CHP-19
Java Exceptions

An exception is a problem that arises during the execution of a program. An exception can occur for many different reasons, including the following:

- A user has entered invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

To understand how exception handling works in Java, you need to understand the three categories of exceptions:

- **Checked exceptions:** A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer. For example, if a file is to be opened, but the file cannot be found, an exception occurs. These exceptions cannot simply be ignored at the time of compilation.
- **Runtime exceptions:** A runtime exception is an exception that occurs that probably could have been avoided by the programmer. As opposed to checked exceptions, runtime exceptions are ignored at the time of compilation.
- **Errors:** These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

Exception Hierarchy:

All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.

Errors are not normally trapped form the Java programs. These conditions normally happen in case of severe failures, which are not handled by the java programs. Errors are generated to indicate errors generated by the runtime environment. Example : JVM is out of Memory. Normally programs cannot recover from errors.

The Exception class has two main subclasses: IOException class and RuntimeException Class.

Here is a list of most common checked and unchecked **Java's Built-in Exceptions.**

Java's Built- in Exceptions

Java defines several exception classes inside the standard package **java.lang**.

The most general of these exceptions are subclasses of the standard type RuntimeException. Since java.lang is implicitly imported into all Java programs, most exceptions derived from RuntimeException are automatically available.

Java defines several other types of exceptions that relate to its various class libraries. Following is the list of Java Unchecked RuntimeException.

Exception	Description
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.

ArrayStoreException Assignment to an array element of an incompatible type.
ClassCastException Invalid cast.
IllegalArgumentException Illegal argument used to invoke a method.
IllegalMonitorStateException Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException Environment or application is in incorrect state.
IllegalThreadStateException Requested operation not compatible with current thread state.
IndexOutOfBoundsException Some type of index is out-of-bounds.
NegativeArraySizeException Array created with a negative size.
NullPointerException Invalid use of a null reference.
NumberFormatException Invalid conversion of a string to a numeric format.
SecurityException Attempt to violate security.
StringIndexOutOfBounds Attempt to index outside the bounds of a string.

UnsupportedOperationException An unsupported operation was encountered.
Following is the list of Java Checked Exceptions Defined in java.lang.

Exception Description
ClassNotFoundException Class not found.
CloneNotSupportedException Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException Access to a class is denied.
InstantiationException Attempt to create an object of an abstract class or interface.
InterruptedException One thread has been interrupted by another thread.
NoSuchFieldException A requested field does not exist.
NoSuchMethodException A requested method does not exist.

Exceptions Methods:

Following is the list of important methods available in the Throwable class.

SN Methods with Description
1
public String getMessage()
Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.
2 **public Throwable getCause()**
Returns the cause of the exception as represented by a Throwable object.
3 **public String toString()**
Returns the name of the class concatenated with the result of getMessage()
4 **public void printStackTrace()**
Prints the result of toString() along with the stack trace to System.err, the error output stream.
5
public StackTraceElement [] getStackTrace()
Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.
6
public Throwable fillInStackTrace()
Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.

Catching Exceptions:

A method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

```
try
{
//Protected code
}catch(ExceptionName e1)
{
```

```
//Catch block
}
```

A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.

Example:

The following is an array is declared with 2 elements. Then, the code tries to access the 3rd element of the array which throws an exception.

```
// File Name : ExcepTest.java
import java.io.*;
public class ExcepTest{
public static void main(String args[]){
try{
int a[]=new int[2];
System.out.println("Access element three :"+ a[3]);
}catch(ArrayIndexOutOfBoundsException e){
System.out.println("Exception thrown :"+ e);
}
System.out.println("Out of the block");
}
}
```

This would produce the following result:

```
Exception thrown :java.lang.ArrayIndexOutOfBoundsException:3
Out of the block
```

Multiple catch Blocks:

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following:

```
try
{
//Protected code
}catch(ExceptionType1 e1)
{
//Catch block
}catch(ExceptionType2 e2)
{
//Catch block
}catch(ExceptionType3 e3)
{
//Catch block
}
```

The previous statements demonstrate three catch blocks, but you can have any number of them after a single try. If an exception occurs in the protected code, the exception is thrown to the first catch block in the list. If the data type of the exception thrown matches ExceptionType1, it gets caught there. If not, the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.

