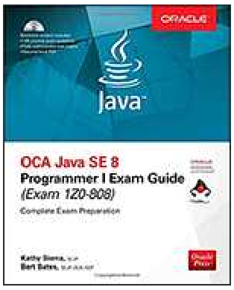

OCA Java SE 8 Programmer I Exam Guide (Exams 1Z0-808)

by Kathy Sierra and Bert Bates

Chapter 5: Flow Control and Exceptions

CERTIFICATION OBJECTIVES

- Use if and switch Statements
- Develop for, do, and while Loops
- Use break and continue Statements
- Use try, catch, and finally Statements
- State the Effects of Exceptions
- Recognize Common Exceptions
- Two-Minute Drill
- Self Test

Can you imagine trying to write code using a language that didn't give you a way to execute statements conditionally? Flow control is a key part of most any useful programming language, and Java offers several ways to accomplish it. Some statements, such as `if` statements and `for` loops, are common to most languages. But Java also throws in a couple of flow control features you might not have used before—exceptions and assertions. (We'll discuss assertions in the next chapter.)

> The `if` statement and the `switch` statement are types of conditional/decision controls that allow your program to behave 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`—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.
>
> 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: USING IF AND SWITCH STATEMENTS (OCA OBJECTIVES 3.3 AND 3.4)

3.3 Create if and if-else and ternary constructs.

3.5 Use a switch statement.

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`. 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. The following code demonstrates a legal `if-else` 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)   // bad practice, but seen on the exam
  y = 2;
z += 8;
a = y + x;
```

Most developers consider it good practice to enclose blocks within curly braces, even if there's only one statement in the block. Be careful with code like the preceding, 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, it's 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 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();
  }
}
```

This brings up the other `if-else` construct, the `if`, `else if`, `else`. The preceding code could (and should) be rewritten like this:

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

There are a couple of rules for using `else` and `else if`:

- You can have zero or one `else` for a given `if`, and it must come after any `else ifs`.
- You can have zero to many `else ifs` for a given `if`, and they must come before the (optional) `else`.
- Once an `else if` succeeds, none of the remaining `else ifs` nor the `else` will be tested.

The following example shows code that is horribly formatted for the real world. As you've probably guessed, it's fairly likely that you'll encounter formatting like this on the exam. In any case, the code demonstrates the use of multiple `else ifs`:

```
int x = 1;
if ( x == 3 ) { }
else if (x < 4) {System.out.println("<4"); }
else if (x < 2) {System.out.println("<2"); }
else { System.out.println("else"); }
```

It produces this output:

```
<4
```

(Notice that even though the second `else if` is true, it is never reached.)

Sometimes you can have a problem figuring out which `if` your `else` should pair with, 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.

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

```
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 Expressions for if Statements

The expression in an `if` statement must be a `boolean` expression. Any expression that resolves to a `boolean` is fine, and 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");
}
```

This 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 &&, the expression is evaluated as though there were parentheses around `(y < 2) | doStuff()`. In other words, it is evaluated as a single expression before the && and a single expression after the &&.

Remember that the only legal expression in an `if` test is a `boolean`. In some languages, 0 == false and 1 == true. Not so in Java! The following code shows `if` statements that might look tempting but are illegal, followed by legal substitutions:

```
int trueInt = 1;
int falseInt = 0;
if (trueInt)           // illegal
if (trueInt == true)   // illegal
```

```
if (1)                  // illegal
if (falseInt == false) // illegal
if (trueInt == 1)      // legal
if (falseInt == 0)     // 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 variables that can be assigned (rather than tested against something else) are a `boolean` or a `Boolean;` all other assignments will result in something non-`boolean`, 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

You've seen how `if` and `else-if` statements can be used to support both simple and complex decision logic. In many cases, the `switch` statement provides a cleaner way to handle complex decision logic. Let's compare the following `if-else if` statement to the equivalently performing `switch` statement:

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

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

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

Note: The reason this `switch` statement emulates the `if` is because of the `break` statements that were placed inside of the `switch`. In general, `break` statements are optional, and as you will see in a few pages, their inclusion or exclusion causes huge changes in how a `switch` statement will execute.

Legal Expressions for switch and case

The general form of the `switch` statement is

```
switch (expression) {
```

```
  case constant1: code block
  case constant2: code block
  default: code block
}
```

A `switch`'s expression must evaluate to a `char`, `byte`, `short`, `int`, an `enum` (as of Java 5), and a `String` (as of Java 7). That means if you're not using an `enum` or a `String`, only variables and values that can be automatically promoted (in other words, implicitly cast) to an `int` are acceptable. You won't be able to compile if you use anything else, including the remaining numeric types of `long`, `float`, and `double`.

A `case` constant must evaluate to the same type that the `switch` expression can use, with one additional—and big—constraint: the `case` constant must be a compile-time constant! Since the `case` argument has to be resolved at compile time, you can use only a constant or `final` variable that is immediately initialized with a literal value. It is not enough to be `final`; it must be a compile-time *constant*. Here's an example:

```
final int a = 1;
final int b;
b = 2;
int x = 0;
switch (x) {
  case a:     // ok
  case b:     // compiler error
```

Also, the `switch` can only check for equality. This means 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");
}
```

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`:

```
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 something like this:

```
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 `case`s with the same value of 80:

```
int temp = 90;
switch(temp) {
  case 80 :  System.out.println("80");
  case 80 :  System.out.println("80");   // won't compile!
  case 90 :  System.out.println("90");
  default :  System.out.println("default");
}
```

It *is* legal to leverage the power of boxing in a `switch` expression. For instance, the following is legal:

```
switch(new Integer(4)) {
  case 4: System.out.println("boxing is OK");
}
```

Exam Watch

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

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

switch(x) {
  0: { }
  1: { }
}
```

In the first example, the `case` omits the colon. The second example omits the keyword `case.`

An Intro to String "equality"

As we've been discussing, the operation of `switch` statements depends on the expression "matching" or being "equal" to one of the cases. We've talked about how we know when primitives are equal, but what does it mean for objects to be equal? This is another one of those surprisingly tricky topics, and for those of you who intend to take the OCP exam, you'll spend a lot of time studying "object equality." For you OCA candidates, all you have to know is that for a `switch` statement, two `Strings` will be considered "equal" if they have the same case-sensitive sequence of characters. For example, in the following partial `switch` statement, the expression would match the case:

```
String s = "Monday";
switch(s) {
  case "Monday":   // matches!
```

But the following would NOT match:

```
String s = "MONDAY";
switch(s) {
  case "Monday":   // Strings are case-sensitive, DOES NOT match
```

Break and Fall-Through in switch Blocks

We're finally ready to discuss the `break` statement and offer more details about flow control within a `switch` statement. The most important thing to remember about the flow of execution through a `switch` statement is this:

> `case` constants are evaluated from the top down, and the first `case` constant that matches the `switch`'s expression is the execution *entry point*.

In other words, once a `case` constant is matched, the Java Virtual Machine (JVM) will execute the associated code block and ALL subsequent code blocks (barring a `break` statement), too! The following example uses a `String` in a `case` statement:

```
class SwitchString {
  public static void main(String [] args) {
    String s = "green";
    switch(s) {
      case "red": System.out.print("red ");
      case "green": System.out.print("green ");
      case "blue": System.out.print("blue ");
      default: System.out.println("done");
    }
  }
}
```

In this example `case "green":` matched, so the JVM executed that code block and all subsequent code blocks to produce the output:

```
green blue done
```

Again, 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 remaining `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; 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. Remember, the matching `case` is 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, prints:

```
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 and then hit the `break` and jumped to the end of the `switch`.

