp on Java operators, and you should understand what equality means when tested with the == operator. Let’s review the highlights of what you’ve learned in this chapter.

The logical operators (`&&`, `||`, `&`, `|`, and `^`) can be used only to evaluate two `boolean` expressions. The difference between `&&` and `&` is that the `&&` operator won’t bother testing the right operand if the left evaluates to `false`, because the result of the `&&` expression can never be `true`. The difference between `||` and `|` is that the `||` operator won’t bother testing the right operand if the left evaluates to `true` because the result is already known to be `true` at that point.

The == operator can be used to compare values of primitives, but it can also be used to determine whether two reference variables refer to the same object.

The `instanceof` operator is used to determine whether the object referred to by a reference variable passes the IS-A test for a specified type.

The + operator is overloaded to perform `String` concatenation tasks and can also concatenate `Strings` and primitives, but be careful—concatenation can be tricky.

The conditional operator (a.k.a. the “ternary operator”) has an unusual, three-operand syntax—don’t mistake it for a complex assert statement.

The `++` and `--` operators will be used throughout the exam, and you must pay attention to whether they are prefixed or postfixed to the variable being updated.

Even though you should use parentheses in real life, for the exam you should memorize Table 4-2 so you can determine how code that doesn’t use parentheses for complex expressions will be evaluated, based on Java’s operator-precedence hierarchy.

Be prepared for a lot of exam questions involving the topics from this chapter. Even within questions testing your knowledge of another objective, the code will frequently use operators, assignments, object and primitive passing, and so on.

CERTIFICATION SUMMARY

This chapter covered a lot of ground, all of which involved ways of controlling your program flow based on a conditional test. First, you learned about `if` and `switch` statements. The `if` statement evaluates one or more expressions to a `boolean` result. If the result is `true`, the program will execute the code in the block that is encompassed by the `if`. If an `else` statement is used and the `if` expression evaluates to `false`, then the code following the `else` will be performed. If no `else` block is defined, then none of the code associated with the `if` statement will execute.

You also learned that the `switch` statement can be used to replace multiple `if-else` statements. The `switch` statement can evaluate integer primitive types that can be implicitly cast to an `int` (those types are `byte`, `short`, `int`, and `char`); or it can evaluate `enums`; and as of Java 7, it can evaluate `Strings`. At runtime, the JVM will try to find a match between the expression in the `switch` statement and a constant in a corresponding `case` statement. If a match is found, execution will begin at the matching case and continue on from there, executing code in all the remaining `case` statements until a `break` statement is found or the end of the `switch` statement occurs. If there is no match, then the `default` case will execute, if there is one.

You've learned about the three looping constructs available in the Java language. These constructs are the `for` loop (including the basic `for` and the enhanced `for`, which was new to Java 5), the `while` loop, and the `do` loop. In general, the `for` loop is used when you know how many times you need to go through the loop. The `while` loop is used when you do not know how many times you want to go through, whereas the `do` loop is used when you need to go through at least once. In the `for` loop and the `while` loop, the expression has to evaluate to `true` to get inside the block and will check after every iteration of the loop. The `do` loop does not check the condition until after it has gone through the loop once. The major benefit of the `for` loop is the ability to initialize one or more variables and increment or decrement those variables in the `for` loop definition.

The `break` and `continue` statements can be used in either a labeled or unlabeled fashion. When unlabeled, the `break` statement will force the program to stop processing the innermost looping construct and start with the line of code following the loop. Using an unlabeled `continue` command will cause the program to stop execution of the current iteration of the innermost loop and proceed with the next iteration. When a `break` or a `continue` statement is used in a labeled manner, it will perform in the same way, with one exception: the statement will not apply to the innermost loop; instead, it will apply to the loop with the label. The `break` statement is used most often in conjunction with the `switch` statement. When there is a match between the `switch` expression and the `case` constant, the code following the `case` constant will be performed. To stop execution, a `break` is needed.

You've seen how Java provides an elegant mechanism in exception handling. Exception handling allows you to isolate your error-correction code into separate blocks so the main code doesn't become cluttered by error-checking code. Another elegant feature allows you to handle similar errors with a single error-handling block, without code duplication. Also, the error handling can be deferred to methods further back on the call stack.

You learned that Java's `try` keyword is used to specify a guarded region—a block of code in which problems might be detected. An exception handler is the code that is executed when an exception occurs. The handler is defined by using Java's `catch` keyword. All `catch` clauses must immediately follow the related `try` block.

