CS2

Theme:

Object Oriented Programming in Java

Handout : Week6

prepared by N S kumar

Preface

Dear friends,

Looks like, it is time to enjoy. There seems to be some Science Festival somewhere round the corner. Aatmatrisha is beckoning you all. I am sure you should enjoy life. Please do enjoy programming also. Believe me. Programming is very intoxicating!

This week, we discuss two major topics – Exceptions and Enumerated data type.

FAQ is still not yet ready. Should be ready by next week. I invite you to trouble me with more questions.

All the best.

N S Kumar
Visiting Professor
Dept. Of Computer Science and Engineering
PES Institutions.
Email : ask.kumaradhara@gmail.com
Date : 22nd February 2015

Enumerated data type:

Your Math teacher would have told you that the concept of set in Mathematics is undefined. We play with sets to make our life simple. We either list out all the values in a set or we specify a rule to generate the elements of the set. The former is called the enumeration and the latter is called the set builder rule.

There are many contexts where the number of instances is fixed – is like a set with a finite number of elements. Dasharatha had 4 Children (Rama, Lakshmana, Bharatha and Shatrugna). Dritarashtra is supposed to have had 100 Children – do not ask me to recite all their names! Not sure whether Gandhari could do it.

Java allows you to create a collection (has number of elements) of fixed number of objects with the enum type. Enum type acts like a class – which has a fixed number of objects.

- enum is like a class – a special class. Enum stands for enumeration.
- These objects can have their own attributes.

Rama : name of his wife(one and only one -ekapatni vratasta), children ...

- They can be initialized by a private constructor. So, the user of the class cannot create objects of this type. The set is closed – cannot add or remove objects. We can refer to them.
- We can access all these objects by using the method called values. This method is called on the class name. This method on the enum type will return us all the objects in that class.
- We can have any number of methods in the enum class. However we can not create new objects or remove existing objects. We can modify the existing objects if we have an interface to do so.

We may have a method to find the wife's name. We may change somebody's wife if the class allows! (just joking!)

- We can compare the objects. There is an implicit ordering. Element on the left is smaller than the one on the right.

A few examples follow.
In 'C' programming, we use this concept of enum to create a number of named constants. Java provides a richer mechanism of creating a number of named objects.

In this example, we create a class called Colour having 3 objects named RED, GREEN and BLUE. These three objects are created when the class is loaded.
We can not create any new object of type Colour.

We can make a reference which will refer to one of the objects in the class.
The method called values on the class Colour returns all of the objects of the class. The attribute called length on the return of this method values tells us the number of objects in the enum class.

We can walk through them using a loop – called enhanced for loop. (Sorry for the forward reference – we will learn more about this in the topic on arrays).
When we display the object, it will display its name. Is it not the most important thing for anybody?

When I was young like you(I am still young!), I could answer most of the puzzles. This one had fooled me!
Indira Gandhi is Jawaharlal Nehru's daughter's _____ . (Fill up the blanks).

```
// L20/d1/Colour.java
public enum Colour
{
        RED, GREEN, BLUE;
}
```

```
// L20/d1/Client.java
// enum:
//      far different compared to 'C'
//      is a collection of objects
//      number of objects is fixed
//      cannot create a new object
//      can  have attributes
//      constructor is private

public class Client
{
        public static void main(String[] args)
        {
        //      Colour c1 = new Colour(); // NO
                Colour c1 = Colour.RED;
                System.out.println(c1);
        //      System.out.println(Colour.values().length); // ok
                for(Colour c : Colour.values())
                {
                        System.out.println(c);
                }
        }
}
```

In the last example, we had 3 objects(RED, GREEN, BLUE) – but no specific attributes. Let us consider an example with attributes. We will make a list of subjects with two attributes – names of the authors of the standard book for that subject and its price. In this example, names of authors are correct and the price is a guess – Please do not come to me either to sell your books or buy books for you based on this price!

Observe the following.