An interesting example 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.

Note: Because fall-through is less than intuitive, Oracle recommends that you add a comment such as `// fall through` when you use fall-through logic.

The Default Case

What if, using the preceding code, you wanted to print `x is an odd number` if none of the `case`s (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'd 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 even"); 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 this:

```
2
default
3
4
```

And if we modify it so the only match is the `default case,` like this,

```
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");
}
```

then running the preceding code prints this:

```
default
3
4
```

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

Exercise 5-1: Creating a switch-case Statement

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

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

CERTIFICATION OBJECTIVE: CREATING LOOPS CONSTRUCTS (OCA OBJECTIVES 5.1, 5.2, 5.3, 5.4, AND 5.5)

5.1 Create and use while loops.

5.2 Create and use for loops including the enhanced for loop.

5.3 Create and use do/while loops.

5.4 Compare loop constructs.

5.5 Use break and continue.

Java loops come in three flavors: `while`, `do`, and `for` (and as of Java 5, the `for` loop has two variations). 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 when you don't know how many times a block or statement should repeat but you want to continue looping as long as some condition is true. A `while` statement looks like this:

```
while (expression) {
  // do stuff
}
```

Or 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. The body of the `while` loop will execute only if the expression (sometimes called the "condition") results in a value of `true`. Once inside the loop, the loop body will repeat until the condition is no longer met because it 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.

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 this:

```
while (int x = 2) { } // not legal
```

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 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
```

Because the expression `(x > 8)` evaluates to `false`, none of the code within the `while` loop ever executes.

Using do Loops

The `do` loop is similar to the `while` loop, except the expression is not evaluated until after the `do` loop's code is executed. Therefore, the code in a `do` loop is guaranteed to execute at least once. The following shows a `do` loop 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`. Remember, the `do` 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` 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
                     // (as the result of assignment)
while (x == 5) { }   // Legal, equality test
while (true) { }     // Legal
```

Using for Loops

As of Java 5, the `for` loop took on a second structure. We'll call the old style of `for` loop the "basic `for` loop," and we'll call the new style of `for` loop the "enhanced `for` loop" (it's also sometimes called the `for-each`). Depending on what documentation you use, you'll see both terms, along with `for-in`. The terms `for-in`, `for-each`, and "enhanced `for`" all refer to the same Java construct.

The basic `for` loop is more flexible than the enhanced `for` loop, but the enhanced `for` loop was designed to make iterating through arrays and collections easier to code.

The Basic for Loop

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

The three `for` declaration parts are separated by semicolons. 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 a typical example of a `for` loop:

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

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

The Basic for Loop: 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, you'll need to separate them with commas as follows:

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

The declaration and initialization happen 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 happen 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 something like this:

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

Basic for Loop: 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.

Basic for Loop: Iteration Expression

After each execution of the body of the `for` loop, the iteration expression is executed. This 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 that doesn't change the value of x
}
```

This 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.

Keep in mind that barring a forced exit, evaluating the iteration expression and then evaluating the conditional expression are always the last two things that happen in a `for` loop!

Examples of forced exits include a `break`, a return, a `System.exit()`, and an exception, which 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 prints only 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 5-1 lists the causes and results of abrupt loop termination.

Basic for Loop: 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");
}
```

Table 5-1: Causes of Early Loop Termination

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

In this example, all the declaration parts are left out, so the `for` loop will act like an endless loop.

For the exam, it's important to know that with the absence of the 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);
```

But 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

Many questions in the Java 8 exams list "Compilation fails" and "An exception occurs at runtime" as possible answers, making them 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 a compiler error and then look for possible exceptions. Only after you've satisfied those two should you dig into the logic and flow of the code in the question.

The Enhanced for Loop (for Arrays)

The enhanced `for` loop, new as of Java 5, is a specialized `for` loop that simplifies looping through an array or a collection. In this chapter we're going to focus on using the enhanced `for` to loop through arrays. In Chapter 6 we'll revisit the enhanced `for`, when we discuss the ArrayList collection class—where the enhanced `for` really comes into its own.

Instead of having *three* components, the enhanced `for` has *two*. Let's loop through an array the basic (old) way and then using the enhanced `for`:

```
int [] a = {1,2,3,4};
for(int x = 0; x < a.length; x++)   // basic for loop
  System.out.print(a[x]);
for(int n : a)                      // enhanced for loop
  System.out.print(n);
```

This produces the following output:

```
12341234
```

More formally, let's describe the enhanced `for` as follows:

```
for(declaration : expression)
```

The two pieces of the `for` statement are

- **`declaration`** The *newly declared* block variable of a type compatible with the elements of the array you are accessing. This variable will be available within the `for` block, and its value will be the same as the current array element.
- **`expression`** This must evaluate to the array you want to loop through. This could be an array variable or a method call that returns an array. The array can be any type: primitives, objects, or even arrays of arrays.

Using the preceding definitions, let's look at some legal and illegal enhanced `for` declarations:

```
int x;
long x2;
long [] la = {7L, 8L, 9L};
int [][] twoDee = {{1,2,3}, {4,5,6}, {7,8,9}};
String [] sNums = {"one", "two", "three"};
Animal [] animals = {new Dog(), new Cat()};

// legal 'for' declarations
for(long y : la ) ;        // loop thru an array of longs
for(int[] n : twoDee) ;    // loop thru the array of arrays
for(int n2 : twoDee[2]) ;  // loop thru the 3rd sub-array
```

```
for(String s : sNums) ;    // loop thru the array of Strings
for(Object o : sNums) ;    // set an Object reference to
                           // each String
for(Animal a : animals) ;  // set an Animal reference to each
                           // element

