MODULE 3

CHAPTER 1

PACKAGES INTERFACES & EXCEPTION HANDLING

PACKAGES

A package in Java is used to group related classes and interfaces

Think of it as a folder in a file directory.

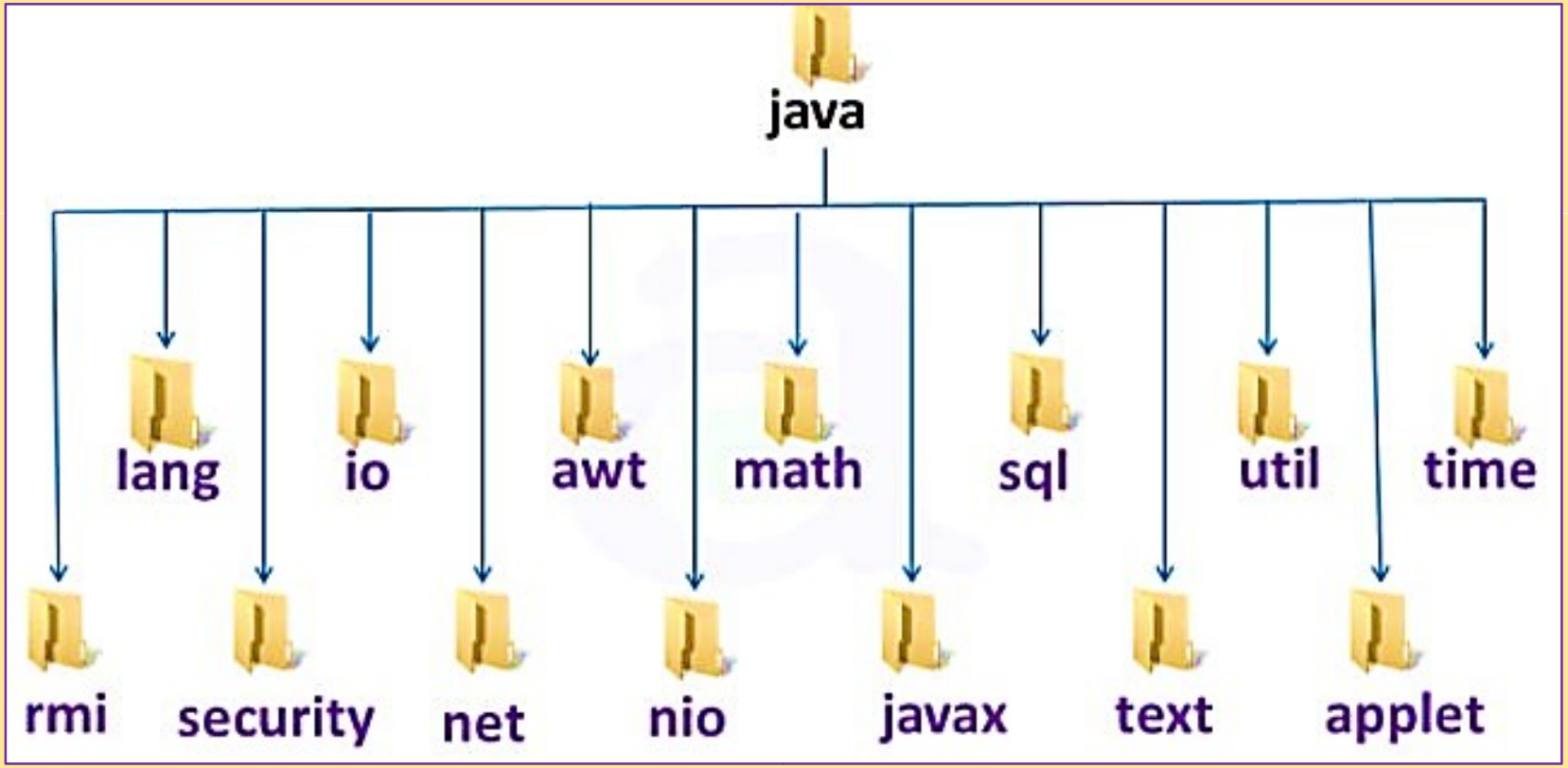
We use packages to avoid name conflicts, and to write a bette maintainable code

