3

Assertions and Java Exceptions

CERTIFICATION OBJECTIVES

- Use try-catch and throw Statements
- Use catch, Multi-catch, and finally Clauses
- Use Autoclose Resources with a try-with-resources Statement
- Create Custom Exceptions and Auto-closeable Resources
- Test Invariants by Using Assertions

✓ Two-Minute Drill

Q&A Self Test

The assertion mechanism gives you a way to do testing and debugging checks on conditions you expect to smoke out while developing, when you don't want the runtime overhead associated with exception handling.

When you do need to use exception handling, you can take advantage of two features added to exception handling in Java 7. First, multi-`catch` gives you a way of dealing with two or more exception types at once, and second, `try`-with-resources lets you close your resources very easily.

CERTIFICATION OBJECTIVE

Working with the Assertion Mechanism (OCP Objective 6.5)

6.5 Test invariants by using assertions.

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

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

You write `print` statements with them:

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

Assertions let you test your assumptions during development, without the expense (in both your time and program overhead) of writing exception handlers, for exceptions that you assume will never happen once the program is out of development and fully deployed.

For the OCP 8 exam, you're expected to know the basics of how assertions work, including how to enable them, how to use them, and how *not* to use them.

Assertions Overview

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

```
private void methodA(int num) {
  if (num >= 0) {
    useNum(num + x);
  } else {  // num < 0 (this should never happen!)
    System.out.println("Yikes! num is a negative number! " + num);
  }
}
```

Because you're so certain of your assumption, you don't want to take the time (or program performance hit) to write exception-handling code. And at runtime, you don't want the `if/else` either because if you do reach the `else` condition, it means your earlier logic (whatever was running prior to this method being called) is flawed.

Assertions let you test your assumptions during development, but the assertion code basically evaporates when the program is deployed, leaving behind no overhead or debugging code to track down and remove. Let's rewrite `methodA()` to validate that the argument was not negative:

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

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

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

Assertions work quite simply. You always assert that something is `true`. If it is, no

problem. Code keeps running. But if your assertion turns out to be wrong (`false`), then a stop-the-world `AssertionError` is thrown (which you should never, ever handle!) right then and there, so you can fix whatever logic flaw led to the problem.

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

Really simple:

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

Simple:

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

The difference between the two is that the simple version adds a second expression separated from the first (boolean expression) by a colon—this expression's string value is added to the stack trace. Both versions throw an immediate `AssertionError`, but the simple version gives you a little more debugging help, whereas the really simple version tells you only that your assumption was false.

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

Assertion Expression Rules

Assertions can have either one or two expressions, depending on whether you're using the "simple" or the "really simple." The first expression (we'll call it expression1) must always result in a `boolean` value! Follow the same rules you use for `if` and `while` tests. The

whole point is to assert `aTest`, which means you're asserting that `aTest` is `true`. If it is `true`, no problem. If it's not `true`, however, then your assumption was wrong and you get an `AssertionError`.

The second expression (expression2), used only with the simple version of an `assert` statement, can be anything that results in a value. Remember, the second expression is used to generate a `String` message that displays in the stack trace to give you a little more debugging information. It works much like `System.out .println()` in that you can pass it a primitive or an object and it will convert it into a `String` representation. It must resolve to a value!

The following code lists legal and illegal expressions for both parts of an `assert` statement. Remember, expression2 (the value to print in the stack trace) is used only with the simple `assert` statement; the second expression exists solely to give you a little more debugging detail:

```
void noReturn() { }
int aReturn() { return 1; }
void go() {
  int x = 1;
  boolean b = true;

  // the following six are legal assert statements
  assert(x == 1);
  assert(b);
  assert true;
  assert(x == 1) : x;
  assert(x == 1) : aReturn();
  assert(x == 1) : new ValidAssert();

  // the following six are ILLEGAL assert statements
  assert(x = 1);              // none of these are booleans
  assert(x);
  assert 0;
  assert(x == 1) : ;          // none of these return a value
  assert(x == 1) : noReturn();
  assert(x == 1) : ValidAssert va;
}
```

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

the boolean test. For example, consider the following question:

Exam Question: An **`assert`** *expression must result in a* **`boolean`** *value, true or false?*

Assume that the word "expression" refers to expression1 of an assert, so the question statement is correct. If the statement were referring to expression2, however, the statement would be incorrect because expression2 can be anything that results in a value, not just a **`boolean`***.*

Using Assertions

If you want to use assertions, the first step is to put them in your code. Next, every time you run your code, you can choose whether to enable the assertions or not.

Running with Assertions

Here's where it gets cool. Once you've written your assertion-aware code, you can choose to enable or disable your assertions at runtime! The first thing to remember is that at runtime assertions are disabled by default.

Enabling Assertions at Runtime

You enable assertions at runtime with a command like this:

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

or

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

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

Disabling Assertions at Runtime

You must also know the command-line switches for disabling assertions:

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

or

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

Because assertions are disabled by default, using the disable switches might seem unnecessary. Indeed, using the switches the way we do in the preceding example just gives you the default behavior (in other words, you get the same result, regardless of whether you

use the disabling switches). But…you can also selectively enable and disable assertions in such a way that they're enabled for some classes and/or packages and disabled for others while a particular program is running.

Selective Enabling and Disabling

The command-line switches for assertions can be used in various ways:

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

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

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

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

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

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

```
com
    |_geeksanonymous
                  |_Foo.class
                  |_twelvesteps
                                |_StepOne.class
                                |_StepTwo.class
```

This tree lists three directories:

```
com
geeksanonymous
twelvesteps
and three classes:
com.geeksanonymous.Foo
com.geeksanonymous.twelvesteps.StepOne
com.geeksanonymous.twelvesteps.StepTwo
```

The subpackage of `com.geeksanonymous` is the `twelvesteps` package. Remember that in Java, the `com.geeksanonymous.twelvesteps` package is treated as a completely distinct package that has no relationship with the packages above it (in this example, the `com.geeksanonymous` package), except they just happen to share a couple of directories. Table 3-1 lists examples of command-line switches for enabling and disabling assertions.

