14

Handling Exceptions

Objectives

After completing this lesson, you should be able to:

- Describe how Java handles unexpected events in a program
- List the three types of `Throwable` classes
- Determine what exceptions are thrown for any foundation class
- Describe what happens in the call stack when an exception is thrown and not caught
- Write code to handle an exception thrown by the method of a foundation class

Topics

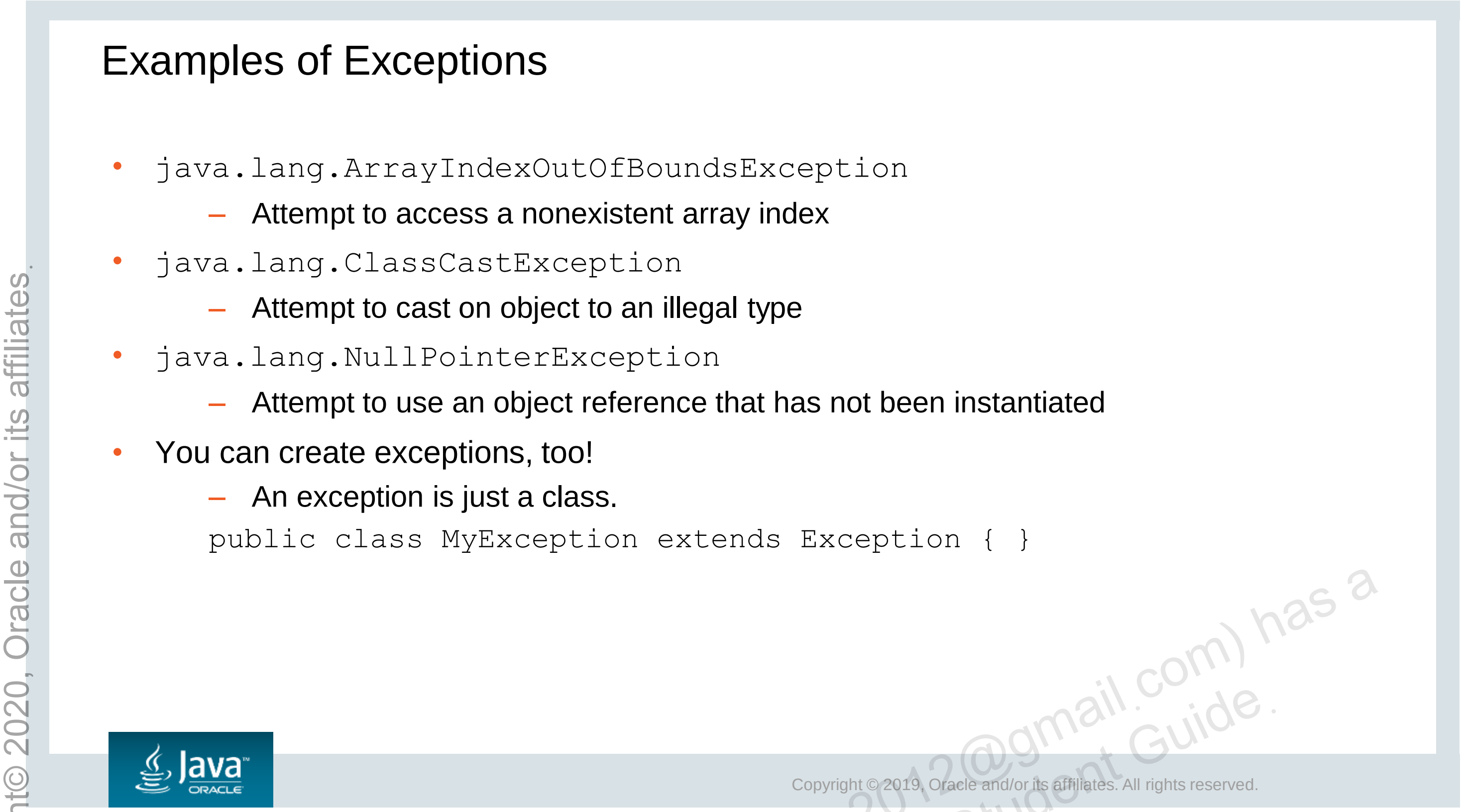
- Handling exceptions: an overview
- Propagation of exceptions
- Catching and throwing exceptions
- Multiple exceptions and errors

What Are Exceptions?

Java handles unexpected situations using exceptions.

- Something unexpected happens in the program.
- Java doesn't know what to do, so it:
  - Creates an exception object containing useful information and
  - Throws the exception to the code that invoked the problematic method
- There are several different types of exceptions.

What if something goes wrong in an application? When an unforeseen event occurs in an application, you say "an exception was thrown."There are many types of exceptions and, in this lesson, you will learn what they are and how to handle them.



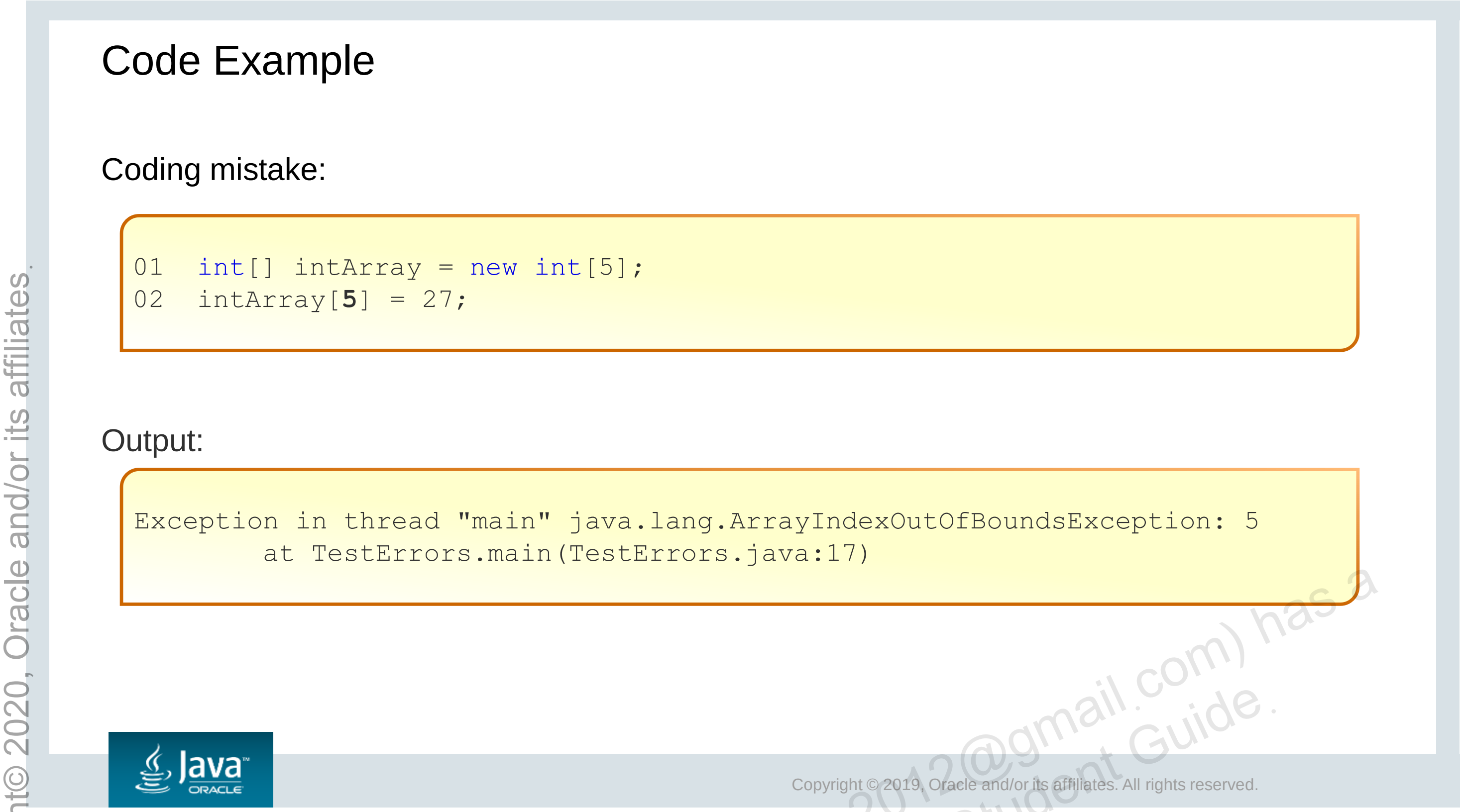
Examples of Exceptions

- `java.lang.ArrayIndexOutOfBoundsException`
  - Attempt to access a nonexistent array index
- `java.lang.ClassCastException`
  - Attempt to cast on object to an illegal type
- `java.lang.NullPointerException`
  - Attempt to use an object reference that has not been instantiated
- You can create exceptions, too!
  - An exception is just a class.

```
public class MyException extends Exception { }
```

Here are just a few of the exceptions that Java can throw. You have probably seen one or more of the exceptions listed above while doing the practices or exercises in this class. Did you find the error message helpful when you had to correct the code?

Exceptions are classes. There are many of them included in the Java API. You can also create your own exceptions by simply extending the `java.lang.Exception` class. This is very useful for handling exceptional circumstances that can arise in the normal flow of an application. (Example: `BadCreditException`) This is not covered in this course, but you can learn more about it and other exception handling topics in the *Java SE Programming* II course.



Code Example

Coding mistake:

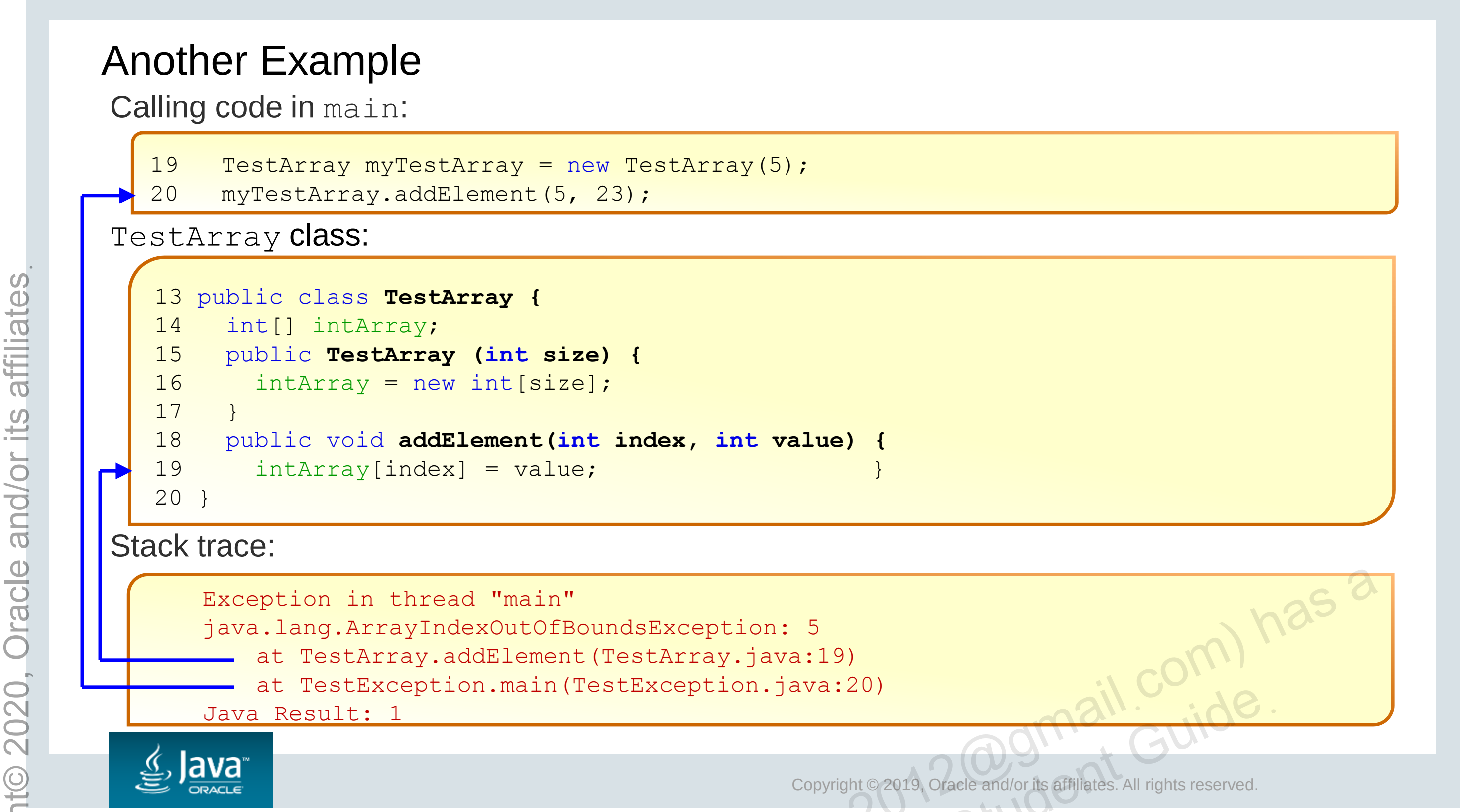
```
int[] intArray = new int[5];
intArray[5] = 27;
```

