OOP-JAVA
CST 205

More Features of JAVA

MODULE 3

Author – Milan George Mathew

Syllabus

Packages and Interfaces - Defining Package, CLASSPATH, Access Protection, Importing Packages, Interfaces.

Exception Handling - Checked Exceptions, Unchecked Exceptions, try Block and catch Clause, Multiple catch Clauses, Nested try Statements, throw, throws and finally.

Input/Output - I/O Basics, Reading Console Input, Writing Console Output, PrintWriter Class, Object Streams and Serialization, Working with Files.

Contents

Packages

Packages are container for classes that are used to keep the class names compartmentalized. Making packages and storing Java Classes in those is a technique to avoid namespace collisions. Related classes are grouped in a package and can be imported as such. For ease of understanding think of it as a file directory or folder. Packages also provides the perks of code reusability.

Packages are of two types

1. Built in packages (from JAVA API)
2. User – defined Packages

Built- in Packages

There are a number of built-in packages in Java which includes:

1. java.lang
2. java.io
3. java.awt
4. java.math
5. java.sql
6. java.util

these have many predefined classes that can be imported by:

```
import java.lang.*
```

where the * is used to call all the classes in the package.

User-defined Packages

A developer can define their own classes to keep a clean code structure and implement code reusability by using packages and then arranging the included classes into that package for further import and reuse.Adding a class to a package is easy as creating a file in a folder.

Every class in a package is supposed to have a package defining statement:

```
package <package_name>;
```

Any class without the package statement is stored in the default package which has no name.

Java uses file system directories to store packages. More than one Java file can have the same package statement as this show that the class belongs to that package.

Package hierarchy

When a package is included in another package then it has to be specified with adding period (.) symbol.

For example, take a package pkg_2 inside a package pkg_1,

Any class inside the pkg_2 should have the package statement

```
package pkg_1.pkg_2;
```

A package statement cannot be renamed without renaming the directory. All the classes in a package should reflect the changed name in their package statement.

CLASSPATH

Classpath is defined so as to direct the Java run-time system to the required location where the java classes are stored. JVM uses the current working directory as the starting point, from here the JVM has to go to the location where the class to be executed is located to run and compile it. The `-classpath` option is used with java or javac to specify the path to the required class

Setting a CLASSPATH environment variable is also a used to specify directory path. A program can be executed from the immediately above directory, otherwise the CLASSPATH is to be set for the program.

Access Protection

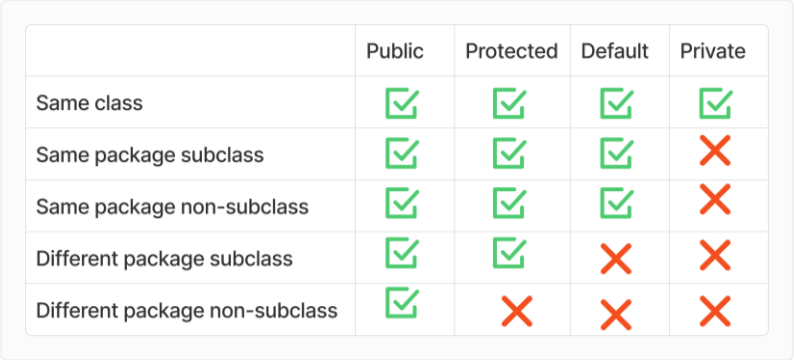
Java classes are the smallest unit of abstraction. There are four categories for visibility for class members in java because of the interplay between classes and packages:

1. Subclasses in same package
2. Non-subclasses in same package
3. Subclasses in different packages

4. Classes which are neither subclass or in same package

There are four access specifiers in java:

1. **default** – visible to subclasses and classes in same package
2. **public** – can be accessed from anywhere
3. **private** – cannot be seen outside the class
4. **protected** – visible to all subclasses and same package classes



	Public	Protected	Default	Private
Same class	✓	✓	✓	✓
Same package subclass	✓	✓	✓	✗
Same package non-subclass	✓	✓	✓	✗
Different package subclass	✓	✓	✗	✗
Different package non-subclass	✓	✗	✗	✗

