JAVA™ 2

SUN™ CERTIFIED PROGRAMMER & DEVELOPER

4

Flow Control, Exceptions, and Assertions

CERTIFICATION OBJECTIVES

- Writing Code Using *if* and *switch* Statements
- Writing Code Using Loops
- Handling Exceptions
- Working with the Assertion Mechanism

✓ Two-Minute Drill

Q&A Self Test

Can you imagine trying to write code using a language that didn't give you a way to execute statements conditionally? In other words, a language that didn't let you say, "If this thing over here is *true*, then I want this thing to happen; otherwise, do this other thing instead." Flow control is a key part of most any useful programming language, and Java offers several ways to do it. Some, like *if* statements and *for* loops, are common to most languages. But Java also throws in a couple flow control features you might not have used before—exceptions and assertions.

The *if* statement and the *switch* statement are types of conditional/decision controls that allow your program to perform differently at a "fork in the road," depending on the result of a logical test. Java also provides three different looping constructs—`for`, `while`, and `do-while`—so you can execute the same code over and over again depending on some condition being *true*. Exceptions give you a clean, simple way to organize code that deals with problems that might crop up at runtime. Finally, the assertion mechanism, added to the language with version 1.4, gives you a way to do debugging checks on conditions you expect to smoke out while developing, when you don't necessarily need or want the runtime overhead associated with exception handling.

With these tools, you can build a robust program that can handle any logical situation with grace. Expect to see a wide range of questions on the exam that include flow control as part of the question code, even on questions that aren't testing your knowledge of flow control.

CERTIFICATION OBJECTIVE

Writing Code Using if and switch Statements (Exam Objective 2.1)

Write code using if *and* switch *statements and identify legal argument types for these statements.*

The *if* and *switch* statements are commonly referred to as *decision statements*. When you use decision statements in your program, you're asking the program to evaluate a given expression to determine which course of action to take. We'll look at the *if* statement first.

if-else Branching

The basic format of an `if` statement is as follows:

```
if (booleanExpression) {
    System.out.println("Inside if statement");
}
```

The expression in parentheses *must* evaluate to a boolean `true` or `false` result. Typically you're testing something to see if it's *true*, and then running a code block (one or more statements) if it *is true*, and (optionally) another block of code if it isn't. We consider it good practice to enclose the blocks within curly braces, even if there's only one statement in the block. The following code demonstrates a legal `if` statement:

```
if (x > 3) {
   System.out.println("x is greater than 3");
} else {
   System.out.println("x is not greater than 3");
}
```

The `else` block is optional, so you can also use the following:

```
if (x > 3) {
   y = 2;
}
z += 8;
a = y + x;
```

The preceding code will assign 2 to *y* if the test succeeds (meaning *x* really is greater than 3), but the other two lines will execute regardless.

Even the curly braces are optional if you have only one statement to execute within the body of the conditional block. The following code example is legal (although not recommended for readability):

```
if (x > 3)
   y = 2;
z += 8;
a = y + x;
```

Be careful with code like this, because you might think it should read as, "*If x* is greater than 3, *then* set *y* to 2, *z* to *z* + 8, and *a* to *y* + *x*." But the last two lines are

going to execute no matter what! They aren't part of the conditional flow. You might find it even more misleading if the code were indented as follows:

```
if (x > 3)
   y = 2;
   z += 8;
   a = y + x;
```

You might have a need to nest *if-else* statements (although, again, not recommended for readability, so nested *if* tests should be kept to a minimum). You can set up an *if-else* statement to test for multiple conditions. The following example uses two conditions so that *if* the first test fails, we want to perform a second test before deciding what to do:

```
if (price < 300) {
   buyProduct();
} else {
   if (price < 400) {
      getApproval();
   }
   else {
      dontBuyProduct();
   }
}
```

Sometimes you can have a problem figuring out which *if* your *else* goes to, as follows:

```
if (exam.done())
if (exam.getScore() < 0.61)
System.out.println("Try again.");
// Which if does this belong to?
else System.out.println("Java master!");
```

We intentionally left out the indenting in this piece of code so it doesn't give clues as to which *if* statement the *else* belongs to. Did you figure it out? Java law decrees that an *else* clause belongs to the innermost *if* statement to which it might possibly belong (in other words, the closest preceding `if` that doesn't have an `else`). In the case of the preceding example, the `else` belongs to the *second* `if` statement in the listing. With proper indenting, it would look like this:

```
if (exam.done())
   if (exam.getScore() < 0.61)
      System.out.println("Try again.");
```

```
  // Which if does this belong to?
else
      System.out.println("Java master!");
```

Following our coding conventions by using curly braces, it would be even easier to read:

```
if (exam.done()) {
   if (exam.getScore() < 0.61) {
      System.out.println("Try again.");
   // Which if does this belong to?
   } else {
      System.out.println("Java master!");
   }
}
```

Don't get your hopes up about the exam questions being all nice and indented properly. Some exam takers even have a slogan for the way questions are presented on the exam: *anything that* can *be made more confusing,* will *be.*

exam Watch

Be prepared for questions that not only fail to indent nicely, but intentionally indent in a misleading way: Pay close attention for misdirection like the following example:

```
if (exam.done())
   if (exam.getScore() < 0.61)
      System.out.println("Try again.");
else
   System.out.println("Java master!"); // Hmmmmm… now where does it belong?
```

Of course, the preceding code is exactly the same as the previous two examples, except for the way it *looks.*

Legal Arguments for if Statements

if statements can test against only a boolean. Any expression that resolves down to a boolean is fine, but some of the expressions can be complex. Assume doStuff() returns true,

```
int y = 5;
int x = 2;
if ((((x > 3) && (y < 2)) | doStuff()) {
System.out.println("true");
}
```

which prints

```
true
```

You can read the preceding code as, "If both (*x* > 3) and (*y* < 2) are *true*, or if the result of `doStuff()` is *true*, then print "true." So basically, if just `doStuff()` alone is *true*, we'll still get "true." If `doStuff()` is false, though, then both (*x* > 3) and (*y* < 2) will have to be *true* in order to print "true."

The preceding code is even more complex if you leave off one set of parentheses as follows,

```
int y = 5;
int x = 2;
if ((x > 3) && (y < 2) | doStuff()) {
System.out.println("true");
}
```

which now prints…nothing! Because the preceding code (with one less set of parentheses) evaluates as though you were saying, "If (*x* > 3) is *true*, *and* either (*y* < 2) or the result of `doStuff()` is *true*, then print "true." So if (*x* > 3) is not *true*, no point in looking at the rest of the expression." Because of the short-circuit && and the fact that at runtime the expression is evaluated as though there were parentheses around `((y< 2) | doStuff())`, it reads as though both the test before the && (*x* > 3) *and* then the rest of the expression *after* the `&& (y<2 | doStuff())` must be *true*.

Remember that the only legal argument to an *if* test is a boolean. Table 4-1 lists illegal arguments that might look tempting, compared with a modification to make each argument legal.

exam Watch

One common mistake programmers make (and that can be difficult to spot), is assigning a `boolean` variable when you meant to test a `boolean` variable. Look out for code like the following:

```
boolean boo = false;
if (boo = true) { }
```

You might think one of three things:

1. ***The code compiles and runs fine, and the if test fails because boo is false.***
2. ***The code won't compile because you're using an assignment (=) rather than an equality test (==).***
3. ***The code compiles and runs fine and the if test succeeds because boo is set to `true` (rather than tested for `true`) in the if argument!***

Well, number 3 is correct. Pointless, but correct. Given that the result of any assignment is the value of the variable after the assignment, the expression (`boo = true`) has a result of `true`. Hence, the *if* test succeeds. But the only variable that can be assigned (rather than tested against something else) is a boolean; all other assignments will result in something nonboolean, so they're not legal, as in the following:

```
int x = 3;
if (x = 5) { }  // Won't compile because x is not a boolean!
```

Because *if* tests require boolean expressions, you need to be really solid on *both* logical operators and *if* test syntax and semantics.

switch Statements

Another way to simulate the use of multiple *if* statements is with the *switch* statement. Take a look at the following *if-else* code, and notice how confusing it can be to have nested *if* tests, even just a few levels deep:

```
int x = 3;
if(x == 1) {
   System.out.println("x equals 1");
}
else if(x == 2) {
      System.out.println("x equals 2");
   }
   else if(x == 3) {
         System.out.println("x equals 3");
      }
      else {
         System.out.println("No idea what x is");
      }
```

TABLE 4-1 Illegal and Legal Arguments to *if*

Illegal Arguments to *if*	Legal Arguments to *if*
`int x = 1;` `if (x) { }`	`int x = 1;` `if (x == 1) { }`
`if (0) { }`	`if (false)`
`if (x = 6)`	`if (x == 6)`

Now let's see the same functionality represented in a *switch* construct:

```
int x = 3;
switch (x) {
   case 1:
      System.out.println("x is equal to 1");
      break;
   case 2:
      System.out.println("x is equal to 2");
      break;
   case 3:
      System.out.println("x is equal to 3");
      break;
   default:
      System.out.println("Still no idea what x is");
}
```

Legal Arguments to switch and case

The only type that a *switch* can evaluate is the primitive `int`! That means only variables and values that can be automatically promoted (in other words, implicitly cast) to an `int` are acceptable. So you can switch on any of the following, but nothing else: `byte`, `short`, `char`, `int`.

You won't be able to compile if you use anything else, including the remaining numeric types of `long`, `float`, and `double`.

The only argument a *case* can evaluate is one of the same type as *switch* can use, with one additional—*and big*—constraint: *the* case *argument must be final!* The *case* argument has to be resolved at compile time, so that means you can use *only a constant final* variable that is assigned a literal value. It is not enough to be final, it must be a compile time constant. For example:

```
final int a = 1;
final int b;
int x = 0;
switch (x) {
  case a:     // ok
  case b:     // compiler error
              // thx to John Paverd !
```

Also, the switch *can only check for equality*. This means that the other relational operators such as *greater than* are rendered unusable in a *case*. The following is an example of a valid expression using a method invocation in a *switch* statement. Note that for this code to be legal, the method being invoked on the object reference must return a value compatible with an `int`.

```
String s = "xyz";
switch (s.length()) {
```

```
    case 1:
       System.out.println("length is one");
       break;
    case 2:
       System.out.println("length is two");
       break;
    case 3:
       System.out.println("length is three");
       break;
    default:
       System.out.println("no match");
}
```

The following example uses final variables in a *case* statement. Note that if the `final` keyword is omitted, this code will not compile.

```
final int one = 1;
final int two = 2;
int x = 1;
switch (x) {
   case one:  System.out.println("one");
              break;
   case two:  System.out.println("two");
              break;
}
```

One other rule you might not expect involves the question, "What happens if I switch on a variable smaller than an `int`?" Look at the following `switch` example:

```
byte g = 2;
switch(g) {
case 23:
case 128:
}
```

This code won't compile. Although the *switch* argument is legal—a `byte` is implicitly cast to an `int`—the second case argument (128) is too large for a `byte`, and the compiler knows it! Attempting to compile the preceding example gives you an error:

```
Test.java:6: possible loss of precision
found   : int
required: byte
    case 128:
         ^
```

It's also illegal to have more than one *case* label using the same value. For example, the following block of code won't compile because it uses two *cases* with the same value of 80:

```
int temp = 90;
switch(temp) {
   case 80 :
      System.out.println("80");
      break;
   case 80 :
      System.out.println("80");
      break;
   case 90:
      System.out.println("90");
      break;
   default:
      System.out.println("default");
}
```

exam Watch

Look for any violation of the rules for *switch* ***and*** *case* ***arguments. For example, you might find illegal examples like the following three snippets:***

```
Integer in = new Integer(4);
switch (in) { }

==================
switch(x) {
   case 0 {
      y = 7;
    }
 }
==================
switch(x) {
   0: { }
   1: { }
}
```

In the first example, you can't switch on an Integer object, only an `int` ***primitive. In the second example, the case uses a curly brace and omits the colon. The third example omits the keyword*** `case`**.**

Default, Break, and Fall-Through in switch Blocks

When the program encounters the keyword `break` during the execution of a *switch* statement, execution will immediately move out of the `switch` block to the next statement *after* the `switch`. If break is omitted, the program just keeps executing the different *case* blocks until either a `break` is found or the *switch* statement ends. Examine the following code:

```
int x = 1;
switch(x) {
   case 1:  System.out.println("x is one");
   case 2:  System.out.println("x is two");
   case 3:  System.out.println("x is three");
}
System.out.println("out of the switch");
```

The code will print the following:

```
x is one
x is two
x is three
out of the switch
```

This combination occurs because the code didn't hit a *break* statement; thus, execution just kept dropping down through each `case` until the end. This dropping down is actually called "fall through," because of the way execution falls from one `case` to the next. *Think of the matching `case` as simply your entry point into the `switch` block!* In other words, you must *not* think of it as, "Find the matching case, execute just that code, and get out." That's not how it works. If you do want that "just the matching code" behavior, you'll insert a *break* into each `case` as follows:

```
int x = 1;
switch(x) {
   case 1:  {
      System.out.println("x is one");
      break;
   }
   case 2:  {
      System.out.println("x is two");
      break;
   }
   case 3:  {
      System.out.println("x is two");
```

```
        break;
    }
}
System.out.println("out of the switch");
```

Running the preceding code, now that we've added the *break* statements, will print

```
x is one
out of the switch
```

and that's it. We entered into the *switch* block at `case` 1. Because it matched the `switch()` argument, we got the `println` statement, then hit the `break` and jumped to the end of the *switch*.