// ILLEGAL 'for' declarations
for(x2 : la) ;             // x2 is already declared
for(int x4 : twoDee) ;     // can't stuff an array into an int
for(int x3 : la) ;         // can't stuff a long into an int
for(Dog d : animals) ;     // you might get a Cat!
```

The enhanced `for` loop assumes that, barring an early exit from the loop, you'll always loop through every element of the array. The following discussions of `break` and `continue` apply to both the basic and enhanced `for` loops.

Using break and continue

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

Remember, `continue` statements must be inside a loop; otherwise, you'll get a compiler error. `break` statements must be used inside either a loop or a `switch` statement.

The `break` statement causes the program to stop execution of the innermost loop 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:

```
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` statements 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 an example of an unlabeled `continue` statement:

```
while (!EOF) {
                      // read a field from a file
  if (wrongField) {
    continue;         // move to the next field in the file
  }
                      // otherwise do other stuff with the field
}
```

In this example, a file is being read 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 after the loop. 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

Although many statements in a Java program can be labeled, it's most common to use labels with loop statements like `for` or `while`, in conjunction with `break` and `continue` statements. A label statement must be placed just before the statement being labeled, and it consists of a valid identifier that ends with a colon (`:`).

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 they 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.

Here's an example of what a label looks like:

```
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:

```
boolean isTrue = true;
outer:
  for(int i=0; i<5; i++) {
    while (isTrue) {
      System.out.println("Hello");
      break outer;
    }     // end of inner while 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 `Good-Bye`.

Let's see what will happen if the `continue` statement is used instead of the `break` statement. The following code example is similar to the preceding one, with the exception of substituting `continue` for `break`:

```
outer:
  for (int i=0; i<5; 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
Good-Bye
```

In this example, `Hello` will be printed five 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`, this loop will finish and `Good-Bye` will be printed.

Exercise 5-2: 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, increment `age` by 1. Every time the program goes through the loop, check whether `age` is 16. If it is, print the message "get your driver's license" and continue to the outer loop. If not, print "Another year."

- The `outer` label should appear just before the `while` loop begins.
- 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 (OCA OBJECTIVES 8.1, 8.2, 8.3, 8.4, AND 8.5)

8.1 Differentiate among checked exceptions, unchecked exceptions, and errors.

8.2 Create a try-catch block and determine how exceptions alter normal program flow.

8.3 Describe the advantages of Exception handling.

8.4 Create and invoke a method that throws an exception.

8.5 Recognize common exception classes (such as NullPointerException, ArithmeticException, ArrayIndexOutOfBoundsException, ClassCastException) (sic)

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. Here are some of the benefits of Java's exception-handling features:

- It arms developers with an elegant mechanism for handling errors that produces efficient and organized error-handling code.
- It allows developers to detect errors easily without writing special code to test return values.
- It lets us keep exception-*handling* code cleanly separated from exception-*generating* code.
- It also lets us use the same exception-handling code to deal with a range of possible exceptions.

Java 7 added several new exception-handling capabilities to the language. For our purposes, Oracle split the various exception-handling topics into two main parts:

1. The OCA exam covers the Java 6 version of exception handling.
2. The OCP exam adds the new exception features added in Java 7.

In order to mirror Oracle's OCA 8 objectives versus the OCP 8 objectives, this chapter will give you only the basics of exception handling—but plenty to handle the OCA 8 exam.

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 group 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 7 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 of the guarded region starts at line 2. If the program executes all the way past 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 of type `MyFirstException` is thrown, execution will immediately transfer to line 7. 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 remaining lines in the `try` block (4 and 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 that your code looks something like this pseudocode:

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

This 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 also guaranteed to 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

Although `try` and `catch` provide a terrific mechanism for trapping and handling exceptions, we are left with the problem of how to clean up after ourselves if an exception occurs. 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 each of the `catch` blocks—let's see why.

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 is encountered and before the `return` executes!

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 exception
 6: }
 7: catch(MySecondException) {
 8:   // Put code here that handles this exception
 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! Okay, 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.

Remember, `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 a `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` clause must immediately follow the last `catch` clause (or it must immediately follow the `try` block if there is no `catch`). It is legal to omit either the `catch` clause or the `finally` clause, but not both.

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()`, which in turn calls method `c()`, the call stack consists of the following:

```
b
a
main
```

We will represent the stack 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. 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 5-1 illustrates a way to think about how the call stack in Java works.

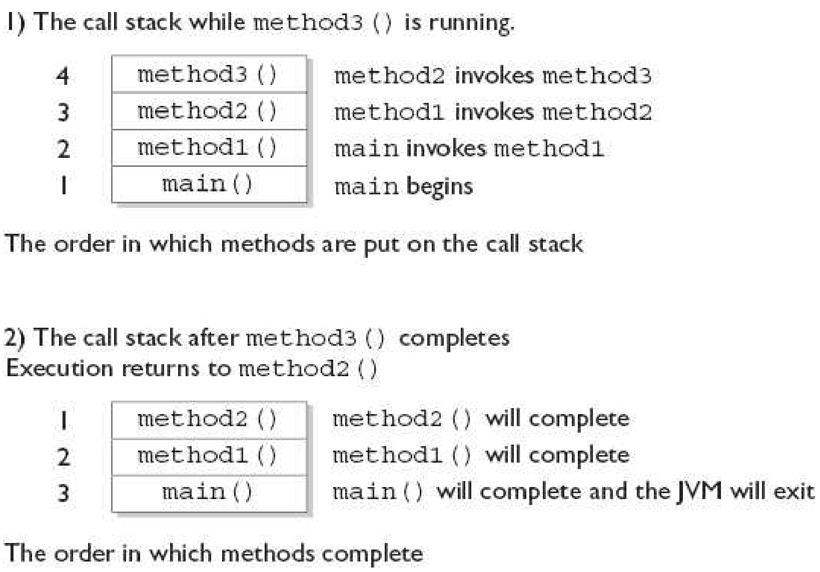

Figure 5-1: The Java method call stack

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 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 it is caught or until it reaches the very bottom of the call stack. This is called "exception propagation."

If an exception reaches 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.

Exam Watch

You can keep throwing an exception down through the methods on the stack. But what happens when you get to the `main()` method at the bottom? You can throw the exception out of `main()` as well. This results in the 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
  }
}
```

It prints out a stack trace something like this:

```
 %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 5-3: Propagating and Catching an Exception

In this exercise, you're going 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, `main()` must deal with it. A `finally` statement will be included to indicate that the program has completed. The method that `main()` will call will be named `reverse`, and it will reverse the order of the characters in a `String`. If the `String` contains no characters, `reverse` will propagate an exception up to the `main()` method.