Table of Access Specifiers

Non nested classes has only two possible access levels **default** and **public**, inner classes can have the **private** or **protected** access levels

Importing Packages

Importing packages saves time of a developer to save time of coding a pre-defined class or a method that is belonging to the imported package. Import statements are defined immediately after package statement before any class definitions. Importing a package can be done by two ways:

1. Using **import** statement –

```
import java.util.*;

class ImpClass extends Date{}
```

2. Extending the class to the package

```
class ImpClass extends java.util.Date{}
```

When a package is imported only the items which are declared **public** within the package will be available to non-subclasses in the importing code.


View in
Github

Interfaces

Interfaces are syntactically similar to class except for the keyword `interface` and the member methods are abstract. There are no implements for methods in an interface but every class that implements the interface is supposed to implement all of those methods.

- Interface does not have instance variables
- One class can implement any number of interfaces
- Any number of classes can implement an interfaces
- Interfaces helps to achieve multilevel inheritance
- Has no concrete methods
- Each class can have its own unique implementation for every method
- One interface, multiple methods – polymorphism
- Dynamic method resolution at run time

Implementing Interfaces

After defining an interface, classes can implement those interfaces by using the `implements` keyword. Multiple classes can implement these and thus we get different classes that will have same method names but different functionalities thus providing the aspect of polymorphism.

A class can extend one class but can implement any number of interfaces thus in the class statement the extends keyword if it exist should come in front and the implements keyword comes after with a list of interfaces if needed , separated by comma.

```
class <classname> [extends superclass] implements interface{}
```

An implemented interface method is supposed to be declares as public.

Interface references

We can declare variables as object references to a interface rather than the class name, that is we can create an object reference like

```
interfacename obj=new class_implementing_the_interface();
```

any class that implements an interface can declare variables with the interface reference

Partial Implementation

If a class includes an interface but does not fully implement the methods required by that interface, then that class must be declared as **abstract**.

Nested Interfaces

An interface can be declared as the member of a class or another interface this interface can be referred to as **member interface** or **nested interface.** A nested interface can be specified public, protected or private access levels but the top-level interface is to be public or default. To use a nested interface, it should be implemented by qualifying the name of the class of interface it is a member to.

Variable in Interfaces

When an interface has a variable, it should be in the scope as a constant, thus they are set with the **final** keyword.

Extended Interfaces

Interfaces can extend another interface using the `extends` keyword. A class that implements an interface that extends another interface has to implement all the member methods in both the interfaces.

Exception Handling

An exception is a condition that occurs unexpected in a code sequence in run time that can cause an interruption in the flow of control. It can be termed as a RUN-TIME ERROR. In Java exception occurs as an object that describes the exceptional condition that occurred.

When an exception is raised an object representing the exception is created and is thrown out into the method in which the exception occurred. The exception is then caught and can be processed, else it may lead to termination of the program. Exception can be generated by the user manually or the JRE.

Exceptions thrown by Java are related to the errors that violate the syntactical rules of Java language or violation of the constraints of Java execution environment.

All exceptions are subclasses of the built-in class Throwable which has two subclasses

- **Exceptions** – This class is used for exceptional condition that can be caught and handled by the user program. **RuntimeException** is a subclass of Exception. This also provides subclass to create exceptions manually.
- **Errors** - This class defines exceptions that are not expected to be caught under normal circumstances by our program. Eg: Stack overflow, Out of Memory error etc.

There are two main types of exceptions:

- Unchecked Exceptions
- Checked Exceptions

Unchecked Exceptions

Unchecked exceptions, also known as runtime exceptions, are exceptions in Java that do not need to be explicitly caught or declared in a method's signature. They typically indicate programming errors or conditions that are beyond the control of the program. Unchecked exceptions extend the **RuntimeException** class or one of its subclasses.