Another way to think of this fall-through logic is shown in the following code:

```
int x = someNumberBetweenOneAndTen;

switch (x) {
   case 2:
   case 4:
   case 6:
   case 8:
   case 10: {
      System.out.println("x is an even number");
      break;
   }
}
```

This *switch* statement will print "x is an even number" or nothing, depending on whether the number is between one and ten and is odd or even. For example, if *x* is 4, execution will begin at `case` 4, but then fall down through 6, 8, and 10, where it prints and then breaks. The `break` at `case` 10, by the way, is not needed; we're already at the end of the *switch* anyway.

The Default Case

What if, using the preceding code, you wanted to print "x is an odd number" if none of the cases (the even numbers) matched? You couldn't put it after the *switch* statement, or even as the last *case* in the *switch*, because in both of those situations it would always print "x is an odd number." To get this behavior, you'll use the `default` keyword. (By the way, if you've wondered why there is a `default` keyword even though we don't use a modifier for default access control, now you'll

see that the `default` keyword is used for a completely different purpose.) The only change we need to make is to add the *default case* to the preceding code:

```
int x = someNumberBetweenOneAndTen;

switch (x) {
   case 2:
   case 4:
   case 6:
   case 8:
   case 10: {
      System.out.println("x is an even number");
      break;
   }
   default: System.out.println("x is an odd number");
}
```

exam Watch

The `default` case doesn't have to come at the end of the switch. Look for it in strange places such as the following:

```
int x = 2;
switch (x) {
   case 2:  System.out.println("2");
   default: System.out.println("default");
   case 3: System.out.println("3");
   case 4: System.out.println("4");
}
```

Running the preceding code prints

```
2
default
3
4
```

and if we modify it so that the only match is the `default` case:

```
int x = 7;
switch (x) {
   case 2:  System.out.println("2");
   default: System.out.println("default");
   case 3: System.out.println("3");
   case 4: System.out.println("4");
}
```

Running the preceding code prints

```
default
3
4
```

The rule to remember is `default` works just like any other case for fall-through!

EXERCISE 4-1

Creating a switch-case Statement

Try creating a *switch-case* statement using a `char` value as the *case*. Include a default behavior if none of the `char` values match.

1. Make sure a `char` variable is declared before the *switch* statement.
2. Each *case* statement should be followed by a `break`.
3. The `default` value can be located at the end, middle, or top.

CERTIFICATION OBJECTIVE

Writing Code Using Loops (Exam Objective 2.2)

Write code using all forms of loops including labeled and unlabeled, use of break and continue, and state the values taken by loop counter variables during and after loop execution.

Java loops come in three flavors: *while*, *do-while*, and *for*. All three let you repeat a block of code as long as some condition is *true*, or for a specific number of iterations. You're probably familiar with loops from other languages, so even if you're somewhat new to Java, these won't be a problem to learn.

Using while Loops

The *while* loop is good for scenarios where you don't know how many times block or statement should repeat, but you want it to continue as long as some condition is *true*. A *while* statement looks like this:

```
int x = 2;
while(x == 2) {
   System.out.println(x);
   ++x;
}
```

In this case, as in all loops, the expression (test) must evaluate to a boolean result. Any variables used in the expression of a *while* loop must be declared before the expression is evaluated. In other words, you can't say

```
while (int x = 2) { }
```

Then again, why would you? Instead of testing the variable, you'd be declaring and initializing it, so it would always have the exact same value. Not much of a test condition!

The body of the *while* loop will *only* execute if the condition results in a *true* value. Once inside the loop, the loop body will repeat until the condition is no longer met and evaluates to *false*. In the previous example, program control will enter the loop body because *x* is equal to 2. However, *x* is incremented in the loop, so when the condition is checked again it will evaluate to *false* and exit the loop.

The key point to remember about a *while* loop is that it might not *ever* run. If the test expression is *false* the first time the *while* expression is checked, the loop body will be skipped and the program will begin executing at the first statement *after* the *while* loop. Look at the following example:

```
int x = 8;
while (x > 8) {
   System.out.println("in the loop");
   x = 10;
}
System.out.println("past the loop");
```

Running this code produces

```
past the loop
```

Although the test variable *x* is incremented within the *while* loop body, the program will never see it. This is in contrast to the *do-while* loop that executes the loop body once, and *then* does the first test.

Using do-while Loops

The following shows a `do-while` statement in action:

```
do {
   System.out.println("Inside loop");
} while(false);
```

The `System.out.println()` statement will print once, *even though the expression evaluates to false*. The *do-while* loop will *always* run the code in the loop *body at least once*. Be sure to note the use of the semicolon at the end of the *while* expression.

exam Watch

As with if tests, look for while loops (and the while test in a do-while loop) with an expression that does not resolve to a boolean. Take a look at the following examples of legal and illegal `while` expressions:

```
int x = 1;
while (x) { }  // Won't compile; x is not a boolean
while (x = 5) { } // Won't compile; resolves to 5 (result of assignment)
while (x == 5) { }  // Legal, equality test
while (true) { } // Legal
```

Using for Loops

The *for* loop is especially useful for flow control when you already know how many times you need to execute the statements in the loop's block. The *for* loop declaration has three main parts, besides the body of the loop:

- Declaration and initialization of variables
- The boolean expression (conditional test)
- The iteration expression

Each of the three *for* declaration parts is separated by a semicolon. The following two examples demonstrate the *for* loop. The first example shows the parts of a *for* loop in a pseudocode form, and the second shows typical syntax of the loop.

```
for (/*Initialization*/ ; /*Condition*/ ;  /* Iteration */) {
   /* loop body */
}

for (int i = 0; i<10; i++) {
   System.out.println("i is " + i);
}
```

Declaration and Initialization

The first part of the *for* statement lets you declare and initialize zero, one, or multiple variables of the same type inside the parentheses after the `for` keyword. If you declare more than one variable of the same type, then you'll need to separate them with commas as follows:

```
for (int x = 10, y = 3; y > 3; y++) { }
```

The declaration and initialization happens before anything else in a for *loop*. And whereas the other two parts—the boolean test and the iteration expression—will run with each iteration of the loop, the declaration and initialization happens just once, at the very beginning. You also must know that *the scope of variables declared in the* for *loop ends with the* for *loop*! The following demonstrates this:

```
for (int x = 1; x < 2; x++) {
   System.out.println(x);  // Legal
}
System.out.println(x);  // Not Legal! x is now out of scope and
can't be accessed.
```

If you try to compile this, you'll get

```
Test.java:19: cannot resolve symbol
symbol  : variable x
location: class Test
  System.out.println(x);
                     ^
```

Conditional (boolean) Expression

The next section that executes is the conditional expression, which (like all other conditional tests) *must* evaluate to a boolean value. You can have only one logical expression, but it can be very complex. Look out for code that uses logical expressions like this:

```
for (int x = 0; ((((x < 10) && (y-- > 2)) | x == 3)); x++) { }
```

The preceding code is legal, but the following is *not*:

```
for (int x = 0; (x > 5), (y < 2); x++) { } // too many
                                           //expressions
```

The compiler will let you know the problem:

```
TestLong.java:20: ';' expected
for (int x = 0; (x > 5), (y < 2); x++) { }
                       ^
```

The rule to remember is this: *You can have only one test expression.* In other words, you can't use multiple tests separated by commas, even though the *other* two parts of a *for* statement can have multiple parts.

Iteration Expression

After each execution of the body of the *for* loop, the iteration expression is executed. This part is where you get to say what you want to happen with each iteration of the loop. Remember that it always happens *after* the loop body runs! Look at the following:

```
for (int x = 0; x < 1; x++) {
    // body code here
}
```

The preceding loop executes just once. The first time into the loop *x* is set to 0, then *x* is tested to see if it's less than 1 (which it is), and then the body of the loop executes. After the body of the loop runs, the iteration expression runs, incrementing *x* by 1. Next, the conditional test is checked, and since the result is now *false*, execution jumps to *below* the `for` loop and continues on. Keep in mind that this *iteration expression is always the last thing that happens*! So although the body may never execute again, the iteration expression *always* runs at the end of the loop block, as long as no

other code within the loop causes execution to leave the loop. For example, a `break`, `return`, exception, or `System.exit()` will all cause a loop to terminate abruptly, without running the iteration expression. Look at the following code:

```
static boolean doStuff() {
   for (int x = 0; x < 3; x++) {
      System.out.println("in for loop");
      return true;
   }
   return true;
}
```

Running this code produces

```
in for loop
```

The statement only prints once, because a `return` causes execution to leave not just the current iteration of a loop, but the entire method. So the iteration expression never runs in that case. Table 4-2 lists the causes and results of abrupt loop termination.

for Loop Issues

None of the three sections of the `for` declaration are required! The following example is perfectly legal (although not necessarily good practice):

```
for( ; ; ) {
   System.out.println("Inside an endless loop");
}
```

In the preceding example, all the declaration parts are left out so it will act like an endless loop. For the exam, it's important to know that with the absence of the

TABLE 4-2 Causes of Early Loop Termination

Code in Loop	What Happens
`break`	Execution jumps immediately to the first statement after the *for* loop.
`return`	Execution immediately jumps back to the calling method.
`System.exit()`	All program execution stops; the VM shuts down.

initialization and increment sections, the loop will act like a *while* loop. The following example demonstrates how this is accomplished:

```
int i = 0;

for (;i<10;) {
   i++;
   //do some other work
}
```

The next example demonstrates a *for* loop with multiple variables in play. A comma separates the variables, and they must be of the same type. Remember that the variables declared in the *for* statement are all local to the *for* loop, and can't be used outside the scope of the loop.

```
for (int i = 0,j = 0; (i<10) && (j<10); i++, j++) {
   System.out.println("i is " + i + "j is " +j);
}
```

exam Watch

Variable scope plays a large role in the exam. You need to know that a variable declared in the for loop can't be used beyond the for loop. But a variable only initialized in the for statement (but declared earlier) can be used beyond the loop. For example, the following is legal,

```
int x = 3;
for (x = 12; x < 20, x++) { }
System.out.println(x);
```

while this is not,

```
for (int x = 3; x < 20; x++) { }System.out.println(x);
```

The last thing to note is that *all three sections of the* for *loop are independent of each other*. The three expressions in the *for* statement don't need to operate on the same variables, although they typically do. But even the iterator expression, which many mistakenly call the "increment expression," doesn't need to increment or set anything; you can put in virtually any arbitrary code statements that you want to happen with each iteration of the loop. Look at the following:

```
int b = 3;
for (int a = 1;  b != 1; System.out.println("iterate")) {
   b = b - a;
}
```

The preceding code prints

```
iterate
iterate
```

exam Watch

Most questions in the new (1.4) exam list "Compilation fails" and "An exception occurs at runtime" as possible answers. This makes it more difficult because you can't simply work through the behavior of the code. You must first make sure the code isn't violating any fundamental rules that will lead to compiler error, and then look for possible exceptions, and only after you've satisfied those two should you dig into the logic and flow of the code in the question.

