Exceptions

Produced by: Dr. Siobhán Drohan (sdrohan@wit.ie)
Eamonn de Leastar (edeleastar@wit.ie)

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

Department of Computing and Mathematics
http://www.wit.ie/

Exceptions

- Motivation / Definition
- Catching and Throwing Exceptions
- Java 7+ and Exceptions
- Exception Hierarchy
- Handling Mechanisms
- 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.

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.

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

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.

- *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 Example

```
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;
}
```

Using a series of if statements

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

Motivation Example

```
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;
    }
}
```

Using exceptions

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

What are Exceptions?

Unexpected conditions in a program.

Objects that signal that some unusual condition has occurred while the program is executing.

Java has many predefined Exception objects, and we can also create our own.

What are Exceptions?

Exceptions are intended to be *detected* and *handled*, so that the program can continue in a sensible way if at all possible.

When an exception occurs…

...the normal flow of execution is disrupted and transferred to code (catch), which can handle the exception condition.

The exception mechanism is a lot cleaner than having to check an error value after every method call that could potentially fail.

Exceptions

- Motivation / Definition
- Catching and Throwing Exceptions
- Java 7+ and Exceptions
- Exception Hierarchy
- Handling Mechanisms
- Defining exceptions
- Common exceptions and errors

try and catch - syntax

Catching an exception means declaring that you can handle exceptions of a particular class from a particular block of code.

To catch exceptions - surround a block of code with a "try, catch" statement.

try and catch - syntax

Catching an exception means declaring that you can handle exceptions of a particular class from a particular block of code.

To catch exceptions - surround a block of code with a "try, catch" statement.

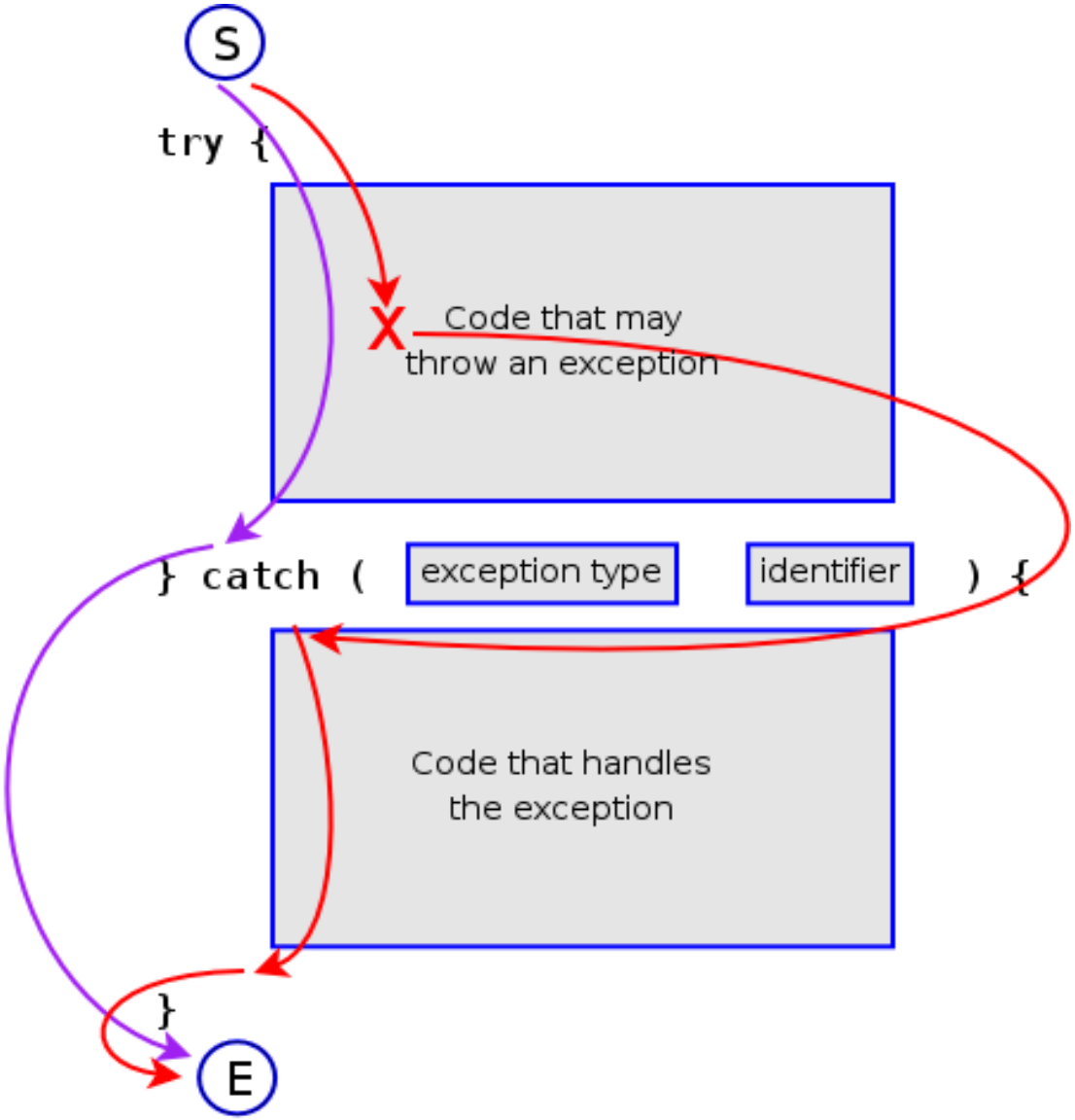
```
public void myMethod()
{
   try{
      //Code that throws exception e:
      //   The try clause is the piece of code which you want
      //   to try to execute.  It contains statements in which an
      //   exception could be raised.
   }
   catch (Exception e){
      //Code that handles exception e:
      //   The catch clauses are the handlers. for the various exceptions
      //   They contain code to handle the Exception and recover from
      //   it (if possible).
    }
}
```

