ALGONQUIN COLLEGE

MODULE 08: EXCEPTIONS AND EXCEPTION HANDLING

Professor : Dave Houtman

Office: T323

Office Hrs: Wednesday 14:15 – 15:30

Wednesday (after lecture*)

* confirm beforehand

Email: houtmad@algonquincollege.com

8.0 Exceptions and Exception Handling

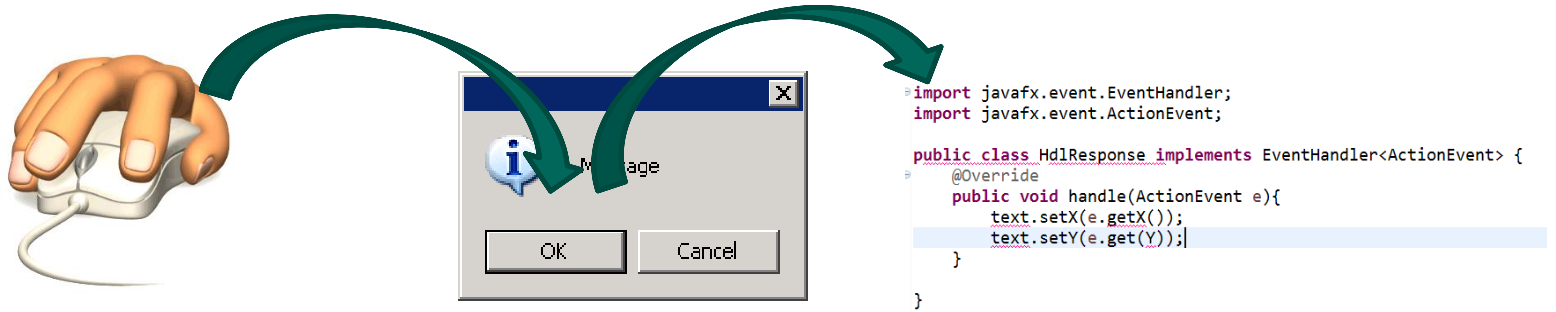
- ***Exceptions*** are thrown by methods when unresolvable problems are encountered during execution
- ***Exception Handling*** (also, less frequently, still referred to as ***Error Handling***) is the subject of dealing with the different kinds of exceptions that can be generated during normal program execution
- If the exception is not handled, the program will terminate abnormally, and generate an error like:

```
Exception in thread "main"
java.lang.ArithmeticException: / by zero
   at Calculator.division(Calculator.java:49)
   at Calculator.<init>(Calculator.java:32)
   at Calculator.main(Calculator.java:7)
```

ALGONQUIN COLLEGE

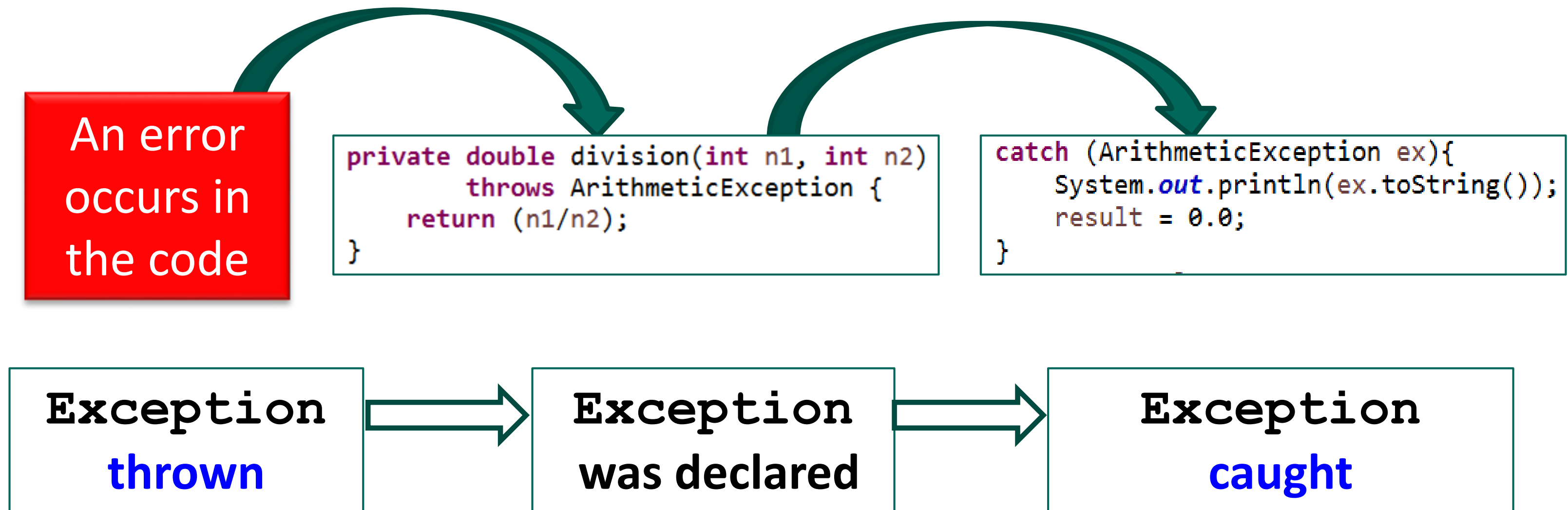
8.0 Exceptions and Exception Handling

Exceptions and Exception Handling are not unlike Events and Event Handling, seen in Module 04. Recall that there are three parts to EventHandling:



8.0 Exceptions and Exception Handling

We may likewise consider that there are three parts to Exceptions and Exception Handling:



As with event handling, the initial 'event'—the error that occurred—occurs outside of normal program execution (although it can be generated deliberately in code), and its handling depends upon (1) making sure we declared the kind of exception expected in advance, and (2) providing a means to deal with it

8.0 Exceptions and Exception Handling

As with event handling, we need to define the exception handling code prior to the exception being thrown. Similarly, the code should be declared in the opposite order to its actual execution so that the correct program structures are in place first. Java's exception-handling model is based on the following three operations:

1. **Provide a try-catch block**. You typically supply a ***try – catch*** block that traps exceptions and deals with them. If you fail to catch the right *kind* of exception, then the JVM will do it for you, and you'll see the usual error message in red on your screen.

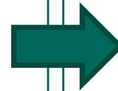
2. **Declare an exception**. The **throws** keyword is used to the declare the kind of exception that could occur. Despite the terminology, ‘declaring an exception’ really means “Declare the (kind of) exception (that the code could generate)”; it should not be confused with declaring a custom exception handler, which we’ll discuss later in this module.

3. **Throw the exception**. You can do this explicitly using the ***throw*** keyword. More often, it happens automatically when your code triggers an error during execution.

8.1 The `try-catch` block

In actual practice, not all three of these operations need to be declared in advance. Depending on the type of error and the structure of the code, only the try-catch block may need to be declared. When used in this fashion, the format of a try-catch statement approximates that of an if-else statement…

```
try{

    // try code here

}
catch {

    // catch exceptions

}
```



```
if (codeIsGoingToWorkRight){

    // do the code

}
else {

    // deal with the
    // problems
}
```

…except of course, that an `if` statement can't know in advance if the code is going to work correctly!

ALGONQUIN COLLEGE

8.1 The `try-catch` block

Consider the following code fragment*, which detects a mismatch in the type of input. Only the try-catch is required to deal with the exception correctly:

```
boolean badInput = true;    // assume default bad input value
do{
   try{
      System.out.print("Please enter an integer value: ")
      int input = input.nextInt();
      System.out.println ("The number entered is: " + input);
      badInput = false;  // no error; input was good
   }
   catch (InputTypeMismatchException ex){
      System.out.println("Bad input: an integer is required");
      input.nextLine();
   }
} while (badInput);
```

*Taken from: Liang D. Y. (2014). *Introduction to Java Programming, Comprehensive Version, 10th Ed*. Toronto, ON: Pearson. pg. 454, with modifications

8.1 The `try-catch` block

Here, if a non-`int` value is input at the prompt, an `InputTypeMismatchException` is generated, causing execution to jump out of the `try` block and into the `catch`.

```
boolean badInput = true;    // assume bad input value
do{
    try{
        System.out.print("Please enter an integer value: ")
        int input = input.nextInt();
        System.out.println ("The number entered is: " + input);
        badInput = false;  // no error; input was good
    }
    catch (InputTypeMismatchException ex){
        System.out.println("Bad input: an integer is required");
        input.nextLine();   // need this to clean up input stream
    }
} while (badInput);
```

ALGONQUIN COLLEGE

8.1 The `try-catch` block

Note that, following execution of the `catch` block, program execution does *not* return to the statement after the one that triggered the exception, but it continues on as usual:

```
boolean badInput = true;    // assume bad input value
do{
   try{
      System.out.print("Please enter an integer value: ")
      int input = input.nextInt();
      System.out.println ("The number entered is: " + input);
      badInput = false;  // no error; input was good
   }
   catch (InputTypeMismatchException ex){
      System.out.println("Bad input: an integer is required");
      input.nextLine();   // need this to clean up input stream
   }
} while (badInput);
```

8.1 The `try-catch` block

Since `badInput` is still `true`, we will continue looping, and return to the start of the `try` again for a second attempt.

```
boolean badInput = true;    // assume bad input value
do{
   try{
      System.out.print("Please enter an integer value: ")
      int input = input.nextInt();
      System.out.println ("The number entered is: " + input);
      badInput = false;  // no error; input was good
   }
   catch (InputTypeMismatchException ex){
      System.out.println("Bad input: an integer is required");
      input.nextLine();   // need this to clean up input stream
   }
  while (badInput);
```

8.1 The `try-catch` block

But if an integer value *is* correctly entered, execution continues and jumps over the `catch` block. And since `badInput` is now `false`, we jump out of the `while` loop.

```
boolean badInput = true;    // assume bad input value
do{
   try{
      System.out.print("Please enter an integer value: ")
      int input = input.nextInt();
      System.out.println ("The number entered is: " + input);
      badInput = false;  // no error; input was good
   }
   catch (InputTypeMismatchException ex){
      System.out.println("Bad input: an integer is required");
      input.nextLine();   // need this to clean up input stream
   }
} while (badInput);
```

ALGONQUIN COLLEGE

8.1 The `try-catch` block

The key benefit of this approach is that we can *separate the detection of the error from the actual handling of it*.