Using break and continue in for Loops

The `break` and `continue` keywords are used to stop either the entire loop (*break*) or just the current iteration (*continue*). Typically if you're using `break` or `continue`, you'll do an *if* test within the loop, and if some condition becomes *true* (or *false* depending on the program), you want to get out immediately. The difference between them is whether or not you continue with a new iteration or jump to the first statement below the loop and continue from there.

exam Watch

`continue` statements must be inside a loop; otherwise, you'll get a compiler error. `break` statements must be used inside either a loop or switch statement. (Note: This does not apply to labeled `break` statements.)

The `break` statement causes the program to stop execution of the innermost looping and start processing the next line of code after the block.

The `continue` statement causes only the current *iteration* of the innermost loop to cease and the next iteration of the same loop to start if the condition of the loop is met. When using a `continue` statement with a *for* loop, you need to consider the effects that *continue* has on the loop iteration. Examine the following code, which will be explained afterward.

```
for (int i = 0; i < 10; i++) {
   System.out.println("Inside loop");
   continue;
}
```

The question is, is this an endless loop? The answer is no. When the `continue` statement is hit, the iteration expression still runs! It runs just as though the current iteration ended "in the natural way." So in the preceding example, *i* will still increment before the condition (*i* < 10) is checked again. Most of the time, a `continue` is used within an `if` test as follows:

```
for (int i = 0; i < 10; i++) {
   System.out.println("Inside loop");
   if (foo.doStuff() == 5) {
     continue;
   }
   // more loop code, that won't be reached when the above if
   //test is true
}
```

Unlabeled Statements

Both the `break` statement and the `continue` statement can be unlabeled or labeled. Although it's far more common to use `break` and `continue` unlabeled, the exam expects you to know how labeled `break` and `continue` work. As stated before, a `break` statement (unlabeled) will exit out of the innermost looping construct and proceed with the next line of code beyond the loop block. The following example demonstrates a `break` statement:

```
boolean problem = true;
while (true) {
   if (problem) {
      System.out.println("There was a problem");
      break;
   }
}
//next line of code
```

In the previous example, the `break` statement is unlabeled. The following is another example of an unlabeled `continue` statement:

```
while (!EOF) {
   //read a field from a file
   if (there was a problem) {
      //move to the next field in the file
      continue;
   }
}
```

In this example, there is a file being read from one field at a time. When an error is encountered, the program moves to the next field in the file and uses the `continue` statement to go back into the loop (if it is not at the end of the file) and keeps reading the various fields. If the `break` command were used instead, the code would stop reading the file once the error occurred and move on to the next line of code. The `continue` statement gives you a way to say, "This particular iteration of the loop needs to stop, but not the whole loop itself. I just don't want the rest of the code in this iteration to finish, so do the iteration expression and then start over with the test, and don't worry about what was below the `continue` statement."

Labeled Statements

You need to understand the difference between labeled and unlabeled `break` and `continue`. The labeled varieties are needed only in situations where you have a nested loop, and need to indicate *which* of the nested loops you want to *break* from, or from which of the nested loops you want to `continue` with the next iteration. A `break` statement will exit out of the *labeled* loop, as opposed to the *innermost* loop, if the `break` keyword is combined with a label. An example of what a label looks like is in the following code:

```
foo:
   for (int x = 3; x < 20; x++) {
      while(y > 7) {
         y--;
      }
  }
```

The label must adhere to the rules for a valid variable name and should adhere to the Java naming convention. The syntax for the use of a label name in conjunction with a `break` statement is the `break` keyword, then the label name, followed by a semicolon. A more complete example of the use of a labeled `break` statement is as follows:

```
outer:
   for(int i=0; i<10; i++) {
      while (y > 7) {
         System.out.println("Hello");
         break outer;
      } // end of inner for loop
      System.out.println("Outer loop."); // Won't print
```

```
   } // end of outer for loop
System.out.println("Good-Bye");
```

Running this code produces

```
Hello
Good-Bye
```

In this example the word *Hello* will be printed one time. Then, the labeled `break` statement will be executed, and the flow will exit out of the loop labeled *outer*. The next line of code will then print out *Good-Bye*. Let's see what will happen if the `continue` statement is used instead of the `break` statement. The following code example is the same as the preceding one, with the exception of substituting `continue` for `break`:

```
outer:
   for (int i=0; i<10; i++) {
      for (int j=0; j<5; j++) {
         System.out.println("Hello");
         continue outer;
      } // end of inner loop
     System.out.println("outer"); // Never prints
   }
System.out.println("Good-Bye");
```

Running this code produces

```
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Good-Bye
```

In this example, *Hello* will be printed ten times. After the `continue` statement is executed, the flow continues with the next iteration of the loop identified with the label. Finally, when the condition in the outer loop evaluates to `false`, the *i* loop will finish and *Good-Bye* will be printed.

EXERCISE 4-2

Creating a Labeled while Loop

Try creating a labeled *while* loop. Make the label *outer* and provide a condition to check whether a variable age is less than or equal to 21. Within the loop, it should increment the age by one. Every time it goes through the loop, it checks whether the age is 16. If it is, it will print a message to get your driver's license and continue to the outer loop. If not, it just prints "Another year."

1. The outer label should appear just before the *while* loop begins. It does not matter if it is on the same line or not.
2. Make sure *age* is declared outside of the *while* loop.

exam Watch

Labeled `continue` and `break` statements must be inside the loop that has the same label name; otherwise, the code will not compile.

CERTIFICATION OBJECTIVE

Handling Exceptions (Exam Objectives 2.3 and 2.4)

Write code that makes proper use of exceptions and exception handling clauses (try, catch, finally) and declares methods and overriding methods that throw exceptions.

Recognize the effect of an exception arising at a specified point in a code fragment. Note that the exception may be a runtime exception, a checked exception, or an error (the code may include try, catch, or finally clauses in any legitimate combination).

An old maxim in software development says that 80 percent of the work is used 20 percent of the time. The 80 percent refers to the effort required to check and handle errors. In many languages, writing program code that checks for and deals with errors is tedious and bloats the application source into confusing spaghetti. Still, error detection and handling may be the most important ingredient of any robust application. Java arms developers with an elegant mechanism for handling errors that produces efficient and organized error-handling code: *exception handling*.

Exception handling allows developers to detect errors easily without writing special code to test return values. Even better, it lets us keep exception-handling code cleanly separated from the exception-generating code. It also lets us use the same exception-handling code to deal with a range of possible exceptions.

The exam has two objectives covering exception handling, but because they're covering the same topic we're covering both objectives with the content in this section.

Catching an Exception Using try and catch

Before we begin, let's introduce some terminology. The term *exception* means "exceptional condition" and is an occurrence that alters the normal program flow. A bunch of things can lead to exceptions, including hardware failures, resource exhaustion, and good old bugs. When an exceptional event occurs in Java, an exception is said to be *thrown*. The code that's responsible for doing something about the exception is called an *exception handler,* and it *catches* the thrown exception.

Exception handling works by transferring the execution of a program to an appropriate exception handler when an exception occurs. For example, if you call a method that opens a file but the file cannot be opened, execution of that method will stop, and code that you wrote to deal with this situation will be run. Therefore, we need a way to tell the JVM what code to execute when a certain exception happens. To do this, we use the `try` and `catch` keywords. The `try` is used to define a block of code in which exceptions may occur. This block of code is called a *guarded region* (which really means "risky code goes here"). One or more `catch` clauses match a specific exception (or class of exceptions—more on that later) to a block of code that handles it. Here's how it looks in pseudocode:

```
1. try {
2.    // This is the first line of the "guarded region"
3.    // that is governed by the try keyword.
4.    // Put code here that might cause some kind of exception.
5.    // We may have many code lines here or just one.
6. }
7. catch(MyFirstException) {
8.    // Put code here that handles this Exception.
9.    // This is the next line of the exception handler.
10.   // This is the last line of the exception handler.
11. }
12. catch(MySecondException) {
13.   // Put code here that handles this exception
14. }
```

```
15.
16. // Some other unguarded (normal, non-risky) code begins here
```

In this pseudocode example, lines 2 through 5 constitute the guarded region that is governed by the `try` clause. Line seven is an exception handler for an exception of type MyFirstException. Line 12 is an exception handler for an exception of type MySecondException. Notice that the `catch` blocks immediately follow the `try` block. This is a requirement; *if you have one or more `catch` blocks, they must immediately follow the `try` block*. Additionally, the `catch` blocks must all follow each other, *without any other statements or blocks in between*. Also, the order in which the catch blocks appear matters, as we'll see a little later.

Execution starts at line 2. If the program executes all the way to line 5 with no exceptions being thrown, execution will transfer to line 15 and continue downward. However, if at any time in lines 2 through 5 (the `try` block) an exception is thrown of type MyFirstException, execution will immediately transfer to line 8. Lines 8 through 10 will then be executed so that the entire `catch` block runs, and then execution will transfer to line 15 and continue.

Note that if an exception occurred on, say, line 3 of the `try` block, the rest of the lines in the `try` block (3 through 5) would never be executed. Once control jumps to the `catch` block, it never returns to complete the balance of the `try` block. This is exactly what you want, though. Imagine your code looks something like this pseudocode:

```
try {
   getTheFileFromOverNetwork
   readFromTheFileAndPopulateTable
}
catch(CantGetFileFromNetwork) {
   useLocalFileInstead
}
```

The preceding pseudocode demonstrates how you typically work with exceptions. Code that's dependent on a risky operation (as populating a table with file data is dependent on getting the file from the network) is grouped into a `try` block in such a way that if, say, the first operation fails, you won't continue trying to run other code that's guaranteed to also fail. In the pseudocode example, you won't be able to read from the file if you can't get the file off the network in the first place.

One of the benefits of using exception handling is that code to handle any particular exception that may occur in the governed region needs to be written only

once. Returning to our earlier code example, there may be three different places in our `try` block that can generate a MyFirstException, but wherever it occurs it will be handled by the same `catch` block (on line 7). We'll discuss more benefits of exception handling near the end of this chapter.

Using finally

Try and *catch* provide a terrific mechanism for trapping and handling exceptions, but we are left with the problem of how to clean up after ourselves. Because execution transfers out of the `try` block as soon as an exception is thrown, we can't put our cleanup code at the bottom of the `try` block and expect it to be executed if an exception occurs. Almost as bad an idea would be placing our cleanup code in the `catch` blocks.

Exception handlers are a poor place to clean up after the code in the `try` block because each handler then requires its own copy of the cleanup code. If, for example, you allocated a network socket or opened a file somewhere in the guarded region, each exception handler would have to close the file or release the socket. That would make it too easy to forget to do cleanup, and also lead to a lot of redundant code. To address this problem, Java offers the `finally` block.

A `finally` block encloses code that is *always* executed at some point after the `try` block, *whether an exception was thrown or not.* Even if there is a *return* statement in the `try` block, the finally block executes right after the *return* statement! This is the right place to close your files, release your network sockets, and perform any other cleanup your code requires. If the `try` block executes with no exceptions, the `finally` block is executed immediately after the `try` block completes. If there was an exception thrown, the `finally` block executes immediately after the proper `catch` block completes.

Let's look at another pseudocode example:

```
 1: try {
 2:    // This is the first line of the "guarded region".
 3: }
 4: catch(MyFirstException) {
 5:    // Put code here that handles this error.
 6: }
 7: catch(MySecondException) {
 8:    // Put code here that handles this error.
 9: }
10: finally {
11:    // Put code here to release any resource we
12:    // allocated in the try clause.
```

```
13: }
14:
15: // More code here
```

As before, execution starts at the first line of the `try` block, line 2. If there are no exceptions thrown in the `try` block, execution transfers to line 11, the first line of the `finally` block. On the other hand, if a MySecondException is thrown while the code in the `try` block is executing, execution transfers to the first line of that exception handler, line 8 in the `catch` clause. After all the code in the `catch` clause is executed, the program moves to line 11, the first line of the `finally` clause. Repeat after me: finally *always runs*! OK, we'll have to refine that a little, but for now, start burning in the idea that *finally always* runs. If an exception is *thrown*, *finally* runs. If an exception is *not* thrown, *finally* runs. If the exception is *caught*, *finally* runs. If the exception is *not* caught, *finally* runs. Later we'll look at the few scenarios in which *finally* might not run or complete.

