Week 9

1. **EXTENDING INTERFACES**

In Java, an interface is a means for achieving abstraction. The Java interface can only have abstract methods, not method bodies. In Java it is used to achieve abstraction and multiple inheritance. In other words, interfaces can have abstract methods and variables. It is not allowed to have a method body.

When one interface inherits from another, the sub-interface inherits all of the methods and constants declared by the super-interface. Furthermore, it can declare new abstract procedures and constants. To extend an interface, use the extends keyword in the same way that you would in a class definition. An interface can directly extend many interfaces, as opposed to a subclass, which can only directly extend one subclass. The following syntax can be used to accomplish this:

|  |
| --- |
| [public] interface InterfaceName extends interfacel[, interface2, , interfaceN]  {//interface body} |

The interface's name is InterfaceName in this case. This interface extends one or more other interfaces according to the extends clause. These are referred to as super interfaces and are listed alphabetically. Commas separate the names of the super interfaces.

|  |
| --- |
| interface Interface1 {  public void func1();  }  //Interface2 extending Interface1  interface Interface2 extends Interface1 {  public void func2();  }  class temp implements Interface2 {  //definition of method declared in interfacel  public void func1() {  System.out.println("Contents of Method func1() in Interface1");  }  public void func2() {  System.out.println("Contents of Method func2() in Interface2");  }  public void func3() {  System.out.println("Contents of Method func3() of Class temp");  }  }  class ExtendingInterface {  public static void main(String[] args) {  Interface2 v2; //Reference variable of Interface2  v2 = new temp(); //assign object of class temp  v2.func1();  v2.func2();  temp xl = new temp();  xl.func3();  }  }  **Output:** |

**Extending Multiple Interfaces**

Only one parent class can be extended by a Java class. Multiple inheritance is prohibited. Interfaces, on the other hand, are not classes and an interface can extend more than one parent interface. The keyword extends is used just once, and the parent interfaces are listed in a comma-separated list.

|  |
| --- |
| public interface Hockey extends Sports, Event |

|  |
| --- |
| **interface A {**  **public void test();**  **public void test1();**  **}**  interface B {  public void test();  public void test2();  }  interface C extends A,B {  public void test3();  }  class D implements C {  public void test() {  System.out.println("Testing  ");  }  public void test1() {  System.out.println("Testing1  ");  }  public void test2() {  System.out.println("Testing2  ");  }  public void test3() {  System.out.println("Testing3");  }  }  public class Main {  public static void main(String[] args) {  D d=new D();  d.test();  d.test1();  d.test2();  d.test3();  }  }  **Output:**  Testing  Testing1  Testing2  Testing3 |

**Implementing Interfaces**

|  |
| --- |
| public class RectanglePlus  implements Relatable {  public int width = 0;  public int height = 0;  public Point origin;  // four constructors  public RectanglePlus() {  origin = new Point(0, 0);  }  public RectanglePlus(Point p) {  origin = p;  }  public RectanglePlus(int w, int h) {  origin = new Point(0, 0);  width = w;  height = h;  }  public RectanglePlus(Point p, int w, int h) {  origin = p;  width = w;  height = h;  }  // a method for moving the rectangle  public void move(int x, int y) {  origin.x = x;  origin.y = y;  }  // a method for computing  // the area of the rectangle  public int getArea() {  return width \* height;  }    // a method required to implement  // the Relatable interface  public int isLargerThan(Relatable other) {  RectanglePlus otherRect = (RectanglePlus)other;  if (this.getArea() < otherRect.getArea())  return -1;  else if (this.getArea() > otherRect.getArea())  return 1;  else  return 0;  }  } |

Because RectanglePlus implements Relatable, the size of any two RectanglePlus objects can be compared.

1. **EXCEPTION HANDLING IN JAVA**

An exception is an error event that can happen during the execution of a program and disrupts its normal flow. Java provides a robust and object-oriented way to handle exception scenarios known as Java Exception Handling. Exceptions in Java can arise from different kinds of situations such as wrong data entered by the user, hardware failure, network connection failure, or a database server that is down. The code that specifies what to do in specific exception scenarios is called exception handling.

There are many typical causes for exceptions in Java, including:

* Loss of network connectivity
* Invalid input data
* Requests for missing or non-existent files
* Exceeding memory limits for the Java Virtual Machine (JVM)
* Code errors

Officially, exceptions are distinct from errors because errors are more serious issues that the application “should not try to catch.” Most developers, however, consider errors to be just a subset of exceptions.

Errors encountered during runtime create exception objects containing basic information about the exception, including the type of exception generated and the system’s state at the time of the exception. The method then throws the exception to the runtime system for processing. After receiving an exception, the runtime system tries to find a way to resolve it. Tracing backward through the call stack from the method where the exception occurred, the runtime system looks for an existing method that can process the exception. These methods are called exception handlers. If no method in the call stack can handle the exception, both the runtime system and the application stop.

Exception handling is crucial for the proper functioning of your applications. With exception handling, developers define the steps for addressing compilation or runtime errors to ensure that there are no interruptions of program flow. As a result, applications are more stable and have a better user experience. And because coding errors that lead to exceptions can be points of vulnerability for cyberattacks, effective exception handling can lead to more secure code.