Packages in Java is a mechanism to encapsulate a group of classe interfaces and sub packages which is used to providing acces protection

Package in Java can be categorized in two form,

- **built-in package**
- **user-defined package**

- **Built-in Package:-**Existing Java package. for example, java.io.* java.lang , java.util etc.
- **User-defined-package:-** Java package created by user t categorized classes and interface



Advantage of Java Package

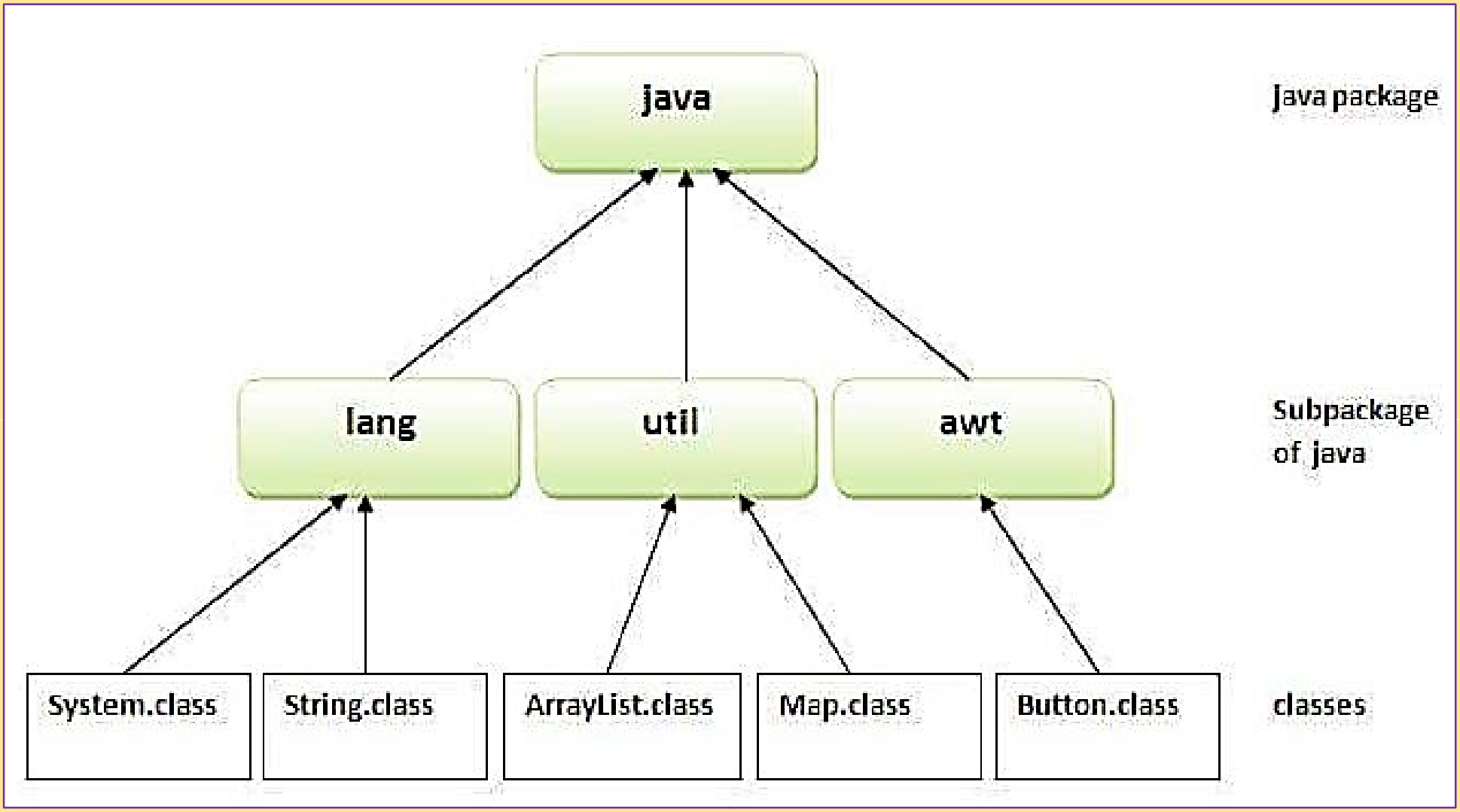
) Java package is used to categorize the classes and interfaces s
hat they can be easily maintained.

