```
        System.out.println(var);
    }
}
class QReference {
    public static void main(String[] args) {
        Base base = new Base();
        Base derived = new Derived();

        System.out.println(base.var);
        System.out.println(derived.var);
        base.printVar();
        derived.printVar();
    }
}
```

a
```
EJava
EJava
EJava
Guru
```

b
```
EJava
Guru
EJava
Guru
```

c
```
EJava
EJava
EJava
EJava
```

d
```
EJava
Guru
Guru
Guru
```

Answer: a

Explanation: With inheritance, the instance variables bind at compile time and the methods bind at runtime. The following line of code refers to an object of the class `Base`, using a reference variable of type `Base`. Hence, both of the following lines of code print `EJava`:

```
System.out.println(base.var);
base.printVar();
```

But the following line of code refers to an object of the class `Derived` using a reference variable of type `Base`:

```
Base derived = new Derived();
```

Because the instance variables bind at compile time, the following line of code accesses and prints the value of the instance variable defined in the class `Base`:

```
System.out.println(derived.var);    // prints EJava
```

In `derived.printVar()`, even though the method `printVar` is called using a reference of type `Base`, the JVM is aware that the method is invoked on a `Derived` object and so executes the overridden `printVar` method in the class `Derived`.

7 Exception handling

Exam objectives covered in this chapter	What you need to know
[8.3] Describe what exceptions are used for in Java.	Need for the exception handlers; their advantages and disadvantages.
[8.1] Differentiate among checked exceptions, `RuntimeExceptions`, and `Errors`.	Differences and similarities between checked exceptions, `RuntimeExceptions`, and `Errors`. Differences and similarities in the way these exceptions and errors are handled in code.
[8.2] Create a `try-catch` block and determine how exceptions alter normal program flow.	How to create a `try-catch-finally` block. Understand the flow of code when the enclosed code throws an exception or error. How to create nested `try-catch-finally` blocks.
[8.4] Invoke a method that throws an exception.	How to determine the flow of control when an invoked method throws an exception. How to apply this to cases when it's thrown without a `try` block, and from a `try` block (with appropriate and insufficient exception handlers). The difference in calling methods that throw or don't throw exceptions.
[8.5] Recognize common exception classes and categories.	Common exception classes and categories, and how to recognize the code that can throw these exceptions and handle them appropriately.

Imagine you're about to board an airplane to Geneva to attend an important conference. At the last minute, you learn that the flight has been cancelled because the pilot isn't feeling well. Fortunately, the airline quickly arranges for an alternative pilot, allowing the flight to take off at its originally scheduled time. What a relief.

This example illustrates how exceptional conditions can modify the initial flow of an action and demonstrates the need to handle those conditions appropriately. In Java, an exceptional condition (like the illness of a pilot) can affect the normal code flow (airline flight operation). In this context, the arrangement for an alternative pilot can be compared to an exception handler.

Depending on the nature of the exceptional condition, you may or may not be able to recover completely. For example, would airline management have been able to get your flight off the ground if instead an earthquake had damaged much of the airport?

In the exam, you'll be asked similar questions with respect to Java code and exceptions. With that in mind, this chapter covers the following:

- Understanding and identifying exceptions arising in code
- Determining how exceptions alter the normal program flow
- Understanding the need to handle exceptions separately in your code
- Using `try-catch-finally` blocks to handle exceptions
- Differentiating among checked exceptions, unchecked exceptions, and errors
- Invoking methods that may throw exceptions
- Recognizing common exception categories and classes

You might feel like we're covering a lot in this chapter, but remember that we aren't going to delve into too much background information because we assume you already know the definitions and uses of classes and methods, class inheritance, arrays, and ArrayLists. Our focus in this chapter will be on the exam objectives and what you need to know about exceptions.

In this chapter, I won't discuss a `try` statement with multiple `catch` clauses, automatic closing of resources with a `try`-with-resources statement, or the creation of custom exceptions. These topics are covered in the next level of Java certification (in the OCP Java SE 7 Programmer II exam).

7.1 Exceptions in Java

[8.3] Describe what exceptions are used for in Java

In this section, you'll learn what exceptions are in Java, why you need to handle exceptions separately from the main code, and all about their advantages and disadvantages.

7.1.1 A taste of exceptions

In figure 7.1, do you think the code in bold in the classes `ArrayAccess`, `OpenFile`, and `MethodAccess` have anything in common?

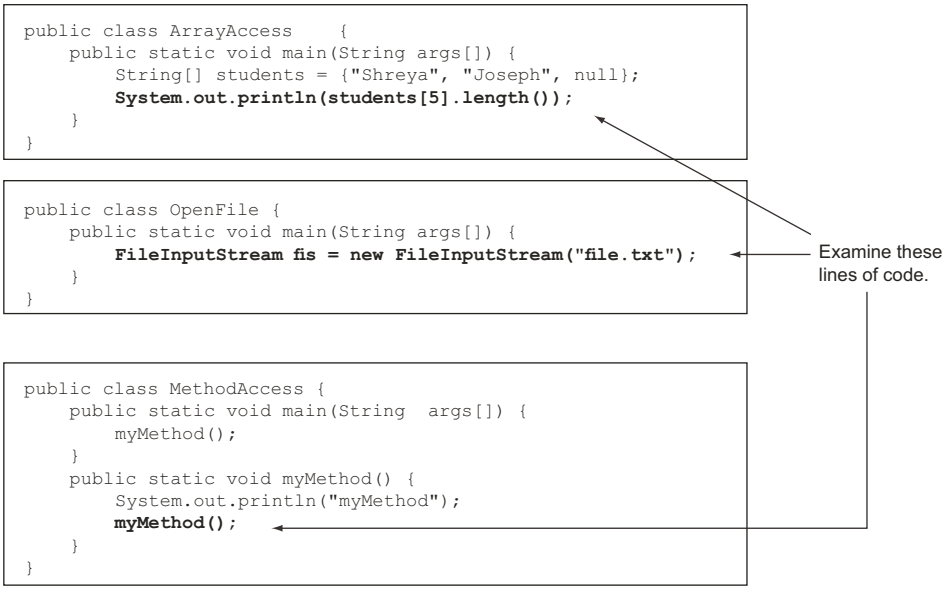


Figure 7.1 Getting a taste of exceptions in Java

I'm sure, given this chapter's title, that this question was easy to answer. Each of these three statements is associated with throwing an exception or an error. Let's look at them individually:

- *Class* `ArrayAccess`—Because the length of the array `students` is `3`, trying to access the element at array position `5` is an exceptional condition, as shown in figure 7.2.
- *Class* `OpenFile`—The constructor of the class `FileInputStream` throws a checked exception `FileNotFoundException` (as shown in figure 7.3). If you try to compile this code without enclosing it within a `try` block or *catching* this exception, your code will fail to compile. (I'll discuss checked exceptions in detail in section 7.3.2.)
- *Class* `MethodAccess`—As you can see in figure 7.4, the method `myMethod` calls itself recursively, without specifying an exit condition. These recursive calls result in a `StackOverflowError` at runtime.

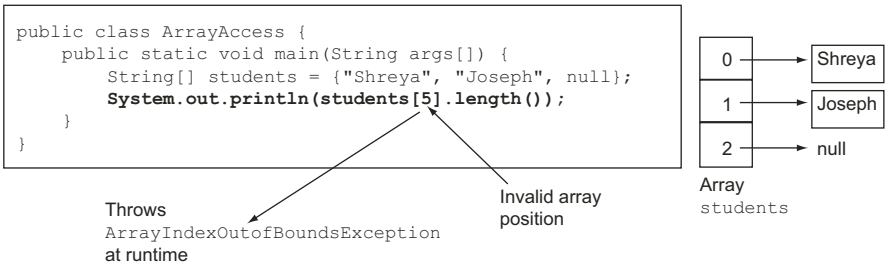


Figure 7.2 An example of `ArrayIndexOutOfBoundsException`

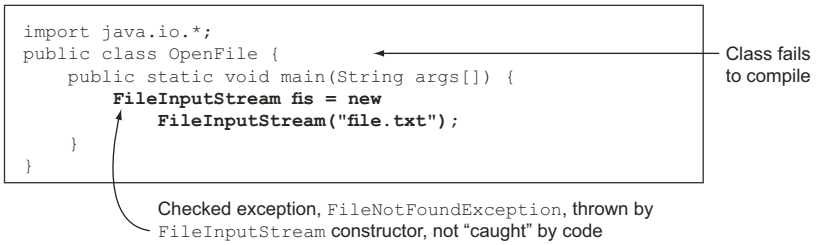


Figure 7.3 An example of `FileNotFoundException`

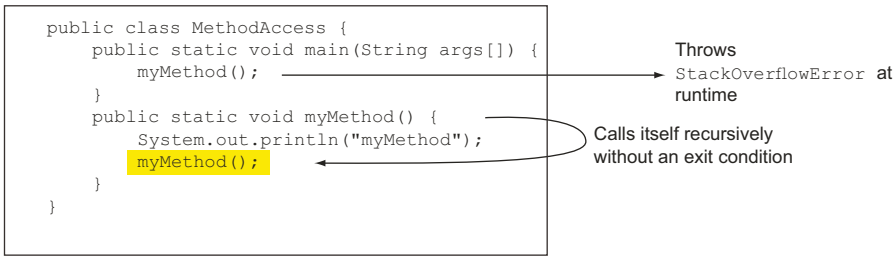


Figure 7.4 An example of `StackOverflowError`

These examples of exceptions are typical of what you'll find on the OCA Java SE 7 Programmer I exam. Let's move on and explore exceptions and their handling in Java so that you can spot code that throws exceptions and handles them accordingly.

File I/O in Java

File I/O isn't covered in the OCA Java SE 7 Programmer I exam, but you may notice it mentioned in questions related to exception handling. I'll cover it quickly here just to the extent that it's required for this exam.

File I/O involves multiple classes that enable you to read data from and write it to a source. This data source can be persistent storage, memory, or even network connections. Data can be read and written as streams of binary or character data. Some file I/O classes only read data from a source, some write data to a source, and some do both.

In this chapter, you'll work with three classes from the file I/O API: `java.io.File`, `java.io.FileInputStream`, and `java.io.FileOutputStream`. `File` is an abstract representation of file and directory pathnames. You can *open* a `File` and then read from and write to it. A `FileInputStream` obtains input bytes using an object of class `File`. It defines the methods `read` to read bytes and `close` to close this stream. A `FileOutputStream` is an output stream for writing data to a `File`. It defines the methods `write` to write bytes and `close` to close this stream.

Creating an object of class `File` can throw the checked exception `java.io.FileNotFoundException`. The methods `read`, `write`, and `close` defined in classes `FileInputStream` and `FileOutputStream` can throw the checked exception `java.io.IOException`. Note that `FileNotFoundException` subclasses `IOException`.

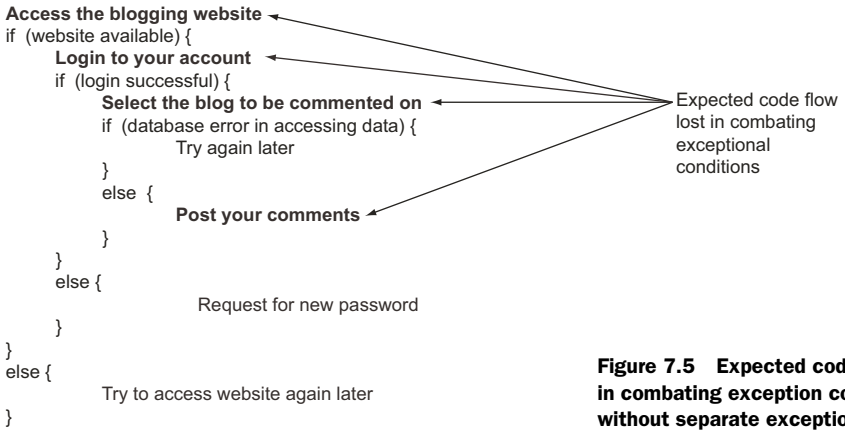


Figure 7.5 Expected code flow lost in combating exception conditions, without separate exception handlers

7.1.2 Why handle exceptions separately?

Imagine you want to post some comments on a blogging website. To make a comment, you must complete the following steps:

a Access the blogging website.
b Log into your account.
c Select the blog you want to comment on.
d Post your comments.

The preceding list might seem like an ideal set of steps. In actual conditions, you may have to verify whether you've completed a previous step before you can progress with the next step. Figure 7.5 modifies the previous steps.

The modified logic (figure 7.5) requires the code to check conditions before a user can continue with the next step. This checking of conditions at multiple places introduces new steps for users and also new paths of execution of the original steps. The difficult part of these modified paths is that they may leave users confused about the steps involved in the tasks they're trying to accomplish. Figure 7.6 shows how exception handling can help.

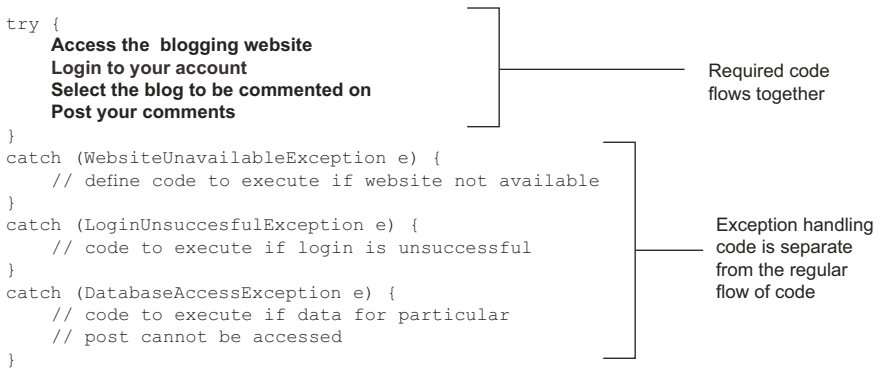


Figure 7.6 Defining exception-handling code separate from the main code logic

The code in figure 7.6 defines the original steps required to post comments to a blog, along with some exception-handling code. Because the exception handlers are defined separately, any confusion with what steps you need to accomplish to post comments on the website has been clarified. Additionally, this code doesn't compromise on checking the completion of a step before moving on to the next step, courtesy of appropriate exception handlers.

