

# Chapter 10

## Exceptions

### OCJP EXAM OBJECTIVES COVERED IN THIS CHAPTER:

- **Handling Exceptions**
  - Describe the advantages of Exception handling and differentiate among checked, unchecked exceptions, and Errors
  - Create try-catch blocks and determine how exceptions alter program flow
  - Create and invoke a method that throws an exception

Many things can go wrong in a program. Java uses exceptions to deal with some of these scenarios. This chapter focuses on how exceptions are created, how to handle them, and how to distinguish between various types of exceptions and errors.

### Understanding Exceptions

A program can fail for just about any reason. Here are just a few possibilities:

- The code tries to connect to a website, but the Internet connection is down.
- You made a coding mistake and tried to access an invalid index in an array.
- One method calls another with a value that the method doesn't support.

As you can see, some of these are coding mistakes. Others are completely beyond your control. Your program can't help it if the Internet connection goes down. What it *can* do is deal with the situation.

First, we'll look at the role of exceptions. Then we'll cover the various types of exceptions, followed by an explanation of how to throw an exception in Java.

### The Role of Exceptions

An *exception* is Java's way of saying, "I give up. I don't know what to do right now. You deal with it." When you write a method, you can either deal with the exception or make it the calling code's problem.

As an example, think of Java as a child who visits the zoo. The *happy path* is when nothing goes wrong. The child continues to look at the animals until the program nicely ends. Nothing went wrong, and there were no exceptions to deal with.

This child's younger sister doesn't experience the happy path. In all the excitement she trips and falls. Luckily, it isn't a bad fall. The little girl gets up and proceeds to look at more animals. She has handled the issue all by herself. Unfortunately, she falls again later in the day and starts crying. This time, she has declared she needs help by crying. The story ends well. Her daddy rubs her knee and gives her a hug. Then they go back to seeing more animals and enjoy the rest of the day.

These are the two approaches Java uses when dealing with exceptions. A method can handle the exception case itself or make it the caller's responsibility. You saw both in the trip to the zoo.

You saw an exception in [Chapter 1](#), "Welcome to Java," with a simple Zoo example. You wrote a class that printed out the name of the zoo:

```
1: public class Zoo {  
2:     public static void main(String[] args) {  
3:         System.out.println(args[0]);  
4:         System.out.println(args[1]);  
5:     } }
```

Then you tried to call it without enough arguments:

```
$ javac Zoo.java  
$ java Zoo Zoo
```

On line 4, Java realized there's only one element in the array and index 1 is not allowed. Java threw up its hands in defeat and threw an exception. It didn't try to handle the exception. It just said, "I can't deal with it," and the exception was displayed:

```
Zoo  
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 1 out of bounds for  
    at Zoo.main(Zoo.java:4)
```

Exceptions can and do occur all the time, even in solid program code. In our example, toddlers falling are a fact of life. When you write more advanced programs, you'll need to deal with failures in accessing files, networks, and outside services. On the exam, exceptions deal largely with mistakes in programs. For example, a program might try to access an invalid position in an array. The key point to remember is that exceptions alter the program flow.



Exceptions are used when “something goes wrong.” However, the word *wrong* is subjective. The following code returns -1 instead of throwing an exception if no match is found:

```
public int indexOf(String[] names, String name) {
    for (int i = 0; i < names.length; i++) {
        if (names[i].equals(name)) { return i; }
    }
    return -1;
}
```

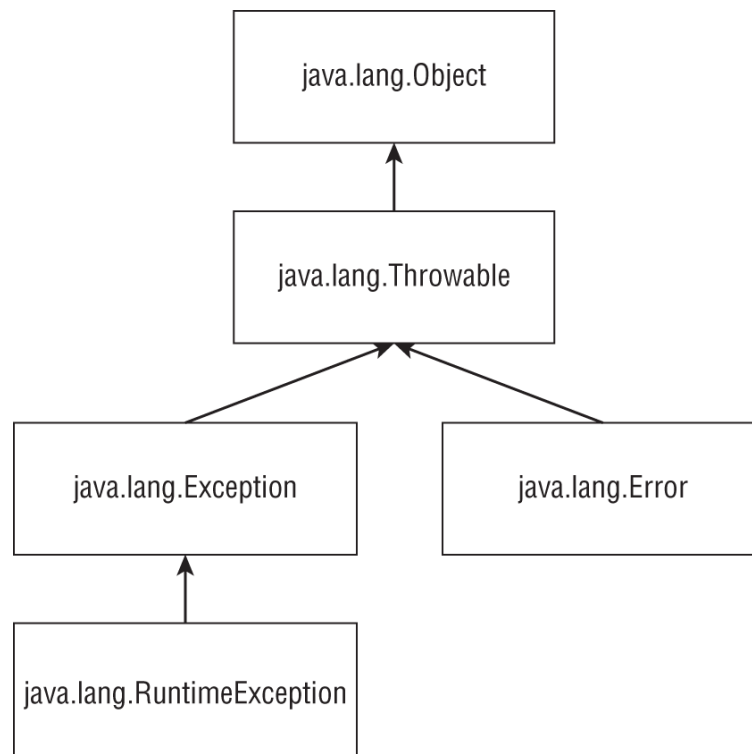
This approach is common when writing a method that does a search. For example, imagine being asked to find the name `Joe` in the array. It is perfectly reasonable that `Joe` might not appear in the array. When this happens, a special value is returned. An exception should be reserved for exceptional conditions like `names` being `null`.

In general, try to avoid return codes. Return codes are commonly used in searches, so programmers are expecting them. In other methods, you will take your callers by surprise by returning a special value. An exception forces the program to deal with the problem or end with the exception if left unhandled, whereas a return code could be accidentally ignored and cause problems later in the program. Even worse, a return value could be confused with real data. In the context of a school, does `-1` mean an error or the number of students removed from a class? An exception is like shouting, “Deal with me!” and avoids possible ambiguity.

---

## Understanding Exception Types

As we’ve explained, an exception is an event that alters program flow. Java has a `Throwable` superclass for all objects that represent these events. Not all of them have the word *exception* in their class name, which can be confusing. [Figure 10.1](#) shows the key subclasses of `Throwable`.



**FIGURE 10.1** Categories of exception

Error means something went so horribly wrong that your program should not attempt to recover from it. For example, the disk drive “disappeared” or the program ran out of memory. These are abnormal conditions that you aren’t likely to encounter and cannot recover from.

For the exam, the only thing you need to know about `Throwable` is that it’s the parent class of all exceptions, including the `Error` class. While you can handle `Throwable` and `Error` exceptions, it is not recommended you do so in your application code. In this chapter, when we refer to exceptions, we generally mean any class that inherits `Throwable`, although we are almost always working with the `Exception` class or subclasses of it.

### Checked Exceptions

A *checked exception* is an exception that must be declared or handled by the application code where it is thrown. In Java, checked exceptions all inherit `Exception` but not `RuntimeException`. Checked exceptions tend to be more anticipated—for example, trying to read a file that doesn’t exist.



Checked exceptions also include any class that inherits `Throwable`, but not `Error` or `RuntimeException`. For example, a class that directly extends `Throwable` would be a checked exception. For the exam, though, you just need to know about checked exceptions that extend `Exception`.

---

Checked exceptions? What are we checking? Java has a rule called the *handle or declare rule*. The *handle or declare rule* means that all checked exceptions that could be thrown within a method are either wrapped in compatible `try` and `catch` blocks or declared in the method signature.

Because checked exceptions tend to be anticipated, Java enforces the rule that the programmer must do something to show the exception was thought about. Maybe it was handled in the method. Or maybe the method declares that it can't handle the exception and someone else should.



While only checked exceptions must be handled or declared in Java, unchecked exceptions (which we will present in the next section) may also be handled or declared. The distinction is that checked exceptions must be handled or declared, while unchecked exceptions can be optionally handled or declared.

---

Let's take a look at an example. The following `fall()` method declares that it might throw an `IOException`, which is a checked exception:

```
void fall(int distance) throws IOException {
    if(distance > 10) {
        throw new IOException();
    }
}
```

Notice that you're using two different keywords here. The `throw` keyword tells Java that you want to throw an `Exception`, while the `throws` keyword simply declares that the method might throw an `Exception`. It also might not. You will see the `throws` keyword again later in the chapter.

Now that you know how to declare an exception, how do you instead handle it? The following alternate version of the `fall()` method handles the exception:

```
void fall(int distance) {
    try {
        if(distance > 10) {
            throw new IOException();
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Notice that the `catch` statement uses `Exception`, not `IOException`. Since `IOException` is a subclass of `Exception`, the `catch` block is al-

lowed to catch it. We'll cover `try` and `catch` blocks in more detail later in this chapter.

## Unchecked Exceptions

An *unchecked exception* is any exception that does not need to be declared or handled by the application code where it is thrown. Unchecked exceptions are often referred to as runtime exceptions, although in Java, unchecked exceptions include any class that inherits `RuntimeException` or `Error`.

A *runtime exception* is defined as the `RuntimeException` class and its subclasses. Runtime exceptions tend to be unexpected but not necessarily fatal. For example, accessing an invalid array index is unexpected. Even though they do inherit the `Exception` class, they are not checked exceptions.

---

### RUNTIME VS. AT THE TIME THE PROGRAM IS RUN

A runtime (unchecked) exception is a specific type of exception. All exceptions occur at the time that the program is run. (The alternative is compile time, which would be a compiler error.) People don't refer to them as "run time" exceptions because that would be too easy to confuse with runtime! When you see *runtime*, it means unchecked.

---

An unchecked exception can often occur on nearly any line of code, as it is not required to be handled or declared. For example, a `NullPointerException` can be thrown in the body of the following method if the `input` reference is `null`:

```
void fall(String input) {  
    System.out.println(input.toLowerCase());  
}
```

We work with objects in Java so frequently, a `NullPointerException` can happen almost anywhere. If you had to declare unchecked exceptions everywhere, every single method would have that clutter! The code will compile if you declare an unchecked exception. However, it is redundant.

---

#### CHECKED VS. UNCHECKED (RUNTIME) EXCEPTIONS

In the past, developers used checked exceptions more often than they do now. According to Oracle, they are intended for issues a programmer “might reasonably be expected to recover from.” Then developers started writing code where a chain of methods kept declaring the same exception and nobody actually handled it. Some libraries started using unchecked exceptions for issues a programmer might reasonably be expected to recover from. Many programmers can hold a debate with you on which approach is better. For the exam, you need to know the rules for how checked versus unchecked exceptions function. You don’t have to decide philosophically whether an exception should be checked or unchecked.

---

### Throwing an Exception

Any Java code can throw an exception; this includes code you write. The exam is limited to exceptions that someone else has created. Most likely, they will be exceptions that are provided with Java. You might encounter an exception that was made up for the exam. This is fine. The question will make it obvious that these are exceptions by having the class name end with `Exception`. For example, `MyMadeUpException` is clearly an exception.

On the exam, you will see two types of code that result in an exception. The first is code that’s wrong. Here’s an example:

```
String[] animals = new String[0];
System.out.println(animals[0]);
```

This code throws an `ArrayIndexOutOfBoundsException` since the array has no elements. That means questions about exceptions can be hidden in questions that appear to be about something else.



On the exam, many questions have a choice about not compiling and about throwing an exception. Pay special attention to code that calls a method on a `null` reference or that references an invalid array or `List` index. If you spot this, you know the correct answer is that the code throws an exception at runtime.

---

The second way for code to result in an exception is to explicitly request Java to throw one. Java lets you write statements like these:

```
throw new Exception();
throw new Exception("Ow! I fell.");
throw new RuntimeException();
throw new RuntimeException("Ow! I fell.");
```

The `throw` keyword tells Java you want some other part of the code to deal with the exception. This is the same as the young girl crying for her daddy. Someone else needs to figure out what to do about the exception.

---

#### THROW VS. THROWS

Anytime you see `throw` or `throws` on the exam, make sure the correct one is being used. The `throw` keyword is used as a statement inside a code block to throw a new exception or rethrow an existing exception, while the `throws` keyword is used only at the end of a method declaration to indicate what exceptions it supports. On the exam, you might start reading a long class definition only to realize the entire thing does not compile due to the wrong keyword being used.

---

When creating an exception, you can usually pass a `String` parameter with a message, or you can pass no parameters and use the defaults. We say *usually* because this is a convention. Someone could create an exception class that does not have a constructor that takes a message. The first two examples create a new object of type `Exception` and throw it. The last two show that the code looks the same regardless of which type of exception you throw.

Additionally, you should know that an `Exception` is an `Object`. This means you can store in a variable, and this is legal:

```
Exception e = new RuntimeException();
throw e;
```

The code instantiates an exception on one line and then throws on the next. The exception can come from anywhere, even passed into a method. As long as it is a valid exception, it can be thrown.

The exam might also try to trick you. Do you see why this code doesn't compile?

```
throw RuntimeException();    // DOES NOT COMPILE
```

If your answer is that there is a missing keyword, you're absolutely right. The exception is never instantiated with the `new` keyword.

Let's take a look at another place the exam might try to trick you. Can you see why the following does not compile?

```
3: try {
4:     throw new RuntimeException();
5:     throw new ArrayIndexOutOfBoundsException(); // DOES NOT COMPILE
6: } catch (Exception e) {
7: }
```



Since line 4 throws an exception, line 5 can never be reached during runtime. The compiler recognizes this and reports an unreachable code error.

The types of exceptions are important. Be sure to closely study everything in [Table 10.1](#). Remember that a `Throwable` is either an `Exception` or an `Error`. You should not catch `Throwable` directly in your code.

**TABLE 10.1** Types of exceptions and errors

Type	How to recognize	Okay for program to catch?	Is program required to handle or declare?
Runtime exception	Subclass of <code>RuntimeException</code>	Yes	No
Checked exception	Subclass of <code>Exception</code> but not subclass of <code>RuntimeException</code>	Yes	Yes
Error	Subclass of <code>Error</code>	No	No

## Recognizing Exception Classes

You need to recognize three groups of exception classes for the exam: `RuntimeException`, checked `Exception`, and `Error`. We'll look at common examples of each type. For the exam, you'll need to recognize which type of an exception it is and whether it's thrown by the Java virtual machine (JVM) or a programmer. So that you can recognize them, we'll show you some code examples for those exceptions. For some exceptions, you also need to know which are inherited from one another.

### *RuntimeException* Classes

`RuntimeException` and its subclasses are unchecked exceptions that don't have to be handled or declared. They can be thrown by the programmer or by the JVM. Common `RuntimeException` classes include the following:

**`ArithmeticException`** Thrown when code attempts to divide by zero

**`ArrayIndexOutOfBoundsException`** Thrown when code uses an illegal index to access an array

**`ClassCastException`** Thrown when an attempt is made to cast an object to a class of which it is not an instance

**`NullPointerException`** Thrown when there is a `null` reference where an object is required

**IllegalArgumentException** Thrown by the programmer to indicate that a method has been passed an illegal or inappropriate argument

**NumberFormatException** Subclass of `IllegalArgumentException` thrown when an attempt is made to convert a string to a numeric type but the string doesn't have an appropriate format

### ***ArithmeticException***

Trying to divide an `int` by zero gives an undefined result. When this occurs, the JVM will throw an `ArithmeticException`:

```
int answer = 11 / 0;
```

Running this code results in the following output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

Java doesn't spell out the word *divide*. That's okay, though, because we know that `/` is the division operator and that Java is trying to tell you division by zero occurred.

The thread `"main"` is telling you the code was called directly or indirectly from a program with a `main` method. On the exam, this is all the output you will see. Next comes the name of the exception, followed by extra information (if any) that goes with the exception.

### ***ArrayIndexOutOfBoundsException***

You know by now that array indexes start with `0` and go up to `1` less than the length of the array—which means this code will throw an `ArrayIndexOutOfBoundsException`:

```
int[] countsOfMoose = new int[3];
System.out.println(countsOfMoose[-1]);
```

This is a problem because there's no such thing as a negative array index. Running this code yields the following output:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
Index -1 out of bounds for length 3
```

At least Java tells us what index was invalid. Can you see what's wrong with this one?

```
int total = 0;
int[] countsOfMoose = new int[3];
for (int i = 0; i <= countsOfMoose.length; i++)
    total += countsOfMoose[i];
```

The problem is that the `for` loop should have `<` instead of `<=`. On the final iteration of the loop, Java tries to call `countsOfMoose[3]`, which is invalid. The array includes only three elements, making `2` the largest possible index. The output looks like this:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
Index 3 out of bounds for length 3
```

### ***ClassCastException***

Java tries to protect you from impossible casts. This code doesn't compile because `Integer` is not a subclass of `String`:

```
String type = "moose";
Integer number = (Integer) type; // DOES NOT COMPILE
```

More complicated code thwarts Java's attempts to protect you. When the cast fails at runtime, Java will throw a `ClassCastException`:

```
String type = "moose";
Object obj = type;
Integer number = (Integer) obj;
```

The compiler sees a cast from `Object` to `Integer`. This could be okay. The compiler doesn't realize there's a `String` in that `Object`. When the code runs, it yields the following output:

```
Exception in thread "main" java.lang.ClassCastException: java.base/java.lang.String
cannot be cast to java.lang.base/java.lang.Integer
```

Java tells you both types that were involved in the problem, making it apparent what's wrong.

### ***NullPointerException***

Instance variables and methods must be called on a non-`null` reference. If the reference is `null`, the JVM will throw a `NullPointerException`. It's usually subtle, such as in the following example, which checks whether you remember instance variable references default to `null`:

```
String name;
public void printLength() {
    System.out.println(name.length());
}
```

Running this code results in this output:

```
Exception in thread "main" java.lang.NullPointerException
```

### ***IllegalArgumentException***

`IllegalArgumentException` is a way for your program to protect itself. You first saw the following setter method in the `Swan` class in [Chapter 7](#), “Methods and Encapsulation.”

```
6: public void setNumberEggs(int numberEggs) { // setter
7:     if (numberEggs >= 0) // guard condition
8:         this.numberEggs = numberEggs;
9: }
```

This code works, but you don’t really want to ignore the caller’s request when they tell you a `Swan` has `-2` eggs. You want to tell the caller that something is wrong—preferably in an obvious way that the caller can’t ignore so that the programmer will fix the problem. Exceptions are an efficient way to do this. Seeing the code end with an exception is a great reminder that something is wrong:

```
public void setNumberEggs(int numberEggs) {
    if (numberEggs < 0)
        throw new IllegalArgumentException(
            "# eggs must not be negative");
    this.numberEggs = numberEggs;
}
```

The program throws an exception when it’s not happy with the parameter values. The output looks like this:

```
Exception in thread "main"
java.lang.IllegalArgumentException: # eggs must not be negative
```

Clearly this is a problem that must be fixed if the programmer wants the program to do anything useful.

### ***NumberFormatException***

Java provides methods to convert strings to numbers. When these are passed an invalid value, they throw a `NumberFormatException`. The idea is similar to `IllegalArgumentException`. Since this is a common problem, Java gives it a separate class. In fact, `NumberFormatException` is a subclass of `IllegalArgumentException`. Here’s an example of trying to convert something non-numeric into an `int`:

```
Integer.parseInt("abc");
```

The output looks like this:

```
Exception in thread "main"
java.lang.NumberFormatException: For input string: "abc"
```

For the exam, you need to know that `NumberFormatException` is a subclass of `IllegalArgumentException`. We’ll cover more about why that is important later in the chapter.

## Checked *Exception* Classes

Checked exceptions have `Exception` in their hierarchy but not `RuntimeException`. They must be handled or declared. Common checked exceptions include the following:

**`IOException`** Thrown programmatically when there's a problem reading or writing a file

**`FileNotFoundException`** Subclass of `IOException` thrown programmatically when code tries to reference a file that does not exist

For the exam, you need to know that these are both checked exceptions. You also need to know that `FileNotFoundException` is a subclass of `IOException`. You'll see shortly why that matters.

## *Error* Classes

Errors are unchecked exceptions that extend the `Error` class. They are thrown by the JVM and should not be handled or declared. Errors are rare, but you might see these:

**`ExceptionInInitializerError`** Thrown when a static initializer throws an exception and doesn't handle it

**`StackOverflowError`** Thrown when a method calls itself too many times (This is called *infinite recursion* because the method typically calls itself without end.)

**`NoClassDefFoundError`** Thrown when a class that the code uses is available at compile time but not runtime

### *ExceptionInInitializerError*

Java runs `static` initializers the first time a class is used. If one of the `static` initializers throws an exception, Java can't start using the class. It declares defeat by throwing an `ExceptionInInitializerError`. This code throws an `ArrayIndexOutOfBoundsException` in a `static` initializer:

```
static {
    int[] countsOfMoose = new int[3];
    int num = countsOfMoose[-1];
}
public static void main(String... args) { }
```

This code yields information about the error and the underlying exception:

```
Exception in thread "main" java.lang.ExceptionInInitializerError
Caused by: java.lang.ArrayIndexOutOfBoundsException: -1 out of bounds for length 3
```

When executed, you get an `ExceptionInInitializerError` because the error happened in a `static` initializer. That information alone wouldn't

be particularly useful in fixing the problem. Therefore, Java also tells you the original cause of the problem: the `ArrayIndexOutOfBoundsException` that you need to fix.

The `ExceptionInInitializerError` is an error because Java failed to load the whole class. This failure prevents Java from continuing.

### ***StackOverflowError***

When Java calls methods, it puts parameters and local variables on the stack. After doing this a very large number of times, the stack runs out of room and overflows. This is called a `StackOverflowError`. Most of the time, this error occurs when a method calls itself.

```
public static void doNotCodeThis(int num) {  
    doNotCodeThis(1);  
}
```

The output contains this line:

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

Since the method calls itself, it will never end. Eventually, Java runs out of room on the stack and throws the error. This is called infinite recursion. It is better than an infinite loop because at least Java will catch it and throw the error. With an infinite loop, Java just uses all your CPU until you can kill the program.

### ***NoClassDefFoundError***

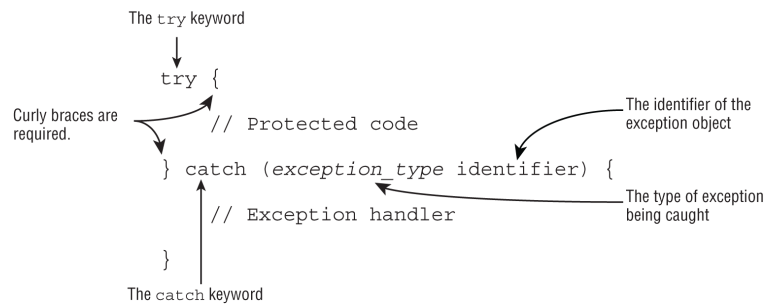
A `NoClassDefFoundError` occurs when Java can't find the class at run-time. Generally, this means a library available when the code was compiled is not available when the code is executed.

## **Handling Exceptions**

What do you do when you encounter an exception? How do you handle or recover from the exception? In this section, we will show the various statements in Java that support handling exceptions and ensuring certain code, like closing a resource, is always executed.

### **Using *try* and *catch* Statements**

Now that you know what exceptions are, let's explore how to handle them. Java uses a `try` statement to separate the logic that might throw an exception from the logic to handle that exception. [Figure 10.2](#) shows the syntax of a *try statement*.



**FIGURE 10.2** The syntax of a try statement

The code in the `try` block is run normally. If any of the statements throws an exception that can be caught by the exception type listed in the `catch` block, the `try` block stops running and execution goes to the `catch` statement. If none of the statements in the `try` block throws an exception that can be caught, the *catch clause* is not run.

You probably noticed the words *block* and *clause* used interchangeably. The exam does this as well, so get used to it. Both are correct. *Block* is correct because there are braces present. *Clause* is correct because they are part of a `try` statement.

There aren't a ton of syntax rules here. The curly braces are required for the `try` and `catch` blocks.

In our example, the little girl gets up by herself the first time she falls. Here's what this looks like:

```

3: void explore() {
4:     try {
5:         fall();
6:         System.out.println("never get here");
7:     } catch (RuntimeException e) {
8:         getUp();
9:     }
10:    seeAnimals();
11: }
12: void fall() { throw new RuntimeException(); }
  
```

First, line 5 calls the `fall()` method. Line 12 throws an exception. This means Java jumps straight to the `catch` block, skipping line 6. The girl gets up on line 8. Now the `try` statement is over, and execution proceeds normally with line 10.

Now let's look at some invalid `try` statements that the exam might try to trick you with. Do you see what's wrong with this one?

```

try // DOES NOT COMPILE
    fall();
catch (Exception e)
    System.out.println("get up");
  
```

The problem is that the braces `{}` are missing. It needs to look like this:

```

try {
    fall();
} catch (Exception e) {
    System.out.println("get up");
}

```

The `try` statements are like methods in that the curly braces are required even if there is only one statement inside the code blocks, while `if` statements and loops are special and allow you to omit the curly braces.

What about this one?

```

try { // DOES NOT COMPILE
    fall();
}

```

This code doesn't compile because the `try` block doesn't have anything after it. Remember, the point of a `try` statement is for something to happen if an exception is thrown. Without another clause, the `try` statement is lonely. As you will see shortly, there is a special type of `try` statement that includes an implicit `finally` block, although the syntax for this is quite different from this example.

## Chaining *catch* Blocks

So far, you have been catching only one type of exception. Now let's see what happens when different types of exceptions can be thrown from the same `try/catch` block.

For the exam, you won't be asked to create your own exception, but you may be given exception classes and need to understand how they function. Here's how to tackle them. First, you must be able to recognize if the exception is a checked or an unchecked exception. Second, you need to determine whether any of the exceptions are subclasses of the others.

```

class AnimalsOutForAWalk extends RuntimeException { }
class ExhibitClosed extends RuntimeException { }
class ExhibitClosedForLunch extends ExhibitClosed { }

```

In this example, there are three custom exceptions. All are unchecked exceptions because they directly or indirectly extend `RuntimeException`. Now we chain both types of exceptions with two `catch` blocks and handle them by printing out the appropriate message:

```

public void visitPorcupine() {
    try {
        seeAnimal();
    } catch (AnimalsOutForAWalk e) { // first catch block
        System.out.print("try back later");
    } catch (ExhibitClosed e) { // second catch block
        System.out.print("not today");
    }
}

```



```

    }
}

```

There are three possibilities for when this code is run. If `seeAnimal()` doesn't throw an exception, nothing is printed out. If the animal is out for a walk, only the first `catch` block runs. If the exhibit is closed, only the second `catch` block runs. It is not possible for both `catch` blocks to be executed when chained together like this.

A rule exists for the order of the `catch` blocks. Java looks at them in the order they appear. If it is impossible for one of the `catch` blocks to be executed, a compiler error about unreachable code occurs. For example, this happens when a superclass `catch` block appears before a subclass `catch` block. Remember, we warned you to pay attention to any subclass exceptions.

In the porcupine example, the order of the `catch` blocks could be reversed because the exceptions don't inherit from each other. And yes, we have seen a porcupine be taken for a walk on a leash.

The following example shows exception types that do inherit from each other:

```

public void visitMonkeys() {
    try {
        seeAnimal();
    } catch (ExhibitClosedForLunch e) { // subclass exception
        System.out.print("try back later");
    } catch (ExhibitClosed e) { // superclass exception
        System.out.print("not today");
    }
}

```

If the more specific `ExhibitClosedForLunch` exception is thrown, the first `catch` block runs. If not, Java checks whether the superclass `ExhibitClosed` exception is thrown and catches it. This time, the order of the `catch` blocks does matter. The reverse does not work.

```

public void visitMonkeys() {
    try {
        seeAnimal();
    } catch (ExhibitClosed e) {
        System.out.print("not today");
    } catch (ExhibitClosedForLunch e) { // DOES NOT COMPILE
        System.out.print("try back later");
    }
}

```

This time, if the more specific `ExhibitClosedForLunch` exception is thrown, the `catch` block for `ExhibitClosed` runs—which means there is no way for the second `catch` block to ever run. Java correctly tells you there is an unreachable `catch` block.

Let's try this one more time. Do you see why this code doesn't compile?