- The class has two attributes – author and price
- The listings of objects are actually constructor calls
- The constructor does not have the access specification. They are private.

(Experiment: check javap; change the constructor to public)

- The class has an interface called disp which the client can call on any object of this class

```
//L20/d1/Books.java
public enum Books
{
	C("Kernighan & Ritchie", 200),
	JAVA("Horstman", 600),
	MATH("Erwin Kreyszig", 500);
	Books(String author, int price)
	{
		this.author = author;
		this.price = price;
	}
	private String author;
	private int price;
	public void disp()
	{
		System.out.println(author + ":" + price);
	}
}
```

```
//L20/d1/Client.java
// enum:
public class Client
{
	public static void main(String[] args)
	{
		Books mybook = Books.JAVA;
		mybook.disp();
		for(Books b : Books.values())
		{
			b.disp();
		}
	}
}
```

Exercise:

1. Create a enum class to represent the course of 2nd semester. The objects are named MATH, SPECIALTOPIC, CS2, ... Let the class have fields – theme, number of credits, number of lecture hours.

a) Find the number of objects in this class

b) Provide a method to get the theme. Test it in the client program.

c) Provide a method to set the theme. Test it in the client program.

Life and Scope:

You may remember we said that a variable has the following attributes.

1. Name
2. Location
3. type
4. value

A variable has a couple of interesting attributes.

5. Life
6. Scope

We say that the variable has LIFE if it has an associated memory – has location. If there is no location, the variable is dead. We say that the variable has SCOPE if it is accessible.

LIFE:

1. The variables could be local – declared with in a method. These variables are created on the stack (Last In First Out – LIFO). When the method completes its work(when the method returns), these variables are removed. If the method calls itself – recursion – each time the method is called, a new local variable is created.
2. The variables could be part of an object – field of the class – declared within the class. created when we call new – they die when the object dies.
3. The variables could be part of the class – shared with all the objects – declared within the class as static. These are created when the class is used for the first time. They remain through out the program from this point onwards.

SCOPE:

A variable can have scope only if the variable has life. **NO SCOPE WITHOUT LIFE.** The scope is affected by access control.

1. A local variable is available in that method only. We cannot declare a local variable in an inner block with the same name. (You may want to check what happens in 'C'. It is different there). A variable declared in an inner block – cannot have a clash with the variable declared in the outer block – lives in that block only and therefore has scope only in that block. The variable declared within a for-loop is not visible outside of the for loop. Two for-loop one following the other can have the same local variable – without any clash.
2. A variable declared in a class can have access specification. So far, we have used two access specifications – public and private.

We can access **public** members anywhere – using an object of the class for instance variable or using an object or class name for static variable.
We can access **private** members only within the methods of the class – using an object of the class for instance variable or using an object or class name for static variable.
Please note that the access is based on the type(class name) and not an instance of a class. Within a method of the class, we can access any member of any object of the same class.

```
//L20/d3/Client.java
// Scope
public class Client
{
	public static void main(String[] args)
	{
		int a = 10;
		{
			// both a and b exist
			// cannot redeclare a (in java)
			// int a = 100; // ERROR
			int b = 20;
			System.out.println("inner block : " +
				"a : " + a + " b : " + b);
		}

		// b does not exist
		System.out.println("outer block : a : " + a);
```

```
            Test t1 = new Test();
            {
                  Test t2 = new Test();
                  // t1 exists; t2 exists;
                  t1 = null;
                  // t1 exists; does not point any object
                  // whatever t1 referred earlier has become
                  //       garbage!!
            } // t2 dies; what t2 refers is now garbage
      }
}

class Test
{
      public int t = 111;
}
```

State:

An object of a class has a number of attributes. These attributes have certain values at a given point during the execution of the program. This set of values of an object defines what is called a state. The state of the object decides how the object behaves when it is passed a message. When we call a method using an object, what happens depends on the state of the object. You would know when to request (or is it an order?) your father for extra pocket money or to allow you to go for a college trip. I am sure you would not ask him when he is tired or angry for whatever reason. A lift moves up and moves down, allows opening and closing of door. If a moving lift and not a stationery lift caters to the “open door” command, we may end up having lots of freebodies (I am not talking about statics!) in the air!