Basic characteristics of unchecked exceptions are:

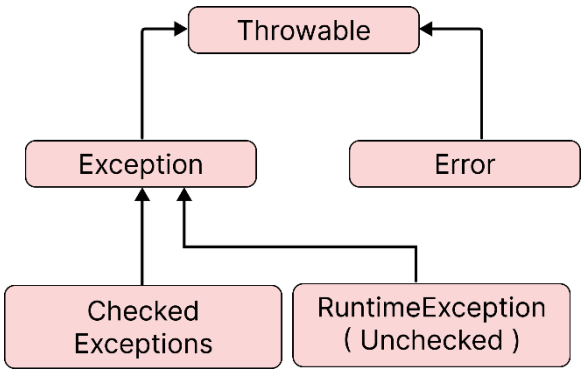
- **Not checked at compile time** – that is the unchecked errors are not expected to be thrown or caught at compile time, they are not expected to be handled by exception

handling methods. The compiler does not force the developer to handle these exceptions.

- **Usually programming errors** – these may not occur dues to syntactical error but may occur due to logical errors, system errors or due to bugs.
- **Can be caught optionally** – even though they are not expected to be handled the developer can take action and handle them manually.

Some unchecked exceptions in java.lang are:

- ArithmeticException
- ArrayIndexOutOfBoundsException
- ArrayStoreException
- ClassCastException
- IllegealThreadStateException
- IndexOutOfBoundsException
- NullPointerException
- NumberFormatException
- SecurityException



Checked Exceptions

Checked exceptions in Java are exceptions that the compiler requires to be either caught (handled) using try-catch blocks or declared in the method signature using the **throws** keyword. Checked exceptions extend the **Exception** class or one of its subclasses.

Basic characteristics of checked exceptions are:

- **Checked at compile time** – compiler checks for these exceptions and force the program to handle these exceptions.
- **Conditions that are beyond the control of the program** – this are the exceptions that interrupt the program while in execution even if there are no possible runtime errors.
- **Must be caught and handled** – these are expected to be caught and handled by the program using try-catch block.

Some checked exceptions include:

- FileNotFoundException
- IOException
- SQLEexception
- InterruptedException

Exception Vs Error

Basis of comparison	Exception	Error
Recoverable	Exception can be recovered by using try-catch block	An error cannot be recovered
Type	It can be classified into two categories i.e. checked and unchecked.	All errors in java are unchecked
Occurrence	Occurs at compile time or run-time	Occurs at run-time
Package	Belongs to java.lang.Exception	Belongs to java.lang.Error
Causes	It is mainly caused by the application itself.	It is mostly caused by the environment in which the application is running.

Exception Handling

The try-catch blocks of code a re used to handle exception. Every try block is expected to have at least one catch or finally block. The block of code that might throw an exception is included in try block. Catch block holds the block of program to be executed if an exception occurs. The finally block is used to run a block of code even if there occurs an error and has gone uncaught, before it terminates.

```
public static void main(String[] args) {
    int a=0,b=3;
    try {
        System.out.println("a/b="+ (b/a));
    } catch (ArithmeticException e) {
        e.printStackTrace();
    }

}
```

In the given snippet of code there is exception occurred as we tried divide a number by zero, if the error is not handled the program will terminate all of a sudden.

Multiple catch Clauses

A code block can produce more than one different exception in run-time based on the inputs or other factors. To handle all of the exceptions with specifically designed handling statements one has to create multiple catch statements. All of the catch statements are made to catch different exceptions and they catch in the order of definition, that is only if the first catch statement doesn't catch the thrown error, then only the second catch statement gets the exception object. If any of the catch statements catch the exception the try-catch block terminates there and the rest of code executes smoothly.

While creating catch clauses the superclass exceptions are to come after subclass exceptions, else they would go unreachable as the superclass exception takes control in first place. Such unreachable codes are considered to be **errors.**

Nested try Statements

A try statement can be nested just by creating a try-catch or try block inside another try block or catch block. Or a method with a try-catch block can be invoked in a try block to implement the same. Every exception generated in a try block is pushed into a stack and if the inner try statement does not have a catch handler for the generated exception it looks for a catch handler in the outer scope and continues till one is found and is handled. If none is found JRE takes action and prints the stack trace.

Note: every try block is supposed to have a catch or finally block not necessarily handling the generated exception if it is an inner try block.

