Two main families

Runtime vs. checked exceptions

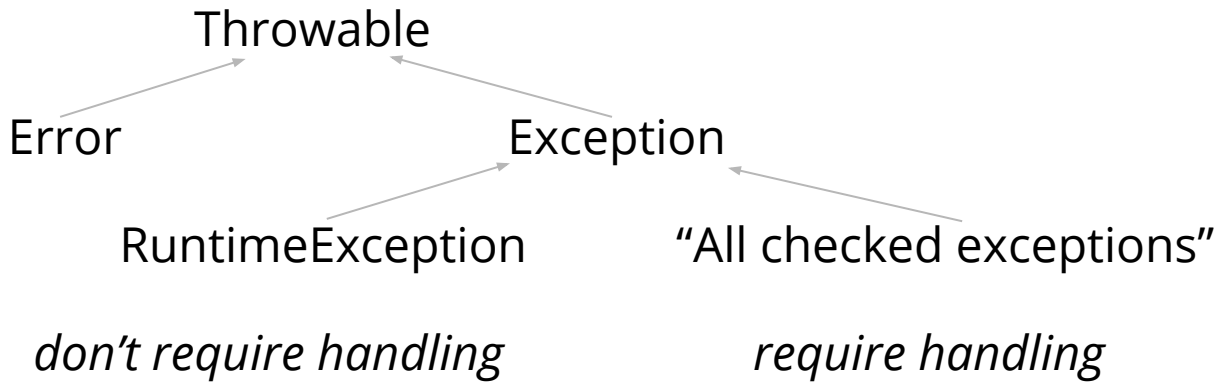
Some exceptions need to be dealt with

Exceptions are just plain Java objects. The only thing different about these objects are that they can be “thrown” from a method and “caught” by some code in some other method.

The Java API contains a class hierarchy from which exceptions are inherited.

At the root we have Throwable. Under it we have Error and Exception. All classes which extend Exception directly are so called checked exceptions, which must be dealt with - except the descendents of RuntimeException.

Hierarchy for exceptions



Checked exceptions require some code

A method which declares that it throws any type of exception which is not in the runtime branch (doesn’t extend RuntimeException directly or indirectly) is throwing a so called checked exception.

Any call to such a method must consider that the method might throw an exception. This must be dealt with in one or two ways:

The code can catch the exception and try to do something about it, or it must be declared in a method which in turn declares that it too throws this type of exception.

Checked exceptions force us to prepare

There are quite a few methods in the Java API which are declared to throw checked exceptions. If you call one of those methods, you must take care of that, which is something good.

Consider a method for reading a file. Such a method would typically declare that it throws some kind of IOException. It forces you to be prepared also for the case that the file doesn't exist, or that the program isn't allowed by the operating system to read the file.

Don't take this as something bad which forces you to "write a lot of extra code". Think of it as a thing that makes you think also about fringe cases!

Runtime exceptions

All exception classes which extend RuntimeException directly or indirectly are called runtime exceptions. Such exceptions might be thrown by a method, but there is no obligation for you to deal with them. Not dealing with them at all will make the exception bubble up all the way to the main method which will crash your application with an error message.

Examples include ArithmeticException (like division by zero), NullPointerException (trying to follow a reference which is null to an object and accessing an instance method or variable) and ClassCastException (trying to perform an illegal cast on a reference).

Some exceptions come from programmer errors

Let’s say we have a method changeEmail(String) which is documented that it won’t accept a null reference as argument.

If some code called changeEmail with a null argument anyway, this is clearly a violation of the rules described by the documentation and thus a programmer error.

If this rule is critical for the system, it is normal to expect the changeEmail() method to return control to the caller abnormally. This is typically done by the method throwing an exception like NullPointerException.

If the caller doesn’t prepare for this, the exception will continue up the chain.

If the programmer error is severe and a bug...

It can be argued that bugs from programmer errors should allow an exception to propagate up the call chain, even up to the main method.

The main method might or might not have a handler for the exception.

If it has a handler and decide the error is critical, it should shut down the application and log the error so that developers can locate the bug and fix it.