This is a simple example illustrating the concept of a state. A phone can be in different states with respect to receiving a call. It may be active(can ring), may be silent or may be in vibratory mode. There will be some default mode when the object is created. The state can be changed. The phone reacts on a call based on its state.

```
//L20/d4/Phone.java
public class Phone
{
	private int phoneNumber;
	private PhoneState phoneState;
	public Phone(int phoneNumber)
	{
		this.phoneNumber = phoneNumber;
		this.phoneState = PhoneState.ACTIVE;
	}
	public void disp()
	{
		System.out.println(phoneNumber + ":" + phoneState);
	}
	public void changeState(PhoneState phoneState)
	{
		this.phoneState = phoneState;
	}
```

```
	public void call()
	{
		switch(phoneState)
		{
			case ACTIVE :
					System.out.println("hello");
					break;
			case SILENT :
					System.out.println("Sh....");
					break;
			case VIBRATORY :
					System.out.println("...---...");
					break;
		}
	}

enum PhoneState
{
	ACTIVE, SILENT, VIBRATORY;
}
```

An object has a state. A class also will have a state only if it has static members.

```
//L20/d4/Client.java
public class Client

{
	public static void main(String[] args)
	{
		Phone phone = new Phone(12345678);
		phone.call();
		phone.changeState(PhoneState.SILENT);
		phone.call();
		phone.changeState(PhoneState.VIBRATORY);
		phone.call();
		phone.changeState(PhoneState.ACTIVE);
		phone.call();
	}
}
```

Excercise:

The Placement cell selects the students based on the CGPA and the grade in CS2. The student class shall have usn, name, CGPA (a double less than or equal to 10.0) and the grade (a character S A B C D F), selectionState. Fill the selectionState as selected (CGPA >= 8.5 or S grade in CS2), waitlisted (CGPA between 7.0 and 8.5 or A grade in CS2) or rejected(rest are rejected).

Assertions:

When we are studying, we will have some check points. After reading for sometime, we check whether we have learnt what we should have. In many online tutorials, there will be some small quiz to check whether the participants are on the right track. If we do not do that, then we will end up asking the relation between Rama and Seetha after listening to the story of Ramayana the whole night.

How do we define a rational number? We could say that a number of the form p / q where p and q are integer, is a rational number. NO. This is wrong. We should ascertain that the value of q is not zero.

In programming, the values of variables should satisfy certain conditions at certain point during the execution. For some reason(most probably because of sloppy programming), this may not happen. We can check this out using the concept of Assertions. We have this concept in 'C' as well. You should experiment with them in 'C' as well.

An assertion has a boolean expression (we call it a predicate) which is expected to be true. If it is not true, the program aborts at that point giving an AssertionError. We may also have an expression following the assertion which will be displayed as a string.

We use assertions for three different cases.

- Before a particular statement(most probably before a loop), we expect a certain condition to be true. Such a condition is called a **Pre-Condition.**
  Argument to the square root method should be a non negative real number.
  To find the gcd of two numbers, none of the two numbers should be zero.
- After a particular statement(most probably after a loop), we expect a certain condition to be met. Such a condition is called a **Post-Condition**.
  The return of the square root method squared should be very close to the

argument given for the square root method.

When the gcd algorithm in its original form is used, the changed arguments should be same and both should divide into the given numbers.

- During the execution of a loop, we expect a certain condition to be always true. Such a condition is said to be an **invariant**. This could be a property of a loop, a function or a data structure or an algorithm.