```

public void visitSnakes() {
    try {
    } catch (IllegalArgumentException e) {
    } catch (NumberFormatException e) { // DOES NOT COMPILE
    }
}

```

Remember we said earlier you needed to know that

`NumberFormatException` is a subclass of `IllegalArgumentException`?

This example is the reason why. Since `NumberFormatException` is a subclass, it will always be caught by the first `catch` block, making the second `catch` block unreachable code that does not compile. Likewise, for the exam you need to know that `FileNotFoundException` is subclass of `IOException` and cannot be used in a similar manner.

To review multiple `catch` blocks, remember that at most one `catch` block will run, and it will be the first `catch` block that can handle it. Also, remember that an exception defined by the `catch` statement is only in scope for that `catch` block. For example, the following causes a compiler error since it tries to use the exception class outside the block for which it was defined:

```

public void visitManatees() {
    try {
    } catch (NumberFormatException e1) {
        System.out.println(e1);
    } catch (IllegalArgumentException e2) {
        System.out.println(e1); // DOES NOT COMPILE
    }
}

```

## Applying a Multi-catch Block

Oftentimes, we want the result of an exception being thrown to be the same, regardless of which particular exception is thrown. For example, take a look at this method:

```

public static void main(String args[]) {
    try {
        System.out.println(Integer.parseInt(args[1]));
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Missing or invalid input");
    } catch (NumberFormatException e) {
        System.out.println("Missing or invalid input");
    }
}

```

Notice that we have the same `println()` statement for two different `catch` blocks. How can you reduce the duplicate code? One way is to have the related exception classes all inherit the same interface or extend the same class. For example, you can have a single `catch` block that just catches `Exception`. This will catch everything and anything. Another

way is to move the `println()` statements into a separate method and have every related `catch` block call that method.

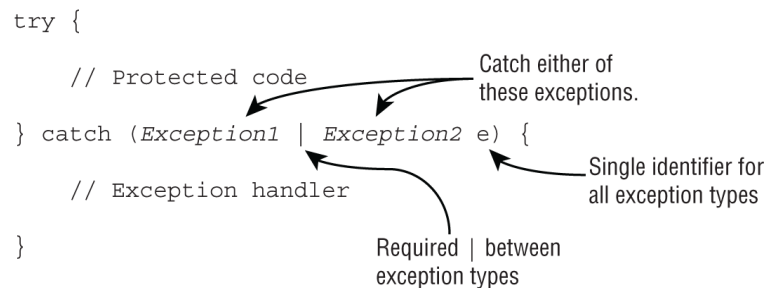
While these solutions are valid, Java provides another structure to handle this more gracefully called a *multi-catch* block. A multi-catch block allows multiple exception types to be caught by the same `catch` block. Let's re-write the previous example using a multi-catch block:

```
public static void main(String[] args) {  
    try {  
        System.out.println(Integer.parseInt(args[1]));    } catch (ArrayIndexOutOfBoundsException  
  
        System.out.println("Missing or invalid input");  
    }  
}
```

This is much better. There's no duplicate code, the common logic is all in one place, and the logic is exactly where you would expect to find it. If you wanted, you could still have a second `catch` block for `Exception` in case you want to handle other types of exceptions differently.

[Figure 10.3](#) shows the syntax of multi-catch. It's like a regular `catch` clause, except two or more exception types are specified separated by a pipe. The pipe (`|`) is also used as the "or" operator, making it easy to remember that you can use either/or of the exception types. Notice how there is only one variable name in the `catch` clause. Java is saying that the variable named `e` can be of type `Exception1` or `Exception2`.

```
try {  
    // Protected code  
} catch (Exception1 | Exception2 e) {  
    // Exception handler  
}
```



The diagram illustrates the syntax of a multi-catch block with several annotations:

- An arrow points from the text "Protected code" to the code inside the `try` block.
- An arrow points from the text "Catch either of these exceptions." to the pipe symbol (`|`) between `Exception1` and `Exception2` in the `catch` clause.
- An arrow points from the text "Single identifier for all exception types" to the variable `e` in the `catch` clause.
- An arrow points from the text "Required | between exception types" to the pipe symbol (`|`) between `Exception1` and `Exception2`.
- An arrow points from the text "Exception handler" to the code inside the `catch` block.

**FIGURE 10.3** The syntax of a multi-catch block

The exam might try to trick you with invalid syntax. Remember that the exceptions can be listed in any order within the `catch` clause. However, the variable name must appear only once and at the end. Do you see why these are valid or invalid?

```
catch(Exception1 e | Exception2 e | Exception3 e) // DOES NOT COMPILE  
  
catch(Exception1 e1 | Exception2 e2 | Exception3 e3) // DOES NOT COMPILE  
  
catch(Exception1 | Exception2 | Exception3 e)
```

The first line is incorrect because the variable name appears three times. Just because it happens to be the same variable name doesn't make it okay. The second line is incorrect because the variable name again appears three times. Using different variable names doesn't make it any bet-

ter. The third line does compile. It shows the correct syntax for specifying three exceptions.

Java intends multi-catch to be used for exceptions that aren't related, and it prevents you from specifying redundant types in a multi-catch. Do you see what is wrong here?

```
try {
    throw new IOException();
} catch (FileNotFoundException | IOException p) {} // DOES NOT COMPILE
```

Specifying it in the multi-catch is redundant, and the compiler gives a message such as this:

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

Since `FileNotFoundException` is a subclass of `IOException`, this code will not compile. A multi-catch block follows similar rules as chaining `catch` blocks together that you saw in the previous section. For example, both trigger compiler errors when they encounter unreachable code or duplicate exceptions being caught. The one difference between multi-catch blocks and chaining `catch` blocks is that order does not matter for a multi-catch block within a single `catch` expression.

Getting back to the example, the correct code is just to drop the extraneous subclass reference, as shown here:

```
try {
    throw new IOException();
} catch (IOException e) { }
```

To review multi-catch, see how many errors you can find in this `try` statement:

```
11: public void doesNotCompile() { // METHOD DOES NOT COMPILE
12:     try {
13:         mightThrow();
14:     } catch (FileNotFoundException | IllegalStateException e) {
15:     } catch (InputMismatchException e | MissingResourceException e) {
16:     } catch (FileNotFoundException | IllegalArgumentException e) {
17:     } catch (Exception e) {
18:     } catch (IOException e) {
19:     }
20: }
21: private void mightThrow() throws DateTimeParseException, IOException { }
```

This code is just swimming with errors. In fact, some errors hide others, so you might not see them all in the compiler. Once you start fixing some errors, you'll see the others. Here's what's wrong:

- Line 15 has an extra variable name. Remember that there can be only one exception variable per `catch` block.

- Line 16 cannot catch `FileNotFoundException` because that exception was already caught on line 14. You can't list the same exception type more than once in the same `try` statement, just like with “regular” `catch` blocks.
- Lines 17 and 18 are reversed. The more general superclasses must be caught after their subclasses. While this doesn't have anything to do with multi-catch, you'll see “regular” `catch` block problems mixed in with multi-catch.

Don't worry—you won't see this many problems in the same example on the exam!

### Adding a *finally* Block

The `try` statement also lets you run code at the end with a *finally clause* regardless of whether an exception is thrown. [Figure 10.4](#) shows the syntax of a `try` statement with this extra functionality.

```
try {
    // Protected code
} catch (exception_type identifier) {
    // Exception handler
} finally {
    // finally block
}
```

The `finally` keyword

The catch block is optional when `finally` is used.

The finally block always executes, whether or not an exception occurs.

**FIGURE 10.4** The syntax of a `try` statement with `finally`

There are two paths through code with both a `catch` and a `finally`. If an exception is thrown, the `finally` block is run after the `catch` block. If no exception is thrown, the `finally` block is run after the `try` block completes.

Let's go back to our young girl example, this time with `finally`:

```
12: void explore() {
13:     try {
14:         seeAnimals();
15:         fall();
16:     } catch (Exception e) {
17:         getHugFromDaddy();
18:     } finally {
19:         seeMoreAnimals();
20:     }
21:     goHome();
22: }
```

The girl falls on line 15. If she gets up by herself, the code goes on to the `finally` block and runs line 19. Then the `try` statement is over, and the code proceeds on line 21. If the girl doesn't get up by herself, she throws an exception. The `catch` block runs, and she gets a hug on line 17. With that hug she is ready to see more animals on line 19. Then the `try` statement is over, and the code proceeds on line 21. Either way, the ending is the same. The `finally` block is executed, and execution continues after the `try` statement.

The exam will try to trick you with missing clauses or clauses in the wrong order. Do you see why the following do or do not compile?

```
25: try { // DOES NOT COMPILE
26:     fall();
27: } finally {
28:     System.out.println("all better");
29: } catch (Exception e) {
30:     System.out.println("get up");
31: }
32:
33: try { // DOES NOT COMPILE
34:     fall();
35: }
36:
37: try {
38:     fall();
39: } finally {
40:     System.out.println("all better");
41: }
```

The first example (lines 25–31) does not compile because the `catch` and `finally` blocks are in the wrong order. The second example (lines 33–35) does not compile because there must be a `catch` or `finally` block. The third example (lines 37–41) is just fine. The `catch` block is not required if `finally` is present.

One problem with `finally` is that any realistic uses for it are out of the scope of the exam. A `finally` block is typically used to close resources such as files or databases—neither of which is a topic on this exam. This means most of the examples you encounter on the exam with `finally` are going to look contrived. For example, you'll get asked questions such as what this code outputs:

```
public static void main(String[] unused) {
    StringBuilder sb = new StringBuilder();
    try {
        sb.append("t");
    } catch (Exception e) {
        sb.append("c");
    } finally {
        sb.append("f");
    }
    sb.append("a");
    System.out.print(sb.toString());
}
```

The answer is `true`. The `try` block is executed. Since no exception is thrown, Java goes straight to the `finally` block. Then the code after the `try` statement is run. We know that this is a silly example, but you can expect to see examples like this on the exam.

There is one additional rule you should know for `finally` blocks. If a `try` statement with a `finally` block is entered, then the `finally` block will always be executed, regardless of whether the code completes successfully. Take a look at the following `goHome()` method. Assuming an exception may or may not be thrown on line 14, what are the possible values that this method could print? Also, what would the return value be in each case?

```
12: int goHome() {
13:     try {
14:         // Optionally throw an exception here
15:         System.out.print("1");
16:         return -1;
17:     } catch (Exception e) {
18:         System.out.print("2");
19:         return -2;
20:     } finally {
21:         System.out.print("3");
22:         return -3;
23:     }
24: }
```

If an exception is not thrown on line 14, then the line 15 will be executed, printing `1`. Before the method returns, though, the `finally` block is executed, printing `3`. If an exception is thrown, then lines 15–16 will be skipped, and lines 17–19 will be executed, printing `2`, followed by `3` from the `finally` block. While the first value printed may differ, the method always prints `3` last since it's in the `finally` block.

What is the return value of the `goHome()` method? In this case, it's always `-3`. Because the `finally` block is executed shortly before the method completes, it interrupts the `return` statement from inside both the `try` and `catch` blocks.

For the exam, you need to remember that a `finally` block will always be executed. That said, it may not complete successfully. Take a look at the following code snippet. What would happen if `info` was `null` on line 32?

