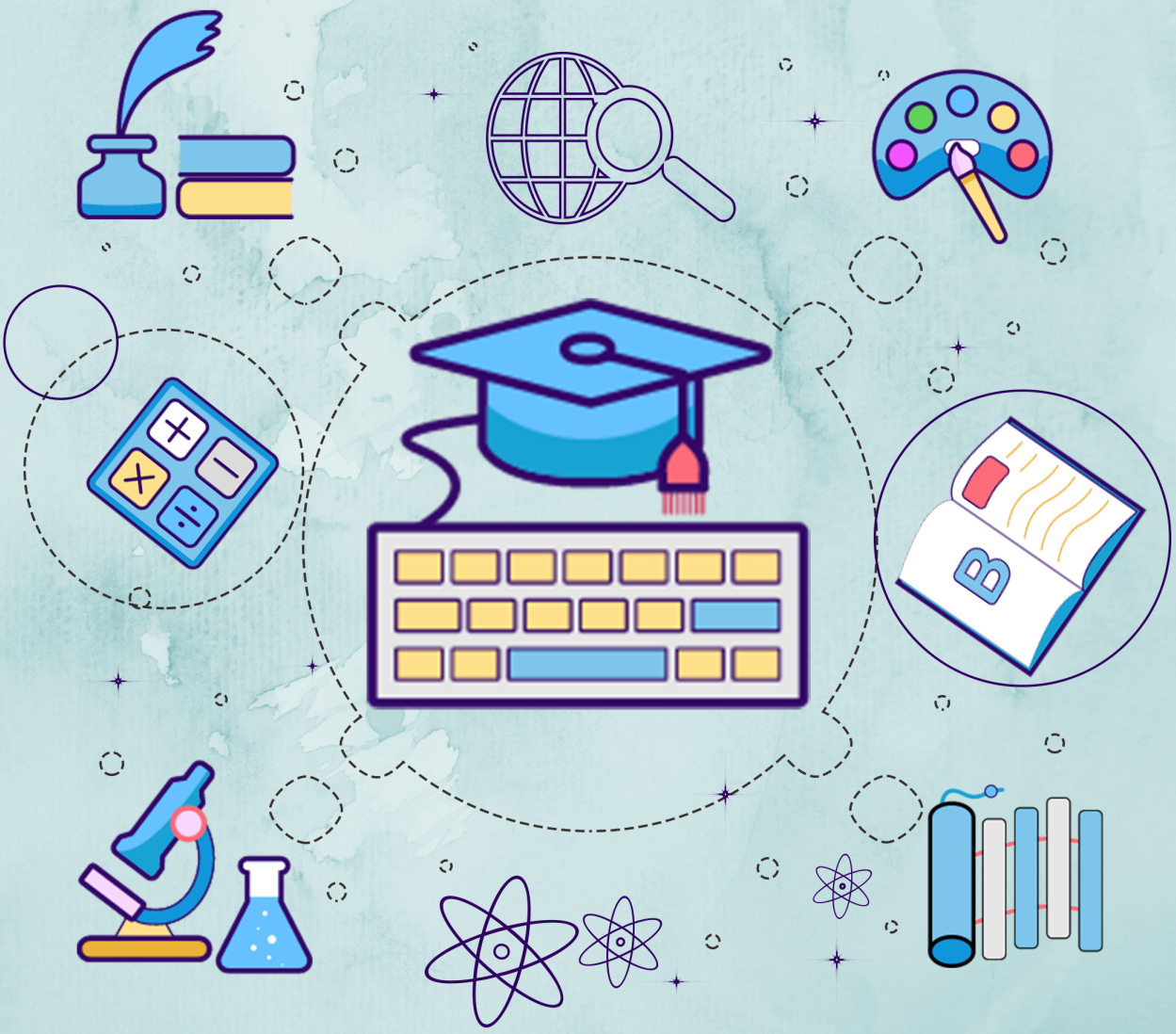
APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

KeralaNotes



SYLLABUS | STUDY MATERIALS | TEXTBOOK

PDF | SOLVED QUESTION PAPERS

www.keralanotes.com

KeralaNotes

KTU STUDY MATERIALS

OBJECT ORIENTED PROGRAMMING USING JAVA

CST 205

Module 3

Related Link :

- KTU S3 STUDY MATERIALS
- KTU S3 NOTES
- KTU S3 SYLLABUS
- KTU S3 TEXTBOOK PDF
- KTU S3 PREVIOUS YEAR SOLVED QUESTION PAPER

