Java Day7

Exception Handling

- Similarly, when we write programs as part of an application, we may have to visualize the challenges that can disrupt the normal flow of execution of the code.
- Once we know what are the different situations that can disrupt the flow of execution, we can take preventive measures to overcome these disruptions.
- In java, this mechanism comes in the form of Exception Handling.

What is an Exception?

- In procedural programming, it is the responsibility of the programmer to ensure that the programs are error-free in all aspects
- Errors have to be checked and handled manually by using some error codes
- But this kind of programming was very cumbersome and led to **spaghetti code**
- Java provides an excellent mechanism for handling runtime errors

- An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions
- The ability of a program to intercept run-time errors, take corrective measures and continue execution is referred to as exception handling

What is an Exception? (Contd.).

- There are various situations when an exception could occur:
  - Attempting to access a file that does not exist
  - Inserting an element into an array at a position that is not in its bounds
  - Performing some mathematical operation that is not permitted
  - Declaring an array using negative values

Uncaught Exceptions

```
class  Demo {
  public static void main(String args[]) {
    int x = 0;
    int y = 50/x;
    System.out.println("y  = " +y);
  }
 }
```

Although this program will compile, but when you execute it, the Java run-time-system will generate an exception and displays the following output on the console :

```
java.lang.ArithmeticException: / by zero
at Demo.main(Demo.java:4)
```

Exception Handling Techniques

- There are several built-in exception classes that are used to handle the very fundamental errors that may occur in your programs
- You can create your own exceptions also by extending the **Exception** class
- These are called user-defined exceptions, and will be used in situations that are unique to your applications

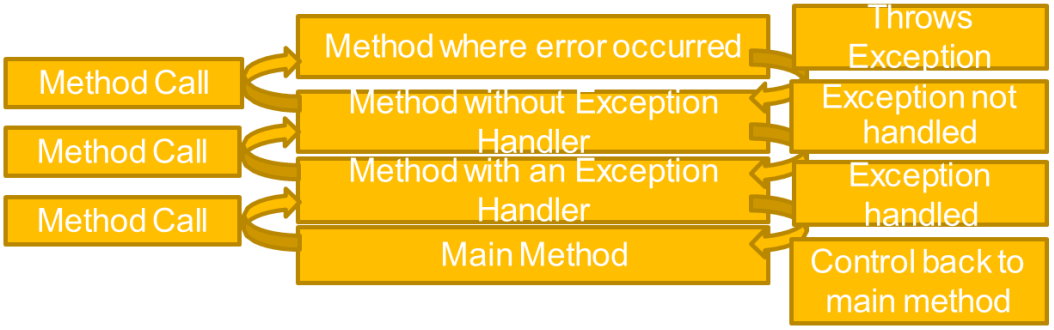
Handling Runtime Exceptions

- Whenever an exception occurs in a program, an object representing that exception is created and thrown in the method in which the exception occurred
- Either you can handle the exception, or ignore it
- In the latter case, the exception is handled by the Java run-time-system and the program terminates
- However, handling the exceptions will allow you to fix it, and prevent the program from terminating abnormally

Advantages - Exceptions

1. Separating Error-Handling Code from "Regular" Code

2. Propagating Errors Up the Call Stack



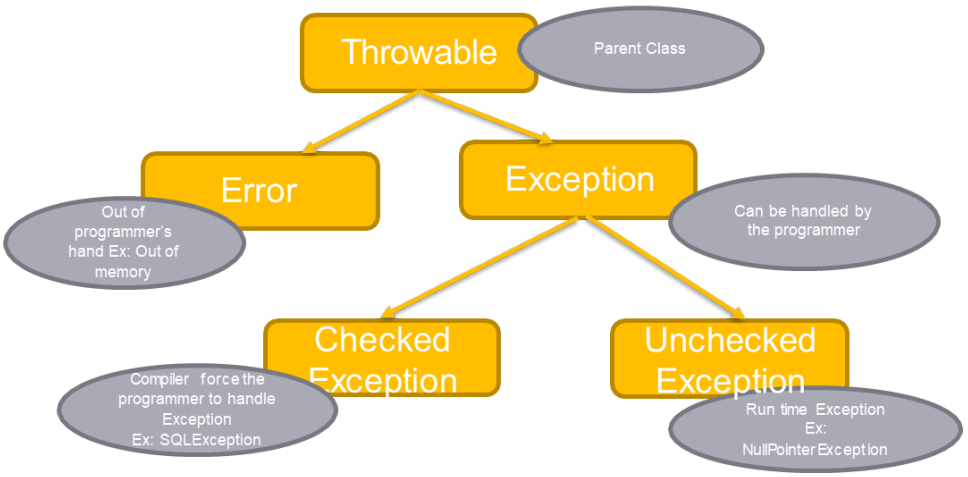
3. Grouping and Differentiating Error Types