`finally` clauses are not required. If you don't write one, your code will compile and run just fine. In fact, if you have no resources to clean up after your `try` block completes, you probably don't need a `finally` clause. Also, because the compiler doesn't even require `catch` clauses, sometimes you'll run across code that has a `try` block immediately followed by a `finally` block. Such code is useful when the exception is going to be passed back to the calling method, as explained in the next section. Using a `finally` block allows the cleanup code to execute even when there isn't a `catch` clause.

The following legal code demonstrates a *try* with a *finally* but no *catch*:

```
try {
  // do stuff
} finally {
    //clean up
}
```

The following legal code demonstrates a *try*, *catch*, and *finally*:

```
try {
   // do stuff
} catch (SomeException ex) {
  // do exception handling
} finally {
   // clean up
}
```

The following *illegal* code demonstrates a *try* without *catch* or *finally*:

```
try {
  // do stuff
}
// need a catch or finally here
System.out.println("out of try block");
```

The following *illegal* code demonstrates a misplaced `catch` block:

```
try {
   // do stuff
}
// can't have code between try/catch
System.out.println("out of try block");
catch(Exception ex) { }
```

exam Watch

It is illegal to use a `try` clause without either a `catch` clause or a `finally` clause. A `try` clause by itself will result in a compiler error. Any `catch` clauses must immediately follow the `try` block. Any `finally` clauses must immediately follow the last `catch` clause. It is legal to omit either the `catch` clause or the `finally` clause, but not both.

exam Watch

You can't sneak any code in between the `try` and `catch` (or `try` and `finally`) blocks. The following won't compile:

```
try {
   // do stuff
}
System.out.print("below the try");  //Illegal!
catch(Exception ex) { }
```

Propagating Uncaught Exceptions

Why aren't `catch` clauses required? What happens to an exception that's thrown in a `try` block when there is no `catch` clause waiting for it? Actually, there's no requirement that you code a `catch` clause for every possible exception that could be thrown from the corresponding `try` block. In fact, it's doubtful that you could accomplish such a feat! If a method doesn't provide a `catch` clause for a particular exception, that method is said to be "ducking" the exception (or "passing the buck").

So what happens to a ducked exception? Before we discuss that, we need to briefly review the concept of the *call stack*. Most languages have the concept of a method stack or a call stack. Simply put, the call stack is the chain of methods that your program executes to get to the current method. If your program starts in method `main()` and `main()` calls method `a()`, which calls method `b()` that in turn calls method `c()`, the call stack consists of the following:

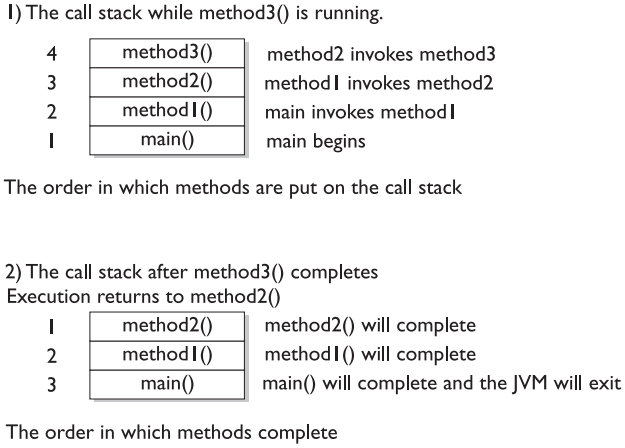
```
c
b
a
main
```

A stack can be represented as growing upward (although it can also be visualized as growing downward). As you can see, the last method called is at the top of the stack, while the first calling method is at the bottom. If you could print out the state of the stack at any given time, you would produce a *stack trace*. The method at the very top of the stack trace would be the method you were currently executing. If we move back down the call stack, we're moving from the current method to the previously called method. Figure 4-1 illustrates a way to think about how the call stack in Java works.

Now let's examine what happens to ducked exceptions. Imagine a building, say, five stories high, and at each floor there is a deck or balcony. Now imagine that on each deck, one person is standing holding a baseball mitt. Exceptions are like balls dropped from person to person, starting from the roof. An exception is first thrown

FIGURE 4-1

The Java method call stack



from the top of the stack (in other words, the person on the roof), and if it isn't caught by the same person who threw it (the person on the roof), it drops down the call stack to the previous method, which is the person standing on the deck one floor down. If not caught there, by the person one floor down, the exception/ball again drops down to the previous method (person on the next floor down), and so on until they are caught or until they reach the very bottom of the call stack. This is called *exception propagation*.

If they reach the bottom of the call stack, it's like reaching the bottom of a very long drop; the ball explodes, and so does your program. An exception that's never caught will cause your application to stop running. A description (if one is available) of the exception will be displayed, and the call stack will be "dumped." This helps you debug your application by telling you what exception was thrown, from what method it was thrown, and what the stack looked like at the time.

You can keep throwing an exception down through the methods on the stack. But what about when you get to the `main()` method at the bottom? You can throw the exception out of `main()` as well. This results in the Java virtual machine (JVM) halting, and the stack trace will be printed to the output. The following code throws an exception,

```
class TestEx {
   public static void main (String [] args) {
       doStuff();
   }
   static void doStuff() {
      doMoreStuff();
   }
   static void doMoreStuff() {
    int x = 5/0;  // Can't divide by zero! ArithmeticException is thrown here
   }
}
```

which prints out the stack trace,

```
 %java TestEx
Exception in thread "main" java.lang.ArithmeticException: / by zero
at TestEx.doMoreStuff(TestEx.java:10)
at TestEx.doStuff(TestEx.java:7)
at TestEx.main(TestEx.java:3)
```

EXERCISE 4-3

Propagating and Catching an Exception

So far you have only seen exceptions displayed in this chapter with pseudocode. In this exercise we attempt to create two methods that deal with exceptions. One of the methods is the `main()` method, which will call another method. If an exception is thrown in the other method, it must deal with it. A `finally` statement will be included to indicate it is all done. The method it will call will be named `reverse()`, and it will reverse the order of the characters in the string. If the string contains no characters, it will propagate an exception up to the `main()` method.

1. Create an enclosing class called Propagate and a `main()` method, which will remain empty for now.
2. Create a method called `reverse()`. It takes an argument of a string and returns a String.
3. Check if the String has a length of 0 by using the `length()` method. If the length is 0, it will throw a new exception.
4. Now let's include the code to reverse the order of the String. Because this isn't the main topic of this chapter, the reversal code has been provided, but feel free to try it on your own.

```
String reverseStr = "";
for(int i=s.length()-1;i>=0;--i) {
   reverseStr += s.charAt(i);
}
return reverseStr;
```

5. Now in the `main()` method we will attempt to call this method and deal with any potential exceptions. Additionally, we will include a `finally` statement that tells us it has finished.

Defining Exceptions

We have been discussing exceptions as a concept. We know that they are thrown when a problem of some type happens, and we know what effect they have on the

flow of our program. In this section we will develop the concepts further and use exceptions in functional Java code. Earlier we said that an exception is an occurrence that alters the normal program flow. But because this is Java, anything that's not a primitive must be…an object. Exceptions are no, well, *exception* to this rule. Every exception is as an instance of a class that has class Exception in its inheritance hierarchy. In other words, exceptions are always some subclass of `java.lang.Exception`.

When an exception is thrown, an object of a particular Exception subtype is instantiated and handed to the exception handler as an argument to the `catch` clause. An actual `catch` clause looks like this:

```
try {
   // some code here
}
catch (ArrayIndexOutOfBoundsException e) {
   e.printStackTrace();
}
```

In this example, *e* is an instance of a class with the tersely named ArrayIndexOutOfBoundsException. As with any other object, you can call its methods.

Exception Hierarchy

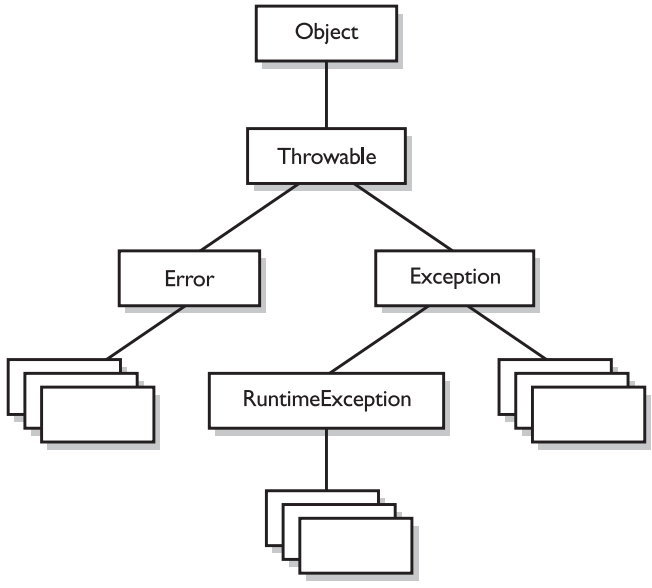
All exception classes are subtypes of class Exception. This class derives from the class Throwable (which derives from the class Object). Figure 4-2 shows the hierarchy for the exception classes.

As you can see, there are two subclasses that derive from Throwable: Exception and Error. Classes that derive from Error represent unusual situations that are not caused by program errors or by anything that would normally happen during program execution, such as the JVM running out of memory. Generally, your application won't be able to recover from an Error, so you're not required to handle them. If your code does *not* handle them (and it usually won't), it will still compile with no trouble. Although often thought of as exceptional conditions, Errors are technically not exceptions because they do not derive from class Exception.

In general, an exception represents something that happens not as a result of a programming error, but rather because some resource is not available or some other condition required for correct execution is not present. For example, if your application is supposed to communicate with another application or computer that is not

FIGURE 4-2

Exception class hierarchy



answering, this is an exception that is not caused by a bug. Figure 4-2 also shows a subtype of Exception called RuntimeException. These exceptions are a special case because they actually do indicate program errors. They can also represent rare, difficult to handle exceptional conditions. Runtime exceptions are discussed in greater detail later in this chapter.

Java provides many exception classes, most of which have quite descriptive names. There are two ways to get information about an exception. The first is from the type of the exception itself. The next is from information that you can get from the exception object. Class Throwable (at the top of the inheritance tree for exceptions) provides its descendants with some methods that are useful in exception handlers. One of these is `printStackTrace()`. As expected, if you call an exception object's `printStackTrace()` method, as in the earlier example, a stack trace from where the exception occurred will be printed.

We discussed that a call stack builds upward with the most recently called method at the top. You will notice that the `printStackTrace()` method prints the most recently entered method first and continues down, printing the name of each method as it works its way down the call stack (this is called *unwinding the stack*) from the top.

exam Watch

For the exam, it is not necessary to know any of the** methods **contained in the Throwable classes, including Exception and Error. You are expected to know that Exception, Error, RuntimeException, and Throwable types can all be thrown using the `throws` keyword, and can all be caught (although you rarely will catch anything other than Exception subtypes).

Handling an Entire Class Hierarchy of Exceptions

We've discussed that the `catch` keyword allows you to specify a particular type of exception to catch. You can actually catch *more* than one type of exception in a single `catch` clause. If the exception class that you specify in the `catch` clause has no subclasses, then only the specified class of exception will be caught. However, if the class specified in the `catch` clause does have subclasses, *any exception object that subclasses the specified class will be caught* as well.

For example, class IndexOutOfBoundsException has two subclasses, ArrayIndexOutOfBoundsException and StringIndexOutOfBoundsException. You may want to write one exception handler that deals with exceptions produced by either type of boundary error, but you might not be concerned with which exception you actually have. In this case, you could write a `catch` clause like the following:

```
try {
   // Some code here that can throw a boundary exception
}
catch (IndexOutOfBoundsException e) {
   e.printStackTrace();
}
```

If any code in the `try` block throws ArrayIndexOutOfBoundsException or StringIndexOutOfBoundsException, the exception will be caught and handled. This can be convenient, but it should be used sparingly. By specifying an exception class' superclass in your `catch` clause, you're discarding valuable information about the exception. You can, of course, find out exactly what exception class you have, but if you're going to do that, you're better off writing a separate `catch` clause for each exception type of interest.

on the Job

Resist the temptation to write a single catchall exception handler such as the following:

```
try {
   // some code
```

```
}
catch (Exception e) {
   e.printStackTrace();
}
```

This code will catch every exception generated. Of course, no single exception handler can properly handle every exception, and programming in this way defeats the design objective. Exception handlers that trap many errors at once will probably reduce the reliability of your program because it's likely that an exception will be caught that the handler does not know how to handle.

