6 Exceptions and assertions

Exam objectives covered in this chapter	What you need to know
[6.1] Use the `throw` statement and `throws` clause	How to define methods that throw exceptions. The difference between the `throw` statement and the `throws` clause. The different combinations of defining overriding methods, when the overridden or overriding methods throw checked exceptions.
[6.2] Use the `try` statement with multi-`catch` and `finally` clauses	How to use this new language feature with Java 7. How to prevent coding the same set of statements for multiple exception handlers. How to catch multiple and unrelated exceptions in a single `catch` block.
[6.3] Auto-close resources with a `try`-with-resources statement	How to use a `try`-with-resources statement so you never forget to close resources such as file handlers and URL connections. When a `try`-with-resources statement throws exceptions, how to handle them. How to define a `try`–with-resources statement without a `catch` or `finally` block.
[6.4] Create custom exceptions	How to create your own exceptions, including subclasses. How to use them in your code.
[6.5] Test invariants by using assertions	The need for and purpose of assertions. The correct syntax of an `assert` statement. Correct and incorrect usage of assertions for variants and flow control. How to enable and disable assertions.

First let's talk about exception handling. Imagine that the pilot of an airplane has an accident just an hour before the flight's scheduled takeoff. The airline manages the situation by arranging for an alternate pilot, enabling the flight to take off at the scheduled time. What a relief for both the passengers and the airline. This example illustrates how an exceptional condition (a pilot's accident) that was outside the immediate control of the airline can modify the initial flow of an action (the predefined airline flight operation). It demonstrates the need to handle such unexpected conditions appropriately (the arrangement of an alternate pilot). Can you imagine the inconvenience that would be caused to all the passengers if the airline had no alternate plan to arrange for another pilot and recover from this situation?

Similarly, it's important for a Java application to handle unexpected or exceptional conditions so it doesn't fail, exit abruptly, or return invalid values. Java's *exception handling* is a mechanism that lets you do just that. It enables you to build robust applications that can recover from exceptional conditions that may arise during the course of a program, by defining an alternate flow of actions.

Now let's talk about assertions. Whenever you board an international flight, the airline staff checks your visa and stamps your passport. Imagine what would happen if they missed stamping your passport at emigration? Would you be in trouble, at immigration, when you fly back to your own country? As you can see, a failed assumption that all passports were stamped at emigration can cause issues with immigration. For such cases, the airline would examine the existing procedures and formulate new checks so that a similar situation doesn't occur again.

Just as a failed assumption can help an airline identify an existing flaw in its operations, a failed assumption in an application can help identify and fix errors in the application logic during its development. It's difficult to find subtle bugs in the implementation of *application logic* when a programmer uses predefined assumptions (believe me—it happens a lot). You must use assertions to check assumptions in the program logic, variable values, and pre- and postconditions of methods, to avoid bugs or errors. An assertion offers a way of indicating what should always be true.

Exception handling helps you gracefully handle an exceptional condition by defining an alternate flow of action. On the other hand, assertions help you identify and fix potential application logic errors.

Exception handling is covered in both the OCA Java SE 7 Programmer I exam (1Z0-803) and OCP Java SE 7 Programmer II exam (1Z0-804). This chapter covers exception-handling topics that are specific to the latter exam. Assertions are covered only by the second exam. This chapter covers

- The `throw` statement and `throws` clause
- The `try` statement with multi-`catch` and `finally` clauses (new in Java 7)
- Auto-closing resources with a `try`-with-resources statement (new in Java 7)
- Custom exceptions
- Testing invariants by using assertions

This chapter assumes that you're already aware of the basic exception handling covered by the OCA Java SE 7 Programmer I exam. If required, you can refer to *OCA Java SE 7 Programmer I Certification Guide: Prepare for the 1Z0-803 Exam* (Manning, 2013), which covers Java's basic exception handling.

Let's get started with the use of the `throw` statement and the `throws` clause.

6.1 Using the throw statement and the throws clause

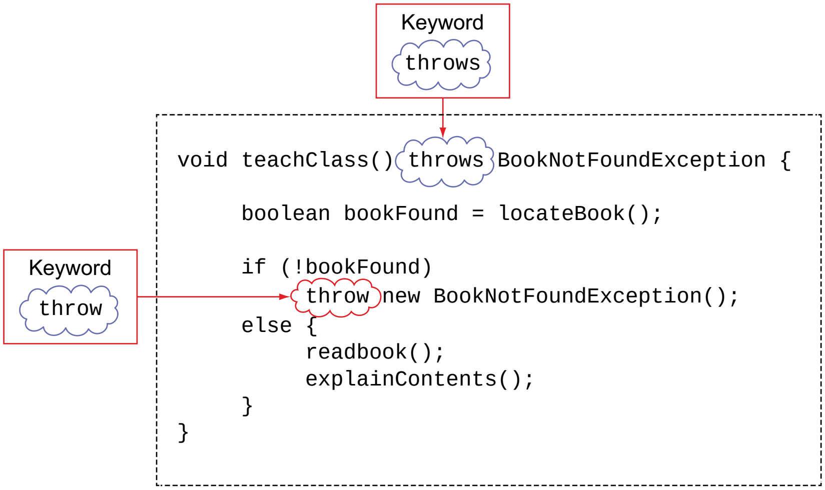
[6.1] Use the throw statement and throws clause

Imagine that you're assigned the task of finding a specific book, and then reading and explaining its contents to a class of students. The required sequence looks like the following:

- Get the specified book.
- Read aloud its contents.
- Explain the contents to a class of students.

But what happens if you can't find the specified book? You can't proceed with the rest of the actions without it, so you need to report back to the person who assigned the task to you. This unexpected event (missing book) prevents you from completing your task. By reporting it, you want the *originator* of this request to take corrective or alternate steps.

Let's code the preceding task as method `teachClass()`, as shown in figure 6.1, and use it to compare the `throw` statement and the `throws` clause. This example code is for demonstration purposes only, because it uses methods `locateBook()`, `readBook()`, and `explainContents()`, which aren't defined.



```
void teachClass() throws BookNotFoundException {

     boolean bookFound = locateBook();

     if (!bookFound)
          throw new BookNotFoundException();
     else {
          readbook();
          explainContents();
     }
}
```

Figure 6.1 Comparing the `throw` statement and the `throws` clause

The code in figure 6.1 is simple to follow. On execution of `throw new BookNotFoundException`, the execution of `teachClass()` halts. The JVM creates an instance of `BookNotFoundException` and sends it off to the caller of `teachClass()` so alternate arrangements can be made.

The `throw` statement is used to *throw* an instance of `BookNotFoundException`. The `throws` statement is used in the declaration of method `teachClass()` to signal that it can throw `BookNotFoundException`.

Why does a method choose to throw an exception as opposed to handling it itself? It's a contract between the *calling* method and the *called* method. Referring to method `teachClass()` shown in figure 6.1, the *caller* of `teachClass` would like to be informed if `teachClass()` is unable to find the specified book. Method `teachClass()` doesn't handle `BookNotFoundException` because its responsibilities don't include how to work around a missing book.

The preceding example helps identify a situation when you'd *want* a method to throw an exception, rather than handling it itself. It shows you how to use and compare the statement `throw` and clause `throws`—to *throw* exceptions and to signal that a method *might* throw an exception. The example also shows that a calling method can define alternate code, when the called method doesn't complete successfully and throws an exception. Apart from testing this logic, the exam will test you on how to create and use methods that throw checked or unchecked (runtime) exceptions and errors, along with several other rules.

Before you move forward with the chapter, it's important to clearly identify all kinds of exception classes as shown in figure 6.2.

Here's a list of different kinds of exceptions:

- *Exception classes*—Refers to `Throwable` class and all its subclasses
- *Error classes*—Refers to `Error` class and all its subclasses
- *Runtime exception classes*—Refers to `RuntimeException` class and all its subclasses
- *Unchecked exception classes*—Refers to runtime exception classes and error classes
- *Checked exceptions classes*—Refers to all exception classes other than the unchecked exception classes. Class `Throwable` and any of its subclasses that aren't a subclass of either `Error` or `RuntimeException` are checked exceptions.

Now let's create a method that throws checked exceptions.

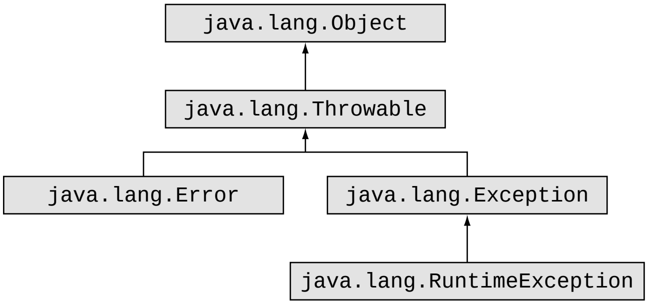


Figure 6.2 Using exception hierarchy to identify all kinds of exception classes

6.1.1 Creating a method that throws a checked exception

Let's create a simple method that doesn't handle the checked exception thrown by it, by using the statement throw and clause throws. Class DemoThrowsException defines method readFile(), which includes a throws clause in its method declaration. The actual throwing of an exception is accomplished by the throw statement:

```
import java.io.FileNotFoundException;
class DemoThrowsException {
    public void readFile(String file) throws FileNotFoundException {
        boolean found = findFile(file);
        if (!found)
            throw new FileNotFoundException("Missing file");
        else {
            //code to read file
        }
    }
    boolean findFile(String file) {
        //code to return true if file can be located
    }
}
```

throws statement indicates method can throw FileNotFoundException or one of its subclasses

If file can't be found, code creates and throws instance of FileNotFoundException by using throw statement

A method can have multiple comma-separated class names of exceptions in its throws clause. Including runtime exceptions or errors in the method declaration isn't required. Including them in the documentation is the preferred way to mention them. A method can still throw runtime exceptions or errors without including them in its throws clause.

EXAM TIP Syntactically, you don't always need a combination of the throw statement and throws clause to create a method that throws an exception (checked or unchecked). You can replace the throw statement with a method that throws an exception.

6.1.2 Using a method that throws a checked exception

To use a method that throws a *checked exception*, you must do one of the following:

- *Handle the exception*—Enclose the code within a try block and *catch* the thrown exception.
- *Declare it to be thrown*—Declare the exception to be thrown by using the throws clause.
- *Handle and declare*—Implement both of the preceding options together.

EXAM TIP The rule of either handling or declaring an exception is also referred to as the handle-or-declare rule. To use a method that throws a checked exception, you must either handle the exception or declare it to be thrown. But this rule only applies to the checked exceptions and not to the unchecked exceptions.

In the following example, method `useReadFile()` *handles* `FileNotFoundException` (a checked exception) thrown by `readFile()` by using a `try-catch` block:

```
class DemoThrowsException {
    public void readFile(String file) throws FileNotFoundException {
        //..code
    }
    void useReadFile(String name) {
        try {
            readFile(name);
        }
        catch (FileNotFoundException e) {
            //code
        }
    }
}
```

useReadFile uses readFile that throws FileNotFoundException

Call to readFile should be enclosed in try block because it throws checked FileNotFoundException

A modified definition of method `useReadFile()` *declares to throw* a `FileNotFoundException`:

```
class DemoThrowsException {
    public void readFile(String file) throws FileNotFoundException {
        //..code
    }
    void useReadFile(String name) throws FileNotFoundException{
        readFile(name);
    }
}
```

useReadFile() declares that it could throw FileNotFoundException.

readFile need not be enclosed in try-catch block

The compiler doesn't complain if you mix the preceding approaches. Method `useReadFile()` can handle `FileNotFoundException` itself and still declare it to be thrown (highlighted in bold):

```
import java.io.*;
class DemoThrowsException {
    public void readFile(String file) throws FileNotFoundException {
        //..code
    }
    void useReadFile(String name) throws FileNotFoundException{

        try {
            readFile(name);
        }
        catch (FileNotFoundException e) {
            //code
        }
    }
}
```

useReadFile() declares that it could throw FileNotFoundException.

readFile is enclosed in try-catch block

So what happens when FileNotFoundException is thrown by method readFile()? Will its catch block handle FileNotFoundException, or will readFile() throw the exception to its calling method? Let's find out in our first "Twist in the Tale" exercise.

Twist in the Tale 6.1

Let's modify some of the code used in the previous examples. Which answer correctly shows the output of class TwistThrowsException?

```
import java.io.*;
class TwistThrowsException {
    public void readFile(String file) throws FileNotFoundException {
        System.out.println("readFile");
        throw new FileNotFoundException();
    }
    void useReadFile(String name) throws FileNotFoundException {
        System.out.println("useReadFile");
        try {
            readFile(name);
        }
        catch (FileNotFoundException e) {
            //code
        }
    }
    public static void main(String args[]) {
        new TwistThrowsException().useReadFile("a");
    }
}
```

a
```
useReadFile
readFile
FileNotFoundException thrown at runtime
```

b
```
useReadFile
FileNotFoundException thrown at runtime
```

c Compilation error

d FileNotFoundException thrown at runtime

e
```
useReadFile
readFile
```

Did you ever happen to debug or fix a piece of code written by someone else? During such a session, have you ever noticed that to avoid coding multiple exception handlers (prior to Java 7), programmers often caught an overly broad exception, which often made its way into the production code? The next tip shows why you must avoid that technique.

Practical issues with catching an overly broad exception

Many times (prior to Java 7), developers took a shorter route to handle all checked exceptions: they defined just one exception handler, `java.lang.Exception`. Here's an example:

```
class MyCustomException extends Exception {}
class MultiCatch {
    void myMethod(Connection con, String fileName) {
        try {
            // code
        }
        catch (Exception e) {
            // code
        }
    }
}
```

Code that might throw MyCustomException, FileNotFoundException, or SQLException

Catch all types of checked and runtime exceptions.

This, of course, has issues, because `java.lang.Exception` is the superclass of all exceptions, including `RuntimeException`. The preceding `catch` block will *silently* catch all types of runtime exceptions also. It might become difficult to debug such code—it might log the same message for different exceptions or might try to handle all types of exceptions in the same way, which might not have been the intent.

6.1.3 Creating and using a method that throws runtime exceptions or errors

While creating a method that throws a runtime exception or error, including the exception or error name in the `throws` clause isn't required. A method that throws a runtime exception or error isn't subject to the handle-or-declare rule.

Let's see this concept in action by modifying the preceding example so method `readFile()` throws a `NullPointerException` (a runtime exception) when a `null` value is passed to it (code changes are shown in bold in this example and throughout the rest of the chapter):

```
import java.io.FileNotFoundException;
class DemoThrowsException {
    public void readFile(String file) throws FileNotFoundException {
        if (file == null)
            throw new NullPointerException();
        boolean found = findFile(file);
        if (!found)
            throw new FileNotFoundException("Missing file");
        else {
            //code to read file
        }
    }
    boolean findFile(String file) {
        //code to return true if file can be located
    }
}
```

throws clause indicates that this method can throw FileNotFoundException

Code throws NullPointerException, but it's not included in throws clause.