) Java package provides access protection.

) In real life situation there may arise scenarios where we need t
efine files of the same name. This may lead to name-spac
ollisions. Java package removes naming collision.

) Reusability: Reusability of code is one of the most importan
equirements in the software industry. Reusability saves time, effo
nd also ensures consistency. A class once developed can be reuse
y any number of programs wishing to incorporate the class in tha
articular program.

) Easy to locate the files.


java
Java package
lang
util
awt
Subpackage
of java
System.class
String.class
ArrayList.class
Map.class
Button.class
classes

To use a class or a package from the library, we need to use th
import keyword:

yntax:

```
import package.name.Class;   // Import a single class
import package.name.*;   // Import the whole package
```

The **package** keyword is used
o create a package in java.

```
//save as Simple.java
package mypack;
public class Simple{
 public static void main(String args[]){
   System.out.println("Welcome to package");
  }
}
```

Access Packages from another package

here are three ways to access the package from outside th ackage.

import package.*;

import package.classname;

fully qualified name

. **Using packagename.***

- If we use **packagename.*** then all the classes and interfaces o this package will be accessible but not subpackages.
- The **"import"** keyword is used to make the classes and interfac of another package accessible to the current package.

xample of package that import the packagename.*

```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.*;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

```
Output:Hello
```

. Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

Example

```
//save by A.java

package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.A;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

```
Output:Hello
```

. Using fully qualified name

If we use fully qualified name then only declared class of th package will be accessible.

Now there is no need to **import**. But you need to use ful qualified name every time when you are accessing the class o interface.

It is generally used when two packages have same class name

e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

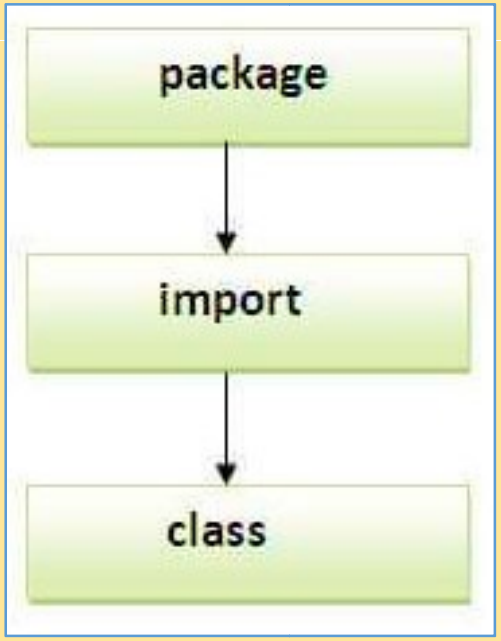
```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
class B{
  public static void main(String args[]){
   pack.A obj = new pack.A();//using fully qualified name
   obj.msg();
  }
}
```

```
Output:Hello
```

- If we import a package, subpackages will not be imported.
- If we import a package, all the classes and interface of tha package will be imported excluding the classes and interfaces o the subpackages.
- Hence, you need to import the subpackage as we

Note: Sequence of the program must be package then impo then class.



INTERFACE

An interface in Java is a blueprint of a class. It has static constant and abstract methods.