This can be used a consistency check. The relation between the grade of the student and his marks is an invariant. When you get more marks, your grade should change suitably.

```
//L21/d1/Assert1.java
//      assertion : to catch programming bugs
//                  to check pre-condition, post-condition,
//                  invariants
//              can be enabled or disabled at runtime
public class Assert1
{
        public static void main(String[] args)
        {
                Scanner scanner = new Scanner(System.in);
                double n;
                n = scanner.nextDouble();
                // Two forms of assertion;
                //      second is considered better
                // check whether the number is positive

                // 1. assert n >= 0;
                // 2. assert n >= 0 : "number is negative";
                assert n >= 0 : "number is negative";
                System.out.println("sqrt : " + Math.sqrt(n));
        }
```

```
}



//L21/d1/Assert2.java
 // assert:
public class Assert2
{
	public static void main(String[] args)
	{
		Rect r = new Rect(10, 20);
		r.setLength(10);
		System.out.println("thats it");
	}
}
class Rect
{
	private int length;
	private int breadth;
	private boolean isSquare;
	public Rect(int l, int b)
	{
		length = l; breadth = b; isSquare = l == b;
	}
	public void setLength(int l)
	{

		length = l;
		// is a contract; is the understanding
		assert isSquare != (length == breadth)
			: "field not set correctly";
	}
```

}

In the above example, when the length is changed, we must check whether the isSquare member is correct. To be precise, this method is wrong as it does not change the isSquare attribute, when the length is changed.

Exception :

A failure in an assertion is associated with something wrong within our program. So we should fix it.

Let us say you want to get a cotton shirt – a colourful one – holi should be round the corner – stitched. You go to a tailor and give your cloth. The cloth has to be washed before stitching. That is the normal procedure. Normally there is no problem. The tailor stitches a shirt – gives it to you to wear for holi – then you can make your shirt very colourful.

But what should the tailor do if the cloth shrinks a lot. Can he decide to go ahead and stick a shirt with no sleeves (or sleeves with no shirt!)? Can he make a few hand kerchiefs? Can he make a shirt for your kitten? There is no way he can decide. He will have to ask you.

What to do when something unusual happens? Murphy manifests always. (something will go wrong). If you know how to take care of it, you should. If you are providing a service to somebody, you cannot decide. How about the tailor throwing the cloth into the dust bin? We would like to recover and do what best we can. We would like the client to decide how to recover.

These happen in programing as well. Most of the times our programs run very well. (we are all students of Science – should we touch wood?). Can we recover from something unusual?

We want to have “Graceful Degradation”. We want to reach a safe state and continue from there. For that we use a concept called Exception.

There was a couple (– quarrelsome – that is normal and not an exception). The husband was complaining that the wife was spending lots of money on items which were rarely used. Wife also got a chance to revert. She told her husband that the husband had brought an article which had not been used for a year. That item was a Fire Extinguisher! The couple had got no chance to use this for one year!

In the example given below, we are accessing an element of an array beyond the array size. The elements of the array are numbered from 0 onwards. (We shall learn about arrays later. You may relate to the arrays you have learnt in 'C' so far). When we index the array outside the array bounds, Java runtime tells us that something unusual has happened. It will abort the program.

By the way, how many of you have read the 19th Chapter of Bhagavadgeetha?
How many of you have visited the 9th floor of F block(called Panini Block) in our college?

Exception:

```
//L21/d2/Client.java
// Exception:
// not normal
// something unusual
// do not know how to take care of it where it happens
// should inform the user that something unusual has happened
// force the user that he should take care of it
//       if he does not, program should abort
//       otherwise move to a safe state and proceed
//                called graceful degradation
// program has a runtime error; exception
//       ArrayIndexOutOfBoundsException
import java.io.*;
```

```
import java.util.*;
public class Client
{
	public static void main(String[] args)
	{
		// we are making an array
		// not exactly the way it is in 'C'
		int a[] = {11, 22, 33, 44};
		System.out.println("elem : " + a[4]);
	}
}
```

When we expect something unusual might happen, we should be ready for it. We should keep a fire extinguisher (even if your spouse complains!). You should not dig a well when your house is on fire(na koopa khananam yuktam pradipte vahnine grihe – says a Subhashita in Sanskrit).

