JV – Object-Oriented Programming in Java

Seminar 5 – Advanced OO and Exception Handling

In the first lecture we were introduced to the basics of the object-oriented (OO) methodology. Since then, we have had a fair amount of practice on the basic concepts. Now, however, it is time to move on to study a few more advanced aspects of OO programming. In the following sections we will discuss the topics of inheritance, encapsulation and polymorphism in detail. These are features of OO that we have not previously examined, and once mastered they will make the design of your programs a lot clearer and more efficient. After this, exception handling will be introduced.

I. Inheritance

Figure 1 shows how different types of staff working within a school can be derived from a fundamental concept of a school employee.

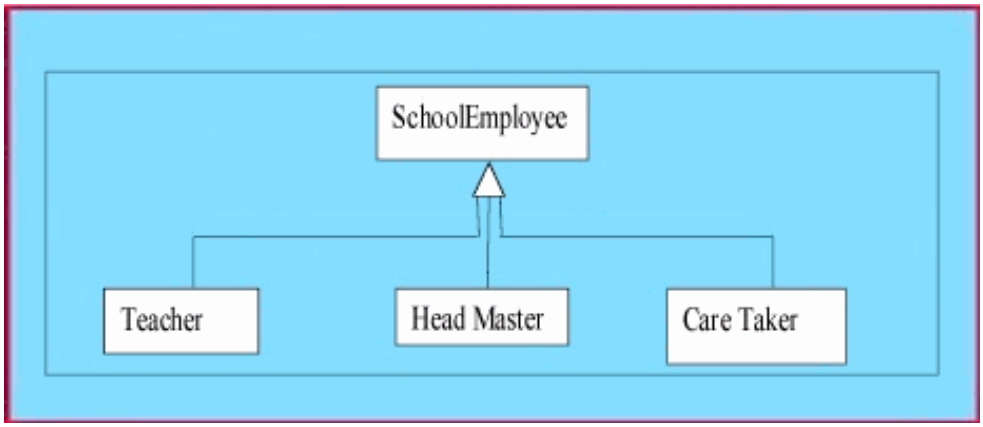

Figure 1: *School employee inheritance example*

The concept of a school employee is a blueprint of the generic school staff and could specify features such as name and salary. Many

different varieties of employee can be derived from this blueprint by adding features for specific jobs. For example:

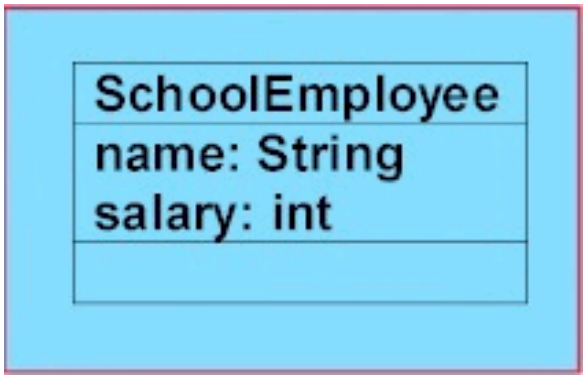

Figure 2: *SchoolEmployee class*

Where

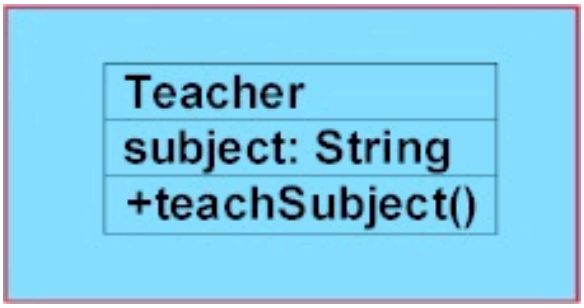

Figure 3: *Teacher class*

inherits all the attributes and methods, public and protected, of the SchoolEmployee class as well as adding its own. More precisely, one could say that a teacher *is-a* school employee with an extra attribute. This is-a relationship is represented by a hollow triangular shape (as shown in **Figure 1**) within a class diagram. Within a program, it would be very wasteful to completely implement a class for each type school staff that is available. Instead, all staff can inherit the characteristics of the generic SchoolEmployee class, after which each particular staff member can then add on its own features.

In Java, the initial class (parent class) is called the superclass, while the receiving class (child class) is called the subclass.