**Difference between Exception and Error**

**A screenshot of a computer error

Description automatically generated**

**Keywords in Exception handling**

**try Block**

Enclose the code that might throw an exception within a try block. If an exception occurs within the try block, that exception is handled by an exception handler associated with it. The try block contains at least one catch block or finally block. The syntax of the try-catch block:

|  |
| --- |
| try{  //code that may throw exception  }catch(Exception\_class\_Name ref){} |

The syntax of a try-finally block:

|  |
| --- |
| try{  //code that may throw exception  }finally{} |

**Example:**

|  |
| --- |
| try {  int result = 10 / 0;  } catch (ArithmeticException e) {  System.out.println(e.getMessage());  } |

In the above code, dividing by zero will cause an ArithmeticException. The statements inside the try block are where we anticipate this error might occur.

**Nested try block**

The try block within a try block is known as a nested try block in java. Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

|  |
| --- |
| public class NestedTryBlock {  public static void main(String args[]) {  try {  try {  System.out.println(" This gives divide by zero error");  int b = 39 / 0;  } catch (ArithmeticException e) {  System.out.println(e);  }  try {  System.out.println(" This gives Array index out of bound exception");  int a[] = new int[5];  a[5] = 4;  } catch (ArrayIndexOutOfBoundsException e) {  System.out.println(e);  }    System.out.println("other statement");  } catch (Exception e) {  System.out.println("handeled");  }    System.out.println("normal flow..");  }  } |

**catch Block**

Java catch block is used to handle the Exception. It must be used after the try block only. You can use multiple catch blocks with a single try.

|  |
| --- |
| try  {  //code that cause exception;  }  catch(Exception\_type e)  {  //exception handling code  } |

|  |
| --- |
| public class Arithmetic {  public static void main(String[] args) {  try {  int result = 30 / 0; // Trying to divide by zero  } catch (ArithmeticException e) {  System.out.println("ArithmeticException caught!");  }  System.out.println("rest of the code executes");  }  }  **Output:**  ArithmeticException caught! |

**Multi-catch Block**

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more catch clauses, each catching a different type of exception. When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed. After one catch statement executes, the others are bypassed, and execution continues after the try/catch block.

Example: Consider a scenario where we want to parse an integer from a string and then use that integer as an array index. Both of these operations can throw exceptions - NumberFormatException and ArrayIndexOutOfBoundsException.

|  |
| --- |
| public class MultiCatchExample {  public static void main(String[] args) {  String numStr = "10a"; // This will cause NumberFormatException  int[] numbers = {1, 2, 3, 4, 5};  try {  int num = Integer.parseInt(numStr); // Parsing integer from string  System.out.println(numbers[num]); // Accessing array element  } catch (NumberFormatException | ArrayIndexOutOfBoundsException e) {  System.out.println("An error occurred: " + e.getMessage());  }  }  }  **Output:**  An error occurred: For input string: "10a" |

**throw Keyword**

The throw keyword is used to explicitly throw an exception from a method or any block of code. We can throw either checked or unchecked exceptions using the throw keyword. The throw keyword is followed by an instance of the exception.

throw exception\_instance;

Let's consider a simple example where we have a method setAge() that sets the age of a person. If someone tries to set a negative age, it's clearly an incorrect value. In such a case, we can throw an IllegalArgumentException.

|  |
| --- |
| public class ThrowExample {  private int age;  public void setAge(int age) {  if (age < 0) {  throw new IllegalArgumentException("Age cannot be negative!");  }  this.age = age;  }  public static void main(String[] args) {  ThrowExample person = new ThrowExample();    try {  person.setAge(-5); // This will cause an exception  } catch (IllegalArgumentException e) {  System.out.println("Error: " + e.getMessage());  }  }  }  Output:  Error: Age cannot be negative! |

**throws Keyword**

The throws keyword is used to declare exceptions. It doesn’t throw an exception but specifies that a method might throw exceptions. It's typically used to inform callers of the exceptions they might encounter.

|  |
| --- |
| return\_type method\_name() throws exception\_class\_name{  //method code  } |

|  |
| --- |
| public class ExceptionHandlingWorks {  public static void main(String[] args) {  exceptionHandler();  }  private static void exceptionWithoutHandler() throws IOException {  try (BufferedReader reader = new BufferedReader(new FileReader(new File("/invalid/file/location")))) {  int c;  // Read and display the file.  while ((c = reader.read()) != -1) {  System.out.println((char) c);  }  }  }  private static void exceptionWithoutHandler1() throws IOException {  exceptionWithoutHandler();  }  private static void exceptionWithoutHandler2() throws IOException {  exceptionWithoutHandler1();  }  private static void exceptionHandler() {  try {  exceptionWithoutHandler2();  } catch (IOException e) {  System.out.println("IOException caught!");  }  }  } |

**finally Block**

Java finally block is a block that is used to execute important code such as closing connection, stream, etc. Java finally block is always executed whether an exception is handled or not. Java finally block follows try or catch block. For each try block, there can be zero or more catch blocks, but only one finally block. The finally block will not be executed if the program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).