Output:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
        at TestErrors.main(TestErrors.java:17)
```

This code shows a common mistake made when accessing an array. Remember that arrays are zero based (the first element is accessed by a zero index), so in an array like the one in the slide that has five elements, the last element is actually `intArray[4]`.

`intArray[5]` tries to access an element that does not exist, and Java responds to this programming mistake by throwing an `ArrayIndexOutOfBounds` exception. The information stored within the exception is printed to the console.



Another Example

Calling code in `main`:

```
19   TestArray myTestArray = new TestArray(5);
20   myTestArray.addElement(5, 23);
```

`TestArray` class:

```
13 public class TestArray {
14   int[] intArray;
15   public TestArray (int size) {
16     intArray = new int[size];
17   }
18   public void addElement(int index, int value) {
19     intArray[index] = value;                    }
20 }
```

Stack trace:

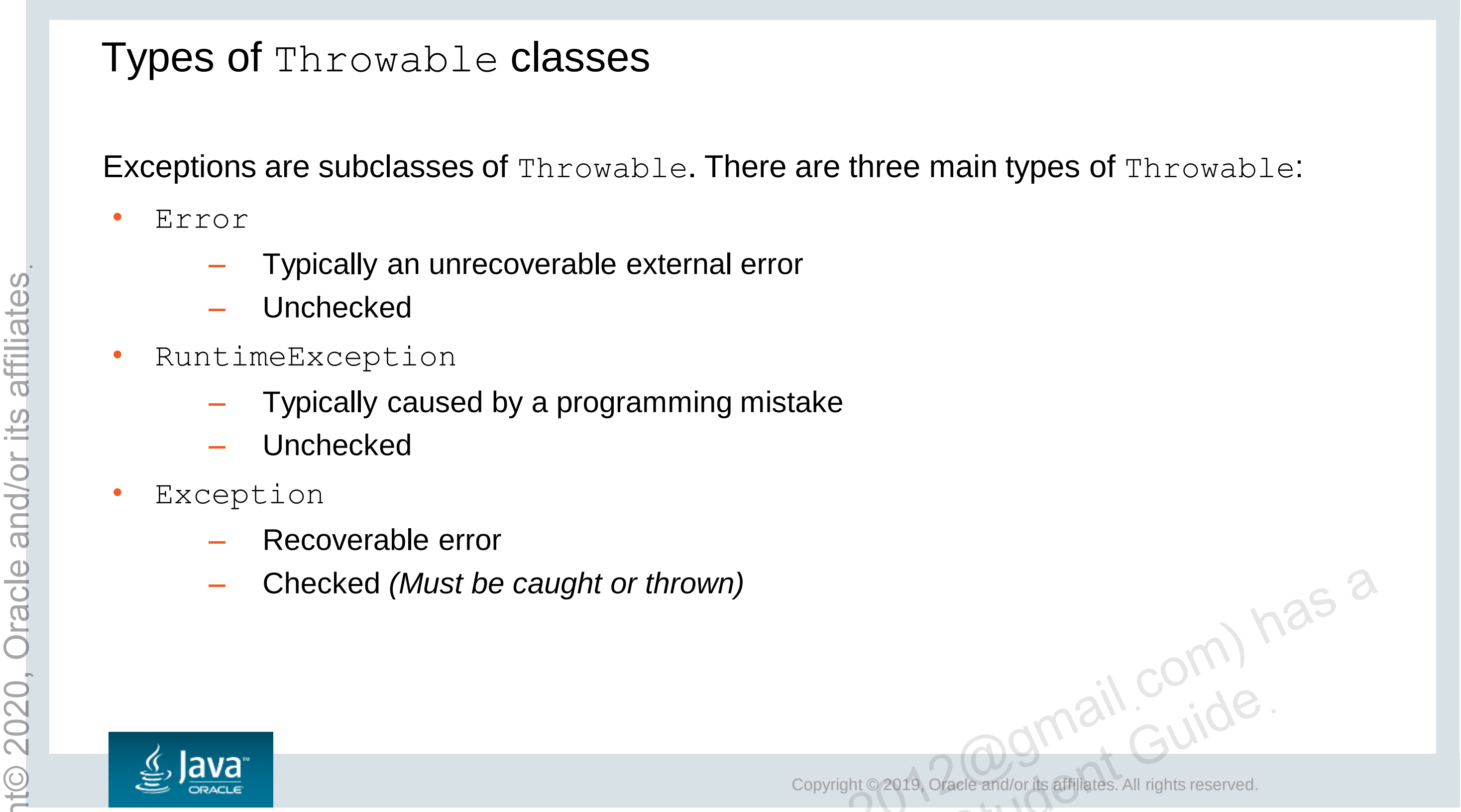
```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 5
    at TestArray.addElement(TestArray.java:19)
    at TestException.main(TestException.java:20)
Java Result: 1
```

Here is a very similar example, except that this time the code that creates the array and tries to assign a value to a nonexistent element has been moved to a different class (`TestArray`). Notice how the error message, shown below, is almost identical to the previous example, but this time the methods `main` in `TestException`, and `addElement` in `TestArray` are explicitly mentioned in the error message. (In NetBeans the message is in red as it is sent to `System.err`).

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
    at TestArray.addElement(TestArray.java:19)
    at TestException.main(TestException.java:20)
Java Result: 1
```

This is called “the stack trace.” It is an unwinding of the sequence of method calls, beginning with where the exception occurred and going backwards.

In this lesson, you learn why that message is printed to the console. You also learn how you can catch or trap the message so that it is not printed to the console, and what other kinds of errors are reported by Java.



Types of Throwable classes

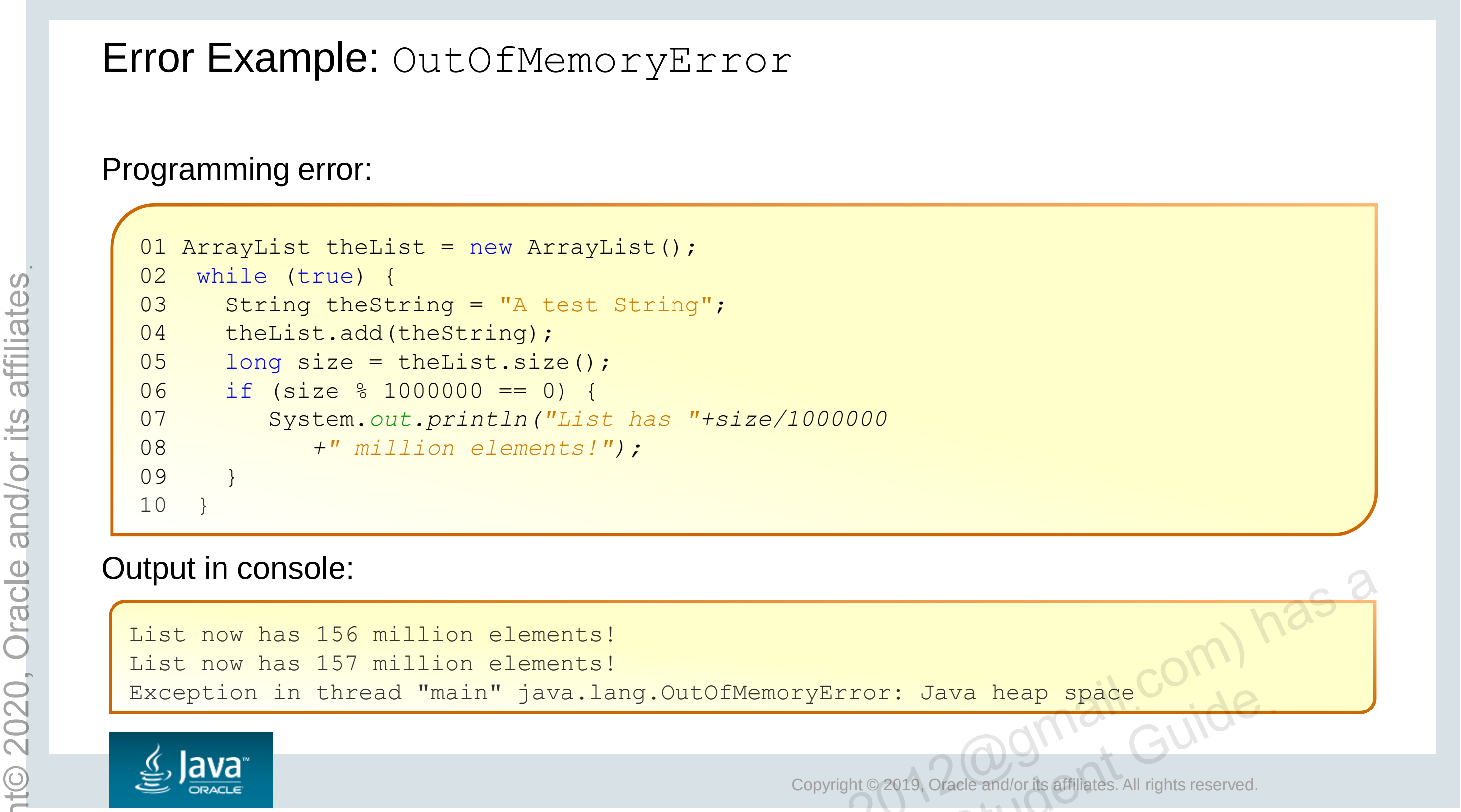
Exceptions are subclasses of `Throwable`. There are three main types of `Throwable`:

- `Error`
  - Typically an unrecoverable external error
  - Unchecked
- `RuntimeException`
  - Typically caused by a programming mistake
  - Unchecked
- `Exception`
  - Recoverable error
  - Checked *(Must be caught or thrown)*

As mentioned in the previous slide, when an exception is thrown, that exception is an object that can be passed to a `catch` block. There are three main types of objects that can be thrown in this way, and all are derived from the class, `Throwable`.

- Only one type, `Exception`, requires that you include a `catch` block to handle the exception. We say that `Exception` is a *checked* exception. You *may* use a `catch` block with the other types, but it is not always possible to recover from these errors anyway.

You learn more about `try/catch` blocks and how to handle exceptions in upcoming slides.



Error Example: `OutOfMemoryError`

Programming error:

```
ArrayList theList = new ArrayList();
 while (true) {
   String theString = "A test String";
   theList.add(theString);
   long size = theList.size();
   if (size % 1000000 == 0) {
      System.out.println("List has "+size/1000000
         +" million elements!");
   }
 }
```

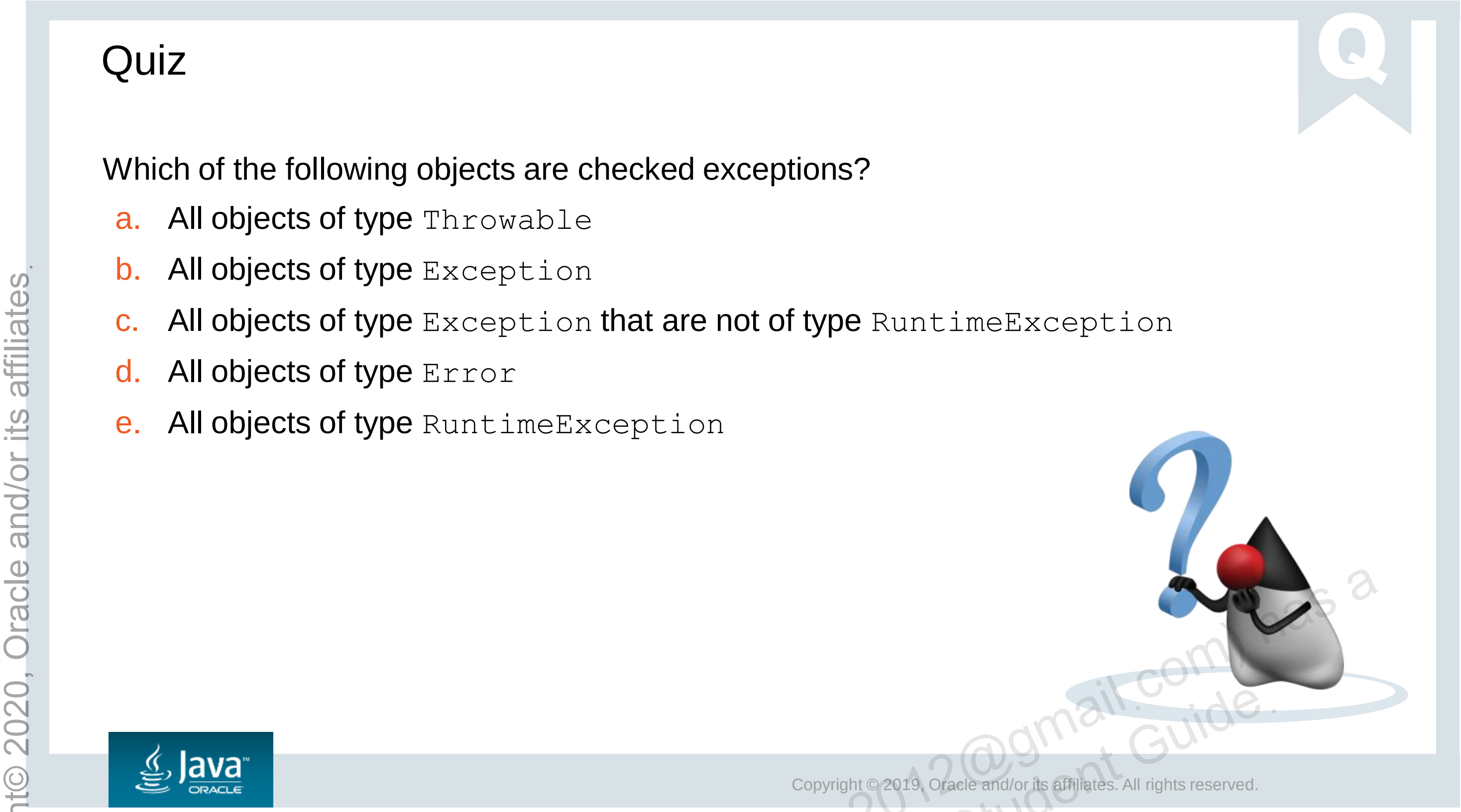
Output in console:

```
List now has 156 million elements!
List now has 157 million elements!
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
```

`OutOfMemoryError` is an `Error`. `Throwable` classes of type `Error` are typically used for exceptional conditions that are external to the application and that the application usually cannot anticipate or recover from. In this case, although it is an external error, it was caused by poor programming.