7.1.3 Do exceptions offer any other benefits?

Apart from separating concerns between defining the regular program logic and the exception handling code, exceptions also can help pinpoint the offending code (code that throws an exception), together with the method in which it is defined, by providing a stack trace of the exception or error. An example:

```
public class Trace {                                        // line 1
    public static void main(String args[]) {                // line 2
        method1();                                          // line 3
    }                                                       // line 4
    public static void method1() {                          // line 5
        method2();                                          // line 6
    }                                                       // line 7
    public static void method2() {                          // line 8
        String[] students = {"Shreya", "Joseph"};           // line 9
        System.out.println(students[5]);                    // line 10
    }                                                       // line 11
}                                                           // line 12
```

`method2()` tries to access the array element of `students` at index `5`, which is an invalid index for array `students`, so the code throws the exception `ArrayIndexOutOfBoundsException` at runtime. Figure 7.7 shows the stack trace when this exception is thrown. It includes the runtime exception message and the list of methods that were involved in calling the code that threw the exception, starting from the entry point of this application, the `main` method. You can match the line numbers specified in the stack trace in figure 7.7 to the line numbers in the code.

NOTE The stack trace gives you a trace of the methods that were called when the JVM encountered an unhandled exception. Stack traces are read from the bottom. In figure 7.7, the trace starts with the `main` method (last line of the stack trace) and continues up to the method containing the code that threw the exception. Depending on the complexity of your code, a stack trace can

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 5
    at Trace.method2(Trace.java:10)
    at Trace.method1(Trace.java:6)
    at Trace.main(Trace.java:3)
```

Offending code in `method2` (line 10)

`method2` called by `method1` (line 6)

`method1` called by `main` (line 3)

Figure 7.7 Tracing the line of code that threw an exception at runtime

range from a few lines to hundreds of lines of code. A stack trace works with handled and unhandled exceptions.

Let's move on and look at more details of exception propagation and at the creation of `try-catch-finally` blocks to take care of exceptions in code.

7.2 What happens when an exception is thrown?

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

[8.4] Invoke a method that throws an exception

In this section, we'll uncover what happens when an exception is thrown in Java. We'll work through a lot of examples to learn how the normal flow of code is disrupted when an exception is thrown. You'll also define an alternative program flow for code that may throw exceptions.

As with all other Java objects, an exception is an object. All types of exceptions subclass `java.lang.Throwable`. When a piece of code hits an obstacle in the form of an exceptional condition, it creates an object of class `java.lang.Throwable` (at runtime an object of the most appropriate subtype is created), initializes it with the necessary information (such as its type, an optional textual description, and the offending program's state), and hands it over to the JVM. The JVM blows a siren in the form of this exception and looks for an appropriate code block that can "handle" this exception. The JVM keeps an account of all the methods that were called when it hit the offending code, so to find an appropriate exception handler it looks through all of the tracked method calls.

Re-examine the class `Trace` and the `ArrayIndexOutOfBoundsException` thrown by it, as mentioned in the previous section (7.1.3). Figure 7.8 illustrates the propagation of the exception `ArrayIndexOutOfBoundsException` thrown by `method2` through all the methods.

To understand how an exception propagates through method calls, it's important to understand how method calls work. An application starts its execution with the method `main`, and `main` may call other methods. When `main` calls another method, the called method should complete its execution before `main` can complete its own execution.

An operating system (OS) keeps track of the code that it needs to execute using a *stack*. A stack is a type of list in which the items that are added last to it are the first ones to be taken off it—Last In First Out. This stack uses a *stack pointer* to point to the instructions that the OS should execute.

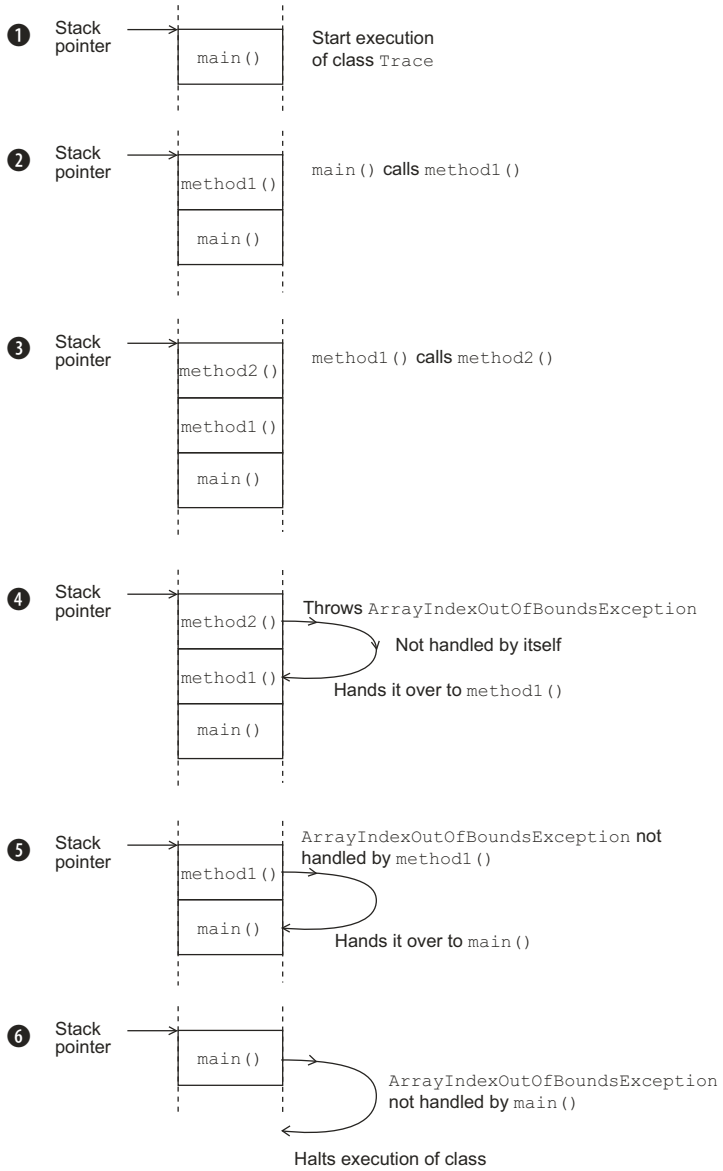


Figure 7.8 Propagation of an exception through multiple method calls

Now that you have this basic information under your belt, here's a step-by-step discussion of exception propagation, as shown in figure 7.8:

1. When the method `main` starts its execution, its instructions are pushed onto the stack.
2. The method `main` calls the method `method1`, and instructions for `method1` are pushed onto the stack.

3. `method1` calls `method2`; instructions for `method2` are pushed onto the stack.
4. `method2` throws an exception: `ArrayIndexOutOfBoundsException`. Because `method2` doesn't handle this exception itself, it's passed on to the method that called it—`method1`.
5. `method1` doesn't define any exception handler for `ArrayIndexOutOfBoundsException`, so it hands this exception over to its calling method—`main`.
6. There are no exception handlers for `ArrayIndexOutOfBoundsException` in `main`. Because there are no further methods that handle `ArrayIndexOutOfBoundsException`, execution of the class `Trace` stops.

You can use `try-catch-finally` blocks to define code that doesn't halt execution when it encounters an exceptional condition.

7.2.1 Creating try-catch-finally blocks

When you work with exception handlers, you often hear the terms `try`, `catch`, and `finally`. Before you start to work with these concepts, I'll answer three simple questions:

- *Try what?*

  First you *try* to execute your code. If it doesn't execute as planned, you handle the exceptional conditions using a *catch* block.
- *Catch what?*

  You *catch* the exceptional event arising from the code enclosed within the *try* block and handle the event by defining appropriate exception handlers.
- *What does finally do?*

  Finally, you execute a set of code, in all conditions, regardless of whether the code in the *try* block throws any exceptions.

Let's compare a `try-catch-finally` block with a real-life example. Imagine you're going river rafting on your vacation. Your instructor informs you that while rafting, you *might* fall off the raft into the river while crossing the rapids. In such a condition, you should try to use your oar or the rope thrown toward you to get back into the raft. You *might* also drop your oar into the river while rowing your raft. In such a condition, you should not panic and should stay seated. Whatever happens, you're paying for this adventure sport.

Compare this to Java code:

- You can compare river rafting to a class whose methods *might* throw exceptions.
- Crossing the rapids and rowing a raft are methods that *might* throw exceptions.
- Falling off the raft and dropping your oar are the exceptions.
- The steps for getting back into the raft and not panicking are the exception handlers—code that executes when an exception arises.
- The fact that you pay for the sport, whether you stay in the boat or not, can be compared to the `finally` block.

Let's implement the previous real-life examples by defining appropriate classes and methods. To start with, here are two barebones exception classes—`FallInRiverException` and `DropOarException`—that can be thrown by methods in the class `RiverRafting`:

```
class FallInRiverException extends Exception {}
class DropOarException extends Exception {}
```

NOTE You can create an exception of your own—a custom exception—by extending the class `Exception` (or any of its subclasses). Although the creation of custom classes is *not* on this exam, you may see questions in the exam that create and use custom exceptions. Perhaps these are included because hardly any checked exceptions from the Java API are on this exam. Coding questions on the exam may create and use custom exceptions.

Following is a definition of class `RiverRafting`. Its methods `crossRapid` and `rowRaft` may throw exceptions of type `FallInRiverException` and `DropOarException`:

```
class RiverRafting {
    void crossRapid(int degree) throws FallInRiverException {
        System.out.println("Cross Rapid");
        if (degree > 10) throw new FallInRiverException();
    }

    void rowRaft(String state) throws DropOarException {
        System.out.println("Row Raft");
        if (state.equals("nervous")) throw new DropOarException();
    }
}
```

Method crossRapid may throw ❶ FallInRiverException

Method rowRaft may throw ❷ DropOarException

Method `crossRapid` at ❶ throws the exception `FallInRiverException`. When you call this method, you should define an exception handler for this exception. Similarly, the method `rowRaft` at ❷ throws the exception `DropOarException`. When you call this method, you should define an exception handler for this exception.

When you execute methods that may throw *checked exceptions* (exceptions that don't extend the class `RuntimeException`), enclose the code within a `try` block. `catch` blocks that follow a `try` block should handle all the checked exceptions thrown by the code enclosed in the `try` block (checked exceptions are covered in detail in section 7.3).

The code shown in figure 7.9 uses the class `RiverRafting` as defined previously and depicts the flow of control when the code on line 3 (`riverRafting.crossRapid(11);`) throws an exception of type `FallInRiverException`.

The example in figure 7.9 shows how exceptions alter the normal program flow. If the code on line 3 throws an exception (`FallInRiverException`), the code on lines 4 and 5 won't execute. In this case, control is transferred to the code block that handles `FallInRiverException`. Then control is transferred to the `finally` block. After the execution of the `finally` block, the code that follows the `try-catch-finally` block is executed. The output of the previous code is:

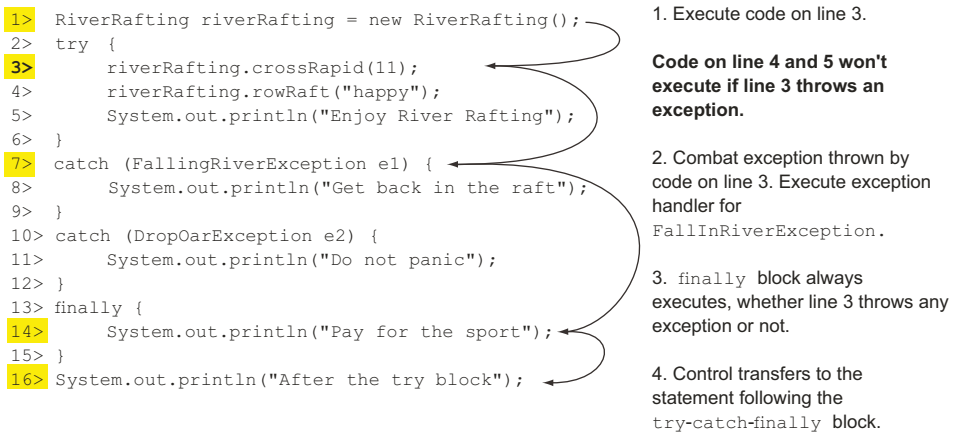


Figure 7.9 Modified flow of control when an exception is thrown

```
Cross Rapid
Get back in the raft
Pay for the sport
After the try block
```

If you modify the previous example code as follows, no exceptions are thrown by the code on line 3 (modifications in bold):

```
class TestRiverRafting {
    public static void main(String args[]) {
        RiverRafting riverRafting = new RiverRafting();
        try {
            riverRafting.crossRapid(7);
            riverRafting.rowRaft("happy");
            System.out.println("Enjoy River Rafting");
        }
        catch (FallInRiverException e1) {
            System.out.println("Get back in the raft");
        }
        catch (DropOarException e2) {
            System.out.println("Do not panic");
        }
        finally {
            System.out.println("Pay for the sport");
        }
        System.out.println("After the try block");
    }
}
```

No exceptions thrown by this line of code

The output of the previous code is as follows:

```
Cross Rapid
Row Raft
Enjoy River Rafting
Pay for the sport
After the try block
```

What do you think the output of the code would be if the method `rowRaft` threw an exception? Try it for yourself!

EXAM TIP The `finally` block executes regardless of whether the `try` block throws an exception.

SINGLE TRY BLOCK, MULTIPLE CATCH BLOCKS, AND A FINALLY BLOCK

For a `try` block, you can define multiple `catch` blocks, but only a single `finally` block. Multiple `catch` blocks are used to handle different types of exceptions. A `finally` block is used to define *cleanup* code—code that closes and releases resources, such as file handlers and database or network connections.

When it comes to code, it makes sense to verify a concept by watching it in action. Let's work through a simple example so that you can better understand how to use the `try-catch-finally` block.

In listing 7.1, the constructor of the class `FileInputStream` may throw a `FileNotFoundException`, and calling the method `read` on an object of `FileInputStream`, such as `fis`, may throw an `IOException`.

Listing 7.1 Code flow with multiple `catch` statements and a `finally` block

```
import java.io.*;
public class MultipleExceptions {
    public static void main(String args[]) {
        FileInputStream fis = null;
        try {
            fis = new FileInputStream("file.txt");
            System.out.println("File Opened");
            fis.read();
            System.out.println("Read File ");
        }
        catch (FileNotFoundException fnfe) {
            System.out.println("File not found");
        }
        catch (IOException ioe) {
            System.out.println("File Closing Exception");
        }
        finally {
            System.out.println("finally");
        }
        System.out.println("Next task..");
    }
}
```

May throw FileNotFoundException

May throw IOException

Positioning of catch and finally blocks can't be interchanged

Table 7.1 compares the code output that occurs depending on whether the system is able or unable to open (and read) file.txt.

In either of the cases described in table 7.1, the `finally` block executes, and after its execution, control is transferred to the statement following the `try-catch` block. Here's the output of the class `MultipleExceptions` if none of its code throws an exception:

Table 7.1 Output of code in listing 7.1 when the system is unable to open file.txt and when the system is able to open file.txt but unable to read it

Output if the system is unable to open file.txt	Output if the system is able to open file.txt, but unable to read it
File not found finally Next task..	File Opened File Closing Exception finally Next task..