Throw Statement

Throw statement is used to explicitly throw an exception to the method manually. A throw statement terminates the try block and passes the exception to the catch blocks. Throw statement can throw an object of the Throwable class or any of its subclass. Primitive types, such as int or char, as well as non-Throwable classes, such as String and Object, cannot be used as exceptions.

A throwable object can be thrown using a parameter in a catch clause, that is first the thrown exception is caught in a catch block with an alias and the alias can be thrown. Else a new operator can be used to create the Exception object and then throw it.

Most of Javas built-in run-time exceptions have at least two constructors:

- One with no parameters – passes the Exception object as it is
- One with a string parameter to pass a message while passing the Exception object

Throws Statement

A throws clause lists the types of exceptions that a method might throw. The throws keyword is used at the method signature after which all the possible exceptions are listed separated by comma.

If a method has an exception and it does not handle that exception, it must specify this using throws, so that callers of the method can guard themselves against that exception. All other exceptions than errors and run-time exceptions that a method can throw must be declared in the throws clause. Otherwise, it will result in an error.

Finally Statement

Finally creates a block of code that will be executed after a try-catch block has completed and before the control goes out from the try-catch block, it will execute even if an exception is thrown or not. When exceptions are thrown, execution in a method takes a nonlinear path and changes the normal flow through the method.

A finally clause is optional for a try and is required if there is no catch block.

Some times the exceptions cause the methods to return prematurely which in case can create problems. Mostly used to close a file or some classes that require a proper close statement before terminating a program is done in the finally block to ensure safe termination.

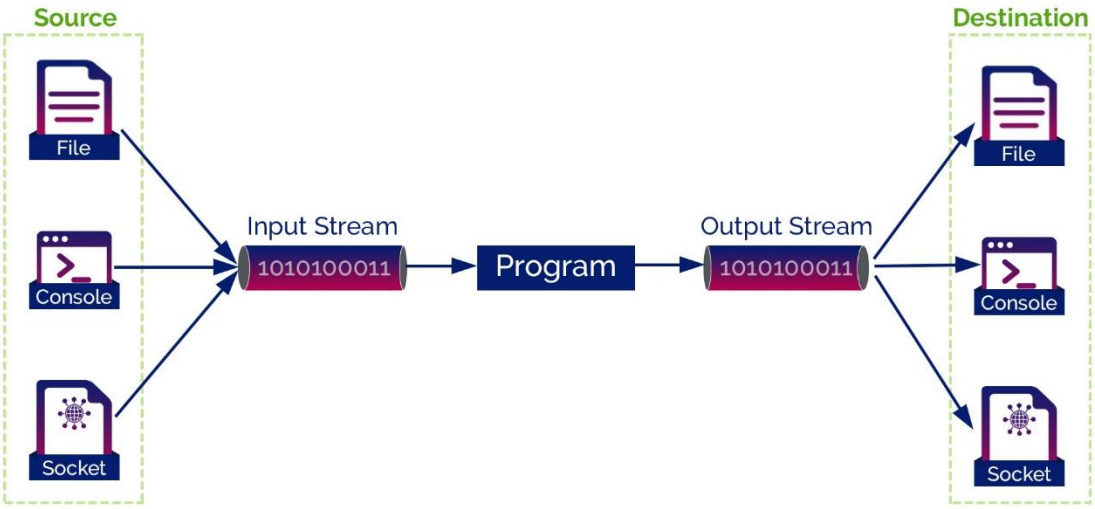
Input/Output

Only print( ) and println( ) are used frequently. All other I/O methods are not used significantly. But Java real life applications are not mainly text-based console programs and thus Java's support for console I/O is limited. Therefore Java programs perform I/O operations through streams.

Streams

A stream is an abstraction that either produces or consumes information. It is taken as a sequence of objects that supports various methods. Streams are linked to physical devices by the Java I/O system.

- Input Stream – disk file, keyboard, network socket etc.
- Output Stream – console, network connection, disk file etc.