Exception Handling Keywords

Java's exception handling is managed using the following keywords: **try, catch, throw, throws and finally.**

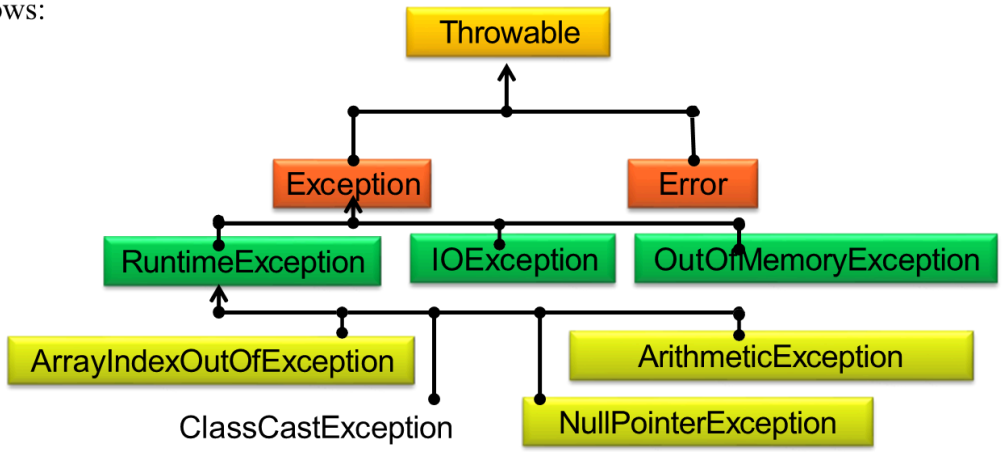
```
try {
  // code comes here
  }
catch(TypeofException  obj) {
   //handle the exception
    }
finally  {
            //code to be executed before the program ends
  }
```

Exception in a Nutshell

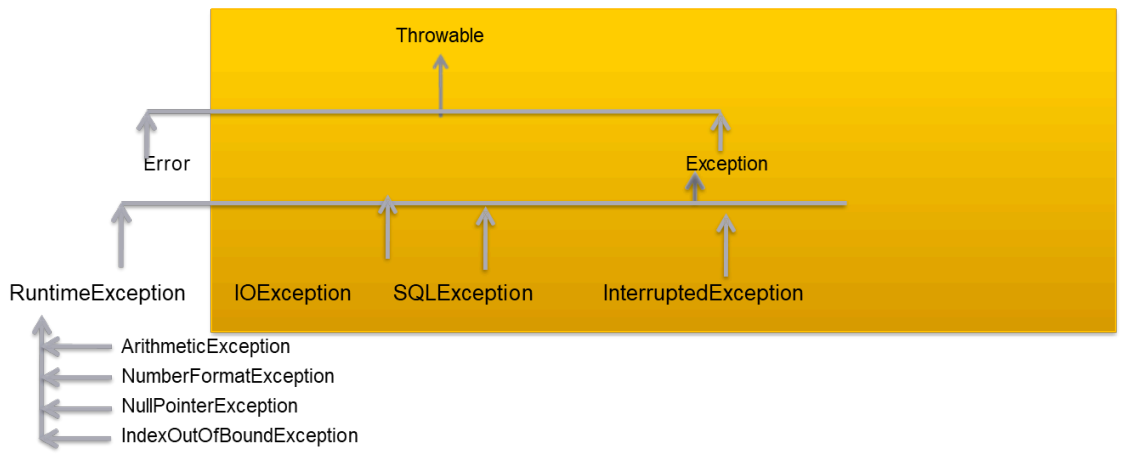


Exception Types

Exceptions are implemented in Java through a number of classes. The exception hierarchy is as follows:



Checked Exceptions



- A checked exception is an exception that usually happens due to user error or it is an error situation that cannot be foreseen by the programmer
- A checked exception must be handled using a try or catch or at least declared to be thrown using throws clause. Non compliance of this rule results in a compilation error

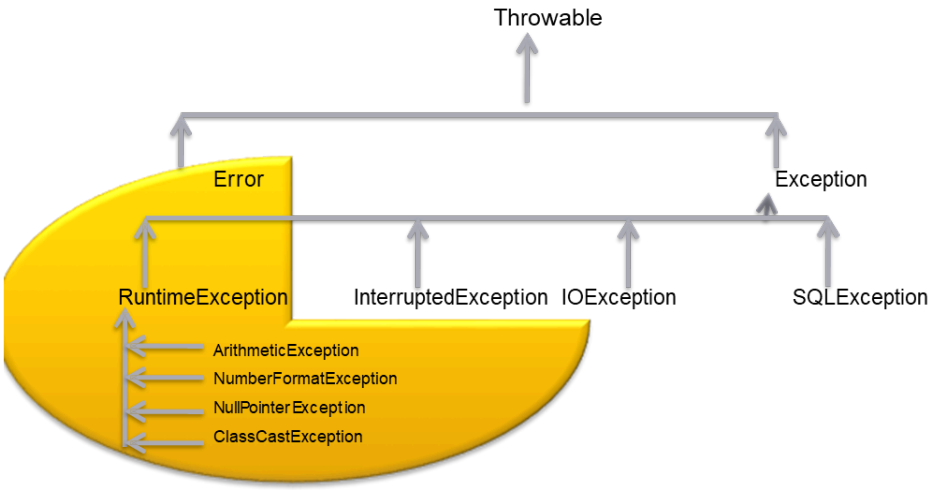
Ex: FileNotFoundException

- If you try to open a file using

```
FileInputStream fx = new FileInputStream("A1.txt");
```

- During execution, the system will throw a FileNotFoundException, if the file A1.txt is not located, which may be beyond the control of a programmer

Unchecked Exception



- An unchecked exception is an exception, which could have been avoided by the programmer

- The class RuntimeException and all its subclasses are categorized as Unchecked Exceptions

- If there is any chance of an unchecked exception occuring in the code, it is ignored during compilation

Error

- Error is not considered as an Exception
- Errors are problems that arise beyond the control of the programmer or the user
- A programmer can rarely do anything about an Error that occurs during the execution of a program
- This is the precise reason Errors are typically ignored in the code
- Errors are also ignored by the compiler
- Ex : Stack Overflow

Example – StackOverflowError

```
public class Tester {

public static void recursivePrint(int num) {
System.out.println("Number: " + num);

if(num == 0)
return;
else
recursivePrint(++num);
}

public static void main(String[] args) {
Tester.recursivePrint(1);
}
}
```

Exception in thread "main" java.lang.StackOverflowError

Try-Catch Block

- Any part of the code that can generate an error should be put in the **try** block
- Any error should be handled in the c**atch** block defined by the **catch** clause
- This block is also called the **catch block**, or the **exception handler**
- The corrective action to handle the exception should be put in the **catch** block

How to Handle exceptions?

```
class ExceptDemo{
  public static void main(String args[]){
    int x, a;
    try{
        x = 0;
        a = 22 / x;
        System.out.println("This will be bypassed.");
    }
    catch (ArithmeticException e){
        System.out.println("Division by zero.");
    }
    System.out.println("After catch statement.");
  }
}
```

Multiple Catch Statements

- A single block of code can raise more than one exception
- You can specify two or more **catch** clauses, each catching a different type of execution
- 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

Multiple Catch Statements (Contd.).

```
class MultiCatch{
  public static void main(String args[]){
    try{
        int l = args.length;
        System.out.println("l = " +l);
        int b = 42 / l;
        int arr[] = { 1 };
        arr[22] = 99;
    }
    catch(ArithmeticException e){
        System.out.println("Divide by 0: "+ e);
    }
```

Multiple Catch Statements (Contd.).

```
    catch(ArrayIndexOutOfBoundsException e){
       System.out.println("Array index oob: "+e);
    }
    System.out.println("After try/catch blocks.");
  }
}
```

