Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

CSE1007
Java Programming

Slot: A2+TA2

Venue: AB1-409

Module -IV

Streams & I/O Collections

Prof. Tulasi Prasad Sariki
SCSE, VIT, Chennai
www.learnersdesk.weebly.com

Course Contents:

- Java I/O streams
- Working with files
- Serialization and deserialization
- Lambda expressions
- Collection framework
  - List
  - Map
  - Set

Java I/O

- 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.
- A stream is an abstraction that either produce or consumes information
- Streams are implemented in the java.io and java.nio packages
- Predefined stream: the java.lang.System class

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

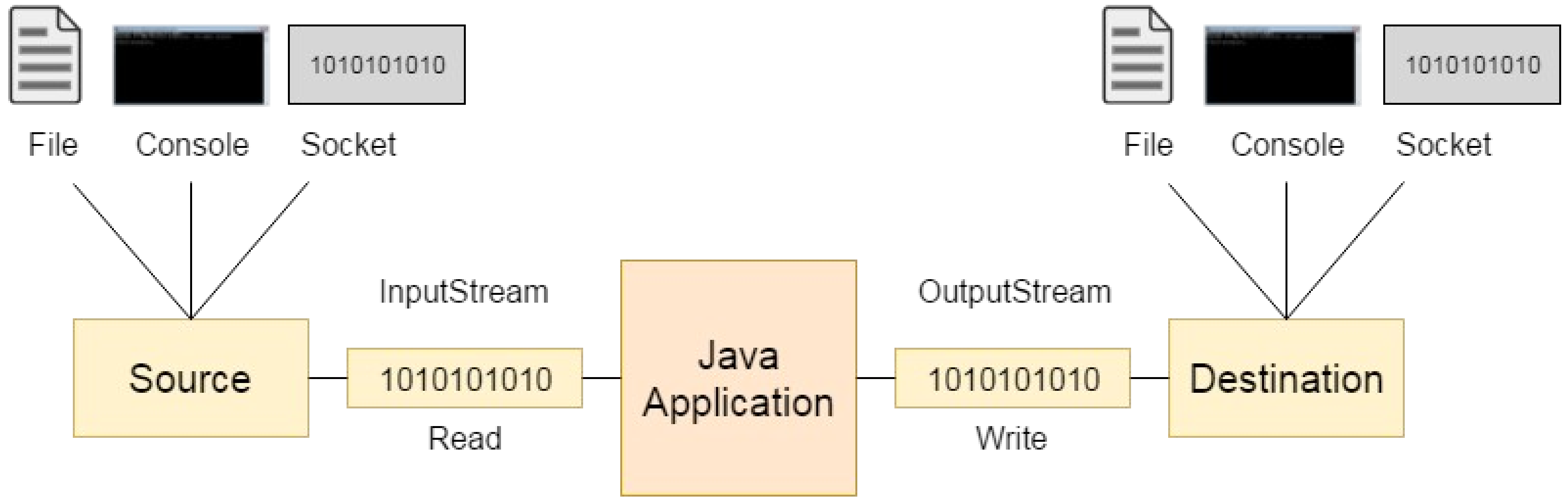
  2) System.in: standard input stream

  3) System.err: standard error stream

```
System.out.println("simple message");
System.err.println("error message");
int i=System.in.read();//returns ASCII code of 1st character
System.out.println((char)i);//will print the character
```

I/O streams

- 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.



