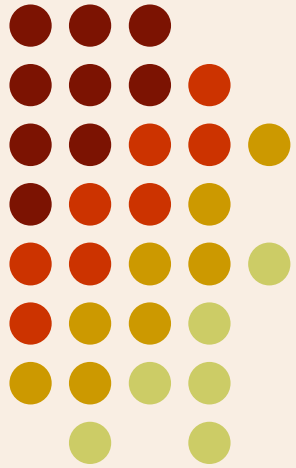
Lecture [6]

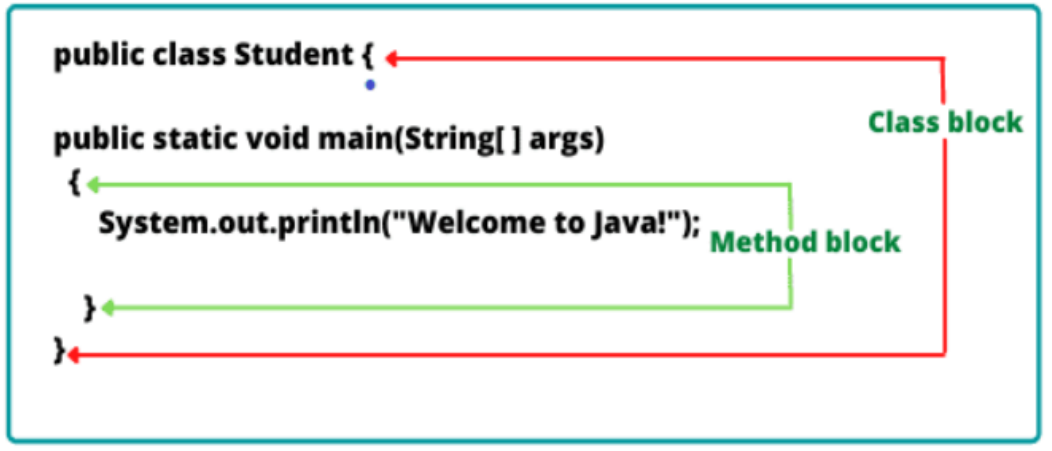
Dr. Gehad Ismail Sayed



Recap – OOP Concepts

Java Block

- A **block** in Java is a set of code enclosed within curly braces { } within any class, method, or constructor.
- It begins with an opening brace ({) and ends with an closing braces (}).
- A block can also be placed within another block that is called nested block in java.
- There are two variants of an **initialization block** in Java -
  1. Static initialization block.
  2. Instance Initialization block.



Recap – OOP Concepts

Why Initialization Block?

- In order to perform any operations while assigning values to an instance data member, an initializer block is used.
- It is used to declare/initialize the common part of various constructors of a class.

```
class Car {
    int speed;
    Car()  { System.out.println("Speed of Car: " + speed);   }
       // Block
       {
           speed = 60;
       }
    public static void main(String[] args) {
           Car obj = new Car();  }
}
```

Speed of Car: 60

Recap – OOP Concepts

Java Block - Static initialization block

- **Static initialization block** can be used to initialize class variables, which are marked with the **static** keyword.
- Moreover, a static initialization block is defined within a class, and it is **automatically called** when its class loads, i.e. when a class containing the static block is executed.
- A **static initialization block** starts with the **static** keyword.
- Ex.

```
Class A
{
    //Static Initialization Block
    static
    { System.out.println("Hello from the static block"); }
    public static void main(String... ar) { }
}
```

Recap – OOP Concepts

Java Block - Static initialization block

- **Static initialization block** can only access static variable of a class

```
class A
{
    static int i;
    static  //static initialization block
    {
        i=10;
        System.out.println("A class is loaded");
        System.out.println("value of i = "+ i);
    }
    public static void main(String... ar) { }
}
```

A class is loaded
value of i = 10

Recap – OOP Concepts

Java Block - Static initialization block

- **Static initialization block** cannot access instance variables
  - Just like a static method, a static initialization block cannot access instance variables. Doing so gives a compile error.

```
class A
{
    int a=20;
    static  //static initialization block
    {
        System.out.println("A class is loaded");
        System.out.println(a);
    }
    public static void main(String... ar) { }
}
```

```
A.java:8: error: non-static variable a
cannot be referenced from a static
context
System.out.println(a);
                   ^
1 error
```

Recap – OOP Concepts

Java Block - Static initialization block

- **Multiple static initialization blocks**
  - Just You can define multiple static initialization blocks within your class and the order in which they are defined (starting from the top) is the order in which they are executed when the class loads.

```
class A
{
    //first static initialization block
    static {  System.out.println("First hello from static block"); }
    public static void main(String... ar) { }
    //second static initialization block
    static {  System.out.println("Second hello from static block");  }
    //third static initialization block
    static {  System.out.println("Third hello from static block"); }
}
```

First hello from static block
Second hello from static block
Third hello from static block

