

To construct a throws declaration, add throws ExceptionName (additional exceptions can be added with commas).
To construct a **try** - **catch** block, use the following syntax

```
...
try {
    //Possibly exception-causing code
}
catch (TheException e) {
    //Handle the exception
}
finally {
    //Optional. Executes regardless of exceptions thrown
}
```

The original code can be modified to work correctly in multiple ways. For example, the following:

```
public void methodC() throws CustomException, SomeException
{
    try{} {
        methodB();
    }
    catch (AnotherException e) {
        //handle it
    }
    methodA();
}
```

The AnotherException from methodB will be handled locally, while CustomException and SomeException will be thrown to the caller to handle it.

Application Exceptions

Application Exception classes should extend the java.lang.Exception class. Some of the JDK classes also throw exception objects inherited from java.lang.Exception. If any of those Exception object is thrown, **it must be caught by the application** some point, by a catch-block. The compiler will enforce that there is a catch-block associated with an exception thrown, if the thrown exception object is inherited from java.lang.Exception and it is not the java.lang.RuntimeException or its inherited objects. However, java.lang.RuntimeException or its inherited objects, can be caught by the application, but that is not enforced by the compiler.

Lets see what is the catching criteria for a catch block to catch the "thrown" exception.

A catch-block will "catch" a thrown exception if and only if:

- the thrown exception object is the same as the exception object specified by the catch-block
- the thrown exception object is the subtype of the exception object specified by the catch-block

```
try {
    throw new Exception( "This will be caught below" );
}
catch( Exception e ) {
    // --- The "thrown" object is the same what is specified at the
    catch-block --
}
```

```
try {
    throw new NullPointerException( "This will be caught below" );
}
catch( Exception e ) {
    // --- NullPointerException is subclass of the Exception class --
}
```

There can be more than one catch-block for a try-block. The catching blocks evaluated sequentially one by one. If a catch-block catch the exception, the others will not be evaluated.

Example:

```
try {
    throw new NullPointerException( "This will be caught below" );
}
catch( Exception e ) {
    // --- The above NullPointerException will be caught by here --
}
catch( NullPointerException e ) { // --- THIS CODE IS NEVER EXECUTED -
    Compiler error
    // ---
}
```

Because `NullPointerException` is subclass of the `Exception` class. All `NullPointerException`s will be caught by the first catch-block.

Instead the above code should be rewritten as follows:

```
try {
    throw new NullPointerException( "This will be caught below" );
}
catch( NullPointerException e ) {
    // --- The above NullPointerException will be caught here ---
}
catch( Exception e ) {
    // --- Any other exception except the NullPointerException will be
    caught here --
}
```

Runtime Exceptions

The `java.lang.RuntimeException` exception class is inherited from `java.lang.Exception`. It is a special exception class, because catching this exception class or its subclasses are not enforced by the Java compiler.

runtime exception

Runtime exceptions are usually caused by data errors, like arithmetic overflow, divide by zero, Runtime exceptions are not business related exceptions. In a well debugged code, runtime exceptions should not occur. Runtime exceptions should only be used in the case that the exception could be thrown by and only by something hard-coded into the program. These should not be able to be triggered by the software's user(s).

NullPointerException

NullPointerException is a RuntimeException. In Java, a special **null** can be assigned to an object reference. NullPointerException is thrown when an application attempts to use an object reference, having the **null** value. These include:

- Calling an instance method on the object referred by a null reference.
- Accessing or modifying an instance field of the object referred by a null reference.
- If the reference type is an array type, taking the length of a null reference.
- If the reference type is an array type, accessing or modifying the slots of a null reference.
- If the reference type is a subtype of Throwable, throwing a null reference.

Applications should throw instances of this class to indicate other illegal uses of the null object.

```
Object obj = null;  
...  
obj.toString(); // --- This will throw a NullPointerException ---
```

The above code shows one of the pitfall of Java, and the most common source of bugs. No object is created and the compiler does not detect it. NullPointerException is one of the most common exceptions thrown in Java.

why do we need **null** ?

The reason we need it is because many times we need to create an object reference, before the object itself is created. Object references cannot exist without a value, so we assign the **null** value to it.

```
public Customer getCustomer()  
{  
    Customer customer = null;  
    try {  
        ...  
        customer = createCustomer();  
        ...  
    }  
    catch ( Exception e)  
    {  
        ...  
    }  
    return customer;  
}
```

In the above code we want to create the Customer inside the try-block, but we also want to return the object reference to the caller, so we need to create the object reference outside of the try-block, because of the scoping rule in Java. This is one of the pitfall of Java.