Multiple Catch Statements involving Exception Superclasses & Subclasses

- When you use multiple catch statements, it is important to remember that exception subclasses must come before any of their exception superclasses
- This is because a catch statement that uses a superclass will catch exceptions of that type as well as exceptions of its subclasses
- Thus, a subclass exception would never be reached if it came after its superclass that manifests as an **unreachable code error**

Nested try Statements

- The **try** statement can be nested
- If an inner **try** statement does not have a **catch** handler for a particular exception, the outer block's catch handler will handle the exception
- This continues until one of the **catch** statement succeeds, or until all of the nested **try** statements are exhausted
- If no catch statement matches, then the Java runtime system will handle the exception

Syntax

```
try
{
    statement 1;
    statement 2;
    try
    {
        statement 1;
        statement 2;
    }
    catch(Exception e)
    {
    }
}
catch(Exception e)
{
}
```

Example for nested try

```
class Nested_Try{
 public static void main(String args[]){
  try{
         try{
               System.out.println("Arithmetic Division");
            int b =39/0;
         } catch(ArithmeticException e){
               System.out.println(e);
          }
        try{
         int a[]=new int[5];
        System.out.println("Accessing Array Elements");
         a[5]=4;
     } catch(ArrayIndexOutOfBoundsException       {
          e)  System.out.println(e);
     }
     System.out.println ("Inside Parent try");
  } catch(Exception e) {
                              System.out.println("Exception caught");
     }
System.out.println("Outside Parent try");
 }
 }
```

Using throws

- Sometimes, a method is capable of causing an exception that it does not handle
- Then, it must specify this behavior so that callers of the method can guard themselves against that exception
- While declaring such methods, you have to specify what type of exception it may throw by using the **throws** keyword
- A **throws** clause specifies a comma-separated list of exception types that a method might throw:
  - type method-name(parameter list) throws exception-list

Using throws (Contd.).

```
class ThrowsDemo{
  static void throwOne(){
    System.out.println("Inside throwOne.");
    throw new FileNotFoundException();
  }
  public static void main(String args[]){
    throwOne();
  }
}
```

What happens when this code is compiled ?

Compilation Error……..why?

Implementing throws

```
import java.io.*;
class ThrowsDemo{
  static void throwOne() throws FileNotFoundException{
    System.out.println("Inside throwOne.");
    throw new FileNotFoundException();
  }
  public static void main(String args[]) {
    try{
        throwOne();
    }
    catch (FileNotFoundException e){
        System.out.println("Caught " + e);
    }
  }
}
```

Rule governing overriding method with throws

- The overriding method must NOT throw checked exceptions that are new or broader than those declared by the overridden method

For eg : A method that declares(throws) an SQLException cannot be overriden by a method that declares an IOException, Exception or any other exception unless it is a subclass of SQLException

- In other words, if a method declares to throw a given exception, the overriding method in a subclass can only declare to throw the same exception or its subclass

- This rule does not apply for unchecked exceptions

Using throw

- System-generated exceptions are thrown automatically

- At times you may want to throw the exceptions explicitly which can be done using the **throw** keyword

- The exception-handler is also in the same block

- The general form of throw is:
  - throw **ThrowableInstance**

- Here, **ThrowableInstance** must be an object of type **Throwable**, or a subclass of **Throwable**

Using throw (Contd.).

```
class ThrowDemo{
  public static void main(String args[]) {
    try {
          int age=Integer.parseInt(args[0]);
          if(age < 18)
             throw new ArithmeticException();
         else
             if(age >=60)
                 throw new ArithmeticException("Employee is retired");
    }
  catch(ArithmeticException e) {
    System.out.println(e);
  }
  System.out.println("After Catch");
    }
}
```

Match the following :

Match the content of Column A with the most appropriate content from column B :

Column A	Column B
a) An exception is	a) Used to throw an exception explicitly
b) Throwable	b) Caused by Dividing an integer by zero
c) ArithmeticException is	c) An event that can disrupt the normal flow of instructions
d) Catch Block	d) This class is at the top of exception hierarchy
e) “throw” is	e) Exception Handler