Exception Matching

If you have an exception hierarchy composed of a superclass exception and a number of subtypes, and you're interested in handling one of the subtypes in a special way but want to handle all the rest together, you need write only two `catch` clauses.

When an exception is thrown, Java will try to find a `catch` clause for the exception type. If it doesn't find one, it will search for a handler for a supertype of the exception. If it does *not* find a `catch` clause that matches a supertype for the exception, then the exception is propagated down the call stack. This process is called *exception matching*.

Let's look at an example:

```
 1: import java.io.*;
 2: public class ReadData {
 3:    public static void main(String args[]) {
 4:       try {
 5:          RandomAccessFile raf =
 6:             new RandomAccessFile("myfile.txt", "r");
 7:          byte b[] = new byte[1000];
 8:          raf.readFully(b, 0, 1000);
 9:       }
10:       catch(FileNotFoundException e) {
11:          System.err.println("File not found");
12:          System.err.println(e.getMessage());
13:          e.printStackTrace();
14:       }
15:       catch(IOException e) {
16:          System.err.println("IO Error");
17:          System.err.println(e.toString());
18:          e.printStackTrace();
19:       }
```

```
20:     }
21: }
```

This short program attempts to open a file and to read some data from it. Opening and reading files can generate many exceptions, most of which are some type of IOException. Imagine that in this program we're interested in knowing only whether the exact exception is a FileNotFoundException. Otherwise, we don't care exactly what the problem is.

FileNotFoundException is a subclass of IOException. Therefore, we could handle it in the `catch` clause that catches all subtypes of IOException, but then we would have to test the exception to determine whether it was a FileNotFoundException. Instead, we coded a special exception handler for the FileNotFoundException and a separate exception handler for all other IOException subtypes.

If this code generates a FileNotFoundException, it will be handled by the `catch` clause that begins at line 10. If it generates another IOException—perhaps EOFException, which is a subclass of IOException—it will be handled by the `catch` clause that begins at line 15. If some other exception is generated, such as a runtime exception of some type, neither `catch` clause will be executed and the exception will be propagated down the call stack.

Notice that the `catch` clause for the FileNotFoundException was placed above the handler for the IOException. *This is really important!* If we do it the opposite way, the program will not compile. *The handlers for the most specific exceptions must always be placed above those for more general exceptions.* The following will not compile:

```
try {
   // do risky IO things
} catch (IOException e) {
  // handle general IOExceptions
} catch (FileNotFoundException ex) {
  // handle just FileNotFoundException
}
```

You'll get the following compiler error:

```
TestEx.java:15: exception java.io.FileNotFoundException has
 already been caught
} catch (FileNotFoundException ex) {
         ^
```

If you think of the people with baseball mitts, imagine that the most general mitts are the largest, and can thus catch many different kinds of balls. An IOException

mitt is large enough and flexible enough to catch any type of IOException. So if the person on the fifth floor (say, Fred) has a big 'ol IOException mitt, he can't help but catch a FileNotFoundException ball with it. And if the guy (say, Jimmy) on the second floor is holding a FileNotFoundException mitt, that FileNotFoundException ball will never get to him, since it will always be stopped by Fred on the fifth floor, standing there with his big-enough-for-any-IOException mitt.

So what do you do with exceptions that are siblings in the class hierarchy? If one Exception class is not a subtype or supertype of the other, then the order in which the `catch` clauses are placed doesn't matter.

Exception Declaration and the Public Interface

So, how do we know that some method throws an exception that we have to catch? Just as a method must specify what type and how many arguments it accepts and what is returned, the exceptions that a method can throw *must* be declared (unless the exceptions are subclasses of `RuntimeException`). The list of thrown exceptions is part of a method's public interface. The `throws` keyword is used as follows to list the exceptions that a method can throw:

```
void myFunction() throws MyException1, MyException2 {
   // code for the method here
}
```

This method has a `void` return type, accepts no arguments, and declares that it *`throws`* two exceptions of type MyException1 and MyException2. (Just because the method *declares* that it `throws` an exception doesn't mean it always *will*. It just tells the world that it *might*.)

Suppose your method doesn't directly throw an exception, but calls a method that *does*. You can choose not to handle the exception yourself and instead just declare it, as though it were *your* method that actually `throws` the exception. If you do declare the exception that your method might get from another method, and you don't provide a `try/catch` for it, then the method will propagate back to the method that called *your* method, and either be caught there or continue on to be handled by a method further down the stack.

Any method that might *throw* an exception (unless it's a subclass of RuntimeException) must *declare* the exception. That includes methods that aren't actually throwing it directly, but are "ducking" and letting the exception pass down to the next method in the stack. If you "duck" an exception, it is just as if you were the one actually

throwing the exception. RuntimeException subclasses are exempt, so the compiler won't check to see if you've declared them. But all non-RuntimeExceptions are considered "checked" exceptions, because the compiler checks to be certain you've acknowledged that "bad things could happen here."

Remember this: Each method must either handle *all* checked exceptions by supplying a `catch` clause *or* list each unhandled checked exception as a thrown exception. This rule is referred to as Java's *handle or declare requirement*. (Sometimes called *catch* or *declare*.)

exam Watch

Look for code that invokes a method declaring an exception, where the calling method doesn't handle or declare the checked exception. The following code has two big problems that the compiler will prevent:

```
void doStuff() {
   doMore();
}
void doMore() {
    throw new IOException();
}
```

First, the `doMore()` method throws a checked exception, but does not declare it! But suppose we fix the `doMore()` method as follows:

```
void doMore() throws IOException { … }
```

The `doStuff()` method is still in trouble because it, too, must declare the IOException, unless it handles it by providing a `try/catch`, with a `catch` clause that can take an IOException.

Again, some exceptions are exempt from this rule. An object of type RuntimeException may be thrown from any method without being specified as part of the method's public interface (and a handler need not be present). And even if a method does declare a RuntimeException, the calling method is under no obligation to handle or declare it. RuntimeException, Error, and all of their subtypes are *unchecked* exceptions and *unchecked exceptions do not have to be specified or handled.*

Here is an example:

```
import java.io.*;
class Test {
   public int myMethod1() throws EOFException {
      return myMethod2();
   }
   public int myMethod2() throws EOFException {
```

```
        // Some code that actually throws the exception goes here
        return 1;
    }
}
```

Let's look at `myMethod1()`. Because EOFException subclasses IOException and IOException subclasses Exception, it is a checked exception and must be declared as an exception that may be thrown by this method. But where will the exception actually come from? The public interface for method `myMethod2()` called here declares that an exception of this type can be thrown. Whether that method actually throws the exception itself or calls another method that throws it is unimportant to us; we simply know that we have to either catch the exception or declare that we throw it. The method `myMethod1()` does not catch the exception, so it declares that it throws it.

Now let's look at another legal example, `myMethod3()`.

```
public void myMethod3() {
    // Some code that throws a NullPointerException goes here
}
```

According to the comment, this method can throw a NullPointerException. Because RuntimeException is the immediate superclass of NullPointerException, it is an *unchecked* exception and need not be declared. We can see that `myMethod3()` does not declare any exceptions.

Runtime exceptions are referred to as *unchecked exceptions*. All other exceptions, meaning all those that do not derive from java.lang.RuntimeException, are *checked exceptions*. A checked exception must be caught somewhere in your code. If you invoke a method that throws a checked exception but you don't catch the checked exception somewhere, your code will not compile. That's why they're called checked exceptions; the compiler checks to make sure that they're handled or declared. A number of the methods in the Java 2 Standard Edition libraries throw checked exceptions, so you will often write exception handlers to cope with exceptions generated by methods you didn't write.

You can also throw an exception yourself, and that exception can be either an existing exception from the Java API or one of your own. To create your own exception, you simply subclass Exception (or one of its subclasses) as follows:

```
class MyException extends Exception { }
```

And if you throw the exception, the compiler will guarantee that you declare it as follows:

```
class TestEx {
   void doStuff() {
     throw new MyException();  // Throw a checked exception
   }
}
```

The preceding code upsets the compiler:

```
TestEx.java:6: unreported exception MyException; must be caught or
declared to be thrown
  throw new MyException();
  ^
```

exam Watch

When an object of a subtype of Exception is thrown, it** must **be handled or declared. These objects are called** checked **exceptions, and include all exceptions except those that are subtypes of RuntimeException, which are** unchecked **exceptions. Be ready to spot methods that don't follow the handle or declare rule, such as

```
class MyException extends Exception {
void someMethod () {
    doStuff();
}
void doStuff() throws MyException {
   try {
      throw new MyException();
    }
   catch(MyException me) {
      throw me;
    }
}
```

You need to recognize that this code won't compile. If you try, you'll get

```
TestEx.java:8: unreported exception MyException; must be caught or
declared to be thrown
doStuff();
^
```

exam Watch

The exam objectives specifically state that you need to know how an Error compares with checked and unchecked exceptions. Objects of type Error are *not* ***Exception objects, although they do represent exceptional conditions. Both Exception and Error share a common superclass, Throwable, thus both can be thrown using the*** `throws` ***keyword. When an Error or a subclass of Error is thrown, it's unchecked.*** *You are not required to catch Error objects or Error subtypes.* ***You can also throw an Error yourself (although you probably won't ever want to) and you*** *can* ***catch one, but again, you probably*** *won't.* ***What, for example, would you actually do if you got an OutOfMemoryError? It's not like you can tell the garbage collector to run; you can bet the JVM fought desperately to save itself (and reclaimed all the memory it could) by the time you got the error. In other words, don't expect the JVM at that point to say, "Run the garbage collector? Oh, thanks*** *so* ***much for telling me. That just never occurred to me. Sure, I'll get right on it…" Even better, what would you do if a VirtualMachineError arose? Your program is toast by the time you'd catch the Error, so there's really no point in trying to catch one of these babies. Just remember, though, that you*** *can****! The following compiles just fine:***

```
class TestEx {
   public static void main (String [] args) {
      badMethod();
   }
   static void badMethod() {  // No need to declare an Error
      doStuff()
   }
   static void doStuff() {  //No need to declare an Error
      try {
        throw new Error();
      }
      catch(Error me) {
         throw me; // We catch it, but then rethrow it
      }
    }
}
```

If we were throwing a checked exception rather than Error, then the `doStuff()` ***method would need to declare the exception. But remember, since Error is not a subtype of Exception, it doesn't need to be declared. You're free to declare it if you like, but the compiler just doesn't care one way or another when or how the Error is thrown, or by whom.***

Because Java has checked exceptions, it's commonly said that Java forces developers to handle errors. Yes, Java forces us to write exception handlers for each exception that can occur during normal operation, but it's up to us to make the exception handlers actually do something useful. We know software managers who melt down when they see a programmer write

```
try {
    callBadMethod();
} catch (Exception ex) { }
```

Notice anything missing? Don't "eat" the exception by catching it without actually handling it. You won't even be able to tell that the exception occurred, because you'll never see the stack trace.

Rethrowing the Same Exception

Just as you can throw a new exception from a `catch` clause, you can also throw the same exception you just caught. Here's a `catch` clause that does this:

```
catch(IOException e) {
   // Do things, then if you decide you can't handle it…
       throw e;
}
```

All other `catch` clauses associated with the same `try` are ignored, and the exception is thrown back to the calling method (the next method down the call stack). If you *throw* a checked exception from a `catch` clause, you must also *declare* that exception! In other words, you must handle *and* declare, as opposed to handle *or* declare. The following example is illegal:

```
public void doStuff() {
   try {
      // risky IO things
   } catch(IOException ex) {
      // can't handle it
       throw ex;  // Can't throw it unless you declare it
   }
}
```

In the preceding code, the `doStuff()` method is clearly able to throw a checked exception—in this case an IOException—so the compiler says, *"Well, that's just*

peachy that you have a `try/catch` *in there, but it's not good enough. If you* might *rethrow the IOException you catch, then you* must *declare it!"*

EXERCISE 4-4

Creating an Exception

In this exercise we attempt to create a custom exception. We won't put in any new methods (it will have only those inherited from Exception), and because it extends Exception, the compiler considers it a checked exception. The goal of the program is to check to see if a command-line argument, representing a particular food (as a string), is considered bad or OK.