1. Create a 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. In `reverse`, check whether the `String` has a length of 0 by using the `String.length()` method. If the length is 0, the `reverse` method will throw an exception.
4. Now 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 you will attempt to call this method and deal with any potential exceptions. Additionally, you will include a `finally` statement that displays when `main()` 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 different. Every exception is 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 the `ArrayIndexOutOfBoundsException` class. 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 5-2 shows the hierarchy for the exception classes.

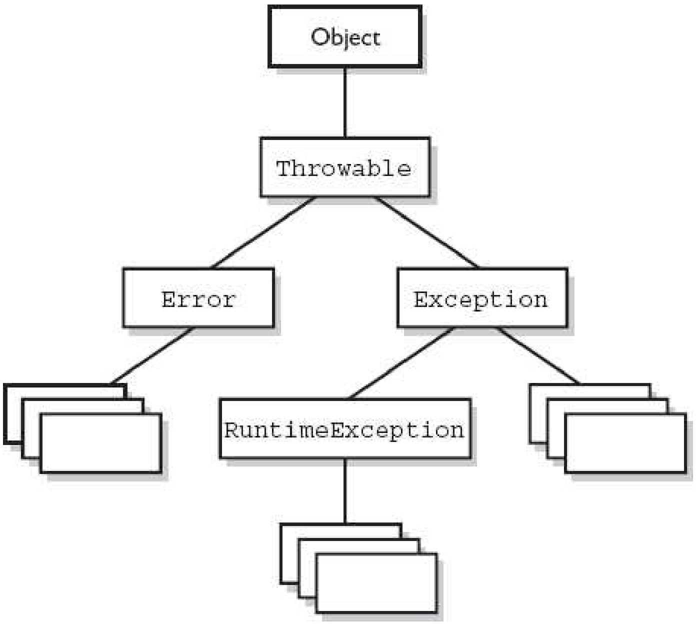

Figure 5-2: Exception class hierarchy

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 and indicate things that would not 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 answering, this is an exception that is not caused by a bug. Figure 5-2 also shows a subtype of `Exception` called `RuntimeException`. These exceptions are a special case because they sometimes 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 you would expect, 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, you don't need 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 `throw` 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's 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 (by looking at the available `catch` clauses from the top down) 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 a compiler error something like this:

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

If you think back to the people with baseball mitts (in the section "Propagating Uncaught Exceptions"), imagine that the most general mitts are the largest and can thus catch many 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 because 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 can throw one of two types of exceptions: either type `MyException1` or type `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 will 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 (which uses the `throw` keyword to throw an exception manually–more on this next) 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 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 {
    // code that actually could throw 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 either have to catch the exception or declare that we threw 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() {
  // code that could throw a NullPointerException goes here
}
```

According to the comment, this method can throw a `NullPointerException`. Because `RuntimeException` is the 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 are *checked* exceptions, and they don't derive from `java.lang.RuntimeException`. 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 they're handled or declared. A number of the methods in the Java API 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 this:

```
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 this:

```
MyException.java:3: unreported exception MyException;
must be caught or declared to be thrown
doStuff();
       ^
```

Notice that `someMethod()` fails either to handle or declare the exception that can be thrown by `doStuff()`. In the next pages, we'll discuss several ways to deal with this sort of situation.

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 `throw` keyword. When an `Error` or a subclass of `Error` (like `StackOverflowError`) is thrown, it's unchecked. You are not required to catch `Error` objects or `Error` subtypes. You can also throw an `Error` yourself (although, other than `AssertionError`, 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.

on the job Because Java has checked exceptions, it's commonly said that Java forces developers to handle exceptions. 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 something like this:

```
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; if a `finally` block exists, it runs, 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 (in the method signature)!"

Exercise 5-4: 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 determine whether a command-line argument representing a particular food (as a string) is considered bad or okay.

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 exception, because if `main()` ducks the exception, who else is back there to catch it? (Actually, `main()` can legally declare exceptions, but don't do that in this exercise.)

As nifty as exception handling is, it's still up to the developer to make proper use of it. Exception handling makes organizing 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 your code will be reusable, readable, and maintainable.

CERTIFICATION OBJECTIVE: COMMON EXCEPTIONS AND ERRORS (OCA OBJECTIVE 8.5)

8.5 Recognize common exception classes (such as NullPointerException, ArithmeticException, ArrayIndexOutOfBoundsException, ClassCastException) (sic)

The intention of this objective is to make sure that you are familiar with some of the most common exceptions and errors you'll encounter as a Java programmer.

Exam Watch

The questions from this section are likely to be along the lines of, "Here's some code that just did something bad, which exception will be thrown?" Throughout the exam, questions will present some code and ask you to determine whether the code will run or whether an exception will be thrown. Since these questions are so common, understanding the causes for these exceptions is critical to your success.

This is another one of those objectives that will turn up all through the real exam (does "An exception is thrown at runtime" ring a bell?), so make sure this section gets a lot of your attention.

Where Exceptions Come From

Jump back a page and take a look at the last sentence. It's important that you understand what causes exceptions and errors and where they come from. For the purposes of exam preparation, let's define two broad categories of exceptions and errors:

- **JVM exceptions** Those exceptions or errors that are either exclusively or most logically thrown by the JVM

- **Programmatic exceptions** Those exceptions that are thrown explicitly by application and/or API programmers

JVM-Thrown Exceptions

Let's start with a very common exception, the `NullPointerException`. As we saw in earlier chapters, this exception occurs when you attempt to access an object using a reference variable with a current value of `null`. There's no way that the compiler can hope to find these problems before runtime. Take a look at the following:

```
class NPE {
  static String s;
  public static void main(String [] args) {
    System.out.println(s.length());
  }
}
```

Surely, the compiler can find the problem with that tiny little program! Nope, you're on your own. The code will compile just fine, and the JVM will throw a `NullPointerException` when it tries to invoke the `length()` method.

Earlier in this chapter we discussed the call stack. As you recall, we used the convention that `main()` would be at the bottom of the call stack, and that as `main()` invokes another method, and that method invokes another, and so on, the stack grows upward. Of course, the stack resides in memory, and even if your OS gives you a gigabyte of RAM for your program, it's still a finite amount. It's possible to grow the stack so large that the OS runs out of space to store the call stack. When this happens, you get (wait for it…) a `StackOverflowError`. The most common way for this to occur is to create a recursive method. A recursive method invokes itself in the method body. Although that may sound weird, it's a very common and useful technique for such things as searching and sorting algorithms.

Take a look at this code:

```
void go() {   // recursion gone bad
   go();
}
```

As you can see, if you ever make the mistake of invoking the `go()` method, your program will fall into a black hole—`go()` invoking `go()` invoking `go()`, until, no matter how much memory you have, you'll get a `StackOverflowError`. Again, only the JVM knows when this moment occurs, and the JVM will be the source of this error.

Programmatically Thrown Exceptions

Now let's look at programmatically thrown exceptions. Remember we defined *programmatically* as meaning something like this:

> Created by an application and/or API developer

For instance, many classes in the Java API have methods that take `String` arguments and convert these `Strings` into numeric primitives. A good example of these classes is the so-called "wrapper classes" that we will study in Chapter 6. Even though we haven't talked much about wrapper classes yet, the following example should make sense.

At some point long ago, some programmer wrote the `java.lang.Integer` class and created methods like `parseInt()` and `valueOf()`. That programmer wisely decided that if one of these methods was passed a `String` that could not be converted into a number, the method should throw a `NumberFormatException`. The partially implemented code might look something like this:

```
int parseInt(String s) throws NumberFormatException {
  boolean parseSuccess = false;
  int result = 0;
  // do complicated parsing
  if (!parseSuccess)   // if the parsing failed
    throw new NumberFormatException();
  return result;
}
```

Other examples of programmatic exceptions include an `AssertionError` (okay, it's not an exception, but it IS thrown programmatically) and throwing an `IllegalArgumentException`. In fact, our mythical API developer could have used `IllegalArgumentException` for her `parseInt()` method. But it turns out that `NumberFormatException` extends `IllegalArgumentException` and is a little more precise, so in this case, using `NumberFormatException` supports the notion we discussed earlier: that when you have an exception hierarchy, you should use the most precise exception that you can.

Of course, as we discussed earlier, you can also make up your very own special custom exceptions and throw them whenever you want to. These homemade exceptions also fall into the category of "programmatically thrown exceptions."

A Summary of the Exam's Exceptions and Errors

OCA 8 Objective 8.5 lists a few specific exceptions and errors; it says "Recognize common exception classes (such as…."). Table 5-2 summarizes the ten exceptions and errors that are most likely a part of the OCA 8 exam.

Table 5-2: Descriptions and Sources of Common Exceptions

Exception	Description	Typically Thrown
`ArrayIndexOutOfBoundsException` (this chapter)	Thrown when attempting to access an array with an invalid index value (either negative or beyond the length of the array).	By the JVM
`ClassCastException` (Chapter 2)	Thrown when attempting to cast a reference variable to a type that fails the IS-A test.	By the JVM
`IllegalArgumentException`	Thrown when a method receives an argument formatted differently than the method expects.	Programmatically
`IllegalStateException`	Thrown when the state of the environment doesn't match the operation being attempted—for example, using a scanner that's been closed.	Programmatically
`NullPointerException` (Chapter 3)	Thrown when attempting to invoke a method on, or access a property from, a reference variable whose current value is `null`.	By the JVM
`NumberFormatException` (this chapter)	Thrown when a method that converts a `String` to a number receives a `String` that it cannot convert.	Programmatically
`ArithmeticException`	Thrown when an illegal math operation (such as dividing by zero) is attempted.	By the JVM
`ExceptionInInitializerError` (Chapter 2)	Thrown when attempting to initialize a static variable or an initialization block.	By the JVM
`StackOverflowError` (this chapter)	Typically thrown when a method recurses too deeply. (Each invocation is added to the stack.)	By the JVM
`NoClassDefFoundError`	Thrown when the JVM can't find a class it needs, because of a command-line error, a classpath issue, or a missing `.class` file.	By the JVM

CERTIFICATION SUMMARY

This chapter covered a lot of ground, all of which involved ways of controlling your program flow based on a conditional test. First, you learned about `if` and `switch` statements. The `if` statement evaluates one or more expressions to a `boolean` result. If the result is `true`, the program will execute the code in the block that is encompassed by the `if`. If an `else` statement is used and the `if` expression evaluates to `false`, then the code following the `else` will be performed. If no `else` block is defined, then none of the code associated with the `if` statement will execute.

You also learned that the `switch` statement can be used to replace multiple `if-else` statements. The `switch` statement can evaluate integer primitive types that can be implicitly cast to an `int` (those types are `byte`, `short`, `int`, and `char`); or it can evaluate `enums`; and as of Java 7, it can evaluate `Strings`. At runtime, the JVM will try to find a match between the expression in the `switch` statement and a constant in a corresponding `case` statement. If a match is found, execution will begin at the matching case and continue on from there, executing code in all the remaining `case` statements until a `break` statement is found or the end of the `switch` statement occurs. If there is no match, then the `default` case will execute, if there is one.

You've learned about the three looping constructs available in the Java language. These constructs are the `for` loop (including the basic `for` and the enhanced `for`, which was new to Java 5), the `while` loop, and the `do` loop. In general, the `for` loop is used when you know how many times you need to go through the loop. The `while` loop is used when you do not know how many times you want to go through, whereas the `do` loop is used when you need to go through at least once. In the `for` loop and the `while` loop, the expression has to evaluate to `true` to get inside the block and will check after every iteration of the loop. The `do` loop does not check the condition until after it has gone through the loop once. The major benefit of the `for` loop is the ability to initialize one or more variables and increment or decrement those variables in the `for` loop definition.

The `break` and `continue` statements can be used in either a labeled or unlabeled fashion. When unlabeled, the `break` statement will force the program to stop processing the innermost looping construct and start with the line of code following the loop. Using an unlabeled `continue` command will cause the program to stop execution of the current iteration of the innermost loop and proceed with the next iteration. When a `break` or a `continue` statement is used in a labeled manner, it will perform in the same way, with one exception: the statement will not apply to the innermost loop; instead, it will apply to the loop with the label. The `break` statement is used most often in conjunction with the `switch` statement. When there is a match between the `switch` expression and the `case` constant, the code following the `case` constant will be performed. To stop execution, a `break` is needed.

You've seen how Java provides an elegant mechanism in exception handling. Exception handling allows you to isolate your error-correction code into separate blocks so the main code doesn't become cluttered by error-checking code. Another elegant feature allows you to handle similar errors with a single error-handling block, without code duplication. Also, the error handling can be deferred to methods further back on the call stack.

You learned that Java's `try` keyword is used to specify a guarded region—a block of code in which problems might be detected. An exception handler is the code that is executed when an exception occurs. The handler is defined by using Java's `catch` keyword. All `catch` clauses must immediately follow the related `try` block.

Java also provides the `finally` keyword. This is used to define a block of code that is always executed, either immediately after a `catch` clause completes or immediately after the associated `try` block in the case that no exception was thrown (or there was a `try` but no `catch`). Use `finally` blocks to release system resources and to perform any cleanup required by the code in the `try` block. A `finally` block is not required, but if there is one, it must immediately follow the last `catch`. (If there is no `catch` block, the `finally` block must immediately follow the `try` block.) It's guaranteed to be called except when the `try` or `catch` issues a `System.exit()`.

An exception object is an instance of class `Exception` or one of its subclasses. The `catch` clause takes, as a parameter, an instance of an object of a type derived from the `Exception` class. Java requires that each method either catches any checked exception it can throw or else declares that it throws the exception. The exception declaration is part of the method's signature. To declare that an exception may be thrown, the `throws` keyword is used in a method definition, along with a list of all checked exceptions that might be thrown.

Runtime exceptions are of type `RuntimeException` (or one of its subclasses). These exceptions are a special case because they do not need to be handled or declared, and thus are known as "unchecked" exceptions. Errors are of type `java.lang.Error` or its subclasses, and like runtime exceptions, they do not need to be handled or declared. Checked exceptions include any exception types that are not of type `RuntimeException` or `Error`. If your code fails either to handle a checked exception or declare that it is thrown, your code won't compile. But with unchecked exceptions or objects of type `Error`, it doesn't matter to the compiler whether you declare them or handle them, do nothing about them, or do some combination of declaring and handling. In other words, you're free to declare them and handle them, but the compiler won't care one way or the other. It's not good practice to handle an `Error`, though, because you can rarely recover from one.

Finally, remember that exceptions can be generated by the JVM or by a programmer.

TWO-MINUTE DRILL

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

Writing Code Using if and switch Statements (OCA Objectives 3.3 and 3.4)

- The only legal expression in an `if` statement is a `boolean` expression—in other words, an expression that resolves to a `boolean` or a `Boolean` reference.
- Watch out for `boolean` assignments (=) that can be mistaken for `boolean` equality (==) tests:

  ```
  boolean x = false;
  if (x = true) { } // an assignment, so x will always be true!
  ```
- Curly braces are optional for `if` blocks that have only one conditional statement. But watch out for misleading indentations.
- `switch` statements can evaluate only to `enum`s or the `byte`, `short`, `int`, `char`, and, as of Java 7, `String` data types. You can't say this:

  ```
  long s = 30;
  switch(s) { }
  ```
- The `case` constant must be a literal or a compile-time constant, including an `enum` or a `String`. You cannot have a case that includes a nonfinal variable or a range of values.
- If the condition in a `switch` statement matches a `case` constant, execution will run through all code in the `switch` following the matching `case` statement until a `break` statement or the end of the `switch` statement is encountered. In other words, the matching `case` is just the entry point into the `case` block, but unless there's a `break` statement, the matching `case` is not the only `case` code that runs.
- The `default` keyword should be used in a `switch` statement if you want to run some code when none of the `case` values match the conditional value.
- The `default` block can be located anywhere in the `switch` block, so if no preceding `case` matches, the `default` block will be entered; 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 (OCA Objectives 5.1, 5.2, 5.3, and 5.4)

- A basic `for` statement has three parts: declaration and/or initialization, boolean evaluation, and the iteration expression.
- If a variable is incremented or evaluated within a basic `for` loop, it must be declared before the loop or within the `for` loop declaration.
- A variable declared (not just initialized) within the basic `for` loop declaration cannot be accessed outside the `for` loop—in other words, code below the `for` loop won't be able to use the variable.
- You can initialize more than one variable of the same type in the first part of the basic `for` loop declaration; each initialization must be comma separated.
- An enhanced `for` statement (new as of Java 5) has two parts: the *declaration* and the *expression*. It is used only to loop through arrays or collections.

- With an enhanced `for`, the *expression* is the array or collection through which you want to loop.
- With an enhanced `for`, the *declaration* is the block variable, whose type is compatible with the elements of the array or collection, and that variable contains the value of the element for the given iteration.
- Unlike with C, you cannot use a number or anything that does not evaluate to a `boolean` value as a condition for an `if` statement or looping construct. You can't, for example, say `if(x)`, unless `x` is a `boolean` variable.
- The `do` loop will **always** enter the body of the loop at least once.

Using break and continue (OCA Objective 5.5)

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

Handling Exceptions (OCA Objectives 8.1, 8.2, 8.3, 8.4, and 8.5)

- Some of the benefits of Java's exception-handling features include organized error-handling code, easy error detection, keeping exception-handling code separate from other code, and the ability to reuse exception-handling code for a range of issues.
- Exceptions come in two flavors: checked and unchecked.
- Checked exceptions include all subtypes of `Exception`, excluding classes that extend `RuntimeException`.
- Checked exceptions are subject to the handle or declare rule; any method that might throw a checked exception (including methods that invoke methods that can throw a checked exception) must either declare the exception using `throws` or handle the exception with an appropriate `try/catch`.
- Subtypes of `Error` or `RuntimeException` are unchecked, so the compiler doesn't enforce the handle or declare rule. You're free to handle them or to declare them, but the compiler doesn't care one way or the other.
- A `finally` block will always be invoked, regardless of whether an exception is thrown or caught in its `try/catch`.
- The only exception to the `finally`-will-always-be-called rule is that a `finally` will not be invoked if the JVM shuts down. That could happen if code from the `try` or `catch` blocks calls `System.exit()`.
- Just because `finally` is invoked does not mean it will complete. Code in the `finally` block could itself raise an exception or issue a `System.exit()`.
- Uncaught exceptions propagate back through the call stack, starting from the method where the exception is thrown and ending with either the first method that has a corresponding catch for that exception type or a JVM shutdown (which happens if the exception gets to `main()` and `main()` is "ducking" the exception by declaring it).
- You can almost always create your own exceptions by extending `Exception` or one of its checked exception subtypes. Such an exception will then be considered a checked exception by the compiler. (In other words, it's rare to extend `RuntimeException`.)
- All `catch` blocks must be ordered from most specific to most general. If you have a `catch` clause for both `IOException` and `Exception`, you must put the `catch` for `IOException` first in your code. Otherwise, the `IOException` would be caught by `catch (Exception e)`, because a `catch` argument can catch the specified exception or any of its subtypes!
- Some exceptions are created by programmers and some by the JVM.

SELF TEST

1. Given that `toLowerCase()` is an aptly named `String` method that returns a `String`, and given the code: ?

```
public class Flipper {
  public static void main(String[] args) {
    String o = "-";
    switch("RED".toLowerCase()) {
      case "yellow":
        o += "y";
      case "red":
        o += "r";
      case "green":
        o += "g";
    }
```

```
      System.out.println(o);
    }
  }
```

What is the result?

A. `-`

B. `-r`

C. `-rg`

D. Compilation fails

E. An exception is thrown at runtime

2. Given: ?