For example, if we use an `if…else` block to check for an error condition before it occurs, we are generally forced to deal with the problem in the `if…else` block itself. The try-catch allows us to separate the code from the potential problems that generate exceptions.

As an additional benefit, we can specify the *type* of exception, and handle different exceptions in an appropriate fashion.

ALGONQUIN COLLEGE

8.1 The `try-catch` block

Frequently, an error occurs in one method but must be dealt with elsewhere, in another method. Consider the following example, a simple calculator program:

```
import java.util.Scanner;

public class Calculator {

    // exit determines whether or not to leave the program
    private boolean exit = false;

    public static void main(String[] args) {
        new Calculator();     // contains all the code
    }

    ...
```

8.1 The `try-catch` block

Inside the `Calculator` constructor, we offer the user some choices, prompt for an operation, and then execute the choice:

```
public Calculator(){
    Scanner in = new Scanner(System.in);
    do{
        System.out.print("Enter first number: ");
        int num1 = in.nextInt();

        System.out.print("Enter the second number: ");
        int num2 = in.nextInt();

        in.nextLine();
        System.out.print("Enter +, -, *, /, or %: ");
        char choice = in.nextLine().charAt(0);

        System.out.println("Result is " +
                doCalculation(choice, num1, num2));
    } while (!exit);
    in.close();
}...
```

ALGONQUIN COLLEGE

8.1 The `try-catch` block

Inside the `doCalculation()` method, we use `switch` to determine which operation is to be performed:

```
...
private double doCalculation(char opn, int num1, int num2){

   double result = 0.0;

   switch(opn){
      case '*':
         result = multiplication(num1, num2);
      break;

      case '/':
         result = division(num1, num2);
      break;

      //… plus additional case statements for '+', '-', etc.

  }
  return result;
}
```

ALGONQUIN COLLEGE

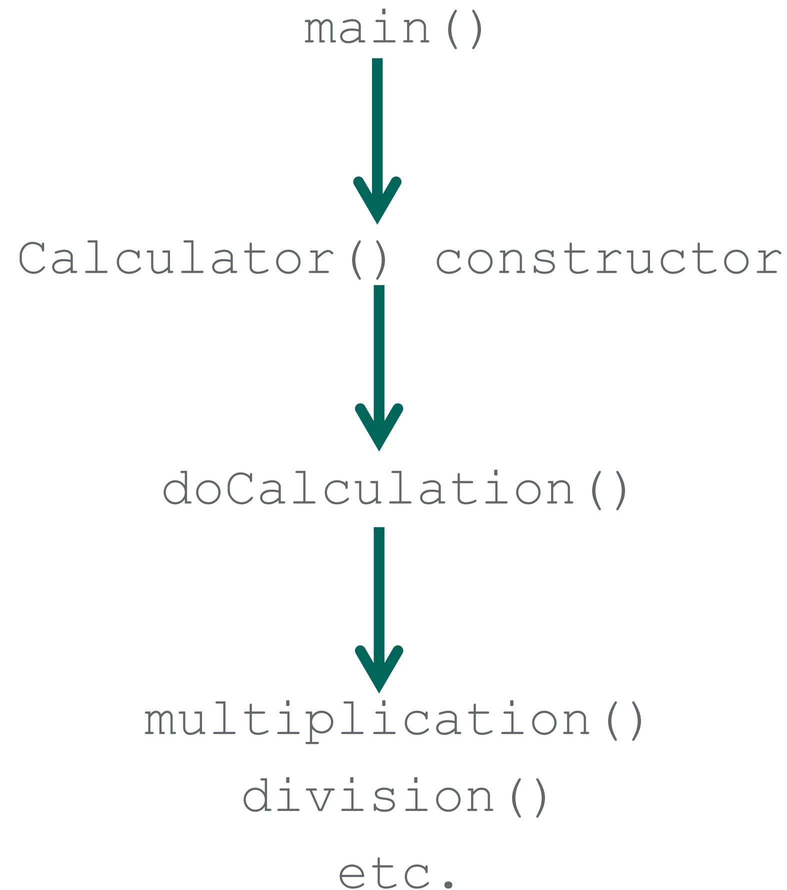
8.1 The `try-catch` block

Finally, we list each of the three functions to be performed

```
    ...
    private double multiplication(int n1, int n2){
        return (n1*n2);
    }

    private double division(int n1, int n2){
        return (n1/n2);
    }

    //… etc.

    private double modulus(int n1, int n2){
        return (n1%n2);
    }

}   // end of class
```