MODULE 3

CHAPTER 1

PACKAGES INTERFACES & EXCEPTION HANDLING

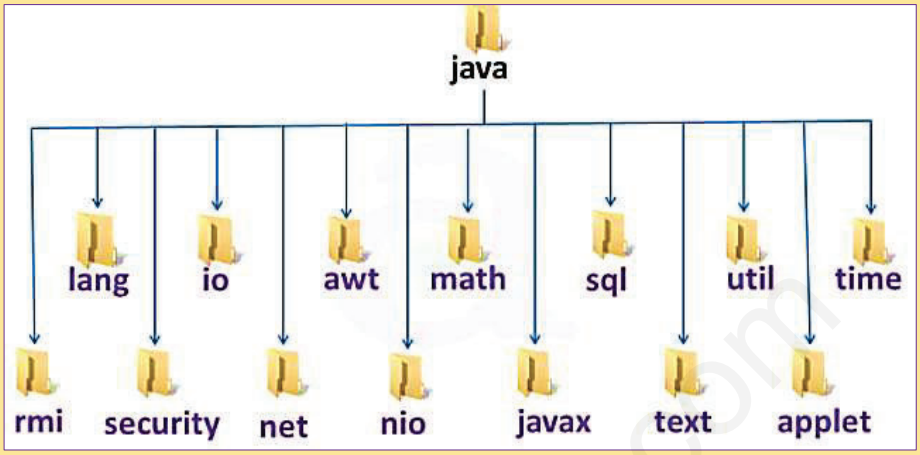
PACKAGES

- 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 better maintainable code
- Packages in Java is a mechanism to encapsulate a group of classes, interfaces and sub packages which is used to providing access 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 to categorized classes and interface



❖Advantage of Java Package

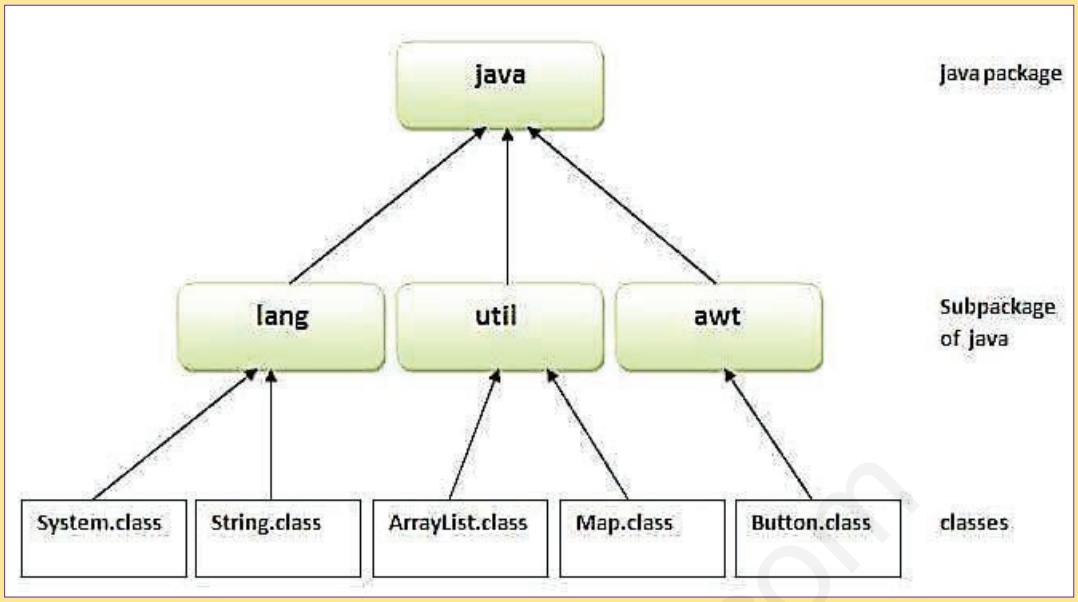
1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) In real life situation there may arise scenarios where we need to define files of the same name. This may lead to name-space collisions. Java package removes naming collision.

4) Reusability: Reusability of code is one of the most important requirements in the software industry. Reusability saves time, effort and also ensures consistency. A class once developed can be reused by any number of programs wishing to incorporate the class in that particular program.

5) Easy to locate the files.