TABLE 3-1 Assertion Command-Line Switches

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

Using Assertions Appropriately

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

So who gets to decide what's appropriate? Oracle. The exam uses Oracle's "official" assertion documentation to define appropriate and inappropriate uses.

Don't Use Assertions to Validate Arguments to a public Method

The following is an inappropriate use of assertions:

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

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

A `public` method might be called from code that you don't control (or from code you have never seen). Because `public` methods are part of your interface to the outside world, you're supposed to guarantee that any constraints on the arguments will be enforced by the method itself. But since assertions aren't guaranteed to actually run (they're typically disabled in a deployed application), the enforcement won't happen if assertions aren't enabled. You don't want publicly accessible code that works only conditionally, depending on whether assertions are enabled.

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

Do Use Assertions to Validate Arguments to a private Method

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

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

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

Don't Use Assertions to Validate Command-Line Arguments

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

Do Use Assertions, Even in public Methods, to Check for Cases That You Know Are Never, Ever Supposed to Happen

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

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

If you assume that a particular code block won't be reached, as in the preceding example where you assert that `x` must be 1, 2, or 3, then you can use `assert false` to cause an `AssertionError` to be thrown immediately if you ever do reach that code. So in the `switch` example, we're not performing a boolean test—we've already asserted that we should never be there, so just getting to that point is an automatic failure of our assertion/assumption.

Don't Use assert Expressions That Can Cause Side Effects!

The following would be a very bad idea:

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

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

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

CERTIFICATION OBJECTIVE

Working with Exception Handling (OCP Objectives 6.1, 6.2, 6.3, and 6.4)

6.1 Use try-catch, and throw statements.

6.2 Use catch, multi-catch, and finally clauses.

6.3 Use Autoclose resources with a try-with-resources statement.

6.4 Create custom exceptions and Auto-closeable resources.

You should already know the basics of `try`, `catch`, and `throw`, but if you need a refresher, head back to Chapter 5 of the *OCA Java SE 8 Programmer I Exam Guide.* For this section, we're assuming you know that material, so we're going to dive right into Java's more advanced exception-handling features.

Use the try Statement with multi-catch and finally Clauses

Sometimes we want to handle different types of exceptions the same way. Especially when all we can do is log the exception and declare defeat. But we don't want to repeat code. So what to do? When you were studying for the OCA 8 exam, you already saw that having a single catch-all exception handler is a bad idea. Prior to Java 7, the best we could do was

```
try {
  // access the database and write to a file
} catch (SQLException e) {
  handleErrorCase(e);
} catch (IOException e) {
  handleErrorCase(e);
}
```

You may be thinking that it is only one line of duplicate code. But what happens when you are catching six different exception types? That's a lot of duplication. Luckily, Java 7 made handling this sort of situation nice and easy with a feature called multi-`catch`:

```
try {
  // access the database and write to a file
} catch (SQLException | IOException e) {
  handleErrorCase(e);
}
```

No more duplication. This is great. As you might imagine, multi-`catch` is short for "multiple `catch`." You just list out the types you want the multi-`catch` to handle separated by pipe (`|`) characters. This is easy to remember because `|` is the "or" operator in Java, which means the `catch` can be read as "`SQLException` or `IOException e`."

You can't use the variable name multiple times in a ***multi****-catch. The following won't compile:*

```
catch(Exception1 e1 | Exception2 e2)
```

It makes sense that this example doesn't compile. After all, the code in the exception handler needs to know which variable name to refer to.

```
catch(Exception1 e | Exception2 e)
```

This one is tempting. When we declare variables, we normally put the variable name right after the type. Try to think of it as a list of types. We are declaring variable e to be caught and it must be one of ***Exception1*** *or* ***Exception2*** *types.*

With multi-`catch`, order doesn't matter. The following two snippets are equivalent to each other:

```
catch(SQLException | IOException e)   // these two statements are
                                      // equivalent
catch(IOException | SQLException e)
```

Just like with exception matching in a regular `catch` block, you can't just throw any two exceptions together. With multi-`catch`, you have to make sure a given exception can only match one type. The following will not compile:

```
catch(FileNotFoundException | IOException e)
catch(IOException | FileNotFoundException e)
```

You'll get a compiler error that looks something like:

```
The exception FileNotFoundException is already caught by the
alternative IOException
```

Since `FileNotFoundException` is a subclass of `IOException`, we could have just written that in the first place! There was no need to use multi-`catch`. The simplified and working version simply says:

```
catch(IOException e)
```

Remember, multi-`catch` is only for exceptions in different inheritance hierarchies. To make sure this is clear, what do you think happens with the following code?

```
catch(IOException | Exception e)
```

That's right. It won't compile because `IOException` is a subclass of `Exception`. Which means it is redundant and the compiler won't accept it.

To summarize, we use multi-`catch` when we want to reuse an exception handler. We can list as many types as we want so long as none of them have a superclass/subclass relationship with each other.

Multi-catch and catch Parameter Assignment

There is one tricky thing with multi-`catch`. And we know the exam creators like tricky things!

The following LEGAL code demonstrates assigning a new value to the single catch parameter:

```
try {
  // access the database and write to a file
} catch (IOException e) {
  e = new IOException();
}
```

Don't assign a new value to the *catch* parameter. It is not good practice and creates confusing, hard-to-maintain code. But it is legal Java code to assign a new value to the *catch* block's parameter when there is only one type listed, and it will compile.