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 class

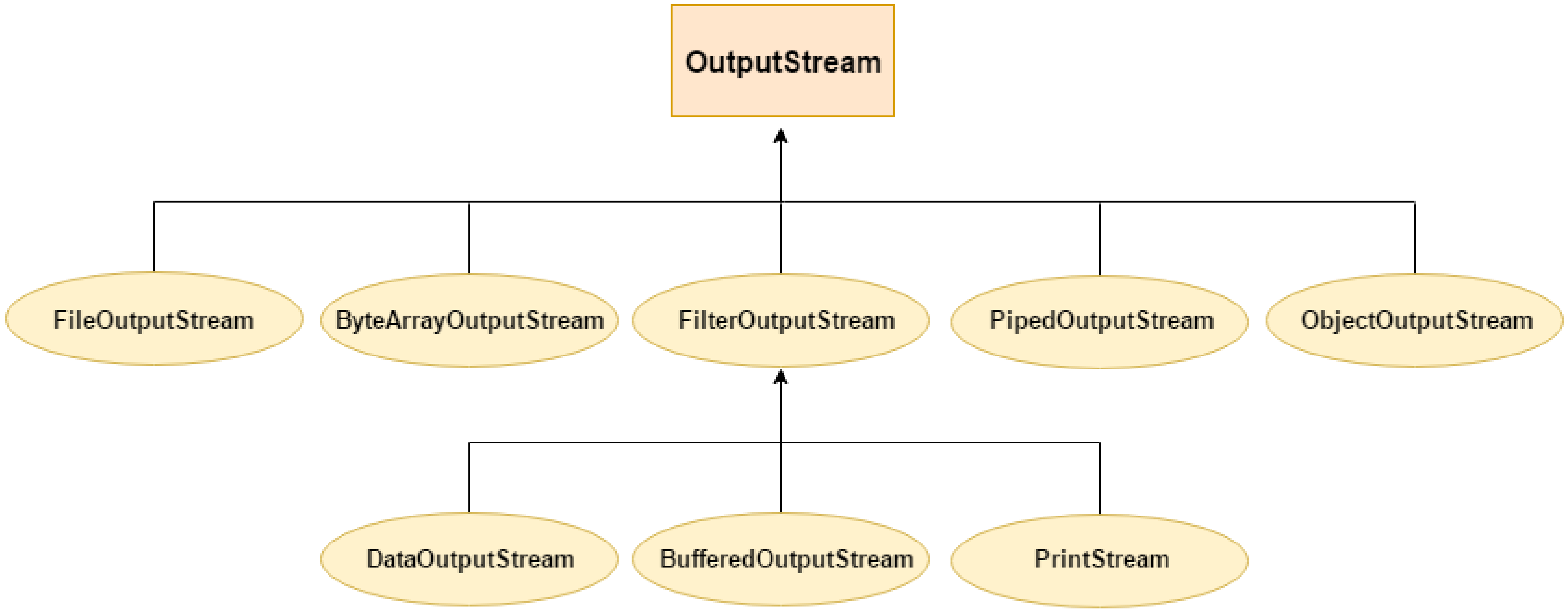
  1) public void write(int)throws IOException is used to write a byte to the current output stream.

  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.