The example shown here has an infinite loop that continually adds an element to an `ArrayList`, guaranteeing that the JVM will run out of memory. The error is thrown up the call stack, and because it is not caught anywhere, it is displayed in the console as follows:

```
List now has 156 million elements!

List now has 157 million elements!

Exception in thread "main" java.lang.OutOfMemoryError: Java heap space

        at java.util.Arrays.copyOf(Arrays.java:2760)

        at java.util.Arrays.copyOf(Arrays.java:2734)

        at java.util.ArrayList.ensureCapacity(ArrayList.java:167)

        at java.util.ArrayList.add(ArrayList.java:351)

        at TestErrors.main(TestErrors.java:22)
```



Quiz

Which of the following objects are checked exceptions?

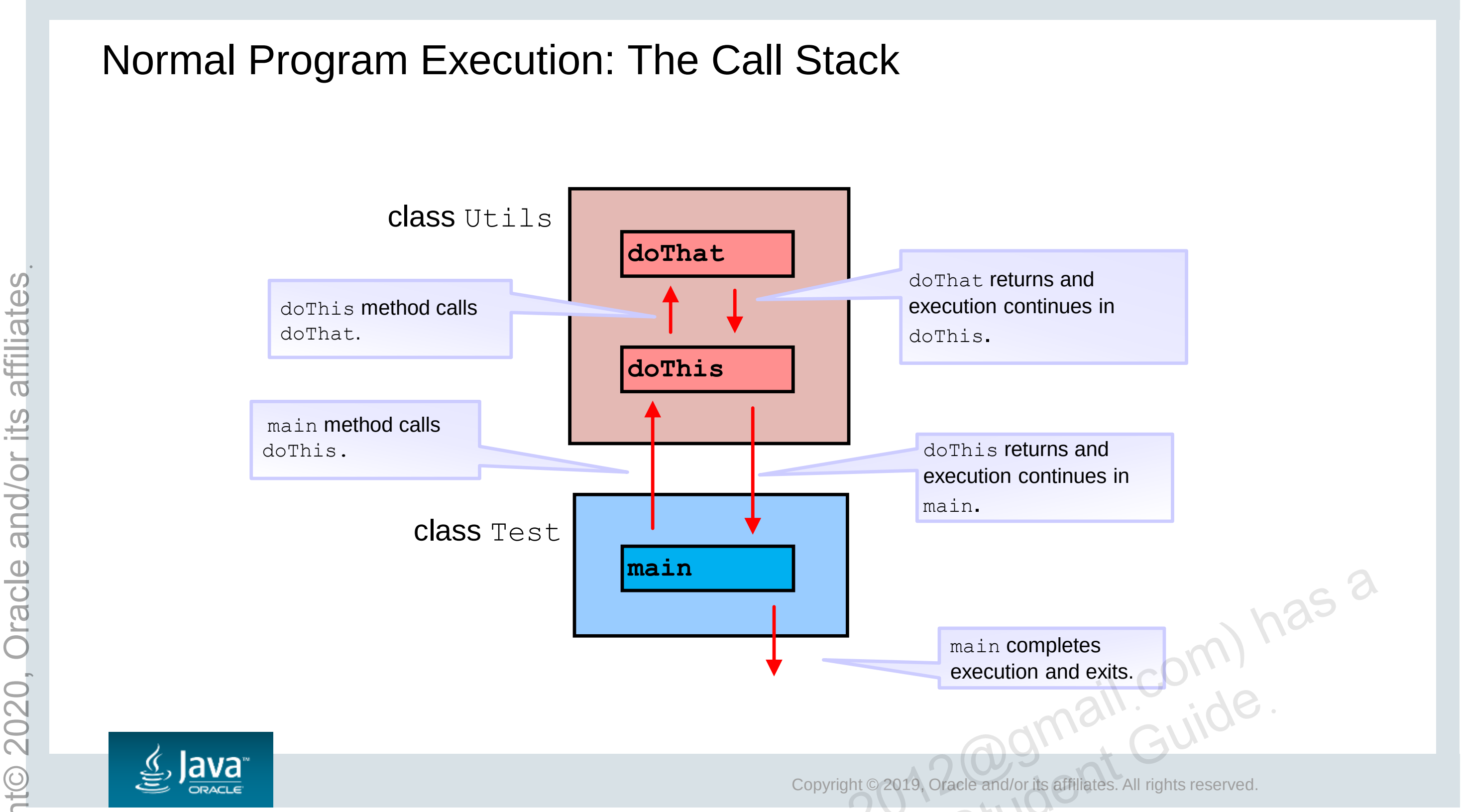
a. All objects of type `Throwable`
b. All objects of type `Exception`
c. All objects of type `Exception` that are not of type `RuntimeException`
d. All objects of type `Error`
e. All objects of type `RuntimeException`

Q
Java ORACLE

Answer: c

Topics

- Handling errors: an overview
- Propagation of exceptions
- Catching and throwing exceptions
- Multiple exceptions and errors



Normal Program Execution: The Call Stack

`doThis` method calls `doThat`.

`doThat` returns and execution continues in `doThis`.

`main` method calls `doThis`.

`doThis` returns and execution continues in `main`.

`main` completes execution and exits.

To understand exceptions, you need to think about how methods call other methods and how this can be nested deeply. The normal mode of operation is that a caller method calls a worker method, which in turn becomes a caller method and calls another worker method, and so on. This sequence of methods is called the *call stack*.

The example shown in the slide illustrates three methods in this relationship.

- The main method in the class `Test`, shown at the bottom of the slide, instantiates an object of type `Utils` and calls the method `doThis` on that object.
- The `doThis` method in turn calls a private method `doThat` on the same object.
- When a method either completes or encounters a return statement, it returns execution to the method that called it. So, `doThat` returns execution to `doThis`, `doThis` returns execution to `main`, and main completes and exits.

How Exceptions Are Thrown

Normal program execution:

1. Caller method calls worker method.
2. Worker method does work.
3. Worker method completes work and then execution returns to caller method.

When an exception occurs, this sequence changes. An exception object is thrown and either:

- Passed to a `catch` block in the current method

or

- Thrown back to the caller method

An `Exception` is one of the subclasses of `Throwable`. `Throwable` objects are thrown either by the runtime engine or explicitly by the developer within the code. A typical thread of execution is described above: A method is invoked, the method is executed, the method completes, and control goes back to the calling method.

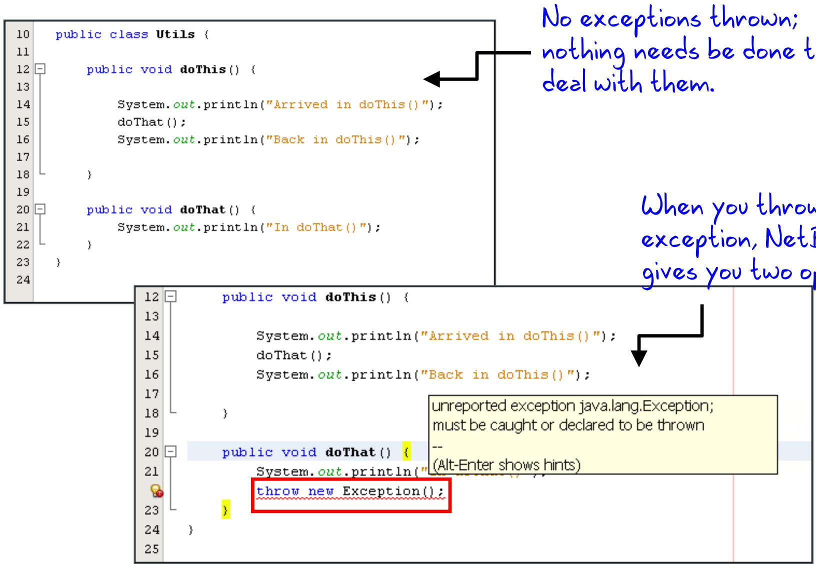
When an exception occurs, however, an `Exception` object containing information about what just happened is thrown. One of two things can happen at this point:

- The `Exception` object is caught by the method that caused it in a special block of code called a `catch` block. In this case, program execution can continue.
- The Exception is not caught, causing the runtime engine to throw it back to the calling method, and look for the exception handler there. Java runtime will keep propagating the exception up the method call stack until it finds a handler. If it is not caught in any method in the call stack, program execution will end and the exception will be printed to the `System.err` (possibly the console) as you saw previously.

Topics

- Handling errors: an overview
- Propagation of exceptions
- Catching and throwing exceptions
- Multiple exceptions and errors

Working with Exceptions in NetBeans



```
public class Utils {

    public void doThis() {

        System.out.println("Arrived in doThis()");
        doThat();
        System.out.println("Back in doThis()");

    }

    public void doThat() {
        System.out.println("In doThat()");
    }
}
```

No exceptions thrown; nothing needs to be done to deal with them.

```
    public void doThis() {

        System.out.println("Arrived in doThis()");
        doThat();
        System.out.println("Back in doThis()");

    }

    public void doThat() {
        System.out.println("In doThat()");
        throw new Exception();
    }
}
```

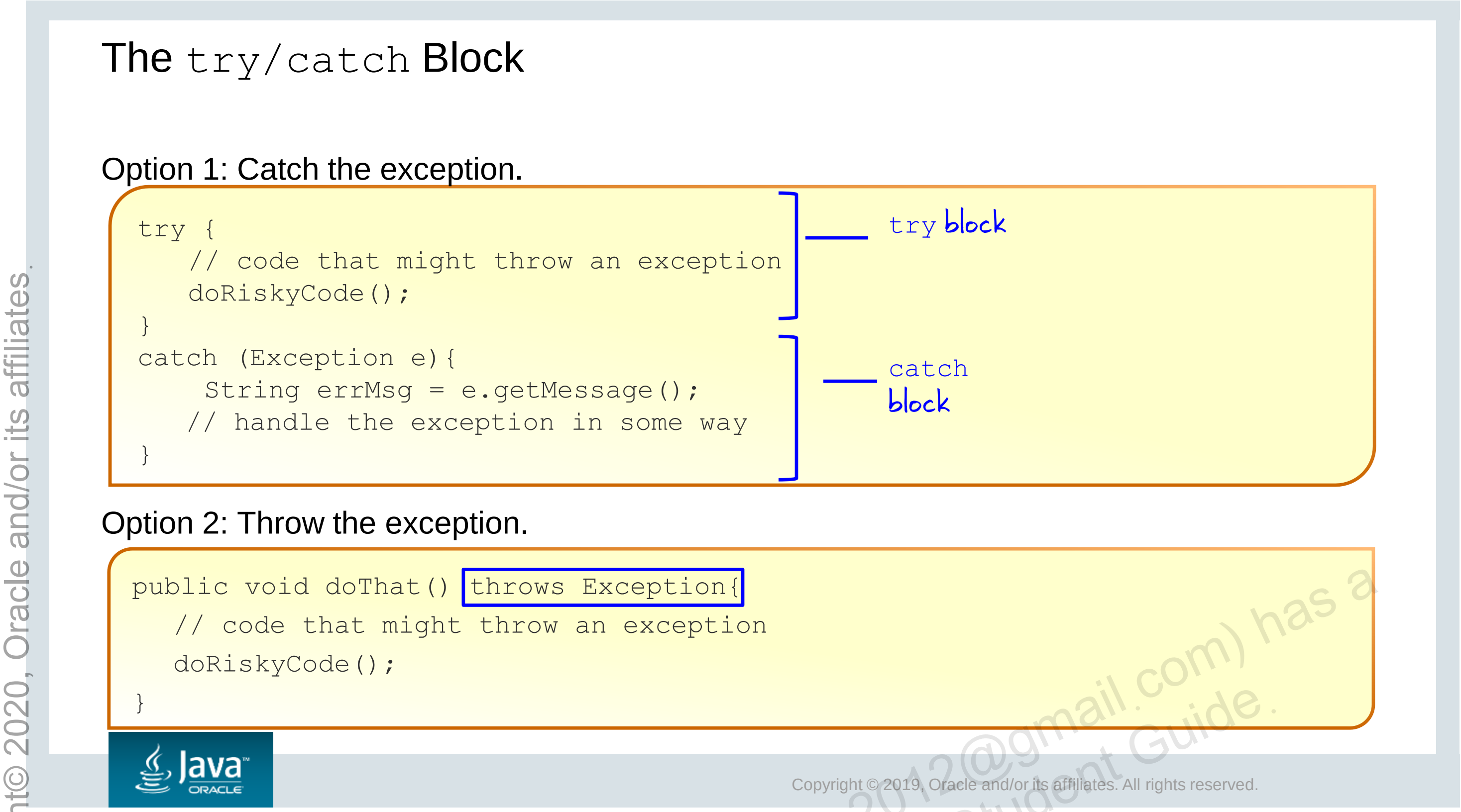
unreported exception java.lang.Exception;
must be caught or declared to be thrown
--
(Alt-Enter shows hints)

When you throw an exception, NetBeans gives you two options.