Recap – OOP Concepts

Java Block - Static initialization block

- **Static initialization block in inheritance**
  - In inheritance, the static initialization block of a superclass is always executed before the static initialization block of a subclass.

```
class B
{
    static  //static initialization block of A
    { System.out.println("Static block of B"); }
}
class A extends B
{
    static  //static initialization block of B
    { System.out.println("Static block of A"); }
    public static void main(String... ar) { }
}
```

Static block of B
Static block of A

Recap – OOP Concepts

Java Block - Instance Initialization Block

- **The instance initialization block of a class** is associated with its instance/object creation.
- **The instance initialization block** is automatically executed when a constructor of its class is called for object creation.
- **Instance initialization block** does not precede with any keyword or name.

```
class A
{
    int a;
    {
        a=10;
        System.out.println("An object is created");
    }
    public static void main(String... ar)
    { A ob = new A();
    System.out.println(ob.a); }
}
```

An object is created
10

Recap – OOP Concepts

Java Block - Instance Initialization Block

- **Instance Initialization block is executed before constructor**
  - **The instance initialization block** is automatically executed when a constructor of its class is called for object creation.
  - **Instance initialization block** does not precede with any keyword or name.

```
class A
{
    //Instance Initialization Block
    {   System.out.println("Instance initialization block is executed");  }
    A()
    {  System.out.println("Constructor is executed");  }
    public static void main(String... ar)
    {
            A ob = new A();
    }
}
```

```
Instance initialization block is executed
Constructor is executed
```

Recap – OOP Concepts

Java Block - Instance Initialization Block

- **Instance Initialization block is called only when you call the constructor**
  - if you don't call a constructor of a class, the defined instance initialization block won't be executed.

```
class A
{
    //Instance Initialization Block
    {
        System.out.println("An object is created");
    }
    public static void main(String... ar)
    {
        int a=10;
        System.out.println(a);
    }
    }
```

10

Recap – OOP Concepts

Java Block - Instance Initialization Block

- **Multiple Instance Initialization Block**
  - You can define multiple instance initialization blocks within your class, and the order in which they are defined(starting from the top) is the order in which they are executed at the time of object creation.

```
class A
{
    {       System.out.println("An object is created");  }
    {       System.out.println("Second notification about the  object creation");  }
    public static void main(String... ar) {  A ob = new A();  }
    {       System.out.println("Third notification about the  object creation");  }
}
```

An object is created
Second notification about the object creation
Third notification about the object creation

Recap – OOP Concepts

Java Block - Instance Initialization Block

- **Instance initialization block can access instance variables and static variable**
  - Unlike the **static initialization block**, which could only access the static variables and the static methods of its class, the **instance initialization block** can access both the instance variables and the static variables of its class.

```
class A
{
    static char ch='a';
    int a=20;
    {
        System.out.println("An object of A is created");
        System.out.println("value of instance variable, a = "+ a);
        System.out.println("value of static character, ch = "+ ch);
    }
    public static void main(String... ar)
    {       A ob= new A();   } }
```

An object of A is created
Value of instance variable, a = 20
Value of static character, ch = a

Recap – OOP Concepts

Java Block - Instance Initialization Block

- **Instance initialization block in inheritance**
  - When the constructor of a subclass is called, therefore:
    1. At first, the constructor of the superclass gets called, and it finishes its execution.
    2. Next, the instance initialization block in the subclass gets executed.
    3. Finally, the constructor of the subclass completes its execution.

```
class B
{
    B() { System.out.println("Constructor of B is called"); }
}
class A extends B
{
    A() {    System.out.println("Constructor of A is called");   }
    {   System.out.println("Instance Initialization block is called");   }
    public static void main(String... ar) {     A ob = new A();    }
}
```

Constructor of B is called
Instance Initialization block is called
Constructor is A called

Recap – OOP Concepts

Java Block - Instance Initialization Block

- **Instance initialization block in inheritance**

What is the output of the following?

```
class B
{
    {    System.out.println("Instance Initialization block is called from super class");    }

    B() { System.out.println("Constructor of B is called"); }
}
class A extends B
{
    A() {    System.out.println("Constructor of A is called");   }
    {   System.out.println("Instance Initialization block is called from sub class");   }
    public static void main(String... ar) {     A ob = new A();    }
}
```

Recap – OOP Concepts

Java Block - Instance Initialization Block

- **Instance initialization block in inheritance**

What is the output of the following?

```
class B
{
    {    System.out.println("Instance Initialization block is called from super class");    }

    B() { System.out.println("Constructor of B is called"); }
}
class A extends B
{
    A() {    System.out.println("Constructor of A is called");   }
    {   System.out.println("Instance Initialization block is called from sub class");   }
    public static void main(String... ar) {     A ob = new A();    }
}
```

Instance Initialization block is called from Super Class
Constructor of B is called
Instance Initialization block is called from Sub Class
Constructor of A is called