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

Syntax:

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

- The **package** keyword is used to 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

There are three ways to access the package from outside the package.

import package.*;

import package.classname;

fully qualified name

1. Using packagename.*

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

Example 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
```

2. 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

3. Using fully qualified name

- If we use fully qualified name then only declared class of this package will be accessible.
- Now there is no need to **import**. But you need to use fully qualified name every time when you are accessing the class or 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

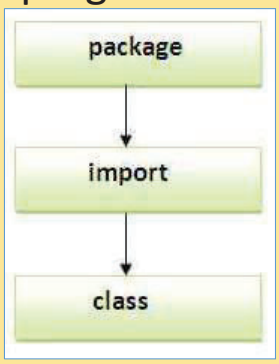
```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
```

```
//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 that package will be imported excluding the classes and interfaces of the subpackages.
- Hence, you need to import the subpackage as well

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



INTERFACE

- An interface in Java is a blueprint of a class. It has static constants 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 by the "**implement**" class
- On implementation of an interface, you must override all of its 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 to create objects)

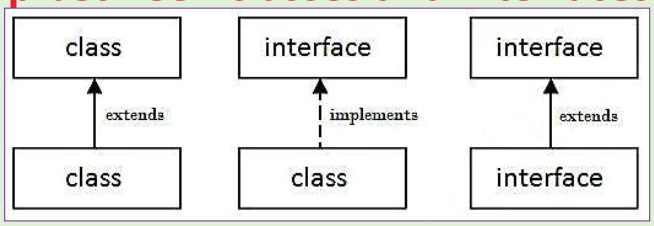
❖Declare an interface

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

```java
// Interface
interface 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
class 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");
  }
}

class 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

1) **To achieve security** - hide certain details and only show the important details of an object (interface).

2) Java does not support "multiple inheritance". However, it can be achieved with interfaces, because the class can implement multiple interfaces.

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

```java
interface FirstInterface {
  public void myMethod(); // interface method
}

interface SecondInterface {
  public void myOtherMethod(); // interface method
}

class DemoClass implements FirstInterface, SecondInterface {
  public void myMethod() {
    System.out.println("Some text..");
  }
  public void myOtherMethod() {
    System.out.println("Some other text...");
  }
}

class MyMainClass {
  public 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 of the program. It is an object which is thrown at runtime.
- Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.
- The core advantage of exception handling is to maintain the normal flow of the application.
- An exception normally disrupts the normal flow of the application that is why we use exception handling.

Let's take a scenario:

- Suppose there are 10 statements in your program and there occurs an exception at statement 5, the rest of the code will not be executed i.e. statement 6 to 10 will not be executed.
- If we perform exception handling, the rest of the statement will be 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 and 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 except 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 as 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

• If we divide any number by zero, there occurs an ArithmeticException.

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

➢A scenario where NullPointerException occurs

• If we have a null value in any variable, performing any operation on the 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 occur NumberFormatException. Suppose I have a string variable that has characters, converting this variable into digit will occur NumberFormatException.

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

➢ A scenario where ArrayIndexOutOfBoundsException occurs

• If you are inserting any value in the wrong index, it would result in 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 be tested for errors while it is being executed.
- The catch statement allows you to define a block of code to be 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
}
```


Consider the following example

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

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

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

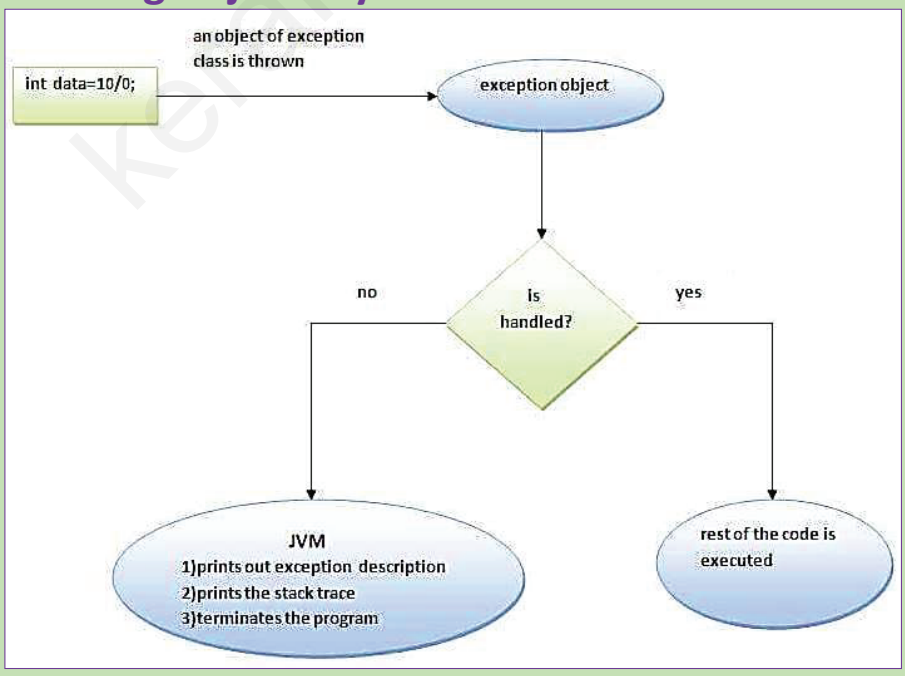
- If an error occurs, we can use try...catch to catch the error and execute some code to handle it

```
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. If exception is not handled, JVM provides a default exception handler that performs the following tasks:
  - Prints out exception description.
  - Prints the stack trace (Hierarchy of methods where the exception occurred).
  - Causes the program to terminate.
- But if exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.