Example:

Here is code segment showing how to use multiple try/catch statements.

```
try
{
file =newFileInputStream(fileName);
x =(byte) file.read();
}catch(IOException i)
{
i.printStackTrace();
return-1;
}catch(FileNotFoundException f)//Not valid!
{
f.printStackTrace();
return-1;
}
```

The throws/throw Keywords:

If a method does not handle a checked exception, the method must declare it using the **throws**keyword. The throws keyword appears at the end of a method's signature.
You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword. Try to understand the different in throws and throw keywords.
The following method declares that it throws a RemoteException:

```
import java.io.*;
public class className
{
public void deposit(double amount)throws RemoteException
{
// Method implementation
throw new RemoteException();
}
//Remainder of class definition
}
```

A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas. For example, the following method declares that it throws a RemoteException and an InsufficientFundsException:

```
import java.io.*;
public class className
{
public void withdraw(double amount)throws RemoteException,
InsufficientFundsException
{
// Method implementation
}
//Remainder of class definition
}
```

The finally Keyword

The finally keyword is used to create a block of code that follows a try block. A finally block of code always executes, whether or not an exception has occurred.

Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.
A finally block appears at the end of the catch blocks and has the following syntax:

```
try
{
//Protected code
}catch(ExceptionType1 e1)
{
//Catch block
}catch(ExceptionType2 e2)
{
//Catch block
}catch(ExceptionType3 e3)
{
//Catch block
}finally
{
//The finally block always executes.
}
```

Example:

```
public class ExcepTest{
public static void main(String args[]){
int a[]=new int[2];
try{
System.out.println("Access element three :"+ a[3]);
}catch(ArrayIndexOutOfBoundsException e){
```

```
System.out.println("Exception thrown :"+ e);
}
finally{
a[0]=6;
System.out.println("First element value: "+a[0]);
System.out.println("The finally statement is executed");
}
}
}
```

This would produce the following result:

```
Exception thrown :java.lang.ArrayIndexOutOfBoundsException:3
First element value:6
The finally statement is executed
```

Note the following:

- A catch clause cannot exist without a try statement.
- It is not compulsory to have finally clauses whenever a try/catch block is present.
- The try block cannot be present without either catch clause or finally clause.
- Any code cannot be present in between the try, catch, finally blocks.

Declaring you own Exception:

You can create your own exceptions in Java. Keep the following points in mind when writing your own exception classes:

- All exceptions must be a child of Throwable.
- If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.
- If you want to write a runtime exception, you need to extend the RuntimeException class.

We can define our own Exception class as below:

```
class MyExceptio nextends Exception{
}
```

You just need to extend the Exception class to create your own Exception class. These are considered to be checked exceptions. The following InsufficientFundsException class is a user-defined exception that extends the Exception class, making it a checked exception. An exception class is like any other class, containing useful fields and methods.

Example:

```
// File Name InsufficientFundsException.java
import java.io.*;
public class InsufficientFundsException extends Exception
{
private double amount;
public InsufficientFundsException(double amount)
{
this.amount = amount;
}
public double getAmount()
{
return amount;
}
}
```

To demonstrate using our user-defined exception, the following CheckingAccount class contains a withdraw() method that throws an InsufficientFundsException.

```
// File Name CheckingAccount.java
import java.io.*;
public class CheckingAccount
{
private double balance;
private int number;
public CheckingAccount(int number)
{
this.number = number;
}
public void deposit(double amount)
{
balance += amount;
}
```

```
public void withdraw(double amount)throws InsufficientFundsException
{
if(amount <= balance)
{
balance -= amount;
}
else
{
double needs = amount - balance;
throw new InsufficientFundsException(needs);
}
}
public double getBalance()
{
return balance;
}
public int getNumber()
{
return number;
}
}
```

The following BankDemo program demonstrates invoking the deposit() and withdraw() methods of CheckingAccount.

```
// File Name BankDemo.java
public class BankDemo
{
public static void main(String[] args)
{
CheckingAccount c =new CheckingAccount(101);
System.out.println("Depositing $500...");
c.deposit(500.00);
try
{
System.out.println("\nWithdrawing $100...");
c.withdraw(100.00);
System.out.println("\nWithdrawing $600...");
c.withdraw(600.00);
}catch(InsufficientFundsException e)
{
System.out.println("Sorry, but you are short $"
+ e.getAmount());
e.printStackTrace();
}
}
}
```

Compile all the above three files and run BankDemo, this would produce the following result:

```
Depositing $500...
Withdrawing $100...
Withdrawing $600...
Sorry, but you are short $200.0
InsufficientFundsException
at CheckingAccount.withdraw(CheckingAccount.java:25)
at BankDemo.main(BankDemo.java:13)
```

Common Exceptions:

In Java, it is possible to define two categories of Exceptions and Errors.