The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

Like abstract classes, interfaces cannot be used to create objects

Interface methods do not have a body - the body is provided b
the "**implement**" class

On implementation of an interface, you must override all of it
methods

Interface methods are by default abstract and public

Interface attributes are by default public, static and final

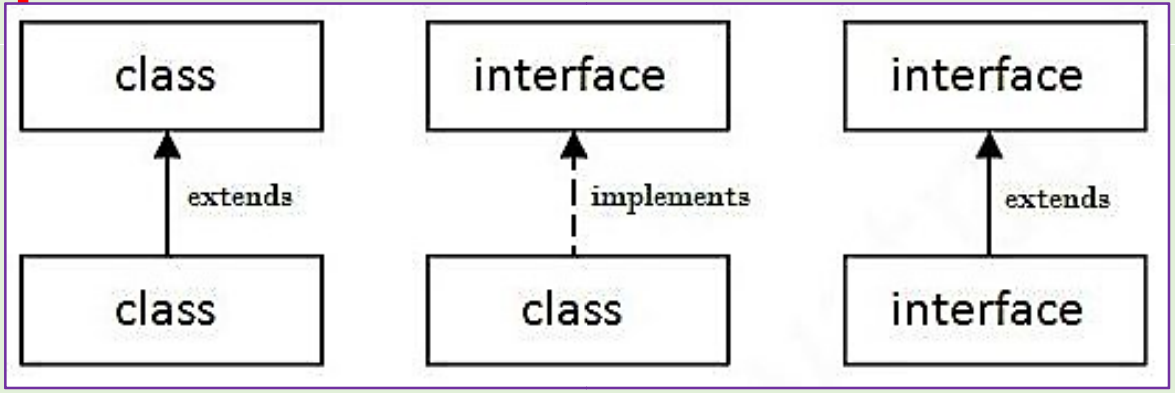
An interface cannot contain a constructor (as it cannot be used t
create objects)

Declare an interface

- An interface is declared by using the **interface** keyword.
- It provides total abstraction; means all the methods in a interface are declared with the empty body, and all the fields ar public, static and final by default.
- A class that implements an interface must implement all th methods declared in the interface.

```
// interface
interface Animal {
  public void animalSound(); // interface method (does not have a body)
  public void run(); // interface method (does not have a body)
}
```

- To access the interface methods, the interface must b "implemented“ by another class with the **implements** keywor (instead of extends).
- The body of the interface method is provided by the "implement class
- **The relationship between classes and interfaces**



- As shown in the figure given above, a class extends another clas an interface extends another interface, but a class implements a interface.

```
/ Interface
nterface Animal {
 public void animalSound(); // interface method (does not have a body)
 public void sleep(); // interface method (does not have a body)


/ Pig "implements" the Animal interface
lass Pig implements Animal {
 public void animalSound() {
   // The body of animalSound() is provided here
   System.out.println("The pig says: wee wee");
 }
 public void sleep() {
   // The body of sleep() is provided here
   System.out.println("Zzz");
 }


lass MyMainClass {
 public static void main(String[] args) {
   Pig myPig = new Pig();  // Create a Pig object
   myPig.animalSound();
   myPig.sleep();
 }
```

Output

```
The pig says: wee wee
Zzz
```

Why And When To Use Interfaces

To achieve security - hide certain details and only show th
portant details of an object (interface).

Java does not support "multiple inheritance". However, it can b
hieved with interfaces, because the class can implement multipl
terfaces.

Note: To implement multiple interfaces, separate them with
comma (see example below).

```
erface FirstInterface {
ublic void myMethod(); // interface method


erface SecondInterface {
ublic void myOtherMethod(); // interface method


ss DemoClass implements FirstInterface, SecondInterface {
ublic void myMethod() {
 System.out.println("Some text..");

ublic void myOtherMethod() {
 System.out.println("Some other text...");



ss MyMainClass {
ublic static void main(String[] args) {
 DemoClass myObj = new DemoClass();
 myObj.myMethod();
 myObj.myOtherMethod();
```

Output

```
Some text...
Some other text...
```

EXCEPTION HANDLING

Exception is an **abnormal condition**.

In Java, an exception is an event that disrupts the normal flow o the program. It is an object which is thrown at runtime.

Exception Handling is a mechanism to handle runtime errors suc as ClassNotFoundException, IOException, SQLExceptio RemoteException, etc.