Multi-catch block

- A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.
- At a time only one exception occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch 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.

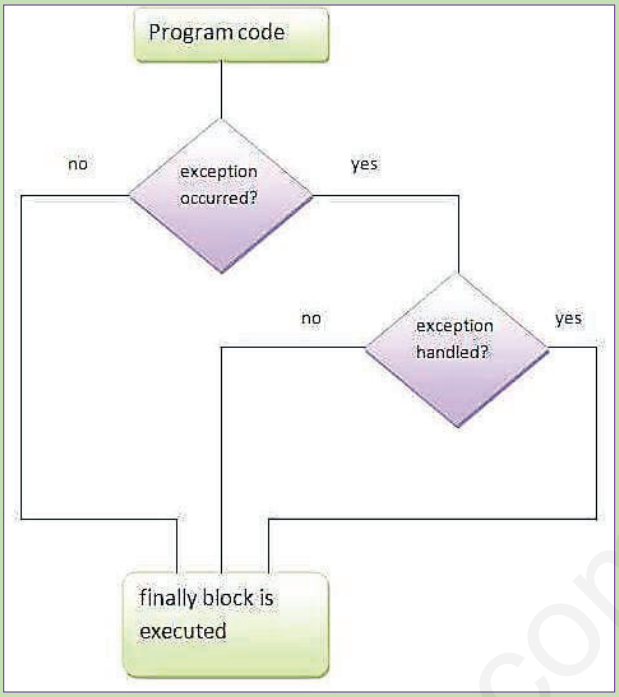
```
....
try
{
   statement 1;
   statement 2;
   try
   {
      statement 1;
      statement 2;
   }
   catch(Exception e)
   {
   }
}
catch(Exception e)
{
}
....
```

```
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 code such as closing connection, stream etc.
- Java finally block is always executed whether exception is handled or not.
- Java finally block follows try or catch block.

Note: If you don't handle exception, before terminating the program, JVM executes finally block(if any).



❖Usage of Java finally

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

```
class 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 executed
       rest of the code...
```

Case 2 - Let's see the java finally example where **exception occurs and 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
```


Case 3 - Let's see the java finally example where **exception occurs and 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 by 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

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


throws keyword

- The Java throws keyword is used to declare an exception.
- It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained
- 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 performing check up before the code being used.

Syntax of java throws

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

Which exception should be declared

- checked exception only, because:
- unchecked Exception: under your control so correct your code.
- error: beyond your control e.g. you are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

Eg:

```
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 Occurred
```

MODULE 3

CHAPTER 2
JAVA INPUT OUTPUT (I/O) & FILES

STREAM

- Java I/O (Input and Output) is used to process the input and produce the output.
- Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.
- We can perform file handling in Java by Java I/O API.

STREAM

- A stream is a sequence of data. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.

- In Java, 3 streams are created for us automatically. All these streams are attached with the console.

1) **System.out** : standard output stream

2) **System.in** : standard input stream

3) **System.err** : standard error stream

- The code to print output and an error message to the console.

```
System.out.println("simple message");
System.err.println("error message");
```

- The code to get input from console.

```
int i=System.in.read();  //returns ASCII code of 1st character
```