```
31: } finally {
32:     info.printDetails();
33:     System.out.print("Exiting");
34:     return "zoo";
35: }
```

If `info` is `null`, then the `finally` block would be executed, but it would stop on line 32 and throw a `NullPointerException`. Lines 33–34 would not be executed. In this example, you see that while a `finally` block will always be executed, it may not finish.

---

#### `SYSTEM.EXIT()`

There is one exception to “the `finally` block always be executed” rule: Java defines a method that you call as `System.exit()` . It takes an integer parameter that represents the error code that gets returned.

```
try {
    System.exit(0);
} finally {
    System.out.print("Never going to get here"); // Not printed
}
```

`System.exit()` tells Java, “Stop. End the program right now. Do not pass go. Do not collect \$200.” When `System.exit()` is called in the `try` or `catch` block, the `finally` block does not run.

---

## Finally Closing Resources

Oftentimes, your application works with files, databases, and various connection objects. Commonly, these external data sources are referred to as *resources*. In many cases, you *open* a connection to the resource, whether it’s over the network or within a file system. You then *read/write* the data you want. Finally, you *close* the resource to indicate you are done with it.

What happens if you don’t close a resource when you are done with it? In short, a lot of bad things could happen. If you are connecting to a database, you could use up all available connections, meaning no one can talk to the database until you release your connections. Although you commonly hear about memory leaks as causing programs to fail, a *resource leak* is just as bad and occurs when a program fails to release its connections to a resource, resulting in the resource becoming inaccessible.

Writing code that simplifies closing resources is what this section is about. Let’s take a look at a method that opens a file, reads the data, and closes it:

```
4: public void readFile(String file) {
5:     FileInputStream is = null;
6:     try {
7:         is = new FileInputStream("myfile.txt");
8:         // Read file data
9:     } catch (IOException e) {
10:        e.printStackTrace();
11:    } finally {
12:        if(is != null) {
13:            try {
14:                is.close();
15:            } catch (IOException e2) {
16:                e2.printStackTrace();
17:            }
18:        }
19:    }
```



```
19:    }  
20: }
```

Wow, that's a long method! Why do we have two `try` and `catch` blocks? Well, the code on lines 7 and 14 both include checked `IOException` calls, so they both need to be caught in the method or rethrown by the method. Half the lines of code in this method are just closing a resource. And the more resources you have, the longer code like this becomes. For example, you may have multiple resources and they need to be closed in a particular order. You also don't want an exception from closing one resource to prevent the closing of another resource.

To solve this, Java includes the *try-with-resources* statement to automatically close all resources opened in a `try` clause. This feature is also known as *automatic resource management*, because Java automatically takes care of the closing.



For the 1Z0-815 exam, you are not required to know any File IO, network, or database classes, although you are required to know *try-with-resources*. If you see a question on the exam or in this chapter that uses these types of resources, assume that part of the code compiles without issue. In other words, these questions are actually a gift, since you know the problem must be about basic Java syntax or exception handling. That said, for the 1Z0-816 exam, you will need to know numerous resources classes.

---

Let's take a look at our same example using a *try-with-resources* statement:

```
4: public void readFile(String file) {  
5:     try (FileInputStream is = new FileInputStream("myfile.txt")) {  
6:         // Read file data  
7:     } catch (IOException e) {  
8:         e.printStackTrace();  
9:     }  
10: }
```

Functionally, they are both quite similar, but our new version has half as many lines. More importantly, though, by using a *try-with-resources* statement, we guarantee that as soon as a connection passes out of scope, Java will attempt to close it within the same method.

In the following sections, we will look at the *try-with-resources* syntax and how to indicate a resource can be automatically closed.

---

#### IMPLICIT *FINALLY* BLOCKS

Behind the scenes, the compiler replaces a try-with-resources block with a try and finally block. We refer to this “hidden” finally block as an implicit finally block since it is created and used by the compiler automatically. You can still create a programmer-defined finally block when using a try-with-resources statement; just be aware that the implicit one will be called first.

---

### Basics of Try-with-Resources

Figure 10.5 shows what a try-with-resources statement looks like. Notice that one or more resources can be opened in the try clause. When there are multiple resources opened, they are closed in the *reverse* order from which they were created. Also, notice that parentheses are used to list those resources, and semicolons are used to separate the declarations. This works just like declaring multiple indexes in a for loop.

```
Any resources that should
automatically be closed
try (FileInputStream in = new FileInputStream("data.txt");
    FileOutputStream out = new FileOutputStream("output.txt");) {
    // Protected code
}
```

Resources are closed at this point.

Required semicolon between  
resource declarations

Last semicolon is optional  
(usually omitted)

**FIGURE 10.5** The syntax of a basic try-with-resources

What happened to the catch block in Figure 10.5? Well, it turns out a catch block is optional with a try-with-resources statement. For example, we can rewrite the previous readFile() example so that the method rethrows the exception to make it even shorter:

```
4: public void readFile(String file) throws IOException {
5:     try (FileInputStream is = new FileInputStream("myfile.txt")) {
6:         // Read file data
7:     }
8: }
```

Earlier in the chapter, you learned that a try statement must have one or more catch blocks or a finally block. This is still true. The finally clause exists implicitly. You just don’t have to type it.



Remember that only a try-with-resources statement is permitted to omit both the catch and finally blocks. A traditional try statement must have either or both. You can easily distinguish between the two by the presence of parentheses, ( ), after the try keyword.

---

Figure 10.6 shows that a try-with-resources statement is still allowed to have catch and/or finally blocks. In fact, if the code within the try block throws a checked exception not declared by the method in which it is defined or handled by another try / catch block, then it will need to be handled by the catch block. Also, the catch and finally blocks are run in addition to the implicit one that closes the resources. For the exam, you need to know that the implicit finally block runs *before* any programmer-coded ones.

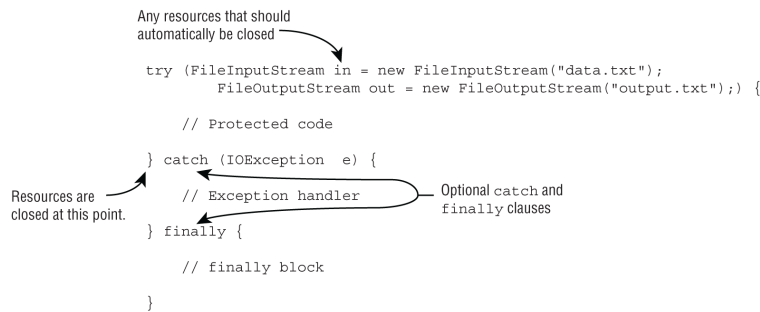


FIGURE 10.6 The syntax of try-with-resources including catch / finally

To make sure that you’ve wrapped your head around the differences, you should be able to fill in Table 10.2 and Table 10.3 with whichever combinations of catch and finally blocks are legal configurations.

TABLE 10.2 Legal vs. illegal configurations with a traditional try statement

	0 finally blocks	1 finally block	2 or more finally blocks
0 catch blocks	Not legal	Legal	Not legal
1 or more catch blocks	Legal	Legal	Not legal

TABLE 10.3 Legal vs. illegal configurations with a try-with-resources statement

	0 finally blocks	1 finally block	2 or more finally blocks
0 catch blocks	Legal	Legal	Not legal
1 or more catch blocks	Legal	Legal	Not legal

You can see that for both of these try statements, two or more programmer-defined finally blocks are not allowed. Remember that the implicit finally block defined by the compiler is not counted here.

---

#### AUTOCLOSEABLE

You can't just put any random class in a try-with-resources statement. Java requires classes used in a try-with-resources implement the `AutoCloseable` interface, which includes a `void close()` method. You'll learn more about resources that implement this method when you study for the 1Z0-816 exam.

---

### Declaring Resources

While try-with-resources does support declaring multiple variables, each variable must be declared in a separate statement. For example, the following do not compile:

```
try (MyFileClass is = new MyFileClass(1), // DOES NOT COMPILE
     os = new MyFileClass(2)) {
}

try (MyFileClass ab = new MyFileClass(1), // DOES NOT COMPILE
     MyFileClass cd = new MyFileClass(2)) {
}
```

A try-with-resources statement does not support multiple variable declarations. The first example does not compile because it is missing the data type and it uses a comma ( , ) instead of a semicolon ( ; ). The second example does not compile because it also uses a comma ( , ) instead of a semicolon ( ; ). Each resource must include the data type and be separated by a semicolon ( ; ).

You can declare a resource using `var` as the data type in a try-with-resources statement, since resources are local variables.

```
try (var f = new BufferedInputStream(new FileInputStream("it.txt"))) {
    // Process file
}
```

Declaring resources is a common situation where using `var` is quite helpful, as it shortens the already long line of code.

### Scope of Try-with-Resources

The resources created in the `try` clause are in scope only within the `try` block. This is another way to remember that the implicit `finally` runs before any `catch` / `finally` blocks that you code yourself. The implicit close has run already, and the resource is no longer available. Do you see why lines 6 and 8 don't compile in this example?

```
3: try (Scanner s = new Scanner(System.in)) {
4:     s.nextLine();
5: } catch (Exception e) {
6:     s.nextInt(); // DOES NOT COMPILE
7: } finally {
```

```
8:     s.nextInt(); // DOES NOT COMPILE
9: }
```

The problem is that `Scanner` has gone out of scope at the end of the `try` clause. Lines 6 and 8 do not have access to it. This is actually a nice feature. You can't accidentally use an object that has been closed. In a traditional `try` statement, the variable has to be declared before the `try` statement so that both the `try` and `finally` blocks can access it, which has the unpleasant side effect of making the variable in scope for the rest of the method, just inviting you to call it by accident.

### Following Order of Operation

You've learned two new rules for the order in which code runs in a try-with-resources statement:

- Resources are closed after the `try` clause ends and before any `catch`/`finally` clauses.
- Resources are closed in the reverse order from which they were created.

Let's review these principles with a more complex example. First, we define a custom class that you can use with a try-with-resources statement, as it implements `AutoCloseable`.