```
class Plane {
  static String s = "-";
  public static void main(String[] args) {
    new Plane().s1();
    System.out.println(s);
  }
  void s1() {
    try { s2(); }
    catch (Exception e) { s += "c"; }
  }
  void s2() throws Exception {
    s3(); s += "2";
    s3(); s += "2b";
  }
  void s3() throws Exception {
    throw new Exception();
  }
}
```

What is the result?

A. `-`

B. `-c`

C. `-c2`

D. `-2c`

E. `-c22b`

F. `-2c2b`

G. `-2c2bc`

H. Compilation fails

3. Given: ?

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

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

A. `ClassCastException`

B. `IllegalStateException`

C. `NumberFormatException`

D. `IllegalArgumentException`

E. `ExceptionInInitializerError`

F. `ArrayIndexOutOfBoundsException`

4. Given: ?

```
public class Flip2 {
  public static void main(String[] args) {
    String o = "-";
    String[] sa = new String[4];
```

```
        for(int i = 0; i < args.length; i++)
          sa[i] = args[i];
        for(String n: sa) {
          switch(n.toLowerCase()) {
            case "yellow": o += "y";
            case "red":    o += "r";
            case "green":  o += "g";
          }
        }
        System.out.print(o);
      }
    }
```

And given the command-line invocation:

`Java Flip2 RED Green YeLLow`

Which are true? (Choose all that apply.)

A. The string `rgy` will appear somewhere in the output

B. The string `rgg` will appear somewhere in the output

C. The string `gyr` will appear somewhere in the output

D. Compilation fails

E. An exception is thrown at runtime

5. Given: ?

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

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

A. `for(int y : x) {`

B. `for(x : int y) {`

C. `int y = 0; for(y : x) {`

D. `for(int y=0, z=0; z<x.length; z++) { y = x[z];`

E. `for(int y=0, int z=0; z<x.length; z++) { y = x[z];`

F. `int y = 0; for(int z=0; z<x.length; z++) { y = x[z];`

6. Given: ?

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

What is the result?

A. `-ic of`

B. `-mf of`

C. `-mc mf`

D. `-ic mf of`

E. `-ic mc mf of`

F. `-ic mc of mf`

G. Compilation fails

7. Given: ?

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

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

A. Compilation succeeds

B. Compilation fails due to an error on line 8

C. Compilation fails due to an error on line 10

D. Compilation fails due to an error on line 12

E. Compilation fails due to an error on line 14

8. Given: ?

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

What is the result?

A. `9 10 d`

B. `8 9 10 d`

C. `9 10 10 d`

D. `9 10 10 d 13`

E. `8 9 10 10 d 13`

F. `8 9 10 9 10 10 d 13`

G. Compilation fails

9. Given: ?

```
class Infinity { }
```

```
 4. public class Beyond extends Infinity {
 5.   static Integer i;
 6.   public static void main(String[] args) {
 7.     int sw = (int)(Math.random() * 3);
 8.     switch(sw) {
 9.       case 0: {  for(int x = 10; x > 5; x++)
10.                    if(x > 10000000) x = 10;
11.                  break; }
12.       case 1: {  int y = 7 * i; break; }
13.       case 2: {  Infinity inf = new Beyond();
14.                  Beyond b = (Beyond)inf; }
15.     }
16.   }
17. }
```

And given that line 7 will assign the value 0, 1, or 2 to `sw`, which are true? (Choose all that apply.)

A. Compilation fails

B. A `ClassCastException` might be thrown

C. A `StackOverflowError` might be thrown

D. A `NullPointerException` might be thrown

E. An `IllegalStateException` might be thrown

F. The program might hang without ever completing

G. The program will always complete without exception

10. Given: ?

```
public class Circles {
  public static void main(String[] args) {
    int[] ia = {1,3,5,7,9};
    for(int x : ia) {
      for(int j = 0; j < 3; j++) {
        if(x > 4 && x < 8) continue;
        System.out.print(" " + x);
        if(j == 1) break;
        continue;
      }
      continue;
    }
  }
}
```

What is the result?

A. `1 3 9`

B. `5 5 7 7`

C. `1 3 3 9 9`

D. `1 1 3 3 9 9`

E. `1 1 1 3 3 3 9 9 9`

F. Compilation fails

11. Given: ?

```
public class OverAndOver {
  static String s = "";
  public static void main(String[] args) {
    try {
      s += "1";
      throw new Exception();
    } catch (Exception e) { s += "2";
    } finally { s += "3"; doStuff(); s += "4";
    }
    System.out.println(s);
  }
  static void doStuff() { int x = 0; int y = 7/x; }
```

```
}
```

What is the result?

A. `12`

B. `13`

C. `123`

D. `1234`

E. Compilation fails

F. `123` followed by an exception

G. `1234` followed by an exception

H. An exception is thrown with no other output

12. Given: ?

```
public class Wind {
  public static void main(String[] args) {
    foreach:
    for(int j=0; j<5; j++) {
      for(int k=0; k< 3; k++) {
        System.out.print(" " + j);
        if(j==3 && k==1) break foreach;
        if(j==0 || j==2) break;
      }
    }
  }
}
```

What is the result?

A. `0 1 2 3`

B. `1 1 1 3 3`

C. `0 1 1 1 2 3 3`

D. `1 1 1 3 3 4 4 4`

E. `0 1 1 1 2 3 3 4 4 4`

F. Compilation fails

13. Given: ?

```
 3. public class Gotcha {
 4.   public static void main(String[] args) {
 5.     // insert code here
 6.
 7.   }
 8.   void go() {
 9.     go();
10.   }
11. }
```

And given the following three code fragments:

I. `new Gotcha().go();`

II. `try { new Gotcha().go(); }`
`catch (Error e) { System.out.println("ouch"); }`

III. `try { new Gotcha().go(); }`
`catch (Exception e) { System.out.println("ouch"); }`

When fragments I–III are added, independently, at line 5, which are true? (Choose all that apply.)

A. Some will not compile

B. They will all compile

C. All will complete normally

D. None will complete normally

E. Only one will complete normally

F. Two of them will complete normally

14. Given the code snippet: ?

```
String s = "bob";
String[] sa = {"a", "bob"};
final String s2 = "bob";
StringBuilder sb = new StringBuilder("bob");

// switch(sa[1]) {          // line 1
// switch("b" + "ob") {     // line 2
// switch(sb.toString()) {  // line 3

// case "ann":  ;           // line 4
// case s:      ;           // line 5
// case s2:     ;           // line 6
}
```

And given that the numbered lines will all be tested by uncommenting one `switch` statement and one `case` statement together, which line(s) will FAIL to compile? (Choose all that apply.)

A. line 1

B. line 2

C. line 3

D. line 4

E. line 5

F. line 6

G. All six lines of code will compile

15. Given that `IOException` is in the `java.io` package and given: ?