❖OutputStream vs InputStream

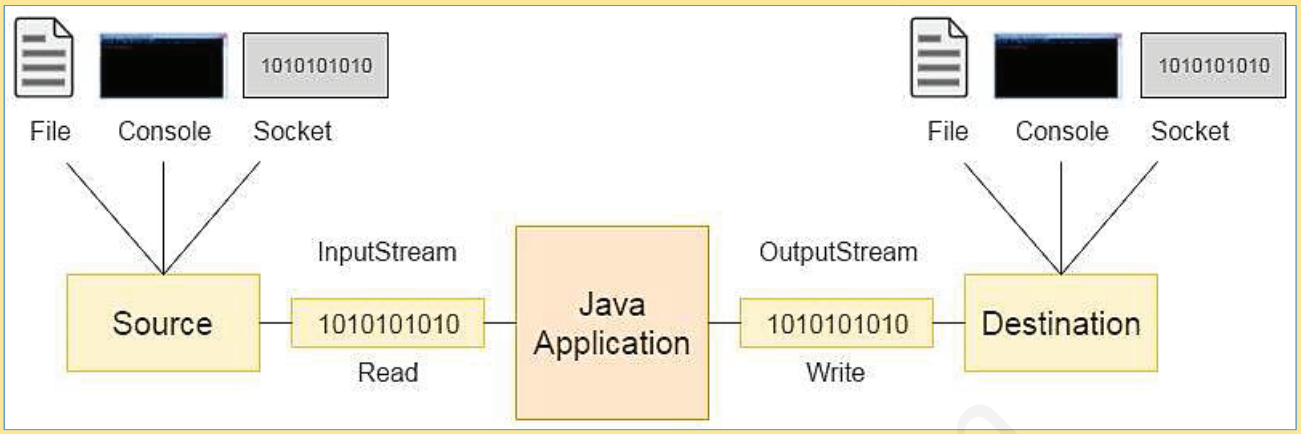
- **OutputStream**

Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.

- **InputStream**

Java application uses an input stream to read data from a source; it may be a file, an array, peripheral device or socket.

❖The working of Java OutputStream and InputStream



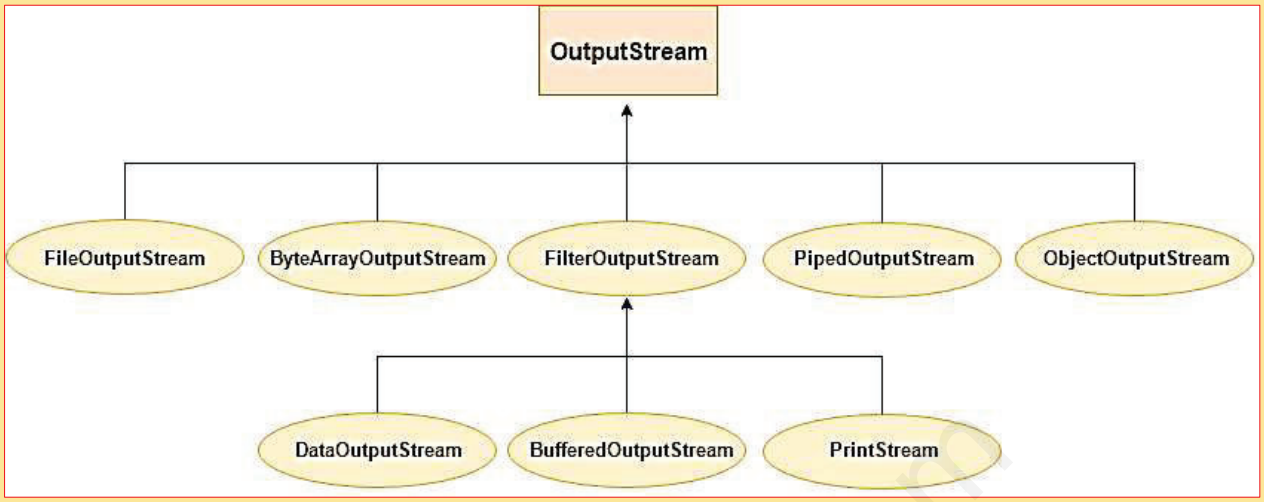
❖OutputStream class

- OutputStream class is an abstract class.
- It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

Useful methods of OutputStream

Method	Description
1) public void write(int)throws IOException	is used to write a byte to the current output stream.
2) public void write(byte[])throws IOException	is used to write an array of byte to the current output stream.
3) public void flush()throws IOException	flushes the current output stream.
4) public void close()throws IOException	is used to close the current output stream.

