

If you need to validate `public` method arguments, you'll probably use exceptions to throw, say, an `IllegalArgumentException` if the values passed to the `public` method are invalid.

### **Do Use Assertions to Validate Arguments to a Private Method**

If you write a `private` method, you almost certainly wrote (or control) any code that calls it. When you assume that the logic in code calling your `private` method is correct, you can test that assumption with an assertion as follows:

```
private void doMore(int x) {
    assert (x > 0);
    // do things with x
}
```

The only difference that matters between the preceding example and the one before it is the access modifier. So, do enforce constraints on `private` methods' arguments, but do not enforce constraints on `public` methods. You're certainly free to compile assertion code with an inappropriate validation of `public` arguments, but for the exam (and real life) you need to know that you shouldn't do it.

### **Don't Use Assertions to Validate Command-Line Arguments**

This is really just a special case of the "Do not use assertions to validate arguments to a public method" rule. If your program requires command-line arguments, you'll probably use the exception mechanism to enforce them.

### **Do Use Assertions, Even in Public Methods, to Check for Cases that You Know Are Never, Ever Supposed to Happen**

This can include code blocks that should never be reached, including the default of a `switch` statement as follows:

```
switch(x) {
    case 1: y = 3; break;
    case 2: y = 9; break;
    case 3: y = 27; break;
    default: assert false; // we're never supposed to get here!
}
```

If you assume that a particular code block won't be reached, as in the preceding example where you assert that `x` must be either 1, 2, or 3, then you can use `assert false` to cause an `AssertionError` to be thrown immediately if you ever do reach that code. So in the `switch` example, we're not performing a boolean test—we've

already asserted that we should never be there, so just getting to that point is an automatic failure of our assertion/assumption.

### Don't Use Assert Expressions that Can Cause Side Effects!

The following would be a very bad idea:

```
public void doStuff() {
    assert (modifyThings());
    // continues on
}
public boolean modifyThings() {
    y = x++;
    return true;
}
```

The rule is, an `assert` expression should leave the program in the same state it was in before the expression! Think about it. `assert` expressions aren't guaranteed to always run, so you don't want your code to behave differently depending on whether assertions are enabled. Assertions must not cause any side effects. If assertions are enabled, the only change to the way your program runs is that an `AssertionError` can be thrown if one of your assertions (think: *assumptions*) turns out to be false.



**Using assertions that cause side effects can cause some of the most maddening and hard-to-find bugs known to man! When a hot tempered Q.A. analyst is screaming at you that your code doesn't work, trotting out the old "well it works on MY machine" excuse won't get you very far.**

## CERTIFICATION SUMMARY

This chapter covered a lot of ground, all of which involves 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`.

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 is new to Java 6), 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 will have 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 that 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 public interface. 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 to either 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.

Exceptions can be generated by the JVM, or by a programmer.

Assertions, added to the language in version 1.4, are a useful debugging tool. You learned how you can use them for testing, by enabling them, but keep them disabled when the application is deployed. If you have older Java code that uses the word `assert` as an identifier, then you won't be able to use assertions, and you must recompile your older code using the `-source 1.3` flag. Remember that as of Java 6, assertions are compiled as a keyword by default, but must be enabled explicitly at runtime.

You learned how `assert` statements always include a boolean expression, and if the expression is `true` the code continues on, but if the expression is `false`, an `AssertionError` is thrown. If you use the two-expression `assert` statement, then the second expression is evaluated, converted to a `String` representation and inserted into the stack trace to give you a little more debugging info. Finally, you saw why assertions should not be used to enforce arguments to `public` methods, and why `assert` expressions must not contain side effects!



## TWO-MINUTE DRILL

Here are some of the key points from each certification objective in this chapter. You might want to loop through them several times.

### Writing Code Using if and switch Statements (Obj. 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.
- ❑ `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 non-`final` 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.
- ❑ 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).
- ❑ 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 6), 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` (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.

## 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.
- ❑ 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, 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, some by the JVM.

### 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`.
- ❑ 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).
- ❑ 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.

## SELF TEST

### 1. Given two files:

```
1. class One {
2.     public static void main(String[] args) {
3.         int assert = 0;
4.     }
5. }
```

```
1. class Two {
2.     public static void main(String[] args) {
3.         assert(false);
4.     }
5. }
```

And the four command-line invocations:

```
javac -source 1.3 One.java
javac -source 1.4 One.java
javac -source 1.3 Two.java
javac -source 1.4 Two.java
```

What is the result? (Choose all that apply.)