Of course, inheritance does not need to stop at deriving one layer of classes. In the above hierarchy, TeacherAssistant could be added as a subclass to the Teacher class. The collection of all the classes extending from a common parent is called an inheritance hierarchy. The path from a particular class to its ancestors in the inheritance hierarchy is its inheritance chain. The main benefits of inheritance are:

1. It increases the level of abstraction within a program.

2. It improves clarity in the design of classes by allowing the implementation of methods to be postponed in the superclasses and appear in subclasses.

3. It increases your ability to reuse classes. Software can be extended by reusing many previously defined classes and adding new methods to the subclasses.

Another point to note before moving on to the implementation of inheritance is that in Java, a subclass can only inherit from one superclass. In other words, multiple inheritance is forbidden.

A. Implementing Inheritance

When implementing inheritance, the first thing that must be done is to implement the parent class. This is because this class will contain attributes and methods that are used by subclasses but are not implemented by subclasses. In our case, consider the class SchoolEmployee. As mentioned above, this generic parent class should at the very least contain some form of reference to an individual school employee. For example:

```
1  // School Employee Class
2  // Dr. Y. Jing
3  // 06 December 2007
4  // The university of Liverpool, UK
5
6  import java.io.*;
7
8  class SchoolEmployee{
9  // --- Fields ---
10
11 protected String name;
12 protected int salary;
13
14 // --- Constructors ---
15
16 public SchoolEmployee(String n, int s){
17     name = n;
18     salary = s;
19     }
20 // --- Methods ---
21
22 public void printSchoolEmployeeDetails(){
23     System.out.println("SchoolEmployee:");
24     System.out.println("Name ="+name);
25     System.out.println("Salary ="+salary);
26     System.out.println("");
27     }
28 }
```

Figure 4: *Class SchoolEmployee*

Notice that the attributes of this class are not declared as private but as protected. As we already know, class attributes and variables are normally declared as private to prevent access from outside of the class. However, although private variables are inherited by subclass objects (in that each such object has its own copy of that variable with its value), such variables cannot be accessed directly by the object itself, and can only be accessed through any protected or public access method of the superclass. A protected member of a superclass can be accessed from any methods of its subclasses and can be accessed from any method of any class within in the same package. Subclasses would normally refer to protected members of the superclass by simply using the member name. This is because the attribute of the superclass is also an attribute if the subclass.

In the above example, the class is given a constructor for initializing the name and salary of the SchoolEmployee. Suppose, though, that we now wish to implement another class to represent a particular type of SchoolEmployee (e.g. a teacher). As stated before, this class must inherit all the methods and attributes of the SchoolEmployee

class, after which it can add some new features specific to itself. The Java syntax for achieving inheritance is:

class subclassName extends superclassName

For example, the statement

class Teacher extends SchoolEmployee

permits all of the variables and methods of the class SchoolEmployee to be inherited by the class Teacher. Failure to use inheritance would mean that all the variables and methods that are common to the classes SchoolEmployee and Teacher would need to be re-coded as part of the definition of the class Teacher. A full example of the Teacher class would be as follows:

```
1  //Teacher Class
2  // Dr. Y. Jing
3  // 06 December 2007
4  // The university of Liverpool, UK
5
6  import java.io.*;
7
8  class Teacher extends SchoolEmployee{
9  // --- Fields ---
10 private String subject;
11
12 // --- Constructor ---
13
14 public Teacher (String n, int s, String sub){
15     super(n,s);
16     subject = sub;
17 }
18
19 // --- Methods ---
20
21 public void printTeacherDetails(){
22     System.out.println("Teacher:");
23     System.out.println("Name=" +name);
24     System.out.println("Salary="+salary);
25     System.out.println("Subject="+subject);
26 }
27 }
```

Figure 5: *Class Teacher*

In the above example, the attributes name and salary have been inherited from the class SchoolEmployee. Subject is an attribute that has been added for the Teacher class. I have also added the method

printTeacherDetails(), since although the method supplied with the parent class would print out the teacher's name and salary, it would not print out the subject that the teacher was associated with. In relation to the two methods printTeacherDetails() and printSchoolEmployeeDetails(), a concept known as polymorphism could have been used to improve the way I have implemented the code here. However, I will discuss this concept later in this lecture, after which I will introduce the improved implementation. For the present program, I have deliberately not used polymorphism as I wanted to demonstrate that completely new methods may be added to the child class that the parent class cannot access, while on the other hand the child class can use the methods of the parent class. Notice that a new constructor has been defined for the Teacher class that includes the initialization of the name, salary and subject of a teacher. The coding of the Teacher constructor has made use of the SchoolEmployee constructor by making a specific reference to the constructor of the superclass through the reserved keyword **super**.