Here you can see the code for the `Utils` class shown in NetBeans.

- In the first screenshot, no exceptions are thrown, so NetBeans shows no syntax or compilation errors.
- In the second screenshot, `doThat explicitly` throws an exception, and NetBeans flags this as something that needs to be dealt with by the programmer. As you can see from the tooltip, it gives the two options for handling the checked exception: Either catch it, using a `try/catch` block, or allow the method to be thrown to the calling method. If you choose the latter option, you must declare in the method signature that it throws an exception.

In these early examples, the `Exception` superclass is used for simplicity. However, as you will see later, you should not throw so general an exception. Where possible, when you catch an exception, you should try to catch a specific exception.



The try/catch Block

Option 1: Catch the exception.

```
try {
   // code that might throw an exception
   doRiskyCode();
}
catch (Exception e){
    String errMsg = e.getMessage();
   // handle the exception in some way
}
```

try block

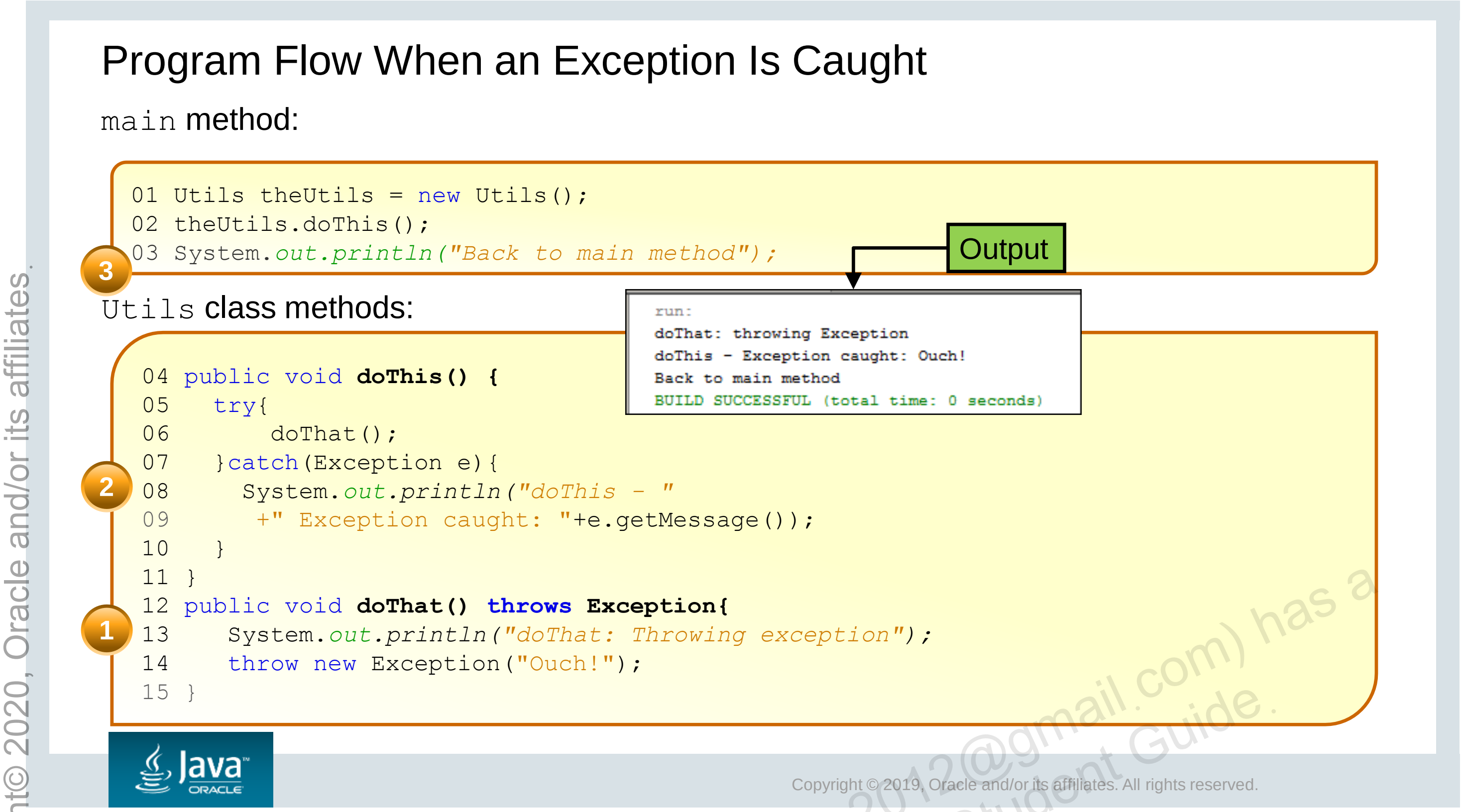
catch block

Option 2: Throw the exception.

```
public void doThat() throws Exception{
   // code that might throw an exception
   doRiskyCode();
}
```

Here is a simple example illustrating both of the options mentioned in the previous slide.

- **Option 1: Catch the exception**.
  - The try block contains code that might throw an exception. For example, you might be casting an object reference and there is a chance that the object reference is not of the type you think it is.
  - The catch block catches the exception. It can be defined to catch a specific exception type (such as ClassCastException) or it can be the superclass Exception, in which case it would catch any subclass of Exception. The exception object will be populated by the runtime engine, so in the catch block, you have access to all the information bundled in it. By catching the exception, the program can continue although it could be in an unstable condition if the error is significant.
  - You may be able to correct the error condition within the catch block. For example, you could determine the type of the object and recast the reference to correct type.
- **Option 2: Declare the method to throw the exception:** In this case, the method declaration includes "throws Exception" (or it could be a specific exception, such as ClassCastException).



Program Flow When an Exception Is Caught

main method:

```
01 Utils theUtils = new Utils();
02 theUtils.doThis();
03 System.out.println("Back to main method");
```

Utils class methods:

```
04 public void doThis() {
05   try{
06       doThat();
07   }catch(Exception e){
08     System.out.println("doThis - "
09      +" Exception caught: "+e.getMessage());
10   }
11 }
12 public void doThat() throws Exception{
13   System.out.println("doThat: Throwing exception");
14   throw new Exception("Ouch!");
15 }
```

Output

```
run:
doThat: throwing Exception
doThis - Exception caught: Ouch!
Back to main method
BUILD SUCCESSFUL (total time: 0 seconds)
```

In this example, a try/catch block has been added to the doThis method. The slide also illustrates the program flow when the exception is thrown and caught by the calling method. The Output insert shows the output from the doThat method, followed by the output from the catch block of doThis and, finally, the last line of the main method.

main method code:

- In line 1, a Utils object is instantiated.
- In line 2, the doThis method of the Utils object is invoked.

Execution now goes to the Utils class:

- In line 6 of doThis, doThat is invoked from within a try block. Notice that in line 7, the catch block is declared to catch the exception.

Execution now goes to the doThat method:

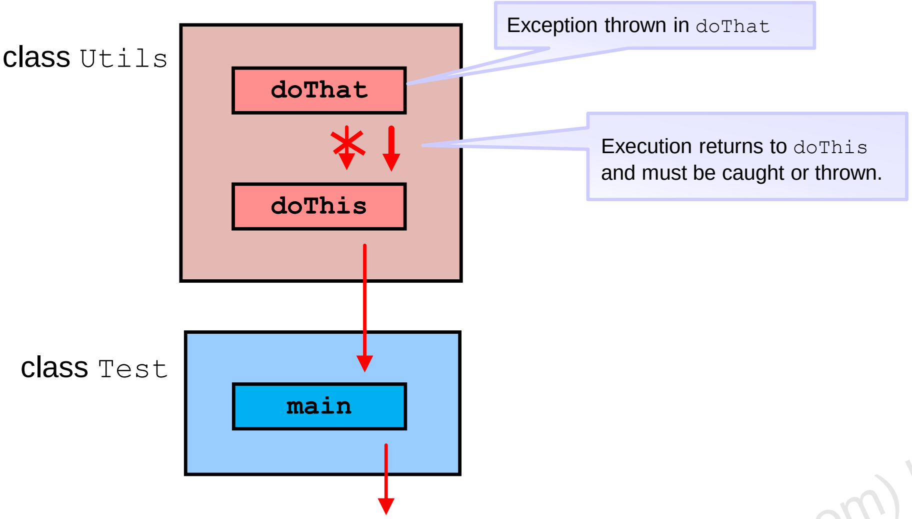
- In line 14, doThat explicitly throws a new Exception object.

Execution now returns to doThis:

- In line 8 of doThis, the exception is caught and the message property from the Exception object is printed. The doThat method completes at the end of the catch block.

Execution now returns to the main method where line 3 is executed.

When an Exception Is Thrown

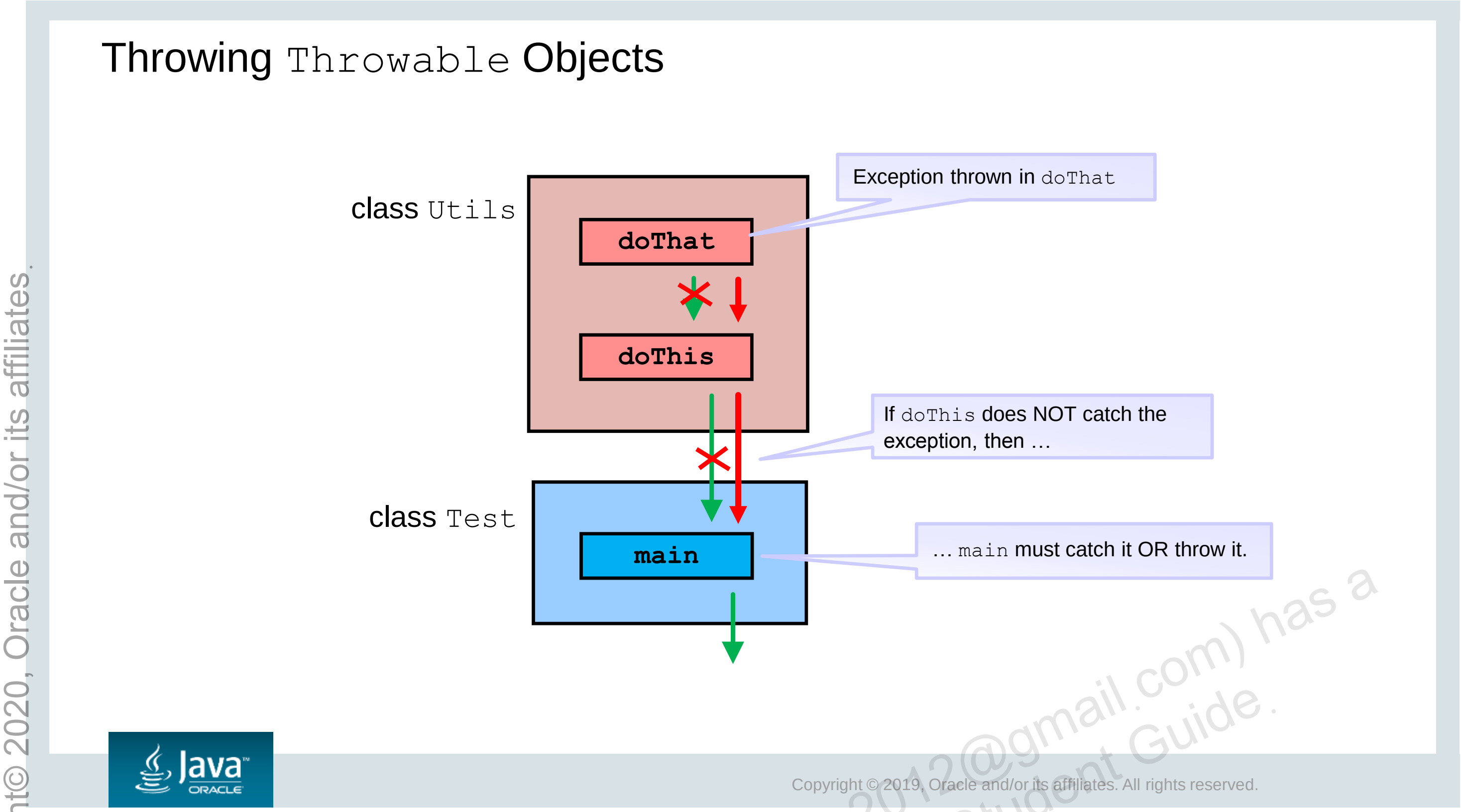


Exception thrown in doThat

Execution returns to doThis and must be caught or thrown.

As mentioned previously, when a method finishes executing, the *normal* flow (on completion of the method or on a return statement) goes back to the calling method and continues execution at the next line of the calling method.