```
1. public class Frisbee {
2.   // insert code here
3.     int x = 0;
4.     System.out.println(7/x);
5.   }
6. }
```

And given the following four code fragments:

I. `public static void main(String[] args) {`

II. `public static void main(String[] args) throws Exception {`

III. `public static void main(String[] args) throws IOException {`

IV. `public static void main(String[] args) throws RuntimeException {`

If the four fragments are inserted independently at line 2, which are true? (Choose all that apply.)

A. All four will compile and execute without exception

B. All four will compile and execute and throw an exception

C. Some, but not all, will compile and execute without exception

D. Some, but not all, will compile and execute and throw an exception

E. When considering fragments II, III, and IV, of those that will compile, adding a `try/catch` block around line 4 will cause compilation to fail

16. Given: ?

```
class MyException extends Exception { }
class Tire {
  void doStuff() { }
}
public class Retread extends Tire {
  public static void main(String[] args) {
```

```
 8.     new Retread().doStuff();
 9.   }
10.   // insert code here
11.     System.out.println(7/0);
12.   }
13. }
```

And given the following four code fragments:

I. `void doStuff() {`

II. `void doStuff() throws MyException {`

III. `void doStuff() throws RuntimeException {`

IV. `void doStuff() throws ArithmeticException {`

When fragments I–IV are added, independently, at line 10, which are true? (Choose all that apply.)

A. None will compile

B. They will all compile

C. Some, but not all, will compile

D. All those that compile will throw an exception at runtime

E. None of those that compile will throw an exception at runtime

F. Only some of those that compile will throw an exception at runtime

Answers

1. ☑ **C** is correct. As of Java 7 it's legal to switch on a String, and remember that switches use "entry point" logic.

   ☒ **A, B, D**, and **E** are incorrect based on the above. (OCA Objective 3.4)

2. ☑ **B** is correct. Once `s3()` throws the exception to `s2()`, `s2()` throws it to `s1()`, and no more of `s2()`'s code will be executed.

   ☒ **A, C, D, E, F, G**, and **H** are incorrect based on the above. (OCA Objectives 8.2 and 8.4)

3. ☑ **C** and **D** are correct. `Integer.parseInt` can throw a `NumberFormatException`, and `IllegalArgumentException` is its superclass (that is, a broader exception).

   ☒ **A, B, E**, and **F** are not in `NumberFormatException`'s class hierarchy. (OCA Objective 8.5)

4. ☑ **E** is correct. As of Java 7 the syntax is legal. The `sa[]` array receives only three arguments from the command line, so on the last iteration through `sa[]`, a `NullPointerException` is thrown.

   ☒ **A, B, C**, and **D** are incorrect based on the above. (OCA Objectives 1.3, 5.2, and 8.5)

5. ☑ **A**, **D**, and **F** are correct. **A** is an example of the enhanced `for` loop. **D** and **F** are examples of the basic `for` loop.

   ☒ **B, C**, and **E** are incorrect. **B** is incorrect because its operands are swapped. **C** is incorrect because the enhanced `for` must declare its first operand. **E** is incorrect syntax to declare two variables in a `for` statement. (OCA Objective 5.2)

6. ☑ **E** is correct. There is no problem nesting `try/catch` blocks. As is normal, when an exception is thrown, the code in the `catch` block runs, and then the code in the `finally` block runs.

   ☒ **A, B, C, D**, and **F** are incorrect based on the above. (OCA Objectives 8.2 and 8.4)

7. ☑ **C** is correct. An overriding method cannot throw a broader exception than the method it's overriding. Class CC4's method is an overload, not an override.

   ☒ **A, B, D**, and **E** are incorrect based on the above. (OCA Objectives 8.2 and 8.4)

8. ☑ **D** is correct. Did you catch the static initializer block? Remember that switches work on "fall-through" logic and that fall-through logic also applies to the default case, which is used when no other case matches.

   ☒ **A, B, C, E, F**, and **G** are incorrect based on the above. (OCA Objective 3.4)

9. ☑ **D** and **F** are correct. Because `i` was not initialized, case 1 will throw a `NullPointerException`. Case 0 will initiate an endless loop, not a stack overflow. Case 2's downcast will *not* cause an exception.

   ☒ **A, B, C, E**, and **G** are incorrect based on the above. (OCA Objectives 3.4 and 8.5)

10. ☑ **D** is correct. The basic rule for unlabeled `continue` statements is that the current iteration stops early and execution jumps to the next iteration. The last two `continue` statements are redundant!

    ☒ **A, B, C, E**, and **F** are incorrect based on the above. (OCA Objectives 5.2 and 5.5)

11. ☑ **H** is correct. It's true that the value of `String` s is `123` at the time that the divide-by-zero exception is thrown, but `finally()` is *not* guaranteed to complete, and in this case `finally()` never completes, so the `System.out.println` (S.O.P) never executes.

    ☒ **A, B, C, D, E, F**, and **G** are incorrect based on the above. (OCA Objectives 8.2 and 8.5)

12. ☑ **C** is correct. A `break` breaks out of the current innermost loop and carries on. A labeled `break` breaks out of and terminates the labeled loops.

    ☒ **A, B, D, E**, and **F** are incorrect based on the above. (OCA Objectives 5.2 and 5.5)

13. ☑ **B** and **E** are correct. First off, `go()` is a badly designed recursive method, guaranteed to cause a `StackOverflowError`. Since `Exception` is not a superclass of `Error`, catching an `Exception` will not help handle an `Error`, so fragment III will not complete normally. Only fragment II will catch the `Error`.

    ☒ **A, C, D**, and **F** are incorrect based on the above. (OCA Objectives 8.1, 8.2, and 8.4)

14. ☑ **E** is correct. A `switch`'s `cases` must be compile-time constants or `enum` values.

    ☒ **A, B, C, D, F**, and **G** are incorrect based on the above. (OCA Objective 3.4)

15. ☑ **D** is correct. This is kind of sneaky, but remember that we're trying to toughen you up for the real exam. If you're going to throw an `IOException`, you have to import the java.io package or declare the exception with a fully qualified name.

    ☒ **A, B, C**, and **E** are incorrect. **A, B**, and **C** are incorrect based on the above. **E** is incorrect because it's okay both to handle and declare an exception. (OCA Objectives 8.2 and 8.5)

16. ☑ **C** and **D** are correct. An overriding method cannot throw checked exceptions that are broader than those thrown by the overridden method. However, an overriding method *can* throw `RuntimeExceptions` not thrown by the overridden method.

    ☒ **A, B, E**, and **F** are incorrect based on the above. (OCA Objective 8.1)