COMP503/ENSE502/ENSE602: Week 8 – Exercises

➢ Get Started!

Exceptions are used in Java to handle exceptional events (i.e. errors) that interrupt program execution. When an error occurs, an exception is thrown. Examples of errors that throw exceptions are

- reading input from the user that does not match the expected format.
- opening a file that does not exist or failure when writing to a file,
- integer division by zero or using invalid array indices.

Exception handling involves specifying try-catch-finally blocks where the

- try block contains code that could cause an error,
- catch block contains code to deal with a specific class of exceptions, and
- finally block contains code that is executed whether or not an exception was thrown.

- Create the class **ExceptionBasics** and enter the source code that appears below.

```
public class ExceptionBasics {

	static boolean CAUSE_EXCEPTION=true;

	public static void main(String args[]){
		try {
			System.out.println("Try Block executed:");
			if(CAUSE_EXCEPTION)
				throw new Exception("This is an exception");
			System.out.println("End of the try block!");
		} catch (Exception e) {
			System.err.println("Catch Block executed on :"+e);
		}
		finally{
			System.out.println("Finally Block is executed");
		}
	}
}
```

  - Run the code, setting CAUSE EXCEPTION=true, noting the console output, and
  - Run the code, setting CAUSE EXCEPTION=false noting the console output.

COMP503/ENSE502/ENSE602: Week 8 – Exercises

The **ExceptionBasics** class creates and throws an Exception using the code

```
throw new Exception("This is an exception");
```

where the new keyword is used to construct a new exception object with an error message. The throw keyword explicitly throws an exception, interrupting the program's normal execution flow to identify the nearest matching catch block.

- Create a new class **`ExceptionBascis2`** and modify the main method from **`ExceptionBasics`** such that 10 exceptions are thrown and caught. When the exception is caught, an output is made to the error stream System.err. For example:

```
1 Exception caught!
2 Exception caught!
3 Exception caught!
…
```

  For this exercise, you do not need a finally block.

COMP503/ENSE502/ENSE602: Week 8 – Exercises

➢ **Exercise 1:** Handling input data validation

Exceptions are very helpful to catch errors that arise in the validation of data input by a user. The statement

```
Scanner scanner = new Scanner(System.in);
```

instantiates a **Scanner** and associates it with the console input stream **System.in**. The **Scanner** object parses input text from the console into a sequence of tokens, using a regular expression. Each token can then be converted into a primitive type or a string of text. For example, the **nextInt()** method scans the next token available, returning the integer value. According to the Java API Specification documentation, this method can throw different classes of exceptions:

- **InputMismatchException** - if the next token does not match the integer regular expression, or is out of range
- **NoSuchElementException** - if input is exhausted
- **IllegalStateException** - if this scanner is closed

Consider the following Java source code that reads in two integers from the console and prints their product.

```
import java.util.InputMismatchException;
import java.util.Scanner;
public class ComputeProduct {
	private static Scanner scanner;
	public static void product() throws InputMismatchException{
		System.out.println("Welcome to the calculator.");
		System.out.println("Enter first number:");
		int value1 = scanner.nextInt();
		System.out.println("Enter second number:");
		int value2 = scanner.nextInt();
		System.out.println("Product is: "+(value1*value2));
	}
	public static void main(String args[]){
		scanner = new Scanner(System.in);
		product();
	}
}
```

Complete the above source code in the Class called **ComputeProduct** by adding try and catch blocks in the product method to catch and handle an **InputMismatchException**. The catch block should print an message to the error stream using **System.err.println** and flush the invalid input by calling **scanner.next()**;

Next, we introduce a simple menu system to the program with the functionality described in the following sample usage:

```
Welcome to the calculator.
1. Compute product
2. quit
1
Enter first number:
3
Enter second number:
5
Product is: 15
1. Compute product
2. quit
1
Enter first number:
abc
Error reading integer value
1. Compute product
2. quit
def
Invalid menu input. Please try again.
1. Compute product
2. quit
4
Invalid menu input. Please try again.
1. Compute product
2. quit
2
Goodbye!
```

Copy the product method and create a new Class **`ComputeProductMenu`** with a main method that implements the above functionality using a while loop to continually waits for input until the quit command is selected, catching all **`InputMismatchException`** errors and outputting a suitable message to the user.

COMP503/ENSE502/ENSE602: Week 8 – Exercises

➢ **Exercise 2:** Creating your own class of exceptions

In this set of exercises, you will learn how to create new exception classes and throw your own exceptions when required. First, we create a simple class called **`CustomerAccount`**, which stores the name of a customer as a string:

a) Write the **`CustomerAccount`** class with an instance variable that stores the name of the customer. Provide standard get, set and toString methods.

b) A key requirement for our class is that it represents a valid customer. That is, we want to ensure that the customer name may not be set to null or the empty string.

Modify the set method for the **`CustomerAccount`** class to throw an **`IllegalArgumentException`** whenever the input customer name parameter is null or the empty string. Note that your constructor should be using this set method to initialise the customer name upon the object's construction.

Ensure that your constructor and set method specifies the kinds of exceptions that can be thrown by using the throws keyword. For example, your method signature for the constructor should look similar to

```
public CustomerAccount(String name) throws IllegalArgumentException
```

Next, consider the following test harness using separate class called **`CustomerAccountTester`**.

```
public class CustomerAccountTester {
        public static void main(String[] args) {
                CustomerAccountExercise8 c1 = new CustomerAccountExercise8("Herr Mustermann");
                c1.setName(null);
        }
}
```

c) Complete the main method in the **`CustomerAccountTester`** class with try-catch blocks that handle the appropriate exceptions thrown by the **`CustomerAccount`** class. When the setName or constructor input parameter is null or the empty string, the exception thrown should be printed to the error stream using **`System.err.println`**.