The java.io package contains all the classes required for input and output operations. There are two types of streams:

- **Byte Streams** – used when reading or writing binary data, handles input and output in the form of bytes
- **Character Streams** – uses Unicode and handles input and output as characters

ByteStream

ByteStream Classes are defined using two class hierarchies where the top level classes are the two abstract classes:

- InputStream
- OutputStram

Each of these abstract classes has several concrete subclasses that handle the differences between various devices, such as disk files, network connections, and even memory buffers. Two of the most important are **read()** and **write().**These methods are overridden by derived stream classes.

Byte Stream Classes

CharacterStream

CharacterStream Classes are defined using two class hierarchies where the top level classes are the two abstract classes:

- Reader
- Writer

Each of these abstract classes has several concrete subclasses that handle the differences between various devices, such as disk files, network connections, and even memory buffers. Two of the most important are **read()** and **write().**These methods are overridden by derived stream classes.

CharacterStream Classes

Predefined Streams

All Java programs automatically import the java.lang package where the System class is defined, it has three predefined stream variables

- **in** – System.in – refers to standard input , that is the keyboard by default, object of InputStream
- **out** – System.out – refers to standard output stream, object of PrintStream
- **err** – System.err – refer to standard error stream, that is the console by default, object of PrintStream

All of these are defined as **final, static** and **public**.

Reading Console Input

Reading console inputs is preferred to be as character-oriented streams which is read from System.in. To obtain a character based stream that is attached to the console, wrap System.in in a **BufferedReader object.** A BufferedReader object is supposed to be a Reader which reads an InputStream. System.in is a InputStream , a Reader InputStreamReader is used to read this InputStream and then passed on to the BufferedReader.

System.in gives ByteStream input which is converted to CharacterStream by InputStreamReader and is passed to BufferedReader for reading lines from the text.

Reader is an abstract class of which InputStreamReader is a concrete subclass that converts bytes to characters using a specified charset

The basic constructors of Buffered reader and InputStreamReader are

```
BufferedReader(Reader inputReader)
```

```
InputStreamReader(InputStream inputStream)
```

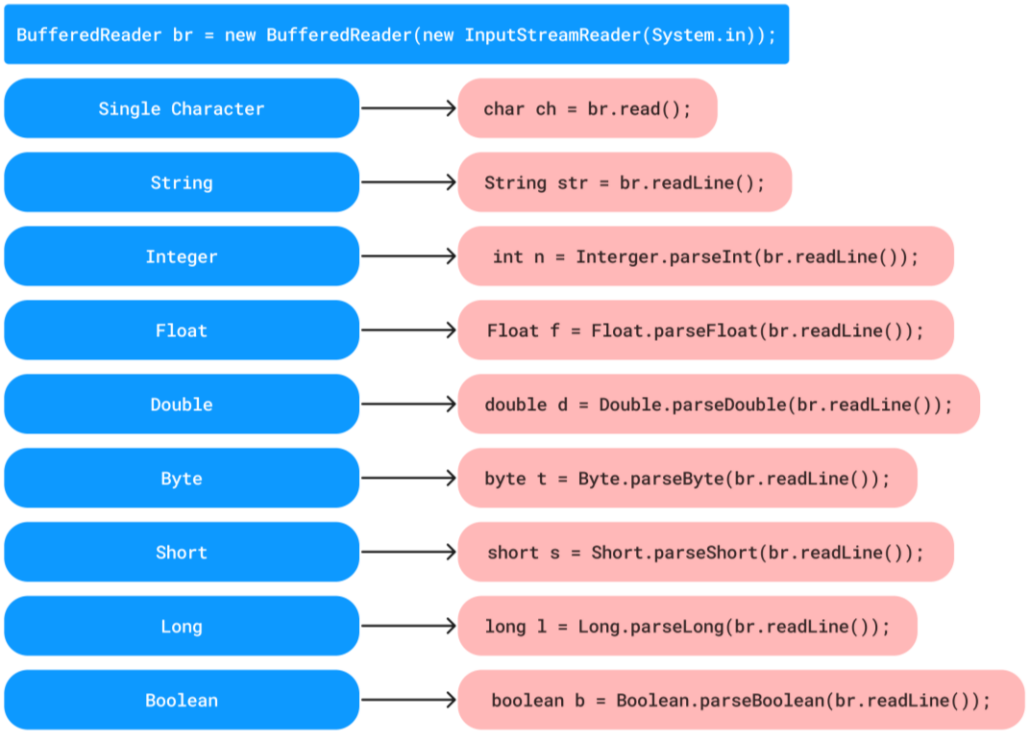
Following line of code creates a BufferedReader that is connected to a keyboard

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```

BufferedReader class is used to read the text from a character-based input stream. To make program run fast and to make reading efficient, buffering can be done using BufferedReader class.