User Defined Exceptions

- Java provides extensive set of in-built exceptions
- But there may be cases where we may have to define our own exceptions which are application specific

  For ex: If we have are creating an application for handling the database of eligible voters, the age should be greater than or equal to 18

  In this case, we can create a user defined exception, which will be thrown in case the age entered is less than 18

- While creating user defined exceptions, the following aspects have to be taken care :
  - The user defined exception class should extend from the Exception class and its subclass
  - If we want to display meaningful information about the exception, we should override the toString() method

Example on User Defined Exceptions

```
class  MyException extends Exception  {
  public MyException() {
    System.out.println("User defined Exception thrown");
  }
  public String toString() {
    return "MyException Object : Age cannot be < 18 " ;
  }
}
```

Contd..

```
class MyExceptionDemo{
  static int flag=0;
  public static void main(String args[]) {
   try {
       int age=Integer.parseInt(args[0]);
       if(age < 18)
           throw new MyException();
   }
   catch(ArrayIndexOutOfBoundsException e) {
       flag=1;
       System.out.println("Exception : "+ e);
   }

   catch(NumberFormatException e) {
       flag=1;
       System.out.println("Exception : "+ e);
   }
   catch (MyExceptionClass e) {
       flag=1;
       System.out.println("Exception : "+ e);
   }
   if(flag==0)
       System.out.println("Everything is fine");
  }
}
```

Using finally

- When an exception occurs, the execution of the program takes a non-linear path, and could bypass certain statements
- A program establishes a connection with a database, and an exception occurs
- The program terminates, but the connection is still open
- To close the connection, **finally** block should be used
- The finally block is guaranteed to execute in all circumstances

```
import java.io.*;
class FinallyDemo{
static void funcA() throws FileNotFoundException
{
    try{
        System.out.println("inside funcA( )");
        throw new FileNotFoundException( );
    }
    finally{
        System.out.println
        ("inside finally of funA( )");
  }
}


  static void funcB(){
    try{
        System.out.println("inside funcB( )");
    }
    finally{
        System.out.println
        ("inside finally of funB( )");
    }
  }
}


static void funcC() {
  try{
      System.out.println("inside funcC( )");
  }
  finally{
      System.out.println
      ("inside finally of funcC( )");
  }
}
```

```
  public static void main(String args[]){
    try{
        funcA();
    }
    catch (Exception e){
        System.out.println("Exception caught");
    }
    funcB( );
    funcC( );
  }
}
```

Java 8 Features

1. Lambda Expression
2. Functional Interfaces
3. Default Methods
4. Predicates
5. Functions
6. Double colon operator(::)
7. Stream API
8. Data and Time API

1. How to write Lambda Expression?

public void m1(){
sop(”Hello”);
}
	() → { sop(”Hello”); }

Lambda Expression

⇒ Anonymous Function

⇒ Doesn’t have any modifier

⇒ Doesn’t have any return type

public void sum(int a,int b){
sop(a+b);
}
	(int a,int b) → { sop(a+b); }

public int m1(String s){
return s.length();
}
	(String s) → { return s.length(); }

Rules to optimize the Lambda Expression:

1. If we have single line of code we can skip curly braces

public void m1(){
sop(”Hello”);
}
	() → sop(”Hello”);

2. Type Inference : Java compiler’s ability to look at each method invocation to determine the type argument(or arguments) that make the invocation applicable.

public void sum(int a,int b){
sop(a+b);
}
	(a,b) → sop(a+b);

3. If we are passing only one argument no need of include that in a paranthesis (String s) → String s.

public int m1(String s){
return s.length();
}
	s → s.length();

2. Functional Interfaces

An interface with only one abstract method

-@FunctionalInterface

```
interface Sample{
    public abstract void m1();
    public abstract void m2();
}
```

This is a normal java interface. In order to make it as functional interface, Java people introduced annotation.

```
interface Sample{
    public abstract void m1();
}
```

Example in multi threading there is a interface available called runnable which is holding only one abstract method.