We do that using a **try block.** This has to be followed by mechanisms to take care of something unusual. We may have to have different ways to douse the fire (water if oil is not fire, dand if oid on fire ...). For that we require a number of **catch blocks.**

A bowler in cricket tries to get a batsman out. The batsman does not normally get out unless he makes a mistake. To pounce on his mistake, we will have a number of catchers – one for the edge, one for a lofted shot on the boundary and so on.

Try and a number of catch blocks together form a single unit.

I want to you check the flow of execution on normal flow and on an exceptional flow.

Normal flow: try block – skip all catch blocks – rest of the code

Exceptional flow: try block – exit from the middle of the try block – one of the catch blocks – rest of the code. We will not resume at the point of exception. We will not come back to the statement following the one where something unusual happens,

when an exception thrown is caught in the try block, an exception object of a particular type is created. Displaying the object will indicate to us what went wrong and where it happened. We can extract lots of information from this exception object.

//L22/Exception1.java

```
// Exception:
import java.io.*;
import java.util.*;
public class Exception1
{
	public static void main(String[] args)
	{
		Scanner scanner = new Scanner(System.in);
		try
		{
			int a[] = {11, 22, 33, 44};
			int i;
			i = scanner.nextInt();
			System.out.println("hattu");
			System.out.println("elem : " + a[i]);
			System.out.println("ippattu");
		}
```

```
            catch(ArrayIndexOutOfBoundsException e)
            {
                System.out.println(e);
                System.out.println("moovattu");
            }
            System.out.println("nalvattu");
        }
}
```

In the following example, there are two possible exceptions.

- Index out of bounds
- Division by zero.

Should we take care of each of these cases separately? Can we have common way of handling these cases? For a botanist, each tree in Lalbagh is different. For me – who understand some other type of tree and not a botanical tree , all trees are same!

There is a catch all handler (There was one player in Zimbabwe who could bat, bowl and he was also the wicket keeper for his team. I think he was also the captain for some time. Not sure whether he was also the manager. I do not get his name).

It is called Exception. But the order of catch blocks matter. Unlike normal method calls, when an exception is thrown, it tries to match the catch blocks in the order of catch blocks – is not the best match, but the first match.

We cannot have the most generalized handler before specialized handler.

The doctors normally give specific medicine if they can find out your illness. Otherwise they fall back on generic medicine which could cure any illness you have.

So, the program below will not compile. The catch for index out of bound should appear before the handler for generic exception called Exception.

```
//L22/Exception2.java
// Exception:
import java.io.*;
public class Exception2
{
	public static void main(String[] args)
	{
		Scanner scanner = new Scanner(System.in);
		try
		{
			int a[] = {11, 22, 33, 44};
			int i;
			i = scanner.nextInt();
			System.out.println(10/i);
			System.out.println("hattu");
			System.out.println("elem : " + a[i]);
			System.out.println("ippattu");
		}
		// compile time error; order matters!
		catch(Exception e)
		{
			System.out.println("catch all handler");
		}
```

```
            catch(ArrayIndexOutOfBoundsException e)
            {
                System.out.println(e);
                System.out.println("moovattu");
            }
            System.out.println("nalvattu");
        }
}
```

Multiple catch Blocks:

You may experiment and find that the index out of bounds is caught by the corresponding catch block and the division by zero(called ArithmeticError) is caught by the generalized handler.

```
// Exception:
// show example with multiple catch block
// catching any exception with Exception
// order of exception handlers
//      not the best match
//      it is the first match

//L22/Exception3.java
// Exception:
import java.io.*;
import java.util.*;
```

```
public class Exception3
{
	public static void main(String[] args)
	{
		Scanner scanner = new Scanner(System.in);
		try
		{
			int a[] = {11, 22, 33, 44};
			int i;
			i = scanner.nextInt();
			// ArithmeticException if i == 0
			System.out.println(10/i);
			System.out.println("hattu");
			System.out.println("elem : " + a[i]);
			System.out.println("ippattu");
		}

		catch(ArrayIndexOutOfBoundsException e)
		{
			System.out.println(e);
			System.out.println("moovattu");
		}
		catch(Exception e)
		{
			System.out.println("catch all handler");
			System.out.println("exception : " + e);
		}
		System.out.println("nalvattu");
	}
}
```