Reading charaters form a BufferedReader is done using a read() method that throws an IOException. It reads a character and returns its integer value as per the charset and returns -1 at the encounter of end of stream.

To read a string a readLine() method is used, it returns a String object.



Using BufferedReader for different data types as console inputs

Writing Console Output

Console output is usually done through **print( )** and **println().**These methods are defined by the class **PrintStream.** PrintStream is an output stream derive from **OutputStream**. It also implements the write() method used to write to the console.

PrintWriter Class

For real-world programs, the recommended method of writing to the console using Java is through a PrintWriter stream as it is one of a character based classes rather than the System.out which is a byte based stream. PrintWriter supports the **print()** and **println( )** methods. If an argument is not a simple type, the PrintWriter methods call the object's **toString( )** method and then print the result. To write to the console by using a PrintWriter, specify System.out for the output stream and flush the stream after each new line.

```
PrintWriter pw = new PrintWriter(System.out, true);
```

System.out	PrintWriter
System.out is a byte stream.	PrintWriter should be used to write a stream of characters
System.out refers to the standard output stream(monitor).	PrintWriter is a subclass of Writer (character stream class)
System: It is a final class defined in the java.lang package.	It is used in real world programs to make it easier to internationalize the program
out: This is an instance of PrintStream type, which is a public and static member field of the System class.	

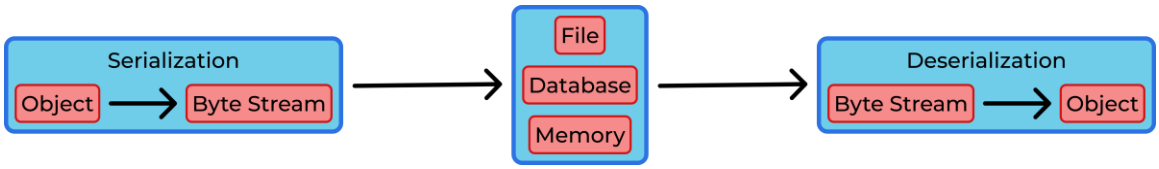
Object Streams and Serialization

Object streams support I/O(input-output) of objects. Object stream classes are:

- ObjectInputStream
- ObjectOutputStream

Serialization

Serialization is the process of writing(converting) the state of an object to a byte stream. This is useful when we have to save the state of a program in a persistent storage area or to send it over network so as to restore the object by deserialization by converting byte streams to object.



Serialization is also neede to implement Remote Method Invocation(RMI). RMI allows a Java object on one machine to invoke a method of a Java object on a different machine. If we attempt to serialize an object at the top of an object graph, all of the other referenced objects are recursively located and serialized. Similarly, during the process of deserialization, all of these objects and their references are correctly restored when deserialization is done at the top.

There are two interfaces that support serialization:

- Serializable
- Externalizable

Serializable

Only an object that implements the Serializable interface can be saved and restored by the serialization facilities. It has no members and is just used to indicate that a class may be serialized. If a class is serializable, all of its subclasses are also serializable. Variables that are declared as **transient** and **static** variables are not saved by the serialization facilities.

Externalizable

Much of the work to save and restore the state of an object occurs automatically. The developer may need to have control over these processes. it may be desirable to use compression or encryption techniques. The **Externalizable** interface is designed for these situations.

The interface defines two methods:

- void **readExternal**(ObjectInput **inStream**) **throws** IOException, ClassNotFoundException
- void **writeExternal**(ObjectOutput **outStream**) **throws** IOException

ObjectOutput

The ObjectOutput interface extends interface and supports object serialization. It defines the methods such as writeObject() method used to serialize objects.