Annotation - @FunctionalInterface

```
@FunctionalInterface //To ensure that the user had defined proper function
al interface.
interface Sample{
    public abstract void m1();
}
```

Rules:

1. If an interface extends Functional Interface and child interface doesn't contain any abstract method then child interface is also functional Interface
2. In child interface we can define same method[no CE:]
3. In the child interface we can't define any new abstract if we want to make it FI

```
//Without Lambda Expression

interface Inter{
    public abstract void m1();
}
```

```
class A implements Inter{
    @Override
    public void m1(){
        System.out.println("Hello - m1 ");
    }
}

public class Main{
    A a=new A();
    a.m1();
}
```

```
//With Lambda Expression
@FunctionalInterface
interface Inter{
    public abstract void m1();
}

public class Main{
    Inter i = () → System.out.println("Hello - M1 Good Nigh");
}
```

```
//Another Example
@FunctionalInterface
interface Inter{
    public abstract void sum(int a,int b);
}
public class Addition{
    public static void main(String[] args){
        Inter i = (a,b) → System.out.println(a+b);
        i.sum(10,10);
```

1. If an interface extends Functional Interface and child interface doesn’t contain any abstract method then child interface is also functional Interface

```
@FunctionalInterface
interface B{
    public abstract void m1();
}

interface A extends B{
    //public abstract void m2();
}
```

2. In child interface we can define same method[no CE:]
3. In the child interface we can't define any new abstract if we want to make it FI

Variable inside lambda expression

```
//Another Example
@FunctionalInterface
/*interface Inter{
    public abstract void sum(int a,int b);
}*/
interface Inter{
    public abstract void m1();
}
class A{
    int x=100;
    public void show(){
        int x=1000;
        Inter i =() -> {
                System.out.println(this.x);
                System.out.println(x);
        };
        i.m1();
    }
}
public class Addition{
    public static void main(String[] args){
```

```
        A a=new A();
        a.show();
    }
}
```

3. Default Methods in functional interface:

🧩 What Are Default Methods?

In Java 8 and later, **interfaces** can include methods with a default implementation using the `default` keyword. This was introduced to support **backward compatibility** and enable **functional programming** features like **lambda expressions**.

✅ Can Functional Interfaces Have Default Methods?

Yes! A **functional interface** can have **default methods**, as long as it has **exactly one abstract method** (the SAM). Default methods don't count toward this because they already have an implementation

```
@FunctionalInterface
interface Calculator {
    // Single Abstract Method (SAM)
    int operate(int a, int b);

    // Default method with implementation
    default void showOperation(String opName) {
        System.out.println("Performing: " + opName);
    }

    // Another default method
    default boolean isPositive(int result) {
        return result > 0;
    }
}
```

🧠 Why Use Default Methods?

- **Backward compatibility**: Add new methods to interfaces without breaking existing implementations.
- **Code reuse**: Share common logic across multiple implementations.
- **Cleaner design**: Avoid utility classes for trivial behaviors.

4. Predicates 1.8

1. Predicate is a interface present in Java.util.function package
2. A Predicate contains a function with a single argument and returns boolean value

Syntax:

```
interface Predicate<T>
{
   public boolean test(T t);
}
```

1. Write a predicate to check whether the given integer is greater than 50 or not.
2. Write a predicate to check the length of given string is greater than 5 or not.

```
class Test{
   public static void main(String[] args){
      Predicate<Integer> p=I->I>50;
      System.out.println(p.test(100)); //true
      Predicate<String> p1=s->s.length()>5;
      System.out.println(p1.test("Happy Morning"));//true
   }
}
```

Predicate Joining

we can join two or more predicate by using following methods

1. and()
2. or()
3. negate()

```
class Test{
    public static void main(String[] args){
        int[] x={1,2,3,4,5,6};
        Predicate<Integer> p1=I->I%2==0;
        Predicate<Integer> p2=I->I>2;
        System.out.println("Print all even numbers:");
        m1(p1,x);
        System.out.println("Print all numbers greater than 2: ");
        m1(p2,x);
        System.out.println("Print all numbers greater than 2 and even: ");
        m1(p2.and(p1),x);
        System.out.println("Print all odd numbers:");
        m1(p1.negate(),x);
        System.out.println("Print all numbers greater than 2 or even: ");
        m1(p2.or(p1),x);
    }
    public static void m1(Predicate<Integer> p,int[] x){
        for(int i:x){
            if(p.test(i)){
                System.out.print(i+" ");
            }
    }
```