The **super** keyword, if present in a constructor, must always be the first statement in a constructor body.

A class that would use the above two classes and therefore demonstrate the above output would be as follows:

```
1  // Run Inheritance Class
2  // Dr. Y. Jing
3  // 06 December 2007
4  // The university of Liverpool, UK
5
6  import java.io.*;
7
8  class RunInheritance{
9
10 // Main method
11
12 public static void main (String[] args){
13 SchoolEmployee se = new SchoolEmployee("Peter Johnson", 24000);
14 Teacher t = new Teacher ("Jack heart", 26000, "Science");
15 se.printSchoolEmployeeDetails();
16 t.printTeacherDetails();
17 }
18 }
```

Figure 6: *Class RunInheritance*

II. Encapsulation

Encapsulation is an approach to program development that attempts to hide much of the implementation details of a class. The interface of each class is defined in such a way as to reveal as little as possible about its inner workings. This is a key concept in working with objects. It cannot be stressed enough that the key to making encapsulation work is to have programs that never directly access instance variables (fields) in a class. Programs should only interact with class data via the object's methods. Therefore, encapsulation is the way to give the object its “black box” behaviour, meaning that an object may totally change how it stores its data, but as long as it continues to use the same methods to manipulate the data, no other object will know or care. This approach to program development leads to classes that maintain their integrity, are readily understood, may be clearly documented and can be developed independently of each other.

Encapsulation is the key behind creating Abstract Data Types (ADT). Encapsulation is illustrated in **Figure 7,** where it is seen that access to class data is only allowed via specific instance methods. The implementation of the data, constructors, instance methods and class methods is hidden from the user.

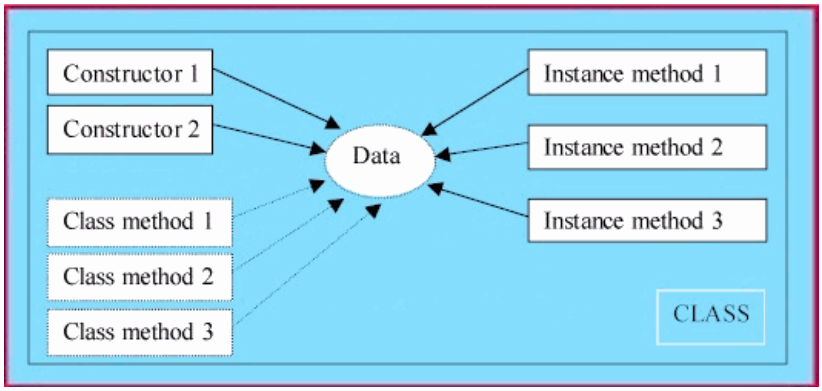

Figure 7: *Encapsulation representation*

An encapsulated group, consisting of data and its associated methods is called an ADT. By denying programmers access to the implementation details, one can safely modify the implementation without worrying that you may inadvertently introduce errors into code that uses that class. A rough template for an encapsulated class can be seen in **Figure 8**.

```
1  // Encapsulated Class
2  // Dr. Y. Jing
3  // 06 December 2007
4  // The university of Liverpool, UK
5
6  class ClassName{
7  // --- Fields ---
8
9  private String attribute1;
10 private int attribute2;
11
12 // --- Constructors ---
13
14 public ClassName(String a1, int a2){
15 attribute1=a1;
16 attribute2=a2;
17 }
18
19 public ClassName(){
20 attribute1="";
21 attribute2=0;
22 }
23
24 // --- Public Instance Methods
25
26 public String getAttribute1(){return attribute1;}
27 public int getAttribute2(){return attribute2;}
28 public void setAttribute1(String newA1){attribute1=newA1;}
29 public void setAttribute1(int newA2){attribute2=newA2;}
30
31 // --- Private Class Methods ---
32
33 private static void someMethod(){
34     /*
35     Code for some internal processsing can be placed
36     here. This method can be called only from bjects
37     instantiated from this class. It is not available
38     to other classes.
39     */
40     }
41 }
```

Figure 8: *Encapsulated class template*

Here it should be noted that access to the data must be through trusted methods of the class as direct access to class data can result in variables not being correctly updated.