Recap – OOP Concepts

Java Block - Instance Initialization Block

- **Instance initialization block in inheritance**

What is the output of the following?

```
class B
{
    {    System.out.println("Instance Initialization block is called from super class");    }

    B() { System.out.println("Constructor of B is called"); }
}
class A extends B
{
    A() {    System.out.println("Constructor of A is called");   }
    static {   System.out.println("Instance Initialization block is called from sub class");   }
    public static void main(String... ar) {     A ob = new A();    }
}
```

Instance Initialization block is called from sub class
Instance Initialization block is called from super class
Constructor of B is called
Constructor of A is called

Recap – OOP Concepts

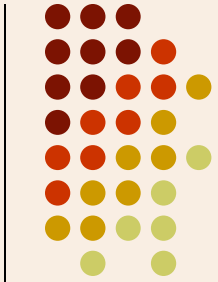
Exception

- An exception is an **unwanted** or **unexpected** event, which occurs during the execution of a program.
- i.e at run time, that **disrupts the normal flow of the program's instructions**.

```
public class Main {

    public static void main(String[] args){

        int []arr = new int[]{1,2,3};//3
        System.out.println(arr[3]);

    }
}
```

```
--- exec-maven-plugin:3.0.0:exec (default-cli) @ JavaOOP ---
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds for
        at com.mycompany.javaoop.Main.main(Main.java:9)
Command execution failed.
```

Recap – OOP Concepts



Error vs Exception

- **Errors:** represent serious and usually irrecoverable conditions like library in compatibility, infinite recursion, or memory leaks.
  - Errors are generated by the **runtime** environment
  - Ex. JVM is out of memory.
  - Normally, programs can't recover from errors.
- **Exception:** indicates conditions that a reasonable application might try to catch.
  - Exceptions are the problems which can occur at **runtime** and **compile time**.
  - It mainly occurs in the code written by the developers.

Recap – OOP Concepts

An exception can occur for many different reasons:

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

Thus the exception can occur because of the user or from the programmer itself

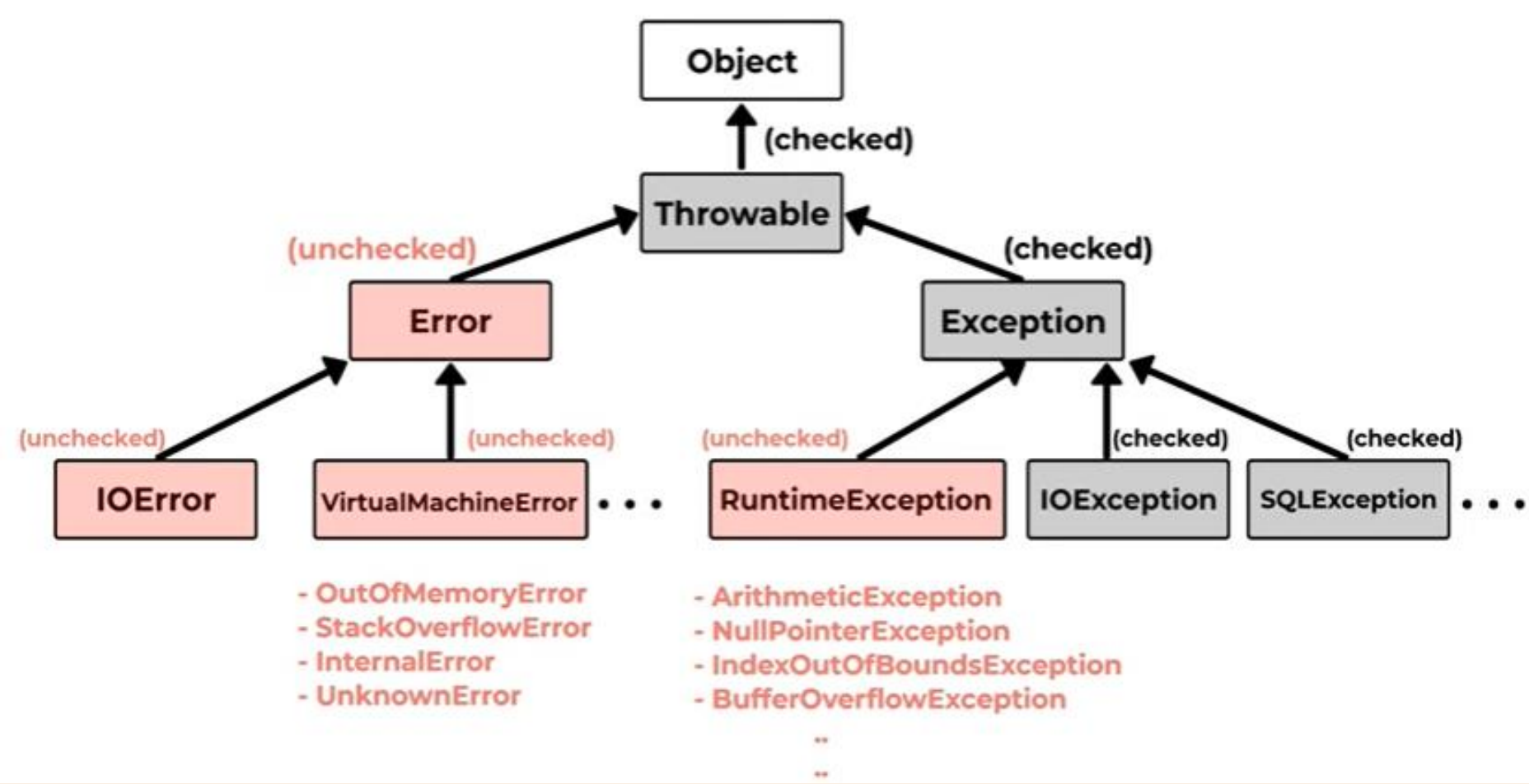
Recap – OOP Concepts

Exception Handling

- **There are three categories of exceptional conditions:**
  1. Checked exceptions (compile-time exceptions)
     - They are exceptions that are checked/notified by the compiler at the compilation-time.
     - These exceptions cannot simply be ignored and must be handled.
  2. Unchecked exceptions (Runtime exceptions)
     - These exception occurs at execution time.
     - These include programming bugs, such as logic errors or improper use of an API
     - Runtime exceptions are ignored at compilation time
  3. Errors
     - They represent serious and usually irrecoverable conditions like a library incompatibility, infinite recursion or memory leaks.

Recap – OOP Concepts

Exception Hierarchy



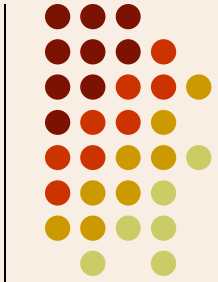
Recap – OOP Concepts

Exception vs. Error

Error and Exception both are subclasses of the java Throwable class that belongs to java.lang package

Basis of Comparison	Exception	Error
Type	It can be classified into two categories i.e. checked and unchecked.	All errors in Java are unchecked.
Occurrence	It occurs at compile time or run time.	It occurs at run time.
Causes	It is mainly caused by the application itself.	It is mostly caused by the environment in which the application is running.
Recoverable/ Irrecoverable	Exception can be recovered by using the try-catch block.	An error cannot be recovered, should not try to catch it.

Recap – OOP Concepts



Exception Handling

- In a program, if there is a chance of raising an exception then compiler always warn us about it and compulsorily, we should handle that checked exception, Otherwise we will get compile time error saying unreported exception XXX must be caught or declared to be thrown.
- To prevent this compile time error we can handle the exception in two ways:
  - By using try catch
  - By using throws keyword

Recap – OOP Concepts

Exception Handling - Unchecked

- **Array Index Out of Bounds Exception**

```
public static void main(String[] args){

    try{
        int []arr = new int[]{1,2,3};//3
        System.out.println(arr[3]);
    }catch(java.lang.ArrayIndexOutOfBoundsException e){
        System.out.println(e);
    }
```

```
ut - Run (JavaOOP)
--- maven-compiler-plugin:3.1:compile (default-compile) @ JavaOOP ---
Changes detected - recompiling the module!
Compiling 1 source file to C:\Users\Acer\Documents\NetBeansProjects\JavaOOP\target\classes

--- exec-maven-plugin:3.0.0:exec (default-cli) @ JavaOOP ---
java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds for length 3
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
```

Recap – OOP Concepts

Exception Handling - Unchecked

- **Null Pointer Exception**

```
public class Main {

    public static void main(String[] args){

        try{
            int []arr = null;//3
            System.out.println(arr[3]);
        }catch(java.lang.NullPointerException e){
            System.out.println(e);
        }
        System.out.println("1");
        System.out.println("2");
```

```
(default-cli) @ JavaOOP ---
ang.NullPointerException: Cannot load from int array because "arr" is null
Main.main(Main.java:10)
```

Recap – OOP Concepts

Exception Handling - Unchecked

- **Number Format Exception**

```
public class Main {

    public static void main(String[] args){

        int val = Integer.parseInt("String96");
        System.out.println(val);//1
        System.out.println("1");
        System.out.println("2");

    }
}
```

```
--- exec-maven-plugin:3.0.0:exec (default-cli) @ JavaOOP ---
Exception in thread "main" java.lang.NumberFormatException: For input string: "String96"
        at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:68
        at java.base/java.lang.Integer.parseInt(Integer.java:652)
        at java.base/java.lang.Integer.parseInt(Integer.java:770)
        at com.mycompany.javaoop.Main.main(Main.java:8)
Command execution failed.
```