Finally Block:

Many events have a pair of actions. A person who is born eventually dies. We are not counting unusual cases – (ever alive – cheeranjeevi). Jaatasya maranam dhruvam – saying in Sankrit. We begin a course(CS2). It ends sometime in April (I can imagine your reaction - what a relief!). In programming, we start something – we say we acquire a resource. We end it. We say we release the resource. A good example for you at this point could be – open a file and then close the same file.

Where should we put this code? We put the beginning code under a try block. We might want to put the ending code after all catch blocks.

But we have a catch here. What if one of the catch blocks itself throws an exception? Then the code following the try-catch block will never be executed. Then we will end up creating a cheeraayushi – ever living – believe me - that is a curse – even in programming.

Can we have some code which will always be executed -no matter what happens? That is called a Finally block.

Finally block will always be executed – on normal or exceptional flow.
We should put the cleanup operations in the finally block.

```
//L22/Exception4.java
// Exception:
import java.io.*;
import java.util.*;
```

```
public class Exception4
{
	public static void main(String[] args)
	{
		Scanner scanner = new Scanner(System.in);
		Scanner myfile  = null; // initialize
		try
		{
			int a[] = {11, 22, 33, 44};
			int i;
			// test with or without an existing file
			// resource creation
			myfile = new Scanner(new File("junk1.txt"));
			i = scanner.nextInt();
			// ArithmeticException if i == 0
			System.out.println(10/i);

			System.out.println("hattu");
			System.out.println("elem : " + a[i]);
			System.out.println("ippattu");
		}
		catch(IOException e)
		{
			System.out.println("io error : " + e);
		}
		catch(ArrayIndexOutOfBoundsException e)
		{
			System.out.println(e);
			System.out.println("moovattu");
		}
```

```
            catch(Exception e)
            {
                  System.out.println("catch all handler");
                  System.out.println("exception : " + e);
            }
            finally
            {
                  // put the code which should be always
                  //      executed
                  System.out.println("always executed");
                  // if file has been opened close it
                  if(myfile != null)
                  {
                        myfile.close();
                  }
            }
            System.out.println("nalvattu");
      }
}
```

Exception propagation:

The try block is alive in all methods called from within it. In this example, the main method calls foo – foo calls bar – something might be not normal in bar. Then the control will directly come back to the catch blocks of the try block from within which foo was called.

Assume that India wins the world cup. You want a holiday to be declared in the college. You ask your teacher. She cannot answer. She will pass the question to the head of your department. He may not be able answer this question. He will pass the question to the principal or the vice Chancellor or somebody in the management who can answer that. That is what happens here.

```
//L22/Exception5.java
// Exception:
import java.io.*;
import java.util.*;
public class Exception5
{
	public static void main(String[] args)
	{
		Scanner scanner = new Scanner(System.in);
		Scanner myfile  = null;
		try
		{
			System.out.println("calling foo");
			foo();
			System.out.println("called foo");
		}
		catch(ArrayIndexOutOfBoundsException e)
		{
			System.out.println(e);
			System.out.println("indexed beyond");
		}
		catch(Exception e)
		{
			System.out.println("catch all handler");
			System.out.println("exception : " + e);
		}
		System.out.println("the end");

	}
```

```
	private static void foo()
	{
		System.out.println("calling bar");
		bar();
		System.out.println("called bar");
	}
	private static void bar()
	{
		System.out.println("in bar");
		int i;
		int a[] = {11, 22, 33, 44};
		Scanner scanner = new Scanner(System.in);
		i = scanner.nextInt();
		// ArithmeticException if i == 0
		System.out.println(10/i);
		System.out.println("after diviision");
		System.out.println("elem : " + a[i]);
		System.out.println("after indexing");
		System.out.println("out of bar");
	}
}
```

When the exception is propagated, that exception object conceptually remembers who started this chain of events. We can get that by calling a method called printStackTrace on the exception object. It gives the total context – which methods were called – which line numbers – what is the order of propagation.