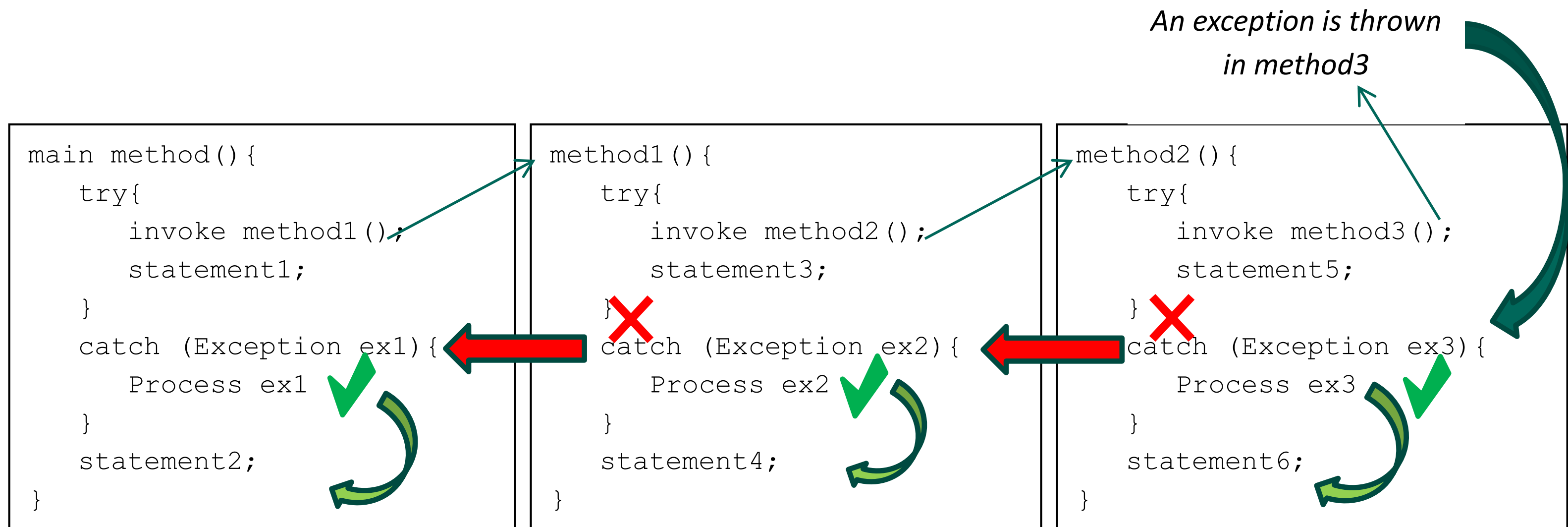
8.1 The `try-catch` block

So there are four levels of nested methods in this program:



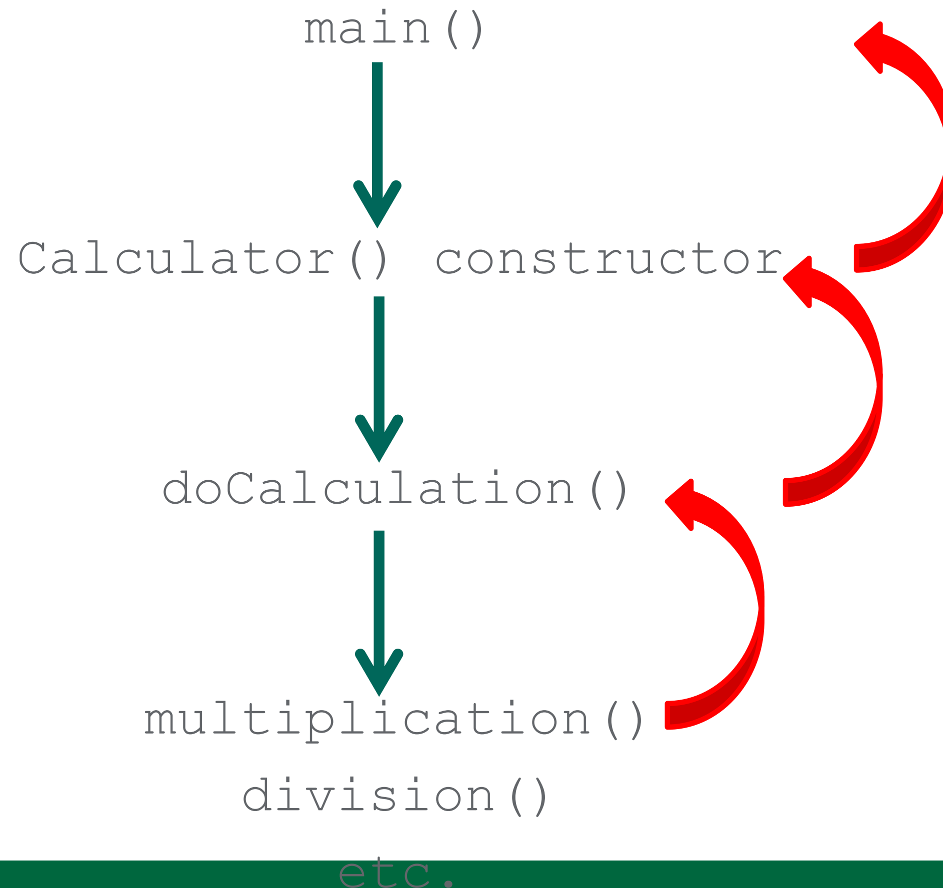
8.1 The try-catch block

Different types of errors produce different types of exceptions. When an exception is thrown, it must be caught by the appropriate exception handler. If this handler cannot be found in one method's catch statements, responsibility for catching the exception works its way progressively back through the original method call stack.



8.1 The `try-catch` block

So if we don't deal with the `ArithmeticException` triggered when we attempt to divide by zero in the `division()` method, the exception will propagate back upward until it finds an appropriate handler:



8.1 The `try-catch` block

The division operation is, as originally written, clearly a source of potential problems. What happens if the user enters a zero value for the denominator?