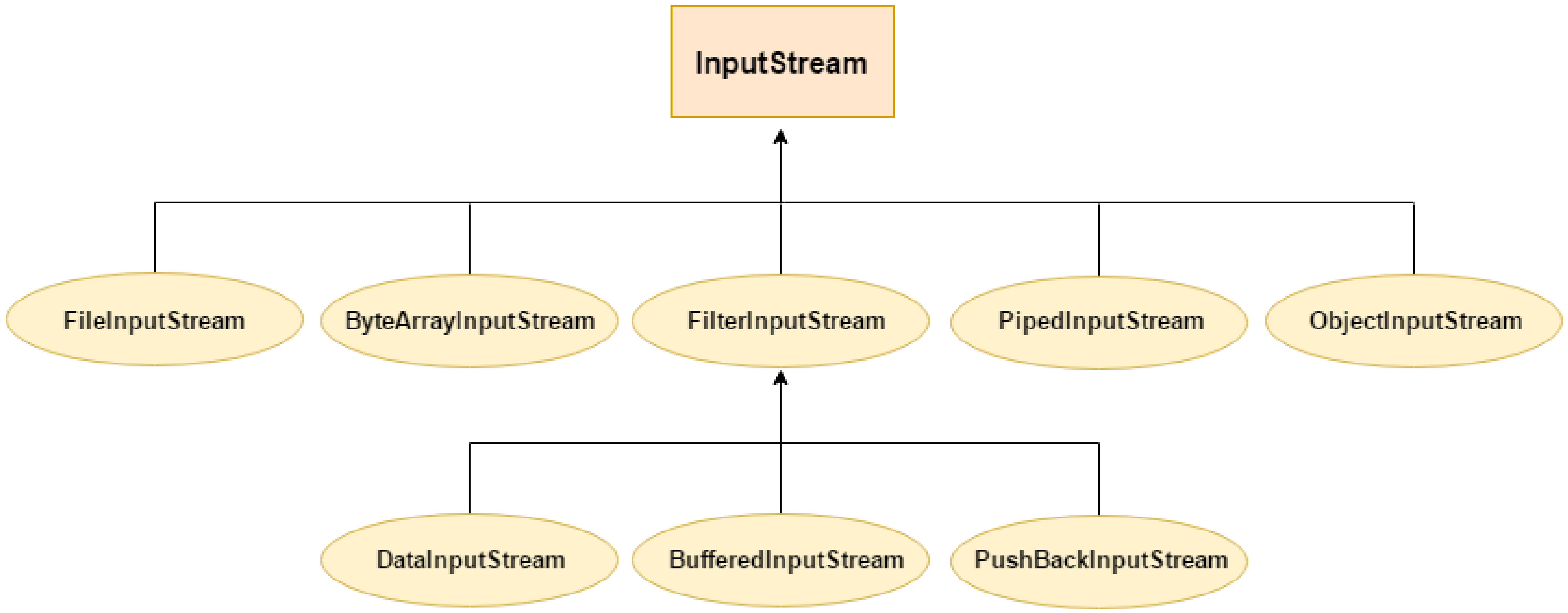
OutputStream Hierarchy



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 class
- 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



Java FileOutputStream Class

- Java FileOutputStream is an output stream used for writing data to a file.
- If you have to write primitive values into a file, use FileOutputStream class.
- You can write byte-oriented as well as character-oriented data through FileOutputStream class.
- But, for character-oriented data, it is preferred to use FileWriter than FileOutputStream.
- Synatx:
  - public class FileOutputStream extends OutputStream

FileOutputStream class methods

protected void finalize() It is used to clean up the connection with the file output stream.

void write(byte[] ary) It is used to write ary.length bytes from the byte array to the file output stream.

void write(byte[] ary, int off, int len) It is used to write len bytes from the byte array starting at offset off to the file output stream.

void write(int b) It is used to write the specified byte to the file output stream.

FileChannel getChannel() It is used to return the file channel object associated with the file output stream.

FileDescriptor getFD() It is used to return the file descriptor associated with the stream.

void close() It is used to closes the file output stream.

Example 1: write byte

```
import java.io.FileOutputStream;
public class FileOutputStreamExample {
  public static void main(String args[]){
      try{
       FileOutputStream fout=new FileOutputStream("D:\\testout.txt");
       fout.write(65);
       fout.close();
       System.out.println("success...");
      }catch(Exception e){System.out.println(e);}
    }
}
```

Example 2: write string

```
import java.io.FileOutputStream;
public class FileOutputStreamExample1 {
  public static void main(String args[]){
      try{
       FileOutputStream fout=new FileOutputStream("/home/rohan/testout1.txt");
       String s="Welcome to java.";
       byte b[]=s.getBytes();//converting string into byte array
       fout.write(b);
       fout.close();
       System.out.println("success...");
      }catch(Exception e){System.out.println(e);}
   }  }
```

Java FileInputStream Class

- Java FileInputStream class obtains input bytes from a file.
- It is used for reading byte-oriented data (streams of raw bytes) such as image data, audio, video etc. You can also read character-stream data.
- But, for reading streams of characters, it is recommended to use FileReader class.
- Java FileInputStream class declaration
  - public class FileInputStream extends InputStream

FileInputStream class methods

int available() It is used to return the estimated number of bytes that can be read from the input stream.

int read() It is used to read the byte of data from the input stream.

int read(byte[] b) It is used to read up to b.length bytes of data from the input stream.

int read(byte[] b, int off, int len) It is used to read up to len bytes of data from the input stream.

long skip(long x) It is used to skip over and discards x bytes of data from the input stream.

FileChannel getChannel() It is used to return the unique FileChannel object associated with the file input stream.

FileDescriptor getFD() It is used to return the FileDescriptor object.

protected void finalize() It is used to ensure that the close method is call when there is no more reference to the file input stream.

void close() It is used to closes the stream.

Example 1: read single character

```
import java.io.FileInputStream;
public class FileInputStreamExample {
    public static void main(String args[]){
        try{
          FileInputStream fin=new FileInputStream("./testout.txt");
          int i=fin.read();
          System.out.println((char)i);
          fin.close();
        }catch(Exception e){System.out.println(e);}
       }
}
```

Example 2: read all characters

```
import java.io.FileInputStream;
public class FileInputStreamExample1 {
    public static void main(String args[]){
        try{
          FileInputStream fin=new FileInputStream("./testout1.txt");
          int i=0;
          while((i=fin.read())!=-1){
           System.out.print((char)i);
          }
          fin.close();
        }catch(Exception e){System.out.println(e);}
       }   }
```

Java BufferedOutputStream Class

- Java BufferedOutputStream class is used for buffering an output stream. It internally uses buffer to store data. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast.
- For adding the buffer in an OutputStream, use the BufferedOutputStream class.
  - OutputStream os = new BufferedOutputStream(new FileOutputStream("./testout.txt"));
- Java BufferedOutputStream class declaration
- public class BufferedOutputStream extends FilterOutputStream

BufferedOutputStream constructors & methods

Constructors:

BufferedOutputStream(OutputStream os) It creates the new buffered output stream which is used for writing the data to the specified output stream.

