**Learn the difference between errors, checked and unchecked exceptions**

Sometimes your program will encounter problems that prevent it from performing the way that it should and when this happens Java represents the issue in one of three ways and depending on how, you the programmer, is expected to handle the issue and this is often related to the severity of the issue. For example, if the program runs into a problem that is so serious that there is no way to recover and no attempt should be made to recover, it is represented as an **Error** object. This might be caused by a hardware issue, memory failure and the like. In these unlikely scenarios the best course of action is to quite.

If the program runs into a problem that is unexpected but not fatal then this issue is represented by an instance of the **RuntimeException** class (or a subclass). These none fatal problems are often caused by careless programming and, in theory, should not happen because you are a careful programmer. Nevertheless the Java language recognises that developers are not perfect and that we make mistakes. So provides the **RuntimeException** type to represent them. These exceptions are called unchecked exceptions. These exceptions might be caused by the code trying to access an array element that does not exist or a null is used instead of an object.

And finally, if a program runs into an issue that could be reasonable expected to occur, such as a file does not exist or the internet connection is down, then these issue are represented as an Exception (or subclass) and are called checked exceptions. The developer is expected to have anticipated this problem and have written code that handles the situation so that the program continues to run without interruption.

## Handling exceptions

As a programmer you are required to handle only checked exceptions and provide alternative code to run if a check exception occurs. The objective of this alternative code is to ensure that the application continues to execute in a way that is useful to the client.

### Exercises:

1. What is the difference between a checked exception and an unchecked exception?
   1. Checked exceptions are the same as unchecked exceptions
   2. Checked exceptions must be caught but not unchecked exceptions
   3. Checked exceptions are only thrown by the developer and unchecked exceptions are only thrown by the JVM
   4. Checked exceptions extend the Exception class and unchecked exceptions extend RuntimeException class
   5. Unchecked exceptions must be caught but not unchecked exceptions

Answer: b, d

1. Which of the following statements are false?
   1. Only checked exceptions are declared
   2. Only unchecked exceptions are declared
   3. The Throwable class is not at the top of the Exception class hierarchy
   4. You can handle all types of exceptions
   5. Errors must be declared and handled

Answer: b, e

# Use try/catch block and learn how exceptions affect execution flow

Exceptions are handled with the try catch structure we saw in the previous subsection. Let’s take a closer look at how to use this structure.

## Add a finally block

The try/catch structure can be enhanced by adding a finally clause that will always be execute regardless of whether an exception is thrown.

## Valid and invalid try catch expressions

A try/catch statement must always have at least one catch or a finally clause. It cannot be a try statement on its own. The following structures are valid.

**try {  
 // code that might throws a checked exception  
} catch (Exception ex) {  
 // alternative code to run if an exception is thrown  
}**

**try {  
 // code that might throws a checked exception  
} finally {  
 // code to always execute  
}**

The try/catch structure can support multiple catch clauses.

**try {  
 // code that might throws a checked exception  
} catch (Exception1 ex) {  
 // alternative code to run if an exception is thrown  
} catch (Exception2 ex) {  
 // alternative code to run if an exception is thrown  
} catch (Exception3 ex) {  
 // alternative code to run if an exception is thrown  
}**

But you must pay attention to the type and class hierarchy of the **Exception**s in the catch clauses and the order in which they appear. We will look at that in more detail in a later subsection.

The catch clause must always appear before the finally clause and there can be only one finally clause. The following example is **not** valid.

**try {  
 // code that might throws a checked exception  
} finally {  
 // code to always execute  
} catch (Exception ex) {  
 // alternative code to run if an exception is thrown  
}**

Ensure you know how to correctly construct a try/catch as the examiners love to try to catch you out with constructs that look valid but actually are invalid.

### Exercises:

1. Which of the following try/catch/finally structures are valid

a.

try {  
 // code block  
} catch (Exception ex) {  
 // code block  
}

b.

try {  
 // code block  
} finally {  
 // code block  
}

c.

try {  
 // code block  
} finally {  
 // code block  
} catch (Exception ex) {  
 // code block  
}

d.

try {  
 // code block  
} catch (Exception1 ex) {  
 // code block  
} catch (Exception2 ex) {  
 // code block  
}

e.

try {  
 // code block  
} finally(Exception ex) {  
 // code block  
}

Answer: a, b, d

1. When is the finally clause called?
   1. Just after the catch clause
   2. Only if the catch clause is executed
   3. Just before the method returns to the calling method
   4. Before evaluating the return statement
   5. After evaluating the return statement

Answer: c, e

1. When do you have to use a finally clause?
   1. Only when you use a catch clause
   2. Only when no catch clause is used
   3. You never have to use a finally clause
   4. Only when you don’t use System.exit(0)
   5. When more than one catch clause is used

Answer: c

# Understand the benefits of exception handling

Exception handling allows the programmer to develop code that rescues the application from problems that it might encounter under normal circumstances.

