Exceptions

Introduction to the Java Programming Language

Produced by

Eamonn de Leastar
edeleastar@wit.ie

Department of Computing, Maths & Physics
Waterford Institute of Technology

http://www.wit.ie

http://elearning.wit.ie

Exceptions

- Definition / Motivation
- Exception types
- Exception Hierarchy
- Catching exceptions
- Throwing exceptions
- Defining exceptions
- Common exceptions and errors

Motivation

Exceptions provide the means to separate the details of what to do when something out of the ordinary happens from the main logic of a program e.g.

- What happens if the file can't be opened?
- What happens if the length of the file can't be determined?
- What happens if enough memory can't be allocated?
- What happens if the read fails?
- What happens if the file can't be closed?

```
readFile
{
   open the file;
   determine its size;
   allocate that much memory;
   read the file into memory;
   close the file;
}
```

Motivation (2)

```
errorCodeType readFile {
    initialize errorCode = 0;

    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
        close the file;
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else {
            errorCode = errorCode and -4;
        }
    } else {
        errorCode = -5;
    }
    return errorCode;
}
```

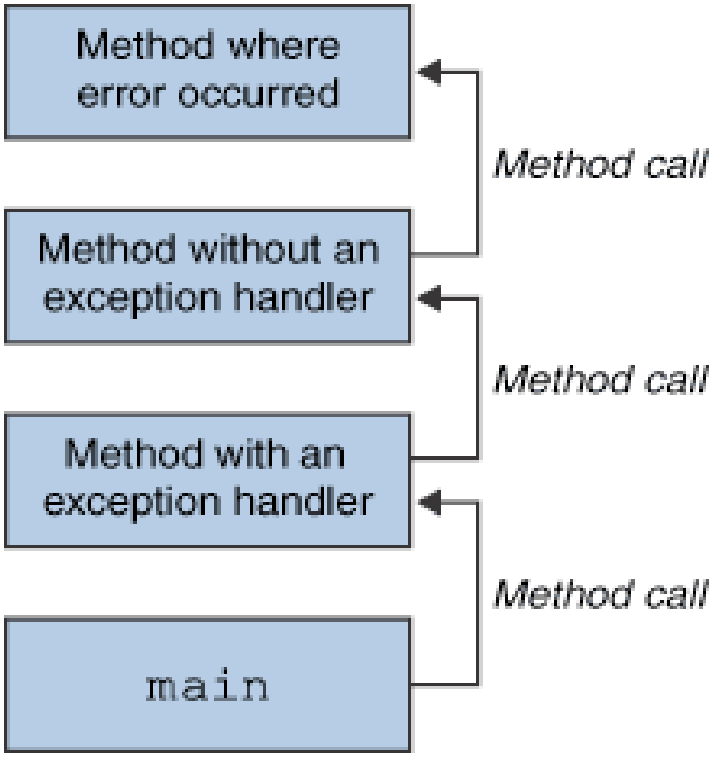
Motivation (3)