We can choose to capture this exception at any of four different levels:

1. Inside main:

```
public static void main(String[] args) {
    try{
        new Calculator();     // contains all the code
    } catch (ArithmeticException ex){…}
    …
```

2. Inside the constructor

```
    try{
       System.out.println("Result is " +
          doCalculation(choice, num1, num2));
    } catch (ArithmeticException ex){…}
```

8.1 The `try-catch` block

3. Inside `doCalculations()`:

```
case '/':
    try{
    result = division(num1, num2);
    } catch (ArithmeticException ex){…}
break;
```

4. Inside `division()`:

```
private double division(int n1, int n2){
    try{
        return (n1/n2);
    } catch (ArithmeticException ex){…}
}
```

ALGONQUIN COLLEGE

8.1 The `try-catch` block

Note that the higher up we capture the exception, the less specific the error message will be. For example, if we capture an error in `main()`, how do we know which of several possible exceptions actually triggered the error?

In general, you'll want to capture errors as close to the source as possible. This has several benefits, including:

1. Error messages can be tailored to specific causes;
2. Debugging code is easier when you generally know where to start looking;
3. Uncaught errors at low levels can trigger other, more problematic errors at higher levels, leading to an error cascade whose source is hard to understand.

8.1 The `try-catch` block

However, attempting to catch an exception too close to the source of an error can be problematic as well. For example, if you attempt to wrap the actual division calculation in the try-catch, this happens:

```
65    private double division(int n1, int n2) {
66  This method must return a result of type double
67          try{
68              return (double)(n1/n2);
69          }catch (Exception ex){
70              System.out.println("Can't divide by 0");
71          }
72    }
```

…the division method is still expected to return a `double` data type, but the `catch` effectively blocks it. We could wrap it all in a `while()` loop, but this seems like a lot of work in which all we really wish to do is to detect the situation in which `n2==0`.

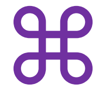
A better solution will be seen shortly. For now, we'll deal with this exception one level up, inside `doCalculations()`.

8.2 Polymorphism in Exception Handling

A more general problem is this: you can't anticipate every possible error (or combinations of errors) that could possibly occur. And the more you try, the more cluttered the code becomes with try-catch blocks. So there needs to be some trade-off between the number of error messages and their specificity.

This partially explains why, when you debug your Java code, you're often forced to deal with error messages that are often vague, even while they direct you to a particular line of code. It's easier to say *where* an error occurs than to indicate *what*, precisely, the error actually *is*.

8.2 Polymorphism in Exception Handling



Java contains a rich exception class that extends from the `Throwable` object:

Object
Throwable
Exception
Error
IOException
RuntimeException
etc.
LinkageError
etc.
InputType Mismatch Exception
NullPointerException
IndexOutOfBoundsException
ArithmeticException
InputTypeMismatchException

8.2 Polymorphism in Exception Handling

`Throwable` exception classes are of three main types:

1. **System errors** are represented by the `Error` class and are thrown by the JVM. They describe internal errors, such as problems with the JVM, as when a specified class is not available in Eclipse when it should be. These exceptions are usually not of concern to us. (Thankfully; they can be tough to debug.)

2. **`Exceptions`** describe general errors caused by the program and by external circumstances

3. **Runtime exceptions** are represented in the `RuntimeException` class. They describe programming errors that always occur during runtime, such as bad casting, bad input (such as the aforementioned type mismatch), accessing out-of-bounds array indices, and arithmetic errors (such as divide-by-zero errors). This is the largest class of errors, numbering perhaps a couple hundred in total.

8.2 Polymorphism in Exception Handling

Multiple exceptions may be dealt with in the same try-catch block. For example, if we wish to check for both a type mismatch and an arithmetic exception, we can employ a single `try` statement, along with separate `catch` statements:

```
private double doCalculation(char opn, int num1, int num2){
   boolean badCalculation = true; double result = 0.0;
   do {
     try{
       switch(opn){
        // case '*': etc.
       } badCalculation = false;
     }
     catch (ArithmeticException ex){
        System.out.println("Bad value input; re-enter input");
     }
     catch (InputTypeMismatchException ex){
        System.out.println("Bad operation input; try again");
     }
   while (badCalculation);
   return result;
}
```

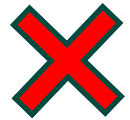
Switch block for calculations goes here

ALGONQUIN COLLEGE

8.2 Polymorphism in Exception Handling

Polymorphism plays a role in determining the order in which exceptions are caught. Assume that, in addition to the two specific exceptions mentioned above, we wish to display a message indicating general exceptions, by catching instances of the `RuntimeException` class:

```
…
catch (RuntimeException ex){
   System.out.println("Unknown general error thrown");
}
catch (ArithmeticException ex){
   System.out.println("Bad value input; re-enter input");
}
catch (InputTypeMismatchException ex){
   System.out.println("Bad operation input; try again");
}
```



Superclass precedes subclass

8.2 Polymorphism in Exception Handling

Consider the situation in which an `ArithmeticException` is thrown by one of the calculations. Since an `ArithmeticException` *is a* `RuntimeException`, the `RuntimeException` handler catches the error before `ArithemticException` gets to see it...