❖OutputStream Hierarchy



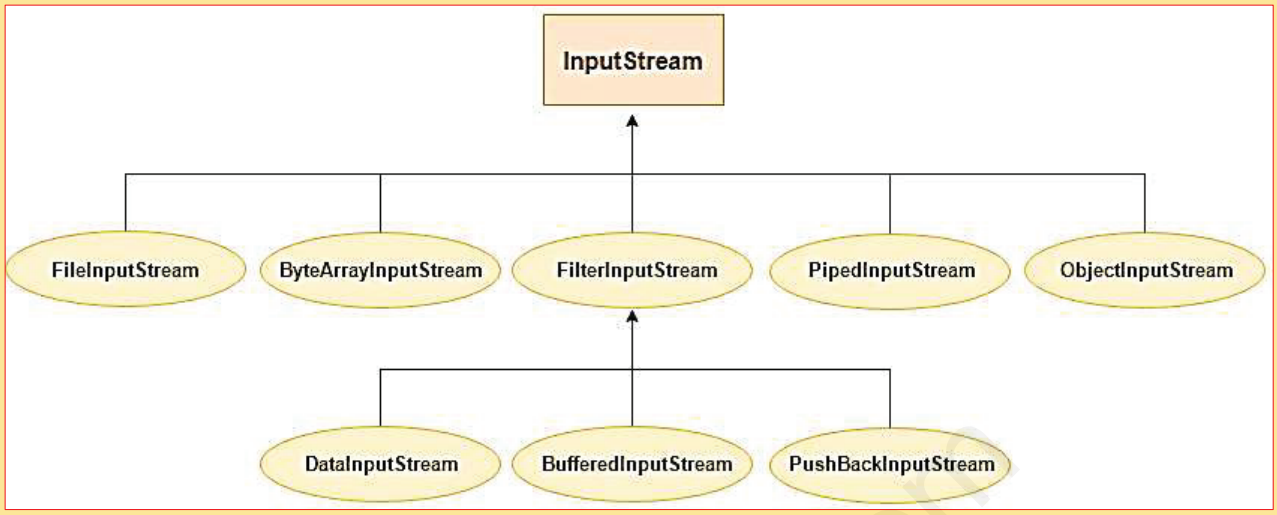
❖InputStream class

- InputStream class is an abstract class.
- It is the superclass of all classes representing an input stream of bytes.

Useful methods of InputStream

Method	Description
1) public abstract int read()throws IOException	reads the next byte of data from the input stream. It returns -1 at the end of the file.
2) public int available()throws IOException	returns an estimate of the number of bytes that can be read from the current input stream.
3) public void close()throws IOException	is used to close the current input stream.

❖InputStream Hierarchy



READING CONSOLE INPUT

➢In Java, there are three different ways for reading input from the user in the command line environment(console).

1.Using Buffered Reader Class

- This is the Java classical method to take input, Introduced in JDK1.0.
- This method is used by wrapping the System.in (standard input stream) in an InputStreamReader which is wrapped in a BufferedReader, we can read input from the user in the command line.
- **Advantage** - The input is buffered for efficient reading
- **Drawback** - The wrapping code is hard to remember.

```
// Java program to demonstrate BufferedReader
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class Test
{
        public static void main(String[] args) throws IOException
        {
                //Enter data using BufferReader
                BufferedReader reader = new BufferedReader(new
                                                InputStreamReader(System.in));
                // Reading data using readLine
                String name = reader.readLine();
                // Printing the read line
                System.out.println(name);
        }
}
```


2. Using Scanner Class

- This is probably the most preferred method to take input.
- The main purpose of the Scanner class is to parse primitive types and strings using regular expressions, however it is also can be used to read input from the user in the command line.

Advantages:

- Convenient methods for parsing primitives (nextInt(), nextFloat(), …) from the tokenized input.
- Regular expressions can be used to find tokens.

Drawback:

- The reading methods are not synchronized

```
// Java program to demonstrate working of Scanner in Java
import java.util.Scanner;
class GetInputFromUser
{
        public static void main(String args[])
        {
                // Using Scanner for Getting Input from User
                Scanner in = new Scanner(System.in);
                    String s = in.nextLine();
                System.out.println("You entered string "+s);
                int a = in.nextInt();
                System.out.println("You entered integer "+a);
                float b = in.nextFloat();
                System.out.println("You entered float "+b);
        }
}
```

Input:

HelloStudents
12
3.4

Output:

You entered string
HelloStudents
You entered integer 12
You entered float 3.4

3. Using Console Class

- It has been becoming a preferred way for reading user's input from the command line.
- In addition, it can be used for reading password-like input without echoing the characters entered by the user; the format string syntax can also be used (like System.out.printf()).

Advantages:

- Reading password without echoing the entered characters.
- Reading methods are synchronized.
- Format string syntax can be used.

Drawback: Does not work in non-interactive environment (such as in an IDE).

```
// Java program to demonstrate working of System.console()
// Note that this program does not work on IDEs as
// System.console() may require console
public class Sample
{
        public static void main(String[] args)
        {
                // Using Console to input data from user
                String name = System.console().readLine();

                System.out.println(name);
        }
}
```


WRITING CONSOLE OUTPUT

- Console output is most easily accomplished with print() and println() methods.
- These methods are defined by the class PrintStream which is the type of object referenced by System.in.
- Because the PrintStream is an output stream derived from the OutputStream, it also implements the low-level method write().
- Thus, write() can be used to write to the console. The simplest form of write() defined by the PrintStream is shown below :

void write(int byteval)

- Following is a short example that uses write() to output the character 'X' followed by a newline to the screen:

```
/* Java Program Example - Java Write Console Output
 * This program writes the character X followed by newline
 * This program demonstrates System.out.write()  */

class WriteConsoleOutput
{
    public static void main(String args[])
    {

        int y;

        y = 'X';

        System.out.write(y);
        System.out.write('\n');

    }
}
```


PrintWriter CLASS

- Java PrintWriter class is the implementation of Writer class.
- It is used to print the formatted representation of objects to the text-output stream.