Recap – OOP Concepts

Exception Handling - Unchecked

- **Arithmetic Exception**

```
public class Main {

    public static void main(String[] args){
        try{
            int x = 5/0;
            System.out.println(x);
        }catch(ArithmeticException ex){
            System.out.println(ex);
        }
        System.out.println("Rest of the code");
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at com.mycompany.javaoop.Main.main(Main.java:8)
Command execution failed.
```

Recap – OOP Concepts

Exception Handling - Unchecked

- **Arithmetic Exception**

What is the output of below code?

```
public class Main {

    public static void main(String[] args){
        try{
      int x = 5/0;
      System.out.println(x);
      System.out.println("Hi 1");
      System.out.println("Hi 2");
      System.out.println("Hi 3");

  }catch(ArithmeticException ex){
      System.out.println(ex);
  }
  System.out.println("Rest of the code");
```

Recap – OOP Concepts

Exception Handling - Unchecked

- **Arithmetic Exception**

What is the output of below code?

```
--- exec-maven-plugin:3.0.0:exec (default
java.lang.ArithmeticException: / by zero
Rest of the code
------------------------------------------
BUILD SUCCESS
```

```
public class Main {

    public static void main(String[] args){
        try{
        int x = 5/0;
        System.out.println(x);
        System.out.println("Hi 1");
        System.out.println("Hi 2");
        System.out.println("Hi 3");

    }catch(ArithmeticException ex){
        System.out.println(ex);
    }
    System.out.println("Rest of the code");
```

Recap – OOP Concepts

Exception Handling - Unchecked

- **Arithmetic Exception**

What is the output of below code?

```
public static void main(String[] args){
    try{
        int arr[]=new int[5];
        arr[10]=7/0;

        System.out.println("Hi 1");
        System.out.println("Hi 2");
        System.out.println("Hi 3");

    }
    catch(ArrayIndexOutOfBoundsException ex){
        System.out.println(ex);
    }
    catch(ArithmeticException ex){
        System.out.println(ex);
    }

    System.out.println("Rest of the code");
```

Note: Multi catch block
Which one called?

Recap – OOP Concepts

Exception Handling - Unchecked

- **Arithmetic Exception**

What is the output of below code?

```
--- exec-maven-plugin:3.0.0:exec (default-c
java.lang.ArithmeticException: / by zero
Rest of the code
------------------------------------------
```

```
public static void main(String[] args){
    try{
        int arr[]=new int[5];
        arr[10]=7/0;

        System.out.println("Hi 1");
        System.out.println("Hi 2");
        System.out.println("Hi 3");

    }
    catch(ArrayIndexOutOfBoundsException ex){
        System.out.println(ex);
    }
    catch(ArithmeticException ex){
        System.out.println(ex);
    }

    System.out.println("Rest of the code");
```

Recap – OOP Concepts

Exception Handling - Unchecked

- **Arithmetic Exception**

Can we combine multiple exceptions?

Yes, using Or operator (union catch)

```
    int arr[]=new int[5];
    arr[10]=7/0;

    System.out.println("Hi 1");
    System.out.println("Hi 2");
    System.out.println("Hi 3");

}
catch(ArrayIndexOutOfBoundsException | ArithmeticException ex){
    System.out.println(ex);
}
```

Recap – OOP Concepts

Exception Handling - Unchecked

What if we don't know the type of exception?

Use the parent class of exception namely "Exception"

```
catch(Exception ex){
    System.out.println(ex);
}
```

Which catch block will be executed?

```
    int arr[]=new int[5];
    arr[10]=7/0;

    System.out.println("Hi 1");
    System.out.println("Hi 2");
    System.out.println("Hi 3");

}
catch(ArrayIndexOutOfBoundsException | ArithmeticException ex){
    System.out.println("ArrayIndexOutOfBoundsException | ArithmeticException");
}
catch(Exception ex){
    System.out.println("Exception");
}
```

Recap – OOP Concepts

Exception Handling - Unchecked

Which catch block will be executed?

Note: the order is important

```
        int arr[]=new int[5];
        arr[10]=7/0;

        System.out.println("Hi 1");
        System.out.println("Hi 2");
        System.out.println("Hi 3");

    }
    catch(ArrayIndexOutOfBoundsException | ArithmeticException ex){
        System.out.println("ArrayIndexOutOfBoundsException | ArithmeticException");
    }
    catch(Exception ex){
        System.out.println("Exception");
    }
```

```
--- exec-maven-plugin:3.0.0:exec (default-cli) @ JavaOOP ---
ArrayIndexOutOfBoundsException | ArithmeticException
Rest of the code
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
```

Recap – OOP Concepts

Exception Handling - Unchecked

Note: we can't reverse the order because the parent Exception class has all types of exceptions

```
    int arr[]=new int[5];
    arr[10]=7/0;

    System.out.println("Hi 1");
    System.out.println("Hi 2");
    System.out.println("Hi 3");

}
catch(Excepti
    System.ou
}
catch(ArrayIndexOutOfBoundsException | ArithmeticException ex){
    System.out.println("ArrayIndexOutOfBoundsException | ArithmeticException");
}
```

exception ArrayIndexOutOfBoundsException has already been caught

exception ArithmeticException has already been caught

....