The exam might trick you by including the names of runtime exceptions and errors in one method's declaration and leaving them out in another. (You *can* include the name of unchecked exceptions in the `throws` clause, but you don't have to.) Assuming that the rest of the code remains the same, the following method declaration is correct:

```
public void readFile(String file)
                  throws NullPointerException, FileNotFoundException {
    //rest of the code remains same
}
```

Though not required, including runtime exceptions in throws clause is valid

EXAM TIP Adding runtime exceptions or errors to a method's declaration isn't required. A method can throw a runtime exception or error irrespective of whether its name is included in its `throws` clause.

Table 6.1 lists common errors and runtime exceptions. All are covered in detail in *OCA Java SE 7 Programmer I Certification Guide.*

Table 6.1 Common errors and exceptions

Runtime exceptions	Errors
`ArrayIndexOutOfBoundsException`	`ExceptionInInitializerError`
`IndexOutOfBoundsException`	`StackOverflowError`
`ClassCastException`	`NoClassDefFoundError`
`IllegalArgumentException`	`OutOfMemoryError`
`IllegalStateException`	
`NullPointerException`	
`NumberFormatException`	

6.1.4 Points to note while using the throw statement and the throws clause

Apart from understanding the need for throwing exceptions and using their syntax, you need to know a few rules about throwing exceptions with the `throw` statement and the `throws` clause. These rules are presented in this section.

A METHOD CAN THROW A SUBCLASS OF CHECKED EXCEPTION MENTIONED IN ITS THROWS CLAUSE, NOT ITS SUPERCLASS

Because class `IOException` is a superclass of class `FileNotFoundException`, method `readFile()` can't throw an object of class `IOException`:

throws clause includes FileNotFoundException

```
class DemoThrowsException {
    public void readFile(String file) throws FileNotFoundException {
        boolean found = findFile(file);
        if (!found)
            throw new IOException("Missing file");
        else {
            //code to read file
        }
    }
    boolean findFile(String file) {
        //code to return true if file can be located
    }
}
```

Won't compile; can't throw object of superclass of checked exception mentioned in throws clause.

Let's modify the definition of method `readFile()` by declaring it might throw an `IOException`. Because `IOException` is a superclass of class `FileNotFoundException`, `readFile()` can throw an object of `FileNotFoundException`:

```
class DemoThrowsException {
    public void readFile(String file) throws IOException {
        boolean found = findFile(file);
        if (!found)
            throw new FileNotFoundException("Missing file");
        else {
            //code to read file
        }
    }
    boolean findFile(String file) {
        //code to return true if file can be located
    }
}
```

throws clause includes IOException

Will compile; can throw object of derived class of checked exception mentioned in throws clause.

Note that this rule doesn't apply to errors and runtime exceptions.

EXAM TIP An overriding method can throw any error or runtime exception, irrespective of whether they're thrown by the overridden method or not.

A METHOD CAN HANDLE THE EXCEPTION AND STILL DECLARE IT TO BE THROWN

This is usually done by methods whose exception handlers might throw the same exception. Method `useReadFile()` handles the `FileNotFoundException` and also declares it to be rethrown:

```
class DemoThrowsException {
    public void readFile(String file) throws FileNotFoundException {
        //..code
    }
    void useReadFile(String name) throws FileNotFoundException {
        try {
            readFile(name);
        }
        catch (FileNotFoundException e) {
            //..code
            throw e;
        }
    }
}
```

useReadFile() could throw FileNotFound-Exception.

Code is valid; useReadFile() handles FileNot-FoundException from readFile().

FileNotFoundException instance will be handed over to method that calls useReadFile()

A method that declares a checked exception to be thrown might not actually throw it

Method `readFile()` includes the name of the checked exception `FileNotFoundException` in its `throws` clause, but doesn't throw it:

```
import java.io.FileNotFoundException;
class DemoThrowsException {
    public void readFile(String file) throws FileNotFoundException {
        System.out.println("readFile:" + file);
    }
}
```

Compiles successfully even if readFile() doesn't throw FileNotFoundException.

But do you think you can call `readFile()` as a method that doesn't throw an exception (without enclosing it within a `try-catch` block or declaring the exception)? Let's check it out in our next "Twist in the Tale" exercise.

Twist in the Tale 6.2

What is the output of class `ThrowsException`?

```
import java.io.FileNotFoundException;
class TwistThrowsException {
    public void readFile(String file) throws FileNotFoundException {
        System.out.println("readFile:" + file);
    }
    public static void main(String args[]) {
        System.out.println("main");
        new TwistThrowsException().readFile("Hello.txt");
    }
}
```

a
```
main
readFile:Hello.txt
```

b
```
main
readFile:Hello.txt
FileNotFoundException thrown at runtime
```

c Compilation error

d `FileNotFoundException` thrown at runtime

You can rethrow exceptions with more-inclusive type checking

Starting with Java 7, the variable type that you use to rethrow an exception can be more generic in the `catch` block:

```
class GenericVariableTypeToRethrowException {
    public static void main(String args[])
                                throws IOException, SQLException {
        String source = "DBMS";
```

Method declares to throw exceptions IOException and SQLException

```
        try {
            if (source.equals("DBMS"))
                throw new SQLException();
            else
                throw new IOException();
        }
        catch (Exception e) {
            throw e;
        }
    }
}
```

Type of variable e in catch block is Exception, more generic than IOException and SQLException

Catch block rethrows caught exception

In the preceding code, the `try` block can throw two types of checked exceptions: `SQLException` or `IOException`. But the type of the variable `e` in the `catch` block is `Exception`, which is a superclass of `SQLException` and `IOException`. Prior to Java 7, this code would fail to compile because `main()` is trying to throw an object of `Exception` from its `catch` block, when its method declaration states that it can throw an `IOException` or `SQLException`. (For checked exceptions, a method can't throw a superclass of the exception included in its declaration.) With Java 7, the preceding code compiles successfully, because the compiler can determine that the type of the checked exception received by the `catch` block would always be either `IOException` or `SQLException`. So it's okay to throw it, even though the type of the variable `e` is `Exception`.

Do you think the code will compile successfully if instead of rethrowing the exception in the `catch` block, you create a new object of class `Exception` and throw it? No, it won't. It would be a direct violation of the contract between the declarations of exceptions that method `main()` states to be throwing and what it *actually* throws. The following modified example won't compile:

```
class GenericVariableTypeToRethrowException2 {
    public static void main(String args[])
                          throws IOException, SQLException {
        String source = "DBMS";
        try {
            if (source.equals("DBMS"))
                throw new SQLException();
            else
                throw new IOException();
        }
        catch (Exception e) {
            throw new Exception();
        }
    }
}
```

Method declares to throw IOException and SQLException

Won't compile; catch block creates and throws Exception instance.

EXAM TIP With Java 7, you can rethrow exceptions with more inclusive type checking.

A method can declare to throw all types of exceptions, even if it doesn't

In the following example, class `ThrowExceptions` defines multiple methods that declare to throw different exception types. Class `ThrowExceptions` compiles successfully, even though its methods don't include the code that might throw these exceptions:

```
class ThrowExceptions {
    void method1() throws Error {}
    void method2() throws Exception {}
    void method3() throws Throwable {}
    void method4() throws RuntimeException {}
    void method5() throws FileNotFoundException {}
}
```

Though a `try` block can define a handler for unchecked exceptions not thrown by it, it can't do so for checked exceptions (other than `Exception`):

```
class HandleExceptions {
    void method6() {
        try {}
        catch (Error e) {}
    }
    void method7() {
        try {}
        catch (Exception e) {}
    }
    void method8() {
        try {}
        catch (Throwable e) {}
    }
    void method9() {
        try {}
        catch (RuntimeException e) {}
    }
    void method10() {
        try {}
        catch (FileNotFoundException e) {}
    }
}
```

Won't compile

In the preceding code, `method6()`, `method7()`, `method8()`, and `method9()` compile even though their `try` blocks don't define code to throw the exception being handled by their `catch` blocks. But `method10()` won't compile.

EXAM TIP A method can declare to throw any type of exception—checked or unchecked—even if it doesn't. But a `try` block can't define a `catch` block for a checked exception (other than `Exception`) if the `try` block doesn't throw that checked exception or uses a method that declares to throw that checked exception.

Before moving on to the next section, let's summarize the points to remember for the `throw` statement and the `throws` clause.

Rules to remember about the throw statement and the throws clause

- The `throw` statement is used within a method to throw an instance of a checked exception, a runtime exception, or an error.
- The `throws` clause is used with a method's declaration to list the exceptions that a method can throw.
- A method can include multiple comma-separated exception and error class names in its `throws` clause.
- The rule of either handling or declaring a checked exception is also referred to as the handle-or-declare rule.
- To use a method that throws a checked exception, you must either handle the exception or declare it to be thrown.
- The handle-or-declare rule only applies to the checked exceptions and not to the unchecked exceptions.
- Including runtime exceptions or errors in a `throws` clause isn't required.
- Adding unchecked exceptions or errors in the `throws` clause adds no obligations about handling them in a `try-catch` block.
- A method can throw a more specific exception subclass than the one mentioned in its `throws` clause, but not a more generic one (superclass).
- A method can handle a checked or unchecked exception and still declare it to be thrown.
- A method that declares a checked exception to be thrown might not throw it.
- With Java 7, you can rethrow exceptions with more inclusive type checking.
- A method can declare to throw any type of exception—checked or unchecked—even if it doesn't. But a `try` block can't define a `catch` block for a checked exception (other than `Exception`) if the `try` block doesn't throw that checked exception or uses a method that declares to throw that checked exception.

You can throw your own custom exceptions from methods by using the `throw` and `throws` statements, in the same way you work with exception classes from the Java API. Why do you need to create custom exception classes, when you can use exception classes that are already defined in the Java API? Let's discover in the next section.

6.2 Creating custom exceptions

[6.4] Create custom exceptions

Take a look at figure 6.3, which shows just a few of the existing exceptions from the Java API.

What information do you gather when a piece of code throws a `FileNotFoundException` or `ArrayIndexOutOfBoundsException`? Without reading the code, you can know that `FileNotFoundException` was probably thrown by code that couldn't locate a specified file. Similarly, `ArrayIndexOutOfBoundsException` will probably be thrown by code that tried to access an array element at an index position out of its bounds.

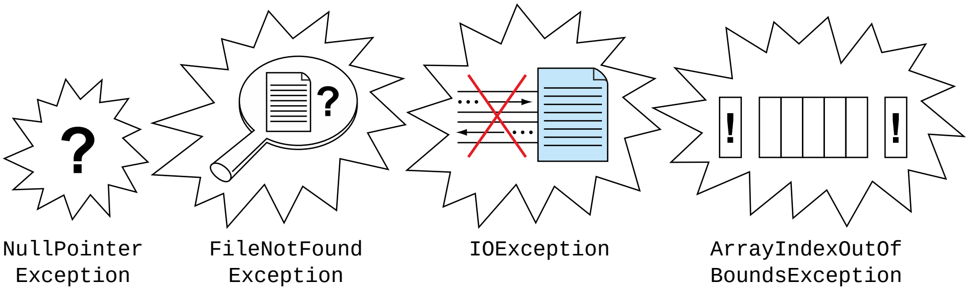


Figure 6.3 Existing exceptions defined in the Java API

The name of an exception can convey a lot of information to other developers or users, which is one of the main reasons for defining a custom exception. A custom exception can also be used to communicate a more descriptive message. For example, assume that you're developing an API to enable users to connect to your server to access its services. How can you communicate failure of the login process to a user, which can arise from multiple reasons such as a database connectivity issue or inappropriate user access privileges? One of the preferred approaches is to define and use custom exceptions.

Custom exceptions also *help* you restrict the escalation of implementing specific exceptions to higher layers. For example, data access code on your server that accesses persistent data may throw a `SQLException`. But users connecting to your server should not be returned this exception. You may catch `SQLException` and throw another checked or unchecked exception (the choice of throwing a checked or unchecked exception is beyond the scope of this book). If you can't find an appropriate existing exception, create one!

6.2.1 Creating a custom checked exception

You can subclass `java.lang.Exception` (or any of its subclasses) to define custom exceptions. To create custom checked exceptions, subclass `java.lang.Exception` or its subclasses (which aren't subclasses of `RuntimeException`). Let's revisit the exception classes in figure 6.4.

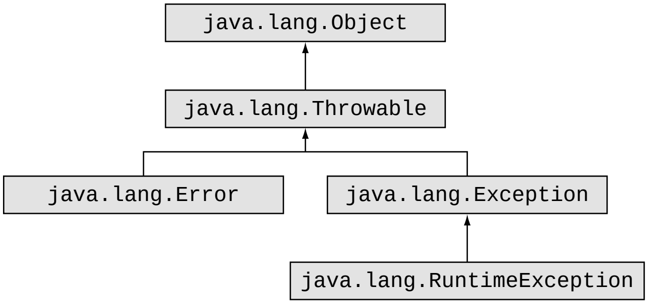


Figure 6.4 Hierarchy of `Exception` and `Error` classes

A word of caution here: even though you can extend class `java.lang.Throwable` to create your own exceptions, it isn't recommended. Class `Throwable` is the superclass of classes `java.lang.Error` and `java.lang.Exception`. The exception handler for this class will *catch* all types of errors and exceptions! For example, say an `OutOfMemoryError` brings the JVM into an unstable state. Catching it as a subclass of `Error` or `Throwable` would be undesirable and potentially dangerous. Obviously, you may not want this behavior.

EXAM TIP Don't extend class `java.lang.Throwable` to create your own exceptions, even though you can. Class `Throwable` is the superclass of classes `java.lang.Error` and `java.lang.Exception`. The exception handler for this class will *catch* all types of errors and exceptions.

Often organizations prefix the name of a custom exception with their organization, project, or module name. Let's create a custom exception, `LoginException`, which is a checked exception:

```
class LoginException extends Exception {}
```

LoginException is a checked exception.

Let's add constructors to it, a no-argument constructor and one that accepts a description of an exception or problem that occurred, as follows:

```
class LoginException extends Exception{
    public LoginException() {
        super();
    }
    public LoginException(String message) {
        super(message);
    }
}
```

No-argument constructor

Constructor that accepts String

Note that an exception class is like any other class to which you add your own methods and variables. Let's define another class, `UserActivity`, which defines method `login()`. This method creates an instance of `LoginException` and throws it if a user is unable to log in successfully. Note that I've tried to show how (checked) exceptions are thrown in real-world applications. Because the main focus here is to *throw* a checked exception, I haven't implemented method `findInDatabase()`, as it would have been implemented in the real world. It's simply reduced to returning a `false` value. Examine the following code:

Add throws clause to method declaration

```
class UserActivity {
    public void login(String user, String pwd) throws LoginException {
        if (!findInDatabase(user, pwd))
            throw new LoginException("Invalid username or password");
    }
```

Instantiate LoginException and throw it.

```
    private boolean findInDatabase(String user, String pwd) {
        // code that returns true if user/ pwd
        // combination found in database
        // false otherwise
        return false;
    }
}
```

6.2.2 Creating a custom unchecked exception

Class Error, class RuntimeException, and their derived classes are collectively referred to as *unchecked exception classes.*

You can subclass java.lang.RuntimeException to create a custom runtime exception. Here is the modified class LoginException:

```
class LoginException extends RuntimeException{
    public LoginException() {
            super();
    }
    public LoginException(String message) {
            super(message);
    }
}
```

Since LoginException extends RuntimeException, now it's an unchecked exception.

