# Recall that the advantage of Java’s static typing is to detect bugs at compile time

For example

* Syntax errors
* Wrong argument types
* Wrong return types

|  |
| --- |
| **public int** testMe( String a){  **return** “”; // wrong return type  }  int a = testMe(1.0); // wrong argument type |

Be aware that there are errors that are not checked!

|  |
| --- |
| **public int** add( int a, int b){  **return** a+b;  } |

The method above returns the mathematically correct answer for all values of a and b. --> False

If a or b is too large, you will get integer overflow.

In this lesson, we focus on Exceptions, which are objects thrown due to errors during **runtime**.

# Exceptions are objects that are thrown when an error is encountered during runtime.

A **runtime error** occurs when the JVM detects an operation after compilation that is **impossible** to carry out e.g.

* The file you are accessing cannot be found
* When you want to access a URL, the internet connection is down

Note also that there are also **runtime error**s which are logic errors

* The index is too large for the array
* The object is a null value
* You try to divide by integer zero

When a **runtime error** occurs, objects called **Exceptions** are **thrown**, and the program terminates, often in an unexpected way.

It is possible for the programmer to have control over such terminations by using a **try-catch block**.

# Example 41 – You put code where exceptions are likely to occur in a try-catch block

In the following example, a divide-by-zero could occur at the line **return x/y**, and if y is 0, the JVM throws an **ArithmeticException** object. You can write a **try-catch block** to handle this error.

**In the try-block, once an Exception is thrown, execution is diverted to the catch block.** The rest of the lines in the try-block are not executed.

**ArithmeticException** is known as an **Unchecked Exception**. We will have more to say on this later.

Notice that you have the option to NOT use a try-catch block to handle the error. This is left to you as an exercise.

|  |
| --- |
| public class TestQuotient {   public static void main(String[] args) {     try{  System.*out*.println(*divide1*(1,0));  System.*out*.println("End Of Calculation");  }catch( ArithmeticException ex){  System.*out*.println("Divide By Zero");  ex.printStackTrace();  }  }  */\*\* TODO 1 Handle this method in main() using try/catch block \*/* public static int divide1( int x, int y){  return x/y;  }  */\*\* TODO 2 Use if/else to handle this \*/* public static int divide2( int x, int y){  return x/y;  } } |

# Example 42 – If you are writing a method to open and read from a text file, the File() object is known to throw a FileNotFoundException. You have two options.

The programmer cannot control whether the file is on the system or not. Thus Exceptions and a try-catch block are an ideal way to handle the scenario.

You are writing a method openFile() to open, read and return the contents of a file in a single string. The File() constructor throws a FileNotFoundException.

You have two design choices that you can make:

* **Handle the exception in openFile() itself** using a try-catch block
* **Delegate the task of handling the Exception to code that calls openFile()**. You then need to **declare** that openFile() throws **FileNotFoundException** object. This is shown below.

|  |
| --- |
| public class TestOpenReadFile {   public static void main(String[] args) { }  public static String openFile(String fileName) throws FileNotFoundException {   **File file = new File(fileName); //**   Scanner scanner = new Scanner(file);  String out = "";  while( scanner.hasNext()){  String s = scanner.nextLine();  out = out + s;  }  return out; }  } |

# There are two kinds of Exceptions, Unchecked Exceptions and Checked Exceptions. The JVM provides guarantees for Checked Exceptions.

A summary of the Exceptions hierarchy is shown below. Image is taken from Liang, “Introduction to Java Programming”, 10th edition.

A diagram of a program

Description automatically generated

**Unchecked Exceptions:** **RuntimeException** and all its subclasses are known as **Unchecked Exceptions**. The compiler does not enforce that you declare in method signature and put code in try-catch block. These are usually thrown due to logic errors, is detected during testing, and can handled by modifying your code.

à Thus, for logic errors, you should usually handle them using if/else and not use try-catch blocks.