(Alt-Enter shows hints)

Recap – OOP Concepts

Exception Handling - Checked

```
3  import java.io.FileReader;
4
5  public class Main {
6
7      public static void main(String[] args){
8
           readFile("D:\\Adel-Info\\Name.txt");
0      }
1
2      static void readFile(String filePa
3
           FileReader reader = new FileReader(filePath);
5
6      }
7  }
```

unreported exception FileNotFoundException; must be caught or d
---
(Alt-Enter shows hints)

```
static void readFile(String filePath){

    try {
        FileReader reader = new FileReader(filePath);
    } catch (FileNotFoundException ex) {
        //Logger.getLogger(Main.class.getName()).log(Level.SEVE
        System.out.println(ex);
    }

}
```

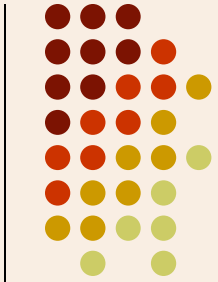
Recap – OOP Concepts

Exception Handling - Checked

```
17  static void readFile(String filePath){
18
19
20      FileInputStream fin=new FileInputStream(filePath);
21      System.out.println("file content: ");
22      int r=0;
23      while((r=fin.read())!=-1){
24          System.out.print((char)r);
25      }
26
27  }
```

```
static void readFile(String filePath){

    try{
        FileInputStream fin=new FileInputStream(filePath);

        System.out.println("file content: ");
        int r=0;
        while((r=fin.read())!=-1){
            System.out.print((char)r);
        }
    } catch(FileNotFoundException  e)  {
        System.out.println(e);
    } catch (IOException e) {
        System.out.println(e);
    }

}
```

Recap – OOP Concepts



Exception Handling: Finally Block

- **Finally block** in java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.

- A **finally block** is always get executed whether the exception has occurred or not.

- **Rule:** For each try block there can be zero or more catch blocks, but **only one finally block**.

Recap – OOP Concepts

Exception Handling: Finally Block

- **What is the output of the following?**

```
try{
    int []arr=new int[5];
    arr[7]=5;
}catch(ArrayIndexOutOfBoundsException ex){
    System.out.println(ex);
    return;
}
System.out.println("Rest of the code");
```

```
--- exec-maven-plugin:3.0.0:exec (default-cli) @ JavaOOP ---
java.lang.ArrayIndexOutOfBoundsException: Index 7 out of bounds for length 5
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
```

Recap – OOP Concepts

Exception Handling: Finally Block

- **What is the output of the following?**

```
public static void main(String[] args){

    try{
        int []arr=new int[5];
        arr[7]=5;
    }catch(ArrayIndexOutOfBoundsException ex){
        System.out.println(ex);
        return;
    }finally{
        System.out.println("Finally");
    }
    System.out.println("Rest of the code");
```

Note: Rest of the code not printed

```
Output - Run (JavaOOP)
--- exec-maven-plugin:3.0.0:exec (default-cli) @ JavaOOP ---
java.lang.ArrayIndexOutOfBoundsException: Index 7 out of bounds for length 5
Finally
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
```

Recap – OOP Concepts

Exception Handling: Finally Block

- **What is the output of the following?**

```
static int getNumber(){
    try{
        return 1;
    }catch(Exception ex){
        return 2;
    }finally{
        return 3;
    }
}
```

```
System.out.println(getNumber());
```

Why not 1?

Recap – OOP Concepts

Exception Handling: 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.

```
public static void main(String[] args){

    try{
        int []arr=new int[5];
        arr[7]=5;
    }catch(ArrayIndexOutOfBoundsException ex){
        System.out.println(ex);
        //return;
        System.exit(0);
    }finally{
        System.out.println("Finally");
    }

}
```

Recap – OOP Concepts

Exception Handling - Try with Resource

Note: **File is type of resources that needed to be closed, thus <u>finally</u> block should be used.**

```
try(FileInputStream fin=new FileInputStream(filePath);){

    System.out.println("file content: ");
    int r=0;
    while((r=fin.read())!=-1){
        System.out.print((char)r);
    }
} catch(FileNotFoundException  e)  {
    System.out.println(e);
} catch (IOException e) {
    System.out.println(e);
}
```

```
static void readFile(String filePath){
    FileInputStream fin = null;
    try{
        fin=new FileInputStream(filePath);
    System.out.println("file content: ");
    int r=0;
    while((r=fin.read())!=-1){
        System.out.print((char)r);
    }
} catch(FileNotFoundException  e)  {
    System.out.println(e);
} catch (IOException e) {
    System.out.println(e);
}finally{
    if(fin != null){
    try{
        fin.close();
    }catch(IOException e){
        System.out.println(e);
    }
}
```