BufferedOutputStream(OutputStream os, int size) It creates the new buffered output stream which is used for writing the data to the specified output stream with a specified buffer size.

Methods:

void write(int b) It writes the specified byte to the buffered output stream.

void write(byte[] b, int off, int len) It write the bytes from the specified byte-input stream into a specified byte array, starting with the given offset

void flush() It flushes the buffered output stream.

Example

```
import java.io.*;
public class BufferedOutputStreamExample{
public static void main(String args[])throws Exception{
FileOutputStream fout=new FileOutputStream("./testout.txt");
BufferedOutputStream bout=new BufferedOutputStream(fout);
String s="Welcome to VIT";
byte b[]=s.getBytes();
bout.write(b);
bout.flush();
bout.close();
 fout.close();
System.out.println("success");
   }   }
```

Java BufferedInputStream Class

- Java BufferedInputStream class is used to read information from stream.
- It internally uses buffer mechanism to make the performance fast.
- When the bytes from the stream are skipped or read, the internal buffer automatically refilled from the contained input stream, many bytes at a time.
- When a BufferedInputStream is created, an internal buffer array is created.
- Java BufferedInputStream class declaration
  - public class BufferedInputStream extends FilterInputStream

BufferedInputStream constructors

BufferedInputStream(InputStream IS)

It creates the BufferedInputStream and saves it argument, the input stream IS, for later use.

BufferedInputStream(InputStream IS, int size)

It creates the BufferedInputStream with a specified buffer size and saves it argument, the input stream IS, for later use.

BufferedInputStream methods

- int available() It returns an estimate number of bytes that can be read from the input stream without blocking by the next invocation method for the input stream.
- int read() It read the next byte of data from the input stream.
- int read(byte[] b, int off, int ln) It read the bytes from the specified byte-input stream into a specified byte array, starting with the given offset.
- void close() It closes the input stream and releases any of the system resources associated with the stream.
- void reset() It repositions the stream at a position the mark method was last called on this input stream.
- void mark(int readlimit) It sees the general contract of the mark method for the input stream.
- long skip(long x) It skips over and discards x bytes of data from the input stream.
- boolean markSupported() It tests for the input stream to support the mark and reset methods.

Example

```
import java.io.*;
public class BufferedInputStreamExample{
 public static void main(String args[]){
  try{
    FileInputStream fin=new FileInputStream("./testout.txt");
    BufferedInputStream bin=new BufferedInputStream(fin);
    int i;
    while((i=bin.read())!=-1){  System.out.print((char)i);  }
    bin.close();
    fin.close();
  }catch(Exception e){System.out.println(e);  }  }  }
```

Java PrintStream Class

- The PrintStream class provides methods to write data to another stream. The PrintStream class automatically flushes the data so there is no need to call flush() method. Moreover, its methods don't throw IOException.
- Class declaration
  - public class PrintStream extends FilterOutputStream implements Closeable. Appendable

Methods of PrintStream class

- void print(boolean b) It prints the specified boolean value.
- void print(char c) It prints the specified char value.
- void print(char[] c) It prints the specified character array values.
- void print(int I)   It prints the specified int value.
- void print(long l)   It prints the specified long value.
- void print(float f)   It prints the specified float value.
- void print(double d) It prints the specified double value.
- void print(String s)   It prints the specified string value.
- void print(Object obj)   It prints the specified object value.

Methods of PrintStream class

- void println(boolean b) It prints the specified boolean value and terminates the line
- void println(char c) It prints the specified char value and terminates the line.
- void println(char[] c) It prints the specified character array values and terminates the line.
- void println(int I) It prints the specified int value and terminates the line.
- void println(long l) It prints the specified long value and terminates the line.
- void println(float f) It prints the specified float value and terminates the line.
- void println(double d) It prints the specified double value and terminates the line.
- void println(String s) It prints the specified string value and terminates the line.