The following ILLEGAL code demonstrates trying to assign a value to the final multi-`catch` parameter:

```
try {
  // access the database and write to a file
} catch (SQLException | IOException e) {
  e = new IOException();
}
```

At least you get a clear compiler error if you try to do this. The compiler tells you:

```
The parameter e of a multi-catch block cannot be assigned
```

Since multi-`catch` uses multiple types, there isn't a clearly defined type for the variable that you can set. Java solves this by making the catch parameter `final` when that happens. And then the code doesn't compile because you can't assign a new value to a `final` variable.

Rethrowing Exceptions

Sometimes we want to do something with the thrown exceptions before we rethrow them:

```
public void couldThrowAnException() throws IOException, SQLException {}

public void rethrow() throws SQLException, IOException {
  try {
    couldThrowAnException();
  } catch (SQLException | IOException e) {
    log(e);
    throw e;
  }
}
```

This is a common pattern called "handle and declare." We want to do something with the exception—log it. We also want to acknowledge we couldn't completely handle it, so we declare it and let the caller deal with it. (As an aside, many programmers believe that logging an exception and rethrowing it is a bad practice, but you never know—you might see this kind of code on the exam.)

You may have noticed that `couldThrowAnException()` doesn't actually throw an exception. The compiler doesn't know this. The method signature is key to the compiler. It can't assume that no exception gets thrown, as a subclass could override the method and throw an exception.

There is a bit of duplicate code here. We have the list of exception types thrown by the methods we call typed twice. Multi-`catch` was introduced to avoid having duplicate code, yet here we are with duplicate code.

Lucky for us, Java helps us out here as well with a feature added in Java 7. This example is a nicer way of writing the previous code:

```
1. public void rethrow() throws SQLException, IOException {
2.   try {
3.     couldThrowAnException();
4.   } catch (Exception e) {    // watch out: this isn't really
5.                              // catching all exception subclasses
6.     log(e);
7.     throw e;                 // note: won't compile in Java 6
8.   }
9. }
```

Notice the multi-catch is gone and replaced with catch(Exception e). It's not bad practice here, though, because we aren't really catching all exceptions. The compiler is treating `Exception` as "any exceptions that the called methods happen to throw." (You'll see this idea of code shorthand again with the diamond operator when you get to generics.)

This is very different from Java 6 code that catches `Exception`. In Java 6, we'd need the `rethrow()` method signature to be `throws Exception` in order to make this code compile.

In Java 7 and later, `} catch (Exception e) {` doesn't really catch ANY `Exception` subclass. The code may say that, but the compiler is translating for you. The compiler says, "Well, I know it can't be just any exception because the throws clause won't let me. I'll pretend the developer meant to only catch `SQLException` and `IOException`. After all, if any others show up, I'll just fail compilation on `throw e;`—just like I used to in Java 6." Tricky, isn't it?

At the risk of being too repetitive, remember that `catch (Exception e)` doesn't necessarily catch all `Exception` subclasses. In Java 7 and later, it means catch all `Exception` subclasses that would allow the method to compile.

Got that? Now why on earth would Oracle do this to us? It sounds more complicated than it used to be! Turns out they were trying to solve another problem at the same time they were changing this stuff. Suppose the API developer of `couldThrowAnException()` decided the method will never throw a `SQLException` and removes `SQLException` from the signature to reflect that.

Imagine we were using the Java 6 style of having one `catch` block per exception or even the multi-`catch` style of

```
} catch (SQLException | IOException e) {
```

Our code would stop compiling with an error like:

```
Unreachable catch block for SQLException
```

It is reasonable for code to stop compiling if we add exceptions to a method. But we don't want our code to break if a method's implementation gets LESS brittle. And that's the advantage of using

```
} catch (Exception e) {
```

Java infers what we mean here and doesn't say a peep when the API we are calling removes an exception.

Don't go changing your API signatures on a whim. Most code was written before Java 7 and will break if you change signatures. Your callers won't thank you when their code suddenly fails compilation because they tried to use your new, shiny, "cleaner" API.

You've probably noticed by now that Oracle values backward compatibility and doesn't change the behavior or "compiler worthiness" of code from older versions of Java. That still stands. In Java 6, we can't write `catch (Exception e)` and merely throw specific exceptions. If we tried, the compiler would still complain:

```
Unhandled exception type Exception.
```

Backward compatibility only needs to work for code that compiles! It's OK for the compiler to get less strict over time.

To make sure you understand what is going on here, think about what happens in this example:

```
public class A extends Exception{}
public class B extends Exception{}
public void rain() throws A, B {}
```

Table 3-2 summarizes handling changes to the exception-related parts of method signatures in Java 6, Java 7, and Java 8.

TABLE 3-2 Exceptions and Signatures

	What happens if rain() adds a new checked exception?	What happens if rain() removes a checked exception from the signature?

Java 6 style:
 	
`public void ahh() throws A, B {`
`  try {`
`  rain();`
`  } catch (A e) {`
`  throw e;`
`  } catch (B e) {`
`  throw e;`
`  }`
`}`
	Add another `catch` block to handle the new exception.	Remove a `catch` block to avoid compiler error about unreachable code.

Java 7 and 8, with duplication:
 	
`public void ahh() throws A, B {`
`  try {`
`  rain();`
`  } catch (A | B e) {`
`  throw e;`
`  }`
`}`
	Add another exception to the multi-`catch` block to handle the new exception.	Remove an expression from the multi-`catch` block to avoid compiler error about unreachable code.