Recap – OOP Concepts

Exception Handling - Try with Resource

- The **try-with-resources** statement is a try statement that declares one or more resources.
- A **resource** is an object that must be closed after the program is finished with it.
- The **try-with-resources** statement ensures that each resource is closed at the end of the statement.
- The following example reads the first line from a file. It uses an instance of **FileReader** and **BufferedReader** to read data from the file. **FileReader** and **BufferedReader** are resources that must be closed after the program is finished with it (use semicolon):

```
static String readFirstLineFromFile(String path) throws IOException {
    try (FileReader fr = new FileReader(path);
         BufferedReader br = new BufferedReader(fr)) {
        return br.readLine();
    }
}
```

Recap – OOP Concepts

Notes

- Exception Handling is mainly used to handle the checked exceptions.
- If there occurs any unchecked exception such as NullPointerException, it is programmers' fault that he is not checking the code before it being used.
- **Which exception should be declared?**
  - Ans: Checked exception only, because:
    - **unchecked exception:** under our control so we can correct our code.
    - **error:** beyond our control. For example, we are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

Recap – OOP Concepts

throws keyword vs. try-catch-finally

- There might be several methods that can cause exceptions. Writing try...catch for each method will be tedious and code becomes long and less-readable.

- **throws** is also useful when you have **checked exception** (an exception that must be handled) that you don't want to catch in your current method.

- The **throws** keyword can be useful for propagating exceptions in the call stack and allows exceptions to not necessarily be handled within the method that declares these exceptions.

Recap – OOP Concepts

Exception Handling – Java Throws Keyword

- **throws** is a keyword in Java which is used in the **signature** of method to indicate that this method might throw one of the listed type exceptions.
- **The caller to these methods has to handle the exception using a try-catch block.**
- Now **Checked Exception** can be **propagated** (forwarded in call stack).

Syntax:

type method_name(parameters) throws exception_list

- **exception_list** is a comma separated list of all the exceptions which a method might throw.

```
class ThrowsExecp
{
    static void fun() throws IllegalAccessException
    {
        System.out.println("Inside fun(). ");
        throw new IllegalAccessException("demo");
    }
    public static void main(String args[])
    {
        try
        {
            fun();
        }
        catch(IllegalAccessException e)
        {
            System.out.println("caught in main.");
        }
    }
}
```

Recap – OOP Concepts

Exception Handling – Java Throws Keyword

```
import java.io.*;
class Main {
public static void findFile() throws NullPointerException, IOException,
InvalidClassException {
    // code that may produce NullPointerException
    // code that may produce IOException
    // code that may produce InvalidClassException
}
  public static void main(String[] args) {
    try{
      findFile();
    } catch(IOException e1){
      System.out.println(e1.getMessage());
    } catch(InvalidClassException e2){
      System.out.println(e2.getMessage());
    }
  } }
```

Note that we have not handled the NullPointerException. This because it is an unchecked exception. It is not necessary to specify it in the throws clause and handle it.

Recap – OOP Concepts

Exception Handling – Java Throw Keyword

- **Throw** keyword is used within a method body, or any block of code, and is used to **explicitly throw a single exception**.
- We specify the **exception object** which is to be thrown.
- The Exception has some **message** with it that provides the error description. These exceptions may be related to user inputs, server, etc.
- We can throw either checked or unchecked exceptions in Java by throw keyword.
- It is mainly used to throw a **custom** exception.
- The **syntax** of the Java throw keyword is given below.
  - throw new exception_class("error message");
  - throw new IOException("sorry device error");

Recap – OOP Concepts

Exception Handling – Java Throw Keyword

```
class Main {
  public static void divideByZero() {
    throw new ArithmeticException("Trying to divide by 0");
  }
  public static void main(String[] args) {
    divideByZero();
  }
}
```

```
Exception in thread "main" java.lang.ArithmeticException: Trying to divide by 0
    at Main.divideByZero(Main.java:3)
    at Main.main(Main.java:7)
exit status 1
```

Recap – OOP Concepts

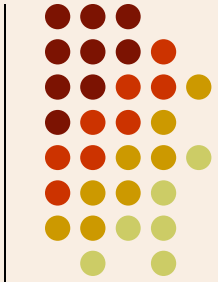
Exception Handling – Java Throw Keyword

```
import java.io.*;
class Main {
  public static void findFile() throws IOException {
    throw new IOException("File not found"); }
  public static void main(String[] args) {
    try {
      findFile();
      System.out.println("Rest of code in try block");
    } catch (IOException e) {
      System.out.println(e.getMessage()); } }
}
```

```
File not found
```

Note that since it is a checked exception, we must specify it in the throws clause. The methods that call this findFile() method need to either handle this exception or specify it using throws keyword themselves.