The core advantage of exception handling is to maintain th normal flow of the application.

An exception normally disrupts the normal flow of the applicatio that is why we use exception handling.

et's take a scenario:

Suppose there are 10 statements in your program and ther occurs an exception at statement 5, the rest of the code will no be executed i.e. statement 6 to 10 will not be executed.

If we perform exception handling, the rest of the statement will b executed. That is why we use exception handling in Java.

```
statement 1;
statement 2;
statement 3;
statement 4;
statement 5;//exception occurs
statement 6;
statement 7;
statement 8;
statement 9;
statement 10;
```

Types of Java Exceptions

There are mainly two types of exceptions: checked an unchecked.

Here, an error is considered as the unchecked exception.

According to Oracle, there are three types of exceptions:

Checked Exception

Unchecked Exception

Error

Checked Exception

The classes which directly inherit Throwable class excep
RuntimeException and Error are known as checked exceptions

e.g. IOException, SQLException etc.

Checked exceptions are checked at compile-time.

Unchecked Exception

The classes which inherit RuntimeException are known a
unchecked exceptions

e.g. ArithmeticException, NullPointerException,

Unchecked exceptions are not checked at compile-time, but they
are checked at runtime

Error

Error is irrecoverable

e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java Exception Keywords

Keyword	Description
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

Common Scenarios of Java Exceptions

A scenario where ArithmeticException occurs

we divide any number by zero, there occurs an ArithmeticExcepti

```
nt a=50/0;   //ArithmeticException
```

scenario where NullPointerException occurs

we have a null value in any variable, performing any operation or
e variable throws a NullPointerException.

```
String s=null;
System.out.println(s.length());   //NullPointerException
```

A scenario where NumberFormatException occurs

The wrong formatting of any value may occ
NumberFormatException. Suppose I have a string variable that h
characters, converting this variable into digit will occ
NumberFormatException.

```
String s="abc";
int i=Integer.parseInt(s);//NumberFormatException
```

A scenario where ArrayIndexOutOfBoundsException occurs

f you are inserting any value in the wrong index, it would result
ArrayIndexOutOfBoundsException as shown below:

```
int a[]=new int[5];
a[10]=50;  //ArrayIndexOutOfBoundsException
```

TRY & CATCH

- The try statement allows you to define a block of code to b tested for errors while it is being executed.
- The catch statement allows you to define a block of code to b executed, if an error occurs in the try block.
- The try and catch keywords come in pairs

Syntax

```
try {
  //  Block of code to try
}
catch(Exception e) {
  //  Block of code to handle errors
}
```

onsider the following example

```
public class MyClass {
  public static void main(String[ ] args) {
    int[] myNumbers = {1, 2, 3};
    System.out.println(myNumbers[10]); // error!
  }
}
```