In documenting the interface to an encapsulated class, there is no need to reveal the constants, variables and methods that users cannot access. Therefore, the appearance of the class becomes uncluttered and uncomplicated; hence the good documentation of the Java API. The concept of an ADT is not new to us. So far, we have been using the String data type fairly extensively. A variable of type String can apply the String constructors to build objects. These objects can then invoke many predefined instance methods such as length(), toUpperCase(), toLowerCase(), trim(), substring(), etc., that we can use within our programs. We never see how these methods are implemented, and without consulting the authors of the String class we have no way of knowing how string data is stored within the class, although we could guess that the string is stored as an array of characters. However, even if the string is represented in this manner, we cannot access this data directly and must rely upon access through the instance methods that are supplied for the class (and are documented within the Java API).

Similarly, the implementations of the constructors of this class are hidden from us. All this hiding prevents users from altering well-engineered code.

In complying with encapsulation, the following rules should always by followed:

1. Class data should be kept private (or at least protected).

2. Constructors and instance methods that are to be accessed from outside the class should be made public.

3. Instance methods that are part of the class but are not accessed from outside the class (i.e. used to help public methods achieve their goals) should be kept private.

4. Class methods should be defined as static and public if they are to be accessed from outside the class; otherwise they should be labelled as static and private.

III. Polymorphism

It is important to understand what happens when a method call is applied to objects of various types in an inheritance hierarchy. When a method (with certain parameters or no parameters) is called from an object instantiated from a subclass within an inheritance hierarchy, here is what happens:

1. The object checks whether or not it has a method with that name and with exactly the same parameters. If so, it uses it.

2. If not, the parent class becomes responsible for handling the message and looks for a method with that name and those parameters. If so, it calls that method.

This process of method handling can continue moving up the inheritance chain (i.e. parent classes are checked until the chain of inheritance stops or until a matching method is found). If there is no matching method anywhere in the chain, a compile time error occurs. Therefore, it is clear that methods with the same name can exist on many levels of the chain. This leads to one of the fundamental rules of inheritance:

> **A method defined in a subclass with the same name and parameter list as a method in one of its ancestor classes hides the method of the ancestor class from the subclass.**

An object's ability to decide what method to apply to itself, depending on where it is in the inheritance hierarchy, is called polymorphism. The idea behind this is that while method calls may be the same, objects may respond differently.

Now let us return to our school employee inheritance hierarchy program and make improvements to its implementation by using polymorphism:

Here is the modified SchoolEmployee class:

```
1  // Modified School Employee Class
2  // Dr. Y. Jing
3  // 06 December 2007
4  // The university of Liverpool, UK
5
6  import java.io.*;
7
8  class SchoolEmployee {
9  // --- Fields ---
10
11 protected String name;
12 protected int salary;
13
14 // Constructor ---
15 public SchoolEmployee(String n, int s){
16     name = n;
17     salary = s;
18 }
19 // --- Methods ---
20
21 public void printStaffDetails(){
22     System.out.println("School Employee:");
23     System.out.println("Name ="+name);
24     System.out.println("Salary ="+salary);
25     System.out.println("");
26 }
27 }
```

Figure 9: *Modified SchoolEmployee class*

Here is the modified Teacher class:

```
1  //Modified Teacher Class
2  // Dr. Y. Jing
3  // 06 December 2007
4  // The university of Liverpool, UK
5
6  import java.io.*;
7
8  class Teacher extends SchoolEmployee{
9  // --- Fields ---
10 private String subject;
11
12 // --- Constructor ---
13
14 public Teacher (String n, int s, String sub){
15     super(n,s);
16     subject = sub;
17 }
18
19 // --- Methods ---
20
21 public void printStaffDetails(){
22     System.out.println("Teacher:");
23     System.out.println("Name=" +name);
24     System.out.println("Salary="+salary);
25     System.out.println("Subject="+subject);
26 }
27 }
```

Figure 10: *Modified Teacher class*

and finally here is the modified class containing the main method:

```
1  // Modified Run Inheritance Class
2  // Dr. Y. Jing
3  // 06 December 2007
4  // The university of Liverpool, UK
5
6  import java.io.*;
7
8  class RunInheritance{
9
10 // Main method
11
12 public static void main (String[] args){
13 SchoolEmployee se = new SchoolEmployee("Peter Johnson", 24000);
14 Teacher t = new Teacher ("Jack heart", 26000, "Science");
15 se.printStaffDetails();
16 t.printStaffDetails();
17 }
18 }
```