Java 7 and 8, without duplication:
 	
`public void ahh() throws A, B {`
`  try {`
`  rain();`
`  } catch (Exception e) {`
`  throw e;`
`  }`
`}`
	Add another exception to the method signature to handle the new exception that can be thrown.	No code changes needed.

There is one more trick. If you assign a value to the `catch` parameter, the code no longer compiles:

```
public void rethrow() throws SQLException, IOException {
  try {
     couldThrowAnException();
  } catch (Exception e) {
    e = new IOException();
    throw e;
  }
}
```

As with multi-`catch`, you shouldn't be assigning a new value to the `catch` parameter in real life anyway. The difference between this and multi-`catch` is where the compiler error occurs. For multi-`catch`, the compiler error occurs on the line where we attempt to assign a new value to the parameter, whereas here, the compiler error occurs on the line where we `throw e`. It is different because code written prior to Java 7 still needs to compile. Because the multi-`catch` syntax is still relatively new, there is no legacy code to worry about.

AutoCloseable Resources with a try-with-resources Statement

The `finally` block is a good place for closing files and assorted other resources, but real-world clean-up code is easy to get wrong. And when correct, it is verbose. Let's look at the code to close our one resource when closing a file:

```
1: Reader reader = null;
2: try {
3:    // read from file
```

```
 4: } catch(IOException e) {
 5:   log(); throw e;
 6: } finally {
 7:   if (reader != null) {
 8:     try {
 9:       reader.close();
10:     } catch (IOException e) {
11:     // ignore exceptions on closing file
12:     }
13:   }
14: }
```

That's a lot of code just to close a single file! But it's all necessary. First, we need to check if the reader is null on line 7. It is possible the `try` block threw an exception before creating the reader, or while trying to create the reader if the file we are trying to read doesn't exist. It isn't until line 9 that we get to the one line in the whole `finally` block that does what we care about—closing the file. Lines 8 and 10 show a bit more housekeeping. We can get an `IOException` on attempting to close the file. While we could try to handle that exception, there isn't much we can do, thus making it common to just ignore the exception. This gives us nine lines of code (lines 6–14) just to close a file.

Developers typically write a helper class to close resources, or they use the open-source Apache Commons helper to get this mess down to three lines:

```
6:   } finally {
7:      HelperClass.close(reader);
8:   }
```

Which is still three lines too many.

Lucky for us, we have *Automatic Resource Management* using "`try`-with-resources" to get rid of even these three lines. The following code is equivalent to the previous example:

```
1:  try (Reader reader =
2:    new BufferedReader(new FileReader(file))) {   // note the new syntax
3:    // read from file
4:  } catch (IOException e) { log(); throw e;}
```

No `finally` left at all! We don't even mention closing the reader. Automatic Resource Management takes care of it for us. Let's take a look at what happens here. We start out by declaring the reader inside the try declaration. Think of the parentheses as a for loop in which we declare a loop index variable that is scoped to just the loop. Here, the reader is scoped to just the try block. Not the catch block, just the try block.

The actual `try` block does the same thing as before. It reads from the file. Or, at least, it comments that it would read from the file. The `catch` block also does the same thing as before. And just like in our traditional `try` statement, `catch` is optional.

Remember that a `try` must have `catch` or `finally`. Time to learn something new about that rule.

This is ILLEGAL code because it demonstrates a `try` without a `catch` or `finally`:

```
1: try {
2:    // do stuff
3: } // need a catch or finally here
```

The following LEGAL code demonstrates a `try`-with-resources with no `catch` or `finally`:

```
1:  try (Reader reader =
2:    new BufferedReader(new FileReader(file))) {
3:    // do stuff
4:  }
```

What's the difference? The legal example does have a `finally` block; you just don't see it. The try-with-resources statement is logically calling a finally block to close the reader. And just to make this even trickier, you can add your own `finally` block to `try`-with-resources as well. Both will get called. We'll take a look at how this works shortly.

Since the syntax is inspired from the `for` loop, we get to use a semicolon when declaring multiple resources in the `try`. For example:

```
try (MyResource mr = MyResource.createResource();   // first resource
     MyThingy mt = mr.createThingy()) {              // second resource
  // do stuff
}
```

There is something new here. Our declaration calls methods. Remember that the `try`-with-resources is just Java code. It is restricted to only declarations. This means if you want to do anything more than one statement long, you'll need to put it into a method.

To review, Table 3-3 lists the big differences that are new for `try`-with-resources.

TABLE 3-3 Comparing Traditional `try` Statement to `try`-with-resources

	try-catch-finally	try-with-resources
Resource declared	Before `try` keyword	In parentheses within `try` declaration
Resource initialized	In `try` block	In parentheses within `try` declaration
Resource closed	In `finally` block	Nowhere—happens automatically
Required keywords	`try` One of `catch` or `finally`	`try`

AutoCloseable and Closeable

Because Java is a statically typed language, it doesn't let you declare just any type in a `try`-with-resources statement. The following code will not compile:

```
try (String s = "hi") {}
```

You'll get a compiler error that looks something like:

```
The resource type String does not implement
java.lang.AutoCloseable
```

`AutoCloseable` only has one method to implement. Let's take a look at the simplest code we can write using this interface:

```
public class MyResource implements AutoCloseable {
  public void close() {
    // take care of closing the resource
  }
}
```

There's also an interface called `java.io.Closeable`, which is similar to `AutoCloseable` but with some key differences. Why are there two similar interfaces, you may wonder? The `Closeable` interface was introduced in Java 5. When `try`-with-resources was invented in Java 7, the language designers wanted to change some things but needed backward compatibility with all existing code. So they created a superinterface with the rules they wanted.