When an exception is thrown, program flow returns to the calling method, but *not* to the point just after the method call. Instead, if there is a `try/catch` block, program flow goes to the `catch` block associated with the `try` block that contains the method call. You will see in the next slide what happens if there is no `try/catch` block in `doThis`.



Throwing `Throwable` Objects

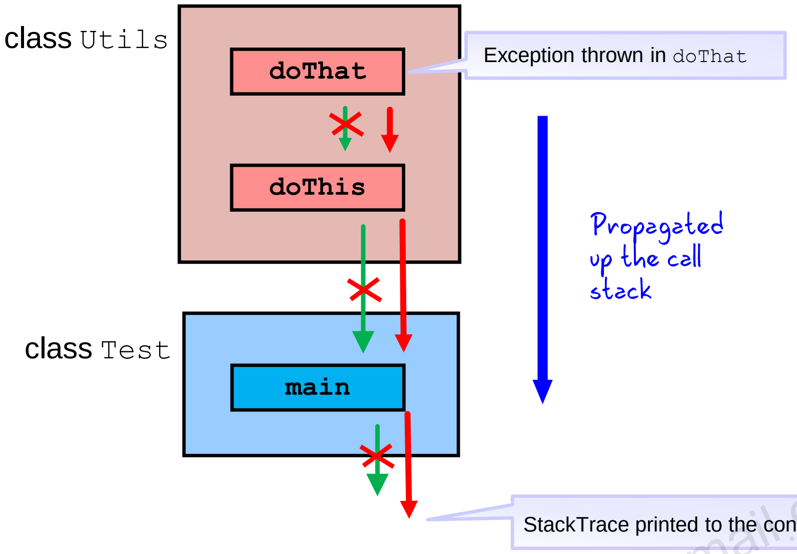
Exception thrown in `doThat`

If `doThis` does NOT catch the exception, then …

… `main` must catch it OR throw it.

The diagram in the slide illustrates an exception originally thrown in `doThat` being thrown to `doThis`. The error is not caught there, so it is thrown to its caller method, which is the `main` method. The thing to remember is that the exception will continue to be thrown back up the call stack until it is caught.

Uncaught Exception



Exception thrown in doThat

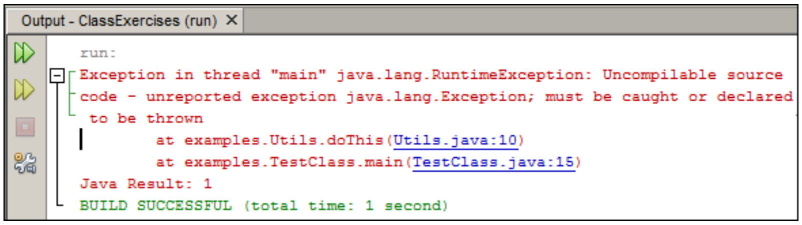
Propagated up the call stack

StackTrace printed to the console

But what happens if none of the methods in the call stack have `try/catch` blocks? That situation is illustrated by the diagram shown in this slide. Because there are no `try/catch` blocks, the exception is propagated all the way up the call stack. But what happens when it gets to the `main` method and is not handled there? This causes the program to exit, and the exception, plus a stack trace for the exception, is printed to the console.

Exception Printed to Console

When the exception is thrown up the call stack without being caught, it will eventually reach the JVM. The JVM will print the exception's output to the console and exit.

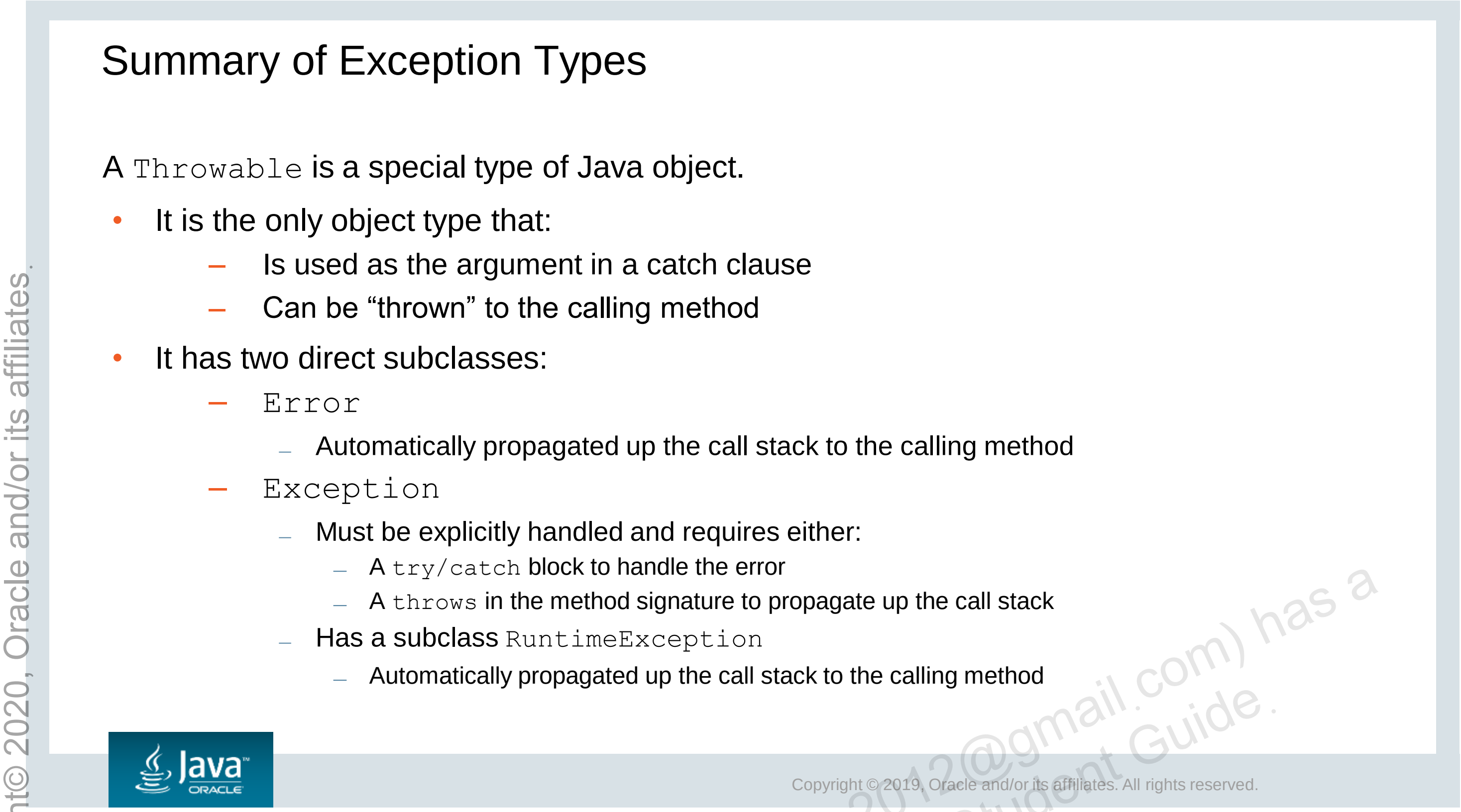


```
run:
Exception in thread "main" java.lang.RuntimeException: Uncompilable source
code - unreported exception java.lang.Exception; must be caught or declared
 to be thrown
        at examples.Utils.doThis(Utils.java:10)
        at examples.TestClass.main(TestClass.java:15)
Java Result: 1
BUILD SUCCESSFUL (total time: 1 second)
```

In the example, you can see what happens when the exception is propagated up the call stack all the way to the `main` method. Did you notice how similar this looks to the first example you saw of an `ArrayIndexOutOfBoundsException`? In both cases, the exception is displayed as a stack trace to the console.

There was something different about the `ArrayIndexOutOfBoundsException`: None of the methods threw that exception! So how did it get passed up the call stack?

The answer is that `ArrayIndexOutOfBoundsException` is a `RuntimeException`. The `RuntimeException` class is a subclass of the `Exception` class, but it is not a checked exception so its exceptions are automatically propagated up the call stack without `throws` being explicitly declared in the method signature.



Summary of Exception Types

A Throwable is a special type of Java object.

- It is the only object type that:
  - Is used as the argument in a catch clause
  - Can be “thrown” to the calling method
- It has two direct subclasses:
  - Error
    - Automatically propagated up the call stack to the calling method
  - Exception
    - Must be explicitly handled and requires either:
      - A try/catch block to handle the error
      - A throws in the method signature to propagate up the call stack
    - Has a subclass RuntimeException
      - Automatically propagated up the call stack to the calling method

An Exception that is not a RuntimeException must be explicitly handled.

- An Error is usually so critical that it is unlikely that you could recover from it, even if you anticipated it. You are not required to check these exceptions in your code.
- An Exception represents an event that could happen and which may be recoverable. You are required to either catch an Exception within the method that generates it or throw it to the calling method.
- A RuntimeException is usually the result of a system error (out of memory, for instance). They are inherited from Exception. You are not required to check these exceptions in your code, but sometimes it makes sense to do so. They can also be the result of a programming error (for instance, ArrayIndexOutOfBounds is one of these exceptions).

The examples later in this lesson show you how to work with an IOException.

Exercise 14-1: Catching an Exception

1. Open the project **Exercise_14-1** in NetBeans.

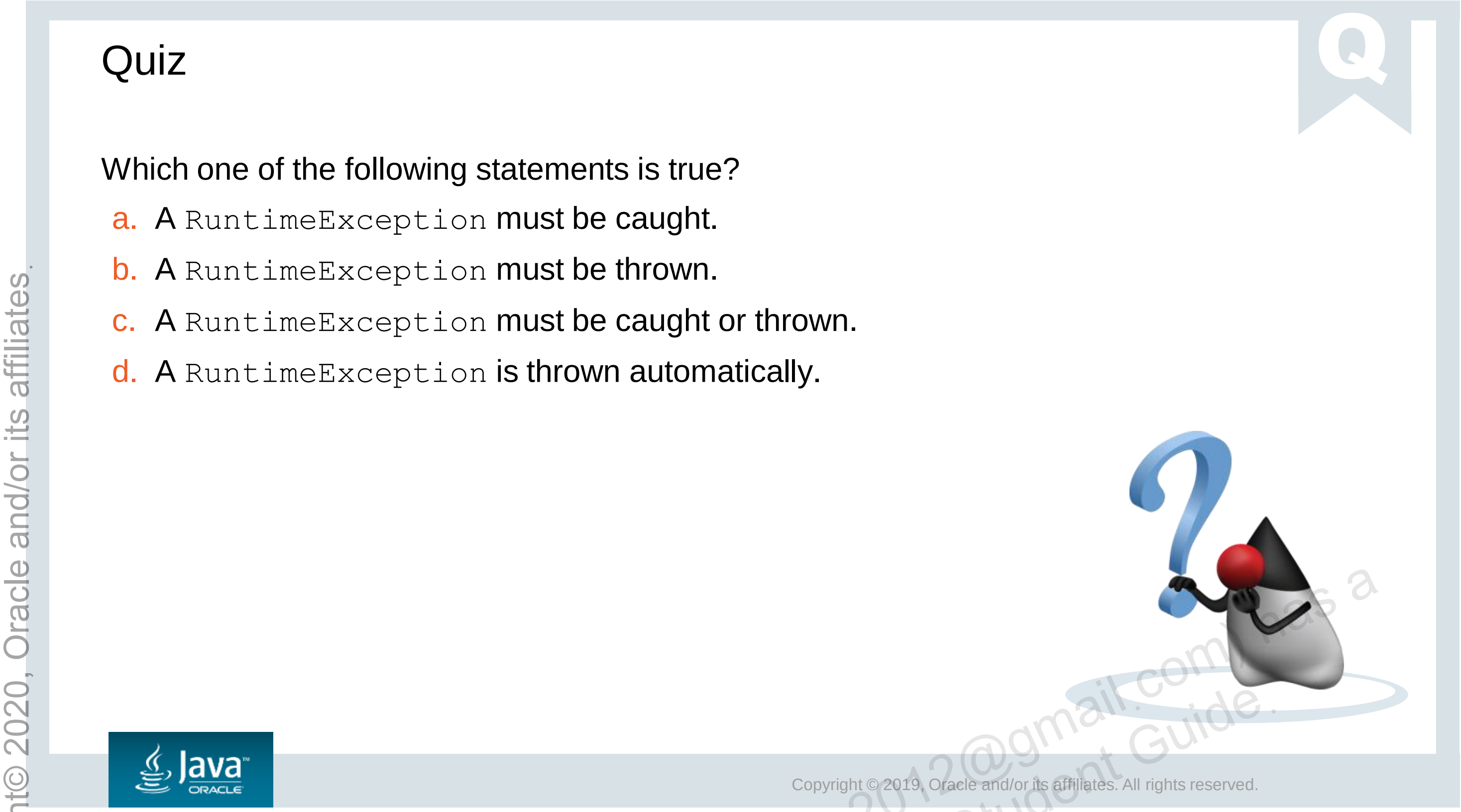
In the Calculator class:

2. Change the divide method signature so that it throws an ArithmeticException.

In the TestClass class:

3. Surround the code that calls the divide method with a try/catch block. Handle the exception object by printing it to the console.
4. Run the TestClass to view the outcome.

In this exercise, you implement exception handling. Change a method signature to indicate that it throws an exception. Then catch the exception in the class that calls the method.