Methods of PrintStream class

- void println(Object obj   It prints the specified object value and terminates the line.
- void println()   It terminates the line only.
- void printf(Object format, Object... args)   It writes the formatted string to the current stream.
- void printf(Locale l, Object format, Object... args)   It writes the formatted string to the current stream.
- void format(Object format, Object... args)   It writes the formatted string to the current stream using specified format.
- void format(Locale l, Object format, Object... args)   It writes the formatted string to the current stream using specified format.

File Class in Java

- The File class is Java’s representation of a file or directory path name.
- Because file and directory names have different formats on different platforms, a simple string is not adequate to name them.
- The File class contains several methods for
  - working with the path name
  - deleting and renaming files
  - creating new directories
  - listing the contents of a directory
  - determining several common attributes of files and directories.

File Class in Java

- It is an abstract representation of file and directory pathnames.
- A pathname, whether abstract or in string form can be either absolute or relative. The parent of an abstract pathname may be obtained by invoking the getParent() method of this class.
- We can create the File class object by passing the filename or directory name to it. A file system may implement restrictions (access permissions) to certain operations on the actual file-system object, such as reading, writing, and executing.
- Instances of the File class are immutable; that is, once created, the abstract pathname represented by a File object will never change.

Creating a File Object?

- A File object is created by passing a String that represents the name of a file, or another File object. For example,
  - File a = new File("/home/rohan/test");
- defines an abstract file name for the test file in directory /home/rohan. This is an absolute abstract file name.

Constructors

- File(File parent, String child) : Creates a new File instance from a parent abstract pathname and a child pathname string.
- File(String pathname) : Creates a new File instance by converting the given pathname string into an abstract pathname.
- File(String parent, String child) : Creates a new File instance from a parent pathname string and a child pathname string.
- File(URI uri) : Creates a new File instance by converting the given file: URI into an abstract pathname

Methods

- boolean canExecute() : Tests whether the application can execute the file denoted by this abstract pathname.
- boolean canRead() : Tests whether the application can read the file denoted by this abstract pathname.
- boolean canWrite() : Tests whether the application can modify the file denoted by this abstract pathname.
- int compareTo(File pathname) : Compares two abstract pathnames lexicographically.
- boolean createNewFile() : Atomically creates a new, empty file named by this abstract pathname .

Methods

- static File createTempFile(String prefix, String suffix) : Creates an empty file in the default temporary-file directory.
- boolean delete() : Deletes the file or directory denoted by this abstract pathname.
- boolean equals(Object obj) : Tests this abstract pathname for equality with the given object.
- boolean exists() : Tests whether the file or directory denoted by this abstract pathname exists.
- String getAbsolutePath() : Returns the absolute pathname string of this abstract pathname.

Methods

- long getFreeSpace() : Returns the number of unallocated bytes in the partition .
- String getName() : Returns the name of the file or directory denoted by this abstract pathname.
- String getParent() : Returns the pathname string of this abstract pathname's parent.
- File getParentFile() : Returns the abstract pathname of this abstract pathname's parent.
- String getPath() : Converts this abstract pathname into a pathname string.

Methods

- boolean isDirectory() : Tests whether the file denoted by this pathname is a directory.
- boolean isFile() : Tests whether the file denoted by this abstract pathname is a normal file.
- boolean isHidden() : Tests whether the file named by this abstract pathname is a hidden file.
- long length() : Returns the length of the file denoted by this abstract pathname.
- String[] list() : Returns an array of strings naming the files and directories in the directory .

Methods

- File[] listFiles() : Returns an array of abstract pathnames denoting the files in the directory.
- boolean mkdir() : Creates the directory named by this abstract pathname.
- boolean renameTo(File dest) : Renames the file denoted by this abstract pathname.
- boolean setExecutable(boolean executable) : A convenience method to set the owner's execute permission.
- boolean setReadable(boolean readable) : A convenience method to set the owner's read permission.

Methods

- boolean setReadable(boolean readable, boolean ownerOnly) : Sets the owner's or everybody's read permission.
- boolean setReadOnly() : Marks the file or directory named so that only read operations are allowed.
- boolean setWritable(boolean writable) : A convenience method to set the owner's write permission.
- String toString() : Returns the pathname string of this abstract pathname.
- URI toURI() : Constructs a file URI that represents this abstract pathname.