1. Let's first create our exception. We will call it BadFoodException. This exception will be thrown when a bad food is encountered.
2. Create an enclosing class called MyException and a `main()` method, which will remain empty for now.
3. Create a method called `checkFood()`. It takes a String argument and throws our exception if it doesn't like the food it was given. Otherwise, it tells us it likes the food. You can add any foods you aren't particularly fond of to the list.
4. Now in the `main()` method, you'll get the command-line argument out of the String array, and then pass that String on to the `checkFood()` method. Because it's a checked exception, the `checkFood()` method must declare it, and `the main()` method must handle it (using a *try/catch*). Do *not* have `main()` declare the method, because if `main()` ducks the exception, who else is back there to catch it?

As useful as exception handling is, it's still up to the developer to make proper use of it. Exception handling makes organizing our code and signaling problems easy, but the exception handlers still have to be written. You'll find that even the most complex situations can be handled, and keep your code reusable, readable, and maintainable.

CERTIFICATION OBJECTIVE

Working with the Assertion Mechanism (Exam Objectives 2.4 and 2.5)

Write code that makes proper use of assertions, and distinguish appropriate from inappropriate uses of assertions.

Identify correct statements about the assertion mechanism.

You know you're not supposed to make assumptions, but you can't help it when you're writing code. You put them in comments:

```
if (x > 2 && y) {
   // do something
} else if (x < 2 || y) {
   // do something
} else {
   // x must be 2
   // do something else
}
```

You write *print* statements with them:

```
while (true) {
   if (x > 2) {
       break;
   }
   System.out.print("If we got here something went horribly
wrong");
}
```

Added to the Java language beginning with version 1.4, assertions let you test your assumptions during development, without the expense (in both your time and program overhead) of writing exception handlers for exceptions that *you assume will never happen* once the program is out of development and fully deployed.

Starting with exam 310-035 (version 1.4 of the Sun Certified Java Programmer exam), you're expected to know the basics of how assertions (in Java) work, including how to enable them, how to use them, and how *not* to use them. Because both objectives test the same concepts, the things you need to know for both are covered together in this section.

Assertions Overview

Suppose you assume that a number passed into a method (say, `methodA()`) will never be negative. While testing and debugging, you want to validate your assumption, but you don't want to have to strip out *print* statements, runtime exception handlers, or *if/else* tests when you're done with development. But leaving any of those in is, at the least, a performance hit. Assertions to the rescue! Check out the following preassertions code:

```
private void methodA(int num) {
   if (num >= 0) {
      // do stuff
   } else {  // num must be < 0
     // This code will never be reached!
     System.out.println("Yikes! num is a negative number! " + num);
   }
   useNum(num + x);
}
```

Because you're so certain of your assumption, you don't want to take the time (or program performance hit) to write exception-handling code. And at runtime, you don't want the *if/else* in there either because if you *do* reach the `else` condition, it means your earlier logic (whatever was running prior to this method being called) is flawed. Assertions let you test your assumptions during development, but the assertion code—in effect—*evaporates* when the program is deployed, leaving behind no overhead or debugging code to track down and remove. Let's rewrite `methodA()` to validate that the argument was not negative:

```
private void methodA(int num) {
   assert (num>=0);  // throws an AssertionError
                     // if this test isn't true
   useNum(num + x);
}
```

Not only do assertions let your code stay cleaner and smaller, but because assertions are inactive unless specifically "turned on" (enabled), the code will run as though it were written like this:

```
private void methodA(int num) {
   useNum(num + x); // we've tested this;
                    // we now know we're good here
}
```

Assertions work quite simply. *You always assert that something is* true. If it *is*, no problem. Code keeps running. But if your assertion turns out to be wrong (*false*), then a stop-the-world AssertionError is thrown (that you should never, ever handle!) right then and there, so you can fix whatever logic flaw led to the problem.

Assertions come in two flavors: simple and *really* simple, as follows:

Really Simple

```
private void doStuff() {
   assert (y > x);
   // more code assuming y is greater than x
}
```

Simple

```
private void doStuff() {
   assert (y > x): "y is " + y " " x is " + x;
   // more code assuming y is greater than x
}
```

The difference between them is that the simple version adds a second expression, separated from the first (boolean expression) by a colon, that adds a little more information to the stack trace. Both versions throw an immediate AssertionError, but the simple version gives you a little more debugging help while the really simple version simply tells you that your assumption was *false*.

on the job

Assertions are typically enabled when an application is being tested and debugged, but** disabled **when the application is deployed. The assertions are still in the code, although ignored by the JVM, so if you do have a deployed application that starts misbehaving, you can always choose to enable assertions in the field for additional testing.

Assertion Expression Rules

Assertions can have either one or two expressions, depending on whether you're using the *simple* or *really simple* flavor. The first expression must always result in a `boolean` value! Follow the same rules you use for *if* and *while* tests. The whole point is to assert *aTest*, which means you're asserting that *aTest* is *true*. If it *is true*, no problem. If it's *not true*, however, then your assumption was wrong and you get an AssertionError.

The second expression, used only with the simple version of an `assert` statement, can be *anything that results in a value*. Remember, the second expression is used to generate a String message that displays in the stack trace to give you a little more debugging information. It works much like `System.out.println()` in that you can pass it a primitive or an object, and it will convert it into a String representation. It *must* resolve to a value!

Table 4-3 lists legal and illegal expressions for both parts of an `assert` statement. Remember, expression2 is used only with the simple `assert` statement, where the second expression exists solely to give you a little more debugging detail.

exam Watch

If you see the word "expression" in a question about assertions, and the question doesn't specify whether it means *expression1* ***(the boolean test) or*** *expression2* ***(the value to print in the stack trace), then always assume the word*** *expression* ***refers to*** *expression1*, ***the boolean test. For example, if we asked you the following question,***

"An assert expression must result in a boolean value, true or false?",

assume that the word *expression* ***refers to*** *expression1* ***of an*** `assert`, ***so the question statement is correct. If the statement were referring to expression2, however, the statement would not be correct, since expression2 can have a result of*** *any* ***value, not just a boolean.***

Enabling Assertions

If you want to use assertions, you have to think first about how to *compile* with assertions in your code, and then about how to *run* with assertions turned on. Both require version 1.4 or greater, and that brings us to the first issue: how to compile with assertions in your code.

TABLE 4-3 Legal and Illegal `assert` Expressions

Expression1		Expression2	
Legal	**Illegal**	**Legal**	**Illegal**
`assert (x ==2)`	`assert (x = 2)`	`: "x is " + x`	`: void`
`boolean z = true;` `assert (z)`	`int z = 0;` `assert (z)`	`public int go() { return 1; }` `: go();`	`public void go() { }` `: go();`
`assert false`	`assert 1`	`: new Foo();`	`: Foo f;`

Compiling with Assertions

Prior to version 1.4, you might very well have written code like this:

```
int assert = getInitialValue();
if (assert == getActualResult()) {
   // do something
}
```

Notice that in the preceding code, `assert` is used as an identifier. No problem prior to 1.4. But remember that you cannot use a keyword/reserved word as an identifier, and beginning with version 1.4, *`assert` is now a keyword!* The bottom line is

You can use "assert" as a keyword or as an identifier, but not both.

You get a choice whenever you compile with version 1.4, as to whether you're compiling *"assertion aware"* code or code written in the old way, where assert is not a reserved word. Let's look at both. You must know this: in version 1.4, *assertions are disabled by default!* If you don't specifically "turn them on" at compile time, then assert will not be recognized as a keyword, because the compiler will act as a version 1.3 compiler, with respect to the word "assert" (in which case your code can happily use `assert` as an identifier).

Compiling Assertion-Aware Code If you're using `assert` as a keyword (in other words, you're actually trying to *assert* something in your code), then you must explicitly enable assertion-awareness at compile time, as follows:

```
javac -source 1.4 com/geeksanonymous/TestClass
```

You can read that as "compile the class TestClass, in the directory com/geeksanonymous, and do it in the 1.4 way, where `assert` is a recognized keyword."

Compiling with Code That Uses Assert as an Identifier If you don't use the `-source 1.4` flag, then the default behavior is as though you said to the compiler, "Compile this code as if you didn't known anything about `assert` as a keyword, so that I may use the word `assert` as an identifier for a method or variable." The following is what you get by default:

```
javac -source 1.3 com/geeksanonymous/TestClass
```

But since that's the default behavior, it's redundant to actually type `-source 1.3`.

Running with Assertions

Here's where it gets cool. Once you've written your assertion-aware code (in other words, code that uses `assert` as a keyword, to actually perform assertions at runtime), you can choose to enable or disable them! Remember, *assertions are disabled by default.*

Enabling Assertions at Runtime You enable assertions at runtime with

```
java -ea com.geeksanonymous.TestClass
```

or

```
java -enableassertions com.geeksanonymous.TestClass
```

The preceding command-line switches tell the JVM to run with assertions enabled.

Disabling Assertions at Runtime You must also know the command-line switches for disabling assertions,

```
java -da com.geeksanonymous.TestClass
```

or

```
java -disableassertions com.geeksanonymous.TestClass
```

Because assertions are disabled by default, using the disable switches might seem unnecessary. Indeed, using the switches the way we do in the preceding example just gives you the default behavior (in other words, you get the same result regardless of whether you use the disabling switches). But…you can also selectively enable and disable assertions in such a way that they're enabled for some classes and/or packages, and disabled for others, while a particular program is running.

Selective Enabling and Disabling The command-line switches to enable and disable assertions can be used in various ways:

- **With no arguments** (as in the preceding examples) Enables or disables assertions in all classes, except for the system classes.
- **With a package name** Enables or disables assertions in the package specified, and any packages below this package in the same directory hierarchy (more on that in a moment).
- **With a class name** Enables or disables assertions in the class specified.

You can combine switches to, say, disable assertions in a single class, but keep them enabled for all others, as follows:

```
java -ea  -da:com.geeksanonymous.Foo
```

The preceding command line tells the JVM to enable assertions in general, but disable them in the class `com.geeksanonymous.Foo`. You can do the same selectivity for a package as follows:

```
java -ea -da:com.geeksanonymous...
```

The preceding command line tells the JVM to enable assertions in general, but disable them in the package *com.geeksanonymous, and all of its subpackages!* You may not be familiar with the term *subpackages*, since there wasn't much use of that term prior to assertions. A *subpackage* is any package in a subdirectory of the named package. For example, look at the following directory tree:

```
com
   |_geeksanonymous
                  |_Foo
                  |_Bar
                  |_twelvesteps
                              |_StepOne
                              |_StepTwo
```

This tree lists three directories,

```
com
geeksanonymous
twelvesteps
```

and four classes:

```
com.geeksanonymous.Foo
com.geeksanonymous.Bar
com.geeksanonymous.twelvesteps.StepOne
com.geeksanonymous.twelvesteps.StepTwo
```

The subpackage of *com.geeksanonymous* is the *twelvesteps* package. Remember that in Java, the *com.geeksanonymous.twelvesteps* package is treated as a completely *distinct* package that has no relationship with the packages above it (in this example, the *com.geeksanonymous* package), except they just *happen* to share a couple of directories. Table 4-4 lists examples of command-line switches for enabling and disabling assertions.

TABLE 4-4 Assertion Command-Line Switches

Command-Line Example	What It Means
`java -ea` `java -enableassertions`	Enable assertions
`java -da` `java -disableassertions`	Disable assertions (the default behavior of version 1.4)
`java -ea:com.foo.Bar`	Enable assertions in class com.foo.Bar
`java -ea:com.foo...`	Enable assertions in package *com.foo*, and *any of its subpackages*
`java -ea -dsa`	Enable assertions in general, but disable assertions in system classes
`java -ea -da:com.foo...`	Enable assertions in general, but disable assertions in package *com.foo and any of its subpackages*

Using Assertions Appropriately

Not all *legal* uses of assertions are considered *appropriate*. As with so much of Java, you can abuse the intended use for assertions, despite the best efforts of Sun's Java engineers to discourage you. For example, you're *never* supposed to handle an assertion failure. That means don't catch it with a `catch` clause and attempt to recover. Legally, however, AssertionError is a subclass of Throwable, so it *can* be caught. But just don't do it! If you're going to try to recover from something, it should be an exception. To discourage you from trying to substitute an assertion for an exception, the AssertionError doesn't provide access to the object that generated it. All you get is the String message.