You can throw this exception (now a runtime, or unchecked, exception) in the same manner as mentioned previously in class UserActivity. The only change is that (though allowed) now you no longer need to include the throws clause in method login()'s declaration, as follows:

```
class UserActivity {
    public void login(String username,String pwd) {
    if (!findInDatabase(username, pwd))
         throw new LoginException("Invalid username or password");
    }
    private boolean findInDatabase(String username, String pwd) {
    // code that returns true if username/ pwd
    // combination found in database
    // false otherwise
    return false;
    }
}
```

No need to define throws clause in method declaration for runtime exception

To create custom Error classes, you can subclass java.lang.Error by extending it. But Error classes represent serious exceptional conditions, which shouldn't be thrown programmatically.

On the exam, you'll see both custom exceptions and exceptions from the Java API. Though overriding methods that throw exceptions isn't explicitly defined as a separate exam topic, you're likely to be tested on it.

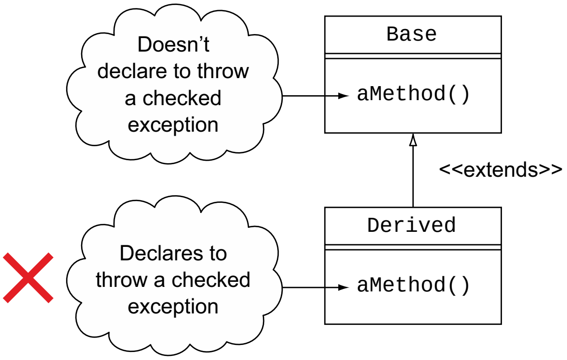

Doesn't declare to throw a checked exception
Base
aMethod()
<<extends>>
Declares to throw a checked exception
Derived
aMethod()

Figure 6.5 If an overridden method doesn't declare to throw any *checked* exception, the overriding method can't declare to throw a *checked* exception.

6.3 Overriding methods that throw exceptions

In this section, you'll work through compilation issues that occur with overridden and overriding methods, when either of them declares to throw a checked exception. Multiple combinations exist, as shown in figures 6.5 and 6.6.

EXAM TIP With overriding and overridden methods, it's all about which checked exceptions an overridden method and an overriding method declare, not about the checked exceptions both actually throw.

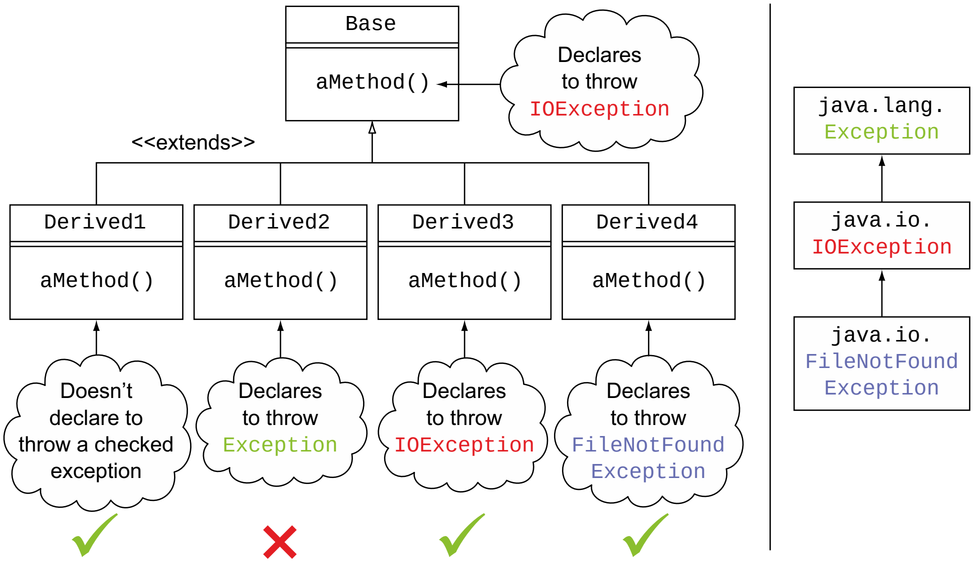

Base
aMethod()
Declares to throw IOException
<<extends>>
Derived1
aMethod()
Derived2
aMethod()
Derived3
aMethod()
Derived4
aMethod()
Doesn't declare to throw a checked exception
✓
Declares to throw Exception
✗
Declares to throw IOException
✓
Declares to throw FileNotFound Exception
✓
java.lang. Exception
java.io. IOException
java.io. FileNotFound Exception

Figure 6.6 If an overridden method declares to throw a *checked* exception, the overriding method can choose to declare to not throw any checked exception, throw the same exception, or throw a more specific exception. The overriding method can't declare to throw a more generic checked exception.

EXAM TIP Method overriding rules apply *only* to checked exceptions. They don't apply to runtime exceptions or errors. Beware: you're likely to be tested on this difference on the exam.

Let's work with all these combinations by using code snippets.

RULE 1: IF A BASE CLASS METHOD DOESN'T DECLARE TO THROW A CHECKED EXCEPTION, AN OVERRIDING METHOD IN THE DERIVED CLASS CAN'T DECLARE TO THROW A CHECKED EXCEPTION

Examine the following code:

```
class Base {
    public void aMethod() {}
    public void noRuntimeException() {}
}
class Derived extends Base {
    public void aMethod() throws Exception {}
    public void noRuntimeException() throws RuntimeException {}
}
```

This line fails to compile.

This line compiles successfully.

RULE 2: IF A BASE CLASS METHOD DECLARES TO THROW A CHECKED EXCEPTION, AN OVERRIDING METHOD IN THE DERIVED CLASS CAN CHOOSE NOT TO DECLARE TO THROW ANY CHECKED EXCEPTION

Examine the following code:

```
class Base {
    public void aMethod() throws IOException {}
    public void withRuntimeException() throws RuntimeException {}

}
class Derived1 extends Base {
    public void aMethod() {}
    public void withRuntimeException() {}
}
```

Both lines compile successfully.

RULE 3: IF A BASE CLASS METHOD DECLARES TO THROW A CHECKED EXCEPTION, AN OVERRIDING METHOD IN THE DERIVED CLASS CANNOT DECLARE TO THROW A SUPERCLASS OF THE EXCEPTION THROWN BY THE ONE IN THE BASE CLASS

Examine the following code:

```
class Base {
    public void aMethod() throws IOException {}
    public void withRuntimeException() throws NullPointerException {}

}
class Derived2 extends Base {
    public void aMethod() throws Exception {}
    public void withRuntimeException() throws RuntimeException{}

}
```

This line fails to compile.

Rule 4: If a base class method declares to throw a checked exception, an overriding method in the derived class can declare to throw the same exception

The following code compiles successfully:

```
class Base {
    void aMethod() throws IOException {}
    void methodUncheckedEx() throws Error {}
}
class Derived3 extends Base {
    void aMethod() throws IOException {}
    void methodUncheckedEx() throws NullPointerException {}
}
```

Rule 5: If a base class method declares to throw a checked exception, an overriding method in the derived class can declare to throw a derived class of the exception thrown by the one in the base class

The following code compiles successfully:

```
class Base {
    void aMethod() throws IOException {}
}
class Derived4 extends Base {
    void aMethod() throws FileNotFoundException {}
}
```

After working with the method overriding rules that include throwing exceptions, let's use the `try` statement with multi-`catch` and `finally` clauses. Starting with Java 7, the `try` statement can *catch* multiple exceptions in the same handler, as discussed in the next section.

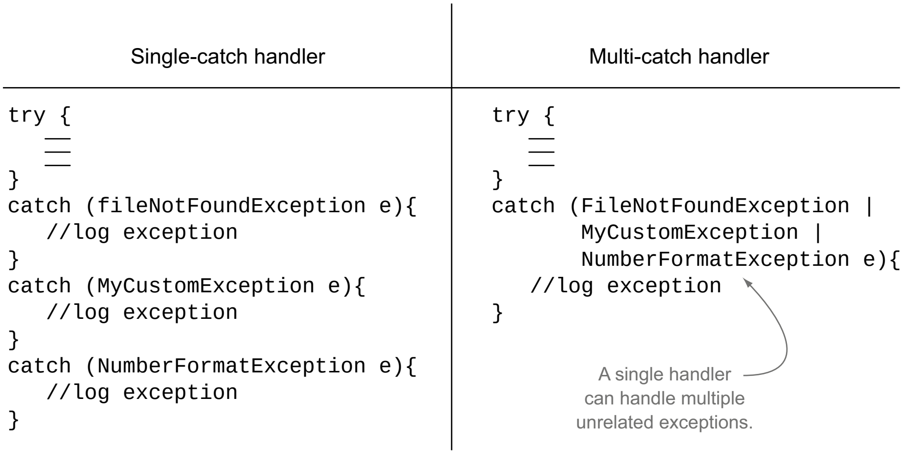
6.4 Using the try statement with multi-catch and finally clauses

[6.2] Use the try statement with multi-catch and finally clauses

Prior to Java 7, if a `try` block needed to execute the same action for multiple exceptions thrown, it had to define separate handlers for each of them. Starting with Java 7, you can *catch* multiple, unrelated exceptions with one handler, also called a multi-`catch`.

6.4.1 Comparing single-catch handlers and multi-catch handlers

The multi-`catch` comes in handy if you need to execute the same action for handling multiple, unrelated exceptions. I specifically mention unrelated exceptions, because an exception handler for, say, `MyException`, can handle `MyException` and all of its subclasses. You can compare this approach of defining separate exception handlers



```
try {

}
catch (fileNotFoundException e){
   //log exception
}
catch (MyCustomException e){
   //log exception
}
catch (NumberFormatException e){
   //log exception
}
```

```
try {

}
catch (FileNotFoundException |
       MyCustomException |
       NumberFormatException e){
   //log exception
}
```

A single handler can handle multiple unrelated exceptions.

Figure 6.7 Comparing the differences between executing the same action with single-`catch` and multi-`catch` exception handlers

(prior to Java 7) with defining only one exception handler to execute the same steps for multiple unrelated exceptions (starting in Java 7) by using figure 6.7.

Now that you know the difference between a multi-`catch` and single-`catch` handler, let's dive into the details of defining multi-`catch` handlers with `finally` clauses.

6.4.2 Handling multiple exceptions in the same exception handler

You should know the basic syntax for creating and using multi-`catch` blocks, together with the *gotchas* that may be used on the exam. So let's start with the basic syntax of multi-`catch` blocks.

Basic syntax of multi-catch block

To *catch* multiple exceptions in a single handler, just separate the different exception types with a vertical bar (`|`). The following is an example of a `try` block that defines code that can handle `FileNotFoundException` and `SQLException` (or any of their subclasses) using a multi-`catch` block:

```
Line1> class MultiCatch {
Line2>     void myMethod(Connection con, String fileName) {
Line3>         try {
Line4>             File file = new File(fileName);
Line5>             FileInputStream fin = new FileInputStream(file);

Line6>             Statement stmt = con.createStatement();
Line7>         }
Line8>         catch (FileNotFoundException | SQLException e) {
Line9>             System.out.println(e.toString());
Line10>        }
Line11>    }
Line12>}
```

Might throw FileNotFoundException (Lines 4–5)

Might throw SQLException (Line 6)

Executes if line 5 throws FileNotFoundException or line 6 throws SQLException or any of their subclasses. (Line 9)

In the preceding code, the exception handler will execute if the code throws `FileNotFoundException`, `SQLException`, or any of their subclasses.

Finally block can follow multi-catch block

A multi-`catch` block can be followed by a `finally` block. Here's an example:

```
class MultiCatchWithFinally {
    void myMethod(Connection con, String fileName) {
        try {
            File file = new File(fileName);
            FileInputStream fin = new FileInputStream(file);

            Statement stmt = con.createStatement();
        }
        catch (FileNotFoundException | SQLException e) {
            System.out.println(e.toString());
        }
        finally {
            System.out.println("finally");
        }
    }
}
```

finally block can follow multi-catch block

The syntax seems to be simple. So let's look at some of the *gotchas* that you need to be aware of for the exam.

Exceptions that you catch in a multi-catch block can't share an inheritance relationship

What happens if you add another line of code in the previous example, which involves reading from `FileInputStream`, which might throw an `IOException`? Let's add `IOException` to the list of exceptions being caught in the multi-`catch` block:

```
class MultiCatch {
    void myMethod(Connection con, String fileName) {
        try {
            File file = new File(fileName);
            FileInputStream fin = new FileInputStream(file);
            fin.read();

            Statement stmt = con.createStatement();
        }
        catch (IOException| FileNotFoundException | SQLException e) {
            System.out.println(e.toString());
        }
    }
}
```

May throw FileNotFoundException

May throw IOException

May throw SQLException

Fails to compile

This code fails compilation with the following error message:

```
MultiCatch.java:13: error: Alternatives in a multi-catch statement cannot
be related by subclassing
            catch (IOException | FileNotFoundException | SQLException e) {
                                 ^
  Alternative FileNotFoundException is a subclass of alternative IOException
1 error
```

Looks like the code fails to compile because the IOException is caught before the FileNotFoundException. In regular catch blocks, if you catch a superclass exception before a derived class exception, the code won't compile. So let's swap the order of IOException and FileNotFoundException in the preceding code:

```
class MultiCatch {
    void myMethod(Connection con, String fileName) {
        try {
            File file = new File(fileName);
            FileInputStream fin = new FileInputStream(file);
            fin.read();

            Statement stmt = con.createStatement();
        }
        catch (FileNotFoundException | IOException| SQLException e) {
            System.out.println(e.toString());
        }
    }
}
```

Might throw FileNotFound-Exception

Might throw IOException

Might throw SQLException

Swapping exception types doesn't make a difference; code fails to compile.

The code fails compilation with the following message:

```
Alternatives in a multi-catch statement cannot be related by subclassing
            catch (FileNotFoundException | IOException | SQLException e) {
                                           ^
  Alternative FileNotFoundException is a subclass of alternative IOException
1 error
```

So the takeaway from the previous examples is that you can't use subclasses as alternative types in a multi-catch block. The correct multi-catch block for code that may throw an IOException, FileNotFoundException, and SQLException is as follows:

```
class MultiCatch {
    void myMethod(Connection con, String fileName) {
        try {
            ..
        }
        catch (IOException | SQLException e) {
            System.out.println(e.toString());
        }
    }
}
```

Code that might throw IOException, FileNotFound-Exception, or SQLException

Catch IOException (superclass of FileNotFoundException) and SQLException or any of their subclasses.

Because multi-catch blocks are used to execute the same piece of code, when multiple exceptions are thrown, it makes no sense to use subclasses.

COMBINING MULTI-CATCH AND SINGLE-CATCH BLOCKS

You can combine multi-catch and single-catch blocks, as shown in the following code:

```
class MultiAndSingleCatch {
    void myMethod(Connection con, String fileName) {
        try {
            ..
        }
        catch (FileNotFoundException e) {}
        catch (IOException | SQLException e) {}
    }
}
```

Catch FileNotFoundException or any of its subclasses.