Quiz

Which one of the following statements is true?

a. A `RuntimeException` must be caught.

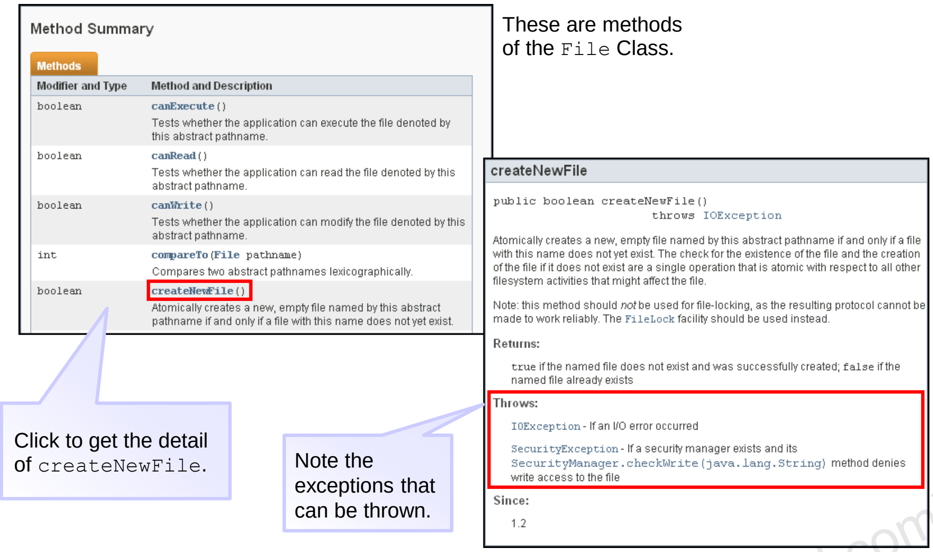
b. A `RuntimeException` must be thrown.

c. A `RuntimeException` must be caught or thrown.

d. A `RuntimeException` is thrown automatically.

Answer: d

Exceptions in the Java API Documentation



Method Summary

Methods

Modifier and Type	Method and Description
`boolean`	`canExecute()` Tests whether the application can execute the file denoted by this abstract pathname.
`boolean`	`canRead()` Tests whether the application can read the file denoted by this abstract pathname.
`boolean`	`canWrite()` Tests whether the application can modify the file denoted by this abstract pathname.
`int`	`compareTo(File pathname)` Compares two abstract pathnames lexicographically.
`boolean`	`createNewFile()` Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist.

These are methods of the `File` Class.

createNewFile

```
public boolean createNewFile()
                      throws IOException
```

Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist. The check for the existence of the file and the creation of the file if it does not exist are a single operation that is atomic with respect to all other filesystem activities that might affect the file.

Note: this method should *not* be used for file-locking, as the resulting protocol cannot be made to work reliably. The `FileLock` facility should be used instead.

Returns:

`true` if the named file does not exist and was successfully created; `false` if the named file already exists

Throws:

`IOException` - If an I/O error occurred

`SecurityException` - If a security manager exists and its `SecurityManager.checkWrite(java.lang.String)` method denies write access to the file

Since:

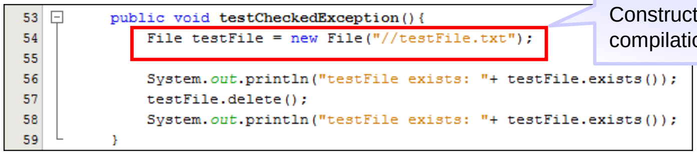
1.2

Click to get the detail of `createNewFile`.

Note the exceptions that can be thrown.

When working with any API, it is necessary to determine what exceptions are thrown by the object's constructors or methods. The example in the slide is for the `File` class. `File` has a `createNewFile` method that can throw an `IOException` or a `SecurityException`. `SecurityException` is a `RuntimeException`, so `SecurityException` is unchecked but `IOException` is a checked exception.

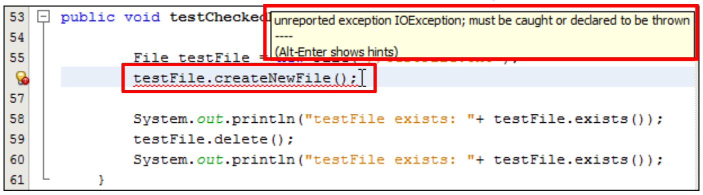
Calling a Method That Throws an Exception



```
    public void testCheckedException(){
        File testFile = new File("//testFile.txt");

        System.out.println("testFile exists: "+ testFile.exists());
        testFile.delete();
        System.out.println("testFile exists: "+ testFile.exists());
    }
```

Constructor causes no compilation problems.

`createNewFile` can throw a checked exception, so the method must throw or catch.



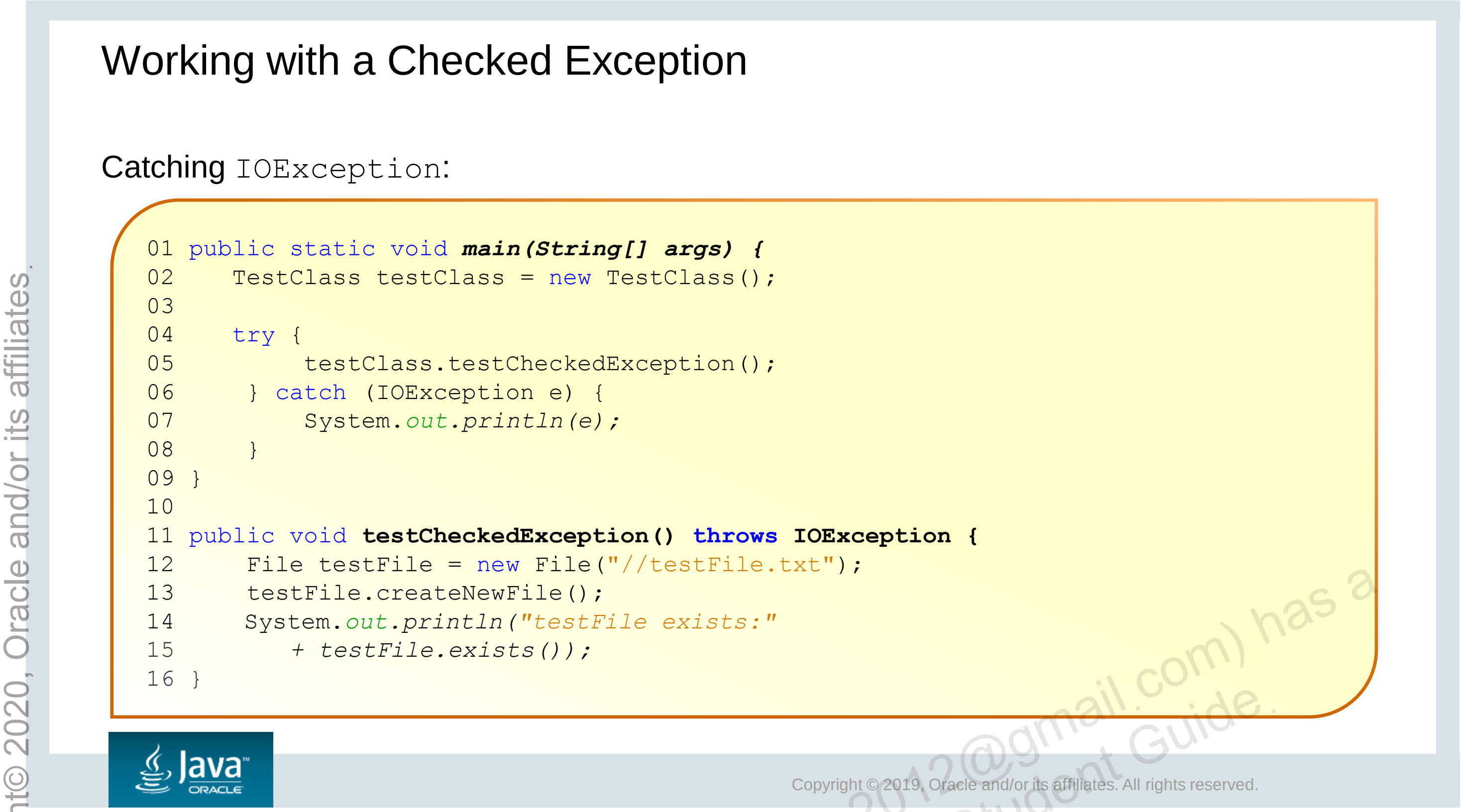
```
public void testChecked

        File testFile =
        testFile.createNewFile();

        System.out.println("testFile exists: "+ testFile.exists());
        testFile.delete();
        System.out.println("testFile exists: "+ testFile.exists());
    }
```

```
unreported exception IOException; must be caught or declared to be thrown
----
(Alt-Enter shows hints)
```

The two screenshots in the slide show a simple `testCheckedException` method. In the first example, the `File` object is created using the constructor. Note that even though the constructor can throw a `NullPointerException` (if the constructor argument is null), you are not forced to catch this exception.

However, in the second example, `createNewFile` can throw an `IOException`, and NetBeans shows that you must deal with this.

Note that File is introduced here only to illustrate an `IOException`. In the next course (*Java SE Programming II*), you learn about the File class and a new set of classes in the package `java.nio`, which provides more elegant ways to work with files.



Working with a Checked Exception

Catching `IOException`:

```
public static void main(String[] args) {
   TestClass testClass = new TestClass();

   try {
        testClass.testCheckedException();
    } catch (IOException e) {
        System.out.println(e);
    }
}

public void testCheckedException() throws IOException {
    File testFile = new File("//testFile.txt");
    testFile.createNewFile();
    System.out.println("testFile exists:"
       + testFile.exists());
}
```

The example in the slide is handling the possible raised exception by:

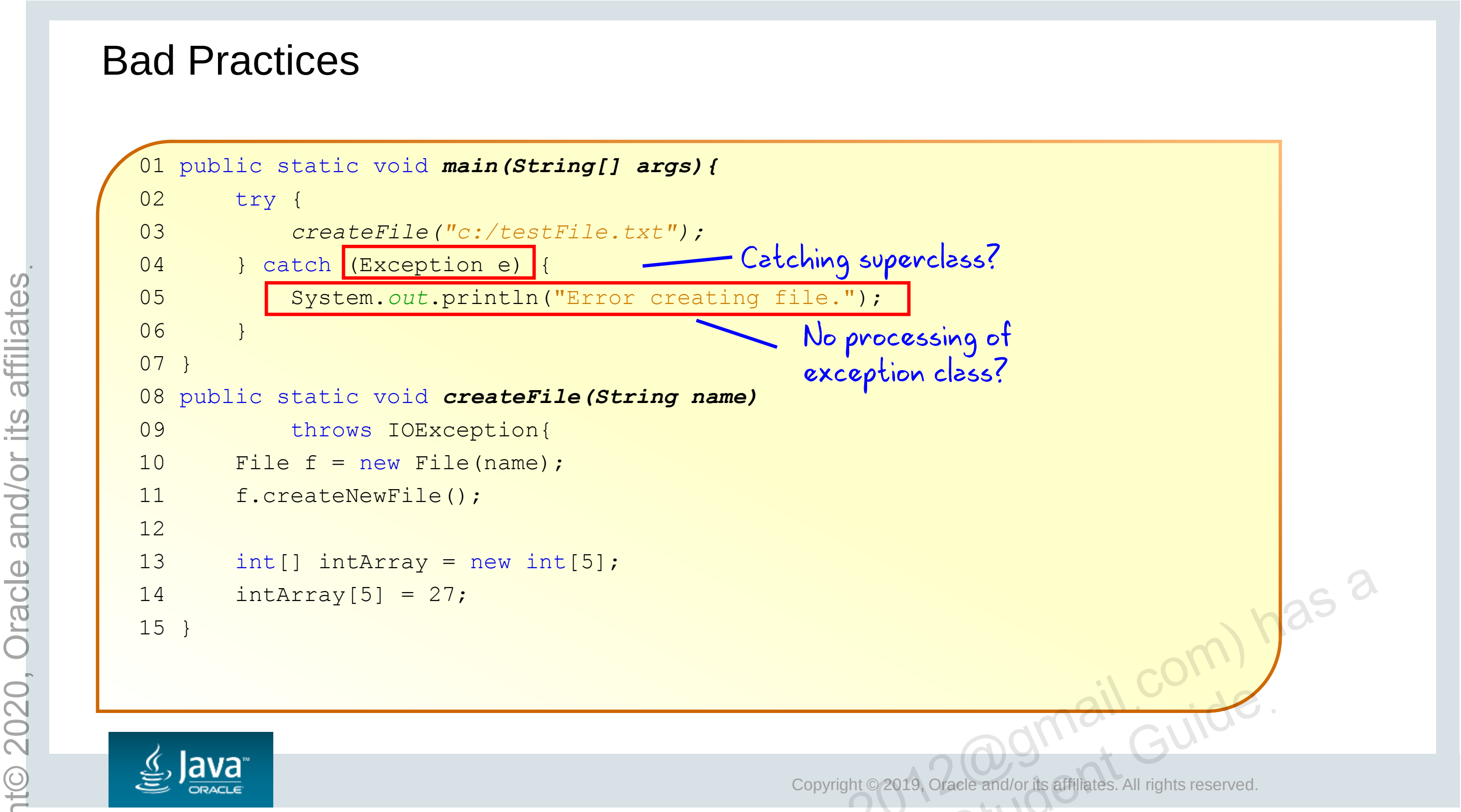
- Throwing the exception from the `testCheckedException` method
- Catching the exception in the caller method

In this example, the `catch` method catches the exception because the path to the text file is not correctly formatted. `System.out.println(e)` calls the `toString` method of the exception, and the result is as follows:

```
java.io.IOException: The filename, directory name, or volume label syntax is
incorrect
```

Best Practices

- Catch the actual exception thrown, not the superclass type.
- Examine the exception to find out the exact problem so you can recover cleanly.
- You do not need to catch every exception.
  - A programming mistake should not be handled. It must be fixed.
  - Ask yourself, “Does this exception represent behavior I want the program to recover from?”