If it doesn't have a handler, the program will crash with an error message which could be reported by the user to the developers, so they can track down the programmer error and fix it. (The user may opt to do nothing, though...)

Critical failure which come from bugs

Such critical failures which stem from programmer mistakes, could be handled by throwing an exception of the type RuntimeException (or any subclass to it).

The thinking behind that, is that runtime exceptions do not pose any restrictions on the calling code which demand that they write a handler to catch and cope with the exception if it is thrown.

This means that the caller will probably not handle the exception and it will propagate up instead. But, hey, if it is a critical problem, the caller surely can’t do anything about it anyway.

Checked exceptions are sometimes critical too

One type of checked exception is the IOException family. It includes FileNotFoundException and FileSystemException.

It is not hard to imagine that any of those exceptions could mean a critical failure for some application. However, they are checked exceptions, so the caller must provide a handler or declare that it could be thrown so that its caller(s) must provide a handler.

But if it is a critical failure, no handler in the world can “fix the problem” like repairing a broken file system. So what to do?

A strategy for critical failure checked exceptions

A strategy which could be used when risking a critical failure thrown as a checked exception is to handle it where it occurs and rethrow it as an unchecked exception with diagnostic information.

The idea behind that is that the calling method couldn’t either be expected to fix the problem, so why force that code to include a handler then?

This also allows for critical exceptions to propagate up to some global exception handler, regardless of it consists of an original checked exception or a plain old runtime exception.

Simple example

```
  public static void main(String[] args){
    initApplication();                                    //1. call initApplication
  }
  static void initApplication(){
    loadSuperImportantFile();                             //2. call loadSuper...
  }
  static void loadSuperImportantFile(){
    // pretend that you see code here which tries to load an important
    // file and fails because of a checked exception

    // pretend that this is the handler of the
    // checked exception:                                 // OOOOOPS - rethrow!
      throw new RuntimeException("Important file not found, omg", e);
    }
  }
}
```

Simple example explained

The main method calls initAppliation() which calls

loadSuperImportantFile() which runs into a NoSuchFileException which is a

checked exception!

Rather than passing the responsibility up the chain of calls, the method deals with the exception and throws a RuntimeException!

This propagates all the way back up to main which crashes with an error message.

User experience

```
$ java Critical
Exception in thread "main" java.lang.RuntimeException: Important file not found, omg
     at Critical.loadSuperImportantFile(Critical.java:24)
     at Critical.initApplication(Critical.java:13)
     at Critical.main(Critical.java:10)
Caused by: java.nio.file.NoSuchFileException: important_file/somepath
     at sun.nio.fs.UnixException.translateToIOException(UnixException.java:86)
     at sun.nio.fs.UnixException.rethrowAsIOException(UnixException.java:102)
     at sun.nio.fs.UnixException.rethrowAsIOException(UnixException.java:107)
     at sun.nio.fs.UnixFileSystemProvider.newByteChannel(UnixFileSystemProvider.java:214)
     at java.nio.file.Files.newByteChannel(Files.java:317)
     at java.nio.file.Files.newByteChannel(Files.java:363)
     at java.nio.file.spi.FileSystemProvider.newInputStream(FileSystemProvider.java:380)
     at java.nio.file.Files.newInputStream(Files.java:108)
     at java.nio.file.Files.newBufferedReader(Files.java:2677)
     at Critical.loadSuperImportantFile(Critical.java:17)
     ... 2 more
```

Handling stuff globally

We could add a handler to the main method which deals with any exception:

```
public static void main(String[] args){
  // This is what a handler could look like:
  try{
    initApplication();
  }catch(Exception e){ // Something really bad happened...
    System.err.println("Severe problem: " + e.getMessage());
    // also, log the problem for the developers! Please!
  }
}
$ javac Critical.java && java Critical
Severe problem: Important file not found, omg: OutFile.txt (Permission
denied)                  # Much nicer message to the user ;-)
```

Next, we'll look at what a handler looks like

In the next we'll look at how you write code to handle an exception, to declare that a method might throw an exception and how to actually throw an exception.