Error Handling

All possible errors in a program may not be detected at the compile time. For example, what happens if `numberOfItems` is zero in the following statement?

```
double unitPrice = totalPrice/numberOfItems;
```

You may make sure that `numberOfItems` is not zero by checking whether it is zero first as:

```
double unitPrice = 0;
if(numberOfItems!=0) unitPrice = totalPrice/numberOfItems;
```

This seems well. However, is it a good way of handling such a potential error? No! Because the statements following the above code may use `unitPrice` and produce a meaningless result if the `numberOfItems` is zero. So, it is better to stop executing the remaining statements which uses `unitPrice` and handle the case in any other context where necessary action could be taken. This may require enclosing the statements using the `unitPrice` in the if statement and providing an else part to handle the error. However, if more such errors may occur in the code, providing if-else statements for each may cause your code to become spaghetti code which may be difficult to understand and modify.
Exception handling mechanism provides a structured means of dealing with such errors in the programs. When such an error occurs, exception handling mechanism will return meaningful information about error and terminates the execution of statements until the error is handled by a special code called exception handler which enables you to perform certain actions before going further. This is done by creating an object containing the detailed information about the error including its type and the state of the program when it occurs. Then the runtime system finds a handler for the error and passes this object to the handler.

Try Block and Method Exception Specification

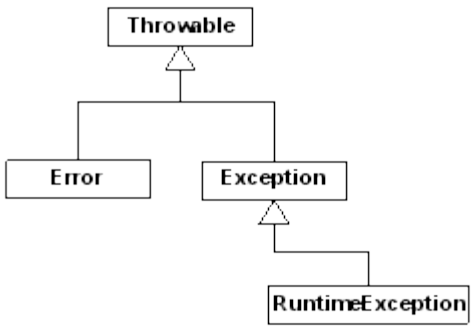
The simple exception handling includes specifying guard regions with `try` block and providing exception handlers with `catch` blocks. The exception object is supplied to the exception handling as the normal method parameter. The type of the exception object may simply be chosen as the `Exception` (as it will be explained shortly). This example illustrates the simplest way you deal with exceptions:

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

Exception handling mechanism allows you to separate program logic from error-handling to produce more clear and traceable code. It can also propagates errors up to the call stack. That is, if a method causes an error, the methods calling that method is searched for a handler. Therefore, you don't need to return complex error codes from the methods, deal with error codes returned from the method calls in the callers. The following code, illustrates this feature:

```
method1(){
     ...
     // an error occurs here, execution is stopped.
     ...
}
method2(){
     ...
     method1();
     // since method1 causes an error execution is stopped
here
     // because method2 does not have an error handler
     ...
}
method3(){
     ...
     try{
        method2();
     ...// these statements are not executed
     ...//since method2 causes an error
     } catch(Exception e) {
          ...
          // error is handled here - execution continues from
here
     }
     // execution continues normally
     ...
}
```

`Throwable` class is the superclass of all exceptions and errors in the Java Programming Language. Only objects of instances of `Throwable` or its subclasses may be thrown in a program. The high level hierarchy of classes for exceptions and errors is shown in the figure.



`Error` corresponds to serious errors (compile time or system), so you usully do not catch such exceptions. The standard exceptions thrown by any method are the instances of subclasses of the `Exception` class (or its subclasses as well). Most of the time, you deal with such exceptions.

The exception object's type name indicates the type of error which has occurred. Some of the exceptions are derived from `RuntimeException` class (e.g. `NullPointerException` – thrown when an application tries to use a `null` value in case an object is required) and some of them are derived directly from `Exception` class (e.g. `IOException` and its subclasses – thrown when an error occurred during input/output operations). For detailed information about the exceptions that may be thrown, please refer to the Java documentation.
In the Java Programming Language, you must specify which exceptions (except the exceptions handled in the method and exceptions of the type `RuntimeException` and its subclasses) may be thrown by a call to a method in method declaration. If you do not handle an exception in method body or specify its type in method declaration, the compiler checks your method implementation and forces you to specify potential exceptions in method specification. By this way, you inform the users of your method to know which kinds of exceptions they should deal with. For this purpose `throws` keyword can be used as:

```
void myMethod(...) throws ExceptionType1, ExceptionType2, …,
ExceptionTypeN {
    ...
}
```

Exception Handlers

Any exception thrown at runtime may be caught and handled in exception handlers. The exception handlers follow the `try` block and takes an exception object as the parameter. If your code may throw more than one kind of exception, you may discriminate between them by using more than one handlers as:

```
try{
    ... // some code which may throw exceptions
} catch(ExceptionType1 e1) {
    ... // handle ExceptionType1
} catch(ExceptionType2 e2) {
    ... // handle ExceptionType1
} catch(ExceptionType3 e3) {
    ... // handle ExceptionType1
}
    ... // execution continues from here after exception
handler is executed
```

If an exception occurs, the correct exception handler is found and executed according to the type of the exception object thrown. Note that exception handlers' order is important. If you handle a more general exception (a superclass) before a specific exception (its subclass), the remaining handlers will become meaningless and compiler warns you. If you want to handle all kinds of exceptions in one handler you may simply handle `Throwable` (as `Throwable` is the root superclass of all exception classes).
Each exception object thrown carries information about the type

of error occurred, state of the program at the time of error, and a small message (these features are inherited from `Throwable`). The exception object's `getMessage()`, `toString()`, and `getStackTrace()` may be used to get information about the error in the exception handler.

Finally Block

When you want to execute some code regardless of exception occurred or not, you may use `finally` block. The `finally` block is usually used to clean up before the execution continues after the try block and exception handlers (to release system resources allocated in `try` block, etc.).
The `finally` block can be added as:

```
try{
    ...
} catch(ExceptionType1 e1) {
    ...
} catch(ExceptionType1 e1) {
    ...
} finally {
    ... // always executed before execution continues with
the next statement
}
```

Creating and Throwing Exceptions

In order to throw an exception you may create an exception object from the existing exception classes and throw it using `throw` keyword as:

```
throw new ExceptionType("a message");
```

It is also possible to create your own exception classes if there is no suitable exception class to use. In order to create a new exception class you create a subclass of an existing exception class (it should at least be descendant of `Throwable`). The following code illustrates how to create a new exception class from `Exception` class:

```
class MyException extends Exception{
        MyException() {}
        MyException(String message) { super(message); }
}
```

Here, you may provide additional functionality in `MyException` class to give more information about the error occurred. After you declare `MyException` it may be thrown and catched as:

```
try{
    ...
throw new MyException("a message");
```

```
    ...
} catch(MyException e){
    ...
}
```