ArithmeticException

```
…
catch (RuntimeException ex){
    System.out.println("Unknown general error thrown");
}
catch (ArithmeticException ex){
    System.out.println("Bad value input; re-enter input");
}
catch (InputTypeMismatchException ex){
    System.out.println("Bad operation input; try again");
}
```

Superclass precedes subclass

ALGONQUIN COLLEGE

8.2 Polymorphism in Exception Handling

Indeed, the compiler warn against code in which a superclass Exception precedes a subclass Exception in the catch block, and will insist that you change the order:

```
…
catch (ArithmeticException ex){
   System.out.println("Bad value input; re-enter input");
}
catch (InputTypeMismatchException ex){
   System.out.println("Bad operation input; try again");
}
catch (RuntimeException ex){
   System.out.println("Unknown general error thrown");
}
```

8.2 Polymorphism in Exception Handling

We should also mention that, as of JDK 7, it is possible to deal with multiple catches on the same line using the | (OR) notation. For example, rather than use separate catches for each exception, we can combine exceptions instead:

```
…
catch (ArithmeticException | InputTypeMismatchException ex){
   System.out.println("Bad input; re-enter calculation");
}
```

But then, of course, there can only be one error message for two different kinds of errors.

8.3 Declaring exceptions with `throws`

In situations in which an error is about to occur, it is preferable to deliberately throw the exception in advance.   This is done by

1. Indicating that a method throws a particular kind of exception by adding the keyword **throws** to the method header, followed by the type of exception
2. Using the keyword **throw** to actually trigger the exception, followed by an instance of the thrown exception itself.

For example:

```
private double division(int n1, int n2)
                                throws ArithmeticException {
     if (n2==0) throw new ArithmeticException();
     return (n1/n2);
}
```

Note that this exception will be caught in `doCalcuations()` using the code shown in the previous slides.

8.3 Declaring exceptions with `throws`

Note that it is possible to declare multiple throws in the same method header. Simply separate each exception with commas, e.g.

```
private void someMethod()
        throws Exception1, Exception2, Exception3… {
    …
}
```

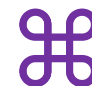
ALGONQUIN COLLEGE

8.3 Declaring exceptions with `throws`

In fact, all exceptions *should* be declared in this way. The fact that you don't *need* to is a reflection of the fact that the compiler does not insist that you check certain classes of exceptions.

The two classes of exception are called **Checked** and **Unchecked**. With checked exceptions, you *must* use `throws` in the method header; with unchecked, you are not required to do so.

8.4 Checked and Unchecked exceptions



Unchecked exceptions include Errors and runtime exceptions; `IOExceptions` are all of the checked variety. `ArithmeticExceptions` and `InputTypeMismatchException` are thus unchecked and so do not need to use the word throws in a method declaration.

8.4 Checked and Unchecked exceptions

Why the distinction? Unchecked exceptions are generally due to programming logic errors, and are unrecoverable: when they happen, the program is going fail.

For example, members of the `Error` class include such exotic exceptions as `LinkageError`, which is triggered when one class is dependent on another class, which has changed; and `VirtualMachineError`, which occurs when the JVM has run out of resources.

If either of these errors occur, then its probably because the user did something dangerous in code, or something very unexpected happened. The Java philosophy is: it's up to you to anticipate these problems and deal with them in advance, in try-catch blocks. So the compiler isn't going to force you to write a `throws`, since you are expected to anticipate these scenarios.

ALGONQUIN COLLEGE

8.4 Checked and Unchecked exceptions

There is considerable debate surrounding checked and unchecked exceptions, and on whether all exceptions should be checked, or all unchecked. There are some who feel that Java's insistence on designating some exceptions as checked and some as unchecked is a mistake.

See any of the following for details on this debate:

Unchecked Exceptions—the Controversy
https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html

Checked vs. Unchecked Exceptions: The Debate is Not Over
http://www.yegor256.com/2015/07/28/checked-vs-unchecked-exceptions.html

Java: Checked vs. Unchecked Exception Explanation
http://stackoverflow.com/questions/6115896/java-checked-vs-unchecked-exception-explanation

throwing runtime exception in Java application
http://softwareengineering.stackexchange.com/questions/184772/throwing-runtime-exception-in-java-application

8.4 Checked and Unchecked exceptions

If we attempt to throw a checked exception, the compiler alerts us to the fact that we *must* provide a mechanism to detect it. For example, if we attempt to throw an `IOException` (which is checked) inside a method then we are asked to add `throws`, or use the try-catch block around the code:

```
private double division(int n1, int n2)  {
    if (n2==0) throw new IOException();
    return (n1/n2);
}

private double multiplication(int n1,
    return (n1*n2);
}
```

Unhandled exception type IOException

2 quick fixes available:

- Add throws declaration
- Surround with try/catch

Press 'F2' for focus

ALGONQUIN COLLEGE

8.4 Checked and Unchecked exceptions

You have already encountered one such checked exception. The file input and output stream classes both throw a checked IO exception:

Constructor Detail

FileInputStream

```
public FileInputStream(String name)
                throws FileNotFoundException
```

Creates a `FileInputStream` by opening a connection to an actual file, the file named b connection.

Hence, when you instantiate a new File IO stream, you are required to handle any exceptions that may be thrown during its use:

```
FileInputStream fis = new FileInputStream(absPath);
ObjectInputStream ois = new ObjectInputStream(fis);
ois.close();
```

Unhandled exception type IOException

2 quick fixes available:

- Add throws declaration
- Surround with try/catch

Press 'F2' for focus

8.4 Checked and Unchecked exceptions

Note that, just because you add a `throws` to the method header doesn't mean that you've solved the problem of checked exceptions; you still need to catch the exception in a try-catch block.