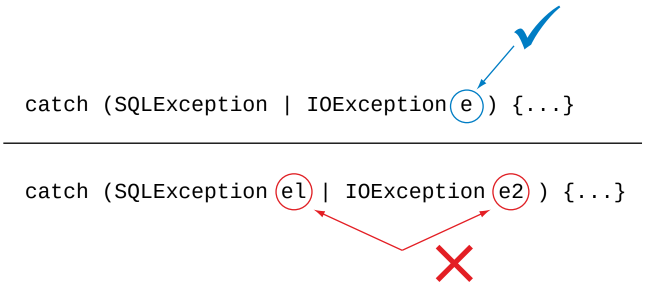
Code that may throw IOException, FileNotFoundException, or SQLException

Catch IOException and SQLException or any of their subclasses (except for FileNotFoundException or its subclasses).

EXAM TIP Watch out for a combination of multi-`catch` and single-`catch` exception handlers on the exam. They can get quite tricky.

Using a single exception variable in the multi-catch block

It's easy to overlook that even though a multi-`catch` handler defines multiple exception types, it must use only one variable. Figure 6.8 defines two multi-`catch` exception handlers. The latter multi-`catch` block uses multiple variables, which is incorrect and won't compile.



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

catch (SQLException el | IOException e2 ) {...}
```

Figure 6.8 A multi-`catch` block that uses multiple exception variables won't compile.

In a multi-catch block, variable e is implicitly final

In a multi-`catch` block, the variable that accepts the exception object is implicitly final. A final variable can't be reassigned a value. So if you try to reassign a value to the variable of the multi-`catch` exception handler, the code won't compile:

```
class MultiCatch {
    void myMethod(Connection con, String fileName) {
        try {
            ..
        }
        catch (IOException| SQLException e) {
            e = new FileNotFoundException();
        }
    }
}
```

Code that may throw IOException or SQLException

Catch IOException and SQLException.

Won't compile; can't reassign value to variable e because it's implicitly final.

Type of exception variable in a multi-catch block

In a multi-`catch` block, the type of the reference variable that accepts the exception object is a common base class of all the exception types mentioned in a multi-`catch` block. In the following code, the type of the reference variable `ex` is `Exception`, the common base class of Exception1 and Exception2. This is why calling info() on ex won't compile:

```
class Exception1 extends IOException{

    public String info() {
        return "I'm Base Exception";
    }
}
```

Custom exception Exception1 extends IOException.

Exception1 defines method info.

```
class Exception2 extends Exception{

    public String info() {
        return "I'm Derived Exception";
    }
}
```

Custom exception Exception2 extends Exception.

Exception2 defines method info.

```
class TestVariableTypeInMultiCatch {
    public static void main(String args[]) {
        try {
            int a = 10;

            if (a <= 10)throw new Exception1();
            else throw new Exception2();
        }
        catch (Exception1 | Exception2 ex) {

            System.out.println(ex.info());
        }
    }
}
```

Type of variable ex is Exception, common superclass of Exception1 and Exception2.

Won't compile; class Exception doesn't define method info.

Okay, let's modify the code from the previous example so it prints out the value of variable ex, in the next "Twist in the Tale" exercise.

Twist in the Tale 6.3

Which answer correctly shows the output of class TestMultiCatch?

```
import java.io.*;

class Exception1 extends IOException{}
class Exception2 extends Exception{}

class TestMultiCatch {
    public static void main(String args[]) {
        try {
            int a = 10;
            if (a <= 10)throw new Exception1();
            else throw new Exception2();          // line1
        }
        catch (Exception1 | Exception2 ex) {
            System.out.println(ex);
        }
    }
}
```

- a Value similar to `Exception1@96a34`
- b Value similar to `Exception2@45a86e`
- c `Exception1`
- d `Exception2`

Do you think this code will compile successfully if you comment the code marked with `//line1`?

And what would happen if both custom exceptions used in the preceding example code implement an interface?

```
interface IEx {
    String info();
}
class Exception1 extends IOException implements IEx{
    public String info() {
        return "I'm Base Exception";
    }
}

class Exception2 extends Exception implements IEx {
    public String info() {
        return "I'm Derived Exception";
    }
}

class TestVariableTypeInMultiCatch {
    public static void main(String args[]) {
        try {
            int a = 10;

            if (a <= 10)throw new Exception1();
            else throw new Exception2();
        }
        catch (Exception1 | Exception2 ex) {
            System.out.println(ex.info());
        }
    }
}
```

Custom exception Exception1 extends IOException; implements IEx.

Custom exception Exception2 extends Exception; implements IEx.

Variable ex is an intersection type, with Exception and IEx as its bounds.

Compiles successfully.

In the preceding code, variable `ex` is of *intersection type* with `Exception` and `IEx` as its bounds. You can call methods accessible to class `Exception` and interface `IEx` on the reference variable `ex`.

The next section covers another major language enhancement in Java: auto-closing resources by using a `try`-with-resources statement.

6.5 Auto-closing resources with a try-with-resources statement

[6.3] Auto-close resources with a try-with-resources statement

Prior to Java 7, developers used `finally` blocks to close resources such as file handlers, and database or network connections. Here's a quick example to show how a `FileInputStream` instance was closed prior to Java 7:

```
try {
    FileInputStream fis = /* instantiate */
    /* other code */
}
finally {
    if (fis != null)
        try {
            fis.close();
        }
    catch (Exception e) {/* */}
}
```

As you can see, closing a resource required a lot of boilerplate code and was error-prone too. What if a developer didn't close a resource in a `finally` block?

Starting with Java 7, you can use a `try`-with-resources statement to *auto-close* resources defined with the `try` statement.

6.5.1 How to use a try-with-resources statement

The `try`-with-resources statement is a type of `try` statement that can declare one or more resources. A *resource* is an object such as file handlers and database or network connections, which *should* be closed after it's no longer required. If you declare a resource by using a `try`-with-resources statement, it automatically *closes* the resource by calling its `close` method, *just* before the end of the `try` block. A resource must implement the `java.lang.AutoCloseable` interface or any of its subinterfaces to be eligible to be declared in a `try`-with-resources statement. Let's start with an example.

AN EXAMPLE

In the following example, a `try`-with-resources statement declares and initializes an object (`fin`) of type `FileInputStream`. Because the `try`-with-resources statement is supposed to automatically call method `close()` on `fin`, the following code doesn't explicitly make this call:

```
class AutoClose {
    void readFileContents(String fileName) {
        File file = new File(fileName);
        try (FileInputStream fin = new FileInputStream(file)){
            //.. some code
        }
```

Code to open FileInputStream; can throw FileNotFoundException

Some code

```
        catch (FileNotFoundException e) {
            System.out.println(e.toString());
        }
    }
}
```

Catch FileNotFoundException that might be thrown by code to initialize fin.

But wait! This code doesn't compile and gives the following compilation error:

```
AutoClose.java:7: error: unreported exception IOException;
must be caught or declared to be thrown
            try (FileInputStream fin = new FileInputStream(file)){
                                     ^
  exception thrown from implicit call to close() on resource variable 'fin'
1 error
```

So what went wrong? The `try`-with-resources statement calls method `close()` *just* before the completion of the `try` block. Note that if method `close()` throws any exception, it should be taken care of by *your* method; it must either catch it or declare it to be thrown. Code that handles both `FileNotFoundException` and `IOException` is correct:

```
class AutoClose {
    void readFileContents(String fileName) {
        File file = new File(fileName);
        try (FileInputStream fin = new FileInputStream(file)){
            //.. some code
        }
        catch (IOException e) {
            System.out.println(e.toString());
        }
    }
}
```

Code to initialize fin can throw FileNotFoundException; calling close on fin can throw IOException.

Details of this code not required

Catch IOException (IOException is superclass of FileNotFoundException).

The following code declares `IOException` to be thrown, and so it's also correct:

```
class AutoClose {
    void readFileContents(String fileName) throws IOException {
        File file = new File(fileName);
        try (FileInputStream fin = new FileInputStream(file)){
            //.. some code
        }
    }
}
```

Declares IOException to be thrown.

Initialization of fin may throw FileNotFoundException; calling close on fin may throw IOException.

Did you notice that the `try` block defined in the preceding code wasn't followed by either a `catch` or a `finally` block? This is unlike a regular `try` block, which must be followed by either a `catch` or a `finally` block.

NOTE FileInputStream implements java.io.Closeable, which, starting with Java 7, extends java.lang.AutoCloseable. So FileInputStream is a valid resource to be used by the try statement.

6.5.2 Suppressed exceptions

In a try-with-resources statement, if both the code in the try block and close() throw an exception, the exception thrown by close() is *suppressed* by the exception thrown by the try block.

The resources initialized by the try-with-resources statement are automatically closed, just before the end of execution of the try block. This happens regardless of whether any exceptions are thrown. Let's understand the flow of code by using class RiverRaft, which implements the AutoCloseable interface:

```
class RiverRaft implements AutoCloseable {
    public RiverRaft() throws Exception {
        System.out.println("Start raft");
    }
    public void crossRapid() throws Exception {
        System.out.println("Cross rapid");
        throw new Exception("Cross Rapid exception");
    }
    public void close() throws Exception {
        System.out.println("Reach river bank");
    }
}
```

The class SuppressedExceptions initializes an instance of RiverRaft by using a try-with-resources statement:

```
public class SuppressedExceptions {
    public static void main(String[] args) throws Exception {
        try ( RiverRaft raft = new RiverRaft(); ) {
            raft.crossRapid();

        }
        catch (Exception e) {
            System.out.println("Exception caught:" + e);
        }
    }
}
```

❶ **Instantiate RiverRaft; no exceptions thrown.**

❷ **Method crossRapid() throws Exception.**

❸ **close() called on raft, before control is transferred to exception handler**

EXAM TIP Though not required, I've deliberately used a semicolon (;) after declaring and initializing raft in the preceding code. Watch out for questions on the exam that include or exclude a semicolon at the end of the resource defined by a try-with-resources statement. A try-with-resources statement can declare multiple resources, which are separated by a semicolon. After the last resource declaration, a semicolon is optional.

For the code at ❶, the try-with-resources statement creates a RiverRaft instance. For the code at ❷, method crossRapid() throws an exception, but the try-with-resources statement executes `close()` ❸ before passing the control to the exception handler. Here's the output of the preceding code:

```
Start raft
Cross rapid
Reach river bank
Exception caught:java.lang.Exception: Cross Rapid exception
```

Now what happens if `close()` in `RiverRaft` also throws an exception? Which exception will be propagated to the exception handler? Will it be the exception from `close()` or from `crossRapid()`? Here's the modified code:

```
public class SuppressedExceptions {
    public static void main(String[] args) throws Exception {
        try ( RiverRaft raft = new RiverRaft(); ) {
            raft.crossRapid();
        }
        catch (Exception e) {
            System.out.println("Exception caught:" + e);
        }
    }
}
class RiverRaft implements AutoCloseable {
    public RiverRaft() throws Exception {
        System.out.println("Start raft");
    }
    public void crossRapid() throws Exception {
        System.out.println("Cross rapid");
        throw new Exception("Cross Rapid exception");
    }
    public void close() throws Exception {
        System.out.println("Reach river bank");
        throw new Exception("Close exception");
    }
}
```

Method crossRapid() throws Exception.

Method close() also throws Exception.

The exception from method `crossRapid()` made it the exception handler. Here's the output of the preceding code:

```
Start raft
Cross rapid
Reach river bank
Exception caught:java.lang.Exception: Cross Rapid exception
```

Because the output of the previous code snippets looks identical, what do you think happened to the exception thrown by method `close()`? This exception was *suppressed* by the exception thrown by `crossRapid()`. This is a new automatic resource management feature in Java 7. In a `try`-with-resources statement, when the code enclosed within the `try` body (`try` block minus its initialization code) throws an exception, followed by an exception thrown by the `try`-with-resources statement (which implicitly calls method `close()`), then the latter is suppressed.

Even though the exception thrown by `close()` is suppressed in the preceding code, it's associated with the exception thrown by `crossRapid()`. You can retrieve the suppressed exceptions by calling `getSuppressed()` on the exception that has suppressed the other exceptions. The `getSuppressed()` method returns an array containing all of the exceptions that were suppressed in order to deliver the exception thrown by `crossRapid()`. The following modified code shows how you can trace the exception thrown by method `close()`. Because only one exception was suppressed, the code accessed the first element of the array returned by `getSuppressed()`:

```
public class SuppressedExceptions {
    public static void main(String[] args) throws Exception {
        try ( RiverRaft raft = new RiverRaft(); ) {
            raft.crossRapid();
        }
        catch (Exception e) {
            System.out.println("Exception caught:" + e);
            Throwable[] exs = e.getSuppressed();
            if (exs.length>0)
                System.out.println(exs[0]);
        }
    }
}
class RiverRaft implements AutoCloseable {
    public RiverRaft() throws Exception {
        System.out.println("Start raft");
    }
    public void crossRapid() throws Exception {
        System.out.println("Cross rapid");
        throw new Exception("Cross Rapid exception");
    }
    public void close() throws Exception {
        System.out.println("Reach river bank");
        throw new Exception("Close exception");
    }
}
```

Retrieves and prints first suppressed exception.

EXAM TIP In a `try`-with-resources statement, when the code enclosed within the `try` body (`try` minus its initialization code) throws an exception, followed by an exception thrown by the `try`-with-resources statement (which implicitly calls method `close()`), then the latter exception is suppressed. `getSuppressed()` never returns `null`. If there aren't any suppressed expressions, the length of the returned array is 0.

Now, let's take a quick look at the nuts and bolts of using a `try`-with-resources statement, which you should know for this exam.

6.5.3 *The right ingredients*

Working with a `try`-with-resources statement can be tricky because it involves several points to be taken care of. Let's start with the declaration of multiple resources in a `try`-with-resources statement.

DECLARING AND INITIALIZING RESOURCES

The variables used to refer to resources are implicitly final variables. You must *declare* and *initialize* resources in the `try`-with-resources statement. You can't define un-initialized resources:

```
void copyFileContents(String inFile, String outFile) throws IOException{
    try (FileInputStream fin;
         FileOutputStream fout;){
         //..rest of the code
    }
}
```

Won't compile; resources must be initialized.

It's acceptable to the Java compiler to initialize the resources in a `try`-with-resources statement to `null`, only as long as they aren't being reassigned a value in the `try` block (as they are implicitly final):

```
void copyFileContents(String inFile, String outFile) throws IOException{
    try (FileInputStream fin = null;
         FileOutputStream fout = null;){
         //..rest of the code
    }
}
```

Will compile successfully, if initialized with null, only as long as it isn't used.

But it doesn't make much sense to initialize a final variable with `null`. Once initialized to `null`, a resource can't be reassigned a value in a `try`-with-resources statement. If you try to do so, the code won't compile:

```
void copyFileContents(String inFile, String outFile) throws IOException{
    try (FileInputStream fin = null;
         FileOutputStream fout = null;){

         fin = new FileInputStream(inFile);
         //..rest of the code
    }
}
```

Assigned null to resources.

Won't compile; you can't reassign another value to fin.

EXAM TIP The variables defined in a `try`-with-resources statement are implicitly final.

SCOPE OF THE RESOURCE DECLARED BY TRY-WITH-RESOURCES IS LIMITED TO IT

The resource declared by `try`-with-resources is closed just before the completion of the `try` block. Also, its scope is limited to the `try` block. So if you try to access it outside the `try` block, it won't compile. Following is an example of a simple method that tries to copy contents of one file to another:

```
class AutoClose {
    void copyFileContents(String inFile, String outFile)
                                               throws IOException{
        try (FileInputStream fin = new FileInputStream(inFile);
            FileOutputStream fout = new FileOutputStream(outFile)){
```

Scope of fout limited to try block

```
            byte [] buffer = new byte[1024];
            int i = 0;
            while ((i = fin.read(buffer)) != -1)
                fout.write(buffer,0,i);
        }
        finally {
            fout = new FileOutputStream(inFile);
        }
    }
}
```

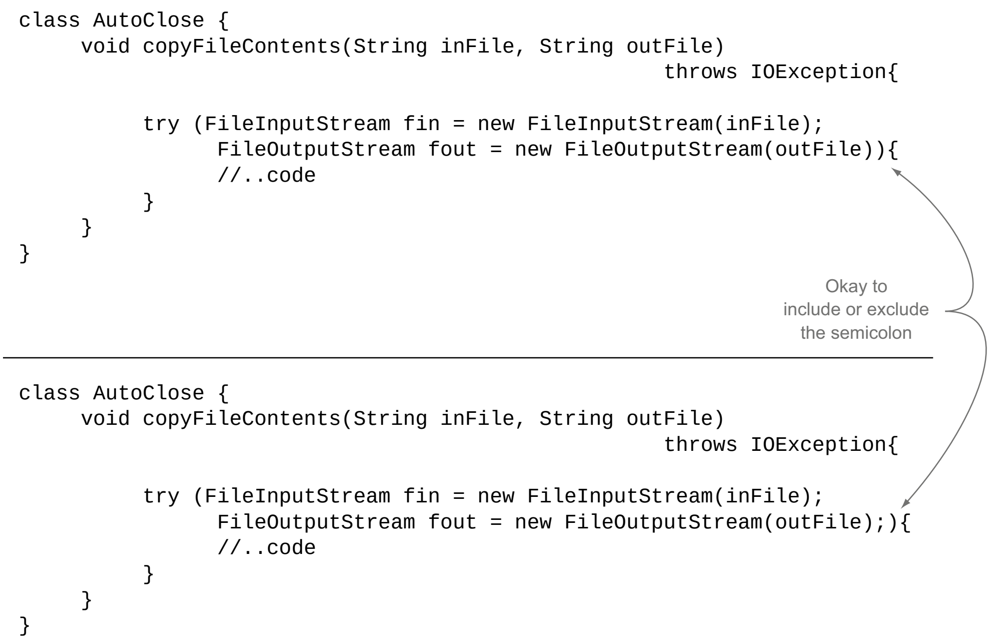
Won't compile; code outside try block can't access variable fout.

A semicolon might not follow the last resource declared by try statement

You can initialize multiple resources in a `try`-with-resources statement, separated by a semicolon (`;`). It isn't obligatory for a semicolon to follow the declaration of the last resource, as shown in figure 6.9.

Resources must implement java.io.AutoCloseable or its subinterfaces (directly or indirectly)

In the previous examples, I used objects of classes `FileInputStream` and `FileOutputStream` in a `try`-with-resources statement. These classes implement the `java.io.Closeable` interface. Starting with Java 7, `java.io.Closeable` was modified to extend the `java.lang.AutoCloseable` interface, so that classes implementing it could be used with a `try`-with-resources statement.



```
class AutoClose {
     void copyFileContents(String inFile, String outFile)
                                                    throws IOException{

          try (FileInputStream fin = new FileInputStream(inFile);
                FileOutputStream fout = new FileOutputStream(outFile)){
                //..code
          }
     }
}
```

```
class AutoClose {
     void copyFileContents(String inFile, String outFile)
                                                    throws IOException{

          try (FileInputStream fin = new FileInputStream(inFile);
                FileOutputStream fout = new FileOutputStream(outFile);){
                //..code
          }
     }
}
```

Figure 6.9 The last resource defined in a `try`-with-resources statement might not be followed by a semicolon (`;`).

To use instances of your own class with a `try`-with-resources statement, you can implement the `java.lang.AutoCloseable` interface:

```
class MyAutoCloseableRes implements AutoCloseable{
    MyAutoCloseableRes() {
        System.out.println("Constructor called");
    }
    public void close() {
        System.out.println("Close called");
    }
}
class AutoClose2 {
    void useCustomResources() {

        try (MyAutoCloseableRes res = new MyAutoCloseableRes();){
            System.out.println("within try-with-resources");
        }
        finally {
            System.out.println("finally");
        }
    }
}

class Test {
    public static void main(String args[]) {
        new AutoClose2().useCustomResources();
    }
}
```

MyAutoCloseableRes implements AutoCloseable, so it can be used in try-with-resources.

MyAutoCloseableRes implements the only method, close(), defined by AutoCloseable.

An object of MyAutoCloseableRes can be used in try-with-resources.

Try block doesn't catch any exception, because no method of MyAutoCloseableRes throws exception

The output of the preceding class `Test` is as follows:

```
Constructor called
within try-with-resources
Close called
finally
```

Definition of interfaces java.lang.AutoCloseable and java.io.Closeable

On the exam, you might get to answer explicit questions on the exceptions that are thrown by method `close()` defined in the `AutoCloseable` and `Closeable` interfaces. Following is the definition of the `AutoCloseable` interface from the Java source code (minus the comments):

```
package java.lang;
public interface AutoCloseable {
    void close() throws Exception;
}
```

Method close() of AutoCloseable throws Exception.

Here's the definition of the `Closeable` interface from the Java source code (minus the comments):

```
package java.io;
public interface Closeable extends AutoCloseable {
    public void close() throws IOException;
}
```

Method close() of interface Closeable throws IOException.

Method `close()` in the `Closeable` interface overrides method `close()` from the `AutoCloseable` base interface. Method `close()`, which implements the `Closeable` interface, won't be able to throw exceptions that are broader than `IOException`. If you implement an interface, you must have valid implementations for each method defined in the interface.

THE RESOURCES DECLARED WITH TRY-WITH-RESOURCES ARE CLOSED IN THE REVERSE ORDER OF THEIR DECLARATION

Class `MyResource` implements the `AutoCloseable` interface. Its constructor accepts a name for its instance, which is printed when the constructor and method `close()` are called:

```
class MyResource implements AutoCloseable{
    String name;
    MyResource(String name) {
        this.name = name;
        System.out.println("Created:"+name);
    }
    public void close() {
        System.out.println("Closed:"+name);
    }
}
class TestAutoCloseOrder {
    public static void main(String args[]) {
        try (MyResource res1 = new MyResource("1");
             MyResource res2 = new MyResource("2")){

            System.out.println("within try-with-resources");
        }
        finally {
            System.out.println("finally");
        }
    }
}
```

MyResource implements AutoCloseable so it can be used with try-with-resources.

Creates MyResource instance and assigns it to res1.

Creates another MyResource instance and assigns it to res2.

Method close() is called first on res2 and then on res1, after execution of code on this line.

Here's the output of the preceding code:

```
Created:1
Created:2
within try-with-resources
Closed:2
Closed:1
finally
```

EXAM TIP The resources declared with the `try`-with-resources are closed in the reverse order of their declaration. In this and previous sections, we covered exceptions and worked with how the exception handlers enable you to *recover* from exceptional conditions during the execution of your program. In the next section, you'll see how the assertions enable you to *test* and *debug* your *assumptions* and flow control *during* the development of your code.

6.6 Using assertions

[6.5] Test invariants by using assertions

While testing your code, have you ever come across a situation that made you think that a variable in your code could be assigned more values than you assumed, or that your code wasn't executing as planned for a combination of values?

Assertions help you test your assumptions about the execution of code. For example, you could test your assumption by using a combination of control-flow and logging statements to print the message `Error: pages should NOT be < 200`, if the value of `pages` is greater than `200`, as follows:

```
void printReport() {
    int pages = 100;
    while (/*some condition*/) {
        if (/*some condition*/) {
            pages++;
        }
    }
    if (pages<200)
        System.out.println("Error: pages should NOT be < 200");
}
```

When assumption is not met, print an error message should not exceed 200

Because it's an assumption, it needs to be tested and fixed during the development phase. So you can use an `assert` statement to verify the preceding assumption:

```
void printReport() {
    int pages = 100;
    while (/*some condition*/) {
        if (/*some condition*/) {
            pages++;
        }
    }
    assert (pages<200): " Error: pages should NOT be < 200";
}
```

Assertion to verify and test assumptions

In the preceding example, the programmer's assumption is coded as an assertion.

An *assertion* offers a way of indicating what should always be true. An assertion is implemented using an `assert` statement that enables you to test your assumptions about the values assigned to variables and the flow of control in your code. An `assert` statement uses a `boolean` expression, which you believe to be true. If this `boolean` expression evaluates to `false`, your code will halt its execution by throwing an `AssertionError`.

Assertions are used for testing and debugging your code. They aren't for error checking, which is why they're off by default. Assertions are disabled by default so they don't become a performance liability in deployed applications.

This exam will query you on testing invariants by using assertions, the short and long form of assertions, and their appropriate and inappropriate use. Assertions were

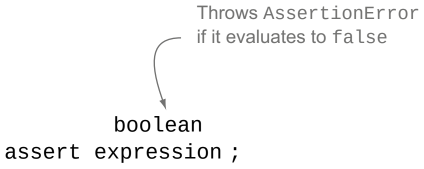


Figure 6.10 The `assert` statement's simple form accepts a `boolean` expression.

introduced in version 1.4, and though powerful, are one of the underused features of Java. Let's start with the forms of assertions.

6.6.1 Exploring the forms of assertions

An assertion is defined by using an `assert` statement that can take two forms. The simpler form uses only a `boolean` expression, as shown in figure 6.10.

If the `boolean` expression used in an `assert` statement (as shown in figure 6.10) evaluates to `false`, the JRE will throw an `AssertionError`. Assuming that assertions have been enabled, the following code

```
public class ThrowAssertionError {
    public static void main(String args[]) {
        assert false;
    }
}
```

Throws AssertionError because boolean value following assert is false.

will throw the following error at runtime:

```
Exception in thread "main" java.lang.AssertionError
    at ThrowAssertionError.main(ThrowAssertionError.java:3)
```

EXAM TIP If the `boolean` expression used in an `assert` statement evaluates to `false`, an `AssertionError` is thrown (if assertions are enabled at runtime).

As you can see, the preceding output doesn't include any custom error message. To include custom details with an `AssertionError`, you can use the longer form of an `assert` statement, as shown in figure 6.11.

The *expression* used in the longer form of an `assert` statement is another way to pass arguments to an `AssertionError`. If the `boolean` expression used in the longer form of the `assert` statement (as shown in figure 6.11) evaluates to `false`, the JRE

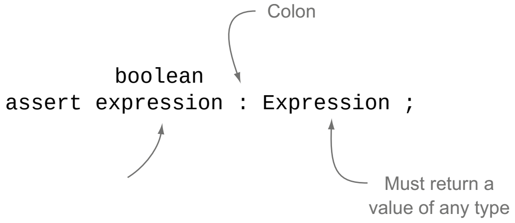


Figure 6.11 An `assert` statement's longer form accepts a `boolean` expression and an expression.

creates an object of AssertionError by passing the *expression* to its constructor. Assuming that assertions have been enabled, the following code uses the longer form of the assert statement:

```
public class ThrowDetailedAssertionError {
    public static void main(String args[]) {
        assert false : "Testing Assertions";
    }
}
```

Throws AssertionError with message "Testing Assertions"

The preceding code throws an AssertionError with the message Testing Assertions at runtime:

```
Exception in thread "main" java.lang.AssertionError: Testing Assertions
    at ThrowDetailedAssertionError.main(ThrowDetailedAssertionError.java:3)
```

The exam also will test you on the valid expressions used for the assert statement. The shorter form of the assert statement uses one expression, which *must* evaluate to a boolean value. The longer form of the assert statement uses two expressions: the first one *must* evaluate to a boolean value, but the second can evaluate to any type of primitive value or object. Look out for incorrect use of methods for the second expression, whose return type is void. What do you think is the output of the class ThrowAssertionError?

```
class Person {
    void getNothing() {}
}
public class ThrowAssertionError {
    public static void main(String args[]) {
        assert false : new Person().getNothing();
        assert true : new ArrayList<String>().clear();
    }
}
```

Line won't compile because getNothing doesn't return a value.

Line won't compile because clear doesn't return a value.

Because the methods getNothing() (class Person) and clear() (class ArrayList) don't return any value, the class ThrowAssertionError won't compile. You might find similar occurrences on the exam.

Class AssertionError uses the String representation of the value passed to its constructor for its detailed message. A String representation of an object is retrieved by calling its method toString(). Let's look at a tricky use of the assert statement. Class TrickAssert defines an assert statement and uses a boolean assignment for its first expression and an object for its second expression. Assuming that assertions are enabled, what do you think is the output of the following code?

```
class Person {
    public String toString() {
        return "Pirates of the Caribbean";
    }
}
```

```
public class TrickAssert {
    public static void main(String args[]) {
        boolean b = false;
        assert (b = true) : new Person();
    }
}
```

A lot of programmers have answered that the preceding code throws an `AssertionError`:

```
Exception in thread "main" java.lang.AssertionError: Pirates of the Caribbean
```

But it doesn't, because the expression `(b = true)` returns `true`. Revisit the example: this expression is assigning the `boolean` literal value `true` to variable `b`; it isn't comparing `b` with `true`. Also, the second expression returns an object of class `Person`. `AssertionError` executes `toString()` to retrieve a `String` representation of an object of class `Person`.

EXAM TIP The longer form of the `assert` statement uses two expressions: the first one *must* evaluate to a `boolean` value, but the second can evaluate to any type of primitive value or object.

If you use an object of class `Throwable` for the second expression in an `assert` statement, it becomes the *cause* of the thrown `AssertionError`. Here's an example:

```
public class DefineCauseOfAssertionError {
    public static void main(String args[]) {
        assert (false) : new FileNotFoundException("java.txt missing");
    }
}
```

Assuming that assertions are enabled, the preceding example gives the following output:

```
Exception in thread "main" java.lang.AssertionError:
java.io.FileNotFoundException: java.txt missing
    at DefineCauseOfAssertionError.main(DefineCauseOfAssertionError.java:4)
Caused by: java.io.FileNotFoundException: java.txt missing
```

It's time for our next "Twist in the Tale" exercise. Let me check whether you've been able to retain a few concepts discussed in previous chapters.

Twist in the Tale 6.4

Let's modify some of the code used in the previous examples. Which answer correctly shows the output of class `AssertionTwist`?