try and catch - example

The parameter ***e*** is of type Exception and we can use it to print out what exception occurred.

```
try{
   //Code that throws exception e:
    myMethod();
}
catch (Exception e){
   //Code that handles exception e:
   System.err.println("Caught Exception:  " + e);
```

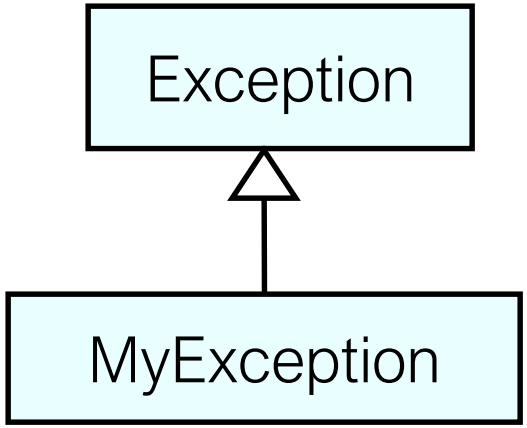
Flow of control in Exception Handing



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 Catch or Specify Requirement

- Valid Java 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

- Motivation / Definition
- Catching and Throwing Exceptions
- Java 7+ and Exceptions
- Exception Hierarchy
- Handling Mechanisms
- Defining exceptions
- Common exceptions and errors

try-with-resources

- Introduced in Java 7.
- It 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.

RECAP - try-with-resources


```
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();
     }
}
```

try-with-resources

- 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.

- All classes implementing the interface java.lang.AutoCloseable can be used inside the try-with-resources construct.

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.

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.