**Checked Exceptions:** All other Exceptions are known as Unchecked Exceptions. Compiler enforces the “handle or declare rule”: (You saw this in Example 42)

* **If your method executes code that throws a checked exception**, you must handle that exception in your method or throw the exception (see next rule)
* **If your method throws a checked exception**, you must declare in method signature

# Clicker Question – Given the following code, what will be printed on the screen? B.socrative.com 593583

In the following code, a **URISyntaxException** is thrown because there is an illegal character (^) in the string passed to it. This exception is a checked exception and is a direct subclass of **Exception**.   
Own reading: [The difference between URI and URL.](https://ae.godaddy.com/blog/uri-vs-url/)

|  |
| --- |
| import java.net.URI; import java.net.URISyntaxException;  public class TestExceptions {   public static void main(String[] args) {   try{  System.*out*.println("A");  *getURIfromURL*("http://finance.yahoo.com/q/h?s=^IXIC");  System.*out*.println("B");  }catch( **URISyntaxException** ex){  System.*out*.println("C");  }catch( **Exception** ex){  System.*out*.println("D");  }  }   public static void getURIfromURL(String myURL) throws URISyntaxException {   URI uri = new URI(myURL); *//may throw a URISyntaxException* System.*out*.println("E");  } } |

Adapted from <https://stackoverflow.com/questions/749709/how-to-deal-with-the-urisyntaxexception>

# Example 43 - You can catch more than one exception, and you must put the most specific exception first

You can instantiate **a BigDecimal** object using a string, and if the string does not represent a valid number, then a NumberFormatException will be thrown eg.

**BigDecimal b = new BigDecimal (**"12g4"**);**

A **divide()** method throws a **IllegalArgumentException** if the RoundingMode is an invalid value.

You can write more than one catch block to handle each of these exceptions, but you must be aware of the inheritance hierarchy.

|  |
| --- |
| public class TestBigDecimal {   public static void main(String[] args) {  System.*out*.println(*divide*("2abc","12.0"));  }   public static String divide (String s1, String s2){  */\*\* The constructors throw a NumberFormatException if the string is not a valid number \*/* BigDecimal b1 = new BigDecimal(s1);  BigDecimal b2 = new BigDecimal(s2);  */\*\* using the deprecated overloaded version of divide()  \* public BigDecimal divide(BigDecimal divisor, int scale, int roundingMode);  \* It throws NumberFormatException if the Rounding Mode is not a valid number  \*/* BigDecimal b3 = b1.divide(b2,3, 20);  return b3.toString();  }  } |

# Example 44 – You can write your methods to throw Exceptions.

Coming back to the quotient example in Example 41, you could also decide to throw an exception if y = 0 is passed to the method.

Notice that the keyword used here is **throw**, together with **new** and the constructor of an **Exception** object. You can also pass an message to the constructor.

The **throws** keyword is used at the method signature only.

if( y == 0) **throw** **new** IllegalArgumentException("Message");

In the catch block, this message can be retrieved as a string using the following methods

|  |
| --- |
| try{  *// code* }catch( Exception ex){  ex.printStackTrace(); // get all the error messages  String s = ex.getMessage(); // get just “Message” } |

# Example 45 – Reading data from a URL is an excellent example of where exceptions are necessary

If you are accessing a webpage to get data, many things could go wrong:

* There could be no internet connection
* There could be a typo in the URL

In the example below, a webpage can be accessed by

* Instantiating a **URL** object with the address
* Calling the url.openStream() method to get an **InputStream** object which provides a stream of data
* This stream data is passed to a Scanner object to read this line-by-line

The URL constructor and the openStream() method both throw exceptions. Adjust the method signature accordingly.

|  |
| --- |
| public class TestDownloadURL {   public static void main(String[] args) {  }   */\*\* Remove the throws Exception and replace it with the specific exception \*/* public static String queryURL(String urlString) throws Exception {  *// Create a URL object* URL url = new URL(urlString);  *// read the URL stream into the Scanner object* Scanner input = new Scanner(url.openStream());  String out = "";  while (input.hasNext()){  out = out + input.nextLine() + "\n";  }  return out;  } } |

# Summary

Exceptions can be thrown, declared and handled. This is what happens.