One thing the language designers wanted to do was make the signature more generic. `Closeable` allows implementors to throw only an `IOException` or a `RuntimeException`. `AutoCloseable` allows any `Exception` at all to be thrown. Look at some examples:

```
// ok because AutoCloseable allows throwing any Exception
class A implements AutoCloseable { public void close() throws Exception{}}

// ok because subclasses or implementing methods can throw
// a subclass of Exception or none at all
class B implements AutoCloseable { public void close() {}}
class C implements AutoCloseable { public void close() throws IOException {}}

// ILLEGAL - Closeable only allows IOExceptions or subclasses
class D implements Closeable { public void close() throws Exception{}}

// ok because Closeable allows throwing IOException
class E implements Closeable { public void close() throws IOException{}}
```

In your code, Oracle recommends throwing the narrowest `Exception` subclass that will compile. However, they do limit `Closeable` to `IOException`, and you must use

`AutoCloseable` for anything more.

The next difference is even trickier. What happens if we call the `close()` multiple times? It depends. For classes that implement `Closeable`, the implementation is required to be *idempotent*—which means you can call `close()` over and over again and nothing will happen the second time and beyond. It will not attempt to close the resource again and it will not blow up. For classes that implement `AutoCloseable`, there is no such guarantee.

If you look at the JavaDoc, you'll notice many classes implement both `AutoCloseable` and `Closeable`. These classes use the stricter signature rules and are idempotent. They still need to implement `Closeable` for backward compatibility, but added `AutoCloseable` for the new contract.

To review, Table 3-4 shows the differences between `AutoCloseable` and `Closeable`. Remember the exam creators like to ask about "similar but not quite the same" things!

TABLE 3-4 Comparing `AutoCloseable` and `Closeable`

	AutoCloseable	Closeable
Extends	None	AutoCloseable
close method throws	Exception	IOException
Must be idempotent (can call more than once without side effects)	No, but encouraged	Yes

A Complex try-with-resources Example The following example is as complicated as `try`-with-resources gets:

```
1: class One implements AutoCloseable {
2:   public void close() {
3:     System.out.println("Close - One");
4: } }
5: class Two implements AutoCloseable {
6:   public void close() {
7:     System.out.println("Close - Two");
8: } }
```

```
 9: class TryWithResources {
10:   public static void main(String[] args) {
11:     try (One one = new One(); Two two = new Two()) {
12:       System.out.println("Try");
13:       throw new RuntimeException();
14:     } catch (Exception e) {
15:       System.out.println("Catch");
16:     } finally {
17:       System.out.println("Finally");
18: } } }
```

Running the preceding code will print:

```
Try
Close - Two
Close - One
Catch
Finally
```

It's actually more logical than it looks at first glance. We first enter the `try` block on line 11, and Java creates our two resources. Line 12 prints `Try`. When we throw an exception on line 13, the first interesting thing happens. The `try` block "ends," and Automatic Resource Management automatically cleans up the resources before moving on to the `catch` or `finally`. The resources get cleaned up, "backward" printing `Close -`

`Two` and then `Close - One`. The `close()` method gets called in the reverse order in which resources are declared to allow for the fact that resources might depend on each other. Then we are back to the regular `try` block order, printing `Catch` and `Finally` on lines 15 and 17.

If you only remember two things from this example, remember that `try`-with-resources is part of the `try` block, and resources are cleaned up in the reverse order in which they were created.

Suppressed Exceptions

We're almost done with exceptions. There's only one more wrinkle to cover in exception handling. Now that we have an extra step of closing resources in the `try`, it is possible for multiple exceptions to get thrown. Each `close()` method can throw an exception in addition to the `try` block itself.

```
1: public class Suppressed {
2:   public static void main(String[] args) {
3:     try (One one = new One()) {
4:       throw new Exception("Try");
5:     } catch (Exception e) {
6:       System.err.println(e.getMessage());
7:       for (Throwable t : e.getSuppressed()) {
8.         System.err.println("suppressed:" + t);
9. } } } }

class One implements AutoCloseable {
  public void close() throws IOException {
    throw new IOException("Closing");
} }
```

We know that after the exception in the `try` block gets thrown on line 4, the `try`-with-resources still calls `close()` on line 3 and the `catch` block on line 5 catches one of the exceptions. Running the code prints:

```
Try
suppressed:java.io.IOException: Closing
```

This tells us the exception we thought we were throwing still gets treated as most important. Java also adds any exceptions thrown by the `close()` methods to a suppressed array in that main exception. The `catch` block or caller can deal with any or all of these. If we remove line 4, the code just prints `Closing`.

In other words, the exception thrown in `close()` doesn't always get suppressed. It becomes the main exception if there isn't already one existing. As one more example, think about what the following prints:

```
class Bad implements AutoCloseable {
  String name;
  Bad(String n) { name = n; }
  public void close() throws IOException {
    throw new IOException("Closing - "  + name);
} }

public class Suppressed {
  public static void main(String[] args) {
    try (Bad b1 = new Bad("1"); Bad b2 = new Bad("2")) {
      // do stuff
    } catch (Exception e) {
      System.err.println(e.getMessage());
      for (Throwable t : e.getSuppressed()) {
        System.err.println("suppressed:" + t);
} } } }
```

The answer is:

```
Closing - 2
```

```
suppressed:java.io.IOException: Closing - 1
```

Until `try`-with-resources calls `close()`, everything is going just dandy. When Automatic Resource Management calls `b2.close()`, we get our first exception. This becomes the main exception. Then, Automatic Resource Management calls `b1.close()` and throws another exception. Since there was already an exception thrown, this second exception gets added as a second exception.

If the `catch` or `finally` block throws an exception, no suppressions happen. The last exception thrown gets sent to the caller rather than the one from the `try`—just like before `try`-with-resources was created.