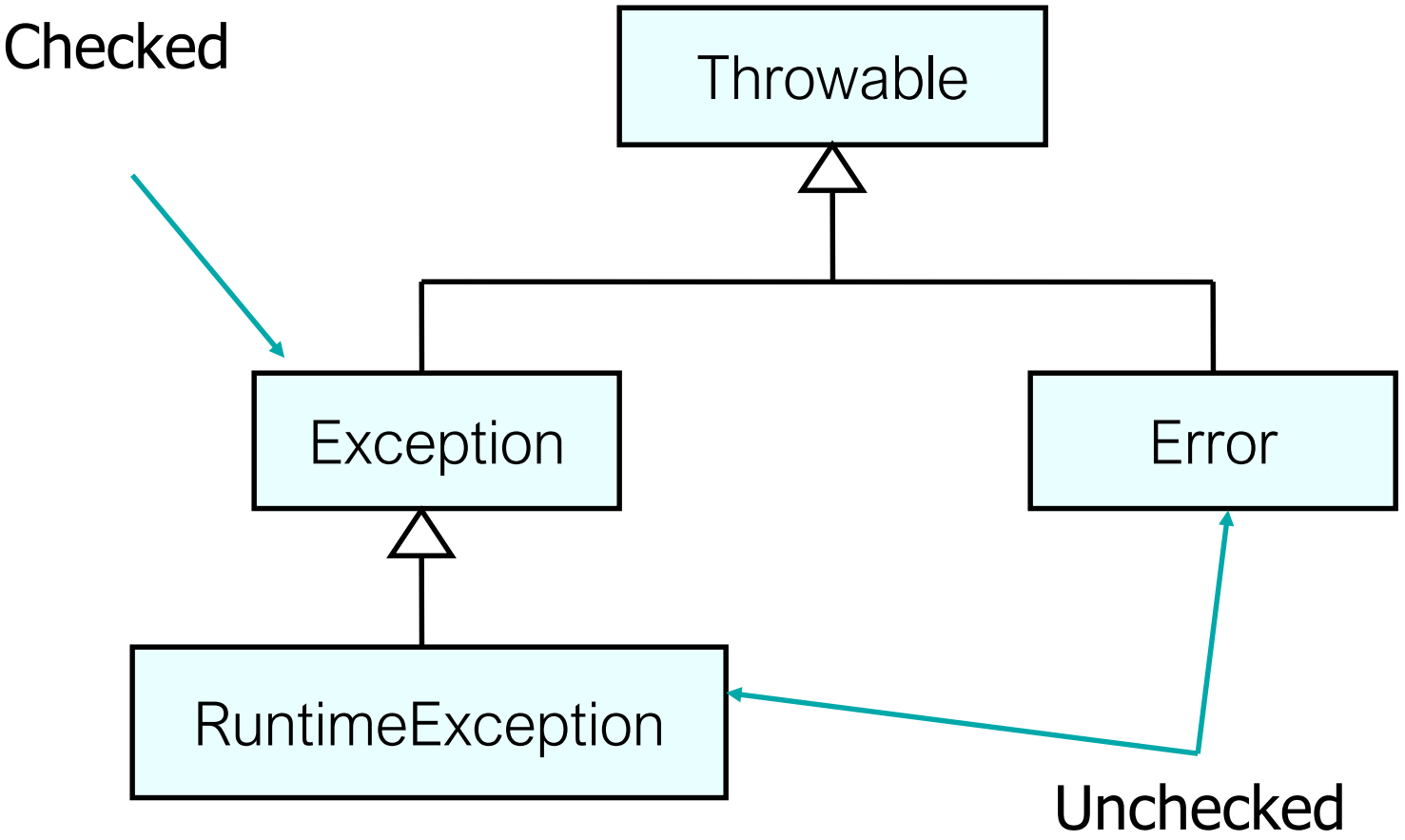
```
catch (IOException|SQLException ex) {
    logger.log(ex);
    throw ex;
}
```