So who gets to decide what is and is not *appropriate*? Sun. Both the exam and this section use Sun's "official" assertion documentation to determine appropriate and inappropriate uses.

exam Watch

If you see the word "appropriate" on the exam, do not mistake that for "legal." *Appropriate* ***always refers to the way in which something is*** *supposed* ***to be used, according to either the developers of the mechanism or best practices officially embraced by Sun. If you see the word "correct" in the context of assertions, as in, "Line 3 is a correct use of assertions," you should also assume that*** *correct* ***is referring to how assertions*** *should* ***be used rather than how they legally*** *could* ***be used.***

Do not use assertions to validate arguments to a *public* method.

The following is an inappropriate use of assertions:

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

A *public* method might be called from code that you don't control (or have ever seen). Because *public* methods are part of your exposed interface to the outside world, you're supposed to guarantee that any constraints on the arguments will be enforced by the method itself. But since assertions aren't guaranteed to actually run (they're typically disabled in a deployed application), the enforcement won't happen if assertions aren't enabled. You don't want publicly accessible code that works only *conditionally*, depending on whether assertions are enabled or disabled.

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 `assert` 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* 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.

Do not 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 2: y = 3;
   case 3: y = 17;
   case 4: y = 27;
   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 2, 3, or 4, 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.

Do not 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() {
   x++ = y;
   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.

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 expression evaluates to *false*, then the code following the *else* will be performed. If the *else* is not used, then none of the code associated with the *if* statement will execute.

You also learned that the `switch` statement is used to replace multiple *if-else* statements. The *switch* statement can evaluate only integer primitive types that can be implicitly cast to an `int`. Those types are `byte`, `short`, `int`, and `char`. At runtime, the JVM will try to find a match between the argument to the *switch* statement and an argument in a corresponding *case* statement. If a match is found, execution will begin at the matching *case*, and continue on from there 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, the *while* loop, and the *do-while* 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-while* 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-while* 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* value, the code following the *case* value will be performed. To stop the execution of the code, the `break` statement 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 follow the *catch*. It is guaranteed to be called *except* in the special cases where the *try* or *catch* code 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 *catch* any checked exception it can throw or else *declare* that it *throws* the exception. The exception declaration is part of the method's *public* interface. To declare 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 *un*checked 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 is not good practice to handle an Error, though, because rarely can you do anything to recover from one.

Assertions, added to the language in version 1.4, are a useful new 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* an identifier, then you won't be able to use assertions, and you must recompile your older code using the default `-source 1.3` flag. If you do want to enable assertions in your code, then you must use the `-source 1.4` flag, causing the compiler to see *assert* as a keyword rather than an identifier.

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 Chapter 4. You might want to *loop* through them several times, but only *if* you're interested in passing the exam.

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 be only 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.

SELF TEST

The following questions will help you measure your understanding of the material presented in this chapter. You've heard this before, and this time we really mean it: this chapter's material is crucial for the exam! Regardless of what the exam question is really testing, there's a good chance that flow control code will be part of the question. Expect to see loops and *if* tests used in questions throughout the entire range of exam objectives.

Flow Control (if and switch) (Sun Objective 2.1)

1. Given the following,

```
1.   public class Switch2 {
2.     final static short x = 2;
3.     public static int y = 0;
4.     public static void main(String [] args) {
5.       for (int z=0; z < 3; z++) {
6.         switch (z) {
7.           case y: System.out.print("0 ");
8.           case x-1: System.out.print("1 ");
9.           case x: System.out.print("2 ");
10.        }
11.      }
12.    }
13.  }
```

what is the result?

A. 0 1 2

B. 0 1 2 1 2 2

C. Compilation fails at line 7.

D. Compilation fails at line 8.

E. Compilation fails at line 9.

F. An exception is thrown at runtime.

2. Given the following,

```
1.   public class Switch2 {
2.     final static short x = 2;
3.     public static int y = 0;
4.     public static void main(String [] args) {
5.       for (int z=0; z < 3; z++) {
6.         switch (z) {
```

```
7.            case x: System.out.print("0 ");
8.            case x-1: System.out.print("1 ");
9.            case x-2: System.out.print("2 ");
10.         }
11.       }
12.     }
13.   }
```

what is the result?

A. 0 1 2

B. 0 1 2 1 2 2

C. 2 1 0 1 0 0

D. 2 1 2 0 1 2

E. Compilation fails at line 8.

F. Compilation fails at line 9.

3. Given the following,

```
1.   public class If1 {
2.     static boolean b;
3.     public static void main(String [] args) {
4.       short hand = 42;
5.       if ( hand < 50 & !b ) hand++;
6.       if ( hand > 50 ) ;
7.       else if ( hand > 40 ) {
8.          hand += 7;
9.          hand++;    }
10.       else
11.         --hand;
12.       System.out.println(hand);
13.    }
14.  }
```

what is the result?

A. 41

B. 42

C. 50

D. 51

E. Compiler fails at line 5.

F. Compiler fails at line 6.

4. Given the following,

```
1.   public class Switch2 {
2.       final static short x = 2;
3.       public static int y = 0;
4.     public static void main(String [] args) {
5.       for (int z=0; z < 4; z++) {
6.         switch (z) {
7.           case x: System.out.print("0 ");
8.           default: System.out.print("def ");
9.           case x-1: System.out.print("1 ");  break;
10.          case x-2: System.out.print("2 ");
11.        }
12.      }
13.    }
14.  }
```

what is the result?

A. `0 def 1`

B. `2 1 0 def 1`

C. `2 1 0 def def`

D. `2 1 def 0 def 1`

E. `2 1 2 0 def 1 2`

F. `2 1 0 def 1 def 1`

5. Given the following,

```
1.   public class If2 {
2.     static boolean b1, b2;
3.     public static void main(String [] args) {
4.       int x = 0;
5.       if ( !b1 ) {
6.         if ( !b2 ) {
7.           b1 = true;
8.           x++;
9.           if ( 5 > 6 ) {
10.            x++;
11.          }
12.          if ( !b1 ) x = x + 10;
13.          else if ( b2 = true ) x = x + 100;
14.          else if ( b1 | b2 ) x = x + 1000;
15.        }
16.      }
```

```
17.       System.out.println(x);
18.     }
19.   }
```

what is the result?

A. 0

B. 1

C. 101

D. 111

E. 1001

F. 1101

Flow Control (loops) (Sun Objective 2.2)

6. Given the following,

```
1. public class While {
2.    public void loop() {
3.       int x= 0;
4.       while ( 1 ) {
5.          System.out.print("x plus one is " + (x + 1));
6.       }
7.    }
8. }
```

Which statement is true?

A. There is a syntax error on line 1.

B. There are syntax errors on lines 1 and 4.

C. There are syntax errors on lines 1, 4, and 5.

D. There is a syntax error on line 4.

E. There are syntax errors on lines 4 and 5.

F. There is a syntax error on line 5.

7. Given the following,

```
1.   class For {
2.     public void test() {
3.
4.         System.out.println("x = "+ x);
```

```
5.       }
6.     }
7.  }
```

and the following output,

```
x = 0
x = 1
```

which two lines of code (inserted independently) will cause this output? (Choose two.)

A. `for (int x = -1; x < 2; ++x) {`

B. `for (int x = 1; x < 3; ++x ) {`

C. `for (int x = 0; x > 2; ++x ) {`

D. `for (int x = 0; x < 2; x++ ) {`

E. `for (int x = 0; x < 2; ++x ) {`

8. Given the following,

```
1.   public class Test {
2.     public static void main(String [] args) {
3.       int I = 1;
4.       do while ( I < 1 )
5.         System.out.print("I is " + I);
6.       while ( I > 1 ) ;
7.     }
8.   }
```

what is the result?

A. `I is 1`

B. `I is 1 I is 1`

C. No output is produced.

D. Compilation error

E. `I is 1 I is 1 I is 1` in an infinite loop.

9. Given the following,

```
11.  int I = 0;
12.  outer:
13.    while (true) {
14.      I++;
15.      inner:
16.        for (int j = 0; j < 10; j++) {
```

```
17.          I += j;
18.          if (j == 3)
19.            continue inner;
20.          break outer;
21.        }
22.        continue outer;
23.     }
24.   System.out.println(I);
25.
26.
```

what is the result?

A. 1

B. 2

C. 3

D. 4

10. Given the following,

```
1.  int I = 0;
2.  label:
3.     if (I < 2) {
4.        System.out.print("I is " + I);
5.        I++;
6.        continue label;
7.     }
```

what is the result?

A. `I is 0`

B. `I is 0 I is 1`

C. Compilation fails.

D. None of the above

Exceptions (Sun Objectives 2.3 and 2.4)

11. Given the following,

```
1.  System.out.print("Start ");
2.  try {
3.     System.out.print("Hello world");
4.     throw new FileNotFoundException();
```

```
 5.  }
 6.  System.out.print(" Catch Here ");
 7.  catch(EOFException e) {
 8.     System.out.print("End of file exception");
 9.  }
10.  catch(FileNotFoundException e) {
11.     System.out.print("File not found");
12.  }
```

and given that EOFException and FileNotFoundException are both subclasses of IOException, and further assuming this block of code is placed into a class, which statement is most true concerning this code?

A. The code will not compile.

B. Code output: `Start Hello world File Not Found.`

C. Code output: `Start Hello world End of file exception.`

D. Code output: `Start Hello world Catch Here File not found.`

12. Given the following,

```
 1.  public class MyProgram {
 2.     public static void main(String args[]){
 3.        try {
 4.           System.out.print("Hello world ");
 5.        }
 6.        finally {
 7.           System.out.println("Finally executing ");
 8.        }
 9.     }
10.  }
```

what is the result?

A. Nothing. The program will not compile because no exceptions are specified.

B. Nothing. The program will not compile because no `catch` clauses are specified.

C. `Hello world.`

D. `Hello world Finally executing`

13. Given the following,

```
1.  import java.io.*;
2.  public class MyProgram {
3.     public static void main(String args[]){
4.        FileOutputStream out = null;
```

```
 5.         try {
 6.            out = new FileOutputStream("test.txt");
 7.            out.write(122);
 8.         }
 9.         catch(IOException io) {
10.             System.out.println("IO Error.");
11.         }
12.         finally {
13.            out.close();
14.         }
15.      }
16.   }
```

and given that all methods of class FileOutputStream, including `close()`, throw an IOException, which of these is true? (Choose one.)

A. This program will compile successfully.

B. This program fails to compile due to an error at line 4.

C. This program fails to compile due to an error at line 6.

D. This program fails to compile due to an error at line 9.

E. This program fails to compile due to an error at line 13.

14. Given the following,

```
 1.  public class MyProgram {
 2.     public static void throwit() {
 3.       throw new RuntimeException();
 4.     }
 5.     public static void main(String args[]){
 6.        try {
 7.           System.out.println("Hello world ");
 8.           throwit();
 9.           System.out.println("Done with try block ");
10.        }
11.        finally {
12.           System.out.println("Finally executing ");
13.        }
14.     }
15.  }
```

which answer most closely indicates the behavior of the program?

A. The program will not compile.

B. The program will print `Hello world`, then will print that a RuntimeException has occurred, then will print `Done with try block`, and then will print `Finally executing`.

C. The program will print `Hello world`, then will print that a RuntimeException has occurred, and then will print `Finally executing.`

D. The program will print `Hello world`, then will print `Finally executing`, then will print that a `RuntimeException` has occurred.

15. Given the following,

```
1.   public class RTExcept {
2.     public static void throwit () {
3.       System.out.print("throwit ");
4.       throw new RuntimeException();
5.     }
6.     public static void main(String [] args) {
7.       try {
8.         System.out.print("hello ");
9.         throwit();
10.      }
11.      catch (Exception re ) {
12.        System.out.print("caught ");
13.      }
14.      finally {
15.        System.out.print("finally ");
16.      }
17.      System.out.println("after ");
18.    }
19.  }
```

what is the result?