CERTIFICATION SUMMARY

Assertions are a useful debugging tool. You learned how you can use them for testing by enabling them but keep them disabled when the application is deployed.

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

Exception handling was enhanced in Java version 7, making exceptions easier to use. First you learned that you can specify multiple exception types to share a `catch` block using the new multi-`catch` syntax. The major benefit is in reducing code duplication by having multiple exception types share the same exception handler. The variable name is listed only once, even though multiple types are listed. You can't assign a new exception to that variable in the `catch` block. Then you saw the "handle and declare" pattern where the exception types in the multi-`catch` are listed in the method signature and Java translates "`catch Exception e`" into that exception type list.

Next, you learned about the `try`-with-resources syntax where Java will take care of calling `close()` for you. The objects are scoped to the `try` block. Java treats them as a `finally` block and closes these resources for you in the opposite order to which they were opened. If you have your own `finally` block, it is executed after `try`-with-resources closes the objects. You also learned the difference between `AutoCloseable` and `Closeable`. `Closeable` was introduced in Java 5, allowing only `IOException` (and `RuntimeException`) to be thrown. `AutoCloseable` was added in Java 7, allowing any type of `Exception`.

TWO-MINUTE DRILL

Here are some of the key points from the certification objectives in this chapter.

Test Invariants Using Assertions (OCP Objective 6.5)

- Assertions give you a way to test your assumptions during development and debugging.
- Assertions are typically enabled during testing but disabled during deployment.
- Assertions are disabled at runtime by default. To enable them, use a command-line flag: `-ea` or `-enableassertions`.
- Selectively disable assertions by using the `-da` or `-disableassertions` flag.
- If you enable or disable assertions using the flag without any arguments, you're enabling or disabling assertions in general. You can combine enabling and disabling switches to have assertions enabled for some classes and/or packages, but not others.
- You can enable and disable assertions on a class-by-class basis, using the following syntax:
  `java -ea-da:MyClass TestClass`
- You can enable and disable assertions on a package-by-package basis, and any package you specify also includes any subpackages (packages further down the directory hierarchy).
- Do not use assertions to validate arguments to `public` methods.
- Do not use `assert` expressions that cause side effects. Assertions aren't guaranteed to always run, and you don't want behavior that changes depending on whether assertions are enabled.
- Do use assertions—even in `public` methods—to validate that a particular code block will never be reached. You can use `assert false;` for code that should never be reached so that an assertion error is thrown immediately if the `assert` statement is executed.

Use the try Statement with Multi-catch and finally Clauses (OCP Objective 6.2)

- If two `catch` blocks have the same exception handler code, you can merge them with multi-`catch` using `catch (Exception1 | Exception2 e)`.
- The types in a multi-`catch` list must not extend one another.
- When using multi-`catch`, the `catch` block parameter is final and cannot have a new value assigned in the `catch` block.
- If you catch a general exception as shorthand for specific subclass exceptions and

rethrow the caught exception, you can still list the specific subclasses in the method signature. The compiler will treat it as if you had listed them out in the catch.

AutoCloseable Resources with a try-with-resources Statement (OCP Objectives 6.3 and 6.4)

- `try`-with-resources automatically calls `close()` on any resources declared in the `try` as `try(Resource r = new Foo())`.
- A `try` must have at least a `catch` or `finally` unless it is a `try`-with-resources. For `try`-with-resources, it can have neither, one, or both of the keywords.
- `AutoCloseable`'s `close()` method throws `Exception` and may be but is not required to be idempotent. `Closeable`'s `close()` throws `IOException` and must be idempotent.
- `try`-with-resources are closed in reverse order of creation and before going on to `catch` or `finally`.
- If more than one exception is thrown in a `try`-with-resources block, it gets added as a suppressed exception.
- The type used in a `try`-with-resources statement must implement `AutoCloseable`.

SELF TEST

The following questions will help you measure your understanding of the material presented in this chapter. Read all of the choices carefully, as there may be more than one correct answer. Choose all correct answers for each question. Stay focused.

1. Which are true? (Choose all that apply.)
   A. It is appropriate to use assertions to validate arguments to methods marked `public`
   B. It is appropriate to catch and handle assertion errors
   C. It is NOT appropriate to use assertions to validate command-line arguments
   D. It is appropriate to use assertions to generate alerts when you reach code that should not be reachable
   E. It is NOT appropriate for assertions to change a program's state
2. Given:

```
 3. public class Clumsy {
 4.   public static void main(String[] args) {
 5.     int j = 7;
 6.     assert(++j > 7);
 7.     assert(++j > 8): "hi";
 8.     assert(j > 10): j=12;
 9.     assert(j==12): doStuff();
10.     assert(j==12): new Clumsy();
11.   }
12.   static void doStuff() { }
13. }
```

Which are true? (Choose all that apply.)

A. Compilation succeeds

B. Compilation fails due to an error on line 6

C. Compilation fails due to an error on line 7

D. Compilation fails due to an error on line 8

E. Compilation fails due to an error on line 9

F. Compilation fails due to an error on line 10

3. Given:

```
class AllGoesWrong {
  public static void main(String[] args) {
    AllGoesWrong a = new AllGoesWrong();
    try {
      a.blowUp();
      System.out.print("a");
```

```
    } catch (IOException e | SQLException e) {
      System.out.print("c");
    } finally {
      System.out.print("d");
    }
  }
  void blowUp() throws IOException, SQLException {
    throw new SQLException();
  }
}
```

What is the result?