Figure 11: *Modified RunInheritance class*

IV. Exception Handling

An exception is an event occurring during the execution of a program that makes continuation impossible or undesirable. Examples of the causes of exceptions include division by zero, array references with indexes out of bounds, trying to open a file that does not exist and incorrect user input. There are many other causes of exceptions. Many programming languages respond to exceptions by aborting execution. This may occur in a very untimely and awkward manner. However, one of the design goals of Java was to provide the language with sufficient features to enable the programmer to write robust programs. An exception handler is a piece of program code that is automatically invoked when an exception occurs. The exception handler can take immediate appropriate remedial action, then either resume execution of the program (at the point where the exception occurred or elsewhere) or terminate the program in a controlled manner.

An example of the use of an exception handler is where a program cannot open a file that the user has requested. Maybe the user spelled the file name incorrectly or the file simply does not exist. Here, instead of our Java program crashing, the program could ask

the user whether or not they would prefer to enter the file name again. If the user chose not to enter the file name again, the program could exit in a controlled fashion, while on the other hand, if a new file name is entered, the program could pick up from where the exception occurred with the new file name. If this exception handler is placed within an appropriate loop, the program can be made robust for multiple consecutive user errors.

In terms of safety critical systems, such fault-tolerant software is of vital importance.

Exceptions can be implicit, in which case it is a signal from the JVM to the program indicating a violation of a semantic constraint of the Java language. An example of this is attempting to index outside the bounds of an array which would automatically throw an index out of bounds exception. Exceptions can also be explicit, in which case they are thrown from within the program to signal that an error condition exists. For example, if input data was only acceptable within a predefined range, then the program might throw an exception if the data was found to lie outside of this range.

In either case, it is essential to grasp the concept that if an exception is thrown, it must be eventually be caught. If you supply an exception handler, then the exception may be dealt with in the program, otherwise, the Java interpreter will handle the exception by reporting on its cause and abandoning the program.

Up until this point, some of you haven't used exception handlers within your programs. However, in methods that read input from the keyboard with the method readLine() from the class BufferedReader, you may have noticed that the signature (method name, parameter list, return type, any checked exceptions, and any modifiers that provide more information about the method, e.g. public String substring(int~beginIndex)) of the method includes the phrase throws IOException. This is because there is no explicit means of handling exceptions within the method. If such a method does happen to throw an exception (i.e. an exception occurs such as incorrect user input), then this exception still must be caught. If no exception handlers have been explicitly defined within the method, then the Java interpreter will catch any exceptions. The program will then exit

with an appropriate error message. In this section, you will learn how to write your own Java code to handle the various exceptions that may arise so that your program does not need to exit each time an exception occurs.

A. Exception Classes

An exception in Java is treated as an object that is an instance of the superclass java.lang.Throwable or an instance of one of its subclasses.

The following is a partial listing of the class Throwable:

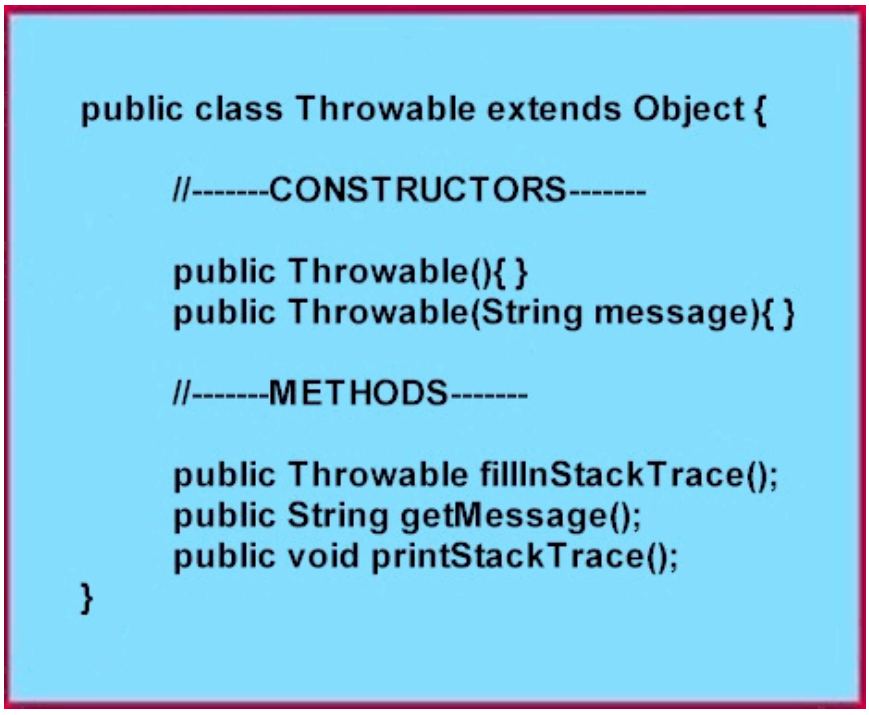

Figure 12: *Partial class Throwable*

Further documentation for this class can be found within the Java API. The superclass Throwable has two immediate subclasses, Error and Exception. Here we are interested in the Exception subclass. Briefly, however (out of interest), the subclass Error is a superclass to

classes that deal with errors that are generally unrecoverable, for example VirtualMachineError.

The Exception subclass is a superclass to a number of subclasses that support exceptions which may be detected and ultimately recovered from. **Figure 13** shows a partial class hierarchy between the top level of exception handling classes. There are many other subclasses that are not shown here. These can be studied from the Java API at your leisure.

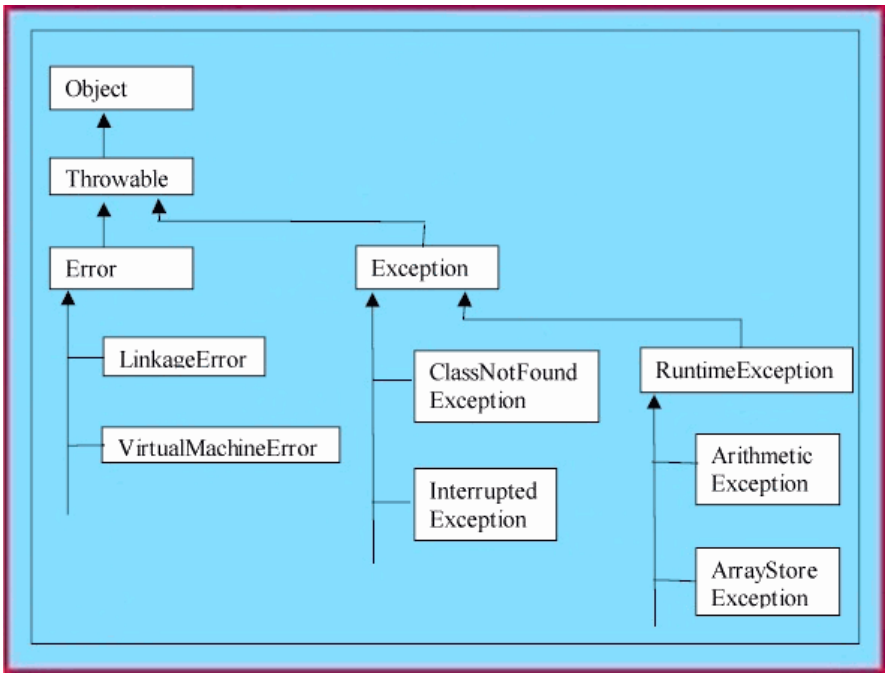

Figure 13: *Hierarchy of exception classes*

From the above figure it is clear that a number of exception classes exist that inherit from the superclass Throwable. The Java interpreter is capable of handling all the exceptions that may be generated. Unfortunately, as mentioned previously, the interpreter abandons the program after having warned of the exceptional condition. We will now learn how to catch these exceptions ourselves.

B. Catching an Exception

To handle exceptions within our programs, we can use a try clause followed by a catch block.

A try clause is used to delimit a block of code where the result of any method calls or other operations might cause an exception. To handle any exceptions that may arise within the try clause, we must have at least one catch block. The syntax for the try clause is as follows:

```
try{
        //statement 1
        //statement 2
        //……..
        //statement n
}
```

The catch block contains a single parameter whose type is any class from the superclass **Throwable** down through the subclass hierarchy. The catch block is only entered if the exception object is the same type as the class stated by the parameter, or if the exception object is an instantiation of a subclass of the parameter. The syntax for the catch block is as follows:

```
catch(exception classObjectName){
        //statement 1
        //statement 2
        //……..
        //statement n
}
```

catch(ArithmeticException ae){….} will only allow an exception ae of the type ArithmeticException to be caught. However, catch(Exception e){….} will allow an exception e of the type Exception or any type below exception in the class hierarchy to be caught. Therefore, if the exception object was of type ArithmeticException, which is a subclass of Exception, the exception will be caught.