```
File Opened
Read File
finally
Next task..
```

It's time now to attempt this chapter's first Twist in the Tale exercise. When you execute the code in this exercise, you'll understand what happens when you change the placement of the exception handlers (answers are in the appendix).

Twist in the Tale 7.1

Let's modify the placement of the `finally` block in listing 7.1 and see what happens.

Given that file.txt doesn't exist on your system, what is the output of the following code?

```
import java.io.*;
public class MultipleExceptions {
    public static void main(String args[]) {
        FileInputStream fis = null;
        try {
            fis = new FileInputStream("file.txt");
            System.out.println("File Opened");
            fis.read();
            System.out.println("Read File");
        }
        finally {
            System.out.println("finally");
        }
        catch (FileNotFoundException fnfe) {
            System.out.println("File not found");
        }
        catch (IOException ioe) {
            System.out.println("File Closing Exception");
        }
        System.out.println("Next task..");
    }
}
```

a The code prints

```
File not found
finally
Next task..
```

b The code prints

```
File Opened
File Closing Exception
finally
Next task..
```

c The code prints `File not found`

d The code fails to compile

In the remainder of this section, we'll look at some frequently asked questions on `try-catch-finally` blocks that often overwhelm certification aspirants.

7.2.2 Will a finally block execute even if the catch block defines a return statement?

Image the following scenario: a guy promises to buy diamonds for his girlfriend and treat her to coffee. The girl inquires about what will happen if he meets with an exceptional condition during the diamond purchase, such as inadequate funds. To the girl's disappointment, the boy replies that he'll still treat her to coffee.

You can compare the `try` block to the purchase of diamonds and the `finally` block to the coffee treat. The girl gets the coffee treat regardless of whether the boy successfully purchases the diamonds. Figure 7.10 shows this conversation.

It is interesting to note that a `finally` block will execute even if the code in the `try` block or any of the `catch` blocks defines a `return` statement. Examine the code in figure 7.11 and its output, and note when the class `ReturnFromCatchBlock` is unable to open file.txt:

As you can see from figure 7.11's code output, the flow of control doesn't return to the method `main` when the `return` statement executes in the `catch` handler of `FileNotFoundException`. It continues with the execution of the `finally` block before the control is transferred back to the `main` method. Note that the control isn't transferred

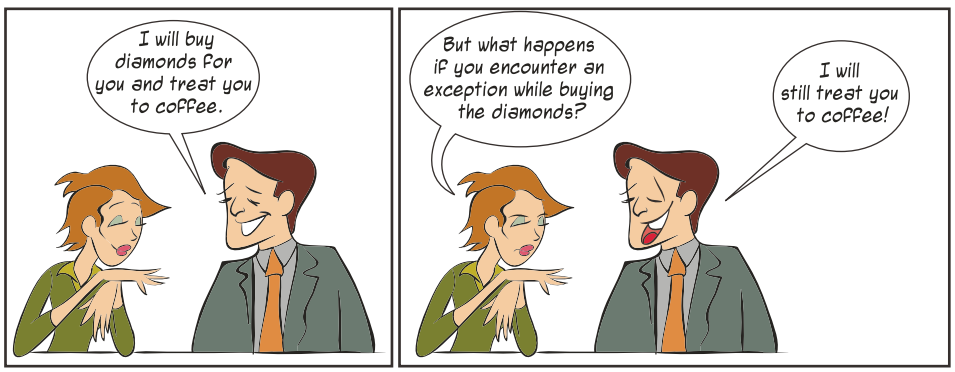


Figure 7.10 A little humor to help you remember that a `finally` block executes regardless of whether an exception is thrown.

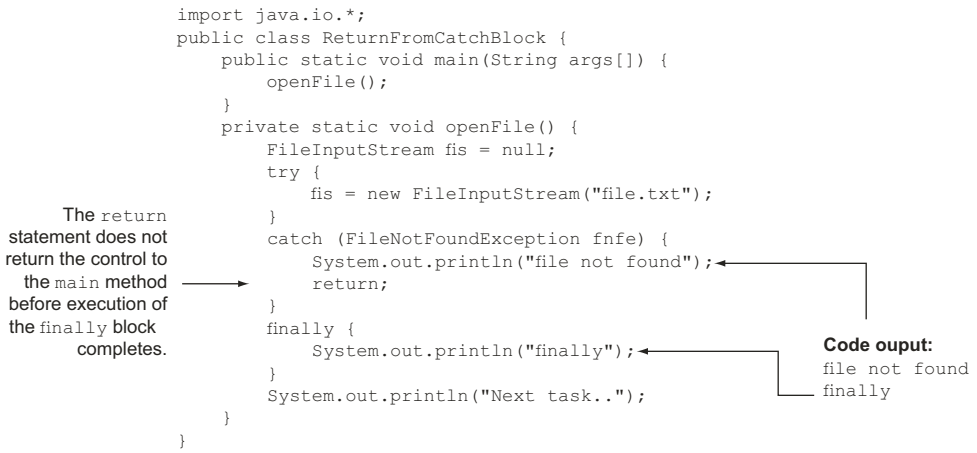


Figure 7.11 The `finally` block executes even if an exception handler defines a `return` statement.

to the `println` statement "`Nexttask...`" that follows the `try` block because the `return` statement is encountered in the `catch` block, as mentioned previously.

Going back to the example of the guy and his girlfriend, a few tragic conditions, such as an earthquake or tornado, can cancel the coffee treat. Similarly, there are a few scenarios in Java in which a `finally` block does not execute:

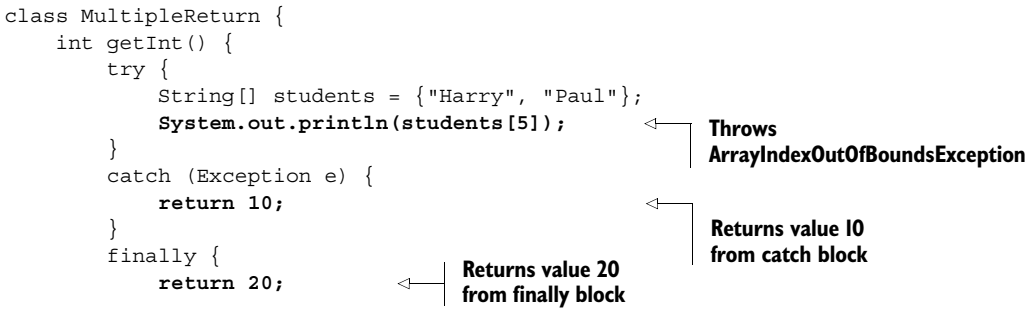
- *Application termination*—The `try` or the `catch` block executes `System.exit`, which terminates the application
- *Fatal errors*—A crash of the JVM or the OS

In the exam, you may be questioned on the correct order of two or more exception handlers. Does order matter? See for yourself in the next section.

7.2.3 What happens if both a catch and a finally block define return statements?

In the previous section, you saw that the `finally` block executes even if a `catch` block defines a `return` statement. For a method that defines a `try-catch-finally` block, what is returned to the calling method if both `catch` and `finally` return a value?

Here's an example:



```
        }
    }
    public static void main(String args[]) {
        MultipleReturn var = new MultipleReturn();
        System.out.println(var.getInt());
    }
}
```

The output of the preceding code is

```
20
```

If both `catch` and `finally` blocks define `return` statements, the calling method will receive a value from the `finally` block.

7.2.4 What happens if a finally block modifies the value returned from a catch block?

If a `catch` block returns a primitive data type, the `finally` block can't modify the value being returned by it. An example:

```
class MultipleReturn {
    int getInt() {
        int returnVal = 10;
        try {
            String[] students = {"Harry", "Paul"};
            System.out.println(students[5]);
        }
        catch (Exception e) {
            System.out.println("About to return :" + returnVal);
            return returnVal;
        }
        finally {
            returnVal += 10;
            System.out.println("Return value is now :" + returnVal);
        }
        return returnVal;
    }
    public static void main(String args[]) {
        MultipleReturn var = new MultipleReturn();
        System.out.println("In Main:" + var.getInt());
    }
}
```

Throws ArrayIndexOutOfBoundsException

Returns value 10 from catch block

Modifies value of variable to be returned in finally block

The output of the preceding code is as follows:

```
About to return :10
Return value is now :20
In Main:10
```

Even though the `finally` block adds `10` to variable `returnVal`, this modified value is not returned to the method `main`. Control in the `catch` block *copies* the value of `returnVal` to be returned before it executes the `finally` block, so the returned value is not modified when `finally` executes.

Will the preceding code behave in a similar manner if the method returns an object? See for yourself:

```
class MultipleReturn {
    StringBuilder getStringBuilder() {
        StringBuilder returnVal = new StringBuilder("10");
        try {
            String[] students = {"Harry", "Paul"};
            System.out.println(students[5]);
        }
        catch (Exception e) {
            System.out.println("About to return :" + returnVal);
            return returnVal;
        }
        finally {
            returnVal.append("10");
            System.out.println("Return value is now :" + returnVal);
        }
        return returnVal;
    }

    public static void main(String args[]) {
        MultipleReturn var = new MultipleReturn();
        System.out.println("In Main:" + var.getStringBuilder());
    }
}
```

Throws ArrayIndexOutOfBoundsException

Returns StringBuilder object value from catch block

Modifies value of variable to be returned in finally block

This is the output of the preceding code:

```
About to return :10
Return value is now :1010
In Main:1010
```

In this case, the `catch` block returns an object of the class `StringBuilder`. When the `finally` block executes, it can access the value of the object referred to by the variable `returnVal` and can modify it. The modified value is returned to the method `main`. Remember that primitives are passed by value and objects are passed by reference.

EXAM TIP Watch out for code that returns a value from the `catch` block and modifies it in the `finally` block. If a `catch` block returns a primitive data type, the `finally` block can't modify the value being returned by it. If a `catch` block returns an object, the `finally` block can modify the value being returned by it.

7.2.5 Does the order of the exceptions caught in the catch blocks matter?

Order doesn't matter for unrelated classes. But it does matter for related classes sharing an IS-A relationship.

In the latter case, if you try to catch an exception of the base class before an exception of the derived class, your code will fail to compile. This behavior may seem bizarre, but there's a valid reason for it. As you know, an object of a derived class can

Figure 7.12 The order of placement of exception handlers is important.

be assigned to a variable of a base class. Similarly, if you try to catch an exception of a base class before its derived class, the exception handler for the derived class can never be reached, so the code will fail to compile.

Examine the code in figure 7.12, which has been modified by defining the `catch` block for `IOException` before the `catch` block for `FileNotFoundException`.

Figure 7.13 depicts an interesting way to remember that the order matters. As you know, a thrown exception looks for an appropriate exception handler, starting with the first handler and working toward the last. Let's compare a thrown exception to a tiger and the exception handlers to doors that allow certain types of creatures to enter. Like a thrown exception, the tiger should start with the first door and move on to the rest of the doors until a match is found.

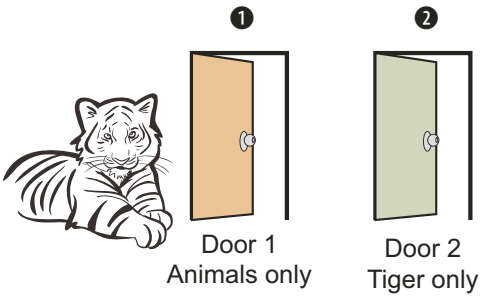


Figure 7.13 A visual way to remember that the order matters for exceptions caught in the `catch` blocks

The tiger starts with the first door, which allows all animals to enter. Voilà. The tiger enters the first door and never reaches the second door, which is meant specifically for tigers. In Java, when such a condition arises, the Java compiler refuses to compile the code because the later exception handler code will never execute. Java doesn't compile code if it contains unreachable statements.

Rules to remember

Here are a few more rules you'll need to answer the questions in the OCA Java SE 7 Programmer I exam:

- A `try` block may be followed by multiple `catch` blocks, and the `catch` blocks may be followed by a single `finally` block.
- A `try` block may be followed by either a `catch` or a `finally` block or both. But a `finally` block alone wouldn't suffice if code in the `try` block throws a checked exception. In this case, you need to catch the checked exception or declare it to be thrown by your method. Otherwise your code won't compile.
- The `try`, `catch`, and `finally` blocks can't exist independently.
- The `finally` block can't appear before a `catch` block.
- A `finally` block always executes, regardless of whether the code throws an exception.

7.2.6 Can I rethrow an exception or the error I catch?

You can do whatever you want with an exception. Rethrow it, pass it on to a method, assign it to another variable, upload it to a server, send it in an SMS, and so on. Examine the following code:

```
import java.io.*;
public class ReThrowException {
    FileInputStream soccer;
    public void myMethod() {
        try {
            soccer = new FileInputStream("soccer.txt");
        }
        catch (FileNotFoundException fnfe) {
            throw fnfe;                                    <-- Use throw keyword to
        }                                                      throw exception caught
    }                                                          in exception handler
}
```

Oops. The previous code fails to compile, and you get the following compilation error message:

```
ReThrowException.java:9: unreported exception java.io.FileNotFoundException;
     must be caught or declared to be thrown
            throw fnfe;
            ^
```

When you rethrow a checked exception, it's treated like a regular thrown checked exception, meaning that all the rules of handling a checked exception apply to it. In the previous example, the code neither "caught" the rethrown `FileNotFoundException` exception nor declared that the method `myMethod` would throw it using the `throw` clause. Hence, the code failed to compile.

The following (modified) code declares that the method `myMethod` throws a `FileNotFoundException`, and it compiles successfully:

```
import java.io.*;
public class ReThrowException {                          Throws
    FileInputStream soccer;                              FileNotFoundException
    public void myMethod() throws FileNotFoundException {
```

```
        try {
            soccer = new FileInputStream("soccer.txt");
        }
        catch (FileNotFoundException fnfe) {
            throw fnfe;                                    <-- Exception rethrown
        }
    }
}
```

Another interesting point to note is that the previous code doesn't apply to a `RuntimeException`. You can rethrow a runtime exception, but you're not required to catch it, nor must you modify your method signature to include the `throws` clause. The simple reason for this rule is that `RuntimeExceptions` aren't checked exceptions, and they may not be caught or declared to be thrown by your code (exception categories are discussed in detail in section 7.3).

7.2.7 Can I declare my methods to throw a checked exception, instead of handling it?

If a method doesn't wish to handle the checked exceptions thrown by a method it calls, it can *throw* these exceptions using the `throws` clause in its own method signature. Examine the following example, in which the method `myMethod` doesn't include an exception handler; instead, it rethrows the `IOException` thrown by a constructor of the class `FileInputStream` using the `throws` clause in its method signature:

```
import java.io.*;
public class ReThrowException2 {                           <-- myMethod throws checked exception
    public void myMethod() throws IOException {
        FileInputStream soccer = new FileInputStream("soccer.txt");
        soccer.close();
    }
}
```

Any method that calls `myMethod` must now either catch the exception `IOException` or declare that it will be rethrown in its method signature.

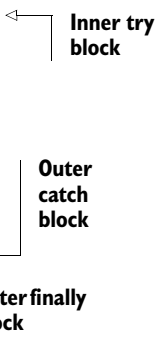
7.2.8 I can create nested loops, so can I create nested try-catch blocks too?

The simple answer is yes, you can define a `try-catch-finally` block within another `try-catch-finally` block. Theoretically, the levels of nesting for the `try-catch-finally` blocks have no limits.

In the following example, another set of `try-catch` blocks is defined in the `try` and `finally` blocks of the outer `try` block:

```
import java.io.*;
public class NestedTryCatch {
    FileInputStream players, coach;
    public void myMethod() {                               <-- Outer try block
        try {
            players = new FileInputStream("players.txt");
```

```
            try {
                coach = new FileInputStream("coach.txt");
                //.. rest of the code
            }
            catch (FileNotFoundException e) {
                System.out.println("coach.txt not found");
            }
            //.. rest of the code
        }
        catch (FileNotFoundException fnfe) {
            System.out.println("players.txt not found");
        }
        finally {
            try {
                players.close();
                coach.close();
            }
            catch (IOException ioe) {
                System.out.println(ioe);
            }
        }
    }
}
```



Now comes another Twist in the Tale exercise that'll test your understanding of the exceptions thrown and caught by nested `try-catch` blocks. In this one, an inner `try` block defines code that throws a `NullPointerException`. But the inner `try` block doesn't define an exception handler for this exception. Will the outer `try` block catch this exception? See for yourself (answer in the appendix).

Twist in the Tale 7.2

Given that players.txt exists on your system and that the assignment of `players`, shown in bold, doesn't throw any exceptions, what is the output of the following code?

```
import java.io.*;
public class TwistInTaleNestedTryCatch {
    static FileInputStream players, coach;
    public static void main(String args[]) {
        try {
            players = new FileInputStream("players.txt");
            System.out.println("players.txt found");
            try {
                coach.close();
            }
            catch (IOException e) {
                System.out.println("coach.txt not found");
            }
        }
        catch (FileNotFoundException fnfe) {
            System.out.println("players.txt not found");
        }
```

```
        catch (NullPointerException ne) {
            System.out.println("NullPointerException");
        }
    }
}
```

a The code prints

```
players.txt found
NullPointerException
```

b The code prints

```
players.txt found
coach.txt not found
```

c The code throws a runtime exception.

d The code fails to compile.

7.3 Categories of exceptions

[8.1] Differentiate among checked exceptions, RuntimeExceptions, and Errors

In this section, you'll learn about the categories of exceptions in Java, including checked exceptions, unchecked exceptions, and errors. I'll use code examples to explain the differences between these categories.

7.3.1 Identifying exception categories

As depicted in figure 7.14, exceptions can be divided into three main categories:

- Checked exceptions
- Runtime exceptions (unchecked exceptions)
- Errors

Of these three types, checked exceptions require most of your attention when it comes to coding and using methods. Normally, you shouldn't try to catch runtime exceptions, and there are few options you can use for the errors, because they're thrown by the JVM.

For the OCA Java SE 7 Programmer I exam, it's important to have a crystal-clear understanding of these three categories of exceptions, including their similarities and differences.

These categories are related to each other, and subclasses of the class `java.lang.Exception`

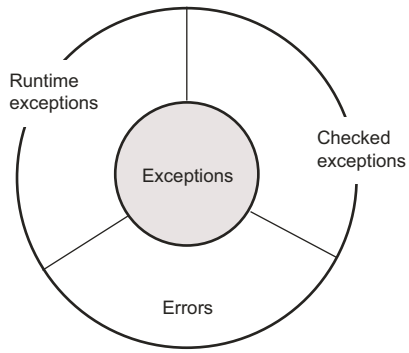


Figure 7.14 Categories of exceptions: checked exceptions, runtime exceptions, and errors

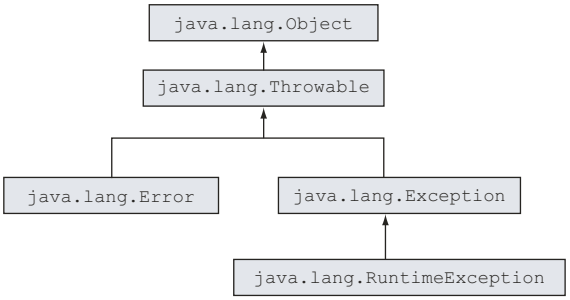


Figure 7.15 Class hierarchy of exception categories

are categorized as checked exceptions if they aren't subclasses of the class `java.lang.RuntimeException`. Subclasses of the class `java.lang.RuntimeException` are categorized as runtime exceptions. Subclasses of the class `java.lang.Error` are categorized as errors. Figure 7.15 illustrates the class hierarchy of these exception categories.

As you can see in figure 7.15, all of the errors and exceptions extend the class `java.lang.Throwable`. Let's examine each of these categories in detail.

7.3.2 Checked exceptions

When we talk about handling exceptions, it's *checked* exceptions that take up most of our attention. What is a checked exception?

- A checked exception is an unacceptable condition *foreseen* by the author of a method but outside the immediate control of the code. For example, `FileNotFoundException` is a checked exception. This exception is thrown if the file that the code is trying to access can't be found. When it's thrown, this condition is outside the immediate control of the author of the code but it was *foreseen* by the author.
- A checked exception is a subclass of class `java.lang.Exception`, but it's not a subclass of `java.lang.RuntimeException`. It's interesting to note, however, that the class `java.lang.RuntimeException` itself is a subclass of the class `java.lang.Exception`.

EXAM TIP In the OCA Java SE 7 Programmer I exam, you may have to select which type of reference variable to use to store the object of the thrown checked exception in a handler. To answer such questions correctly, remember that a checked exception is a subclass of the `java.lang.Exception` class, but not a subclass of `java.lang.RuntimeException`.

If a method uses another method that may throw a checked exception, one of the two following things should be true:

- The method should be enclosed within a `try-catch` block

or

- The method should specify this exception to be thrown in its method signature

Let's quickly review the class `OpenFile`, which we discussed in an earlier section:

```
import java.io.*;
public class OpenFile {
    public static void main(String args[]) {
        FileInputStream fis = new FileInputStream("file.txt");
    }
}
```

Throws checked exception FileNotFoundException

As mentioned earlier, this will fail to compile because the code in bold throws the checked exception `FileNotFoundException`, and the previous class neither enclosed this line of code within a `try` block nor specified that it would be thrown in its method signature.

If a method chooses not to handle the checked exception thrown by its code, it may choose to throw it, and its method signature must specify that. Examine the following definition of the `FileInputStream` constructor (from the source code of the Java API), which is used in the previously mentioned class `OpenFile`:

```
public FileInputStream(String name) throws FileNotFoundException {
        this(name != null ? new File(name) : null);
}
```

Now examine the corresponding description of this constructor in the Java API documentation:

```
public FileInputStream(File file)
                throws FileNotFoundException
```

A checked exception is part of the API and is well documented. Checked exceptions are unacceptable conditions that a programmer *foresees* at the time of writing a method. By declaring these exceptions as checked exceptions, the author of the method makes its users aware of the exceptional conditions that can arise from its use. The user of a method with a checked exception must handle the exceptional condition accordingly.

7.3.3 Runtime exceptions (also known as unchecked exceptions)

Although you'll spend most of your time and energy combating checked exceptions, it's the runtime exceptions that'll give you the most headaches. This is particularly true when you're preparing to work on real-life projects. Some examples of runtime exceptions are `NullPointerException` (the most common one), `ArrayIndexOutOfBoundsException`, and `ClassCastException`.

What is a runtime exception?

- A runtime exception is a representation of a programming error. These occur from inappropriate use of another piece of code. For example, `NullPointerException` is a runtime exception that occurs when a piece of code tries to execute some code on a variable that hasn't been assigned an object and points to `null`. Another example is `ArrayIndexOutOfBoundsException`, which is thrown when a piece of code tries to access an array list element at a nonexistent position.
- A runtime exception is a subclass of `java.lang.RuntimeException`.

- A runtime exception may not be a part of the method signature, even if a method may throw it.

How about an example, now that you've read the previous definitions of runtime exceptions? Examine the following code, which throws a runtime exception (`ArrayIndexOutOfBoundsException`):

```
public class InvalidArrayAccess {
    public static void main(String args[]) {
        String[] students = {"Shreya", "Joseph"};
        System.out.println(students[5]);
        System.out.println("All seems to be well");
    }
}
```

students[5] tries to access nonexistent array position; Exception thrown: ArrayIndexOutOfBoundsException

The preceding code doesn't print output from `System.out.println()`.

It's possible to create an exception handler for the exception `ArrayIndexOutOfBoundsException` thrown by the previous example code, as follows:

```
public class InvalidArrayAccess {
    public static void main(String args[]) {
        String[] students = {"Shreya", "Joseph"};
        try {
            System.out.println(students[5]);
        }
        catch (ArrayIndexOutOfBoundsException e){
            System.out.println("Exception");
        }
        System.out.println("All seems to be well");
    }
}
```

The output of the previous code is as follows:

```
Exception
All seems to be well
```

In the same way you can *catch* a checked exception, you can also catch a `RuntimeException`. In the previous code, you can prevent `ArrayIndexOutOfBoundsException` from being thrown by using appropriate checks:

```
public class InvalidArrayAccess {
    public static void main(String args[]) {
        String[] students = {"Shreya", "Joseph"};
        int pos = 1;
        if (pos > 0 && pos < students.length)
            System.out.println(students[pos]);
    }
}
```

This line won't execute because pos is greater than length of array students.

7.3.4 Errors

Whether you're preparing for the OCA Java SE 7 Programmer I exam or your real-life projects, you need to know when the JVM throws errors. These errors are considered to be *serious* exceptional conditions and they can't be directly controlled by your code.

What's an error?

- An error is a serious exception thrown by the JVM as a result of an error in the environment state that processes your code. For example, `NoClassDefFoundError` is an error thrown by the JVM when it's unable to locate the .class file that it's supposed to run. `StackOverflowError` is another error thrown by the JVM when the size of the memory required by the stack of a Java program is greater than what the JRE has offered for the Java application. This error usually occurs as a result of infinite or highly nested loops.
- An error is a subclass of class `java.lang.Error`.
- An error need not be a part of a method signature.
- An error can be caught by an exception handler, but it shouldn't be.

The following example shows how it's possible to "catch" an error:

```
public class CatchError {
    public static void main(String args[]) {
        try {
            myMethod();
        }
        catch (StackOverflowError s) {
            System.out.println(s);
        }
    }
    public static void myMethod() {
        System.out.println("myMethod");
        myMethod();
    }
}
```

A class can catch and handle an error, but it shouldn't—it should instead let the JVM handle the error itself.

I agree that you shouldn't handle errors in your code. But if you do, will the exception handler that handles the code be executed? See for yourself by answering the question in the following Twist in the Tale exercise (answer in the appendix).

Twist in the Tale 7.3

Will the code in the error-handling block execute? What do you think is the output of the following code?