### Exercises:

1. Which of the following statements are true?
   1. Unchecked exceptions happen because the programmer wrote poor quality code
   2. Exception handling covers up problems caused by bad programming practices
   3. Checked exceptions are thrown when recoverable problems occur
   4. Exception handling allows a programmer to offer alternative code to run
   5. Exception handling reflect the real world experiences of programmes

Answer: a, c, d, e

# Develop code that throws an exception

The Java APIs that you will be using as part of your everyday code declare many exceptions which you can reuse for your own purposes. You can also create your own collection of exceptions specifically for your application’s needs.

### Exercises:

1. Which of the following statements are true?
   1. All checked exceptions must be either declared or thrown
   2. Custom exceptions must extend the Trowable class
   3. A parent exception can catch a child exception
   4. Custom checked exceptions must extend the Exception class
   5. A child exception can catch a parent exception

Answer: a, c, d

1. What is the output of the following code snippet?

private static int calculateScore() {  
  
 int score = 1;  
 try {  
 score--;  
 } catch (Exception ex) {  
 score = score + 1;  
 return score;  
 } finally {  
 --score;  
 }

return score;  
  
}

* 1. -1
  2. 1
  3. 0
  4. 1
  5. 2

Answer: a

1. What is the output of the following code snippet?

private static int calculateScore() {  
  
 int score = 1;  
 try {  
 score--;  
 throw new Exception();  
 } catch (Exception ex) {  
 score = score + 1;  
 return score;  
 } finally {  
 --score;  
 }  
  
}

* 1. -1
  2. 0
  3. 1
  4. -2
  5. Does not compile

Answer : c

# Learn common exceptions

There are a number of common exceptions that you need to recognise and you need to know which type they are: errors, runtime exceptions or checked exceptions and whether they are thrown by the developer or the JVM.

## Errors

Errors are thrown by the JVM and extend the **Error** class. You should not declare or handle these errors and include the following:

**StackOverflowError**: Thrown when a method calls itself more times than space on the stack.

**Example**

public void process(int i){

process(i++); // exception at runtime.

}

**ExceptionInInitializerError**: Thrown when a static initializer code block doesn’t handle an exception.

**Example**

public class Car {

int speed = 0;

static {

speed = 10/0;

// throws DivideByZeroException but this is wrapped into a

// ExceptionInInitializationError and thrown outside.

}

}

**NoClassDefFoundError**: Thrown when no class of the type is found on the class path

Normally their class names end in Error.

## Runtime exceptions

These exception are thrown by either the JVM or the developer and don’t need to be declared or handled.

They include the following that are thrown by the JVM:

**NullPointerException**: Thrown when a null is used when an object is expected

**Example**

String name = null;

System.out.println(name.length()); // exception

**ArrayIndexOutOfBoundsException**: Thrown when an invalid index is accessed

**Example**

int[] numbers = new int[]{ 1, 2, 3}; // numbers is of length 3.

System.out.println(numbers [3]); //exception

**ArithmeticException**: Thrown when a dived by zero is attempted

**ClassCastException**: Thrown when casting incompatible types

**Example**

Object name = “asdf”;

StringBuffer sbName = (StringBuffer) name; // exception at runtime because s is referring to a String.

Those that are thrown by the developer include the following:

**NumberFormatException**: Thrown when a String cannot be converted to a number because the String is not a number.

**Example**

Integer.parseInt(“five”);

**IllegalArgumentException**: Thrown when an illegal argument has been passed to a method.

**Example**

public void process(int keytype) {

if(keytype != 1) throw new IllegalArgumentException();

}

## Checked exceptions

Checked exceptions must be declared and handled and extend from **Exception**. They are always thrown by the developer.

They include:

**IOException**: thrown when there is an issue reading or writing to a given output stream.

**FileNotFoundException**: Thrown when code cannot file a file with the given name.

### Exercise:

1. Which of the following exceptions are thrown by the JVM?
   1. NullPointerException
   2. ArrayIndexOutOfBoundsException
   3. ArithmeticException
   4. ClassCastException
   5. NumberFormatException

Answer: a, b, c, d

1. Which of the following exceptions are thrown by the developer?
   1. FileNotFoundException
   2. ExceptionInInitializerError
   3. IOException
   4. NumberFormatException
   5. StackOverflowError

Answer: a, c, d

1. What exception is thrown by the following code?

String[] names = new String[]{ “John”, ”Amid”, ”Jose”, ”Li” };   
System.out.println(names [4]);

* 1. NumberFormatException
  2. ExceptionInInitializerError
  3. ArrayIndexOutOfBoundsException
  4. NullPointerException
  5. InvalidArrayConstructionException

Answer: c

1. What exception is thrown by the following code?

public void callMe(){  
 callMe();   
}

* 1. NullPointerException
  2. InvalidCallException
  3. CallOverflowException
  4. StackOverflowError
  5. CodeInitialisationError

Answer: d