Class declaration

public class PrintWriter extends Writer

Methods of PrintWriter class

Method	Description
void println(boolean x)	It is used to print the boolean value.
void println(char[] x)	It is used to print an array of characters.
void println(int x)	It is used to print an integer.

PrintWriter append(char c)	It is used to append the specified character to the writer.
PrintWriter append(CharSequence ch)	It is used to append the specified character sequence to the writer.
PrintWriter append(CharSequence ch, int start, int end)	It is used to append a subsequence of specified character to the writer.
boolean checkError()	It is used to flushes the stream and check its error state.
protected void setError()	It is used to indicate that an error occurs.
protected void clearError()	It is used to clear the error state of a stream.
PrintWriter format(String format, Object... args)	It is used to write a formatted string to the writer using specified arguments and format string.
void print(Object obj)	It is used to print an object.
void flush()	It is used to flushes the stream.
void close()	It is used to close the stream.

Eg:

```
import java.io.File;
import java.io.PrintWriter;
public class PrintWriterExample {
    public static void main(String[] args) throws Exception {
            //Data to write on Console using PrintWriter
      PrintWriter writer = new PrintWriter(System.out);
      writer.write("Javatpoint provides tutorials of all technology.");
 writer.flush();
      writer.close();
//Data to write in File using PrintWriter
      PrintWriter writer1 =null;
        writer1 = new PrintWriter(new File("D:\\testout.txt"));
        writer1.write("Like Java, Spring, Hibernate, Android, PHP etc.");
                    writer1.flush();
        writer1.close();
    }
}
```

Outpt

```
Javatpoint provides tutorials of all technology.
```

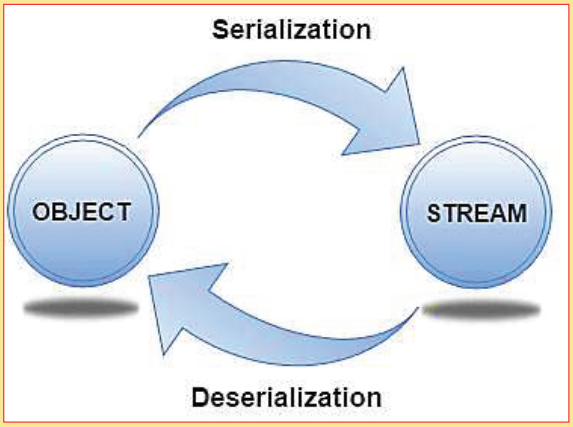
SERIALIZATION

- Serialization in Java is the process of converting the Java code Object into a Byte Stream, to transfer the Object Code from one Java Virtual machine to another and recreate it using the process of Deserialization.
- Most impressive is that the entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.
- For serializing the object, we call the writeObject() method of ObjectOutputStream, and for deserialization we call the readObject() method of ObjectInputStream class.

- We must have to implement the *Serializable* interface for serializing the object.

Advantages of Java Serialization

- It is mainly used to travel object's state on the network (which is known as marshaling).



❖ObjectOutputStream class

- The ObjectOutputStream class is used to write primitive data types, and Java objects to an OutputStream.
- Only objects that support the java.io.Serializable interface can be written to streams.

Constructor

1) public ObjectOutputStream(OutputStream out) throws IOException {}	creates an ObjectOutputStream that writes to the specified OutputStream.

Important Methods

Method	Description
1) public final void writeObject(Object obj) throws IOException {}	writes the specified object to the ObjectOutputStream.
2) public void flush() throws IOException {}	flushes the current output stream.
3) public void close() throws IOException {}	closes the current output stream.

❖ObjectInputStream class

- An ObjectInputStream deserializes objects and primitive data written using an ObjectOutputStream.

Constructor

1) public ObjectInputStream(InputStream in) throws IOException {}	creates an ObjectInputStream that reads from the specified InputStream.

Important Methods

Method	Description
1) public final Object readObject() throws IOException, ClassNotFoundException{}	reads an object from the input stream.
2) public void close() throws IOException {}	closes ObjectInputStream.

Example of Java Serialization

- In this example, we are going to serialize the object of Student class. The writeObject() method of ObjectOutputStream class provides the functionality to serialize the object. We are saving the state of the object in the file named f.txt.