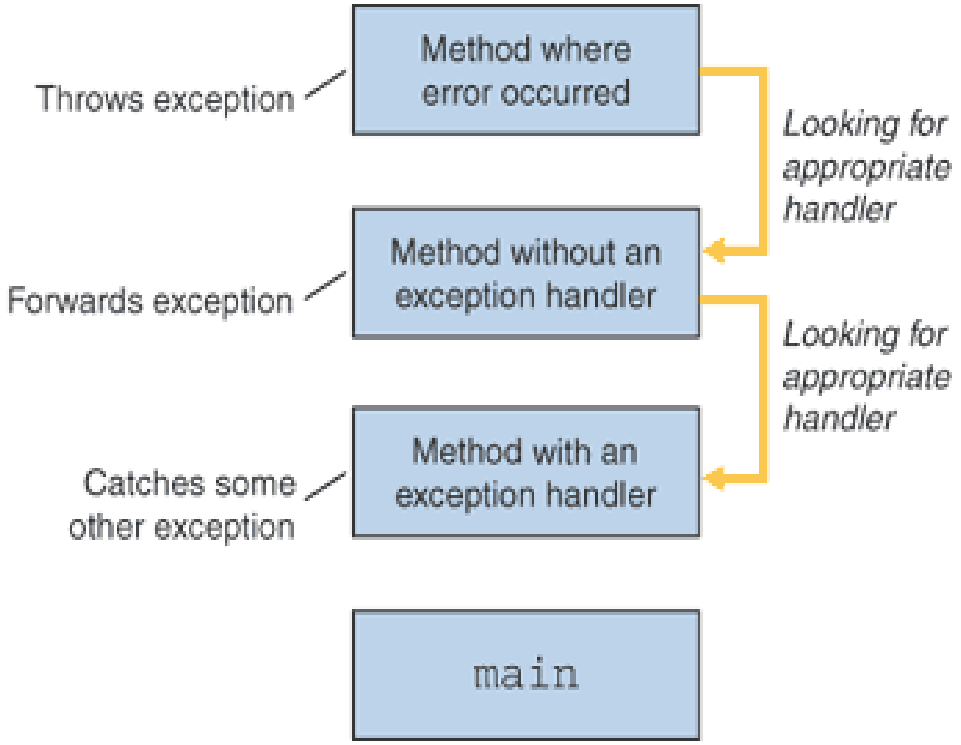
```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
       doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

What is Exception?

- Exceptions are unexpected conditions in a program.
- Exceptions happen at the different levels in a program
  - They are usually handled at the different levels:
    - Where they occur
    - At another level
- Exception examples:
  - Opening file that does not exist
  - Incorrect format found in an input stream
  - Network error during communication activity

Throwing/Forwarding/Catching





The Catch or Specify Requirement

- Valid Java programming language code must honor the *Catch or Specify Requirement*.
- This means that code that might throw certain exceptions must be enclosed by either of the following:
  - A try statement that catches the exception. The try must provide a handler for the exception, as described in Catching and Handling Exceptions.
  - A method that specifies that it can throw the exception. The method must provide a throws clause that lists the exception, as described in Specifying the Exceptions Thrown by a Method.
- Code that fails to honor the Catch or Specify Requirement will not compile.
- Not all exceptions are subject to the Catch or Specify Requirement.

Exceptions

- Definition / Motivation
- Exception types
- Exception Hierarchy
- Catching exceptions
- Throwing exceptions
- Defining exceptions
- Common exceptions and errors

Three Kinds of Exception in Java

1. **Checked Exception**: Exceptional conditions that a well-written application should anticipate and recover from

   e.g. attempt to open non-existent file

   Checked exceptions *are* subject to the Catch or Specify Requirement

2. **Errors**: Exceptional conditions that are external to the application, and that the application usually cannot anticipate or recover from

   e.g. hardware malfunction

   Errors *are not* subject to the Catch or Specify Requirement