COMP503/ENSE502/ENSE602: Week 8 – Exercises

Suppose that we wish to distribute our CustomerAccount class such that others programmers can use and extend its functionality. It would be easier to understand our class if there were specialised exceptions associated with the two types of restrictions we have placed on the customer object: the name cannot be null nor can it be the empty string. We can extend an existing exception class to achieve this. The Java class hierarchy has many exception classes which can be extended, but the goal is to find one that is closest to the meaning of the error we wish to convey to the programmer using our class. The **`IllegalArgumentException`** suits this purpose, and we can extend it by defining a new class **`CustomerAccountNameCannotBeNull`** as shown here:

```
public class CustomerAccountNameCannotBeNull extends IllegalArgumentException
{

	public CustomerAccountNameCannotBeNull(String message){
		super(message);
	}
	public CustomerAccountNameCannotBeNull(){
		super();
	}
}
```

The **`CustomerAccountNameCannotBeNull`** contains the default constructor and a constructor taking a string input parameter that gives details of the error that occurred.

d) Create a new exception class **`CustomerAccountNameCannotBeEmpty`** that extends the class **`IllegalArgumentException`**.

e) Update the **`CustomerAccount`** class such that the appropriate exception is thrown according to the input provided to the constructor/set method. Don't forget to specify the exceptions your methods (and constructors) can throw using the throws keyword

f) Run the test harness **`CustomerAccountTester`**. What happens? Do you need to modify the catch block to accommodate these new exception subclasses?

➢ **Exercise 3**: The **MoneyTransfer** Class

a) Construct a **MoneyTransfer** class comprising
- instance variables **CustomerAccount to** and **CustomerAccount from** and
- an instance variable **double amount** to transfer an amount of money
- an instance variable **long timeStamp** that records the time the transfer occurred (see below)
- get and set methods
- constructor with input parameters for to, from, and amount,
- toString method

In addition to setting the **to**, **from** and **amount** instance variables, the constructor should also record when the money was transferred (e.g. the same time that the **MoneyTransfer** object was constructed). Therefore, in the constructor you can set the **timeStamp** instance variable as follows:

```
this.timeStamp=System.currentTimeMillis();
```

which returns the current time in milliseconds. To convert this number into a readable date and time in the **toString** method, simply import the **java.util.Date** package and invoke

```
Date date=new Date(timeStamp);
String dateString = date.toString();
```

Now, we have the following constraints when instantiating **MoneyTransfer** objects. This means we should throw an exception if at least one of these constraints are not satisfied when we first construct the object or otherwise update the object's data using a set method:
- **to** and **from** customer accounts cannot be null
- **to** and **from** cannot be identical (i.e. the same object)
- **amount** may not be zero
- **amount** may not be negative

COMP503/ENSE502/ENSE602: Week 8 – Exercises

In the next parts, we show how exceptions can be used to store useful information about the error. For this scenario, we record the timestamp of the failed money transfer, e.g. when the exception was thrown.

b) Create a subclass of **`IllegalArgumentException`** named **`MoneyTransferException`** and complete the following:
   - Define an instance variable **`long timeStamp`** in the **`MoneyTransferException`** class with suitable get and set methods
   - Define the constructors
     `public MoneyTransferException(String message)`
     and
     `public MoneyTransferException()`

     invoking the superclass constructors using the super keyword.

c) Update the **`MoneyTransfer`** class by throwing an exception when the input to the constructor or set method does not satisfy the constraints listed above.

Exceptions thrown in the **`MoneyTransfer`** class should be created with a descriptive message identifying the cause of the error according to the constraints listed above. For example, a **`MoneyTransfer`** object cannot have a negative amount of money, and so the set method for the instance variable amount would throw an exception as follows:

```
public void setAmount(double amount) throws MoneyTransferException {
      if (amount < 0) {
         MoneyTransferException me = new
           MoneyTransferException("Transfer amount cannot be negative");
         me.setTimeStamp(this.timeStamp);
         throw me;
      }

      this.amount = amount;
}
```

d) Complete the following test harness by writing additional try-catch blocks to completely test the functionality of the **`MoneyTransferTester`** class.

```
public class MoneyTransferTester {
	public static void main(String args[]){
		CustomerAccountUpdate to = new CustomerAccountUpdate("Bob");
		CustomerAccountUpdate from = new CustomerAccountUpdate("Alice");
		try{
			new MoneyTransfer(from,to,100.00);
		}catch(MoneyTransferException me){
			System.err.println(me);
		}

		try{
			new MoneyTransfer(from,to,-99.00);
		}catch(MoneyTransferException me){
			System.err.println(me);
		}
	}

}
```

The output of the test harness is given as follows:

```
exercises.MoneyTransferException: Transfer amount cannot be negative

.........
```

The following exercise are related file IO streams. Please make sure you are handling errors using different exceptions.

- **Exercise 4: Appending text to existing file**

Write a Java program to append text to an existing file. (Note: you can use **append**(…a text as input parameter…) method!)

- **Exercise 5: Finding longest word**

Write a Java program to find the longest word in a text file.