A. `ad`

B. `acd`

C. `cd`

D. `d`

E. Compilation fails

F. An exception is thrown at runtime

4. Given:

```
class BadIO {
  public static void main(String[] args) {
    BadIO a = new BadIO();
    try {
      a.fileBlowUp();
      a.databaseBlowUp();
      System.out.println("a");
    } // insert code here
      System.out.println("b");
    } catch (Exception e) {
      System.out.println("c");
  } }
  void databaseBlowUp() throws SQLException {
    throw new SQLException();
  }
  void fileBlowUp() throws IOException {
    throw new IOException();
} }
```

Which, inserted independently at `// insert code here`, will compile and produce the output `b`? (Choose all that apply.)

A. `catch(Exception e) {`

B. `catch(FileNotFoundException e) {`

C. `catch(IOException e) {`

D. `catch(IOException | SQLException e) {`

E. `catch(IOException e | SQLException e) {`

F. `catch(SQLException e) {`

G. `catch(SQLException | IOException e) {`

H. `catch(SQLException e | IOException e) {`

5. Given:

```
class Train {
  class RanOutOfTrack extends Exception { }
  class AnotherTrainComing extends Exception { }

  public static void main(String[] args) throws RanOutOfTrack,
      AnotherTrainComing {
    Train a = new Train();
    try {
      a.drive();
      System.out.println("toot! toot!");
    } // insert code here
      System.out.println("error locomoting");
      throw e;
    }
  }
  void drive() throws RanOutOfTrack, AnotherTrainComing {
    throw new RanOutOfTrack();
} }
```

Which, inserted independently at `// insert code here`, will compile and produce the output `error driving` before throwing an exception? (Choose all that apply.)

A. `catch(AnotherTrainComing e) {`

B. `catch(AnotherTrainComing | RanOutOfTrack e) {`

C. `catch(AnotherTrainComing e | RanOutOfTrack e) {`

D. `catch(Exception e) {`

E. `catch(IllegalArgumentException e) {`

F. `catch(RanOutOfTrack e) {`

G. None of the above—code fails to compile for another reason

6. Given:

```
class Conductor {
  static String s = "-";
  class Whistle implements AutoCloseable {
    public void toot() {     s += "t";   }
    public void close() {    s += "c";   }
  }
  public static void main(String[] args) {
    new Conductor().run();
    System.out.println(s);
  }
  public void run() {
    try (Whistle w = new Whistle()) {
       w.toot();
       s += "1";
       throw new Exception();
    } catch (Exception e) { s += "2";
    } finally { s += "3"; }
} }
```

What is the result?

A. `-t123t`

B. `-t12c3`

C. `-t123`

D. `-t1c3`

E. `-t1c23`

F. None of the above; `main()` throws an exception

G. Compilation fails

7. Given:

```
public class MultipleResources {
  class Lamb implements AutoCloseable {
    public void close() throws Exception {
       System.out.print("l");
  } }
  class Goat implements AutoCloseable {
    public void close() throws Exception {
       System.out.print("g");
  } }
  public static void main(String[] args) throws Exception {
    new MultipleResources().run();
  }
  public void run() throws Exception {
    try (Lamb l = new Lamb();
         System.out.print("t");
         Goat g = new Goat();) {
      System.out.print("2");
    } finally {
      System.out.print("f");
} } }
```

What is the result?

A. `2glf`

B. `2lgf`

C. `tglf`

D. `t2lgf`

E. `t2lgf`

F. None of the above; `main()` throws an exception

G. Compilation fails

8. Given:

```
1: public class Animals {
2:   class Lamb {
3:     public void close() throws Exception { }
4:   }
5:   public static void main(String[] args) throws Exception {
6:     new Animals().run();
7:   }
8:
9:   public void run() throws Exception {
10:    try (Lamb l = new Lamb();) {
11:    }
12:  }
13: }
```

And the following possible changes:

C1. Replace line 2 with class `Lamb` implements `AutoCloseable {`

C2. Replace line 2 with class `Lamb` implements `Closeable {`

C3. Replace line 11 with `} finally {}`

What change(s) allow the code to compile? (Choose all that apply.)

A. Just C1 is sufficient

B. Just C2 is sufficient

C. Just C3 is sufficient

D. Both C1 and C3 are required

E. Both C2 and C3 are required

F. The code compiles without any changes

9. Given:

```
public class Animals {
  class Lamb implements Closeable {
    public void close() {
      throw new RuntimeException("a");
    } }
  public static void main(String[] args)  {
    new Animals().run();
  }
  public void run()  {
    try (Lamb l = new Lamb();) {
      throw new IOException();
    } catch(Exception e) {
      throw new RuntimeException("c");
} } }
```

Which exceptions will the code throw?

A. `IOException` with suppressed `RuntimeException a`

B. `IOException` with suppressed `RuntimeException c`

C. `RuntimeException a` with no suppressed exception

D. `RuntimeException c` with no suppressed exception

E. `RuntimeException a` with suppressed `RuntimeException c`

F. `RuntimeException c` with suppressed `RuntimeException a`

G. Compilation fails

10. Given:

```
public class Animals {
  class Lamb implements AutoCloseable {
    public void close() {
      throw new RuntimeException("a");
  } }
  public static void main(String[] args) throws IOException {
    new Animals().run();
  }
  public void run() throws IOException {
    try (Lamb l = new Lamb();) {
      throw new IOException();
    } catch(Exception e) {
      throw e;
} } }
```

Which exceptions will the code throw?