his will generate an error, because myNumbers[10] does not exist

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
        at MyClass.main(MyClass.java:4)
```

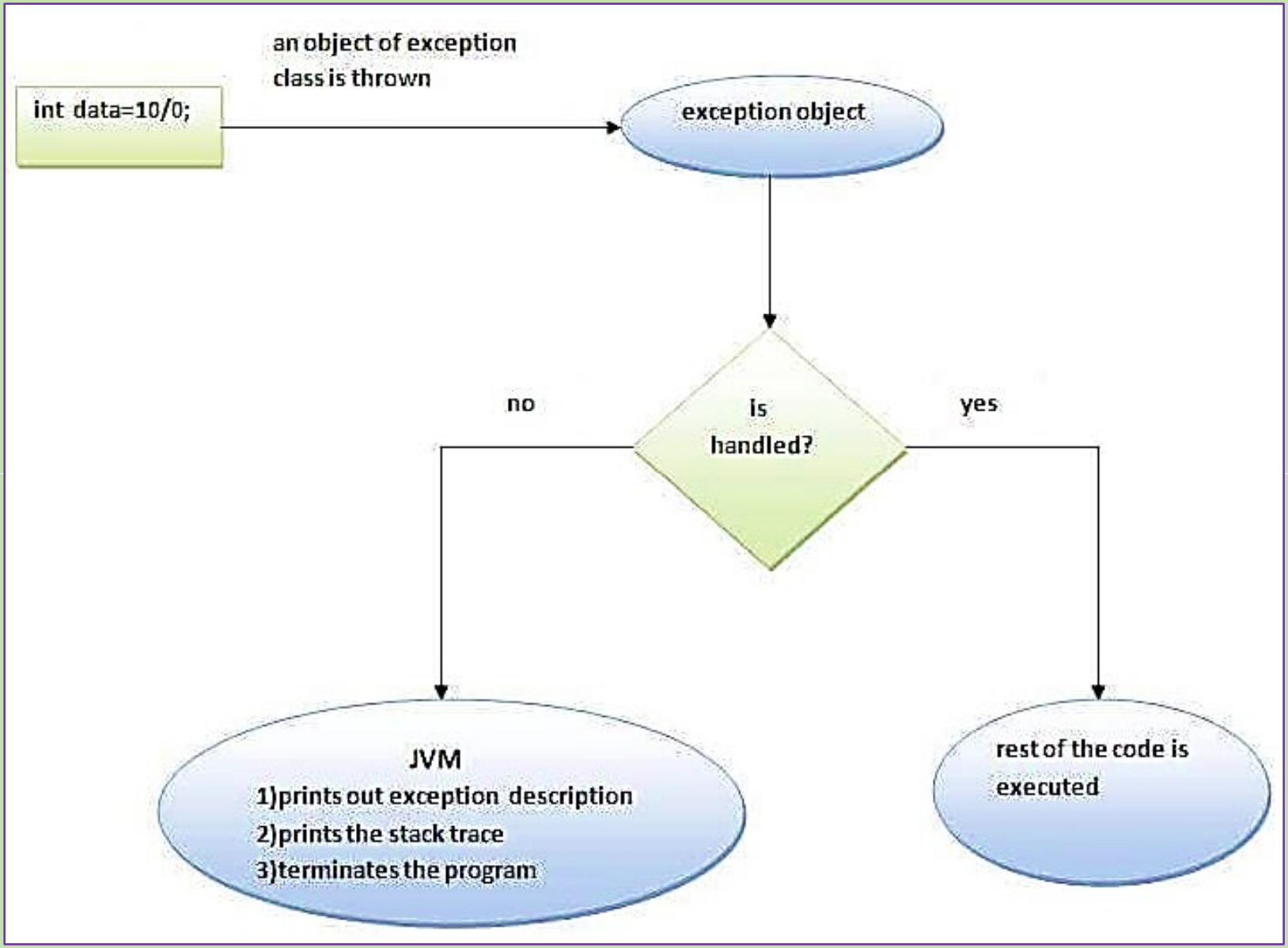
If an error occurs, we can use try...catch to catch the error an execute some code to handle it

```
public class MyClass {
  public static void main(String[ ] args) {
    try {
      int[] myNumbers = {1, 2, 3};
      System.out.println(myNumbers[10]);
    } catch (Exception e) {
      System.out.println("Something went wrong.");
    }
  }
}
```

Output

```
Something went wrong.
```

Internal working of java try-catch block



- The JVM firstly checks whether the exception is handled or not. exception is not handled, JVM provides a default exceptio handler that performs the following tasks:

Prints out exception description.

Prints the stack trace (Hierarchy of methods where the exceptio occurred).

Causes the program to terminate.

- But if exception is handled by the application programme normal flow of the application is maintained i.e. rest of the code executed.

Multi-catch block

A try block can be followed by one or more catch blocks. Eac catch block must contain a different exception handler. So, if yo have to perform different tasks at the occurrence of differen exceptions, use java multi-catch block.

At a time only one exception occurs and at a time only one catc block is executed.

All catch blocks must be ordered from most specific to mos general, i.e. catch for ArithmeticException must come before catc for Exception.