So it's not as if you can do the first option—adding `throws`—and not the second. Placing the file IO calls directly inside a try-catch block satisfies the compiler's requirement that the checked exception has been handled correctly. But, as described above, wrapping the IO calls directly inside a try-catch may not be the best course of action, depending on the nature of the error. You may want to catch the error at a higher level, in which case you still need to use `throws` in the method header.

8.5 Try-with-resources

We add, as an aside, that starting with JDK 7 it is possible to enclose any resource opened inside a try-catch block using the **try-with-resources** syntax. It has the form:

```
try (declare and instantiate resources){
   // Use the new resources here
}
catch{
   // etc.
}
```

where resources are opened inside an initial set of parentheses () located just after the `try` statement, but before the first {.

Resources opened in this fashion are automatically closed when the try-catch is exited, saving the programmer responsibility for invoking the `close()` method. This is particularly important with file IO, since failure to close file handles and IO streams may lead to data loss.

8.5 Try-with-resources

For example, try-with-resources has the following form when used with file input streams:

```
try (FileInputStream fis = new FileInputStream(path);
    ObjectInputStream ois = new ObjectInputStream(fis);){
    for (int i = 0; i < numObjects; i++)
         ObjAr[i] = (ObjType) ois.readObject();
}
catch (IOException ex) {
     // handle file IO errors here
}

// fis.close() and ois.close() invoked automatically
```

8.6 File Stream IO using EOFException

While we're on the subject of file IO streams, now is a good time to address the subject of reading some unknown number of objects from a file *into* your program.

Recall that when we wish to read strings from a file into a program, we use `Scanner`, which has a `hasNext()` method:

```
File file = new File ("EmployeeInfo.inf");
Scanner input = new Scanner(file);
while (input.hasNext()){
   String firstName = input.nextLine();
   String lastName = input.nextLine();
   String Address = input.nextLine();
}
```

8.6 File Stream IO using EOFException

`hasNext()` scans the remaining data in the string file to see if the next token—such as a carriage return—exists or not. If not, then we have read in the last string, and the input loop exists.

With IO streams, no such equivalent method exists: *there is no such thing as* *`hasNextObject().`* Instead java uses an unconventional method to detect the end of the file: it deliberately throws an error.

ALGONQUIN COLLEGE

8.6 File Stream IO using EOFException

An `EOFException` is a IO Exception—and therefore checked—that is triggered when we attempt to read beyond the end of a file. A typical example of how this is used is:

```
int objctr = 0
try (FileInputStream fis = new FileInputStream(path);
    ObjectInputStream ois = new ObjectInputStream(fis);){
    while(true){  // loop until EOFException triggered
       objAr[objctr++] = (ObjType) ois.readObject();
    }
}
catch (EOFException ex){
 System.out.println("There were " + objctr + " objects"};
}
catch (IOException ex) {
 // handle file IO errors here
}
```

ALGONQUIN COLLEGE

8.6 File Stream IO using EOFException

This may seem odd, deliberately triggering an exception as a way of resolving a problem. And *it is*, somewhat. But in fact throwing an exception is sometimes the only way to resolve a particular problem.

Used responsibly, it is considered good programming practice; used incorrectly, it dirties the code with unnecessary try-catch blocks without actually solving any real underlying problems.

So while deliberately throwing exceptions is not considered *bad* practice, overuse of this technique is certainly not considered *good* practice.

8.7 Exception class methods

We have considered the various circumstances under which exceptions are thrown. But what, exactly, does the exception object *contain*?

Since every exception inherits from `Throwable`, every exception will have the following methods:

java.lang.Throwable	
`+getMessage(): String`	Returns a message that describes this exception object
`+toString(): String`	Returns a string consisting of: 1. the name of the exception class 2. a colon ":" 3. the getMessage() method
`+printStackTrace():void`	Prints out the call stack trace information to the console
`+getStackTrace(): StackTraceElement[]`	Returns an array of stack trace elements, i.e. what's happened that led to this condition

*Taken from: Liang D. Y. (2014). *Introduction to Java Programming, Comprehensive Version, 10th Ed*. Toronto, ON: Pearson. pg. 461, with modifications

8.7 Exception class methods

The following code demonstrates how these methods can be put to use. First, we declare a flawed method designed to throw an exception by attempting to add the contents of an array beyond the last element of the array:

```
public class TestException {

   private static int sum(int[] list){
      int result = 0;
      for (int i = 0; i <= list.length; i++)
         result += list[i];  // sum each element in array
      return result;
   }
   ...
```

*Taken from: Liang D. Y. (2014). *Introduction to Java Programming, Comprehensive Version, 10th Ed*. Toronto, ON: Pearson. pg. 462, with modifications

8.7 Exception class methods

In `main()`, we call the method and then capture, and output, the exception generated:

```
public static void main(String[] args){
    try{
        System.out.println(sum(new int[]{1, 2, 3, 4, 5}));
    }
    catch (Exception ex){
        ex.printStackTrace();
        System.out.println("\n" + ex.getMessage());
        System.out.println("\n" + ex.toString());
        System.out.println("\nTrace Info Obtained from
            getStackTrace");
        StackTraceElement[] stAr = ex.getStackTrace();
        for (StackTraceElement st: stAr){
            System.out.print("method " + st.getMethodName());
            System.out.print("(" + st.getClassName() + ":");
            System.out.println(st.getLineNumber() + ")");
        }
    }
}
}
```