5. Functions

1. Function is exactly same as predicates except that function can return any type of result but function should (can) return only one value and that value can be any type as per our requirement.
2. Function interface present in java.util.function package.
3. Function interface contains only one method i.e. apply()

Syntax:

```
interface function(T,R){
    public R apply(T t);
}
```

```
//Write a function to return string length
public class Test{
    public static void main(String[] args){
        Function<String, Integer> i=s->s.length();
        System.out.println(i.apply("Hello"));
    }
}
```

Difference between predicate and function

Feature	`Predicate<T>`	`Function<T, R>`
Purpose	Tests a condition (returns `boolean`)	Transforms input `T` to output `R`
Method Signature	`boolean test(T t)`	`R apply(T t)`
Generic Parameters	One: `<T>`	Two: `<T, R>`
Return Type	`boolean`	Generic type `R`
Use Case	`.filter()` in streams	`.map()` in streams
Example	`Predicate<String> p = s -> s.isEmpty();`	`Function<String, Integer> f = s -> s.length();`

6. Stream API

1. what is the difference between stream and java.io

   Ans: java.io → save data into files. stream→ when we are working with group of object(collection)

2. what is the difference between collection and stream

   Ans: collection → group of objects stored in a single location

```
import java.util.stream.Stream;
public class Test{
   public static void main(String[] args){
      ArrayList<Integer> l1=new ArrayList<Integer>(Arrays.asList(10,20,30,4
0,50));
      System.out.println(l1);
      Stream s=l1.stream();
      /*
      Stream → Interface, present in java.util.stream
      1. We got stream by invoke stream() on top of collection object
      2. We can process the object in two steps
         1. Configuration
            a. filter → to filter the objects on the basis of boolean condition
            b. map → If we want to make a new object for every present in c
ollection
         2. Processing
            a. count()
            b. min()
            c. max()
            d. toArray()

      */
      Stream s1=s.filter(predicate);

   }
}
```

```
import java.util.stream.Stream;
public class Test{
   public static void main(String[] args){
      ArrayList<Integer> l1=new ArrayList<Integer>(Arrays.asList(10,20,30,4
0,50));
      System.out.println(l1);

      /*
      Stream → Interface, present in java.util.stream
      1. We got stream by invoke stream() on top of collection object
```

```
        2. We can process the object in two steps
            1. Configuration
                a. filter --> to filter the objects on the basis of boolean condition
                b. map --> If we want to make a new object for every present in c
ollection
            2. Processing
                a. count()
                b. min()
                c. max()
                d. toArray()

        */
        //Write a program to take out the list the number greater than 20
        List<Integer> collect=l1.stream().filter(i->i>20).collect(Collectors.toList
());
        System.out.println(collect);
        //Write a program to take out the list where we add 5 to each element.
        List<Integer> collect2= l1.stream().map(i->i+5).collect(Collectors.toList
());
        System.out.println(collect2);
        //Terminal Operations
        //1. toArray()
        Stream<Integer> s=Stream.of(1,2,3,5,6);//of method will make a strea
m of it
        Object[] array=s.toArray();
        for(Object a:array){
            System.out.print(a+" ");
        }

        //2. count()
        Stream<Integer> s1=Stream.of(1,2,3,5,6);
        Long count = s1.count();
        System.out.println(count);

        //3. forEach()
        Stream<Integer> s2=Stream.of(1,2,3,5,6);
        s2.forEach(ele -> System.out.print(ele+" ");
```

```
    //4. min() and max()
    Stream<Integer> s3=Stream.of(1,2,3,5,6);
    Optional<Integer> min = s3.min((o1,o2)-> o1.compareTo(o2));
    Optional<Integer> max = s3.max((o1,o2)-> o1.compareTo(o2));
    System.out.println("min: "+min.get());
    System.out.println("max: "+max.get());

    //5. anyMatch()
    List<Integer> listofNum=Arrays.asList(22,33,44,55,66);
    boolean anyMatch = listofNum.stream().anyMatch(i->i==2);
    System.out.println(anyMatch);//false

    //6. allMatch()
    List<Integer> list=Arrays.asList(22,33,44,55,66);
    boolean allMatch=list.stream().allMatch(ele -> ele%2==0);
    System.out.println(allMatch);//false
    }
}
```