```
public class MultipleCatchBlock1 {

   public static void main(String[] args) {


        try{
            int a[]=new int[5];
            a[5]=30/0;
           }
           catch(ArithmeticException e)
             {
              System.out.println("Arithmetic Exception occurs");
             }
           catch(ArrayIndexOutOfBoundsException e)
             {
              System.out.println("ArrayIndexOutOfBounds Exception occurs");
             }
           catch(Exception e)
             {
              System.out.println("Parent Exception occurs");
             }
           System.out.println("rest of the code");
   }
}
```

Output

```
Arithmetic Exception occurs
rest of the code
```

Nested try block

The try block within a try block is known as 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.

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

```
class Excep6{
 public static void main(String args[]){
  try{
    try{
     System.out.println("going to divide");
     int b =39/0;
    }catch(ArithmeticException e){System.out.println(e);}

    try{
    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..");
 }
}
```

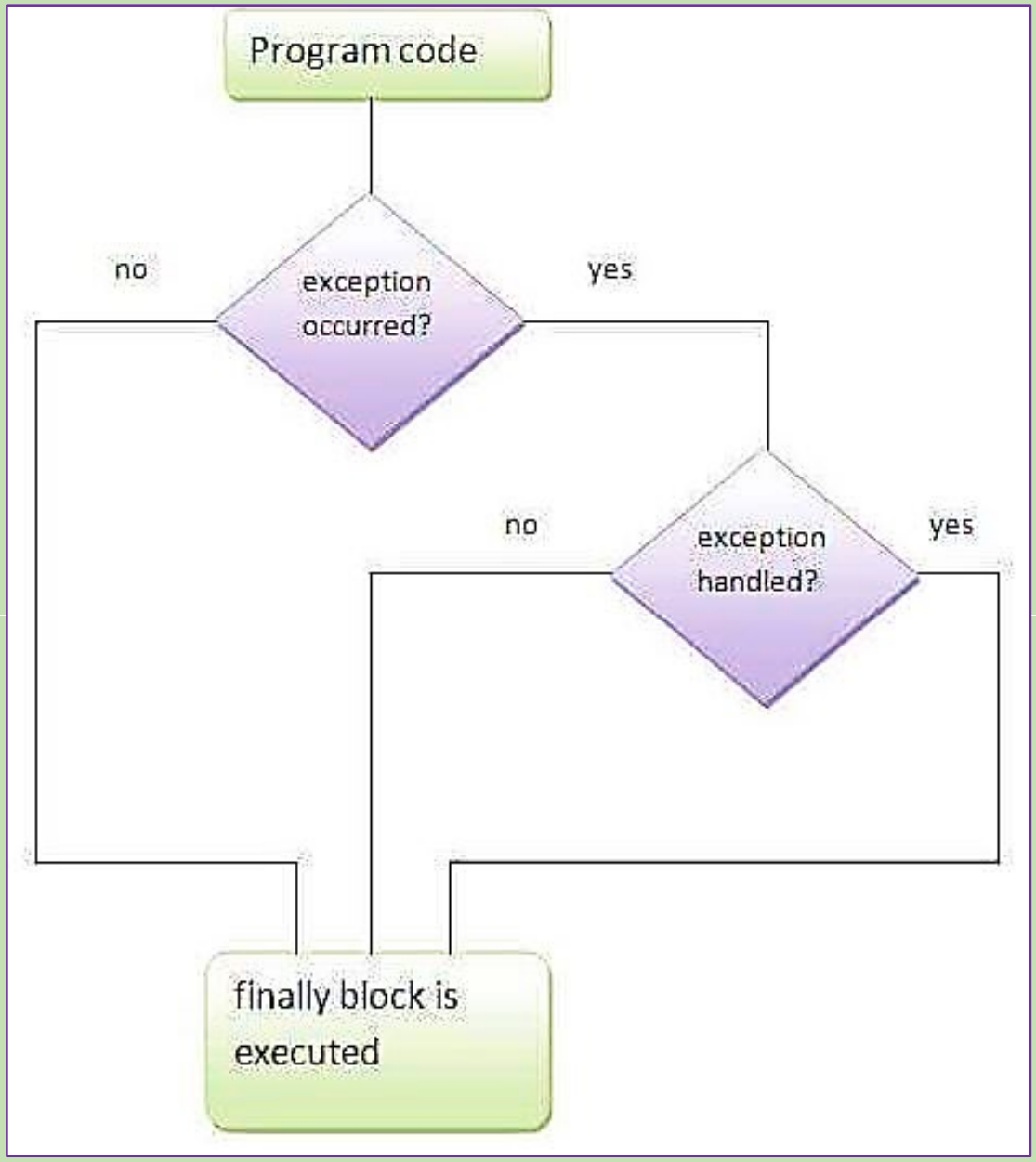
finally block

Java finally block is a block that is used to execute important cod such as closing connection, stream etc.

Java finally block is always executed whether exception is handle or not.

Java finally block follows try or catch block.

ote: If you don't handle exception, before terminating th ogram, JVM executes finally block(if any).


Program code
exception occurred?
no
yes
exception handled?
no
yes
finally block is executed

Usage of Java finally

Case 1 - Let's see the java finally example where **exception doesn't occur.**

