Exception Handling

Exception handling is a mechanism for making programs more robust, i.e. have them continue executing when errors, failures, or exceptional conditions occur, rather than crash.

This is especially important for embedded software such as a system controlling an airplane. But most applications should tolerate failures such as bad user input, e.g.

You can catch and handle such exceptions using the try-catch construct, e.g.

```java
import java.util.Scanner;
import java.util.NoSuchElementException;
public class EMEgA{
    public static void main(String[] args){
        Scanner in = new Scanner(System.in);
        System.out.println("Enter student number & name separated by ;");
        String line = in.nextLine();
        Scanner in1 = new Scanner(line).useDelimiter(";");
        String nt = in1.next();
        // may throw NoSuchElementException
        int sno = Integer.parseInt(nt);
        // may throw NumberFormatException
        String sname = in1.next();
        // may throw NoSuchElementException
        System.out.println("name = " + sname);
        System.out.println("number = " + sno);
    }
    catch(NumberFormatException e){
        // handler for NumberFormatException
        System.out.println("Invalid number");
    }
    catch(NoSuchElementException e){
        // handler for NoSuchElementException
        System.out.println("Not enough arguments");
    }
}
```
The finally clause is used to indicate that some statements must be executed even when an exception is thrown, usually to do some cleaning up operations or ensure that an object’s state is consistent. E.g.

```java
import java.io.FileReader;
import java.io.IOException;
import java.util.Scanner;
import java.util.NoSuchElementException;
```

When an exception is thrown, e.g. `Integer.parseInt` throws `NumberFormatException`, the execution of the try block is aborted and the corresponding catch handler block is executed.

If there is no matching catch clause, the exception propagates out of the try block and may be caught by an outside try block. If the exception is never caught, the program’s execution will terminate abnormally.

You can also throw exceptions yourself using the throw statement, e.g.

```java
throw new RunTimeException("Invalid CallZone");
```

```
public class EHegB
{
    public static void main(String[] args) throws IOException
    {
        FileReader r = new FileReader("sinput.txt");
        // may throw FileNotFoundException
        Scanner in = new Scanner(r);
        try
        {
            for(String inp = in.nextLine(); in.hasNextLine();)
            {
                inp = in.nextLine();
                String inln = in.nextLine();
                int sno = Integer.parseInt(nt);
                String name = inln.next();
                System.out.println("number= " + sno);
            }
        }
        catch(NumberFormatException e)
        {
            System.out.println("Invalid number");
        }
        catch(NoSuchElementException e)
        {
            System.out.println("Not enough arguments");
        }
        finally
        {
            r.close(); // may throw IOException
        }
    }
}
```
If a Throwable th is thrown somewhere in { statements_try }:
the block is immediately exited, and then we execute

```java
if(th instanceof C1)
{ statements_C1
} else if(th instanceof C2)
{ statements_C2
} else if(th instanceof C3)
{ statements_C3
}...
```

followed by { statements_finally }.

The finally block is always executed, whether an exception is thrown or not, and if one is, whether it is caught or not.

try-catch Control Flow

```java
try
{ statements_try
} catch(C1 th)
{ statements_C1
} catch(C2 th)
{ statements_C2
} catch(C3 th)
{ statements_C3
}... finally
{ statements_finally
}
```

In normal control flow, when nothing is thrown in { statements_try }:
execute all of { statements_try }
and then all of { statements_finally }.

Classes C1, C2, ... must be subclasses of Throwable.

Make sure you import C1, C2, ....

Order the catches as you would order if's.

Can intentionally create and throw exceptions.

The finally block is always executed:

- after a normal try,
- after a normal catch,
- after a throw in a catch handler,
- after an uncaught throwable.

Often better to put finally with a separate try.
Checked and Unchecked Exceptions

Exceptions and errors are organized into a hierarchy whose root is the Throwable class (see Horstmann p. 554).

All exceptions except instances of RuntimeException are checked (by the compiler).

When you call a method that may throw a checked exception, your program must say what it will do if the checked exception is thrown, either catch it or declare that it may throw it. E.g. IOException in EHegB, or the following

```java
import java.io.FileReader;
import java.io.IOException;
import java.util.Scanner;
import java.util.NoSuchElementException;
```

Checked exceptions are due to external circumstances that programmer cannot control, e.g. FileNotFoundException; they must either be caught or a declaration that the exception can be thrown must appear.

Unchecked exceptions are usually due to preventable errors by the programmer, e.g. NoSuchElementException thrown by next() of Scanner; programmer should have called hasNext() to check first. It is better to fix this problem so that the exception is never thrown.

```java
public class EHegB {
    public static void main(String[] args) {
        try {
            FileReader r = new FileReader("inut.txt"); // may throw FileNotFoundException
            Scanner in = new Scanner(r);
            try {
                for(String inp = in.nextLine(); in.hasNextLine();)
                    inp = in.nextLine();
                String nt = inp.next();
                int sno = Integer.parseInt(nt);
                String name = inp.next();
                System.out.println("name = " + name);
                System.out.println("numbers = " + sno);
            }
            catch(NumberFormatException e) {
                System.out.println("Invalid number");
            }
            catch(NoSuchElementException e) {
                System.out.println("Not enough arguments");
            }
            finally {
                r.close(); // may throw IOException
            }
            catch(IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

To set the message attribute of an exception, use the 1 argument constructor, e.g.