- A. Only one compilation will succeed
- B. Exactly two compilations will succeed
- C. Exactly three compilations will succeed
- D. All four compilations will succeed
- E. No compiler warnings will be produced
- F. At least one compiler warning will be produced

### 2. Given:

```
class Plane {
    static String s = "-";
    public static void main(String[] args) {
        new Plane().s1();
        System.out.println(s);
    }
    void s1() {
        try { s2(); }
        catch (Exception e) { s += "c"; }
    }
    void s2() throws Exception {
```

## 402 Chapter 5: Flow Control, Exceptions, and Assertions

```
s3(); s += "2";  
s3(); s += "2b";  
}  
void s3() throws Exception {  
    throw new Exception();  
}  
}
```

What is the result?

- A. -
- B. -c
- C. -c2
- D. -2c
- E. -c22b
- F. -2c2b
- G. -2c2bc
- H. Compilation fails

3. Given:

```
try { int x = Integer.parseInt("two"); }
```

Which could be used to create an appropriate catch block? (Choose all that apply.)

- A. `ClassCastException`
- B. `IllegalStateException`
- C. `NumberFormatException`
- D. `IllegalArgumentException`
- E. `ExceptionInInitializerError`
- F. `ArrayIndexOutOfBoundsException`

4. Which are true? (Choose all that apply.)

- A. It is appropriate to use assertions to validate arguments to methods marked `public`
- B. It is appropriate to catch and handle assertion errors
- C. It is NOT appropriate to use assertions to validate command-line arguments
- D. It is appropriate to use assertions to generate alerts when you reach code that should not be reachable
- E. It is NOT appropriate for assertions to change a program's state

**5.** Given:

```

1. class Loopy {
2.     public static void main(String[] args) {
3.         int[] x = {7,6,5,4,3,2,1};
4.         // insert code here
5.         System.out.print(y + " ");
6.     }
7. }
8. }

```

Which, inserted independently at line 4, compiles? (Choose all that apply.)

- A. `for(int y : x) {`
- B. `for(x : int y) {`
- C. `int y = 0; for(y : x) {`
- D. `for(int y=0, z=0; z<x.length; z++) { y = x[z];`
- E. `for(int y=0, int z=0; z<x.length; z++) { y = x[z];`
- F. `int y = 0; for(int z=0; z<x.length; z++) { y = x[z];`

**6.** Given:

```

class Emu {
    static String s = "-";
    public static void main(String[] args) {
        try {
            throw new Exception();
        } catch (Exception e) {
            try {
                try { throw new Exception();
                } catch (Exception ex) { s += "ic "; }
                throw new Exception();
            } catch (Exception x) { s += "mc "; }
            finally { s += "mf "; }
        } finally { s += "of "; }
        System.out.println(s);
    } }

```

What is the result?

- A. -ic of
- B. -mf of
- C. -mc mf

## 404 Chapter 5: Flow Control, Exceptions, and Assertions

- D. -ic mf of
- E. -ic mc mf of
- F. -ic mc of mf
- G. Compilation fails

### 7. Given:

```
3. class SubException extends Exception { }
4. class SubSubException extends SubException { }
5.
6. public class CC { void doStuff() throws SubException { } }
7.
8. class CC2 extends CC { void doStuff() throws SubSubException { } }
9.
10. class CC3 extends CC { void doStuff() throws Exception { } }
11.
12. class CC4 extends CC { void doStuff(int x) throws Exception { } }
13.
14. class CC5 extends CC { void doStuff() { } }
```

What is the result? (Choose all that apply.)

- A. Compilation succeeds
- B. Compilation fails due to an error on line 8
- C. Compilation fails due to an error on line 10
- D. Compilation fails due to an error on line 12
- E. Compilation fails due to an error on line 14

### 8. Given:

```
3. public class Ebb {
4.     static int x = 7;
5.     public static void main(String[] args) {
6.         String s = "";
7.         for(int y = 0; y < 3; y++) {
8.             x++;
9.             switch(x) {
10.                case 8: s += "8 ";
11.                case 9: s += "9 ";
12.                case 10: { s+= "10 "; break; }
13.                default: s += "d ";
14.                case 13: s+= "13 ";
```