```
public class TwistInTaleCatchError {
    public static void main(String args[]) {
        try {
            myMethod();
        }
        catch (StackOverflowError s) {
            for (int i=0; i<2; ++i)
                System.out.println(i);
        }
    }
    public static void myMethod() {
        myMethod();
    }
}
```

a
```
0
java.lang.StackOverFlowError
```

b
```
0
1
```

c
```
0
1
2
java.lang.StackOverFlowError
```

7.4 Common exception classes and categories

[8.5] Recognize common exception classes and categories

In this section, we'll take a look at common exception classes and categories of exceptions. You'll also learn about the scenarios in which these exceptions are thrown and how to handle them.

For the OCA Java SE 7 Programmer I exam, you should be familiar with the scenarios that lead to these commonly thrown exception classes and categories, and how to handle them. Table 7.2 lists common errors and exceptions.

Table 7.2 Common errors and exceptions

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

The OCA Java SE 7 Programmer I exam objectives require that you understand which of the previously mentioned errors and exceptions are thrown by the JVM and which should be thrown programmatically. From the discussion of errors earlier in this chapter, you know that the errors represent issues associated with the JRE, such as `OutOfMemoryError`. As a programmer, you *shouldn't* throw or catch these errors—leave them for the JVM. The definition of the runtime exception notes that these are the kind of exceptions that are thrown by the JVM, which shouldn't be thrown by you programmatically.

Let's review each of these in detail.

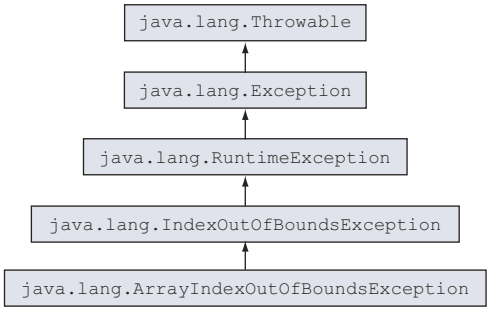


Figure 7.16 Class hierarchy of `ArrayIndexOutOfBoundsException`

7.4.1 ArrayIndexOutOfBoundsException and IndexOutOfBoundsException

As shown in figure 7.16, `ArrayIndexOutOfBoundsException` and `IndexOutOfBoundsException` are runtime exceptions, which share an IS-A relationship. `IndexOutOfBoundsException` is subclassed by `ArrayIndexOutOfBoundsException`.

`ArrayIndexOutOfBoundsException` is thrown when a piece of code tries to access an array out of its bounds (either an array is accessed at a position less than 0 or at a position greater than or equal to its length). `IndexOutOfBoundsException` is thrown when a piece of code tries to access a list, like an `ArrayList`, using an illegal index.

Assuming that an array and list have been defined as follows,

```
String[] season = {"Spring", "Summer"};
ArrayList<String> exams = new ArrayList<>();
exams.add("SCJP");
exams.add("SCWCD");
```

The following lines of code will throw `ArrayIndexOutOfBoundsException`:

```
System.out.println(season[5]);
System.out.println(season[-9]);
```

Can't access position >= array length

Can't access array at negative position

The following lines of code will throw `IndexOutOfBoundsException`:

```
System.out.println(exams.get(-1));
System.out.println(exams.get(4));
```

Can't access list at negative position

Can't access list at position >= its size

Why do you think the JVM has taken the responsibility to throw this exception itself? One of the main reasons is that this exception isn't known until runtime and depends on the array or list position that's being accessed by a piece of code. Most often, a variable is used to specify this array or list position, and its value may not be known until runtime.

NOTE When you try to access an invalid array position, `ArrayIndexOutOfBoundsException` is thrown. When you try to access an invalid `ArrayList` position, `IndexOutOfBoundsException` is thrown.

You can avoid these exceptions from being thrown if you check whether the index position you are trying to access is greater than or equal to `0` and less than the size of your array or `ArrayList`.

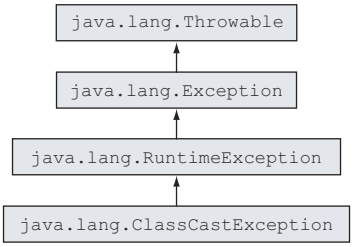


Figure 7.17 Class hierarchy of `ClassCastException`

7.4.2 ClassCastException

Before I start discussing the example I'll use for this exception, take a quick look at figure 7.17 to review the class hierarchy of this exception.

Examine the code in listing 7.2, where the line of code that throws the `ClassCastException` is shown in bold.

Listing 7.2 An example of code that throws `ClassCastException`

```
import java.util.ArrayList;
public class ListAccess {
    public static void main(String args[]) {
        ArrayList<Ink> inks = new ArrayList<Ink>();
        inks.add(new ColorInk());
        inks.add(new BlackInk());
        Ink ink = (BlackInk)inks.get(0);
    }
}
class Ink{}
class ColorInk extends Ink{}
class BlackInk extends Ink{}
```

Throws ClassCastException

`ClassCastException` is thrown when an object fails an IS-A test with the class type to which it's being cast. In the preceding example, class `Ink` is the base class for classes `ColorInk` and `BlackInk`. The JVM throws a `ClassCastException` in the previous case because the line of code in bold tries to explicitly cast an object of `ColorInk` to `BlackInk`.

Note that this line of code avoided the compilation error because the variable `inks` defines an `ArrayList` of type `Ink`, which is capable of storing objects of type `Ink` and all its subclasses. The code then correctly adds the allowed objects: one each of `BlackInk` and `ColorInk`. If the code had defined an `ArrayList` of type `BlackInk` or `ColorInk`, the code would have failed the compilation, as follows:

```
import java.util.ArrayList;
public class Invalid {
    public static void main(String args[]) {
        ArrayList<ColorInk> inks = new ArrayList<ColorInk>();
        inks.add(new ColorInk());
        Ink ink = (BlackInk)inks.get(0);
    }
}
class Ink{}
class ColorInk extends Ink{}
class BlackInk extends Ink{}
```

Compilation issues

Here's the compilation error thrown by the previously modified piece of code:

```
Invalid.java:6: inconvertible types
found   : ColorInk
required: BlackInk
        Ink ink = (BlackInk)inks.get(0);
                            ^
```

You can use the `instanceof` operator to verify whether an object can be cast to another class before casting it. Assuming that the definition of classes `Ink`, `ColorInk`, and `BlackInk` are the same as defined in the previous example, the following lines of code will avoid the `ClassCastException`:

```
import java.util.ArrayList;
public class AvoidClassCastException {
    public static void main(String args[]) {
        ArrayList<Ink> inks = new ArrayList<Ink>();
        inks.add(new ColorInk());
        inks.add(new BlackInk());
        if (inks.get(0) instanceof BlackInk) {
            BlackInk ink = (BlackInk)inks.get(0);          <-- No ClassCastException
        }
    }
}
```

In the previous example, the condition `(inks.get(0)instanceofBlackInk)` evaluates to `false`, so the then part of the `if` statement doesn't execute.

In the following Twist in the Tale exercise, we'll introduce an interface used in the casting example in listing 7.2 (answer in the appendix).

Twist in the Tale 7.4

Let's introduce an interface used in listing 7.2 and see how it behaves. Following is the modified code. Examine the code and select the correct options:

```
class Ink{}
interface Printable {}
class ColorInk extends Ink implements Printable {}
class BlackInk extends Ink{}

class TwistInTaleCasting {
    public static void main(String args[]) {
        Printable printable = null;
        BlackInk blackInk = new BlackInk();
        printable = (Printable)blackInk;
    }
}
```

a `printable = (Printable)blackInk` will throw compilation error.

b `printable = (Printable)blackInk` will throw runtime exception.

c `printable = (Printable)blackInk` will throw checked exception.

d The following line of code will fail to compile:

```
printable = blackInk;
```

7.4.3 IllegalArgumentException

As the name of this exception suggests, `IllegalArgumentException` is thrown to specify that a method has passed illegal or inappropriate arguments. Its class hierarchy is shown in figure 7.18.

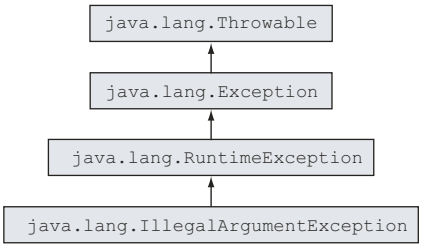


Figure 7.18 Class hierarchy of `IllegalArgumentException`

Even though it's a runtime exception, programmers usually use this exception to validate the arguments that are passed to a method. The exception constructor is passed a descriptive message, specifying the exception details. Examine the following code:

```
public void login(String username, String pwd, int maxLoginAttempt) {

    if (username == null || username.length() < 6)
        throw new IllegalArgumentException
                   ("Login:username can't be shorter than 6 chars");

    if (pwd == null || pwd.length() < 8)
        throw new IllegalArgumentException
                   ("Login: pwd cannot be shorter than 8 chars");

    if (maxLoginAttempt < 0)
        throw new IllegalArgumentException
                   ("Login: Invalid loginattempt val");

    //.. rest of the method code
}
```

The previous method validates the various method parameters passed to it and throws an appropriate `IllegalArgumentException` if they don't meet the requirements of the method. Each object of the `IllegalArgumentException` is passed a different `String` message that briefly describes it.

7.4.4 IllegalStateException

According to the Java API documentation, an `IllegalStateException` "signals that a method has been invoked at an illegal or inappropriate time. In other words, the Java environment or Java application is not in an appropriate state for the requested operation."[1]

The class hierarchy for `IllegalStateException` is shown in figure 7.19.

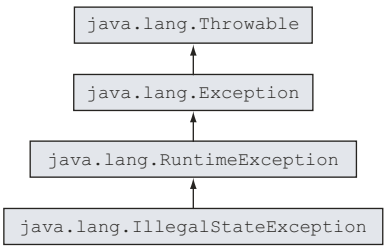


Figure 7.19 Class hierarchy of `IllegalStateException`

[1] The `IllegalStateException` documentation can be found in the Javadoc: http://docs.oracle.com/javase/7/docs/api/java/lang/IllegalStateException.html.

As an author of code, you can throw `IllegalStateException` to signal to the calling method that the method being requested for execution can't be called for the current state of an object.

For example, what happens if an application tries to modify an SMS that is already in transmission? Examine the following code:

```
class SMS {
    private String msg;
    private boolean inTransit = false;

    public void create() {
        msg = "A new Message";
    }

    public void transmit() {
        ........                                    <-- Code to transmit message
        inTransit = true;
    }

    public void modify() {
        if (!inTransit)
            msg = "new modified message";
        else
            throw new IllegalStateException
                                ("Msg in transit. Can't modify it");
    }
  }

public class ThrowIllegalStateException {
    public static void main(String[] args) {
        SMS sms = new SMS();
        sms.create();
        sms.transmit();
        sms.modify();
    }
  }
```

On execution, the class `ThrowIllegalStateException` throws the following exception:

```
Exception in thread "main" java.lang.IllegalStateException: Msg in transit.
     Can't modify it
    at SMS.modify(ThrowIllegalStateException.java:18)
    at ThrowIllegalStateException.main(ThrowIllegalStateException.java:27)
```

In the example code, the method `modify` in the class `SMS` throws an `IllegalStateException` if another piece of code tries to call it after the method `send` has been executed.

7.4.5 NullPointerException

`NullPointerException`, shown in figure 7.20, is the quintessential exception. I imagine that almost all Java programmers have had a taste of this exception, but let's look at an explanation for it.

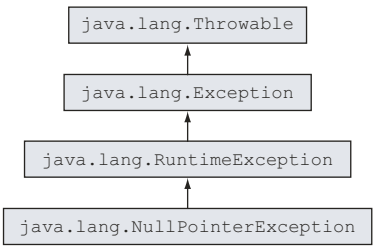


Figure 7.20 Class hierarchy of `NullPointerException`

This exception is thrown by the JVM if you try to access a method or a variable with a `null` value. The exam can have interesting code combinations to test you on whether a particular piece of code will throw a `NullPointerException`. The key is to ensure that the reference variable has been assigned a non-`null` value. In particular, I'll address the following cases:

- Accessing members of a reference variable that is explicitly assigned a `null` value.
- Accessing members of an uninitialized instance or `static` reference variable. These are implicitly assigned a `null` value.
- Using an uninitialized local variable, which may *seem* to throw a `NullPointerException`.
- Attempting to access nonexistent array positions.
- Using members of an array element that are assigned a `null` value.

Let's get started with the first case, in which a variable is explicitly assigned a `null` value:

```
import java.util.ArrayList;
class ThrowNullPointerException {
    static ArrayList<String> list = null;
    public static void main(String[] args) {
        list.add("1");
    }
}
```

list is null

Attempt to call method add on list throws NullPointerException

The preceding code tries to access the method `add` on variable `list`, which has been assigned a `null` value. It throws an exception, as follows:

```
Exception in thread "main" java.lang.NullPointerException
    at ThrowNullPointerException.main(ThrowNullPointerException.java:5)
```

By default, the `static` and instance variables of a class are assigned a `null` value. In the previous example, the static variable `list` is assigned an explicit `null` value. When the method `main` tries to execute the method `add` on variable `list`, it calls a method on a `null` value. This call causes the JVM to throw a `NullPointerException` (which is a `RuntimeException`). If you define the variable `list` as an instance variable and don't assign an explicit value to it, you'll get the same result (`NullPointerException` being thrown at runtime). Because the `static` method `main` can't access the instance variable `list`, you'll need to create an object of class `ThrowNullPointerException` to access it:

```
import java.util.ArrayList;
class ThrowNullPointerException {
    ArrayList<String> list;
    public static void main(String[] args) {
        ThrowNullPointerException obj = new ThrowNullPointerException();
        obj.list.add("1");
    }
}
```

list is implicitly assigned a null value

Attempt to call method add on list throws NullPointerException

You can prevent a `NullPointerException` from being thrown by checking whether an object is `null` before trying to access its member:

```
import java.util.ArrayList;
class ThrowNullPointerException {
    static ArrayList<String> list;
    public static void main(String[] args) {
        if (list!=null)
            list.add("1");
    }
}
```

Ascertain that list is not null

What happens if you modify the previous code as follows? Will it still throw a `NullPointerException`?