Swing - JApplet



- **JApplet** - A base class that let's you write code that will run within the context of a browser, like for an interactive web page. Applets are small Internet-based program written in Java
- It is a special type of program that is embedded in the webpage to generate the dynamic content.
- It runs inside the browser and works at client side.
- JFrame and JApplet are top level containers. If you wish to create a desktop application, you will use JFrame and if you plan to host your application in browser you will use JApplet.

Swing - JApplet

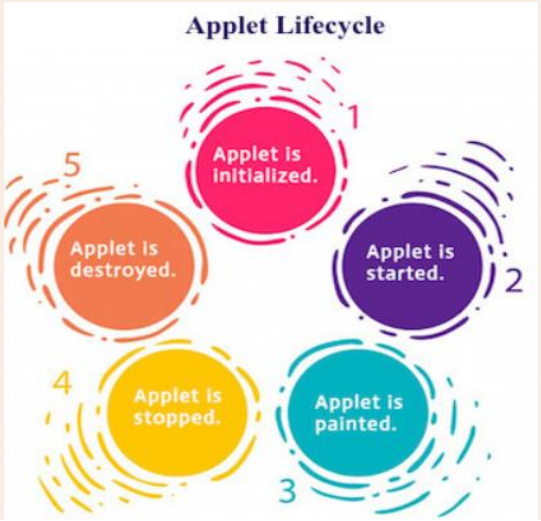
Advantage of Applet

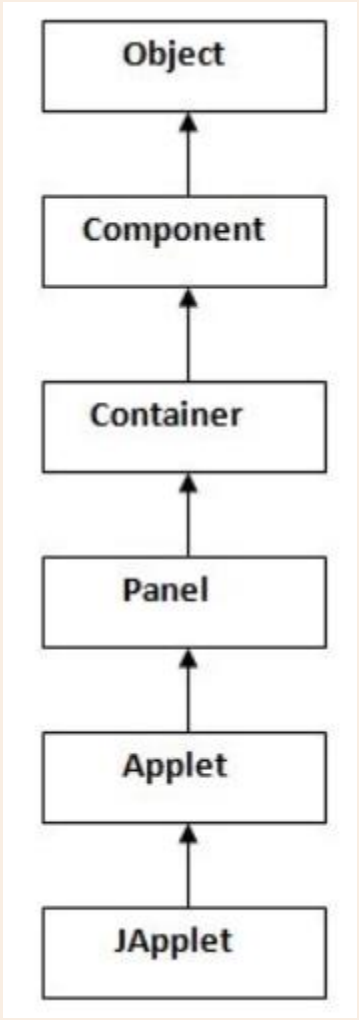
- It works at client side so less response time.
- Secured
- It can be executed by browsers running under many platforms, including Linux, Windows, Mac Os etc.

Drawback of Applet

- Plugin is required at client browser to execute applet.

Swing - JApplet





Method	Description
init()	is used to initialized the Applet. It is invoked only once.
start()	is invoked after the init() method or browser is maximized. It is used to start the Applet.
paint(Graphics g)	is used to paint the Applet. It provides Graphics class object that can be used for drawing oval, rectangle, arc etc.
stop()	is used to stop the Applet. It is invoked when Applet is stop or browser is minimized.
destroy()	is used to destroy the Applet. It is invoked only once.

Swing - JApplet

How to run an Applet?

- By HTML file
  1. Create an applet and compile it.
  2. After that create an html file and place the applet code in html file.
  3. Now click the html file.

JApplet – By HTML

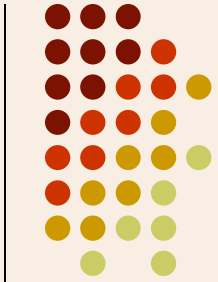
Note: class must be public because its object is created by Java Plugin software that resides on the browser.

```
//First.java
import java.applet.Applet;
import java.awt.Graphics;
public class First extends Applet{

public void paint(Graphics g){
        g.drawString("welcome",150,150);
}
}
```

myapplet.html

```
<html>
<body>
<applet code="First.class" width="300" height="300">
</applet>
</body>
</html>
```

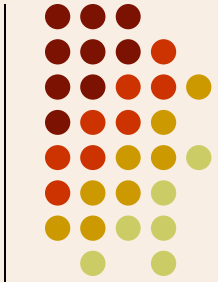
Swing - JApplet



How to run an Applet?

- By appletViewer tool (for testing purpose).
  1. create an applet that contains applet tag in comment and compile it.
  2. After that run it using command prompt by: appletviewer ClassName.java.
  3. Now Html file is not required but it is for testing purpose only.

JApplet – By appletViewer



```
//First.java
import java.applet.Applet;
import java.awt.Graphics;
public class First extends Applet{

public void paint(Graphics g){
g.drawString("welcome to applet",150,150);
}

}
/*
<applet code="First.class" width="300" height="
300">
</applet>
*/
```

```
c:\>javac First.java
c:\>appletviewer First.java
```