```java
new RuntimeException("too few arguments").
```

To retrieve the message attribute of an exception:

```java
e.getMessage();
```

To print a stack trace:

```java
e.printStackTrace();
```
E.g. catching an exception that we threw:

```java
import java.util.Scanner;
public class EHegD
{
    public static void main(String[] args)
    {
        String prog = null;
        int no = 0;
        String term = null;
        Scanner in = new Scanner(System.in);
        while(!inputDone)
        {
            System.out.println("Enter course, e.g. ITEC 1630 W");
            try
            {
                String input = in.nextLine();
                int sep1 = input.indexOf(" ");
                int sep2 = input.indexOf(" ",sep1+1);
                prog = input.substring(0,sep1);
                no = Integer.parseInt(
                    input.substring(sep1+1,sep2));
                term = input.substring(sep2+1);
                if(!term.equals("F") && !term.equals("W")
                    throw new IllegalArgumentException("Incorrect term code");
            }
            catch(NumberFormatException e)
            {
                System.out.println(e.getMessage());
            }
            catch(IndexOutOfBoundsException e)
            {
                System.out.println("Missing field in input");
            }
            catch(NullPointerException e)
            {
                System.out.println("Incorrect format for course number");
            }
            catch(IllegalArgumentException e)
            {
                System.out.println(e.getMessage());
            }
            System.out.println(prog + ":" + no + "":" + term);
        }
    }
}
```

One can use throw and catch to exit from several nested loops, e.g.

```java
import java.util.Scanner;
public class EHegD
{
    public static void main(String[] args)
    {
        try
        {
            boolean inputDone = false;
            String prog = null;
            String term = null;
            Scanner in = new Scanner(System.in);
            while(!inputDone)
            {
                System.out.println("Enter course, e.g. ITEC 1630 W"," "+" blank line to exit");
                try
                {
                    String input = in.nextLine();
                    if(input.equals(""))
                        throw new Throwable();
                    int sep1 = input.indexOf(" ");
                    int sep2 = input.indexOf(" ",sep1+1);
                    prog = input.substring(0,sep1);
                    no = Integer.parseInt(
                        input.substring(sep1+1,sep2));
                    term = input.substring(sep2+1);
                }
            }
        }
    }
```
import java.util.Scanner;

public class EHege {
    public static void main(String[] args) {
        while(true) {
            boolean inputDone = false;
            String prog = null;
            int no = 0;
            String term = null;
            Scanner in = new Scanner(System.in);
            while(!inputDone) {
                System.out.println("Enter course, e.g. ITEC 1630 W, blank line to exit");
                try {
                    input = in.nextLine();
                    if(!input.equals("")) {
                        int sep1 = input.indexOf(" ");
                        int sep2 = input.indexOf(" ", sep1+1);
                        prog = input.substring(0, sep1);
                        no = Integer.parseInt(input.substring(sep1+1, sep2));
                        term = input.substring(sep2+1);
                    } else {
                        System.out.println(prog + ", " + no + ", " + term);
                    }
                } catch(IndexOutOfBoundsException e) {
                    System.out.println("Missing field in input");
                } catch(NumberFormatException e) {
                    System.out.println("Incorrect format for course number");
                } catch(IllegalArgumentException e) {
                    System.out.println(e.getMessage());
                } catch(Throwable t) {
                    System.out.println(t.getMessage());
                }
            }
        }
    }
}

But this is not recommended. Same e.g. without using throw for deep exit:

if(!term.equals("F") && !term.equals("W")) {
    throw new IllegalArgumentException("Incorrect term code");
} 
inputDone = true;

if(!term.equals("F") && !term.equals("W")) {
    throw new IllegalArgumentException("Incorrect term code");
} 
inputDone = true;

if(!term.equals("F") && !term.equals("W")) {
    throw new IllegalArgumentException("Incorrect term code");
} 
inputDone = true;

if(!term.equals("F") && !term.equals("W")) {
    throw new IllegalArgumentException("Incorrect term code");
} 
inputDone = true;

if(!term.equals("F") && !term.equals("W")) {
    throw new IllegalArgumentException("Incorrect term code");
} 
inputDone = true;
Defining Your Own Exception Classes

Sometimes standard exception classes are not very appropriate to your application. Then you can design your own. E.g.

```java
public class IllegalTermCodeException extends IllegalArgumentException {
    public IllegalTermCodeException() {
    }
    public IllegalTermCodeException(String message) {
        super(message);
    }
}

if (!term.equals("F") && !term.equals("M"))
    throw new IllegalTermCodeException(term + " is not a legal term code");
```

Generally, one defines 2 constructors, one that takes no arguments and one that takes a message string as argument.

It is best to use exceptions that are as specific as possible.