Bad Practices

```
public static void main(String[] args){
    try {
        createFile("c:/testFile.txt");
    } catch (Exception e) {
        System.out.println("Error creating file.");
    }
}
public static void createFile(String name)
        throws IOException{
    File f = new File(name);
    f.createNewFile();

    int[] intArray = new int[5];
    intArray[5] = 27;
}
```

Catching superclass?

No processing of exception class?

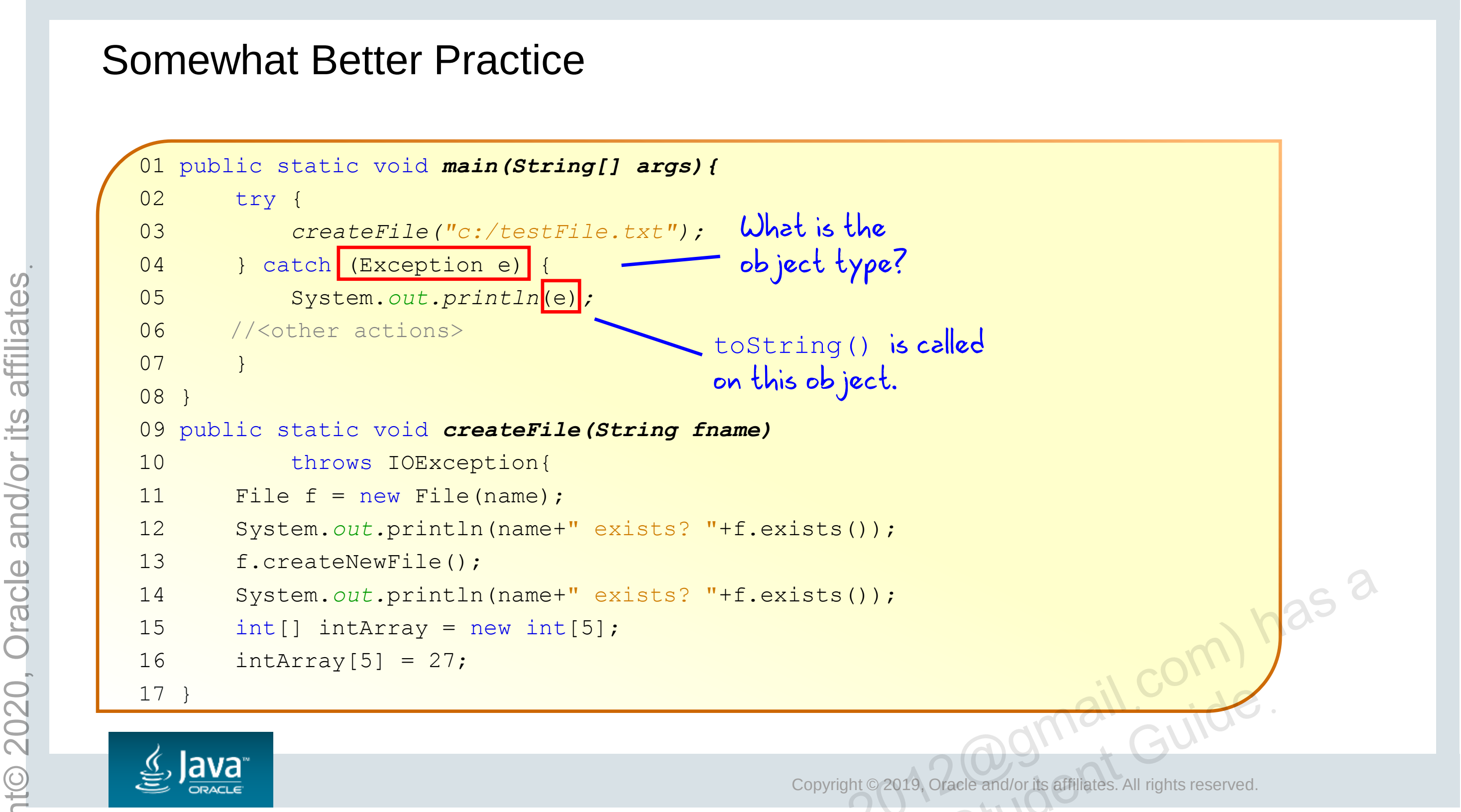
The code in the slide illustrates two poor programming practices.

1. The `catch` clause catches an `Exception` type rather than an `IOException` type (the expected exception from calling the `createFile` method).
2. The `catch` clause does not analyze the `Exception` object and instead simply assumes that the expected exception has been thrown from the `File` object.

A major drawback of this careless programming style is shown by the fact that the code prints the following message to the console:

```
There is a problem creating the file!
```

This suggests that the file has not been created, and indeed any further code in the `catch` block will run. But what is actually happening in the code?



Somewhat Better Practice

```
01 public static void main(String[] args){
02     try {
03         createFile("c:/testFile.txt");
04     } catch (Exception e) {
05         System.out.println(e);
06     //<other actions>
07     }
08 }
09 public static void createFile(String fname)
10         throws IOException{
11     File f = new File(name);
12     System.out.println(name+" exists? "+f.exists());
13     f.createNewFile();
14     System.out.println(name+" exists? "+f.exists());
15     int[] intArray = new int[5];
16     intArray[5] = 27;
17 }
```

What is the object type?

toString() is called on this object.

Putting in a few `System.out.println` calls in the `createFile` method may help clarify what is happening. The output now is:

```
C:/testFile.txt exists? false (from line 12)
C:/testFile.txt exists? true (from line 14)
java.lang.ArrayIndexOutOfBoundsException: 5
```

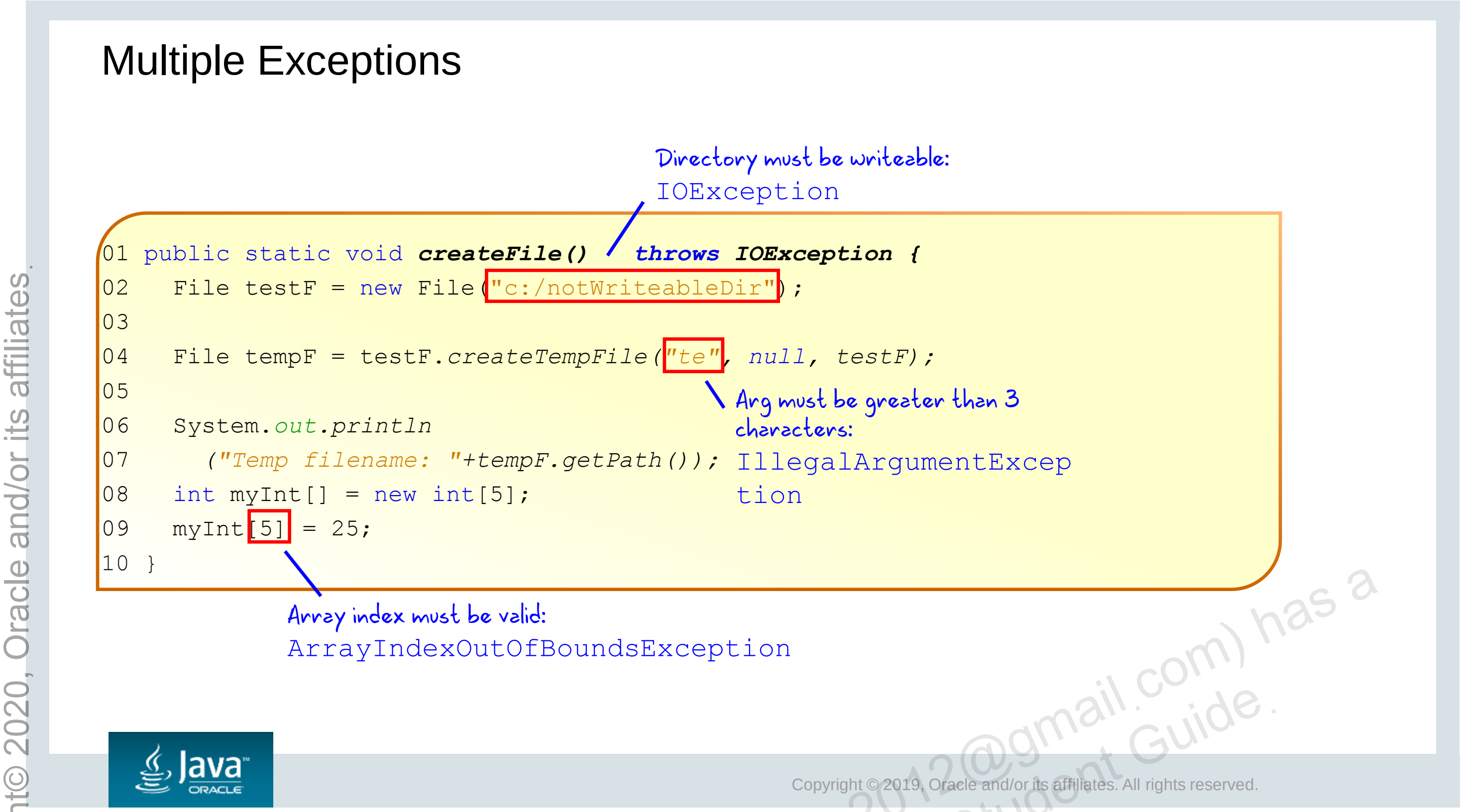
So the file is being created! And you can see that the exception is actually an `ArrayIndexOutOfBoundsException` that is being thrown by the final line of code in `createFile`.

In this example, it is obvious that the array assignment can throw an exception, but it may not be so obvious. In this case, the `createNewFile` method of `File` actually throws another exception—a `SecurityException`. Because it is an unchecked exception, it is thrown automatically.

If you check for the specific exception in the `catch` clause, you remove the danger of assuming what the problem is.

Topics

- Handling errors: an overview
- Propagation of exceptions
- Catching and throwing exceptions
- Multiple exceptions and errors



Multiple Exceptions

```
public static void createFile()  throws IOException {
  File testF = new File("c:/notWriteableDir");

  File tempF = testF.createTempFile("te", null, testF);

  System.out.println
    ("Temp filename: "+tempF.getPath());
  int myInt[] = new int[5];
  myInt[5] = 25;
}
```

Directory must be writeable: IOException

Arg must be greater than 3 characters: IllegalArgumentException

Array index must be valid: ArrayIndexOutOfBoundsException

The example in the slide shows a method that could potentially throw three different exceptions. It uses the `createTempFile` `File` method, which creates a temporary file. (It ensures that each call creates a new and different file and also can be set up so that the temporary files created are deleted on exit.)

The three exceptions are the following:

`IOException`

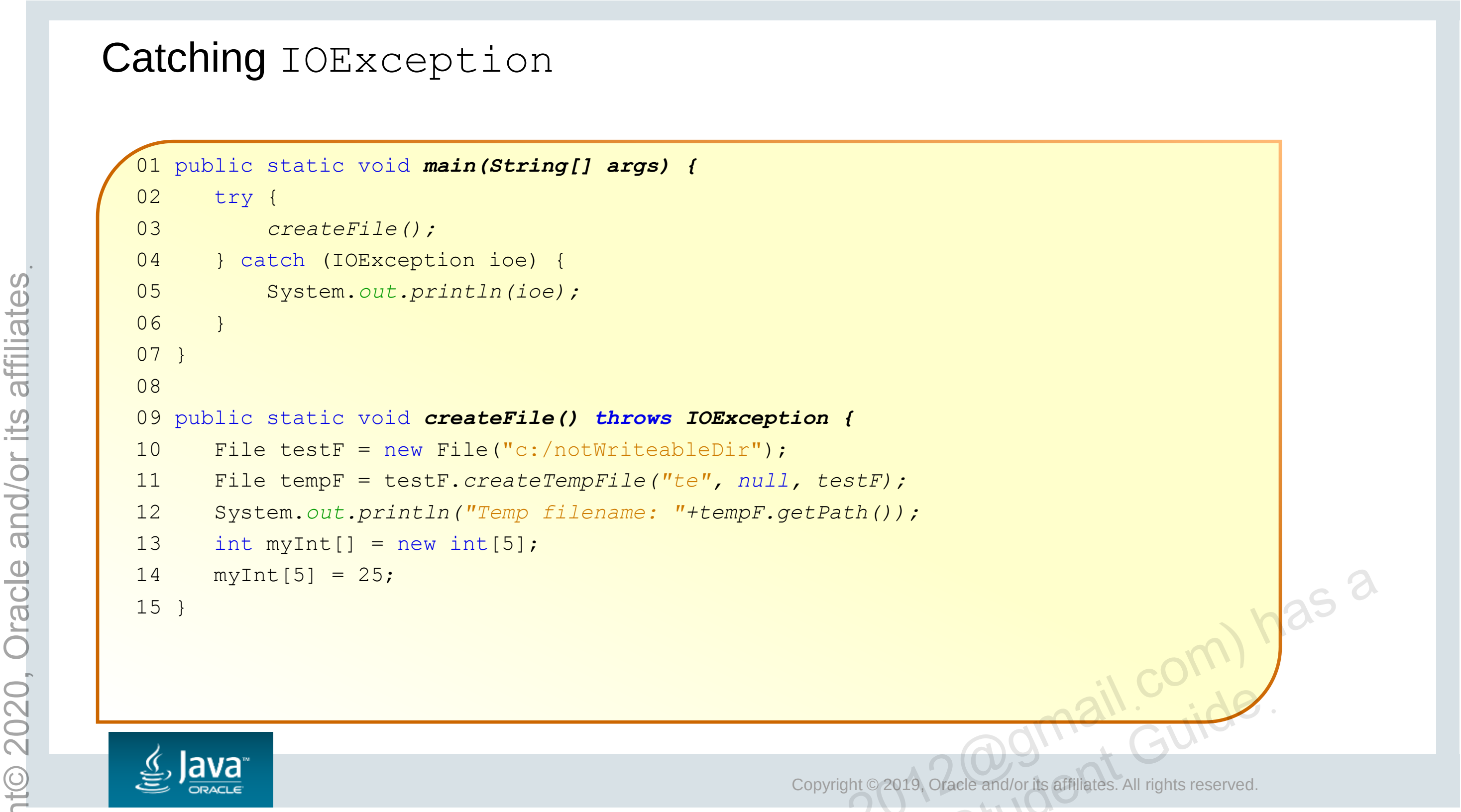
`c:\notWriteableDir` is a directory, but it is not writable. This causes `createTempFile()` to throw an `IOException` (checked).