If no exception is thrown within the try clause, then the catch block is not executed. However, if an exception does arise within the try clause, then the remaining statements of the try clause are not executed.

The following are two examples of the try/catch block combination in use.

Note: this code is not a full program in itself but may be used to illustrate how exceptions may be handled within a program:

```
// Partial Exception Handling Class
// This code is not a full program in itself but may be used
// to illustrate how exceptions may be handled within a program
// Dr. Y. Jing
// 06 December 2007
// The university of Liverpool, UK

int myInteger;
int check;
do {
    check =0;
    try {
        System.out.println("\n Please Enter an Integer");
        myInteger=new Interger(keyboardInput.readLine()).intValue;
        System.out.println("Value of myInteger="+myInteger);
    }
    catch (NumberFormatException nfe){
    System.out.println("\n You did not enter an Integer. Please try again.");
    check = 1;
    }
}while (check == 1);
```

Figure 14: *Partial exception handling program*

Here is a full program to play with. This program has been written in a long-winded way in order to demonstrate how single pairs of try clauses and catch blocks can be used effectively. After this example, a more efficient way of writing the program will be presented:

```
1  // Hnadle Exception Class
2  // Dr. Y. Jing
3  // 06 December 2007
4  // The university of Liverpool, UK
5  import java.io.*;
6  class TrycatchDemo1{
7  // --- Fields ---
8  static BufferedReader keyboard = new BufferedReader(new InputStreamReader(System.in));
9  // Main method since he IO exception isn't handled in my code,
10 // I'm throwing this exception to the system
11 public static void main (String[] args) throws IOException{
12     int firstInt, secondInt, check;
13     float result;
14     boolean tryAgain1, tryAgain2;
15     do {
16         tryAgain2=false;
17         firstInt=0;
18         secondInt=0;
19         System.out.println("Program to divid 1st integer by 2nd Integer");
20         do {
21             tryAgain1=false;
22             try {
23                 System.out.println("\n Please Enter 1st Integer:");
24                 firstInt = new Integer(keyboard.readLine()).intValue();
25                 System.out.println("\n Please Enter 2nd Integer:");
26                 secondInt = new Integer(keyboard.readLine()).intValue();
27                 }
28             catch(NumberFormatException nfe){
29                 System.out.println("\n You didn't enter an integer, Please try again");
30                 tryAgain1=true;
31                 }
32             }while (tryAgain1);
33             try {
34                 result = (float)(firstInt/secondInt);
35                 System.out.println("Result="+result);
36             }
37             catch(ArithmeticException ae){
38                 System.out.println("Division by zero has occurred. Please try again.");
39                 tryAgain2=true;
40                 }
41         }while (tryAgain2);
42 }
43 }
```

Figure 15: *Example 1 of error handling*

In the above code we have supplied code to handle NumberFormatExceptions and ArithmeticExceptions. However, we have not included explicit code to handle any IO exceptions even though the main method does use IO. Instead, we added the throws IOException to the signature of the main method. As mentioned earlier, the Java interpreter is capable of handling all the exceptions that may be generated. Here, throws IOException will make use of this interpreter facility. Unfortunately however (as there is no explicit exception handling code), if an IO exception does arise, the interpreter will abandon the program after having warned the user of the exceptional condition. In this case, this action will suffice.

C. Catching Multiple Exceptions

In the above program, I am only catching one exception at a time. However that above code can be shortened so that I only require one