```
public class MyFileClass implements AutoCloseable {
    private final int num;
    public MyFileClass(int num) { this.num = num; }
    public void close() {
        System.out.println("Closing: " + num);
    }
}
```

This is a pretty simple class that prints the number, set by the constructor, when a resource is closed. Based on these rules, can you figure out what this method prints?

```
public static void main(String... xyz) {
    try (MyFileClass a1 = new MyFileClass(1);
        MyFileClass a2 = new MyFileClass(2)) {
        throw new RuntimeException();
    } catch (Exception e) {
        System.out.println("ex");
    } finally {
        System.out.println("finally");
    }
}
```

Since the resources are closed in the reverse order from which they were opened, we have `Closing: 2` and then `Closing: 1`. After that, the `catch` block and `finally` block are run—just as they are in a regular `try` statement. The output is as follows:

```
Closing: 2
Closing: 1
ex
finally
```

For the exam, make sure you understand why the method prints the statements in this order. Remember, the resources are closed in the reverse order from which they are declared, and the implicit `finally` is executed before the programmer-defined `finally`.



### Real World Scenario

#### TRY-WITH-RESOURCES GUARANTEES

Does a try-with-resources statement guarantee a resource will be closed? Although this is beyond the scope of the exam, the short answer is “no.” The try-with-resources statement guarantees only the `close()` method will be called. If the `close()` method encounters an exception of its own or the method is implemented poorly, a resource leak can still occur. For the exam, you just need to know try-with-resources is guaranteed to call the `close()` method on the resource.

---

## Throwing Additional Exceptions

A `catch` or `finally` block can have any valid Java code in it—including another `try` statement. What happens when an exception is thrown inside of a `catch` or `finally` block?

To answer this, let’s take a look at a concrete example:

```
16: public static void main(String[] a) {
17:     FileReader reader = null;
18:     try {
19:         reader = read();
20:     } catch (IOException e) {
21:         try {
22:             if (reader != null) reader.close();
23:         } catch (IOException inner) {
24:             }
25:     }
26: }
27: private static FileReader read() throws IOException {
28:     // CODE GOES HERE
29: }
```

The easiest case is if line 28 doesn’t throw an exception. Then the entire `catch` block on lines 20–25 is skipped. Next, consider if line 28 throws a `NullPointerException`. That isn’t an `IOException`, so the `catch` block on lines 20–25 will still be skipped, resulting in the `main()` method terminating early.

If line 28 does throw an `IOException`, the `catch` block on lines 20–25 gets run. Line 22 tries to close the `reader`. If that goes well, the code com-

pletes, and the `main()` method ends normally. If the `close()` method does throw an exception, Java looks for more `catch` blocks. This exception is caught on line 23. Regardless, the exception on line 28 is handled. A different exception might be thrown, but the one from line 28 is done.

Most of the examples you see with exception handling on the exam are abstract. They use letters or numbers to make sure you understand the flow. This one shows that only the last exception to be thrown matters:

```
26: try {
27:     throw new RuntimeException();
28: } catch (RuntimeException e) {
29:     throw new RuntimeException();
30: } finally {
31:     throw new Exception();
32: }
```

Line 27 throws an exception, which is caught on line 28. The `catch` block then throws an exception on line 29. If there were no `finally` block, the exception from line 29 would be thrown. However, the `finally` block runs after the `catch` block. Since the `finally` block throws an exception of its own on line 31, this one gets thrown. The exception from the `catch` block gets forgotten about. This is why you often see another `try/catch` inside a `finally` block—to make sure it doesn't mask the exception from the `catch` block.

Next we are going to show you one of the hardest examples you can be asked related to exceptions. What do you think this method returns? Go slowly. It's tricky.

```
30: public String exceptions() {
31:     StringBuilder result = new StringBuilder();
32:     String v = null;
33:     try {
34:         try {
35:             result.append("before_");
36:             v.length();
37:             result.append("after_");
38:         } catch (NullPointerException e) {
39:             result.append("catch_");
40:             throw new RuntimeException();
41:         } finally {
42:             result.append("finally_");
43:             throw new Exception();
44:         }
45:     } catch (Exception e) {
46:         result.append("done");
47:     }
48:     return result.toString();
49: }
```

The correct answer is `before_catch_finally_done`. First on line 35, "before\_" is added. Line 36 throws a `NullPointerException`. Line 37 is skipped as Java goes straight to the `catch` block. Line 38 does catch the exception, and "catch\_" is added on line 39. Then line 40 throws a

`RuntimeException`. The `finally` block runs after the `catch` regardless of whether an exception is thrown; it adds `"finally_"` to `result`. At this point, we have completed the inner `try` statement that ran on lines 34–44. The outer `catch` block then sees an exception was thrown and catches it on line 45; it adds `"done"` to `result`.

Did you get that right? If so, you are well on your way to acing this part of the exam. If not, we recommend reading this section again before moving on.

## Calling Methods That Throw Exceptions

When you're calling a method that throws an exception, the rules are the same as within a method. Do you see why the following doesn't compile?

```
class NoMoreCarrotsException extends Exception {}
public class Bunny {
    public static void main(String[] args) {
        eatCarrot(); // DOES NOT COMPILE
    }
    private static void eatCarrot() throws NoMoreCarrotsException {
    }
}
```

The problem is that `NoMoreCarrotsException` is a checked exception. Checked exceptions must be handled or declared. The code would compile if you changed the `main()` method to either of these:

```
public static void main(String[] args)
    throws NoMoreCarrotsException { // declare exception
    eatCarrot();
}

public static void main(String[] args) {
    try {
        eatCarrot();
    } catch (NoMoreCarrotsException e) { // handle exception
        System.out.print("sad rabbit");
    }
}
```

You might have noticed that `eatCarrot()` didn't actually throw an exception; it just declared that it could. This is enough for the compiler to require the caller to handle or declare the exception.

The compiler is still on the lookout for unreachable code. Declaring an unused exception isn't considered unreachable code. It gives the method the option to change the implementation to throw that exception in the future. Do you see the issue here?

```
public void bad() {
    try {
        eatCarrot();
    }
```



```

        } catch (NoMoreCarrotsException e ) { // DOES NOT COMPILE
            System.out.print("sad rabbit");
        }
    }

    public void good() throws NoMoreCarrotsException {
        eatCarrot();
    }

    private void eatCarrot() { }

```

Java knows that `eatCarrot()` can't throw a checked exception—which means there's no way for the `catch` block in `bad()` to be reached. In comparison, `good()` is free to declare other exceptions.



When you see a checked exception declared inside a `catch` block on the exam, check and make sure the code in the associated `try` block is capable of throwing the exception or a subclass of the exception. If not, the code is unreachable and does not compile. Remember that this rule does not extend to unchecked exceptions or exceptions declared in a method signature.

---

## Declaring and Overriding Methods with Exceptions

Now that you have a deeper understanding of exceptions, let's look at overriding methods with exceptions in the method declaration. When a class overrides a method from a superclass or implements a method from an interface, it's not allowed to add new checked exceptions to the method signature. For example, this code isn't allowed:

```

class CanNotHopException extends Exception { }
class Hopper {
    public void hop() { }
}
class Bunny extends Hopper {
    public void hop() throws CanNotHopException { } // DOES NOT COMPILE
}

```

Java knows `hop()` isn't allowed to throw any checked exceptions because the `hop()` method in the superclass `Hopper` doesn't declare any. Imagine what would happen if the subclasses versions of the method could add checked exceptions—you could write code that calls `Hopper`'s `hop()` method and not handle any exceptions. Then if `Bunny` were used in its place, the code wouldn't know to handle or declare `CanNotHopException`.

An overridden method in a subclass is allowed to declare fewer exceptions than the superclass or interface. This is legal because callers are already handling them.

```

class Hopper {
    public void hop() throws CanNotHopException { }
}
class Bunny extends Hopper {
    public void hop() { }
}

```

An overridden method not declaring one of the exceptions thrown by the parent method is similar to the method declaring it throws an exception that it never actually throws. This is perfectly legal.

Similarly, a class is allowed to declare a subclass of an exception type. The idea is the same. The superclass or interface has already taken care of a broader type. Here's an example:

```

class Hopper {
    public void hop() throws Exception { }
}
class Bunny extends Hopper {
    public void hop() throws CanNotHopException { }
}

```

`Bunny` could declare that it throws `Exception` directly, or it could declare that it throws a more specific type of `Exception`. It could even declare that it throws nothing at all.

This rule applies only to checked exceptions. The following code is legal because it has an unchecked exception in the subclass's version:

```

class Hopper {
    public void hop() { }
}
class Bunny extends Hopper {
    public void hop() throws IllegalStateException { }
}

```

The reason that it's okay to declare new unchecked exceptions in a subclass method is that the declaration is redundant. Methods are free to throw any unchecked exceptions they want without mentioning them in the method declaration.

## Printing an Exception

There are three ways to print an exception. You can let Java print it out, print just the message, or print where the stack trace comes from. This example shows all three approaches:

```

5: public static void main(String[] args) {
6:     try {
7:         hop();
8:     } catch (Exception e) {
9:         System.out.println(e);
10:        System.out.println(e.getMessage());
11:        e.printStackTrace();

```

```
12:    }  
13: }  
14: private static void hop() {  
15:     throw new RuntimeException("cannot hop");  
16: }
```

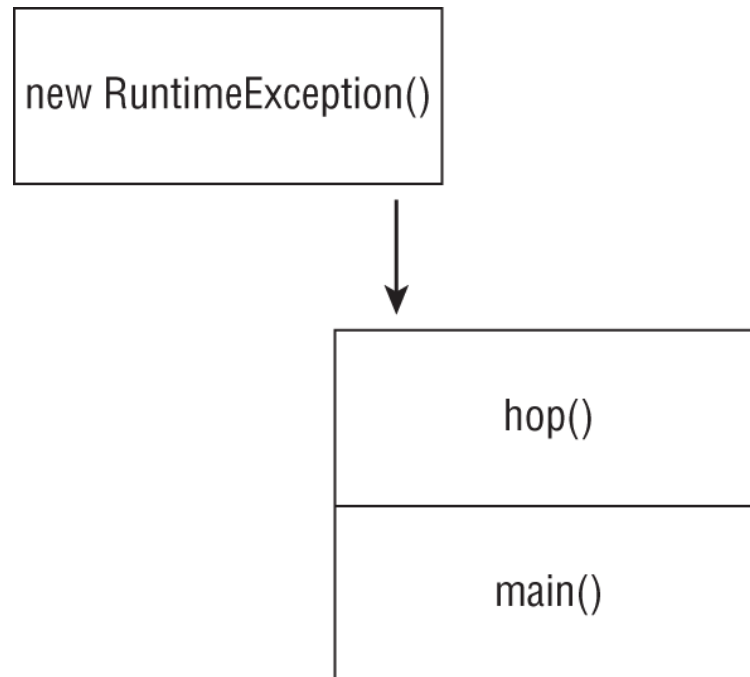
This code results in the following output:

```
java.lang.RuntimeException: cannot hop  
cannot hop  
java.lang.RuntimeException: cannot hop  
    at Handling.hop(Handling.java:15)  
    at Handling.main(Handling.java:7)
```

The first line shows what Java prints out by default: the exception type and message. The second line shows just the message. The rest shows a stack trace.

The stack trace is usually the most helpful one because it is a picture in time the moment the exception is thrown. It shows the hierarchy of method calls that were made to reach the line that threw the exception. On the exam, you will mostly see the first approach. This is because the exam often shows code snippets.

The stack trace shows all the methods on the stack. [Figure 10.7](#) shows what the stack looks like for this code. Every time you call a method, Java adds it to the stack until it completes. When an exception is thrown, it goes through the stack until it finds a method that can handle it or it runs out of stack.



**FIGURE 10.7** A method stack



Because checked exceptions require you to handle or declare them, there is a temptation to catch them so they “go away.” But doing so can cause problems. In the following code, there’s a problem reading the file:

```
public static void main(String... p) {
    String textInFile = null;
    try {
        textInFile = readInFile();
    } catch (IOException e) {
        // ignore exception
    }
    // imagine many lines of code here
    System.out.println(textInFile.replace(" ", ""));
}
private static String readInFile() throws IOException {
    throw new IOException();
}
```

The code results in a `NullPointerException`. Java doesn’t tell you anything about the original `IOException` because it was handled. Granted, it was handled poorly, but it was handled.

When writing this book, we tend to swallow exceptions because many of our examples are artificial in nature. However, when you’re writing your own code, you should print out a stack trace or at least a message when catching an exception. Also, consider whether continuing is the best course of action. In our example, the program can’t do anything after it fails to read in the file. It might as well have just thrown the `IOException`.

---

## Summary

An exception indicates something unexpected happened. A method can handle an exception by catching it or declaring it for the caller to deal with. Many exceptions are thrown by Java libraries. You can throw your own exceptions with code such as `throw new Exception()`.

All exceptions inherit `Throwable`. Subclasses of `Error` are exceptions that a programmer should not attempt to handle. Classes that inherit `RuntimeException` and `Error` are runtime (unchecked) exceptions. Classes that inherit `Exception`, but not `RuntimeException`, are checked exceptions. Java requires checked exceptions to be handled with a `catch` block or declared with the `throws` keyword.

A `try` statement must include at least one `catch` block or a `finally` block. A multi-catch block is one that catches multiple unrelated exceptions in a single `catch` block. If a `try` statement has multiple `catch` blocks chained together, at most one `catch` block can run. Java looks for

an exception that can be caught by each `catch` block in the order they appear, and the first match is run. Then execution continues after the `try` statement. If both `catch` and `finally` throw an exception, the one from `finally` gets thrown.

A try-with-resources block is used to ensure a resource like a database or a file is closed properly after it is created. A try-with-resources statement does not require a `catch` or `finally` block but may optionally include them. The implicit `finally` block is executed before any programmer-defined `catch` or `finally` blocks.

`RuntimeException` classes you should know for the exam include the following:

- `ArithmeticException`
- `ArrayIndexOutOfBoundsException`
- `ClassCastException`
- `IllegalArgumentException`
- `NullPointerException`
- `NumberFormatException`

`IllegalArgumentException` is typically thrown by the programmer, whereas the others are typically thrown by the standard Java library.

Checked `Exception` classes you should know for the exam include the following:

- `IOException`
- `FileNotFoundException`

`Error` classes you should know for the exam include the following:

- `ExceptionInInitializerError`
- `StackOverflowError`
- `NoClassDefFoundError`

For the exam, remember that `NumberFormatException` is a subclass of `IllegalArgumentException`, and `FileNotFoundException` is a subclass of `IOException`.

When a method overrides a method in a superclass or interface, it is not allowed to add checked exceptions. It is allowed to declare fewer exceptions or declare a subclass of a declared exception. Methods declare exceptions with the keyword `throws`.

## Exam Essentials

**Understand the various types of exceptions.** All exceptions are subclasses of `java.lang.Throwable`. Subclasses of `java.lang.Error` should never be caught. Only subclasses of `java.lang.Exception` should be handled in application code.

**Differentiate between checked and unchecked exceptions.** Unchecked exceptions do not need to be caught or handled and are subclasses of `java.lang.RuntimeException` and `java.lang.Error`. All other subclasses of `java.lang.Exception` are checked exceptions and must be handled or declared.

**Understand the flow of a try statement.** A `try` statement must have a `catch` or a `finally` block. Multiple `catch` blocks can be chained together, provided no superclass exception type appears in an earlier `catch` block than its subclass. A multi-catch expression may be used to handle multiple exceptions in the same `catch` block, provided one exception is not a subclass of another. The `finally` block runs last regardless of whether an exception is thrown.

**Be able to follow the order of a try-with-resources statement.** A try-with-resources statement is a special type of `try` block in which one or more resources are declared and automatically closed in the reverse order of which they are declared. It can be used with or without a `catch` or `finally` block, with the implicit `finally` block always executed first.

**Identify whether an exception is thrown by the programmer or the JVM.** `IllegalArgumentException` and `NumberFormatException` are commonly thrown by the programmer. Most of the other unchecked exceptions are typically thrown by the JVM or built-in Java libraries.

**Write methods that declare exceptions.** The `throws` keyword is used in a method declaration to indicate an exception might be thrown. When overriding a method, the method is allowed to throw fewer or narrower checked exceptions than the original version.

**Recognize when to use throw versus throws.** The `throw` keyword is used when you actually want to throw an exception—for example, `throw new RuntimeException()`. The `throws` keyword is used in a method declaration.

## Review Questions

The answers to the chapter review questions can be found in the Appendix.

1. Which of the following statements are true? (Choose all that apply.)
  1. Exceptions of type `RuntimeException` are unchecked.
  2. Exceptions of type `RuntimeException` are checked.
  3. You can declare unchecked exceptions.
  4. You can declare checked exceptions.
  5. You can handle only `Exception` subclasses.
  6. All exceptions are subclasses of `Throwable`.
2. Which of the following pairs fill in the blanks to make this code compile? (Choose all that apply.)

```
6: public void ohNo(ArithmeticException ae) _____ Exception {  
7: if(ae==null) _____ Exception();
```

```

8: else _____ ae;
9: }

```

1. On line 6, fill in `throw`
  2. On line 6, fill in `throws`
  3. On line 7, fill in `throw`
  4. On line 7, fill in `throw new`
  5. On line 8, fill in `throw`
  6. On line 8, fill in `throw new`
  7. None of the above
3. What is printed by the following? (Choose all that apply.)

```

1: public class Mouse {
2:     public String name;
3:     public void findCheese() {
4:         System.out.print("1");
5:         try {
6:             System.out.print("2");
7:             name.toString();
8:             System.out.print("3");
9:         } catch (NullPointerException e | ClassCastException e) {
10:            System.out.print("4");
11:            throw e;
12:        }
13:        System.out.print("5");
14:    }
15:    public static void main(String... tom) {
16:        Mouse jerry = new Mouse();
17:        jerry.findCheese();
18:    } }

```

1. 1
  2. 2
  3. 3
  4. 4
  5. 5
  6. The stack trace for a `NullPointerException`
  7. None of the above
4. Which of the following statements about `finally` blocks are true? (Choose all that apply.)
1. A `finally` block is never required with a regular `try` statement.
  2. A `finally` block is required when there are no `catch` blocks in a regular `try` statement.
  3. A `finally` block is required when the program code doesn't terminate on its own.
  4. A `finally` block is never required with a `try-with-resources` statement.
  5. A `finally` block is required when there are no `catch` blocks in a `try-with-resources` statement.
  6. A `finally` block is required in order to make sure all resources are closed in a `try-with-resources` statement.
  7. A `finally` block is executed before the resources declared in a `try-with-resources` statement are closed.
5. Which exception will the following method throw?

```

3: public static void main(String[] other) {
4:     Object obj = Integer.valueOf(3);
5:     String str = (String) obj;
6:     obj = null;
7:     System.out.println(obj.equals(null));
8: }

```

1. `ArrayIndexOutOfBoundsException`
  2. `IllegalArgumentException`
  3. `ClassCastException`
  4. `NumberFormatException`
  5. `NullPointerException`
  6. None of the above
6. What does the following method print?

```

11: public void tryAgain(String s) {
12:     try(FileReader r = null, p = new FileReader("")) {
13:         System.out.print("X");
14:         throw new IllegalArgumentException();
15:     } catch (Exception s) {
16:         System.out.print("A");
17:         throw new FileNotFoundException();
18:     } finally {
19:         System.out.print("O");
20:     }
21: }

```

1. XAO
  2. XOA
  3. One line of this method contains a compiler error.
  4. Two lines of this method contain compiler errors.
  5. Three lines of this method contain compiler errors.
  6. The code compiles, but a `NullPointerException` is thrown at runtime.
  7. None of the above
7. What will happen if you add the following statement to a working `main()` method?