You may remember answering questions on 'reference to context in langauge courses - “the most unkindest cut of all”'.

```
//L22/Exception6.java
// Exception:
import java.io.*;
import java.util.*;
public class Exception6
{
	public static void main(String[] args)
	{
		Scanner scanner = new Scanner(System.in);
		Scanner myfile  = null;
		try
		{
			System.out.println("calling foo");
			foo();
			System.out.println("called foo");
		}
		catch(ArrayIndexOutOfBoundsException e)
		{
			System.out.println(e);
			System.out.println("indexed beyond");
			// Gives the whole history;
			// who called whom; which file; while line
			e.printStackTrace();
		}
		catch(Exception e)
		{
			System.out.println("catch all handler");
			System.out.println("exception : " + e);
		}
		System.out.println("the end");
	}
```

```
	private static void foo()
	{
		System.out.println("calling bar");
		bar();
		System.out.println("called bar");
	}
	private static void bar()
	{
		System.out.println("in bar");
		int i;
		int a[] = {11, 22, 33, 44};
		Scanner scanner = new Scanner(System.in);
		i = scanner.nextInt();
		// ArithmeticException if i == 0
		try
		{
		System.out.println(10/i);
		System.out.println("after diviision");
		System.out.println("elem : " + a[i]);
		System.out.println("after indexing");
		}
		catch(ArithmeticException e)
		{
			System.out.println(e);
		}

		// exception handled everything fine here onwards
		System.out.println("out of bar");
	}
}
```

Handling resources:

We have already said the programs have to take care of allocating the resource (like open a file) and deallocating the resource(like closing the file).

This is an example where that dichotomy is clearly expressed.

A file is opened in the try block. The same file is closed in the finally block.

```
//L23/Exception1.java
import java.io.*;
import java.util.*;
public class Exception1
{
	public static void main(String[] args)
	{
		Scanner myfile  = null;
		try
		{
			myfile = new Scanner(new File("junk1.txt"));
			System.out.println("file opened");
		}
		catch(IOException e)
		{
			System.out.println("exception : " + e);
		}
		finally
		{
			System.out.println("closing the file if opened");
			if(myfile != null)
			{
				myfile.close();
			}
		}
		System.out.println("the end");
```

```
	}
}
```

We tend to forget reclaiming the resource at the end. Can that responsibility be transferred to the java runtime itself? Can we expect that when you have your snacks – chocolates, chips or whatever junk – you will also clean the place? We do not know about whether you will do, but we can make java do it for us.

Try-with-resource :

- create resources(objects of some class) within a pair of parentheses following try.
- These objects should support a method called close
- When these objects die, the close method will be called by Java runtime

```
//L23/Exception2.java
import java.io.*;
import java.util.*;
public class Exception2
{
	public static void main(String[] args)
	{
		try(Scanner myfile = new Scanner(new File("junk.txt"))
		{
			System.out.println("in try block");
		}
		catch(IOException e)
		{
			System.out.println("exception : " + e);
		}
		System.out.println("the end");
	}
}
```

```
// Exception:
// try with resource
// file opened as part of the try
// will be closed automatically when the block is exited
//       in any which way!
// cleaner code; readable; guaranteed that the file will be closed
```

// Please note these topics have not been covered so far. We may cover these later.

```
// These as of now are for those who are brave!
```

Throw an exception :

In the examples so far, the exception was thrown by java runtime itself. We can also throw an exception object. Here is such an example.