`IllegalArgumentException`

The first argument passed to `createTempFile` should be three or more characters long. If it is not, the method throws an `IllegalArgumentException` (unchecked).

`ArrayIndexOutOfBoundsException`

As in previous examples, trying to access a nonexistent index of an array throws an `ArrayIndexOutOfBoundsException` (unchecked).



Catching `IOException`

```java
public static void main(String[] args) {
    try {
        createFile();
    } catch (IOException ioe) {
        System.out.println(ioe);
    }
}

public static void createFile() throws IOException {
    File testF = new File("c:/notWriteableDir");
    File tempF = testF.createTempFile("te", null, testF);
    System.out.println("Temp filename: "+tempF.getPath());
    int myInt[] = new int[5];
    myInt[5] = 25;
}
```

The example in the slide shows the minimum exception handling (the compiler insists on at least the `IOException` being handled).

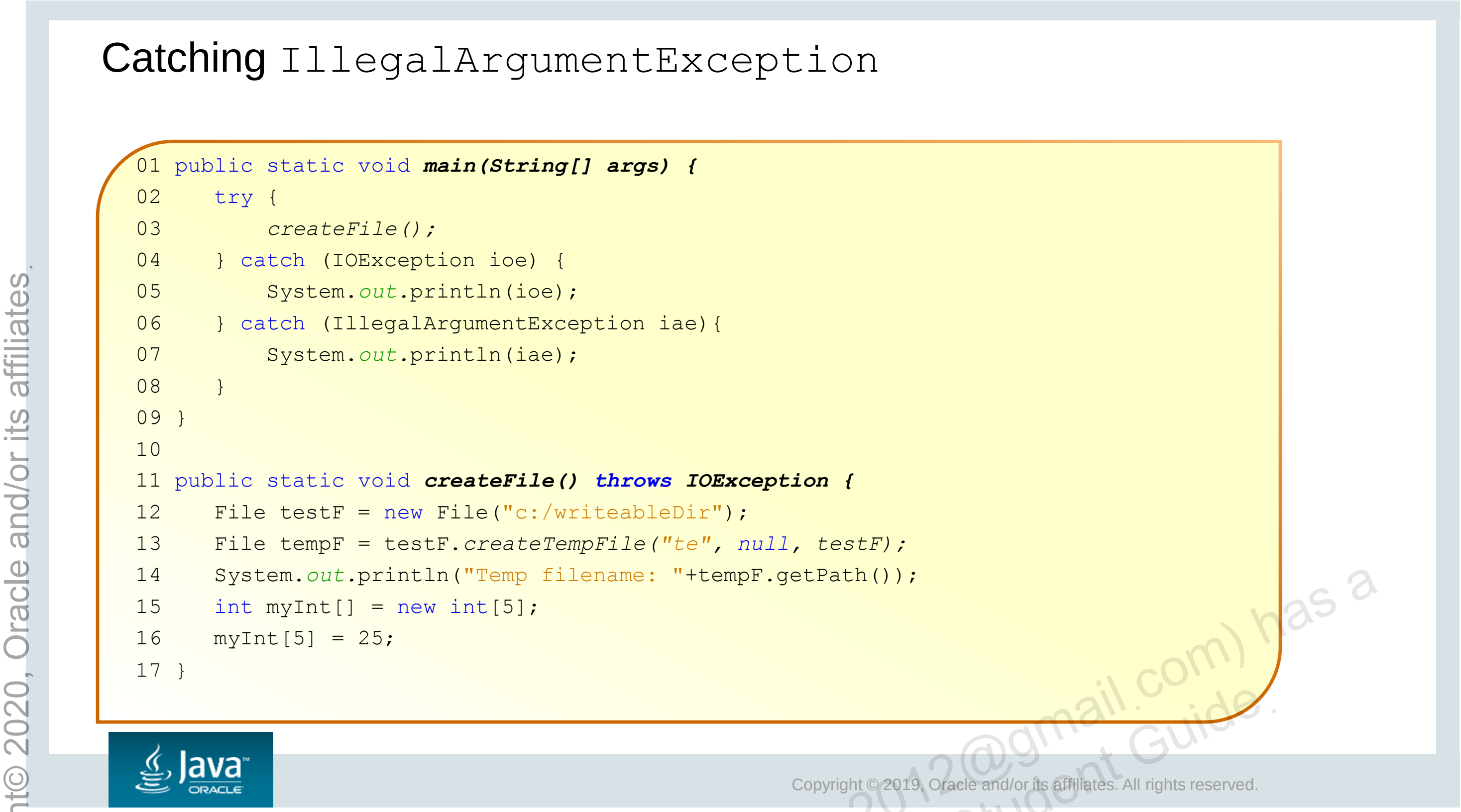
With the directory is set as shown at `c:/notWriteableDir`, the output of this code is:

```
java.io.IOException: Permission denied
```

However, if the file is set as `c:/writeableDir` (a writable directory), the output is now:

```
Exception in thread "main" java.lang.IllegalArgumentException: Prefix string
too short
    at java.io.File.createTempFile(File.java:1782)
    at MultipleExceptionExample.createFile(MultipleExceptionExample.java:34)
    at MultipleExceptionExample.main(MultipleExceptionExample.java:18)
```

The argument `"te"` causes an `IllegalArgumentException` to be thrown, and because it is a `RuntimeException`, it gets thrown all the way out to the console.



Catching `IllegalArgumentException`

```
public static void main(String[] args) {
   try {
       createFile();
   } catch (IOException ioe) {
       System.out.println(ioe);
   } catch (IllegalArgumentException iae){
       System.out.println(iae);
   }
}

public static void createFile() throws IOException {
   File testF = new File("c:/writeableDir");
   File tempF = testF.createTempFile("te", null, testF);
   System.out.println("Temp filename: "+tempF.getPath());
   int myInt[] = new int[5];
   myInt[5] = 25;
}
```

The example in the slide shows an additional `catch` clause added to catch the potential `IllegalArgumentException`.

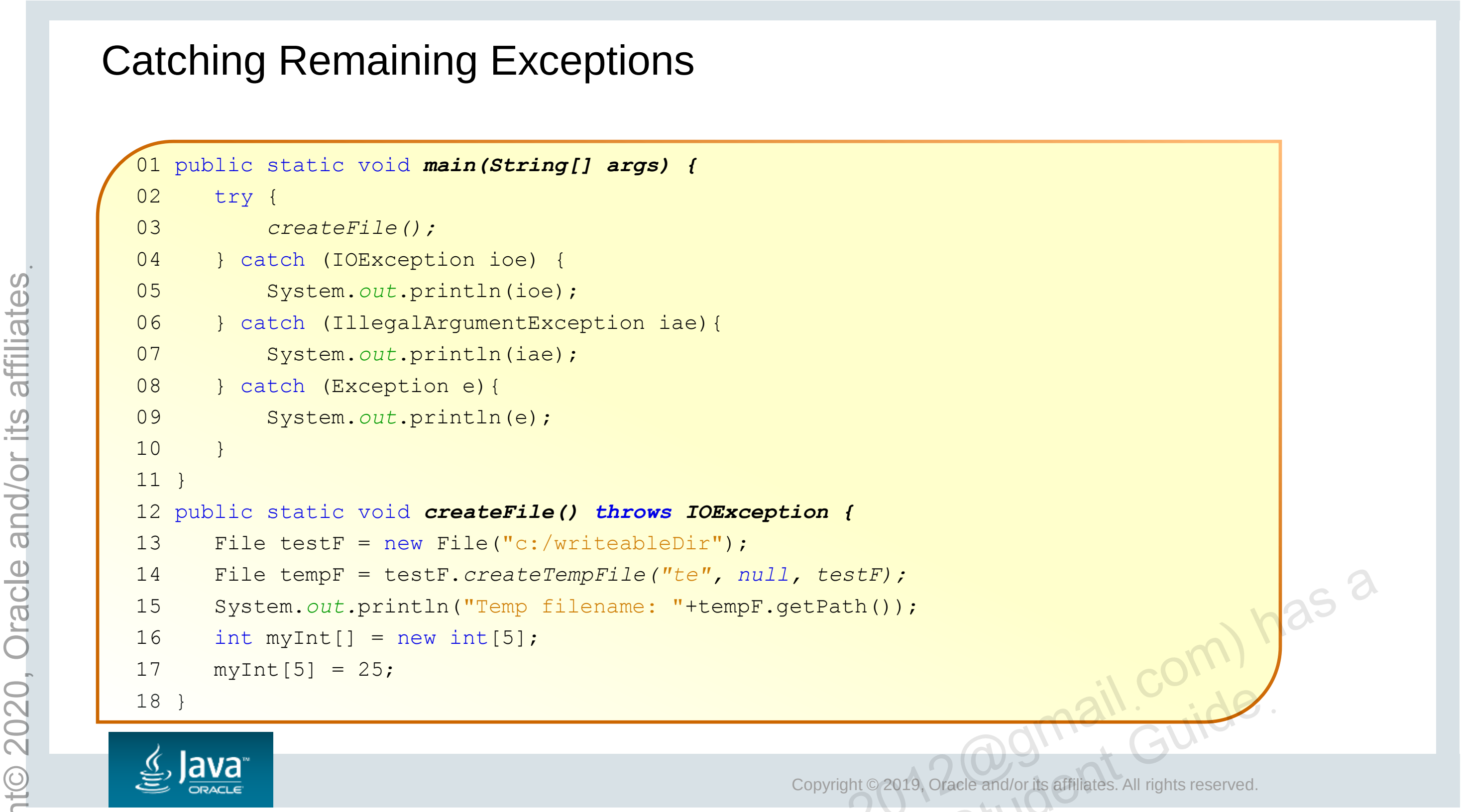
With the first argument of the `createTempFile` method set to `"te"` (fewer than three characters), the output of this code is:

```
java.lang.IllegalArgumentException: Prefix string too short
```

However, if the argument is set to `"temp"`, the output is now:

```
Temp filename is /Users/kenny/writeableDir/temp938006797831220170.tmp
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
... < some code omitted > ...
```

Now the temporary file is being created, but there is still another argument being thrown by the `createFile` method. And because `ArrayIndexOutOfBoundsException` is a `RuntimeException`, it is automatically thrown all the way out to the console.



Catching Remaining Exceptions

```
public static void main(String[] args) {
   try {
       createFile();
   } catch (IOException ioe) {
       System.out.println(ioe);
   } catch (IllegalArgumentException iae){
       System.out.println(iae);
   } catch (Exception e){
       System.out.println(e);
   }
}
public static void createFile() throws IOException {
   File testF = new File("c:/writeableDir");
   File tempF = testF.createTempFile("te", null, testF);
   System.out.println("Temp filename: "+tempF.getPath());
   int myInt[] = new int[5];
   myInt[5] = 25;
}
```

The example in the slide shows an additional `catch` clause to catch all the remaining exceptions.

For the example code, the output of this code is:

```
Temp filename is /Users/kenny/writeableDir/temp7999507294858924682.tmp
java.lang.ArrayIndexOutOfBoundsException: 5
```

Finally, the `catch` exception clause can be added to catch any additional exceptions.

Summary

In this lesson, you should have learned how to:

- Describe the different kinds of errors that can occur and how they are handled in Java
- Describe what exceptions are used for in Java
- Determine what exceptions are thrown for any foundation class
- Write code to handle an exception thrown by the method of a foundation class

Practices Overview

- 14-1: Adding exception handling