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

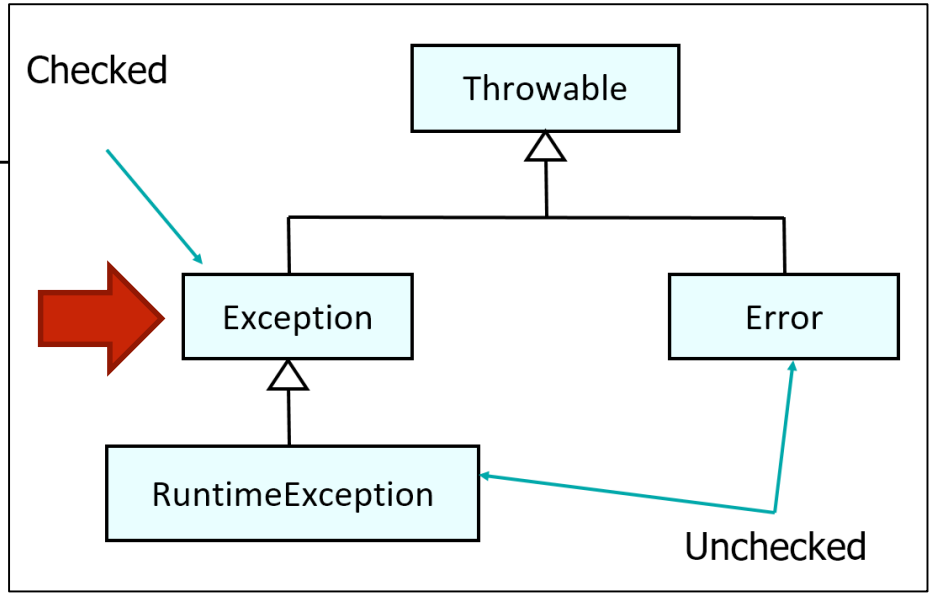
Exceptions

- Motivation / Definition
- Catching and Throwing Exceptions
- Java 7+ and Exceptions
- Exception Hierarchy
- Handling Mechanisms
- Defining exceptions
- Common exceptions and errors

Exception Hierarchy



Three Kinds of Exception in Java



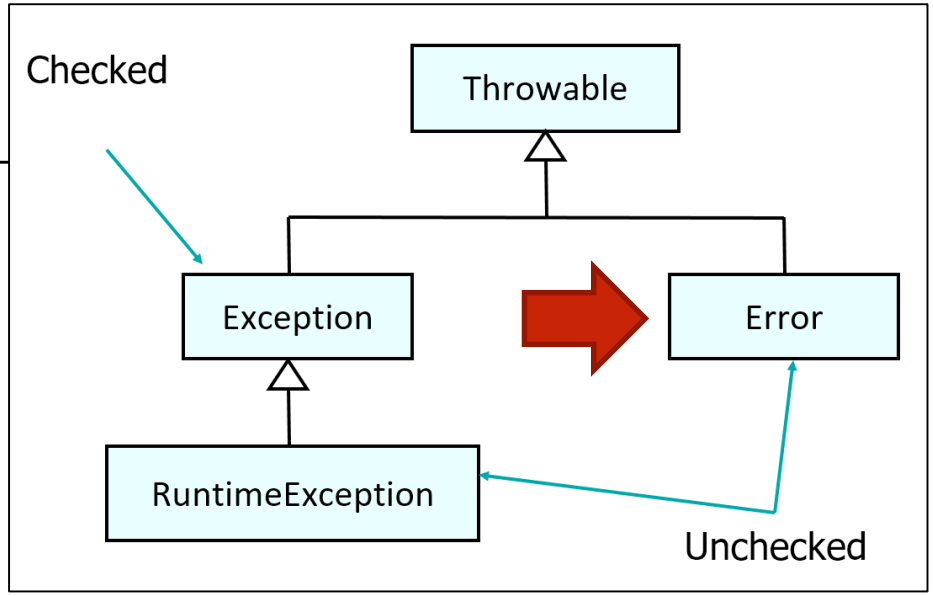
1. Checked Exception:

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.

Three Kinds of Exception in Java



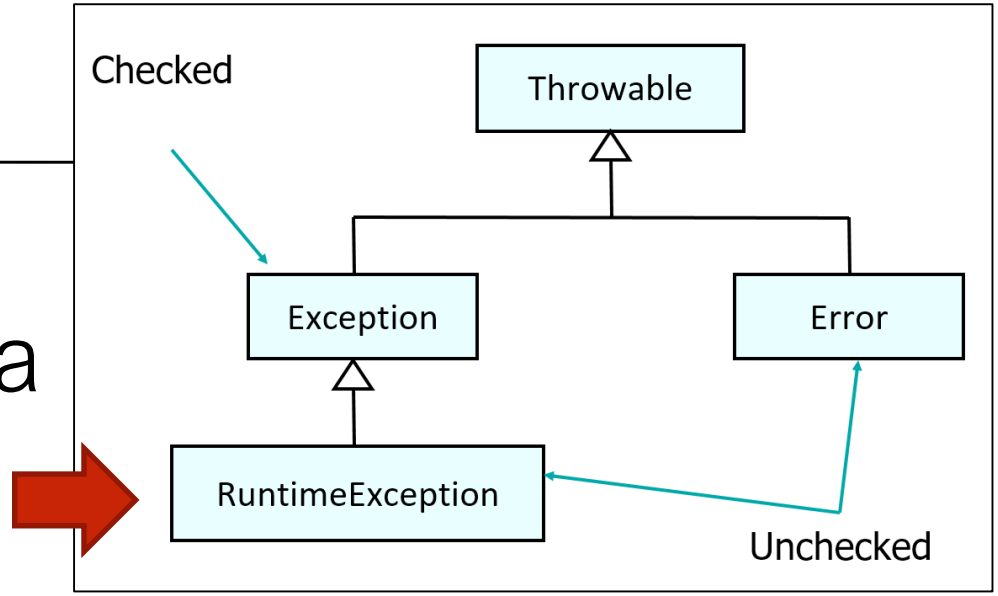
2. Errors:

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

Three Kinds of Exception in Java



3. Runtime Exceptions:

These are exceptional conditions that are internal to the application that usually happen at runtime. Typically can be avoided through good programming practices!

e.g. ArithmeticException(dividing by zero)
NullPointerException (trying to access a null object)

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

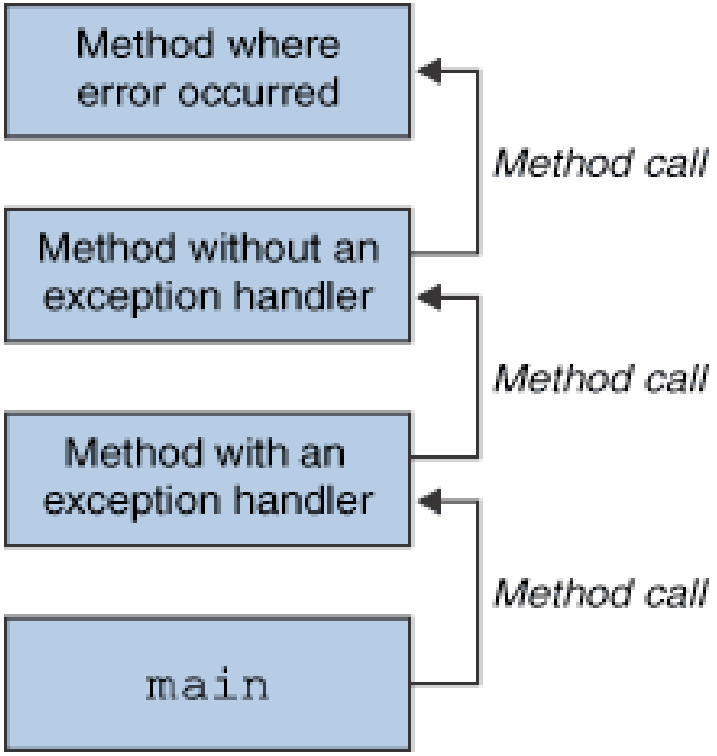
Exceptions

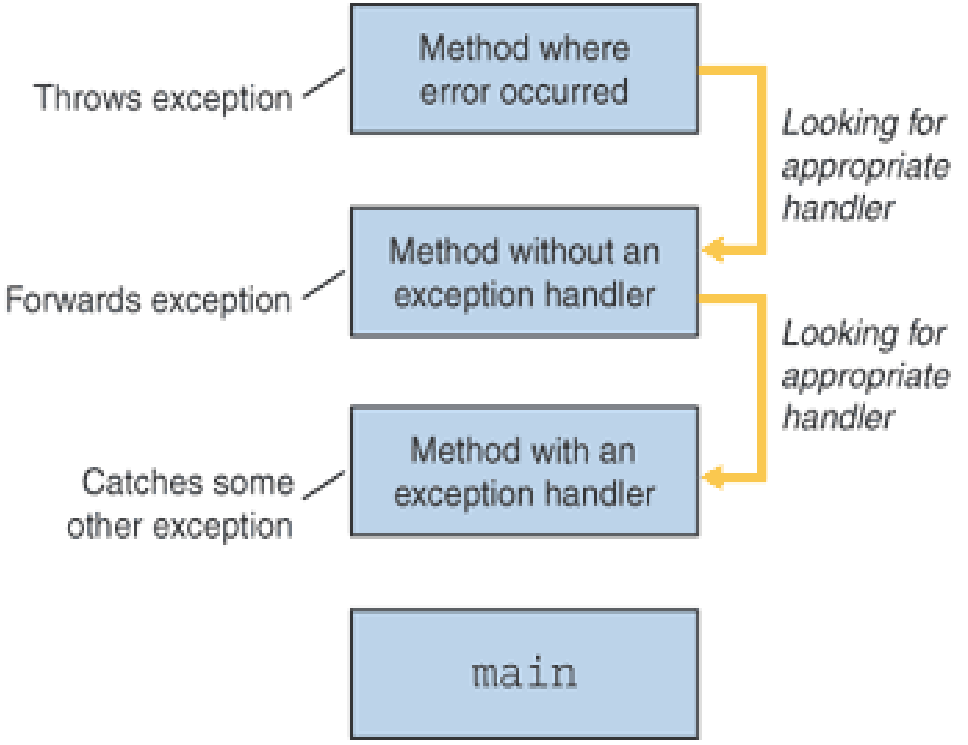
- Motivation / Definition
- Catching and Throwing Exceptions
- Java 7+ and Exceptions
- Exception Hierarchy
- Handling Mechanisms
- Defining exceptions
- Common exceptions and errors