Example

```
import java.io.File;
 class FileDemo {
      public static void main(String[] args) {
       String fname =args[0];
       File f = new File(fname);
       System.out.println("File name :"+f.getName());
       System.out.println("Path: "+f.getPath());
       System.out.println("Absolute path:" +f.getAbsolutePath());
       System.out.println("Parent:"+f.getParent());
       System.out.println("Exists :"+f.exists());
       if(f.exists())  {
          System.out.println("Is writeable:"+f.canWrite());
          System.out.println("Is readable"+f.canRead());
          System.out.println("Is a directory:"+f.isDirectory());
          System.out.println("File Size in bytes "+f.length());
       } } }
```

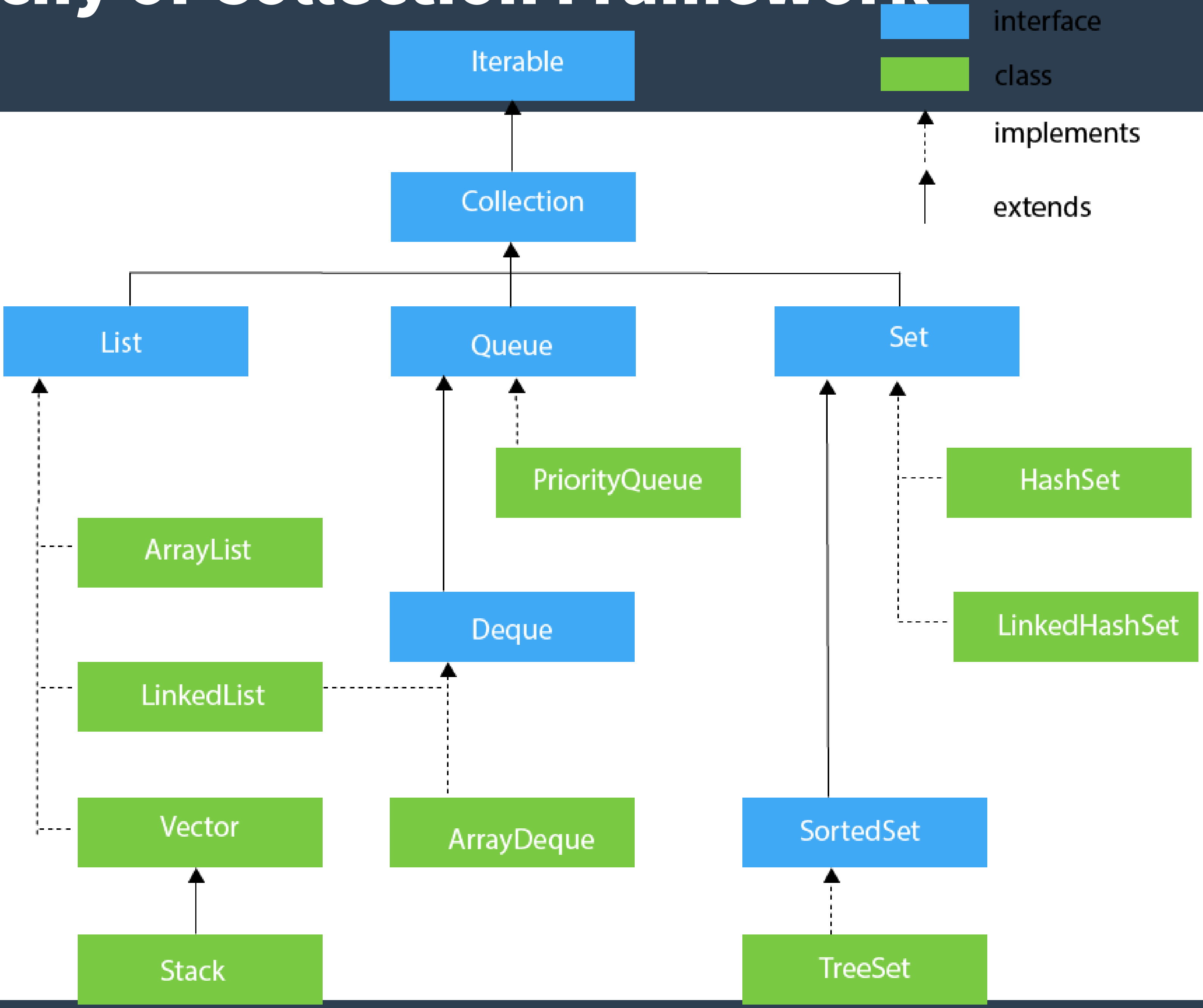
Collections in Java

- The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects.
- Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.
- The Java "Collection" classes make it easy to store and manipulate collections of information.
- We should be familiar with the collection classes so we can leverage their many built-in features in our own code.