3. **Runtime Exceptions:** These are exceptional conditions that are internal to the application, and that the application usually cannot anticipate or recover from

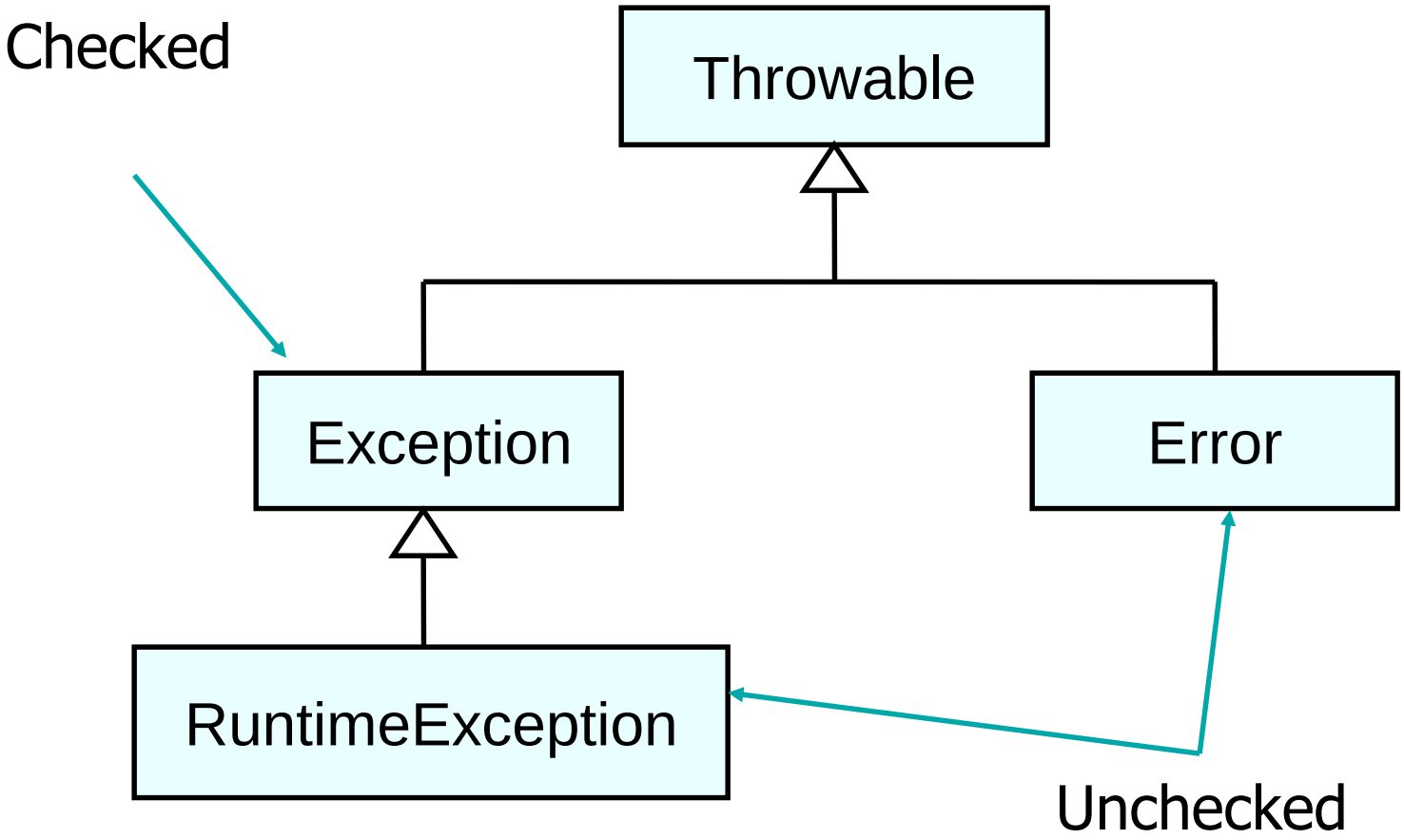
   e.g. API misuse (supply null in place of file name)

   Runtime exceptions *are not* subject to the Catch or Specify Requirement

Exceptions

- Definition / Motivation
- Exception types
- Exception Hierarchy
- Catching exceptions
- Throwing exceptions
- Defining exceptions
- Common exceptions and errors

Exception Hierarchy…



…Exception Hierarchy

Throwable	Top of the exception hierarchy in Java, all exceptions are of this type.
Error (unchecked exception)	Represents serious problems in program, that usually cannot be covered from e.g hardware malfunction.
Exception (checked exception)	Superclass for all exceptions including user-defined exceptions. Users extend from this class exceptions that can be recovered from.
RuntimeException (unchecked exception)	Generally caused by illegal operations, bad API usage etc... These exceptions indicate serious bug that cannot be recovered from and should be eliminated from application.

Exceptions

- Definition / Motivation
- Exception types
- Exception Hierarchy
- Catching exceptions
- Throwing exceptions
- Defining exceptions
- Common exceptions and errors

Handling Exceptions in Java

- There are two different mechanisms for handling Java exceptions:
  - Handling exceptions directly in the method where they are caught.
  - Propagating exceptions up the call stack to the calling method
    - The calling method then handles the exceptions
- Which way you will handle exceptions depend on the overall design of the system.

try-catch block

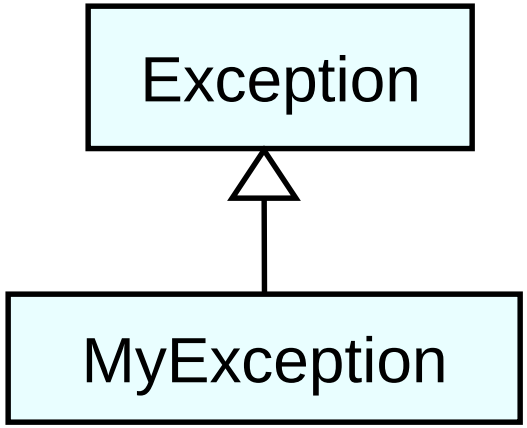
- Exceptions are handled in a try-catch block
  - Checked exceptions can be wrapped in a try-catch block unless they are propagated to a calling method
  - Exceptions in a catch block can be any exception of Throwable type