```
import java.util.ArrayList;
class ThrowNullPointerException {
    public static void main(String[] args) {
        ArrayList<String> list;
        if (list!=null)
            list.add("1");
    }
}
```

Fails to compile

Interestingly, the previous code fails to compile. `list` is defined as a local variable inside the method `main`, and by default, local variables aren't assigned a value—not even a `null` value. If you attempt to use an uninitialized local variable, your code will fail to compile. Watch out for similar questions in the exam.

Another set of conditions when code may throw the `NullPointerException` involves use of arrays:

```
class ThrowAnotherNullPointerException {
    static String[] oldLaptops;
    public static void main(String[] args) {
        System.out.println(oldLaptops[1]);

        String[] newLaptops = new String[2];
        System.out.println(newLaptops[1].toString());
    }
}
```

Throws NullPointerException

Throws NullPointerException

The variable `oldLaptops` is assigned a `null` value by default because it's a `static` variable. Its array elements are neither initialized nor assigned a value. The code that tries to access the array's second element throws a `NullPointerException`.

In the second case, two array elements of the variable `newLaptops` are initialized and assigned a default value of `null`. If you call method `toString` on the second element of variable `newLaptops`, it results in a `NullPointerException` being thrown.

If you modify that line as shown in the following code, it won't throw an exception—it'll print the value `null`:

```
System.out.println(newLaptops[1]);
```

No RuntimeException. Prints "null".

EXAM TIP In the exam, watch out for code that tries to use an uninitialized local variable. Because such variables aren't initialized with even a `null` value, you can't print their value using the `System.out.println` method. Such code *won't* compile.

Let's modify the previous code that uses the variable `oldLaptops` and check your understanding of `NullPointerExceptions`. Here's another Twist in the Tale hands-on exercise for you (answers in the appendix).

Twist in the Tale 7.5

Let's check your understanding of the `NullPointerException`. Here's a code snippet. Examine the code and select the correct answers.

```
class TwistInTaleNullPointerException {
    public static void main(String[] args) {
        String[][] oldLaptops =
             { {"Dell", "Toshiba", "Vaio"}, null,
{"IBM"}, new String[10] };
        System.out.println(oldLaptops[0][0]);          // line 1
        System.out.println(oldLaptops[1]);             // line 2
        System.out.println(oldLaptops[3][6]);          // line 3
        System.out.println(oldLaptops[3][0].length()); // line 4
        System.out.println(oldLaptops);                // line 5
    }
}
```

a Code on line 1 will throw `NullPointerException`
b Code on lines 1 and 3 will throw `NullPointerException`
c Only code on line 4 will throw `NullPointerException`
d Code on lines 3 and 5 will throw `NullPointerException`

7.4.6 NumberFormatException

What happens if you try to convert "87" and "9m#" to numeric values? The former value is okay, but you can't convert the latter value to a numeric value unless it's an encoded value, straight from a James Bond movie, that can be converted to anything.

As shown in figure 7.21, `NumberFormatException` is a runtime exception. It's thrown to indicate that the application has tried to convert a string (with an inappropriate format) to one of the numeric types.

Multiple classes in the Java API define parsing methods. One of the most frequently used

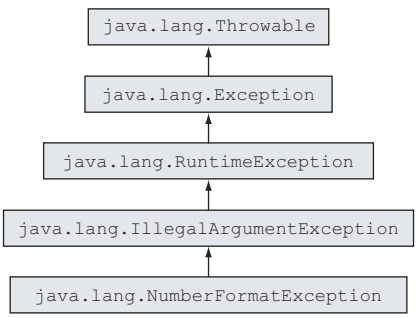


Figure 7.21 Class hierarchy of `NumberFormatException`

methods is `parseInt` from the class `Integer`. It's used to parse a `String` argument as a signed (negative or positive) decimal integer. Here are some examples:

Valid String values that can be converted to numeric values

```
System.out.println(Integer.parseInt("-123"));
System.out.println(Integer.parseInt("123"));
System.out.println(Integer.parseInt("+123"));
System.out.println(Integer.parseInt("123_45"));
System.out.println(Integer.parseInt("12ABCD"));
```

Will throw NumberFormatException. Use of underscores in string values isn't allowed.

Will throw NumberFormatException. Characters ABCD can't be converted to integers in base 10.

Starting in Java 7, you can use underscores (_) in numeric literal values. But you can't use them in `String` values passed to the method `parseInt`. The letters `ABCD` are not used in the decimal number system, but they can be used in the hexadecimal number system, so you can convert the hexadecimal literal value `"12ABCD"` to the decimal number system by specifying the base of the number system as `16`:

```
System.out.println(Integer.parseInt("123ABCD", 16));
```

Prints 1223629

Note that the argument `16` is passed to the method `parseInt`, not to the method `println`. The following will not compile:

```
System.out.println(Integer.parseInt("123ABCD"), 16);
```

Won't compile

You may throw `NumberFormatException` from your own method to indicate that there's an issue with the conversion of a `String` value to a specified numeric format (decimal, octal, hexadecimal, binary), and you can add a customized exception message. One of the most common candidates for this exception is methods that are used to convert a command-line argument (accepted as a `String` value) to a numeric value. Please note that all command-line arguments are accepted in a `String` array as `String` values.

The following is an example of code that rethrows `NumberFormatException` programmatically:

```
public class ThrowNumberFormatException {
    public static int convertToNum(String val) {
        int num = 0;
        try {
            num = Integer.parseInt(val, 16);
        }
        catch (NumberFormatException e) {
            throw new NumberFormatException(val+
                  " cannot be converted to hexadecimal number");
        }
        return num;
    }
    public static void main(String args[]) {
        System.out.println(convertToNum("16b"));
        System.out.println(convertToNum("65v"));
    }
}
```

Rethrows NumberFormatException thrown by method parseInt with modified exception message

The conversion of the hexadecimal literal `16b` to the decimal number system is successful. But the conversion of the hexadecimal literal `65v` to the decimal number system fails, and the previous code will give the following output:

```
363
Exception in thread "main" java.lang.NumberFormatException: 65v cannot be
     converted to hexadecimal number
    at
     ThrowNumberFormatException.convertToNum(ThrowNumberFormatException.java:
     8)
    at ThrowNumberFormatException.main(ThrowNumberFormatException.java:14)
```

Now let's take a look at some of the common errors that are covered on this exam.

7.4.7 ExceptionInInitializerError

The `ExceptionInInitializerError` error is typically thrown by the JVM when a static initializer in your code throws any type of `RuntimeException`. Figure 7.22 shows the class hierarchy of `ExceptionInInitializerError`.

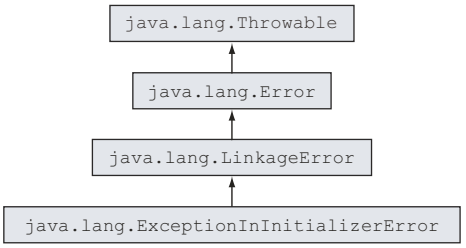


Figure 7.22 Class hierarchy of `ExceptionInInitializerError`

A `static` initializer block is defined using the keyword `static`, followed by curly braces, in a class. This block is defined within a class, but not within a method. It's usually used to execute code when a class loads for the first time. Runtime exceptions arising from any of the following will throw this error:

- Execution of an anonymous `static` block
- Initialization of a `static` variable
- Execution of a `static` method (called from either of the previous two items)

The `static` initializer block of the class defined in the following example will throw a `NumberFormatException`, and when the JVM tries to load this class, it'll throw an `ExceptionInInitializerError`:

```
public class DemoExceptionInInitializerError {
    static {
        int num = Integer.parseInt("sd", 16);
    }
}
```

Following is the error message when JVM tries to load the class `DemoExceptionInInitializerError`:

```
java.lang.ExceptionInInitializerError
Caused by: java.lang.NumberFormatException: For input string: "sd"
    at
     java.lang.NumberFormatException.forInputString(NumberFormatException.jav
     a:48)
```

```
    at java.lang.Integer.parseInt(Integer.java:447)
    at
     DemoExceptionInInitializerError.<clinit>(DemoExceptionInInitializerError
     .java:3)
```

EXAM TIP Beware of code that seems to be simple in the OCA Java SE 7 Programmer I exam. The class `DemoExceptionInInitializerError` (mentioned previously) seems deceptively simple, but it's a good candidate for an exam question. As you know, this class throws the error `ExceptionInInitializerError` when the JVM tries to load it.

In the following example, initialization of a static variable results in a `NullPointerException` being thrown. When this class is loaded by the JVM, it throws an `ExceptionInInitializerError`:

```
public class DemoExceptionInInitializerError1 {
    static String name = null;
    static int nameLength = name.length();
}
```

The error message when the JVM tries to load the `DemoExceptionInInitializerError1` class is as follows:

```
java.lang.ExceptionInInitializerError
Caused by: java.lang.NullPointerException
    at
     DemoExceptionInInitializerError1.<clinit>(DemoExceptionInInitializerErro
     r1.java:3)
Exception in thread "main"
```

Now let's move on to the exception thrown by a `static` method, which may be called by the `static` initializer block or to initialize a `static` variable. Examine the following code, in which `MyException` is a user-defined `RuntimeException`:

```
public class DemoExceptionInInitializerError2 {
    static String name = getName();
    static String getName() {
        throw new MyException();
    }
}
class MyException extends RuntimeException{}
```

MyException is a runtime exception

This is the error thrown by the class `DemoExceptionInInitializerError2`:

```
java.lang.ExceptionInInitializerError
Caused by: MyException
    at
     DemoExceptionInInitializerError2.getName(DemoExceptionInInitializerError
     2.java:4)
    at
     DemoExceptionInInitializerError2.<clinit>(DemoExceptionInInitializerErro
     r2.java:2)
```

Did you notice that the error `ExceptionInInitializerError` can be caused only by a runtime exception? This happens for valid reasons, of course.

If a static initializer block throws an error, it doesn't recover from it to come back to the code to throw an `ExceptionInInitializerError`. This error can't be thrown if a `static` initializer block throws an object of a checked exception because the Java compiler is intelligent enough to determine this condition and doesn't allow you to throw an unhandled checked exception from a static initialization block.

EXAM TIP `ExceptionInInitializerError` can be caused by an object of `RuntimeException` only. It can't occur as the result of an error or checked exception thrown by the `static` initialization block.

7.4.8 StackOverflowError

The `StackOverflowError` error extends `VirtualMachineError` (as shown in figure 7.23). As its name suggests, it should be left to be managed by the JVM.

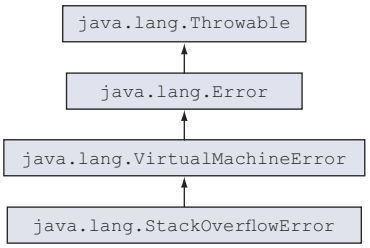


Figure 7.23 Class hierarchy of `StackOverflowError`

This error is thrown by the JVM when a Java program calls itself so many times that the memory stack allocated to execute the Java program "overflows." Examine the following code, in which a method calls itself recursively without an exit condition:

```
public class DemoStackOverflowError{
    static void recursion() {
        recursion();
    }
    public static void main(String args[]) {
        recursion();
    }
}
```

Calls itself recursively, without exit condition

The following error is thrown by the previous code:

```
Exception in thread "main" java.lang.StackOverflowError
    at DemoStackOverflowError.recursion(DemoStackOverflowError.java:3)
```

7.4.9 NoClassDefFoundError

What would happen if you failed to set your classpath and, as a result, the JVM was unable to load the class that you wanted to access or execute? Or what happens if you try to run your application before compiling it? In both these conditions, the JVM would throw `NoClassDefFoundError` (class hierarchy shown in figure 7.24).

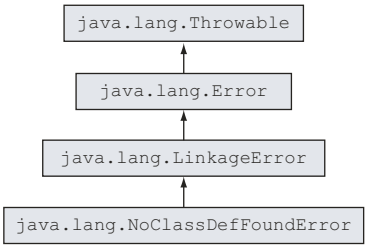


Figure 7.24 Class hierarchy of `NoClassDefFoundError`

This is what the Java API documentation says about this error:

Thrown if the Java Virtual Machine or a `ClassLoader` instance tries to load in the definition of a class (as part of a normal method call or as part of creating a new instance using the `new` expression) and no definition of the class could be found.[2]

Because this particular error is not a coding issue, I don't have a coding example for you. As you can see from the error hierarchy diagram in figure 7.24, this is a linkage error arising from a missing class file definition at runtime. This error should not be handled by the code and should be left to be handled exclusively by the JVM.

NOTE Don't confuse the exception thrown by `Class.forName()`, used to load the class, and `NoClassDefFoundError` thrown by the JVM. `Class.forName()` throws `ClassNotFoundException`.

7.4.10 OutOfMemoryError

What happens if you create and use a *lot* of objects in your application—for example, if you load a large chunk of persistent data to be processed by your application. In such a case, the JVM may run out of memory *on the heap*, and the garbage collector may not be able to free more memory for the JVM. In this case, the JVM is unable to create any more objects on the heap. An `OutOfMemoryError` will be thrown (class hierarchy shown in figure 7.25).

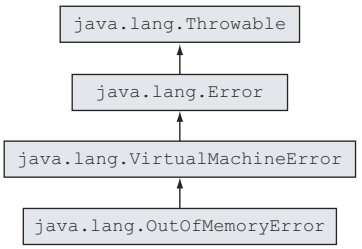


Figure 7.25 Class hierarchy of `OutOfMemoryError`

You'll always work with a finite heap size, no matter what platform you work on, so you can't create and use an unlimited number of objects in your application. To get around this error, you need to either limit the number of resources or objects that your application creates or increase the heap size on the platform you're working with.

A number of tools are available (which are beyond the scope of this book) that can help you monitor the number of objects created in your application.

7.5 Summary

In this chapter, we discussed the need for exception handling, as well as the advantages of defining the exception-handling code separate from the program logic. You saw how this approach helps separate concerns about defining the regular program logic and exception-handling code. We also looked at the code syntax, specifically the `try-catch-finally` blocks, for implementing the exception-handling code. Code that throws an exception should be enclosed within a `try` block that is immediately followed by a `catch` and/or a `finally` block. A `try` block can be followed by multiple `catch` blocks, but only a single `finally` block. A `finally` block can't be placed before

[2] The `NoClassDefFoundError` documentation can be found in the Javadoc: http://docs.oracle.com/javase/7/docs/api/java/lang/NoClassDefFoundError.html.

a `try` block. A `try` block must be followed by at least one `catch` or `finally` block. The `try`, `catch`, and `finally` blocks can't exist independently.

Next, we delved into the different categories of exceptions: checked exceptions, runtime or unchecked exceptions, and errors. Checked exceptions are subclasses of the class `java.lang.Exception`. Unchecked exceptions are subclasses of the class `java.lang.RuntimeException`, which itself is a subclass of the class `java.lang.Exception`. Errors are subclasses of `java.lang.Error`. All of these exceptions are subclasses of `java.lang.Throwable`.

A checked exception is an unacceptable condition foreseen by the author of a method but outside the immediate control of the code. A runtime exception represents a programming error—these occur because of inappropriate use of another piece of code. Errors are serious exceptions, thrown by the JVM, as a result of an error in the environment state that processes your code.

In the final sections of this chapter, we covered commonly occurring exceptions and errors, such as `NullPointerException`, `IllegalArgumentException`, `StackOverflowError`, and more. For each of these errors and exceptions, I explained the conditions in which they may be thrown in code and whether they should be explicitly handled in exception handlers.

7.6 Review notes

This section lists the main points of all the sections covered in this chapter.

A taste of exceptions:

- Trying to access an array element at a position that is greater than or equal to the array's length is considered an exceptional condition.
- Trying to execute a method that throws a checked exception without handling the thrown exception causes a compilation error.
- Trying to call a method recursively without defining an exit condition is considered an exceptional condition.
- Proper handling of exceptions is important in Java. It enables a programmer to define alternative code to execute in case an exceptional condition is met.

Why handle exceptions separately:

- Exception handlers refer to the blocks of code that execute when an exceptional condition arises during code execution.
- Handling exceptions separately enables you to define the main logic of your code together.
- Without the use of separate exception handlers, the main logic of your code would be lost in combating the exceptional conditions. (See figure 7.5 for an example.)
- Separate exception handlers enable you to define alternative code to execute separately from your main code flow when an exceptional condition is met.
- Exception handlers separate the concerns of defining the regular program logic from exception-handling code.

- Exceptions also help pinpoint the offending code, together with the method in which it is defined, by providing a stack trace of the exception or error.
- The stack trace of an exception enables you to get a list of all the exceptions that executed before the exception was reported.
- The stack trace of an exception is available within an exception handler.
- The JVM may send the stack trace of an unhandled exception to the Java console.

An exception is thrown:

- An exception is an object of the class `java.lang.Throwable`.
- When a piece of code hits an obstacle in the form of an exceptional condition, it creates an object of subclass `java.lang.Throwable`, initializes it with the necessary information (such as its type and optionally a textual description and the offending program's state), and hands it over to the JVM.
- The JVM keeps an account of all the methods that were called when it hits the code that throws an exception. To find an appropriate exception handler, it looks through all these method calls.
- Enclose the code that may throw an exception within a `try` block.
- Define `catch` blocks to include alternative code to execute when an exceptional condition arises.
- A `try` block can be followed by one or more `catch` blocks.
- The `catch` blocks must be followed by zero or one `finally` block.
- The `finally` block executes regardless of whether the code in the `try` block throws an exception.
- A `try` block can't define multiple `finally` blocks.
- The order in which the `catch` blocks are placed matters. If the caught exceptions have an inheritance relationship, the base class exceptions can't be caught before the derived class exceptions. An attempt to do this will result in compilation failure.
- A `finally` block will execute even if a `try` or `catch` block defines a `return` statement.
- A `try` block may be followed by either a `catch` or a `finally` block or both.
- If both `catch` and `finally` blocks define `return` statements, the calling method will receive the value from the `finally` block.
- If a `catch` block returns a primitive data type, a `finally` block can't modify the value being returned by it.
- If a `catch` block returns an object, a `finally` block can modify the value being returned by it.
- A `finally` block alone wouldn't suffice with a `try` block if code in the `try` block throws a checked exception. In this case, you need to catch the checked exception or define in the method signature that the exception is thrown, or your code won't compile.

- None of the `try`, `catch`, and `finally` blocks can exist independently.
- The `finally` block can't appear before a `catch` block.
- You can rethrow an error that you *catch* in an exception handler.
- You can either handle an exception or declare it to be thrown by your method. In the latter case, you need not handle the exception in your code. This applies to checked exceptions.
- You can create nested exception handlers.
- A `try`, `catch`, or `finally` block can define another `try-catch-finally` block. Theoretically, there is no limit on the allowed level of nesting of `try-catch-finally` blocks.

Categories of exceptions:

- Exceptions are divided into three categories: checked exceptions, runtime (or unchecked exceptions), and errors. These three categories share IS-A relationships (inheritance).
- Subclasses of the class `java.lang.RuntimeException` are categorized as runtime exceptions.
- Subclasses of the class `java.lang.Error` are categorized as errors.
- Subclasses of the class `java.lang.Exception` are categorized as checked exceptions if they are not subclasses of class `java.lang.Runtime`.
- The class `java.lang.RuntimeException` is a subclass of the class `java.lang.Exception`.
- The class `java.lang.Exception` is a subclass of the class `java.lang.Throwable`.
- The class `java.lang.Error` is also a subclass of the class `java.lang.Throwable`.
- The class `java.lang.Throwable` inherits the class `java.lang.Object`.

Checked exceptions:

- A checked exception is an unacceptable condition foreseen by the author of a method, but outside the immediate control of the code.
- `FileNotFoundException` is a checked exception. This exception is thrown if the file that the code is trying to access can't be found.
- All checked exceptions are a subclass of the `java.lang.Exception` class, not a subclass of `java.lang.RuntimeException`. It's interesting to note, however, that the class `java.lang.RuntimeException` itself is a subclass of the class `java.lang.Exception`.
- If a method calls another method that may throw a checked exception, either it must be enclosed within a `try-catch` block or the method should declare this exception to be thrown in its method signature.

Runtime exceptions:

- Runtime exceptions represent programming errors. These occur from inappropriate use of another piece of code. For example, `NullPointerException` is a

runtime exception that occurs when a piece of code tries to execute some code on a variable that hasn't been assigned an object and points to `null`. Another example is `ArrayIndexOutOfBoundsException`, which is thrown when a piece of code tries to access an array of list elements at a nonexistent position.

- A runtime exception is a subclass of `java.lang.RuntimeException`.
- A runtime exception isn't a part of the method signature, even if a method may throw it.
- A runtime exception may not necessarily be caught by a `try-catch` block.

Errors:

- An error is a serious exception, thrown by the JVM as a result of an error in the environment state, which processes your code. For example, `NoClassDefFoundError` is an error thrown by the JVM when it is unable to locate the .class file it is supposed to run.
- `StackOverflowError` is another error, thrown by the JVM when the size of the memory required by the stack of the Java program is greater than what the JRE has offered for the Java application. This error usually occurs as a result of infinite or highly nested loops.
- An error is a subclass of the class `java.lang.Error`.
- An error need not be a part of a method signature.
- Though you can handle the errors syntactically, there is little that you can do when these errors occur. For example, when the JVM throws `OutOfMemoryError`, your code execution will halt, even if you define an exception handler for it.

Commonly occurring exceptions, categories, and classes:

- For the OCA Java SE 7 Programmer I exam, you need to be aware of commonly occurring exception categories and classes, when are they thrown, whether you should handle them, and how to handle them.
- `ArrayIndexOutOfBoundsException` is a runtime exception that's thrown when a piece of code tries to access an array position out of its bounds—when an array is accessed either at a position less than 0 or at a position greater than or equal to its length.
- `IndexOutOfBoundsException` is a runtime exception that's thrown when a piece of code tries to access a list position that's out of its bounds—when the position is either less than 0 or greater than or equal to the list's size.
- The class `ArrayIndexOutOfBoundsException` extends the class `java.lang.IndexOutOfBoundsException`, which extends the class `java.lang.RuntimeException`.
- In typical programming conditions, the `ArrayIndexOutOfBoundsException` shouldn't be thrown programmatically.
- One of the main reasons for the JVM taking the responsibility for throwing this exception itself is that this exception isn't known until runtime and depends on

the array or list position that's being accessed by a piece of code. Most often, a variable is used to specify this array or list position, and its value may not be known until runtime.

- `ClassCastException` is a runtime exception. `java.lang.ClassCastException` extends `java.lang.RuntimeException`.
- `ClassCastException` is thrown when an object fails an IS-A test with the class type it is being cast to.
- You can use the operator `instanceof` to verify whether an object can be cast to another class before casting it.
- `IllegalArgumentException` is a runtime exception. `java.lang.IllegalArgumentException` extends `java.lang.RuntimeException`.
- `IllegalArgumentException` is thrown to specify that a method has been passed illegal or inappropriate arguments.
- Even though `IllegalArgumentException` is a runtime exception, programmers usually use this exception to validate the arguments that are passed to a method, and the exception constructor is passed a descriptive message specifying the exception details.
- `IllegalStateException` is a runtime exception. `java.lang.IllegalStateException` extends `java.lang.RuntimeException`.
- `IllegalStateException` may be thrown programmatically.
- As a programmer, you can throw `IllegalStateException` to signal to the calling method that the method that's being requested for execution isn't ready to start its execution or is in a state in which it can't execute.
- For example, you can throw `IllegalStateException` from your code if an application tries to modify an SMS that has already been sent.
- `NullPointerException` is a runtime exception. The class `java.lang.NullPointerException` extends `java.lang.RuntimeException`.
- `NullPointerException` is thrown by the JVM if you try to access a method or variable of an uninitialized reference variable.
- `NumberFormatException` is a runtime exception. `java.lang.NumberFormatException` extends `java.lang.IllegalArgumentException`. `java.lang.IllegalArgumentException` extends `java.lang.RuntimeException`.
- You can throw `NumberFormatException` from your own method to indicate that there's an issue with the conversion of a `String` value to a specified numeric format (decimal, octal, hexadecimal, or binary).
- Runtime exceptions arising from any of the following may throw `ExceptionInInitializerError`:
  - Execution of an anonymous `static` block
  - Initialization of a `static` variable
  - Execution of a `static` method (called from either of the previous two items)

- The error `ExceptionInInitializerError` can be thrown only by an object of a runtime exception.
- `ExceptionInInitializerError` can't be thrown if a `static` initializer block throws an object of a checked exception, because the Java compiler is intelligent enough to determine this condition and it doesn't allow you to throw an unhandled checked exception from a `static` initialization block.
- `StackOverflowError` is an error. `java.lang.StackOverflowError` extends `java.lang.VirtualMachineError`.
- Because `StackOverflowError` extends `VirtualMachineError`, it should be left to be managed by the JVM.
- The `StackOverflowError` error is thrown by the JVM when a Java program calls itself so many times that the memory stack allocated to execute the Java program "overflows."
- `NoClassDefFoundError` is an `Error`. `java.lang.NoClassDefFoundError` extends `java.lang.LinkageError`. `java.lang.LinkageError` extends `java.lang.Error`.
- `NoClassDefFoundError` is thrown by the JVM or a `ClassLoader` when it is unable to load the definition of a class required to create an object of the class.
- Don't confuse the exception thrown by `Class.forName()`, used to load the class, and `NoClassDefFoundError` thrown by the JVM. `Class.forName()` throws `ClassNotFoundException`.
- `OutOfMemoryError` is thrown by the JVM when it's unable to create objects on the heap and the garbage collector may not be able to free more memory for the JVM.

7.7 Sample exam questions

Q7-1. What is the output of the following code:

```
class Course {
    String courseName;
    Course() {
        Course c = new Course();
        c.courseName = "Oracle";
    }
}
class EJavaGuruPrivate2 {
    public static void main(String args[]) {
        Course c = new Course();
        c.courseName = "Java";
        System.out.println(c.courseName);
    }
}
```

a The code will print `Java`.

b The code will print `Oracle`.

c The code will not compile.

d The code will throw an exception or an error at runtime.

Q7-2. Select the correct option(s):

- **a** You cannot handle runtime exceptions.
- **b** You should not handle errors.
- **c** If a method throws a checked exception, it must be either handled by the method or specified in its `throws` clause.
- **d** If a method throws a runtime exception, it may include the exception in its `throws` clause.
- **e** Runtime exceptions are checked exceptions.

Q7-3. Examine the following code and select the correct option(s):

```
class EJavaGuruExcep2 {
    public static void main(String args[]) {
        EJavaGuruExcep2 var = new EJavaGuruExcep2();
        var.printArrValues(args);
    }
    void printArrValues(String[] arr) {
        try {
            System.out.println(arr[0] + ":" + arr[1]);
        }
        catch (NullPointerException e) {
            System.out.println("NullPointerException");
        }
        catch (IndexOutOfBoundsException e) {
            System.out.println("IndexOutOfBoundsException");
        }
        catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("ArrayIndexOutOfBoundsException");
        }
    }
}
```

- **a** If the class `EJavaGuruExcep2` is executed using the following command, it prints `NullPointerException`:

  ```
  javaEJavaGuruExcep2
  ```

- **b** If the class `EJavaGuruExcep2` is executed using the following command, it prints `IndexOutOfBoundsException`:

  ```
  javaEJavaGuruExcep2
  ```

- **c** If the class `EJavaGuruExcep2` is executed using the following command, it prints `ArrayIndexOutOfBoundsException`:

  ```
  javaEJavaGuruExcep2one
  ```

- **d** The code will fail to compile.

Q7-4. What is the output of the following code?