Handling Exceptions in Java

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

Throwing/Forwarding/Catching





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.

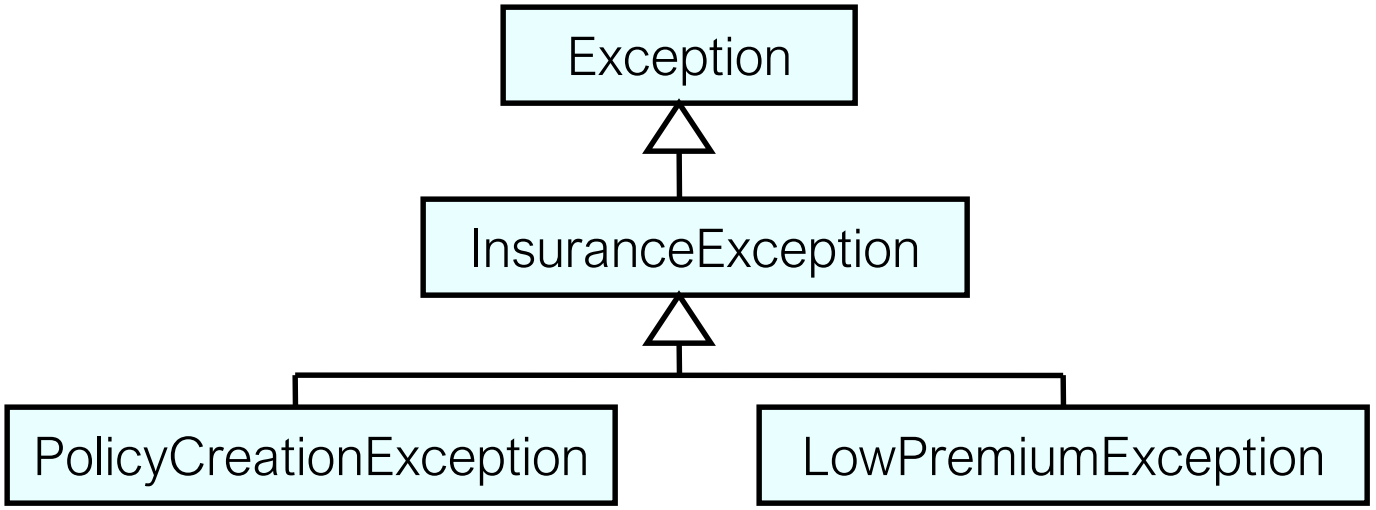
```
public void myMethod()
{
   try
   {
      //code that can throw checked/unchecked exceptions
    }
   catch (Throwable e)
   {
     System.out.println(e.printStackTrace());
   }
}
```

Exceptions

- Motivation / Definition
- Catching and Throwing Exceptions
- Java 7+ and Exceptions
- Exception Hierarchy
- Handling Mechanisms
- 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

- Motivation / Definition
- Catching and Throwing Exceptions
- Java 7+ and Exceptions
- Exception Hierarchy
- Handling Mechanisms
- 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()
{
   try
   {
     // do something
   }
   catch(Exception e)
   {
     e.printStackTrace();
   }
}
```

checked

Any questions?