try clause. In this case I would wish to be able to catch multiple exceptions as it would be possible for more than one exception to arise within the try clause. Many catch blocks can be included within a method in order to catch a number of known exceptions. In the above code there are two try clauses and two catch blocks. This could be replaced by one try clause followed by 2 consecutive catch blocks. For Example:

```
1  // Exception Handling Class
2  // Dr. Y. Jing
3  // 06 December 2007
4  // The university of Liverpool, UK
5
6  import java.io.*;
7
8  class TryCatchDemo2{
9  // --- Fields ---
10
11 static BufferedReader keyboard = new BufferedReader(new InputStreamReader(System.in));
12
13 // Main method
14 // Since the IO exception isn't handled in my code, I am throwing the exeption to the system
15
16 public static void main(String[] args) throws IOException{
17     int firstInt, secondInt, check;
18     float result;
19     boolean tryAgain;
20         do {
21             tryAgain=false;
22             firstInt=0;
23             secondInt=0;
24             System.out.println("Program to divid 1st integer by 2nd Integer");
25             try {
26                 System.out.println("\n Please Enter 1st Integer:");
27                 firstInt = new Integer(keyboard.readLine()).intValue();
28                 System.out.println("\n Please Enter 2nd Integer:");
29                 secondInt = new Integer(keyboard.readLine()).intValue();
30                 result = (float)(firstInt/secondInt);
31                 System.out.println("Result="+result);
32                 }
33             catch(NumberFormatException nfe){
34                 System.out.println("\n You didn't enter an integer, Please try again");
35                 tryAgain=true;
36                 }
37             catch(ArithmeticException ae){
38                 System.out.println("Division by zero has occurred. Please try again.");
39                 tryAgain=true;
40                 }
41         }while (tryAgain);
42     }
43 }
```

Figure 16: *Error handling example*

When using more than one catch block to explicitly trap exceptions, make sure that the class type for each block parameter is not a superclass of a following catch block parameter. If you need to use a superclass in a block as a “catch all” for any exceptions that you have not explicitly coded, make sure that you place it as the parameter to the last catch block of the sequence. If you do not, any catch blocks after this block will not be used by the program.

D. Finally

At this point in the lecture, it should be noted there is another useful feature of Java that allows the programmer to define a block of code that is guaranteed to be executed before the computer exits from a method, regardless of whether an exception was thrown or a return statement executed within a try block. This is the finally block.

The syntax of this block is:

```
finally{
        //statement 1
        //statement 2
        //……..
        //statement n
}
```

This block may be used to release any permanent resources that the methods may have used such as closing any open files.

If there is a local catch block to handle an exception being thrown from the try block, the code of the catch block is executed before the code of the finally block.

In **Chapter 9** of the text, read the following sections:

9.1 - 9.5
Skim 9.7 - 9.9

In **Chapter 10**, read the following sections:

10.1 - 10.6
Skim 10.7 - 10.10

In **Chapter 13**, read the following sections:

13.1 - 13.5

Chapter 13 is a relatively short chapter and deals with exception handling. You may wish to use this chapter almost like an index - check the first page of the chapter, p. 639, and find the exception type you need, then look up how to use it. You can read the rest of what you don't need once you've accomplished your task goals for this week.

You may wish to review chapters on loops, conditional statements, and constructors in order to accomplish this week's assignments. If you are still unclear on anything from previous weeks, feel free to ask questions in the Q & A section.

You will notice in each chapter various sections entitled Common Programming Errors, Testing and Debugging, Performance Tips and Software Engineering Observation, etc. These sections are meant to be skimmed over or even ignored on a first reading, but bear them in mind as you may wish to return to them later.

Helpful Additional Readings

I'm posting a 'polymorphic dog' handout that will help clarify some of the readings from chapter 10. Hopefully. ;-)

If you happen to get really stuck on a concept we've introduced in this week's material, please post a message as a response to this thread in the main folder to let me know. I have a ton of extra materials and instruction guides that I can post that may help clarify things for you. **All you have to do is ask! :-)**

Helpful Websites

General inheritance information:

http://java.sun.com/docs/books/tutorial/java/concepts/inheritance.html

http://www.developer.com/java/article.php/948121

http://www.javacoffeebreak.com/java104/java104.html

http://max.cs.kzoo.edu/~abrady/JavaLangNotes/inheritance.html

Polymorphism information:

http://www.javaworld.com/javaworld/javatips/jw-javatip30.html

http://www.developer.com/java/other/article.php/966001

http://www.phillipmorelock.com/examples/polyMorphism.html

http://developer.java.sun.com/developer/onlineTraining/Programming/BasicJava2/oo.html#poly

Error handling information:

http://java.sun.com/docs/books/tutorial/essential/exceptions/index.html

http://www.ida.liu.se/~TDDI48/2004/slides/exceptions.pdf

http://www.camtp.uni-mb.si/books/Thinking-in-Java/Chapter10.html