Writing Code Using Loops (OCA Objectives 5.1, 5.2, 5.3, and 5.4)

- ❑ A basic `for` statement has three parts: declaration and/or initialization, boolean evaluation, and the iteration expression.
- ❑ If a variable is incremented or evaluated within a basic `for` loop, it must be declared before the loop or within the `for` loop declaration.
- ❑ A variable declared (not just initialized) within the basic `for` loop declaration cannot be accessed outside the `for` loop—in other words, code below the `for` loop won't be able to use the variable.
- ❑ You can initialize more than one variable of the same type in the first part of the basic `for` loop declaration; each initialization must be separated by a comma.
- ❑ An enhanced `for` statement (new as of Java 5) has two parts: the *declaration* and the *expression*. It is used only to loop through arrays or collections.
- ❑ With an enhanced `for`, the *expression* is the array or collection through which you want to loop.
- ❑ With an enhanced `for`, the *declaration* is the block variable, whose type is compatible with the elements of the array or collection, and that variable contains the value of the element for the given iteration.
- ❑ You cannot use a number (old C-style language construct) or anything that does not evaluate to a `boolean` value as a condition for an `if` statement or looping construct. You can't, for example, say `if(x)`, unless `x` is a `boolean` variable.
- ❑ The `do` loop will enter the body of the loop at least once, even if the test condition is not met.

Using break and continue (OCA Objective 5.5)

- ❑ An unlabeled `break` statement will cause the current iteration of the innermost looping construct to stop and the line of code following the loop to run.
- ❑ An unlabeled `continue` statement will cause the current iteration of the innermost loop to stop, the condition of that loop to be checked, and if the condition is met, the loop to run again.
- ❑ If the `break` statement or the `continue` statement is labeled, it will cause similar action to occur on the labeled loop, not the innermost loop.

Handling Exceptions (OCA Objectives 8.1, 8.2, 8.3, and 8.4)

- ❑ Exceptions come in two flavors: checked and unchecked.
- ❑ Checked exceptions include all subtypes of `Exception`, excluding classes that extend `RuntimeException`.
- ❑ Checked exceptions are subject to the handle or declare rule; any method that might throw a checked exception (including methods that invoke methods that can throw a checked exception) must either declare the exception using `throws`, or handle the exception with an appropriate `try/catch`.
- ❑ Subtypes of `Error` or `RuntimeException` are unchecked, so the compiler doesn't enforce the handle or declare rule. You're free to handle them or to declare them, but the compiler doesn't care one way or the other.
- ❑ If you use an optional `finally` block, it will always be invoked, regardless of whether an exception in the corresponding `try` is thrown or not, and regardless of whether a thrown exception is caught or not.
- ❑ The only exception to the `finally`-will-always-be-called rule is that a `finally` will not be invoked if the JVM shuts down. That could happen if code from the `try` or `catch` blocks calls `System.exit()`.
- ❑ Just because `finally` is invoked does not mean it will complete. Code in the `finally` block could itself raise an exception or issue a `System.exit()`.
- ❑ Uncaught exceptions propagate back through the call stack, starting from the method where the exception is thrown and ending with either the first method that has a corresponding catch for that exception type or a JVM shutdown (which happens if the exception gets to `main()`, and `main()` is "ducking" the exception by declaring it).
- ❑ You can create your own exceptions, normally by extending `Exception` or one of its subtypes. Your exception will then be considered a checked exception (unless you are extending from `RuntimeException`), and the compiler will enforce the handle or declare rule for that exception.
- ❑ All `catch` blocks must be ordered from most specific to most general. If you have a `catch` clause for both `IOException` and `Exception`, you must put the `catch` for `IOException` first in your code. Otherwise, the `IOException` would be caught by `catch(Exception e)`, because a `catch` argument can catch the specified exception or any of its subtypes! The compiler will stop you from defining `catch` clauses that can never be reached.
- ❑ Some exceptions are created by programmers, and some by the JVM.