Collections in Java

- **The three most important types are "List", "Set", and "Map".**
  - A List is like an array, except it grows and shrinks automatically as needed.
  - The Set is like the List, but automatically rejects duplicate elements.
  - The Map is a key/value dictionary that supports the efficient storage and retrieval of information by a key.

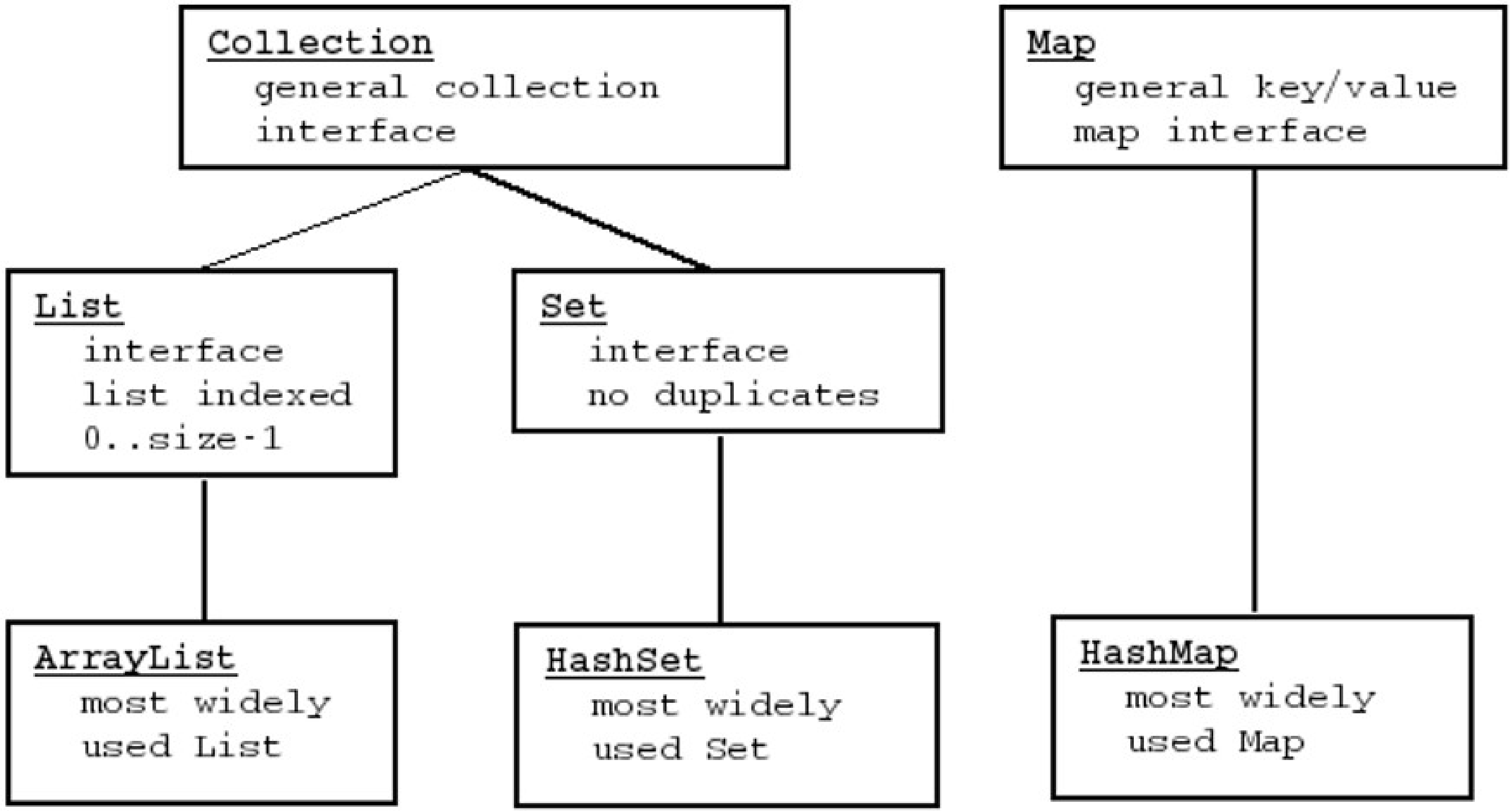
Hierarchy of Collection Framework



List, Set and Map

- The Collection interface is a general interface that includes sub-interfaces List and Set.
- If a method has a parameter of type Collection, you can pass it a List or Set and it will work fine.
- List is an interface, and ArrayList is the typically used class that implements List.
- Set is an interface, and HashSet is the commonly used class that implements Set.
- The Map interface is separate from the Collection interface. The Map interface defines a key/value lookup dictionary, and HashMap is the most commonly used Map.

Methods in the Collection interface



Lists

- The List is probably the single most useful and widely used type of Collection.
- List is a general interface, and ArrayList and LinkedList are implementing classes.
- ArrayList is the best general purpose List.
- A List is a linear structure where each element is known by an index number 0, 1, 2, ... len-1 (like an array).
- Lists can only store objects, like String and Integer, but not primitives like int.
- You cannot create a List of int, but you can create a list of Integer objects.
- Another way to say this is that the collection classes can only store pointers.

Basic List

- Here is code to create a new list to contain Strings:
- List<String> words = new ArrayList<String>();
- The "words" variable is declared to be type "List<String>" -- "List" being the general interface for all lists, and the "<String>" is the generic syntax means this is a list that contains String elements.
- On the right hand side the "new AarrayList<String>()" creates a new ArrayList of Strings, also using the "List<String>" syntax.
- The ArrayList class implements the List interface, which is how we can store a pointer to an ArrayList in a List variable.
- Using the general List type for the variable as shown here is the standard way to store an ArrayList.

Basic List

List add()

A new ArrayList is empty. The add() method adds a single element to the end of the list, like this:

```
words.add("this");
words.add("and");
words.add("that");
// words is now: {"this", "and", "that"}
words.size() // returns 3
```

The size() method returns the int size of a list (or any collection).

Basic List

- For all the collection classes, creating a new one with the default constructor gives us an empty collection.