JVM Exceptions: - These are exceptions/errors that are exclusively or logically thrown by the JVM. Examples : NullPointerException, ArrayIndexOutOfBoundsException, ClassCastException,

Programmatic exceptions:- These exceptions are thrown explicitly by the application or the API programmers. Examples: IllegalArgumentException, IllegalStateException.

CHP-20
Java Inheritance

Inheritance can be defined as the process where one object acquires the properties of another. With the use of inheritance, the information is made manageable in a hierarchical order.

When we talk about inheritance, the most commonly used keyword would be **extends** and **implements**. These words would determine whether one object IS-A type of another. By using these keywords we can make one object acquire the properties of another object.

IS-- A .Relationship:

IS-A is a way of saying : This object is a type of that object. Let us see how the **extends** keyword is used to achieve inheritance.

```
public class Animal{
}
public class Mammal extends Animal{
}
public class Reptile extends Animal{
}
public class Dog extends Mammal{
}
```

Now, based on the above example, In Object Oriented terms the following are true:

Animal is the superclass of Mammal class.
Animal is the superclass of Reptile class.
Mammal and Reptile are subclasses of Animal class.
Dog is the subclass of both Mammal and Animal classes.

Now, if we consider the IS-A relationship, we can say:

Mammal IS-A Animal
Reptile IS-A Animal

CHAPTER

Dog IS-A Mammal
Hence : Dog IS-A Animal as well

With use of the extends keyword the subclasses will be able to inherit all the properties of the superclass except for the private properties of the superclass.

We can assure that Mammal is actually an Animal with the use of the instance operator.

Example:

```
public class Dog extends Mammal{
public static void main(String args[]){
Animal a =new Animal();
Mammal m =new Mammal();
Dog d =new Dog();
System.out.println(m instanceof Animal);
System.out.println(d instanceof Mammal);
System.out.println(d instanceof Animal);
}
}
```
This would produce the following result:
```
true
true
true
```
Since we have a good understanding of the **extends** keyword, let us look into how the **implements**keyword is used to get the IS-A relationship.
The **implements** keyword is used by classes by inherit from interfaces. Interfaces can never be extended by the classes.

Example:

```
public interface Animal{}
public class Mammal implements Animal{
}
public class Dog extends Mammal{
}
```

The instanceof Keyword:

Let us use the **instanceof** operator to check determine whether Mammal is actually an Animal, and dog is actually an Animal
```
interfaceAnimal{}
class Mammal implements Animal{}
public class Dog extends Mammal{
public static void main(String args[]){
```

```
Mammal m =new Mammal();
Dog d =new Dog();
System.out.println(m instanceof Animal);
System.out.println(d instanceof Mammal);
System.out.println(d instanceof Animal);
}
}
```
This would produce the following result:
```
true
true
true
```

HAS-- A .relationship:

These relationships are mainly based on the usage. This determines whether a certain class **HAS-A**certain thing. This relationship helps to reduce duplication of code as well as bugs.
Lets us look into an example:
```
public class Vehicle{}
public class Speed{}
public class Van extends Vehicle{
privateS peed sp;
}
```
This shows that class Van HAS-A Speed. By having a separate class for Speed, we do not have to put the entire code that belongs to speed inside the Van class which makes it possible to reuse the Speed class in multiple applications.
In Object-Oriented feature, the users do not need to bother about which object is doing the real work. To achieve this, the Van class hides the implementation details from the users of the Van class. So basically what happens is the users would ask the Van class to do a certain action and the Van class will either do the work by itself or ask another class to perform the action.
A very important fact to remember is that Java only supports only single inheritance. This means that a class cannot extend more than one class. Therefore following is illegal:
```
public class extendsAnimal,Mammal{}
```
However, a