```
public void myMethod()
{
   try
   {
      //code that throws exception e
   }
   catch (Exception e)
   {
      //code that handles exception e
    }
}
```

Catching Multiple Exceptions

- It is possible to catch multiple exceptions in a catch block
- Order of exceptions is important as more generic exceptions should be handled at the end

```
public void myMethod()
{
   try
   {
      //code that throws exception e1
      //code that throws exception e2
   }
   catch(MyException e1)
   {
      //code that handles exception e1
   }
   catch(Exception e2)
   {
      //code that handles exception e2
   }
}
```



finally block

- Executes always at the end after the last catch block
- Commonly used for cleaning up resources (closing files, streams, etc.)

```
public void myMethod()
{
  try
  {
    //code that throws exception e1
    //code that throws exception e2
  }
  catch (MyException e1)
  {
    //code that handles exception e1
  }
  catch (Exception e2)
  {
    //code that handles exception e2
  }
  finally
  {
    //clean up code, close resources
  }
}
```

The try-with-resources statement (1)

- Introduced in Java 7.

- The try-with-resources statement is a try statement that declares one or more resources. A *resource* is an object that must be closed after the program is finished with it.

- The try-with-resources statement ensures that each resource is closed at the end of the statement.

The try-with-resources statement (2)

```
static String readFirstLineFromFile(String path) throws IOException {
    BufferedReader br = new BufferedReader(new FileReader(path));
    //try with a finally block, pre Java 7.
    try {
        return br.readLine();
    }
    finally {
        if (br != null)
            br.close();
    }
}
```

```
static String readFirstLineFromFile(String path) throws IOException {
    //try-with-resources, Java 7.  br will be closed regardless of
    //whether the try statement completes normally or abruptly
    try (BufferedReader br = new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

The try-with-resources statement (3)

- A try-with-resources statement can have catch and finally blocks just like an ordinary try statement.

- In a try-with-resources statement, any catch or finally block is run after the resources declared have been closed.

Multiple Exception Handling

- In Java 7 and later, you can catch more than one type of exception with one exception handler i.e.
  - A single catch block can handle more than one type of exception. This feature can reduce code duplication and lessen the temptation to catch an overly broad exception.

- In the catch clause, specify the types of exceptions that block can handle, and separate each exception type with a vertical bar (|).

```
catch (IOException|SQLException ex) {
    logger.log(ex);
    throw ex;
}
```

Propagating Exceptions

- Can be used instead of try-catch block
  - Let the calling method handle the exception
- Need to declare that the method (in which code is defined) throws the exception
  - Keyword throws is used in method declaration

```
public void myMethod() throws Exception
{
  //code that throws exception e
}
```

Handling Generic Exceptions

- If you catch generic exception that will catch all the exceptions of that particular type.
- For example, catching Throwable will handle checked and unchecked exceptions.

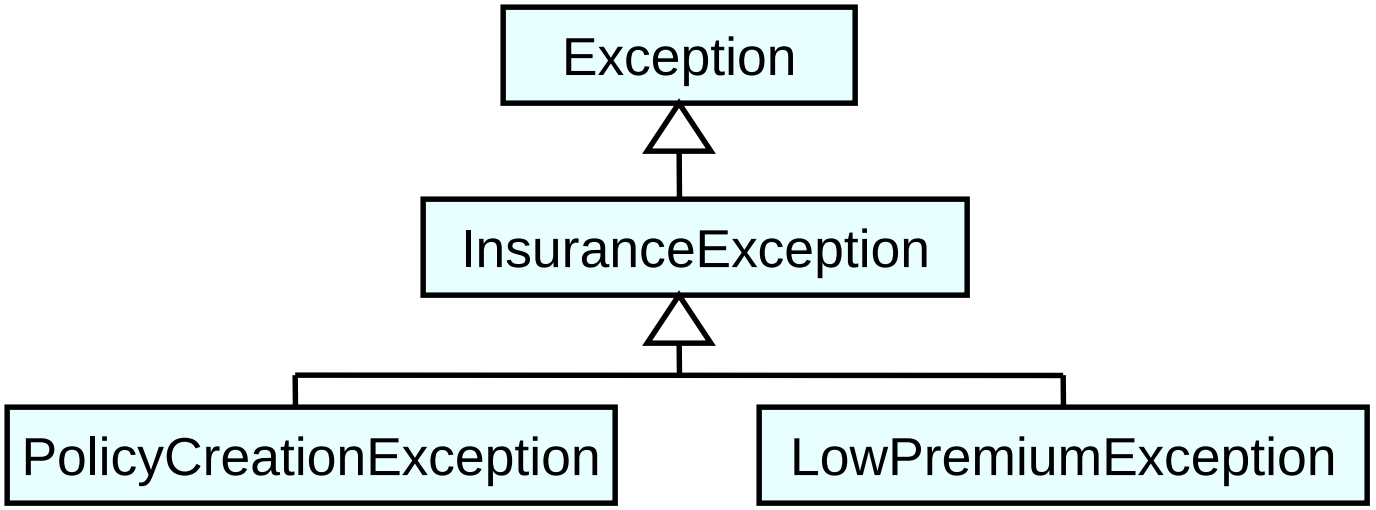
```
public void myMethod()
{
   try
   {
      //code
    }
   catch (Throwable e)
   {
     System.out.println(e.printStackTrace());
   }
}
```