This example checks whether a file exists – if not throws an IO Exception.

```
//L23/Exception3.java
import java.io.*;
import java.util.*;
public class Exception3
{
        public static void main(String[] args)
        {
                File myfile  = null;
                try
                {
                        myfile = new File("junk1.txt");
                        if(! myfile.exists())
                        {
                                throw new IOException("no such file");
                        }
```

```
            }
            catch(IOException e)
            {
                  System.out.println("exception : " + e);
            }
            System.out.println("the end");
      }
}
//    we can throw a builtin exception or our own exception
//    The second concept is not being covered in this course
```

Checked and Unchecked Exception :

The exceptions are of two kinds.

Unchecked Exception :

Those are under the control of the programmer. If the programmer is careful, these can be avoided. We do not have to tell the user of my method and I can take care of it myself.

Checked Exception :

Those not under the control of the programmer. What can we do if the file reading fails? This is external to my code. But this could happen. Either we should provide remedy for it or tell the user of my method that something could go wrong if you use my code. Remember the list of side effects most of the medicines indicate!

```
//L23/Exception4.java
import java.io.*;
import java.util.*;
public class Exception4
{
	public static void main(String[] args)
	{
		foo();
	}
	private static void foo()
	{
		int[] a = {11, 22, 33, 44};
		Scanner scanner = new Scanner(System.in);
		int i = scanner.nextInt();
		// could raise an exception; index out of bounds
		// compiler does not complain
		// unchecked exception
		System.out.println("ele : " + a[i]);
	}
}
```

If we do not handle the checked exception or do not indicate it could be thrown, we get a compile time error.

```
// Exception:
//	checked and unchecked exception
//	there are cases where something becomes unusual
//	program can take care of them
//	ex: index out of bounds in an array - unchecked
//	there are cases where the programmer cannot do anything
//	opening a file; filename provided by the user;  and file not exist
//	either we should take care of such usage or tell the user
//		something catastrophic could happen
//		- like side effects of medicines
```

```
//L23/Exception5.java
// Exception:
import java.io.*;
import java.util.*;
public class Exception5
{
	public static void main(String[] args)
	{
		foo();
	}
	private static void foo()
	{
		Scanner scanner = new Scanner(System.in);
		String filename = scanner.next();
		Scanner myfile = new Scanner(new File(filename));
		System.out.println("file opened");
	}
}

// compile time error:
//Exception5.java:15: error: unreported exception FileNotFoundException; must be caught or declared to be thrown
//		Scanner myfile = new Scanner(new File(filename));
//		                             ^
//1 error
```

Now, the user of foo should be careful. It says it might throw an exception.

```
//L23/Exception6.java
// Exception:
import java.io.*;
import java.util.*;
public class Exception6
{
	public static void main(String[] args)
	{
		try
		{
			foo();
		}
		catch(Exception e)
		{
			System.out.println(e);
			e.printStackTrace();
		}
	}
	// throws clause ; required for checked exception only
	// like mandatory warning in medicine


	private static void foo() throws IOException
	{
		Scanner scanner = new Scanner(System.in);
		String filename = scanner.next();
		Scanner myfile = new Scanner(new File(filename));
		System.out.println("file opened");
	}
}
```

This is another example of using a checked exception – take care of this in the method itself. Do not have to indicate anything about the side effects of using the medicine.

```
//L23/Exception7.java
import java.io.*;
import java.util.*;
public class Exception7
{
	public static void main(String[] args)
	{
		foo();
	}
	// throws clause not required; taken care here itself
	// medicine is self contained; no side effects!
	private static void foo()
	{
		Scanner scanner = new Scanner(System.in);
		String filename = scanner.next();
		try
		{
			Scanner myfile = new Scanner(new File(filename));
		}
		catch(Exception e)
		{
			System.out.println(e);
			e.printStackTrace();
		}
		System.out.println("file opened");
	}
}
```

Trust this piece of my writeup is not exceptional – is just normal.

Enjoy.