```
lass TestFinallyBlock{
public static void main(String args[]){
try{
 int data=25/5;
 System.out.println(data);
}
catch(NullPointerException e){System.out.println(e);}
finally{System.out.println("finally block is always executed");}
System.out.println("rest of the code...");
}
```

```
Output:5
       finally block is always execute
       rest of the code...
```

nse 2 - Let's see the java finally example where exception occurs nd not handled.

```
class TestFinallyBlock1{
  public static void main(String args[]){
  try{
   int data=25/0;
   System.out.println(data);
  }
  catch(NullPointerException e){System.out.println(e);}
  finally{System.out.println("finally block is always executed");}
  System.out.println("rest of the code...");
  }
}
```

```
Output:finally block is always executed
       Exception in thread main java.lang.ArithmeticException:/ by zero
```

nse 3 - Let's see the java finally example where exception occurs nd handled

```
public class TestFinallyBlock2{
 public static void main(String args[]){
 try{
  int data=25/0;
  System.out.println(data);
 }
 catch(ArithmeticException e){System.out.println(e);}
 finally{System.out.println("finally block is always executed");}
 System.out.println("rest of the code...");
 }
}
```

```
Output:Exception in thread main java.lang.ArithmeticException:/ by zero
       finally block is always executed
       rest of the code...
```

throw keyword

The Java throw keyword is used to explicitly throw an exception.

We can throw either checked or uncheked exception in java throw keyword

```
public class TestThrow1{
  static void validate(int age){
    if(age<18)
      throw new ArithmeticException("not valid")
    else
      System.out.println("welcome to vote");
  }
  public static void main(String args[]){
      validate(13);
      System.out.println("rest of the code...");
  }
}
```

Output

```
eption in thread main java.lang.ArithmeticException:not valid
```

throws keyword

he Java throws keyword is used to declare an exception.

t gives an information to the programmer that there may occur
xception so it is better for the programmer to provide the excepti
andling code so that normal flow can be maintained

xception Handling is mainly used to handle the checked exceptio

f there occurs any unchecked exception such
ullPointerException, it is programmers fault that he is n
erforming check up before the code being used.

ntax of java throws

```
return_type method_name() throws exception_class_name{
//method code
}
```

hich exception should be declared

checked exception only, because:

nchecked Exception: under your control so correct your code.

rror: beyond your control e.g. you are unable to do anything here occurs VirtualMachineError or StackOverflowError.

```java
import java.io.*;
class ThrowExample {
  void myMethod(int num)throws IOException, ClassNotFoundException{
     if(num==1)
        throw new IOException("IOException Occurred");
     else
        throw new ClassNotFoundException("ClassNotFoundException");
  }
}

public class Example1{
  public static void main(String args[]){
   try{
     ThrowExample obj=new ThrowExample();
     obj.myMethod(1);
   }catch(Exception ex){
     System.out.println(ex);
    }
  }
}
```

Output

Output:

```
java.io.IOException: IOException Occurr
```