ALGONQUIN COLLEGE

8.7 Exception class methods

The output is shown below:

```
java.lang.ArrayIndexOutOfBoundsException: 5
        at TestException.sum(TestException.java:7)
        at TestException.main(TestException.java:13)

5

java.lang.ArrayIndexOutOfBoundsException: 5

Trace Info Obtained from getStackTrace
method sum(TestException:7)
method main(TestException:13)
```

- `ex.printStackTrace()`
- `ex.getMessage()`
- `ex.toString()`
- Contents of StackTrace array output using `getStackTrace()`

8.7 Exception class methods

Furthermore, the exception class is overloaded with both a no-arg constructor and a one-String constructor. If the `throw` statement is used, this string can be used to modify the output of `getMessage()`. So for example, adding

```
throw new Exception("Wrong number of arguments in loop");
```

inside a method would ensure that “Wrong number of arguments in loop” was output by calling `ex.getMethod();`

8.8 Custom Exceptions: Extending the Exception class

In addition to throwing known exceptions, we can, as indicated at the start of this module, declare our own exceptions. For example, using internal or external classes, we could define the following exception:

```
public class DivideByZeroException extends ArithemticException {
   @Override
   public String getMessage(){
      return ("Attempt to divide by zero");
   }

   public String seeReference(){
      return ("Liang: Chapter 12.9");
   }

   // Additional new methods, including overridden methods
}
```

8.8 Custom Exceptions: Extending the Exception class

We can use this code in our original calculator program to specify the exact nature of a problem which the `ArithmeticException` was not specifically designed to cover:

```
private double division(int n1, int n2)
                                        throws DivideByZeroException{
     if (n2==0) throw new DivideByZeroException();
     return (n1/n2);
}
```

Then catch this error in `doCalculations()`:

```
     ...
     catch (DivideByZeroException ex){
        System.out.println(ex.getMessage);
     }
     catch (ArithmeticException ex){
        System.out.println("Bad value input; re-enter input");
     }
     catch (InputTypeMismatchException ex){
        System.out.println("Bad operation input; try again");
     }
```

8.9 `finally`

One last feature of the try-catch syntax needs to be addressed. As previously described, the `try` *or* `catch` will not always be executed in a try-catch block, since

1. there could be an error in the `try` statement, so it fails at the beginning of the block, or
2. the `catch` may be of the wrong type, so the exception is caught elsewhere, or not at all

In certain circumstances, it may be useful to have some statements executed regardless of which of the two block of code gets executed. This is what the `finally` block does in the following code:

```
try{
   // statements to execute go here
}
catch (SomeException ex){
   // deal with the exception
}
finally{
   // this code always gets executed
}
```

ALGONQUIN COLLEGE

8.9 finally

`finally` is the 'final' catch in any try-catch block: it catches everything.

```
void myMethod(){
    try{
        // call a method
    }
    catch (SomeException ex){
        // handle ex
    }
    finally{
        // this code always executed
    }

    // other statements
}
```

1

2

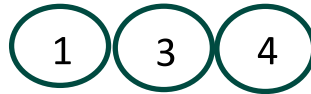
3

4

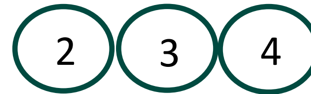
ALGONQUIN COLLEGE

8.9 `finally`

So if execution proceeds normally, the order of execution is



If an exception is thrown in the `try` and caught, the order of execution is



If an exception is thrown in the `try` and *not* caught, the order or execution is

(This assumes that the thrown exception is caught somewhere else.)

Notes

1. `finally` is *always* executed, even when a `return` statement is encountered prior to reaching `finally`.

2. `catch` may be omitted when `finally` is used; the code that would otherwise go in `catch` goes in `finally` instead.

3. Exceptions may be *rethrown* inside `catch` statements. This is sometimes necessary if a condition has been modified, but still needs to be checked for correctness before execution recommences.

4. As with constructors, exceptions may be chained, so that one exception calls another that is more appropriate for dealing with a particular kind of problem.

5. Exceptions are time-consuming to process, and should be avoided whenever possible. For example, if an object could be `null`, its generally better to simply check for `null` rather than throw a `NullPointerException` and deal with the problem in a `catch` statement. In other words, if you can use an `if` statement to test for a problematic condition before and error occurs, then that's probably the way to do it. But if a problem can be triggered by several possible causes, `try-catch` is the better choice.

Summary: Exceptions and Exception Handling

At the start of this module we introduced exception handling in analogy to event handling. But as you've seen, exceptions are more complex than events, which appear relatively straightforward by comparison.

thrown internally

Declare exception to be thrown using `throws`

Caught using `catch`, following `try`

`Exception` thrown → **`Exception` was declared** → **`Exception` caught**

thrown by code using `throw`

Can throw your own exception class

Unchecked exceptions do not need to be declared

Which exception is caught depends on polymorphism

Don't need `catch`, if you have `finally`

Can use Exception class methods to output information