```
System.out.print(4 / 0);
```

1. It will not compile.
  2. It will not run.
  3. It will run and throw an `ArithmeticException`.
  4. It will run and throw an `IllegalArgumentException`.
  5. None of the above
8. What is printed by the following program?

```

1: public class DoSomething {
2:     public void go() {
3:         System.out.print("A");
4:         try {
5:             stop();
6:         } catch (ArithmeticException e) {

```



```

7:         System.out.print("B");
8:     } finally {
9:         System.out.print("C");
10:    }
11:    System.out.print("D");
12: }
13: public void stop() {
14:     System.out.print("E");
15:     Object x = null;
16:     x.toString();
17:     System.out.print("F");
18: }
19: public static void main(String n[]) {
20:     new DoSomething().go();
21: }
22: }

```

1. AE
  2. AEBCD
  3. AEC
  4. AECD
  5. AE followed by a stack trace
  6. AEBCD followed by a stack trace
  7. AEC followed by a stack trace
  8. A stack trace with no other output
9. What is the output of the following snippet, assuming a and b are both 0?

```

3: try {
4:     System.out.print(a / b);
5: } catch (RuntimeException e) {
6:     System.out.print(-1);
7: } catch (ArithmeticException e) {
8:     System.out.print(0);
9: } finally {
10:    System.out.print("done");
11: }

```

1. -1
  2. 0
  3. done-1
  4. done0
  5. The code does not compile.
  6. An uncaught exception is thrown.
  7. None of the above
10. What is the output of the following program?

```

1: public class Laptop {
2:     public void start() {
3:         try {
4:             System.out.print("Starting up_");
5:             throw new Exception();
6:         } catch (Exception e) {
7:             System.out.print("Problem_");
8:             System.exit(0);
9:         } finally {

```

```

10:         System.out.print("Shutting down");
11:     }
12: }
13: public static void main(String[] w) {
14:     new Laptop().start();
15: } }

```

1. Starting up\_
  2. Starting up\_Problem\_
  3. Starting up\_Problem\_Shutting down
  4. Starting up\_Shutting down
  5. The code does not compile.
  6. An uncaught exception is thrown.
11. What is the output of the following program?

```

1: public class Dog {
2:     public String name;
3:     public void runAway() {
4:         System.out.print("1");
5:         try {
6:             System.out.print("2");
7:             int x = Integer.parseInt(name);
8:             System.out.print("3");
9:         } catch (NumberFormatException e) {
10:            System.out.print("4");
11:        }
12:    }
13: public static void main(String... args) {
14:     Dog webby = new Dog();
15:     webby.name = "Webby";
16:     webby.runAway();
17:     System.out.print("5");
18: } }

```

1. 1234
  2. 1235
  3. 124
  4. 1245
  5. The code does not compile.
  6. An uncaught exception is thrown.
  7. None of the above
12. What is the output of the following program?

```

1: public class Cat {
2:     public String name;
3:     public void knockStuffOver() {
4:         System.out.print("1");
5:         try {
6:             System.out.print("2");
7:             int x = Integer.parseInt(name);
8:             System.out.print("3");
9:         } catch (NullPointerException e) {
10:            System.out.print("4");
11:        }
12:     System.out.print("5");
13: }

```

```

14:     public static void main(String args[]) {
15:         Cat loki = new Cat();
16:         loki.name = "Loki";
17:         loki.knockStuffOver();
18:         System.out.print("6");
19:     } }

```

1. The output is 12 , followed by a stack trace for a `NumberFormatException` .
  2. The output is 124 , followed by a stack trace for a `NumberFormatException` .
  3. The output is 12456 .
  4. The output is 1256 , followed by a stack trace for a `NumberFormatException` .
  5. The code does not compile.
  6. An uncaught exception is thrown.
  7. None of the above
13. Which of the following statements are true? (Choose all that apply.)
1. You can declare a method with `Exception` as the return type.
  2. You can declare a method with `RuntimeException` as the return type.
  3. You can declare any subclass of `Error` in the `throws` part of a method declaration.
  4. You can declare any subclass of `Exception` in the `throws` part of a method declaration.
  5. You can declare any subclass of `Object` in the `throws` part of a method declaration.
  6. You can declare any subclass of `RuntimeException` in the `throws` part of a method declaration.
14. Which of the following can be inserted on line 8 to make this code compile? (Choose all that apply.)

```

7: public void whatHappensNext() throws IOException {
8:     // INSERT CODE HERE
9: }

```

1. `System.out.println("it's ok");`
  2. `throw new Exception();`
  3. `throw new IllegalArgumentException();`
  4. `throw new java.io.IOException();`
  5. `throw new RuntimeException();`
  6. None of the above
15. What is printed by the following program? (Choose all that apply.)

```

1: public class Help {
2:     public void callSuperhero() {
3:         try (String raspberry = new String("Olivia")) {
4:             System.out.print("Q");
5:         } catch (Error e) {
6:             System.out.print("X");
7:         } finally {
8:             System.out.print("M");
9:         }

```

```

10:    }
11:    public static void main(String[] args) {
12:        new Help().callSuperhero();
13:        System.out.print("S");
14:    } }

```

1. SQM
  2. QXMS
  3. QSM
  4. QMS
  5. A stack trace
  6. The code does not compile because `NumberFormatException` is not declared or caught.
  7. None of the above
16. Which of the following do not need to be handled or declared?  
(Choose all that apply.)
1. `ArrayIndexOutOfBoundsException`
  2. `IllegalArgumentException`
  3. `IOException`
  4. `Error`
  5. `NumberFormatException`
  6. Any exception that extends `RuntimeException`
  7. Any exception that extends `Exception`
17. Which lines can fill in the blank to make the following code compile?  
(Choose all that apply.)

```

void rollOut() throws ClassCastException {}

public void transform(String c) {
    try {
        rollOut();
    } catch (IllegalArgumentException | _____) {
    }
}

```

1. `IOException` a
  2. `Error` b
  3. `NullPointerException` c
  4. `RuntimeException` d
  5. `NumberFormatException` e
  6. `ClassCastException` f
  7. None of the above. The code contains a compiler error regardless of what is inserted into the blank.
18. Which scenario is the best use of an exception?
1. An element is not found when searching a list.
  2. An unexpected parameter is passed into a method.
  3. The computer caught fire.
  4. You want to loop through a list.
  5. You don't know how to code a method.
19. Which of the following can be inserted into `Lion` to make this code compile? (Choose all that apply.)

```

class HasSoreThroatException extends Exception {}
class TiredException extends RuntimeException {}
interface Roar {
    void roar() throws HasSoreThroatException;
}
class Lion implements Roar {
    // INSERT CODE HERE
}

```

1. public void roar() {}
2. public int roar() throws RuntimeException {}
3. public void roar() throws Exception {}
4. public void roar() throws HasSoreThroatException {}
5. public void roar() throws IllegalArgumentException {}
6. public void roar() throws TiredException {}

20. Which of the following are true? (Choose all that apply.)

1. Checked exceptions are allowed, but not required, to be handled or declared.
2. Checked exceptions are required to be handled or declared.
3. Errors are allowed, but not required, to be handled or declared.
4. Errors are required to be handled or declared.
5. Unchecked exceptions are allowed, but not required, to be handled or declared.
6. Unchecked exceptions are required to be handled or declared.

21. Which of the following pairs fill in the blanks to make this code compile? (Choose all that apply.)

```

6: public void ohNo(IOException ie) _____ Exception {
7:     _____ FileNotFoundException();
8:     _____ ie;
9: }

```

1. On line 6, fill in throw
2. On line 6, fill in throws
3. On line 7, fill in throw
4. On line 7, fill in throw new
5. On line 8, fill in throw
6. On line 8, fill in throw new
7. None of the above

22. Which of the following can be inserted in the blank to make the code compile? (Choose all that apply.)

```

public void dontFail() {
    try {
        System.out.println("work real hard");
    } catch (_____ e) {
    } catch (RuntimeException e) {}
}

```

1. var
2. Exception
3. IOException
4. IllegalArgumentException

- 5. RuntimeException
- 6. StackOverflowError
- 7. None of the above

23. What does the output of the following method contain? (Choose all that apply.)

```

12: public static void main(String[] args) {
13:     System.out.print("a");
14:     try {
15:         System.out.print("b");
16:         throw new IllegalArgumentException();
17:     } catch (IllegalArgumentException e) {
18:         System.out.print("c");
19:         throw new RuntimeException("1");
20:     } catch (RuntimeException e) {
21:         System.out.print("d");
22:         throw new RuntimeException("2");
23:     } finally {
24:         System.out.print("e");
25:         throw new RuntimeException("3");
26:     }
27: }

```

- 1. abce
- 2. abde
- 3. An exception with the message set to "1"
- 4. An exception with the message set to "2"
- 5. An exception with the message set to "3"
- 6. Nothing; the code does not compile.

24. What does the following class output?

```

1: public class MoreHelp {
2:     class Sidekick implements AutoCloseable {
3:         protected String n;
4:         public Sidekick(String n) { this.n = n; }
5:         public void close() { System.out.print("L"); }
6:     }
7:     public void requiresAssistance() {
8:         try (Sidekick is = new Sidekick("Adeline")) {
9:             System.out.print("O");
10:        } finally {
11:            System.out.print("K");
12:        }
13:    }
14:    public static void main(String... league) {
15:        new MoreHelp().requiresAssistance();
16:        System.out.print("I");
17:    } }

```

- 1. LOKI
- 2. OKLI
- 3. OLKI
- 4. OKIL
- 5. The output cannot be determined until runtime.
- 6. Nothing; the code does not compile.
- 7. None of the above

25. What does the following code snippet return, assuming a and b are both 1 ?

```
13: try {  
14:     return a / b;  
15: } catch (ClassCastException e) {  
16:     return 10;  
17: } catch (RuntimeException e) {  
18:     return 20;  
19: } finally {  
20:     return 30;  
21: }
```

1. 1
2. 10
3. 20
4. 30
5. The code does not compile.
6. An uncaught exception is thrown.
7. None of the above

[Support](#)   [Sign Out](#)