WRITING CUSTOM EXCEPTIONS IN JAVA

A quick recap from intro to Exceptions

- Java.lang.**Throwable** is the superclass for all the Exceptions and Errors.
- There can be exceptions which can't be prevented or there are no recovery possible, and they are called an error and are subclasses of the **Error** class.
- Most exceptions can be prevented or recovered using workarounds, and they are all children of the**Exception** class.
- A **checked exception** is an exception that the developer doesn't have any control over the occurrences and he developer must provide a workaround by handling it using try-catch-finally blocks in the method or is explicitly identified as being handled elsewhere through the use of throws clause in that method signature.
- A **runtime exception i**s an exception that represents mostly a coding mistake and doesn't need to be handled, but should be avoided. Exception and all its subclasses, except RuntimeException are checked exceptions. RuntimeException and all its subclasses are unchecked exceptions.

Custom Exceptions

Custom exceptions are user defined Exception classes that extend either Exception class or RuntimeException class.

Why custom exceptions

1. If no existing class meets your need you might want to create a custom exception. This gives you the ability to add custom data to your exceptions.

2. We can wrap system exceptions to provide more data. You can add more data and include system exception as an inner exception. Thus we can provide a more meaningful message.

3. A public method might call some helper methods and these helper methods might throw various internal or implementation or backend related exceptions. These implementation details often should not be made visible to the public method's caller and also these exceptions might not be present in the public method's contract interface. Therefore we can wrap the lower-level exception in a higher level exception that is present in the public method's contract interface.

How to decide whether to subclass Exception or RuntimeException

If the exception will describe a coding mistake, you should subclass **runtime exception**. These exceptions could have been avoided in code and hence need not be handled.

However if you are wrapping up a specific exception in a custom exception which should be handled up in the hierarchy, use **Exception**, or if possible a more specific Exception class. Here your parent classes in the hierarchy should already know about this exception.

Example

Consider an example for a custom exception class:

```
public class MyException extends Exception {

//Constructor that accepts only an error message

public MyException (String message){

    super(message);

  }

//Constructor that accepts an error message and a Throwable

  Public MyException(String message, Throwable cause){

   Super (message, cause);

  }

}
```

All Exceptions inherit from Throwable and hence we can wrap our specific exception inside our custom exception using the second constructor here.

You may use this exception from the code as below:

```
try{

  //some code that throw ABCException

}

catch(ABCException ex)

{

  throw new MyException ( "Some meaningful text message", ex);

}
```

All Exceptions inherit from Throwable and hence we can wrap our specific exception inside our custom exception

Simple Hands On

1. Create a custom exception class MyException extending RuntimeException. It should have a constructor that accepts a string message and calls the super constructor passing that string message.
2. Create a class MyClass and a method myMethod() inside it without any throws clause.
3. Throw our custom exception inside myMethod() as throw new MyException()
4. Now invoke myMethod() from main method of MyClass passing “My custom exception”. Don’t declare any throws on the main method now.
5. Compile all java files.
6. Run MyClass and see the output. You can post it here as comments.
7. Now change the parent of MyException from RuntimeException to Exception.
8. Compile all java files.
9. You will get compilation errors now as you are not handling a checked exception.
10. First, try to handle it using throws alone. You will need to add throws MyException on both myMethod() and then main also().
11. Second, try to declare throws on myMethod(), but handle it in main using try-catch-finally. Now you will not need throws on your main method.
12. Third, try to handle the exception in myMethod() itself by enclosing the throw statement inside a try and catching it there and exiting the program. Now you will not need any throws statement.

Source : http://javajee.com/writing-custom-exceptions-in-java