Output

```
success
```

```
import java.io.*;
class Persist{
 public static void main(String args[]){
  try{
  //Creating the object
  Student s1 =new Student(211,"ravi");
  //Creating stream and writing the object
  FileOutputStream fout=new FileOutputStream("f.txt");
  ObjectOutputStream out=new ObjectOutputStream(fout);
  out.writeObject(s1);
  out.flush();
  //closing the stream
  out.close();
  System.out.println("success");
  }catch(Exception e){System.out.println(e);}
 }
}
```


WORKING WITH FILES

- File handling is an important part of any application.
- Java has several methods for creating, reading, updating, and deleting files.
- The File class from the java.io package, allows us to work with files.
- To use the File class, create an object of the class, and specify the filename or directory name:

Example

```
import java.io.File;  // Import the File class

File myObj = new File("filename.txt"); // Specify the filename
```

- The File class has many useful methods for creating and getting information about files. For example:

Method	Type	Description
canRead()	Boolean	Tests whether the file is readable or not
canWrite()	Boolean	Tests whether the file is writable or not
createNewFile()	Boolean	Creates an empty file
delete()	Boolean	Deletes a file
exists()	Boolean	Tests whether the file exists
getName()	String	Returns the name of the file
getAbsolutePath()	String	Returns the absolute pathname of the file
length()	Long	Returns the size of the file in bytes
list()	String[]	Returns an array of the files in the directory
mkdir()	Boolean	Creates a directory

❖Create a File

- To create a file in Java, you can use the createNewFile() method.
- This method returns a boolean value: true if the file was successfully created, and false if the file already exists.
- Note that the method is enclosed in a try...catch block.
- This is necessary because it throws an IOException if an error occurs (if the file cannot be created for some reason):

Example

```
import java.io.File;  // Import the File class
import java.io.IOException;  // Import the IOException class to handle errors

public class CreateFile {
  public static void main(String[] args) {
    try {
      File myObj = new File("filename.txt");
      if (myObj.createNewFile()) {
        System.out.println("File created: " + myObj.getName());
      } else {
        System.out.println("File already exists.");
      }
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

The output will be:

```
File created: filename.txt
```


❖Write To a File

- In the following example, we use the FileWriter class together with its write() method to write some text to the file we created in the example above.
- Note that when we are done writing to the file, we should close it with the close() method:

Example

```
import java.io.FileWriter;   // Import the FileWriter class
import java.io.IOException;  // Import the IOException class to handle errors

public class WriteToFile {
  public static void main(String[] args) {
    try {
      FileWriter myWriter = new FileWriter("filename.txt");
      myWriter.write("Files in Java might be tricky, but it is fun enough!");
      myWriter.close();
      System.out.println("Successfully wrote to the file.");
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

The output will be:

```
Successfully wrote to the file.
```


❖Read Files

```
import java.io.File;  // Import the File class
import java.io.FileNotFoundException;  // Import this class to handle errors
import java.util.Scanner; // Import the Scanner class to read text files

public class ReadFile {
  public static void main(String[] args) {
    try {
      File myObj = new File("filename.txt");
      Scanner myReader = new Scanner(myObj);
      while (myReader.hasNextLine()) {
        String data = myReader.nextLine();
        System.out.println(data);
      }
      myReader.close();
    } catch (FileNotFoundException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

The output will be:

```
Files in Java might be tricky, but it is fun enough!
```

❖Get File Information

```
import java.io.File;  // Import the File class

public class GetFileInfo {
  public static void main(String[] args) {
    File myObj = new File("filename.txt");
    if (myObj.exists()) {
      System.out.println("File name: " + myObj.getName());
      System.out.println("Absolute path: " + myObj.getAbsolutePath());
      System.out.println("Writeable: " + myObj.canWrite());
      System.out.println("Readable " + myObj.canRead());
      System.out.println("File size in bytes " + myObj.length());
    } else {
      System.out.println("The file does not exist.");
    }
  }
}
```

The output will be:

```
File name: filename.txt
Absolute path: C:\Users\MyName\filename.txt
Writeable: true
Readable: true
File size in bytes: 0
```


❖Delete a File

➢To delete a file in Java, use the delete() method:

```
import java.io.File;  // Import the File class

public class DeleteFile {
  public static void main(String[] args) {
    File myObj = new File("filename.txt");
    if (myObj.delete()) {
      System.out.println("Deleted the file: " + myObj.getName());
    } else {
      System.out.println("Failed to delete the file.");
    }
  }
}
```

The output will be:

```
Deleted the file: filename.txt
```

❖Delete a Folder

```
import java.io.File;

public class DeleteFolder {
  public static void main(String[] args) {
    File myObj = new File("C:\\Users\\MyName\\Test");
    if (myObj.delete()) {
      System.out.println("Deleted the folder: " + myObj.getName());
    } else {
      System.out.println("Failed to delete the folder.");
    }
  }
}
```

The output will be:

```
Deleted the folder: Test
```