Writing Code Using if and switch Statements

- The *if* statement must have all expressions enclosed by at least one pair of parentheses.
- The only legal argument to an *if* statement is a boolean, so the *if* test can beonly on an expression that resolves to a boolean or a boolean variable.
- Watch out for boolean assignments (=) that can be mistaken for Boolean equality (==) tests:

  ```
  boolean x = false;
  if (x = true) { } // an assignment, so x will always be true!
  ```
- Curly braces are optional for *if* blocks that have only one conditional statement. But watch out for misleading indentations.
- Switch statements can evaluate only the byte, short, int, and char data types. You can't say

  ```
  long s = 30;
  switch(s) { }
  ```

- The case argument must be a literal or final variable! You cannot have a *case* that includes a non-final variable, or a range of values.
- If the condition in a *switch* statement matches a *case* value, execution will run through all code in the *switch* following the matching case statement until a *break* or the end of the *switch* statement is encountered. In other words, *the matching case is just the entry point into the case block*, but unless there's a break statement, the matching *case* is not the only *case* code that runs.
- The default keyword should be used in a switch statement if you want to execute some code when none of the *case* values match the conditional value.
- The default block can be located anywhere in the *switch* block, so if no *case* matches, the default block will be entered, and if the *default* does not contain a *break*, then code will continue to execute (fall-through) to the end of the *switch* or until the break statement is encountered.

Writing Code Using Loops

- A *for* statement does not require any arguments in the declaration, but has three parts: declaration and/or initialization, boolean evaluation, and the iteration expression.

- If a variable is incremented or evaluated within a *for* loop, it must be declared before the loop, or within *for* loop declaration.
- A variable declared (not just initialized) within the *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 in the first part of the *for* loop declaration; each variable initialization must be separated by a comma.
- 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-while* loop will enter the body of the loop at least once, even if the test condition is not met.

Using break and continue

- An unlabeled break statement will cause the current iteration of the innermost looping construct to stop and the next line of code following the loop to be executed.
- An unlabeled continue statement will cause the current iteration of the innermost loop to stop, and the condition of that loop to be checked, and if the condition is met, perform the loop 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.
- If a continue statement is used in a for loop, the iteration statement is executed, and the condition is checked again.

Catching an Exception Using try and catch

- 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 the throws keyword, 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, and you're free 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(), in which case the JVM will not start your finally block.
- 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, 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.

    For example, if you have a catch clause for both IOException and Exception, you must put the catch for IOException first (in order, top to bottom 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 (because it sees that the more specific exception will be caught first by the more general catch).

Working with the Assertion Mechanism

- Assertions give you a way to test your assumptions during development and debugging.
- Assertions are typically enabled during testing but disabled during deployment.

- You can use assert as a keyword (as of version 1.4) or an identifier, but not both together. To compile older code that uses assert as an identifier (for example, a method name), use the -source 1.3 command-line flag to javac.
- Assertions are disabled at runtime by default. To enable them, use a command-line flag -ea or -enableassertions.
- You can selectively disable assertions using the -da or -disableassertions flag.
- If you enable or disable assertions using the flag without any arguments,you're enabling or disabling assertions in general. You can combine enabling and disabling switches to have assertions enabled for some classes and/or packages, but not others.
- You can enable or disable assertions in the system classes with the -esa or -dsa flags.
- You can enable and disable assertions on a class-by-class basis, using the following syntax:

  ```
  java -ea -da:MyClass TestClass
  ```
- You can enable and disable assertions on a package basis, and any package you specify also includes any subpackages (packages further down the directory hierarchy).
- Do not use assertions to validate arguments to public methods.
- Do not use assert expressions that cause side effects. Assertions aren't guaranteed to always run, so you don't want behavior that changes depending on whether assertions are enabled.
- Do use assertions—even in public methods—to validate that a particular code block will never be reached. You can use assert false; for code that should never be reached, so that an assertion error is thrown immediately if the assert statement is executed.
- Do not use assert expressions that can cause side effects.