A. `hello throwit caught`

B. Compilation fails

C. `hello throwit RuntimeException caught after`

D. `hello throwit RuntimeException`

E. `hello throwit caught finally after`

F. `hello throwit caught finally after RuntimeException`

Assertions (Sun Objectives 2.5 and 2.6)

16. Which of the following statements is true?

A. In an *assert* statement, the expression after the colon (:) can be *any* Java expression.

B. If a *switch* block has no default, adding an *assert* default is considered appropriate.

C. In an *assert* statement, if the expression after the colon (:) does not have a value, the assert's error message will be empty.

D. It is appropriate to handle assertion failures using a `catch` clause.

17. Which two of the following statements are true? (Choose two.)

A. It is sometimes good practice to throw an AssertionError explicitly.

B. It is good practice to place assertions where you think execution should never reach.

C. Private `getter()` and `setter()` methods should not use assertions to verify arguments.

D. If an AssertionError is thrown in a `try-catch` block, the `finally` block will be bypassed.

E. It is proper to handle assertion statement failures using a *catch* (`AssertionException ae`) block.

18. Given the following,

```
1.   public class Test {
2.     public static int y;
3.     public static void foo(int x) {
4.       System.out.print("foo ");
5.       y = x;
6.     }
7.     public static int bar(int z) {
8.       System.out.print("bar ");
9.       return y = z;
10.    }
11.    public static void main(String [] args ) {
12.      int t = 0;
13.      assert t > 0 : bar(7);
14.      assert t > 1 : foo(8);
15.      System.out.println("done ");
16.    }
17.  }
```

what is the result?

A. `bar`

B. `bar done`

C. `foo done`

D. `bar foo done`

E. Compilation fails

F. An error is thrown at runtime.

19. Which two of the following statements are true? (Choose two.)

A. If assertions are compiled into a source file, and if no flags are included at runtime, assertions will execute by default.

B. As of Java version 1.4, assertion statements are compiled by default.

C. With the proper use of runtime arguments, it is possible to instruct the VM to disable assertions for a certain class, and to enable assertions for a certain package, at the same time.

D. The following are all valid runtime assertion flags:
`-ea`, `-esa`, `-dsa`, `-enableassertions`, `-disablesystemassertions`

E. When evaluating command-line arguments, the VM gives `-ea` flags precedence over `-da` flags.

20. Given the following,

```
1.   public class Test2 {
2.     public static int x;
3.     public static int foo(int y) {
4.       return y * 2;
5.     }
6.     public static void main(String [] args) {
7.       int z = 5;
8.       assert z > 0;
9.       assert z > 2: foo(z);
10.      if ( z < 7 )
11.      assert z > 4;
12.      switch (z) {
13.        case 4: System.out.println("4 ");
14.        case 5: System.out.println("5 ");
15.        default: assert z < 10;
16.      }
17.      if ( z < 10 )
18.        assert z > 4: z++;
19.      System.out.println(z);
20.    }
21.  }
```

which line is an example of an inappropriate use of assertions?

A. Line 8
B. Line 9
C. Line 11
D. Line 15
E. Line 18

SELF TEST ANSWERS

Flow Control (if and switch) (Sun Objective 2.1)

1. ☑ **C.** Case expressions must be constant expressions. Since *x* is marked `final`, lines 8 and 9 are legal; however *y* is not a `final` so the compiler will fail at line 7.
   ☒ **A, B, D, E,** and **F,** are incorrect based on the program logic described above.

2. ☑ **D.** The *case* expressions are all legal because `x` is marked `final`, which means the expressions can be evaluated at compile time. In the first iteration of the *for* loop `case x-2` matches, so 2 is printed. In the second iteration, `x-1` is matched so 1 and 2 are printed (remember, once a match is found all remaining statements are executed until a break statement is encountered). In the third iteration, `x` is matched so 0 1 and 2 are printed.
   ☒ **A, B, C, E, and F** are incorrect based on the program logic described above.

3. ☑ **D.** In Java, `boolean` instance variables are initialized to *false*, so the *if* test on line 5 is `true` and `hand` is incremented. Line 6 is legal syntax, a do nothing statement. The *else-if* is *true* so `hand` has 7 added to it and is then incremented.
   ☒ **A, B, C, E,** and **F** are incorrect based on the program logic described above.

4. ☑ **F.** When `z == 0 , case x-2` is matched. When `z == 1, case x-1` is matched and then the `break` occurs. When `z == 2, case x`, then `default`, then `x-1` are all matched. When `z == 3, default`, then `x-1` are matched. The rules for `default` are that it will fall through from above like any other `case` (for instance when `z == 2`), and that it will match when no other `case`s match (for instance when `z == 3`).
   ☒ **A, B, C, D,** and **E** are incorrect based on the program logic described above.

5. ☑ **C.** As instance variables, `b1` and `b2` are initialized to *`false`*. The *if* tests on lines 5 and 6 are successful so `b1` is set to `true` and `x` is incremented. The next *if* test to succeed is on line 13 (note that the code is not testing to see if `b2` is `true`, it is setting `b2` to be `true`). Since line 13 was successful, subsequent *else-if's* (line 14) will be skipped.
   ☒ **A, B, D, E,** and **F** are incorrect based on the program logic described above.

Flow Control (loops) (Sun Objective 2.2)

6. ☑ **D.** Using the integer 1 in the *while* statement, or any other looping or conditional construct for that matter, will result in a compiler error. This is old C syntax, not valid Java.
   ☒ **A, B, C, E,** and **F** are incorrect because line 1 is valid (Java is case sensitive so While is a valid class name). Line 5 is also valid because an equation may be placed in a String operation as shown.

7. ☑ **D** and **E.** It doesn't matter whether you preincrement or postincrement the variable in a *for* loop; it is always incremented after the loop executes and before the iteration expression is evaluated.
☒ **A** and **B** are incorrect because the first iteration of the loop must be zero. **C** is incorrect because the test will fail immediately and the *for* loop will not be entered.

8. ☑ **C.** There are two different looping constructs in this problem. The first is a *do-while* loop and the second is a *while* loop, nested inside the *do-while*. The body of the *do-while* is only a single statement—brackets are not needed. You are assured that the *while* expression will be evaluated at least once, followed by an evaluation of the *do-while* expression. Both expressions are *false* and no output is produced.
☒ **A, B, D,** and **E** are incorrect based on the program logic described above.

9. ☑ **A.** The program flows as follows: `I` will be incremented after the *while* loop is entered, then `I` will be incremented (by zero) when the *for* loop is entered. The *if* statement evaluates to *false*, and the `continue` statement is never reached. The `break` statement tells the JVM to break out of the *outer* loop, at which point `I` is printed and the fragment is done.
☒ **B, C,** and **D** are incorrect based on the program logic described above.

10. ☑ **C.** The code will not compile because a `continue` statement can only occur in a looping construct. If this syntax were legal, the combination of the `continue` and the *if* statements would create a kludgey kind of loop, but the compiler will force you to write cleaner code than this.
☒ **A, B,** and **D** are incorrect based on the program logic described above.

Exceptions (Sun Objectives 2.3 and 2.4)

11. ☑ **A.** Line 6 will cause a compiler error. The only legal statements after `try` blocks are either `catch` or `finally` statements.
☒ **B, C,** and **D** are incorrect based on the program logic described above. If line 6 was removed, the code would compile and the correct answer would be **B.**

12. ☑ **D.** `Finally` clauses are always executed. The program will first execute the `try` block, printing `Hello world`, and will then execute the `finally` block, printing `Finally executing`.
☒ **A, B,** and **C** are incorrect based on the program logic described above. Remember that either a `catch` or a `finally` statement must follow a `try`. Since the *finally* is present, the *catch* is not required.

13. ☑ **E.** Any method (in this case, the `main()` method) that throws a checked exception (in this case, `out.close()`) must be called within a `try` clause, or the method must declare that it throws the exception. Either `main()` must declare that it throws an exception, or the call to `out.close()` in the `finally` block must fall inside a (in this case nested) `try-catch` block.
☒ **A, B, C,** and **D** are incorrect based on the program logic described above.

14. ☑ **D.** Once the program throws a RuntimeException (in the `throwit()` method) that is not caught, the `finally` block will be executed and the program will be terminated. If a method does not handle an exception, the `finally` block is executed before the exception is propagated.
☒ **A, B,** and **C** are incorrect based on the program logic described above.

15. ☑ **E.** The `main()` method properly catches and handles the RuntimeException in the `catch` block, *finally* runs (as it always does), and then the code returns to normal.
☒ **A, B, C, D,** and **F** are incorrect based on the program logic described above. Remember that properly handled exceptions do not cause the program to stop executing.

Assertions (Sun Objectives 2.5 and 2.6)

16. ☑ **B.** Adding an assertion statement to a `switch` statement that previously had no default case is considered an excellent use of the assert mechanism.
☒ **A** is incorrect because only Java expressions that return a value can be used. For instance, a method that returns `void` is illegal. **C** is incorrect because the expression after the colon must have a value. **D** is incorrect because assertions throw errors and not exceptions, and assertion errors do cause program termination and should not be handled.

17. ☑ **A** and **B. A** is correct because it is sometimes advisable to thrown an assertion error even if assertions have been disabled. **B** is correct. One of the most common uses of `assert` statements in debugging is to verify that locations in code that have been designed to be unreachable are in fact never reached.
☒ **C** is incorrect because it is considered appropriate to check argument values in *private* methods using assertions. **D** is incorrect; *finally* is never bypassed. **E** is incorrect because AssertionErrors should never be handled.

18. ☑ **E.** The `foo()` method returns `void`. It is a perfectly acceptable method, but because it returns `void` it cannot be used in an *assert* statement, so line 14 will not compile.
☒ **A, B, C, D,** and **F** are incorrect based on the program logic described above.

19. ☑ **C** and **D. C** is true because multiple VM flags can be used on a single invocation of a Java program. **D** is true, these are all valid flags for the VM.
☒ **A** is incorrect because at runtime assertions are ignored by default. **B** is incorrect because as of Java 1.4 you must add the argument `-source 1.4` to the command line if you want the compiler to compile assertion statements. **E** is incorrect because the VM evaluates all assertion flags left to right.

20. ☑ **E.** *Assert* statements should not cause side effects. Line 18 changes the value of `z` if the assert statement is *false*.
☒ **A** is fine; a second expression in an *assert* statement is not required. **B** is fine because it is perfectly acceptable to call a method with the second expression of an *assert* statement. **C** is fine because it is proper to call an *assert* statement conditionally. **D** is fine because it is considered good form to add a default *assert* statement to *switch* blocks that have no default case.

EXERCISE ANSWERS

Exercise 4.1: Creating a switch-case Statement

The code should look something like this:

```
char temp = 'c';
switch(temp) {
   case 'a': {
      System.out.println("A");
      break;
   }
   case 'b': {
      System.out.println("B");
      break;
   }
   case 'c':
      System.out.println("C");
      break;
   default:
      System.out.println("default");
}
```

Exercise 4-2: Creating a Labeled while Loop

The code should look something like this:

```
class LoopTest {
   public static void main(String [] args) {
      int age = 12;
      outer:
      while(age < 21) {
         age += 1;
         if(age == 16) {
            System.out.println("Obtain driver's license");
            continue outer;
         }
         System.out.println("Another year.");
      }
   }
}
```

Exercise 4-3: Propagating and Catching an Exception

The code should look something like this:

```
class Propagate {
   public static void main(String [] args) {
      try {
         System.out.println(reverse("Hello"));
      }
      catch (Exception e) {
         System.out.println("The string was blank");
      }
      finally {
         System.out.println("All done!");
      }
   }
   public static String reverse(String s) throws Exception {
      if (s.length() == 0 ) {
         throw new Exception();
      }
      String reverseStr = "";
      for(int i=s.length()-1;i>=0;--i) {
         reverseStr += s.charAt(i);
      }
      return reverseStr;
   }
}
```

Exercise 4-4: Creating an Exception

The code should look something like this:

```
class BadFoodException extends Exception {}
class MyException {
   public static void main(String [] args) {
     try {
       checkFood(args[0]);
     } catch(BadFoodException e) {
        e. printStackTrace();
     }
   }
   public static void checkWord(String s) {
      String [] badFoods = {"broccoli","brussel sprouts","sardines"};
      for(int i=0;i<badFoods.length;++i) {
         if (s.equals(badFoods[i]))
            throw new BadWFoodException();
      }
      System.out.println(s + " is ok with me.");
   }
}
```