ObjectInput

The ObjectInput interface extends the DataInput interface and defines the method such as readObject( ) method used to deserialize object.

Note: all of the methods of these interfaces throws IOException where ObjectInput also throws ClassNotFoundException

ObjectOutputStream

The ObjectOutputStream class extends the OutputStream class and implements the ObjectOutput interface. It is used to write objects to a stream.

```
ObjectOutputStream(OutputStream outStream) throws IOException
```

The outStream argument is used to define the output stream to which the serialized object is to be written, like a file. There is also an inner class to ObjectOuputStream called **PutField**. It facilitates the writing of persistent fields.

ObjectInputStream

The ObjectInputStream class extends the InputStream class and implements the ObjectInput interface. It is used to read objects from a stream.

```
ObjectInputStream(InputStream inStream) throws IOException
```

The inStream argument is used to define the input stream from which the serialized object is to be read, to be deserialized. There is also an inner class to ObjectInputStream called **GetField**. It facilitates the reading of persistent fields

Advantages of Serialization

Serialization helps in:

- Saving in the state of an object
- Transfer the Object Code form one JVM to another and recreate it using **Deserialization.** The object can be sent through network or by files

Entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.

Notes:

- If there is any static data member in a class, it will not be serialized because static is the part of class not object.
- In case of array or collection, all the objects of array or collection must be serializable. If any object is not serializable, serialization will be failed.
- If you don't want to serialize any data member of a class, you can mark it as **transient**.
- The serialization process at runtime associates an id with each Serializable class which is known as SerialVersionUID.

File Handling

Java files are byte-oriented and thus it provide methods to

- Read bytes from a file
- Write bytes to a file

The java.io.File class allows us to work with files. The File object of a file is created before working with a file.

Note : while creating File objects in Windows the path is supposed to have double backslash as a single backslash is considered to be an escape character sequence with some combination.

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 director
`mkdir()`	Boolean	Creates a directory

Some useful methods of File Class

Working with Files

Java uses file stream classes to handle File objects. The most commonly used are:

- **FileInputStream** - FileInputStream is an input stream to **read data** from a file in the form of sequence of bytes
- **FileOutputStream** - FileOutputStream class is an output stream for **writing data** to a file

To open a file just create an object of any one of these classes.

- FileInputStream – to read
- FileOutputStream – to write

The basic constructors are:

```
FileInputStream(String fileName) throws FileNotFoundException
```

```
FileOutputStream(String fileName) throws FileNotFoundException
```

The filename here is provided is given as a string. It can also be passed on as a File object after creation of File object using:

```
File file = new File(String path);
```

When an **output file** is opened, any file that is already existing with the same name as output file is **destroyed**. To avoid this truncation of data we can pass a Boolean value parameterized as append after passing the File object in the constructor so that the file will be written without deleting the already existent data.

FileNotFoundException is thrown if:

- Input stream tries to find a non-existent file
- Output stream tries to create a file but fails

Note: every file object once opened is supposed to be closed before termination of the program, using the close() method.

Reading from a file

To read from a file we have to:

1. Create a file object using the FileInputStream
2. Use the read() function to read a single byte on each call

Writing to a file

To write to a file we have to:

1. Create a file object using the FileOutputStream
2. Use write method to write byte specified values to the file

FileReader

The FileReader class creates a Reader that we can use to read the contents of a file. This created reader can be passed on to a BufferedReader to do functions like reading a line etc..

FileWriter

FileWriter creates a Writer that you can use to write to a file. For the FileWriter it can only write an character buffer array to a file and not a whole line so a special function is used to convert the String to be written to a file is stored in, the getChars() method.

```
public void getChars(int srhStartIndex, int srhEndIndex,char[]
                destArray, int destStartIndex)
```

The parameters of this method are defined as follows:

srhStartIndex: Index of the first character in the string to copy.

srhEndIndex: Index after the last character in the string to copy.

destArray: Destination array where characters will get copied.

destStartIndex: Index in the array starting from where the chars will be pushed into the array.