Objectives 2

Flow Control

Writing Code Using if and switch Statements (Objectives 2.1)

- The only legal expression in an if statement is a boolean expression, in other words 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.

Writing Code Using if and switch Statements (Objectives 2.1) [contd.]

- switch statements can evaluate only to enums or the byte, short, int, and char data types. You can't say,
  - long s = 30;
  - switch(s) { }
- The case constant must be a literal or final variable, or a constant expression, including an enum. You cannot have a case that includes a nonfinal variable, or a range of values.
- If the condition in a switch statement matches a case constant, execution will run through all code in the switch following the matching case statement until a break statement 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.

Writing Code Using if and switch Statements (Objectives 2.1) [contd.]

- The default keyword should be used in a switch statement if you want to run 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 (Objective 2.2)

- 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).

Writing Code Using Loops (Objective 2.2) [contd.]

- 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.

Writing Code Using Loops (Objective 2.2) [contd.]

- 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 (Objective 2.2)

- 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.

Working with the Assertion Mechanism (Objective 2.3)

- 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.

Working with the Assertion Mechanism (Objective 2.3) [contd.]

- Selectively disable assertions by 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 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-by-package basis, and any package you specify also includes any subpackages (packages further down the directory hierarchy).

Working with the Assertion Mechanism (Objective 2.3) [contd.]

- 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, and 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.

Handling Exceptions (Objectives 2.4, 2.5, and 2.6)

- 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.

Handling Exceptions (Objectives 2.4, 2.5, and 2.6) [contd.]

- 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().

Handling Exceptions (Objectives 2.4, 2.5, and 2.6) [contd.]

- 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.

Handling Exceptions (Objectives 2.4, 2.5, and 2.6) [contd.]

- 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, some by the JVM.