Java also provides the `finally` keyword. This is used to define a block of code that is always executed, either immediately after a `catch` clause completes or immediately after the associated `try` block in the case that no exception was thrown (or there was a `try` but no `catch`). Use `finally` blocks to release system resources and to perform any cleanup required by the code in the `try` block. A `finally` block is not required, but if there is one, it must immediately follow the last `catch`. (If there is no `catch` block, the `finally` block must immediately follow the `try` block.) It's guaranteed to be called except when the `try` or `catch` issues a `System.exit()`.

An exception object is an instance of class `Exception` or one of its subclasses. The `catch` clause takes, as a parameter, an instance of an object of a type derived from the `Exception` class. Java requires that each method either catches any checked exception it can throw or else declares that it throws the exception. The exception declaration is part of the method's signature. To declare that an exception may be thrown, the `throws` keyword is used in a method definition, along with a list of all checked exceptions that might be thrown.

Runtime exceptions are of type `RuntimeException` (or one of its subclasses). These exceptions are a special case because they do not need to be handled or declared, and thus are known as "unchecked" exceptions. Errors are of type `java.lang.Error` or its subclasses, and like runtime exceptions, they do not need to be handled or declared. Checked exceptions include any exception types that are not of type `RuntimeException` or `Error`. If your code fails either to handle a checked exception or declare that it is thrown, your code won't compile. But with unchecked exceptions or objects of type `Error`, it doesn't matter to the compiler whether you declare them or handle them, do nothing about them, or do some combination of declaring and handling. In other words, you're free to declare them and handle them, but the compiler won't care one way or the other. It's not good practice to handle an `Error`, though, because you can rarely recover from one.

Finally, remember that exceptions can be generated by the JVM or by a programmer.

CERTIFICATION SUMMARY

The most important thing to remember about `Strings` is that `String` objects are immutable, but references to `Strings` are not! You can make a new `String` by using an existing `String` as a starting point, but if you don't assign a reference variable to the new `String`, it will be lost to your program—you will have no way to access your new `String`. Review the important methods in the `String` class.

The `StringBuilder` class was added in Java 5. It has exactly the same methods as the old `StringBuffer` class, except `StringBuilder`'s methods aren't thread-safe. Because `StringBuilder`'s methods are not thread-safe, they tend to run faster than `StringBuffer` methods, so choose `StringBuilder` whenever threading is not an issue. Both `StringBuffer` and `StringBuilder` objects can have their value changed over and over without your having to create new objects. If you're doing a lot of string manipulation, these objects will be more efficient than immutable `String` objects, which are, more or less, "use once, remain in memory forever." Remember, these methods ALWAYS change the invoking object's value, even with no explicit assignment.

Next we discussed key classes and interfaces in the new Java 8 calendar and time-related packages. Similar to `Strings`, all of the calendar classes we studied create immutable objects. In addition, these classes use factory methods exclusively to create new objects. The keyword `new` cannot be used with these classes. We looked at some of the powerful features of these classes, like calculating the amount of time between two different dates or times. Then we took a look at how the `DateTimeFormatter` class is used to parse `Strings` into calendar objects and how it is used to beautify calendar objects.

The next topic was arrays. We talked about declaring, constructing, and initializing one-dimensional and multidimensional arrays. We talked about anonymous arrays and the fact that arrays of objects are actually arrays of references to objects.

Next, we discussed the basics of `ArrayLists`. `ArrayLists` are like arrays with superpowers that allow them to grow and shrink dynamically and to make it easy for you to insert and delete elements at locations of your choosing within the list. We discussed the idea that `ArrayLists` cannot hold primitives, and that if you want to make an `ArrayList` filled with a given type of primitive values, you use "wrapper" classes to turn a primitive value into an object that represents that value. Then we discussed how with autoboxing, turning primitives into wrapper objects, and vice versa, is done automatically.

Finally, we discussed a specific subset of the topic of lambdas, using the `Predicate` interface. The basic idea of lambdas is that you can pass a bit of code from one method to another. The `Predicate` interface is one of many "functional interfaces" provided in the Java 8 API. A functional interface is one that has only one method to be implemented. In the case of the `Predicate` interface, this method is called `test()`, and it takes a single argument and returns a `boolean`. To wrap up our discussion of lambdas, we covered some of the tricky syntax rules you need to know to write valid lambdas.