A. `IOException` with suppressed `RuntimeException a`

B. `IOException` with suppressed `Exception e`

C. `RuntimeException a` with no suppressed exception

D. `Exception e` with no suppressed exception

E. `RuntimeException a` with suppressed `Exception e`

F. `RuntimeException c` with suppressed `RuntimeException a`

G. Compilation fails

11. Given:

```
public class Concert {
  static class PowerOutage extends Exception {}
  static class Thunderstorm extends Exception {}
  public static void main(String[] args) {
    try {
    new Concert().listen();
    System.out.print("a");
    } catch(PowerOutage | Thunderstorm e) {
     e = new PowerOutage();
     System.out.print("b");
    } finally { System.out.print("c"); }
  }
  public void listen() throws PowerOutage, Thunderstorm{ }
}
```

What will this code print?

A. `a`

B. `ab`

C. `ac`

D. `abc`

E. `bc`

F. Compilation fails

A SELF TEST ANSWERS

1. ☑ C, D, and E are correct statements.

   ☒ A is incorrect. It is acceptable to use assertions to test the arguments of `private` methods. B is incorrect. While assertion errors can be caught, Oracle discourages you from doing so. (OCP Objective 6.5)

2. ☑ E is correct. When an `assert` statement has two expressions, the second

expression must return a value. The only two-expression `assert` statement that doesn't return a value is on line 9.

☒ A, B, C, D, and F are incorrect based on the above. (OCP Objective 6.5)

3. ☑ E is correct. `catch (IOException e | SQLException e)` doesn't compile. While multiple exception types can be specified in the multi-`catch`, only one variable name is allowed. The correct syntax is `catch (IOException | SQLException e)`. Other than this, the code is valid. Note that it is legal for `blowUp()` to have `IOException` in its signature even though that `Exception` can't be thrown.

   ☒ A, B, C, D, and F are incorrect based on the above. If the `catch` block's syntax error were corrected, the code would output `cd`. The multi-`catch` would catch the `SQLException` from `blowUp()` since it is one of the exception types listed. And, of course, the `finally` block runs at the end of the `try/catch`. (OCP Objective 6.2)

4. ☑ C, D, and G are correct. Since order doesn't matter, both D and G show correct use of the multi-`catch` block. And C catches the `IOException` from `fileBlowUp()` directly. Note that `databaseBlowUp()` is never called at runtime. However, if you remove the call, the compiler won't let you catch the `SQLException` since it would be impossible to be thrown.

   ☒ A, B, E, H, and F are incorrect. A is incorrect because it will not compile. Since there is already a `catch` block for `Exception`, adding another will make the compiler think there is unreachable code. B is incorrect because it will print `c` rather than `b`. Since `FileNotFoundException` is a subclass of `IOException`, the thrown `IOException` will not match the `catch` block for `FileNotFoundException`. E and H are incorrect because they are invalid syntax for multi-`catch`. The catch parameter `e` can only appear once. F is incorrect because it will print `c` rather than `b`. Since the `IOException` thrown by `fileBlowUp()` is never caught, the thrown exception will match the `catch` block for `Exception`. (OCP Objective 6.2)

5. ☑ B, D, and F are correct. B uses multi-`catch` to identify both exceptions `drive()` may throw. D still compiles since it uses the new enhanced exception typing to recognize that `Exception` may only refer to `AnotherTrainComing` and `RanOutOfTrack`. F is the simple case that catches a single exception. Since `main` declares that it can throw `AnotherTrainComing`, the `catch` block doesn't need to handle it.

   ☒ A, C, E, and G are incorrect. A and E are incorrect because the `catch` block will not handle `RanOutOfTrack` when `drive()` throws it. The `main` method will still throw the exception, but the `println()` will not run. C is incorrect because it is invalid syntax for multi-`catch`. The `catch` parameter `e` can only appear once. G is incorrect because of the above. (OCP Objective 6.2)

6. ☑ E is correct. After the exception is thrown, Automatic Resource Management calls `close()` before completing the `try` block. From that point, `catch` and `finally`

execute in the normal order.

☒ F is incorrect because the `catch` block catches the exception and does not rethrow it.

A, B, C, D, and G are incorrect because of the above. (OCP Objective 6.3)

7. ☑ G is correct. `System.out.println` cannot be in the declaration clause of a `try`-with-resources block because it does not declare a variable. If the `println` was removed, the answer would be A because resources are closed in the opposite order in which they are created.

   ☒ A, B, C, D, E, and F are incorrect because of the above. (OCP Objective 6.3)

8. ☑ A is correct. If the code is left with no changes, it will not compile because `try`-with-resources requires `Lamb` to implement `AutoCloseable` or a subinterface. If C2 is implemented, the code will not compile because `close()` throws `Exception` instead of `IOException`. Unlike the traditional `try`, `try`-with-resources does not require `catch` or `finally` to be present.

   ☒ B, C, D, E, and F are incorrect because of the above. (OCP Objective 6.3)

9. ☑ D is correct. While the exception caught by the `catch` block matches choice A, it is ignored by the `catch` block. The `catch` block just throws `RuntimeException c` without any suppressed exceptions.

   ☒ A, B, C, E, F, and G are incorrect because of the above. (OCP Objective 6.3)

10. ☑ A is correct. After the `try` block throws an `IOException`, Automatic Resource Management calls `close()` to clean up the resources. Since an exception was already thrown in the `try` block, `RuntimeException a` gets added to it as a suppressed exception. The `catch` block merely rethrows the caught exception. The code does compile, even though the `catch` block catches an `Exception` and the method merely throws an `IOException`. In Java 7, the compiler is able to pick up on this.

    ☒ B, C, D, E, F, and G are incorrect because of the above. (OCP Objective 6.3)

11. ☑ F is correct. The exception variable in a `catch` block may not be reassigned when using multi-`catch`. It CAN be reassigned if we are only catching one exception.

    ☒ C would have been correct if `e = new PowerOutage();` were removed. A, B, D, and E are incorrect because of the above. (OCP Objectives 6.2 and 6.4)