```
class EJava {
    void method() {
        try {
```

```
            guru();
            return;
        }
        finally {
            System.out.println("finally 1");
        }
    }
    void guru() {
        System.out.println("guru");
        throw new StackOverflowError();
    }
    public static void main(String args[]) {
        EJava var = new EJava();
        var.method();
    }
}
```

a
```
guru
finally 1
```

b
```
guru
finally 1
Exception in thread "main" java.lang.StackOverflowError
```

c
```
guru
Exception in thread "main" java.lang.StackOverflowError
```

d
```
guru
```

e The code fails to compile.

Q7-5. Select the incorrect statement(s):

a Exceptions enable a developer to define the programming logic separate from the exception-handling code.

b Exception handling speeds up execution of the code.

c Exception handing is used to define code that should execute when a piece of code throws an exception.

d Code that handles all the checked exceptions can still throw unchecked exceptions.

Q7-6. Select the incorrect statement(s):

a `java.lang.Throwable` is the base class of all types of exceptions.

b If a class is a subclass of `java.lang.Exception`, it may or may not be a checked exception.

c `Error` is an unchecked exception.

d `Error` and checked exceptions need not be part of a method signature.

Q7-7. What is the output of the following code?

```
class TryFinally {
   int tryAgain() {
       int a = 10;
```

```
        try {
            ++a;
        }
        finally {
            a++;
        }
        return a;
    }

    public static void main(String args[]) {
        System.out.println(new TryFinally().tryAgain());
    }
}
```

a 10

b 11

c 12

d Compilation error

e Runtime exception

Q7-8. What is the output of the following code?

```
class EJavaBase {
    void myMethod() throws ExceptionInInitializerError {
        System.out.println("Base");
    }
}
class EJavaDerived extends EJavaBase {
    void myMethod() throws RuntimeException {
        System.out.println("Derived");
    }
}
class EJava3 {
    public static void main(String args[]) {
        EJavaBase obj = new EJavaDerived();
        obj.myMethod();
    }
}
```

a `Base`

b `Derived`

c
```
Derived
Base
```

d
```
Base
Derived
```

e Compilation error

Q7-9. Which of the following statements are true?

a A user-defined class may not throw an `IllegalStateException`. It must be thrown only by Java API classes.

b `System.out.println` will throw `NullPointerException` if an uninitialized instance variable of type `String` is passed to it to print its value.

c `NumberFormatException` is thrown by multiple methods from the Java API when invalid numbers are passed on as `String`s to be converted to the specified number format.

d `ExceptionInInitializerError` may be thrown by the JVM when a `static` initializer in your code throws a `NullPointerException`.

Q7-10. What is the output of the following code?

```
class EJava4 {
    void foo() {
        try {
            String s = null;
            System.out.println("1");
            try {
                System.out.println(s.length());
            }
            catch (NullPointerException e) {
                System.out.println("inner");
            }
            System.out.println("2");
        }
        catch (NullPointerException e) {
            System.out.println("outer");
        }
    }
    public static void main(String args[]) {
        EJava4 obj = new EJava4();
        obj.foo();
    }
}
```

a
```
1
inner
2
outer
```

b
```
1
outer
```

c
```
1
inner
```

d
```
1
inner
2
```

7.8 Answers to sample exam questions

Q7-1. What is the output of the following code:

```
class Course {
    String courseName;
    Course() {
        Course c = new Course();
        c.courseName = "Oracle";
    }
```

```
}
class EJavaGuruPrivate2 {
    public static void main(String args[]) {
        Course c = new Course();
        c.courseName = "Java";
        System.out.println(c.courseName);
    }
}
```

a The code will print `Java`.
b The code will print `Oracle`.
c The code will not compile.
d **The code will throw an exception or an error at runtime.**

Answer: d

Explanation: This class will throw `StackOverflowError` at runtime. The easiest way to look for a `StackOverflowError` is to locate recursive method calls. In the question's code, the constructor of the class `Course` creates an object of the class `Course`, which will call the constructor again. Hence, this becomes a recursive call and ends up throwing `StackOverflowError` at runtime. (As you know, an exception or an error can be thrown only at runtime, not compile time.)

Q7-2. Select the correct option(s):

a You cannot handle runtime exceptions.
b **You should not handle errors.**
c **If a method throws a checked exception, it must be either handled by the method or specified in its `throws` clause.**
d **If a method throws a runtime exception, it may include the exception in its `throws` clause.**
e Runtime exceptions are checked exceptions.

Answer: b, c, d

Explanation: Option (a) is incorrect. You can handle runtime exceptions the way you can handle a checked exception in your code: using a `try-catch` block.

Option (b) is correct. You shouldn't try to handle errors in your code. Or, to put it another way, you can't do much when an error is thrown by your code. Instead of trying to handle errors in your code, you should resolve the code that results in these errors. For example, `StackOverflowError` is an error that will be thrown by your code if your code executes a method recursively without any exit condition. This repetition will consume all the space on the stack and result in a `StackOverflowError`.

Option (c) is correct. If you fail to implement either of these options, your code won't compile.

Option (d) is correct. It isn't mandatory for runtime exceptions to be included in a method's `throws` clause. Usually this inclusion is unnecessary, but if you do include it, your code will execute without any issues.

Option (e) is incorrect. Runtime exception and all its subclasses are *not* checked exceptions.

Q7-3. Examine the following code and select the correct option(s):

```
class EJavaGuruExcep2 {
    public static void main(String args[]) {
        EJavaGuruExcep2 var = new EJavaGuruExcep2();
        var.printArrValues(args);
    }
    void printArrValues(String[] arr) {
        try {
            System.out.println(arr[0] + ":" + arr[1]);
        }
        catch (NullPointerException e) {
            System.out.println("NullPointerException");
        }
        catch (IndexOutOfBoundsException e) {
            System.out.println("IndexOutOfBoundsException");
        }
        catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("ArrayIndexOutOfBoundsException");
        }
    }
}
```

a If the class `EJavaGuruExcep2` is executed using the following command, it prints `NullPointerException`:

   ```
   javaEJavaGuruExcep2
   ```

b If the class `EJavaGuruExcep2` is executed using the following command, it prints `IndexOutOfBoundsException`:

   ```
   javaEJavaGuruExcep2
   ```

c If the class `EJavaGuruExcep2` is executed using the following command, it prints `ArrayIndexOutOfBoundsException`:

   ```
   javaEJavaGuruExcep2one
   ```

d **The code will fail to compile.**

Answer: d

Explanation: The key to answering this question is to be aware of the following two facts:

- Exceptions are classes. If an exception's base class is used in a `catch` block, it can catch all the exceptions of its derived class. If you try to catch an exception from its derived class afterward, the code won't compile.
- `ArrayIndexOutOfBoundsException` is a derived class of `IndexOutOfBounds-Exception`.

The rest of the points try to trick you into believing that the question is based on the arguments passed to a `main` method.

Q7-4. What is the output of the following code?

```
class EJava {
    void method() {
        try {
            guru();
            return;
        }
        finally {
            System.out.println("finally 1");
        }
    }
    void guru() {
        System.out.println("guru");
        throw new StackOverflowError();
    }
    public static void main(String args[]) {
        EJava var = new EJava();
        var.method();
    }
}
```

a
```
guru
finally 1
```

b
```
guru
finally 1
Exception in thread "main" java.lang.StackOverflowError
```

c
```
guru
Exception in thread "main" java.lang.StackOverflowError
```

d
```
guru
```

e The code fails to compile.

Answer: b

Explanation: No compilation errors exist with the code.

The method `guru` throws `StackOverflowError`, which is not a checked exception. Even though your code should not throw an error, it is possible syntactically. Your code will compile successfully.

The call to the method `guru` is immediately followed by the keyword `return`, which is supposed to end the execution of the method `method`. But the call to `guru` is placed within a `try-catch` block, with a `finally` block. Because `guru` doesn't handle the error `StackOverflowError` itself, the control looks for the exception handler in the method `method`. This calling method doesn't handle this error, but defines a `finally` block. The control then executes the `finally` block. Because the code can't find an appropriate handler to handle this error, it propagates to the JVM, which abruptly halts the code.

Q7-5. Select the incorrect statement(s):

a Exceptions enable a developer to define the programming logic separate from the exception-handling code.

b **Exception handling speeds up execution of the code.**
c Exception handing is used to define code that should execute when a piece of code throws an exception.
d Code that handles all the checked exceptions can still throw unchecked exceptions.

Answer: b

Explanation: No direct relationship exists between exception handling and improved execution of code. Code that handles all the checked exceptions can throw unchecked exceptions and vice versa.

Q7-6. Select the incorrect statement(s):

a `java.lang.Throwable` is the base class of all type of exceptions.
b If a class is a subclass of `java.lang.Exception`, it may or may not be a checked exception.
c **`Error` is an unchecked exception.**
d **`Error` and checked exceptions need not be part of a method signature.**

Answer: c, d

Explanation:

Option (a) is a true statement. A checked exception is a subclass of `java.lang.Exception`, and a runtime exception is a subclass of `java.lang.RuntimeException`. `java.lang.RuntimeException` is a subclass of `java.lang.Exception`, and `java.lang.Exception` is a subclass of `java.lang.Throwable`. Hence, all the exceptions are subclasses of `java.lang.Throwable`.

Option (b) is also a true statement. Unchecked exceptions are subclasses of class `java.lang.RuntimeException`, which itself is a subclass of `java.lang.Exception`. Hence, a class can be a subclass of class `java.lang.Exception` and either a checked or an unchecked exception.

Option (c) is a false statement. `Error` is *not* an exception. It does not subclass `java.lang.Exception`.

Option (d) is also a false statement. `Error` need not be part of a method signature, but checked exceptions must be a part of the method signatures.

Q7-7. What is the output of the following code?

```
class TryFinally {
    int tryAgain() {
        int a = 10;
        try {
            ++a;
        }
        finally {
            a++;
        }
        return a;
    }
```

```
    public static void main(String args[]) {
        System.out.println(new TryFinally().tryAgain());
    }
}
```

a 10

b 11

c **12**

d Compilation error

e Runtime exception

Answer: c

Explanation: The `try` block executes, incrementing the value of variable `a` to `11`. This step is followed by execution of the `finally` block, which also increments the value of variable `a` by `1`, to `12`. The method `tryAgain` returns the value `12`, which is printed by the method `main`.

There are no compilation issues with the code. A `try` block can be followed by a `finally` block, without any `catch` blocks. Even though the `try` block doesn't throw any exceptions, it compiles successfully. The following is an example of a `try-catch` block that won't compile because it tries to *catch* a checked exception that's never thrown by the `try` block:

```
try {
    ++a;
}
catch (java.io.FileNotFoundException e) {
}
```

Q7-8. What is the output of the following code?

```
class EJavaBase {
    void myMethod() throws ExceptionInInitializerError {
        System.out.println("Base");
    }
}
class EJavaDerived extends EJavaBase {
    void myMethod() throws RuntimeException {
        System.out.println("Derived");
    }
}
class EJava3 {
    public static void main(String args[]) {
        EJavaBase obj = new EJavaDerived();
        obj.myMethod();
    }
}
```

a `Base`

b `Derived`

c `Derived`
`Base`

d
```
Base
Derived
```

e Compilation error

Answer: b

Explanation: The rule that if a base class method doesn't throw an exception, an overriding method in the derived class can't throw a exception applies only to checked exceptions. It doesn't apply to runtime (unchecked) exceptions or errors. A base or overridden method is free to throw any `Error` or runtime exception.

Q7-9. Which of the following statements are true?

a A user-defined class may not throw an `IllegalStateException`. It must be thrown only by Java API classes.

b `System.out.println` will throw `NullPointerException` if an uninitialized instance variable of type `String` is passed to it to print its value.

c **`NumberFormatException` is thrown by multiple methods from the Java API when invalid numbers are passed on as `String`s to be converted to the specified number format.**

d **`ExceptionInInitializerError` may be thrown by the JVM when a `static` initializer in your code throws a `NullPointerException`.**

Answer: c, d

Option (a) is incorrect. A user-defined class can throw any exception from the Java API.

Option (b) is incorrect. An uninitialized instance variable of type `String` will be assigned a default value of `null`. When you pass this variable to `System.out.println` to print it, it will print `null`. If you try to access any member (variable or method) of this `null` object, then `NullPointerException` will be thrown.

Q7-10. What is the output of the following code?

```
class EJava4 {
    void foo() {
        try {
            String s = null;
            System.out.println("1");
            try {
                System.out.println(s.length());
            }
            catch (NullPointerException e) {
                System.out.println("inner");
            }
            System.out.println("2");
        }
        catch (NullPointerException e) {
            System.out.println("outer");
        }
    }
    public static void main(String args[]) {
```

```
        EJava4 obj = new EJava4();
        obj.foo();
    }
}
```

a
```
1
inner
2
outer
```

b
```
1
outer
```

c
```
1
inner
```

d
```
1
inner
2
```

Answer: d

Explanation: First of all, nested `try-catch` statements don't throw compilation errors.

Because the variable `s` hasn't been initialized, an attempt to access its method `length()` will throw a `NullPointerException`. The inner `try-catch` block handles this exception and prints `inner`. The control then moves on to complete the remaining code in the outer `try-catch` block, printing `2`. Because the `NullPointerException` was already handled in the inner `try-catch` block, it's not handled in the outer `try-catch` block.