

```
catch (Exception e) {  
    // --- I caught all application and runtime exceptions ---  
    log_it( e );  
  
    // --- I don't know how to handle this, so I throw it again ---  
    throw e;  
}
```

Rule number 2

When there are a list of catch clauses, it is evaluated sequentially, applying the first rule, for each clause. If there is a catch, the following catch clauses are ignored.

The following piece of code demonstrates rule number 2.

The `NullPointerException` is caught by the first catch clause, the following catch clauses are ignored. The second catch clause will catch all `RuntimeException`'s, except the `NullPointerException`, because that will be caught by the first clause. All exceptions that are defined by the application, will be caught by the last catch clause, see below.

```
catch (NullPointerException e) {  
    // --- I caught a Nulpointer ex. ---  
    log_it( e );  
    // --- I don't know how to handle this, so I throw it again ---  
    throw e;  
}  
  
catch (RuntimeException e) {  
    // --- I caught runtime ex. but not NullPointerException ---  
    log_it( e );  
    // --- I don't know how to handle this, so I throw it again ---  
    throw e;  
}  
  
catch (Exception e) {  
    // --- I caught all application but not Runtime exceptions ---  
    log_it( e );  
    // --- I don't know how to handle this, so I throw it again ---  
    throw e;  
}
```

Rule number 3

If there is a catch clause in the list which will never be executed, because its super-class are listed before it, the compiler will give an error message.

The following piece of code will not compile, it will give an error message saying that the last catch clause will never be executed. This demonstrates that the exception handling is built into the language. The compiler checks the list of catch clauses and reports an error if class follows any of it super-class on the list.

```
catch (NullPointerException e) {  
    // --- I caught a Nullpointer ex. ---  
    log_it( e );  
    // --- I don't know how to handle this, so I throw it again ---  
    throw e;  
}
```

```

}
catch (Exception e) {
    // --- I caught all application but not Runtime exceptions ---
    log_it( e );
    // --- I don't know how to handle this, so I throw it again ---
    throw e;
}
catch (RuntimeException e) { // -- COMPILATION ERROR, THIS CODE WILL
NEVER BE EXECUTED ---
    // --- I caught runtime ex. but not NullPointerException ---
    log_it( e );
    // --- I don't know how to handle this, so I throw it again ---
    throw e;
}

```

Rule number 4

The compiler enforces that all Exception and its sub-class exceptions must be handled by the programmer, except the RuntimeException and its sub-classes.

From the application's point of view there are two types of exceptions. One is called RuntimeException, that suppose to be thrown when there is an anticipated bug in the code. For example, if the program tries to access an array, with a number that is out side of the array size, then a RuntimeException is thrown. There are many other possibilities which indicate a coding bug.

The other type of exception is called the application exception. During application design a list of possible exception is defined that can anticipated during executing the business logic code. For example, the application tries to retrieve a customer, but the customer does not exist. This may not necessarily be a bug. In this case an application exception should be thrown. Application exceptions are sub-classes of the Exception class.

The java language enforces that all application exceptions are dealt with by the application programmer.

If a method throws an application exception, that must be declared in the method signature by the **throws** keyword. The caller of that method can either catch the exception by a catch block or it can make sure that it is being declared to be thrown in the current method, and so on. The search for a catch block goes backward on the methods track trace, until a match is found. The compiler ensures that someone has to handle the exception.

The following method can throw a CustomerNotFoundException, so it must declare it at the method signature.

```

public String method() throws CustomerNotFoundException
{
    ...
    throw new CustomerNotFoundException();
    ...
}

```

Rule number 5

If the try block would never throw an exception that is specified in the catch list, the compiler gives an error.

The following code would not compile because inside the try block no Exception is thrown.

```
try {  
  
    ...  
  
} catch ( Exception e ) {  
  
}
```

The above code won't compile, because the try block does not throw exception. The compiler can be fooled with the following code.

```
try {  
    if ( 1 == 0 ) throw new Exception();  
    ...  
  
} catch ( Exception e ) {  
  
}
```

The above code will compile now.

Example of handling exceptions

Let's examine the following code:

```
public void methodA() throws SomeException, AnotherException  
{  
    //methodbody  
}  
  
public void methodB() throws CustomException  
{  
    //Methodbody  
}  
  
public void methodC()  
{  
    methodB();  
    methodA();  
}
```

In the code sample, methodC is invalid. Because methodA and methodB pass (or throw) exceptions, methodC must be prepared to handle them. This can be handled in two ways: a **try - catch** block, which will handle the exception within the method and a **throws** clause which would in turn throw the exception to the caller to handle. The above example will cause a compilation error, as Java is very strict about exception handling. So the programmer forced to handle any possible error condition at some point.

A method can do two things with an exception. Ask the calling method to handle it by the **throws** declaration. Or handle the exception inside the method by the **try-catch** block.