```
public class AssertionTwist {
    public static void main(String args[]) {
        evenOdd(-11);
    }
    static void evenOdd(int num) {
        if (num%2 == 0)
            System.out.println("Even");
        else if (num%2 == 1)
            System.out.println("Odd");
        else {
            System.out.println("This should never be printed");
            assert false : new Person();
        }
    }
}
class Person {
    private String toString () {
        return "Pirates of the Caribbean";
    }
}
```

- **a** `Odd`
- **b** This should never be printed:

  `AssertionError: Pirates of the Caribbean`
- **c** This should never be printed:

  `AssertionError: Person@6b97fdOdd`
- **d** Compilation error
- **e** A runtime exception

Do you think class `AssertionTwist` will give the same output for executing `evenOdd(-10)`?

In this section, you examined the nitty-gritty of the syntax for the `assert` statement. In the next section, you'll look at how to test invariants in your code.

6.6.2 Testing invariants in your code

The exam objective specifically states, test invariants by using assertions. You can use assertions to test your assumptions about multiple types of invariants:

- Internal invariants
- Control-flow invariants
- Class invariants

Let's start with internal invariants.

INTERNAL INVARIANTS

When you implement the business logic for a method, you can make multiple assumptions about the values assigned to your variables. Here's an example, which assumes that a variable of type `State` can be assigned only the value `ON` or `OFF`:

```
enum State {ON, OFF};
public class InternalAssumption {
    private void machineState(State state) {
        switch (state) {
            case ON: System.out.println("state is ON");
                             break;
            case OFF: System.out.println("state is OFF");
                             break;
        }
    }
}
```

Let's modify the code to include this assumption using an `assert` statement:

```
enum State {ON, OFF};
public class InternalAssumptionWithAssert {
    private void machineState(State state) {
        switch (state) {
            case ON: System.out.println("state is ON");
                             break;
            case OFF: System.out.println("state is OFF");
                             break;
            default: assert false;
        }
    }
}
```

If value other than ON or OFF is assigned to state, AssertionError is thrown

If the control of flow reaches line `assert false;`, the assumption of the programmer is invalidated and the code throws an error (`AssertionError`). One of my colleagues argued that there is no need for an `assert` statement here because a variable of type `State` can't be assigned values other than `ON` or `OFF`. If you agree, what happens if more values are added to enum `State` during enhancement or maintenance of the project? In this case, the original assumptions won't hold true.

Let's take a look at another example, in which a programmer assumes that the `protocol` variable within `transmitFile()` can exist only with values `FTP`, `HTTP`, and `HTTPS`:

```
public class InternalAssumption {
    void transmitFile(String protocol, String fileName) {
        if (protocol.equals("FTP")) {
            //..code to transmit file using FTP
        }
        else if (protocol.equals("HTTP")) {
            //..code to transmit file using HTTP
        }
        else if (protocol.equals("HTTPS")) {
            //..code to transmit file using HTTPS
        }
        else {
            System.out.println("Control shouldn't reach here");
            System.out.println("Only FTP, HTTP, HTTPS supported");
        }
    }
}
```

Assumption that protocol can take only values FTP, HTTP, HTTPS

Programmers often document their assumptions (within the code), as shown in the previous example. A better alternative is to use assertions, so relevant action can be taken when these assumptions fail. Following is the modified code:

```
public class InternalAssumption {
        void transmitFile(String protocol, String fileName) {
        if (protocol.equals("FTP")) {
            //..code to transmit file using FTP
        }
        else if (protocol.equals("HTTP")) {
            //..code to transmit file using HTTP
        }
        else if (protocol.equals("HTTPS")) {
            //..code to transmit file using HTTPS
        }
        else {
            assert false:"Only FTP, HTTP, HTTPS supported";
        }
    }
}
```

Code throws AssertionError (if assertions are enabled and) if protocol takes value other than FTP, HTTP, or HTTPS

Note that there is an important difference between *unreachable code* as defined by the Java specification and *code that shouldn't be reached*, as used in the previous example. Unreachable code won't compile. For example, in class `UnreachableCode`, the statement placed after the `return` statement is unreachable and won't compile:

```
class UnreachableCode {
    void unreachableStatement() {
        return;
        System.out.println("code CANNOT reach here");
    }
}
```

Code won't compile: unreachable code.

On the other hand, *code that shouldn't be reached* is code that shouldn't execute in a method if all goes as per a programmer's assumption. Though it compiles successfully, its execution represents a flaw in the implementation of programming logic. Here's an example:

```
class CodeThatShouldNotReach {
    void unreachableStatement() {
        int a = (int)(Math.random() * 4) + 1;
        if (a>=2)
            return;
        else if (a <2)
            return;
        System.out.println("code SHOULD NOT reach here");
    }
}
```

Code compiles successfully, but its execution means flaw in implementation logic

EXAM TIP *Unreachable code* isn't the same as a programmer's assumption of *code that shouldn't be reached*. Unreachable code won't compile as per Java's rules. On the other hand, code that shouldn't be reached compiles successfully. But execution of this code means a flaw in the implementation of application logic.

Apart from internal invariants, you may make assumptions about the flow of control in your code, as discussed in the next section.

CONTROL-FLOW INVARIANTS

You can use assertions at locations that you assume control should never be reached. In method `processImage()`, a programmer assumes that an image (for an application) can be stored only at a server or on a local disk. The programmer uses a comment to state this assumption:

```
public class ControlFlowAssumption {
    void processImage(String fileName) {
        if (/* image stored on server*/) {
            // get image from server
            return;
        }
        else if (/* image stored on local disk*/){
            // get image from local system
            return;
        }
        // control should never reach here!
        // because an image can be stored
        // either on server or on local system
    }
}
```

Assumption that code control will never reach this line

Replace the preceding comments in bold with an `assert` statement as follows:

```
public class ControlFlowAssumption {
    void getImage(String fileName) {
        if (/* image stored on server*/) {
            // get image from server
            return;
        }
        else if (/* image stored on local disk*/){
            // get image from local system
            return;
        }
        assert false;
    }
}
```

Use of assert statement for control-flow invariants if control reaches here

CLASS INVARIANTS

Class invariants are methods that you use to validate the state of an object. The state of an object can change due to explicit assignment or reassignment of its fields, or due to any processing of field values. After each modification, class invariants can be used to check whether all fields have valid data.

Assume that you create a class to define and operate a data structure that stores a sorted list of students in a class, sorted on their overall performance in an academic year. Its methods to add new students or retrieve a student at a particular position might assume that the existing student list is already sorted. A class invariant might be that this list is sorted. You can place code to check this assumption in your methods. For example

```
import java.util.*;
class Student {}
class SortedStudents {
    List<Student> students = null;
    private boolean sorted() {
        // returns true if list students is sorted
        // false otherwise
    }
    public void addStudent(Student newStudent) {
        assert sorted();
        // code to add new student to list
    }
    public Student getStudent(String studentID) {
        assert sorted();
        // code to search list and retrieve matching Student object
    }
}
```

Class invariant

Using class invariant to validate object state

It's easy to confuse the appropriate and inappropriate use of assertions, and you're sure to see questions on it on the exam. In the next section, let's cover some rules that will help you answer these questions.

6.6.3 *Understanding appropriate and inappropriate uses of assertions*

The simple fact that assertions can be turned on or off, and are turned off by default, makes them inappropriate for all your assumptions. Let's work with some examples in detail.

DON'T USE ASSERTIONS TO CHECK METHOD PARAMETERS OF PUBLIC METHODS

You can't control the values of the arguments that are passed to your public methods; these values can be called by classes written by other programmers. To validate the parameters to public methods, you must use code that is guaranteed to execute. Because assertions can be turned off, you can't guarantee whether your assertion code will execute.

In the example that follows, the public method `result()` in class `DemoAssertion1` validates the arguments passed to it and throws an `IllegalArgumentException` for invalid values. It doesn't use an assertion to do so:

```
public class DemoAssertion1 {
    public double result(int score, int maxVal) {
        double resultVal = 0;
        if (score > 0 && maxVal > 0) {
            resultVal = /* some calculation */
        }
        else {
            throw new IllegalArgumentException();
        }
        return resultVal;
    }
}
```

Doesn't uses assertion to validate method parameters of a public method.

USE ASSERTIONS TO CHECK METHOD PARAMETERS OF PRIVATE METHODS

A private method can be called by the methods of the class in which it's defined. When a nonprivate method calls a private method, it usually passes validated parameter values to it. You can assert this assumption by using an `assert` statement. In the following example, method `calcResult()` uses the `assert` statement to check that both values passed to it are greater than zero. If the assertion fails, it will throw an `AssertionError` (if assertions are enabled), or else return the calculated value:

```
class DemoAssertion2 {
    private double calcResult(int score, int maxVal){
        assert(score > 0 && maxVal > 0);
        return (score / maxVal * 100);
    }
}
```

Use assert to validate method parameters of a private method. Don't use Assertions to modify variable values or state of an object.

Examine the following code:

```
public class DemoAssertion3 {
    int number = 10;
    private void inappropriateAssertCondition(State state) {
        assert (++number > 5);
        // some other code;
    }
}
```

❶ **Increments variable value to evaluate assert expression.**

The code in bold at ❶ increments the value of the `number` variable, and then determines whether its value is greater than 5. This line of code will *silently increment* the value of the instance variable `number` whenever method `inappropriateAssertCondition()` executes on any instance of class `DemoAssertion3`. Depending on whether assertions are enabled on the host system, the same code might output different values, which might not be expected.

Similarly, you shouldn't modify the state of an object in an `assert` statement:

```
public class DemoAssertion4 {
    int number = 10;
    MyClass myClass = new MyClass();
    private void inappropriateAssertCondition(State state) {
        assert (++number > 5): myClass.modifyDescription
                                        ("assertion error");
        // some other code;
    }
}
```

Modifies state of object myClass.

```
class MyClass {
    String description = "No error";
    public String modifyDescription(String val) {
        description = val;
        return description;
    }
}
```

In the preceding example, *expression2* (`myClass.modifyDescription(..)`) passed to the `assert` statement modifies the state of object `myClass`, which it shouldn't.

Don't depend on assertions to make your code function correctly

As mentioned previously, assertions can be turned on or off by the host machine on which a piece of Java code is supposed to execute. Also, assertions are turned off by default. So it's never advisable to define code, using assertions, which you must execute for correct functioning of your code.

Use assertions for situations that can never occur (even in public methods)

Use assertions for situations that can never occur, even in public methods. Use assertions to test invariants—internal invariants, control-flow invariants, or class invariants—in your code (discussed in detail in section 6.6.2).

By default, assertions are disabled, because they're meant to test and verify your code during the development and testing phases. But they can be easily enabled when the program starts. You, as a programmer, can test your code during the development and testing phase and execute the same code without the assertions enabled (and the related extra overhead) on the production server. Of course, if the code on the production server behaves unexpectedly, you can enable assertions to determine whether any of your logic is flawed, which is negating your assumptions.

EXAM TIP Assertions can be turned on or off for specific classes or packages.

You can enable assertions for a class, say, `DemoAssertion`, by using either of the following commands:

```
java –ea  DemoAssertion
java –enableassertions DemoAssertion
```

All your `assert` statements are equivalent to blank statements, if assertions are disabled. For example, if assertions are disabled, the following method

```
private static double calcResult(int score, int maxVal) {
    assert (score > 0 && maxVal > 0);
    return (score/maxVal*100);
}
```

will execute as if no `assert` statement were defined in it, as follows:

```
private static double calcResult(int score, int maxVal) {
    return (score/maxVal*100);
}
```

Similarly, you can use the command options `-da` and `-disableassertions` to disable assertions.

6.7 Summary

Exception handling is a mechanism for handling errors and exceptional conditions that arise during the course of a program. It also helps you define exception-handling code separate from the main application logic. This chapter specifically covers the `throw` statement and the `throws` clause, the `try` statement with multi-`catch` and `finally` clauses, the `try`-with-resources statement, and custom exceptions and assertions.

I started this chapter by covering the `throw` statement and the `throws` clause. The `throws` clause is added to a method declaration to specify that a method can throw any of the listed exceptions (or its subclasses) during its execution. When an exceptional condition is encountered, a method might handle the exception itself, or *throw* it to the calling method, by using the `throw` statement within a method.

You can create custom exceptions by extending class `java.lang.Exception` or any of its derived classes. The name of an exception can convey a lot of information to other developers or users, and this is one of the main reasons for defining a custom exception. Custom exceptions also help you restrict the escalation of implementation-specific exceptions to higher layers. For example, you can wrap a data connection exception in your own custom exception and rethrow the exception. You can create custom checked exceptions by subclassing `java.lang.Exception` and a custom runtime exception by subclassing `java.lang.RuntimeException`. You can throw and handle custom exceptions like regular exceptions.

Starting with Java 7, you can handle multiple exceptions in the same handler by using a `try` statement with multi-`catch`. This prevents a programmer from duplicating the code required to handle multiple exceptions or to *catch* a much generalized exception. You need to follow the rules to catch multiple exceptions in a single block:

- Exception names are separated by a vertical bar in the `catch` block.
- The variable that accepts the exception object in the `catch` handler is implicitly final.
- Exceptions can't share an inheritance relationship.
- You can use `multi`-catch and single-`catch` blocks for the same `try` block.

Prior to Java 7, developers used `finally` blocks to close resources such as file handlers, databases, or network connections. With the new language feature of the `try`-with-resources statement, you can declare resources and automatically close them. The scope of the resources declared with a `try`-with-resources statement is limited to it. Resources (classes) that implement the `java.lang.AutoCloseable` interface or any of its subinterfaces can be used with the `try`-with-resources statement. If multiple resources are declared by a `try`-with-resources statement, they're closed in the *reverse* order of their declaration.

Assertions enable you to test your assumptions about the execution of your code. Assertions are coded by using the `assert` statement. They're used for testing and debugging your code, and so are turned off by default. This also ensures that `assert` statements don't become a performance liability when an application is deployed. An `assert` statement takes two forms: the simpler form uses just a `boolean` expression, and the second form uses a `boolean` expression with an expression, which results in any value. If the `boolean` expression in an `assert` statement evaluates to `false`, an `AssertionError` is thrown. If an `assert` statement uses its second form, its expression is passed to the constructor of `AssertionError`, so that its details are displayed in the call stack when this error is thrown. You can use assertions to test internal invariants, control-flow invariants, and class invariants. Internal invariants are used to assert values in a loop, conditional statements, or a `switch-case`. Control-flow invariants are used to *assert* the business logic implemented in code, to *assert* that a particular line of code won't execute for a set of values. Class assertions are used to *assert* the state of all objects of a class, before or after a method executes on its object. You can also use assertions to validate parameter values to a private method. You must not use assertions to verify method parameters passed to public methods or to define code that must execute for correct functioning of your code. Because assertions can be enabled and disabled, and are usually disabled by default, you can't be sure about execution of your assertions code.

REVIEW NOTES

This section lists the main points covered in this chapter.

Using the throw statement and the throws clause

- The `throws` clause is part of a method declaration and lists exceptions that can be thrown by a method.
- The `throws` clause is used with a method declaration to specify that the method won't handle the mentioned exception (or subclasses) and might throw it to the calling method. The calling method should handle this thrown exception appropriately or declare it to be rethrown.
- The `throw` statement is used to throw an exception from a method, constructor, or an initialization block. When an exceptional condition occurs in a method, that method can handle it (by using a `try` statement), or throw the exception to the calling method by using the `throw` statement.
- A method indicates that it throws a checked exception by including its name in the `throws` clause, in its method declaration.
- When you use a method that throws a checked exception, you must either enclose the code within a `try` block or declare it to be rethrown in the calling method's declaration. This is also known as the handle-or-declare rule.
- A method can throw a runtime exception or error irrespective of whether its name is included in the `throws` clause.

- A method can throw a subclass of the exception mentioned in its `throws` clause but not a superclass.
- A method can handle the exception and still declare it to be thrown.
- A method can declare to throw any type of exception, checked or unchecked, even if it doesn't. But a `try` block can't define a `catch` block for a checked exception (other than `Exception`) if the `try` block doesn't throw that checked exception or use a method that declares to throw that checked exception.

Custom exceptions

- You can create a custom exception by extending class `java.lang.Exception` or any of its subclasses.
- You can subclass `java.lang.Exception` or its subclasses (which don't subclass `RuntimeException`) to create custom checked exceptions.
- You can subclass `java.lang.RuntimeException` or its subclasses to create custom runtime exceptions.
- You can add variables and methods to a custom exception class, like a regular class.
- The name of an exception can convey a lot of information to other developers or users, which is one of the main reasons for defining a custom exception. A custom exception can also be used to communicate a more descriptive message.
- Custom exceptions help you restrict the escalation of implementation-specific exceptions to higher layers. For example, `SQLException` thrown by data access code can be wrapped within a custom exception and rethrown.
- You can throw and catch custom exceptions like the other exception classes.

Overriding methods that throw exceptions

- With overriding and overridden methods, it's all about which checked exceptions an overridden method and an overriding method declare, not about the checked exceptions both actually throw.
- If a method in the base class doesn't declare to throw any checked exception, the overriding method in the derived class can't throw any checked exception.
- If a method in a base class declares to throw a checked exception, the overriding method in the derived class can choose not to declare to throw any checked exception.
- If a method in a base class declares to throw a checked exception, the overriding method in the derived class can declare to throw the same exception or a subclass of the exception thrown by the method in the base class. An overriding method in the derived class can't override a method in the base class, if it declares to throw a more generic checked exception.
- Method overriding rules apply only to checked exceptions. They don't apply to runtime (unchecked) exceptions or errors.

try statement with multi-catch and finally clauses

- A multi-catch handler can be used to handle more than one unrelated exception.
- To *catch* multiple exceptions in a single handler, separate the exceptions in a list by using a vertical bar (|).
- A finally block can follow a multi-catch block, like a regular catch block.
- The exceptions that you catch in a multi-catch block can't share an inheritance relationship. If you try to do so, your code won't compile.
- You can combine multi-catch and single-catch blocks.
- The same rules apply when combining multi-catch and single-catch blocks—that is, more specific exceptions at the top and more general ones at the bottom.
- You must define a single exception variable in the multi-catch block.
- In a multi-catch block, the variable that accepts the exception object is implicitly final.
- In a multi-catch block, the type of variable that accepts the exception object is the most specific, common super-type of all featured exception classes. Most of the time it's likely to be Exception, but it could be more specific (for example, IOException for classes FileNotFoundException and EOFException). If the exception classes implement a common interface, then the variable is of an intersection type, with the exception class and interface as its bounds.
- Multi-catch blocks save you from duplicating code, if you need to execute the same code for handling multiple exceptions.

Auto-close resources with try-with-resources statement

- You can use a try-with-resources statement to define resources with a try statement that will be automatically closed after the try block completes its execution.
- The try-with-resources is a type of try statement that can declare one or more resources.
- A resource is an object such as file handlers, databases, or network connections, which *should* be closed after it's no longer required.
- A resource must implement the java.lang.AutoCloseable interface or any of its subinterfaces (directly or indirectly) to be eligible to be declared by a try-with-resources statement.
- The java.lang.AutoCloseable interface defines method close().
- If declared within the try-with-resources statement, a resource is automatically closed by calling its close() method at the end of the try block.
- If method close() throws any exception, it should be taken care of by the method that defines the try block; the method must either catch it or declare it to be thrown.
- A try-with-resources block might not be followed by a catch or a finally block. This is unlike a regular try block, which must be followed by either a catch or a finally block.

- The resource declared by `try`-with-resources is closed immediately after the completion of the `try` block. Its scope is limited to the `try` block, and if you try to access it outside the `try` block, your code won't compile.
- The variables used to refer to resources are implicitly final variables. You must *declare* and *initialize* resources in the `try`-with-resources statement.
- It's acceptable to the Java compiler to initialize the resources in a `try`-with-resources statement to `null`, only as long as they aren't being assigned a value in the `try` block.
- You can initialize multiple resources in a `try`-with-resources statement, separated by a semicolon (;). The semicolon after the last resource is optional.
- Multiple classes like `FileInputStream` and `FileOutputStream` from file I/O implement the `java.io.Closeable` interface, which extends the `java.lang.AutoCloseable` interface.

Assertions

- An *assertion* offers a way of asserting what should always be `true`.
- An assertion is implemented by using an `assert` statement that enables you to test your assumptions about the values assigned to variables and the flow of control in your code.
- An `assert` statement uses a `boolean` expression, which you believe to be true. If this `boolean` expression evaluates to `false`, an `AssertionError` is thrown (if assertions are enabled).
- Assertions are used for testing and debugging your code. They're off by default.
- Assertions are disabled by default so they don't become a performance liability in deployed applications.
- An assertion is defined using an `assert` statement that can take two forms. The simpler form uses only a `boolean` expression: `assert <boolean expression>;`.
- The longer form of an `assert` statement includes an expression with the `boolean` expression: `assert <boolean expression>:<expression>;`. The second expression used here must return a value (of any type).
- In the longer form of an `assert` statement, when the `boolean` expression evaluates to `false`, the JRE creates an object of `AssertionError` by passing the value of the second expression to `AssertionError`'s constructor.
- In the longer form of an `assert` statement, if you use a method with no return value (`void`) for the second expression, your code won't compile.
- In the longer form of an `assert` statement, you can create an object for the second expression. Note that a constructor creates and returns an object, and so it satisfies the requirement that the second expression must return a value.
- You can test multiple types of invariants in your code by using assertions: internal invariants, control-flow invariants, and class invariants.
- An assertion is used to verify that code that shouldn't execute, never executes.

- Assertions can't be used to verify unreachable code, because unreachable code doesn't compile.
- You can use assertions at locations that you assume control should never be reached.
- You must not use assertions to check method parameters for public methods.
- You can use assertions to check method parameters for private methods.
- You must not use assertions to modify variable values or the state of an object.
- Assertions can be enabled or disabled at the launch of a program.
- Because assertions can be enabled or disabled, don't use them to define code that must execute in all cases.
- Use the command-line option `-ea` or `-enableassertions` to enable assertions.
- Use the command-line option `-da` or `-disableassertions` to disable assertions.
- All `assert` statements are equivalent to blank statements, if the assertions are disabled.
- A generalized `-da` switch (no assertions enabled) corresponds to the default JRE behavior.

SAMPLE EXAM QUESTIONS

Q 6-1. Consider the following code and then select the correct options. (Choose all that apply.)

```
class ThrowException {
    public void readFile(String file) throws IOException {        // line1
        System.out.println("readFile");                           // line2
    }
    void useReadFile(String name) throws IOException{             // line3
        try {
            readFile(name);
        }
        catch (IOException e) {
            System.out.println("catch-IOException");
        }
    }
    public static void main(String args[]) throws Throwable{      // line4
        new ThrowException().useReadFile("foo");                  // line5
    }
}
```

a Code on both line 1 and line 2 causes compilation failure.

b Code on either line 1 or line 2 causes compilation failure.

c If code on line 1 is changed to the following, the code will compile successfully:

```
public void readFile(String file) {d)
```

d A change in code on line 3 can prevent compilation failure:

```
void useReadFile(String name) {
```

- e Code on either line 4 or line 5 causes compilation failure. If line 4 is changed to the following, a `ThrowException` will compile:

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

Q 6-2. Given the following line of code

```
String s = "assert";
```

which of the following code options will compile successfully? (Choose all that apply.)

- a `assert(s == null : s = new String());`
- b `assert s == null : s = new String();`
- c `assert(s == null) : s = new String();`
- d `assert(s.equals("assert"));`
- e `assert s.equals("assert");`
- f `assert s == "assert" ; s.replace('a', 'z');`
- g `assert s = new String("Assert") : s.toString();`
- h `assert s == new String("Assert") : System.out.println(s);`
- i `assert(s = new String("Assert") : System.out.println(s));`

Q 6-3. What's the output of the following code?

```
class Box implements AutoCloseable {
    Box() {
        throw new RuntimeException();
    }
    public void close() throws Exception {
        System.out.println("close");
        throw new Exception();
    }
}
class EJavaFactory{
    public static void main(String args[]) {
        try (Box box = new Box()) {
            box.close();
        }
        catch (Exception e) {
            System.out.println("catch:"+e);
        }
        finally {
            System.out.println("finally");
        }
    }
}
```

- a
```
catch:java.lang.RuntimeException
close
finally
```
- b
```
catch:java.lang.RuntimeException
finally
```

c
```
catch:java.lang.RuntimeException
close
```

d
```
close
finally
```

e Compilation exception

Q 6-4. Paul has to modify a method that throws a `SQLException` when the method can't find matching data in a database. Instead of throwing a `SQLException`, the method must throw a custom exception, `DataException`. Assuming the modified method is defined as follows, which option presents the most appropriate definition of `DataException`?

```
void accessData(String parameters) {
    try {
        //..code that might throw SQLException
    }
    catch (SQLException e) {
        throw new DataException("Error with Data access");
    }
}
```

a
```
class DataException extends Exception {
    DataException() {super();}
    DataException(String msg) { super(msg); }
}
```

b
```
class DataException extends RuntimeException {
    DataException() {}
    DataException(String msg) {}
}
```

c
```
class DataException {
    DataException() {super();}
    DataException(String msg) { super(msg); }
}
```

d
```
class DataException extends Throwable {
    DataException() {super();}
    DataException(String msg) { super(msg); }
}
```

Q 6-5. Which of the following options show appropriate usage of assertions? (Choose all that apply.)

```
// INSERT CODE HERE
    assert (b != 0) : "Can't divide with zero";
    return (a/b);
}
```

a `public float divide(int a, int b) {`

b `public static float divide(int a, int b) {`

c `private static float divide(int a, int b) {`

d `private float divide(int a, int b) {`

Q 6-6. Which options can be inserted at `//INSERT CODE HERE` so the code compiles successfully? (Choose all that apply.)

```
class BreathingException extends Exception {}
class DivingException extends Exception {}

class Swimmer {
    public void swim() throws BreathingException {}
    public void dive() throws DivingException {}
}

class Swimming{
    public static void main(String args[])throws
                                 BreathingException, DivingException {
        try {
            Swimmer paul = new Swimmer();
            paul.swim();
            paul.dive();
        }
        //INSERT CODE HERE
    }
}
```

a
```
catch(DivingException | BreathingException e){
    throw e;
}
```

b
```
catch(Exception e){
    throw e;
}
```

c
```
catch(DivingException | BreathingException e){
    throw new DivingException();
}
```

d
```
catch(Exception e){
    throw new Exception();
}
```

e
```
catch(Exception e){
    throw new RuntimeException();
}
```

Q 6-7. Selvan defines two custom exception classes:

```
class BaseException extends IOException {}
class DerivedException extends FileNotFoundException {}
```

When Paul tries to use these exception classes in his own interfaces, the code doesn't compile:

```
interface Base {
    void read() throws BaseException;
}
interface Derived extends Base {
    void read() throws DerivedException;
}
```

Which definition of `read()` in the `Derived` interface compiles successfully?

- **a** `void read() throws FileNotFoundException;`
- **b** `void read() throws IOException;`
- **c** `void read();`
- **d** `void read() throws Exception;`
- **e** `void read() throws BaseException;`
- **f** `void read() throws RuntimeException;`
- **g** `void read() throws Throwable;`

Q 6-8. Given the following code, select the correct options:

```
class AssertTest {
    static int foo = 10;
    static boolean calc() {
        ++foo;
        return false;
    }
    public static void main(String args[]) {
        assert (calc());
        System.out.println(foo);
    }
}
```

- **a** If `AssertTest` is executed using the following command, it will print 11:

  ```
  java -enable AssertTest
  ```

- **b** If `AssertTest` is executed using the following command, it will print 10:

  ```
  java -ea AssertTest
  ```

- **c** If `AssertTest` is executed using the following command, it will print 10:

  ```
  java -da AssertTest
  ```

- **d** If `AssertTest` is executed using the following command, it will throw an `AssertionError` and print `11`:

  ```
  java -enableAssertions AssertTest
  ```

- **e** None of the above

Q 6-9. What's the output of the following code?

```
class Box implements AutoCloseable {
    public void emptyContents() {
        System.out.println("emptyContents");
    }
    public void close() {
        System.out.println("close");
    }
}
```

```
class EJavaFactory{
    public static void main(String args[]) {
        try (Box box = new Box()) {
            box.close();
            box.emptyContents();
        }
    }
}
```

a
```
close
emptyContents
java.lang.RuntimeException
```

b
```
close
java.lang.RuntimeException
```

c
```
close
emptyContents
close
```

d
```
close
java.lang.NullPointerException
```

Q 6-10. Which of the following statements are correct? (Choose all that apply.)

- a You must initialize the resources in the `try`-with-resources statement.
- b `try`-with-resources can be followed only by the `finally` block.
- c Method `close()` on the resource is called irrespective of whether an exception is thrown during its initialization.
- d If the `close` method is called on a resource within a `try` block, the implicit call on `close()` will throw an exception.
- e The resources declared with `try`-with-resources are accessible within the `catch` and `finally` blocks, if an exception is thrown during implicit closing of the resources.

Q 6-11. What's the output of the following code?

```
class Box implements AutoCloseable {
    public void open() throws Exception {
        throw new Exception();
    }
    public void close() throws Exception {
        System.out.println("close");
        throw new Exception();
    }
}
class EJavaFactory{
    public static void main(String args[]) {
        try (Box box = new Box()) {
            box.open();
        }
        catch (Exception e) {
            System.out.println("catch:"+e);
        }
        finally {
            System.out.println("finally");
        }
    }
}
```

a
```
catch:java.lang.Exception
finally
```

b
```
catch:java.lang.Exception
```

c
```
close
finally
```

d
```
close
catch:java.lang.Exception
finally
```

e
```
close
catch:java.lang.Exception
catch:java.lang.Exception
finally
```

Q 6-12. Select the correct statements. (Choose all that apply.)

```
class Assert {
    public void construction(double load){
        double maxLoad = 1322976;
        // calculations to re-assign a value to maxLoad
        try {
            if (maxLoad > 18753) assert false;    // line1
        }
        catch (AssertionError e) {
            // log error
            // recalculate load
        }
     }
}
```

a When assertions fail, you should avoid recalculating variable values in the error handler.

b Code on line1 can throw an `AssertionException`.

c Class `Assert` shows inappropriate use of assertions.

d Class `Assert` won't compile.

e Class `Assert` will throw a runtime error.

Q 6-13. What's the output of the following code?

