Exceptions

Using exceptions for error handling ensures that code is less likely to miss errors.

The Java programming language provides three kinds of exceptions: checked exceptions, runtime exceptions, and errors. There are some general rules that provide guidance on which kind of exception to use.

Use checked exceptions for conditions from which the caller can reasonably be expected to recover. By throwing a checked exception, you force the caller to handle the exception in a catch clause or to propagate it outward. Each checked exception that a method is declared to throw is therefore an indication to the API user that the associated condition is a possible outcome of invoking the method.

The unchecked exceptions: runtime exceptions and errors, generally shouldn't be caught. If a program throws an unchecked exception or an error, it is usually the case that recovery is impossible and continued execution would do more harm than good. If a program does not catch such an exception, it will cause the current thread to halt with an appropriate error message (caught by the default JVM exception handler).

Of course, reality is not always so clear cut. For example, resource exhaustion could be caused by a programming error such as allocating an unreasonably large array or by an unexpected and possibly temporary shortage of resources. It is a matter of judgment on the part of the API designer whether a given instance of resource exhaustion is likely to allow for recovery. If recovery is likely, use a checked exception; if not, use a runtime exception. If it isn't clear whether recovery is possible, you're probably better off using an unchecked exception.

Working with Exceptions

The Java platform libraries provide a basic set of unchecked exceptions that cover a large fraction of the exception-throwing needs of most APIs. It is good practice to reuse these when possible.

Reusing existing exceptions makes your API easier to learn and use because it matches established conventions. The API is easier to read because they aren't cluttered with unfamiliar exceptions.

Arguably, all erroneous method invocations are either an illegal argument or an illegal state. However, by convention, if a caller uses null as an argument where null values are prohibited, a `NullPointerException` is thrown not `IllegalArgumentException`.

Similarly, if a caller passes an out-of- range value in a parameter representing an index an `IndexOutOfBoundsException` is thrown rather than `IllegalArgumentException`.

Exception	Occasion for Use
`IllegalArgumentException`	Non-null parameter value is inappropriate
`IllegalStateException`	Object state is inappropriate for method invocation
`NullPointerException`	Parameter value is null where prohibited
`IndexOutOfBoundsException`	Index parameter value is out of range
`ConcurrentModificationException`	Concurrent modification of an object has been detected where it is prohibited
`UnsupportedOperationException`	Object does not support method

It is disconcerting when a method throws an exception that has no apparent connection to the task that it performs. This often happens when a method propagates an exception thrown by a lower-level abstraction. This can reveal implementation details that should be hidden by the API. If the implementation of the higher layer changes in a subsequent release, the exceptions that it throws will change too, potentially breaking existing client programs.

To avoid this problem, higher layers should catch lower-level exceptions and, in their place, throw exceptions that can be explained in terms of the higher-level abstraction. This idiom is known as exception translation.

```
try {
    // Use lower-level abstraction to do our bidding
    ...
} catch(LowerLevelException e) {
    throw new HigherLevelException(...);
}
```

Documenting Exceptions

A description of the exceptions thrown by a method is an important part of the documentation required to use the method properly. Declare checked exceptions individually, and document precisely the conditions under which each one is thrown using

the Javadoc @throws tag. Don't take the shortcut of declaring that a method throws some superclass of multiple exception classes that it can throw. As an extreme example, never declare that a method "throws Exception" because such a declaration greatly hinders the use of the method, as it effectively obscures any other exception that may be thrown in the same context.

Java does not require programmers to declare the unchecked exceptions that a method is capable of throwing, it is wise to document them as carefully as the checked exceptions. A well-documented list of the unchecked exceptions that a method can throw effectively describes the preconditions for its successful execution. It is essential that each method's documentation describe its preconditions, and documenting its unchecked exceptions is the best way to satisfy this requirement.

Use the Javadoc @throws tag to document each unchecked exception that a method can throw, but do not use the throws keyword to include unchecked exceptions in the method declaration. It is important that the programmers using your API be aware of which exceptions are checked and which are unchecked, as their responsibilities differ in these two cases. The documentation generated by the Javadoc @throws tag in the absence of the method header generated by the throws declaration provides a strong visual cue to help the programmer distinguish checked exceptions from unchecked.

Java 7 SE

This edition brought automatic resource management and catch block simplifications. If there are many exceptions caught in a single try block, the code quickly becomes cumbersome. In this form a single catch block can catch multiple exceptions. This is useful for exception translation.

```
catch(IOException | SQLException ex){
     throw new HigherLevelException(ex.getMessage());
}
```

The try-with-resources block allows the programmer to create a resource in the try statement itself and use it inside the try-catch block. When the execution comes out of try-catch block, runtime environment automatically close these resources.

```
try (Resource mr = new Resource()) {
  System.out.println("Resource created");
} catch (ResourceException e) {
  e.printStackTrace();
}
```

Exercise

Starting from the following code

```
import java.io.File;
import java.util.Scanner;

public class PredictRainfall {
    private Scanner input;

    public PredictRainfall(String dataSource) {
        this.input = new Scanner(new File(dataSource));

    }
 }
```

Design an API to allow a programmer to create a PredictRainfall object. The constructor takes a parameter that identifies a data source. The data source can take a variety of forms, such as the name of a file, a URL or a DBMS connection. An exemplar implementation will work with text files as shown above.

A PredictRainfall object can read data from its data source. It should include a method to allow an application programmer to make a prediction of the expected rainfall given three rainfall readings.

You need to consider the type of exceptions that could arise when using a PredictRainfall object. Identify the type of these exceptions (checked or unchecked) and where and how they should be handled.

Your API should apply some recommended practices regarding exceptions, including

- Using specific exceptions
- Throw early
- Catch the exception only when it can be handled appropriately
- Do not suppress exceptions
- Preserve the original exception
- Document the exceptions
- Clean up resources
- Preserve loose coupling (i.e. hide implementation specific details from higher levels)

Example Data File

Rainfall data is available from Met Eireann (http://www.met.ie/climate/daily-data.asp). The data is available for each of its weather stations. I have edited this to simplify the format for a single weather station (Cork Airport).


```
date,rain
01/01/1962 00:00,
02/01/1962 00:00,
03/01/1962 00:00,
…
…
22/08/2016 00:00,3.3
23/08/2016 00:00,10
24/08/2016 00:00,0
25/08/2016 00:00,0.1
26/08/2016 00:00,0.4
27/08/2016 00:00,0.2
28/08/2016 00:00,0.1
29/08/2016 00:00,0
30/08/2016 00:00,2
31/08/2016 00:00,0.2
```