|  |
| --- |
| try {  // Code that might throw an exception  } catch (ExceptionType1 e1) {  // Code to handle ExceptionType1  } catch (ExceptionType2 e2) {  // Code to handle ExceptionType2  }  // ... more catch blocks if necessary ...  finally {  // Code to be executed always, whether an exception occurred or not  } |

In this example, we have used FileInputStream to read the simple.txt file. After reading a file the resource FileInputStream should be closed by using finally block.

|  |
| --- |
| public class FileInputStreamExample {  public static void main(String[] args) {  FileInputStream fis = null;  try {  File file = new File("sample.txt");  fis = new FileInputStream(file);  int content;  while ((content = fis.read()) != -1) {  // convert to char and display it  System.out.print((char) content);  }  } catch (IOException e) {  e.printStackTrace();  } finally {  if (fis != null) {  try {  fis.close();  } catch (IOException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  }  }  }  } |

A diagram of a method

Description automatically generated

**Fig 2: Complete picture of exception handling**

1. **TYPES OF EXCEPTION**

**A diagram of a type of exception

Description automatically generated**

**Fig 3: Hierarchy of exceptions**

There are mainly two types of exception in Java:

* Built-in Exception
* User Defined Exception

**Java Built-in Exceptions**

Built-in Exceptions are those exceptions that are pre-defined in Java Libraries. These are the most frequently occurring Exceptions. An example of a built-in exception can be ArithmeticException; it is a pre-defined exception in the Exception class of java.lang package. These can be further divided into two types:

* Checked Exception
* Unchecked Exception

**Checked Exceptions**

Checked exceptions are caught at compile time, indicating potentially recoverable errors. The compiler enforces handling them before runtime. For instance, accessing a missing file like "file.txt" can throw a FileNotFoundException, which can be handled using the throws keyword to specify potential exceptions at compile time.

**Class Not Found Exception:** The ClassNotFoundException occurs when the Java Virtual Machine cannot locate a required class, typically triggered by functions like Class.forName() or ClassLoader.loadClass().

|  |
| --- |
| public class classNotFound  {  static String classname = "missingClass";  public static void main() throws ClassNotFoundException  {  Class.forName(classname);  }  }  **Output:**  java.lang.ClassNotFoundException: missingClass |

The Class.forName() function returns the object of the class or interface whose name is passed in the parameter as a string. Now, if there is no class with the given name, then this will cause the ClassNotFoundException and terminate the execution of our code.

**Unchecked Exceptions**

An Unchecked Exception is an exception that occurs during runtime, often due to logical errors or improper usage of functions. These exceptions, also known as Runtime Exceptions, don't require explicit declaration using the throws keyword and can lead to bugs or unexpected behavior in code. Arithmetic Exception, such as division by zero, is a typical example of an unchecked exception.

**Arithmetic Exceptions**

An ArithmeticException is thrown when the code does the wrong arithmetic or mathematical operation while executing. Divide by 0 is the most common type of wrong mathematical operation.

|  |
| --- |
| class arithmeticException  {  public static void main()  {  int a = 10, b = 0;  int c = a / b;  }  }  **Output:**  java.lang.ArithmeticException: / by zero |

**Java User-Defined Exceptions**

In Java, besides using the Built-in Exception, if we want, we can create our Exceptions with messages and conditions for JVM to understand when to throw them. User-defined Exceptions are also called custom exceptions since they are not predefined and can be altered by the programmer. It is a beneficial tool to debug programs and handle edge cases without terminating the execution of the whole code. To create a user-defined exception in Java, follow these steps:

1. Create a new class that extends the Exception class:

|  |
| --- |
| class MyException extends Exception {  // Constructors can be added here  } |

Optionally, define constructors for your exception class. For example, a default constructor:

|  |
| --- |
| class MyException extends Exception {  public MyException() {  super();  }  } |

Or a parameterized constructor to provide more details about the exception:

|  |
| --- |
| class MyException extends Exception {  public MyException(String message) {  super(message);  }  } |

To raise this exception, create an instance of your custom exception and throw it using the throw keyword:

|  |
| --- |
| public class Main {  public static void main(String[] args) {  try {  throw new MyException("This is my custom exception message.");  } catch (MyException e) {  System.out.println("Caught custom exception: " + e.getMessage());  }  }  } |

By following these steps, you can create and use your custom exceptions in Java to handle specific situations that are not covered by built-in exceptions.

|  |
| --- |
| class customException extends Exception {  public customException (String s)  {  // s is the message of the exception  super(s);  }  }  class customexp{  public static void main()  {  try {  int x = -10;  // conditions for throwing the exception  if(x<0)  throw new customException("This is the message for custom exception");  }  catch (customException e) {  System.err.print("Exception!! ");  System.err.println(e.getMessage());  }  }  }  **Output:**  Exception!! This is the message for the custom exception |

Here, we have made a custom exception by extending the Java Exception class. We have also defined a String to pass a message to display if the exception is thrown. Since x is negative, the code is throwing the exception. It is also printing the message.