```
class Box implements AutoCloseable {
    public void close() throws Exception {
        System.out.println("close");
        throw new Exception();
    }
}
class EJavaFactory{
    public static void main(String args[]) {
        try (Box box = new Box()) {
            box.close();
            box.close();
        }
        catch (Exception e) {
            System.out.println("catch:"+e);
        }
        finally {
            System.out.println("finally");
        }
    }
}
```

a
```
catch:java.lang.Exception
```

b
```
close
catch:java.lang.Exception
finally
```

c
```
close
close
catch:java.lang.Exception
finally
```

d
```
close
close
catch:java.lang.Exception
catch:java.lang.Exception
finally
```

e
```
close
close
java.lang.CloseException
finally
```

f
```
close
java.lang.RuntimeException
finally
```

Q 6-14. Examine the classes defined as follows and select the correct options. (Choose all that apply.)

```
class Jumpable extends RuntimeException {}
class Closeable extends java.lang.Closeable{}
class Thunder extends Throwable {}
class Storm extends java.io.FileNotFoundException{}
```

a The classes `Jumpable` and `Closeable` are unchecked exceptions.

b It isn't mandatory to include the name of exception `Jumpable` in the `throws` clause of a method.

- c You shouldn't define a custom exception class the way class `Thunder` has been defined.
- d Class `Storm` is a checked exception.
- e All these classes will not compile successfully.

Q 6-15. Assuming that assertions are enabled, what's the output of the following code?

```
class MyAssertClass {
    public static void main(String... args) {
        String name = new String("Shreya");
        boolean fail = false;
        if (name == "Shreya") {
            System.out.println(name);
        }
        else {
            // I explicitly set the name to Shreya
            // code cannot reach here
            assert false;
        }
    }
}
```

- a No output
- b `java.lang.AssertionError`
- c `java.lang.Exception`
- d `java.lang.RuntimeException`
- e `java.lang.AssertionException`
- f None of the above

Q 6-16. What's the output of the following code?

```
class Admission implements AutoCloseable{
    String id;
    Admission(String id) {
        this.id = id;
    }
    public void close() {
        System.out.println("close:"+id);
    }
}
class CloseableResources {
    public static void main(String... args) {
        try (Admission admn1 = new Admission("2765");
            Admission admn2 = new Admission("8582");){}
    }
}
```

a
```
close:8582
close:2765
```

b
```
close:2765
close:8582
```

c Compilation error

d Runtime exception

Q 6-17. Which of the options, when inserted at `//INSERT CODE HERE`, will print `close`? (Choose all that apply.)

```
//INSERT CODE HERE
    public void close() {
        System.out.println("close");
    }
}
class EJavaFactory2 {
    public static void main(String... args) {
        try (Carton  arton = new Carton()) {}
    }
}
```

a `class Carton implements AutoCloseable{`

b `class Carton implements java.lang.Closeable{`

c `class Carton implements java.lang.AutoCloseable{`

d `class Carton implements ImplicitCloseable{`

e `class Carton implements java.io.Closeable{`

ANSWERS TO SAMPLE EXAM QUESTIONS

A 6-1. f

[6.1] Use throw and throws statements

Explanation: The code as is (without any changes) compiles successfully. Options (a) and (b) are incorrect because a method (`readFile()`) can declare a checked exception (`IOException`) to be thrown in its `throws` clause, even if the method doesn't throw it.

Option (c) is incorrect. If line 1 is changed so that method `readFile()` doesn't declare to throw an `IOException`, method `useReadFile()` won't compile. The `catch` block in method `useReadFile()` tries to handle `IOException`, which isn't thrown by its `try` block. This causes compilation error.

Option (d) is incorrect. Code on line 3 will compile as it is. It's okay for `useReadFile()` to handle the exception thrown by `readFile()` using `try-catch` and still declare it to be rethrown using the `throws` clause.

Option (e) is incorrect. The code compiles successfully.

Option (f) is correct. Even though `useReadFile()` handles `IOException` and doesn't actually throw it, it declares it to be rethrown in its `throws` clause. So, the method that uses `useReadFile()` must either handle the checked exception—`IOException` (or one of its superclasses)—or declare it to be rethrown. Because `Exception` is a superclass of

IOException, replacing throws Throwable with throws Exception in the declaration of method main() enables the code to compile.

A 6-2. b, c, d, e, f

[6.5] Test invariants by using assertions

Explanation: Option (a) is incorrect. For the longer form of the assert statement that uses two expressions, you can't enclose both expressions within a single parentheses.

Options (b) and (c) are correct. It's optional to include the individual expressions used in the longer form within a single parentheses.

Options (d) and (e) are correct. The shorter form of the assert statement uses only one expression, which might or might not be included within parentheses.

Option (f) is correct. The semicolon placed after the condition s == "assert" delimits the assert statement, and the statement following the semicolon is treated as a separate statement. It's equivalent to an assert statement using its shorter form followed by another statement, as follows:

```
assert s == "assert" ;
s.replace('a', 'z');
```

Option (g) is incorrect because the first expression in an assert statement must return a boolean value. In this code, the first expression returns an object of class String.

Option (h) is incorrect because the second expression in an assert statement must return a value of any type. In this code, the return type of method println() is void.

Option (i) is incorrect. It incorrectly encloses both expressions of the assert statement within a single pair of parentheses. If parentheses were removed, it's also an illegal usage of the long form because it uses an expression that doesn't return a boolean value for its first expression and its second expression doesn't return any value.

A 6-3. b

[6.3] Auto-close resources with a try-with-resources statement

Explanation: The constructor of class Box throws a RuntimeException, and so the box variable isn't initialized in the try-with-resources statement. Method close() of class Box isn't called implicitly because execution didn't proceed inside the try block.

When try-with-resources throws an exception, the control is transferred to the catch block. In this case, the exception handler prints catch:java.lang.RuntimeException. The finally block always executes, thereafter printing finally.

A 6-4. b

[6.4] Create custom exceptions

Explanation: Option (a) is incorrect because it defines custom exception `DataException` as a checked exception. To use checked `DataException`, `accessData()` must specify it to be thrown in its `throws` clause.

Option (b) is correct because it defines custom exception `DataException` as an unchecked or runtime exception. Even though method `accessData()` throws a `DataException`, a runtime exception, it need not declare its name in its `throws` clause.

Option (c) is incorrect. Class `DataException` extends class `Object` and not `Exception`. Also, this code won't compile because class `Object` doesn't define a constructor that matches `Object(String)`.

Option (d) is incorrect. If `DataException` extends `Throwable`, `accessData()` won't compile because it's also a checked exception and therefore must be handled or declared.

A 6-5. c, d

[6.5] Test invariants by using assertions

Explanation: Options (a) and (b) are incorrect because assertions must not be used to check method arguments for nonprivate methods. Nonprivate methods can be called by objects of other classes, and you can't validate their method arguments by using assertions. Assertions can be disabled at runtime, so they aren't the right option to validate method arguments to public methods. You should throw exceptions in this case. For example, when you come across invalid values that are passed to a nonprivate method, you can throw an `IllegalArgumentException`.

A 6-6. a, b, c, e

[6.1] Use throw and throws statements
[6.2] Use the try statement with multi-catch and finally clauses

Explanation: Method `swim()` throws a `BreathingException`, and method `dive()` throws a `DivingException`, both checked exceptions. Method `main()` is declared to throw both a `BreathingException` or `DivingException`.

The code that is inserted at `//INSERT CODE HERE` must do the following:

- The option must use the correct syntax of `try` with single-`catch` or multi-`catch`.
- Because method `main()` throws both a `BreathingException` and `DivingException`, the `try` statement might not handle it.
- It must not throw a checked exception more generic than the ones declared by method `main()` itself—that is, `BreathingException` and `DivingException`.

Option (a) is correct. It defines the correct syntax to use `try` with multi-`catch`, and rethrows the caught instance of `BreathingException` or `DivingException`.

Option (b) is correct. Though the type of the exception caught in the exception handler is `Exception`, the superclass of the exceptions thrown by `main()`, the compiler knows that the `try` block can throw only two checked exceptions, `BreathingException` or `DivingException`. So this thrown exception is caught and rethrown with more inclusive type checking.

Option (c) is correct. The multi-`catch` statement is correct syntactically. Also, the `throw` statement throws a checked `DivingException`, which is declared by method `main()`'s `throws` clause.

Option (d) is incorrect. The `catch` block throws an instance of checked exception `Exception`, the superclass of the exceptions declared to be thrown by `main()`, which isn't acceptable.

Option (e) is correct. It isn't obligatory to include the names of the runtime exceptions thrown by a method in its `throws` clause.

A 6-7. c, e, f

[6.1] Use throw and throws statements
[6.4] Create custom exceptions

Explanation: Though the question seems to be testing you on the hierarchy of exception classes `IOException` and `FileNotFoundException`, it's not. This question is about creating custom exception classes with overridden methods that throw exceptions.

To understand the explanation, note the following class hierarchies:

- Exception `FileNotFoundException` extends `IOException`.
- Exception `BaseException` extends `IOException`.
- Exception `DerivedException` extends `FileNotFoundException`.
- Exception `DerivedException` doesn't extend `BaseException`.

This class hierarchy implies the following:

- `BaseException` isn't a type of `DerivedException`.
- `BaseException` and `FileNotFoundException` are unrelated exceptions.

To override `read()` in the `Base` interface, `read()` in the `Derived` interface must either declare to throw a `BaseException`, any derived classes of `BaseException`, any runtime exception or errors, or no exception.

Option (a) is incorrect because `FileNotFoundException` is unrelated to `BaseException`.

Options (b), (d), and (g) are incorrect because `IOException`, `Exception`, and `Throwable` are all superclasses of `BaseException` (and not subclasses) and therefore invalid when overriding method `read()`.

A 6-8. c

[6.5] Test invariants by using assertions

Explanation: Options (a) and (d) are incorrect. `-enable` and `-enableAssertions` are invalid switch options. The correct switch options to enable assertions are `-ea` and `-enableassertions`. If you use an invalid argument (like `-enable`) the program will not run, but will exit immediately with an "Unrecognized option" error. Option (b) is incorrect. With assertions enabled, `assert(calc())` will evaluate to `assert(false)` and throw an `AssertionError`, exiting the application, before printing any values.

Option (c) is correct. With assertions disabled, the `assert (calc())` statement is equivalent to an empty statement or a nonexistent line of code. So method `calc()` isn't executed, and the value of the static variable `foo (10)` is printed.

A 6-9. c

[6.3] Auto-close resources with a try-with-resources statement

Explanation: An implicit or explicit call to method `close()` doesn't set a resource to `null`, and you can call methods on it. So the `try` block results in the following:

- Explicit call to method `close()`
- Explicit call to method `emptyContents()`
- Implicit call to method `close()`

No exceptions are thrown, and the code prints the output as shown in option (c).

A 6-10. a, b

[6.2] Use the try statement with multi-catch and finally clauses
[6.3] Auto-close resources with a try-with-resources statement

Explanation: Option (c) is incorrect. If a resource couldn't be initialized because of an exception, it doesn't exist. There's no point in auto-closing such a resource.

Option (d) is incorrect. Subsequent calls to `close()` don't throw an exception.

Option (e) is incorrect. The resources declared within a `try`-with-resources statement are accessible only within the `try` block.

A 6-11. d

[6.3] Auto-close resources with a try-with-resources statement

Explanation: The code `box.open()` within the `try` block throws an `Exception`. Before the control is transferred to the exception handler, the resource `box` is implicitly closed by calling its `close()` method, which also throws an `Exception`.

If an exception is thrown from the `try` block and one or more exceptions are thrown from the `try`-with-resources statement, then those exceptions thrown from the try-with-resources statement are suppressed. So the exception thrown by the implicit call to box.close() is suppressed. A suppressed exception forms the cause of the exception that suppresses it. It can be retrieved by using method getSuppressed() on the exception handler, as follows:

```
catch (Exception e) {
    System.out.println("catch:"+e);
    for (Throwable th : e.getSuppressed())
        System.out.println(th);
}
```

It prints the following:

```
catch:java.lang.Exception
java.lang.Exception
```

A 6-12. c

[6.5] Test invariants by using assertions

Explanation: Option (a) is incorrect, and (c) is correct. Though verifying the value of variable maxLoad using the assert statement is valid and appropriate usage of an assertion, handling an AssertionError isn't. You should never handle an AssertionError. Assertions enable you to test your assumptions while you're developing your applications. You must not try to recover from an AssertionError. If an assumption fails, you must correct the code or logic accordingly. If you foresee exceptional conditions in the production version of your application, use exceptions instead of assertions.

Option (b) is incorrect because the code on line 1 can throw an AssertionError and not an AssertionException.

Options (d) and (e) are incorrect. Class Assert will compile successfully. Note that assert (small *a*) is a keyword, and not Assert (capital *A*). When executed with assertions enabled, the assert statement might throw an AssertionError, which will be handled by the catch block. So class Assert won't throw a runtime exception and might throw an AssertionError.

A 6-13. c

[6.3] Auto-close resources with a try-with-resources statement

Explanation: This question tries to trick you with explicit calls of method close() placed in the try block. Though close() is implicitly called to close a resource, it's possible to call it explicitly in the try block. But the explicit call to close() is independent of the implicit call to close() for each resource defined in the try-with-resources statement.

The first call to method close() prints close. Because this method call throws an exception, the control is ready to be transferred to the catch block and thus the second explicit call to method close() doesn't execute. But before the control is moved to the catch block, the implicit call to method close() is made, which again prints close and throws an exception. The exception thrown by the implicit call of method close() is suppressed by the exception thrown by the explicit call of method close(), placed before the end of the try block.

The control is then transferred to the catch block and, last, to the finally block.

A 6-14. b, c, d, e

[6.4] Create custom exceptions

Explanation: Option (a) is incorrect because java.lang.Closeable is undefined. So class Jumpable won't compile.

Option (e) is a correct option because class Jumpable won't compile.

A 6-15. b

[6.5] Test invariants by using assertions

Explanation: When the assertion fails, an instance of AssertionError is thrown. The condition (name == "Shreya") evaluates to false because it compares the object references and not the String values.

A 6-16. a

[6.3] Auto-close resources with a try-with-resources statement

Explanation: The code compiles successfully, and no runtime exceptions are thrown during its execution. The resources initialized in a try-with-resources statement are closed in the reverse order.

A 6-17. a, c, e

[6.3] Auto-close resources with a try-with-resources statement

Explanation: The code will print close, if class Carton implements the java.lang.AutoCloseable interface or any of its subinterfaces. Options (a) and (c) are correct, because Carton implements the interface java.lang.AutoCloseable itself.

Option (b) is incorrect. The Closeable interface that extends the java.lang.AutoCloseable interface is defined in the java.io package.

Option (d) is incorrect. The Java API doesn't define any interface with the name ImplicitCloseable in the java.lang package.

Option (e) is correct because java.io.Closeable is a subinterface of java.lang.AutoCloseable.