Exceptions

- Definition / Motivation
- Exception types
- Exception Hierarchy
- Catching exceptions
- Throwing exceptions
- Defining exceptions
- Common exceptions and errors

Creating new Exceptions

- It is possible to create new exception types specific to the application
- These must be subclasses of Exception class
- For example, exception hierarchy for an insurance application could be:



Throwing Exceptions

- To throw new exception:
  - Use keyword throw
  - Create a new instance of exception

```
public class PolicyFactory
{
  public Policy createPolicy(Policyable aPolicyable)
        throws PolicyCreationException
  {
    if (aPolicyable.doesMatchInsuranceCriteria())
    {
      return aPolicyable.createPolicy();
    }
    else
    {
      throw new PolicyCreationException();
    }
}
```

Exceptions

- Definition / Motivation
- Exception types
- Exception Hierarchy
- Catching exceptions
- Throwing exceptions
- Defining exceptions
- Common exceptions and errors

Some Common Java Exceptions

- Unchecked, subclass of RuntimeException:
  - NullPointerException
    - Thrown if a message is sent to null object
  - ArrayIndexOutOfBoundsException
    - Thrown if an array is accessed by illegal index
- Checked:
  - IOException
    - Generic class for exceptions produced by input/output operations
  - NoSuchMethodException
    - Thrown when a method cannot be found (Good example here)
  - ClassNotFoundException
    - Thrown when application tries to load class but definition cannot be found (good example here).

Some Common Java Errors

- NoSuchMethodError
  - Application calls method that no longer exist in the class definition
    - Usually happens when a class definition removes a method and is recompiled, but other classes using the “removed” method are not recompiled.
- NoClassDefFoundError
  - JVM tries to load class and class cannot be found
    - Usually happens if classpath is not set, or class somehow gets removed from the classpath
- ClassFormatError
  - JVM tries to load class from file that is incorrect
    - Usually happens if class file is corrupted, or if it isn’t class file

Silent Fail Problem

- What happens if an exception occurs in this code?
- Who is monitoring the stack trace log file?

```
public void process()
{
  // do something
}
```
unchecked

```
public void process() throws Exception
{
  // do something
}
```
checked

```
public void process()
{
  try
  {
    // do something
  }
  catch(Exception e)
  {
    e.printStackTrace();
  }
}
```
checked

Checked Vs Unchecked?

- Because Java does not require methods to catch or to specify unchecked exceptions programmers may be tempted to write code that throws only unchecked exceptions (or make all their exception subclasses inherit from RuntimeException).

- This allows programmers to write code without bothering with compiler errors and without bothering to specify or to catch any exceptions.

- Seems convenient to the programmer, as it sidesteps the intent of the catch or specify requirement.

Oracle Advice (Java Tutorial)

- Generally speaking, do not throw a RuntimeException or create a subclass of RuntimeException simply because you don't want to be bothered with specifying the exceptions your methods can throw.

- Bottom line guideline: If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception.

- For alternative view see:

  http://www.mindview.net/Etc/Discussions/CheckedExceptions

Summary

- What exceptions are
- What types of exceptions exist in Java
- Exceptions hierarchy
- Catching exceptions
- Throwing exceptions
- Defining exceptions
- Common exceptions and errors