- However, we can also call the constructor passing an existing collection argument, and this creates a new collection that is a copy.
- // Create words2 which is a copy of words
- List<String> words2 = new ArrayList<String>(words);
- Note: this just copies the elements (pointers) that are in "words" into "words2" .

Basic List

- List Foreach
- To iterates over all the elements in a list
- int lengthSum = 0;
- for (String str: words) {
- lengthSum += str.length();
- }
- Each time through the loop, the "str" variable above takes on the next String element in the words list.
- It is not valid to modify (add or remove) elements from a list while a foreach is iterating over that list, so it would be an error to put a words.add("hi") inside the above loop.
- The foreach simply goes through the list once from start to finish, and "break" works to exit the loop early.

Basic List

- List get()
- Is an other forms of iteration.
- The elements in a List are indexed 0..size-1, with the first element at 0, the next at 1, the next at 2, and so on up through size-1.
- The get(int index) method returns an element by its index number:
- // suppose words is {"this", "and", "that"}
- words.get(0) // returns "this"
- words.get(2) // returns "that"
- words.get(3) // ERROR index out of bounds

Basic List

- List For Loop

```
for (int i=0; i<words.size(); i++) {
String str = words.get(i); // do something with str
}
```

- // iterate through the words backwards, skipping every other one

```
for (int i = words.size()-1; i >= 0; i = i-2) {
String str = words.get(i);
// do something with str
}
```

Basic List Methods

- Basic list methods that works to set/add/remove elements in a list using index numbers to identify elements.
- get(int index) -- returns the element at the given index.
- set(int index, Object obj) -- sets the element at the given index in a list.
- set() does not change the length of the list, it just changes what element is at an index.
- add(Object obj) -- adds a new element at the end of the list.
- add(int index, Object obj) -- adds a new element into the list at the given index, shifting any existing elements at greater index positions over to make a space.
- remove(int index) -- removes the element at the given index, shifting any elements at greater index positions down to take up the space

Collection Utility Methods

- int size() -- number of elements in the collection
- boolean isEmpty() -- true if the collection is empty
- boolean contains(Object target) -- true if the collection contains the given target element
- boolean containsAll(Collection coll) -- true if the collection contains all of the elements
- in the given collection.
- void clear() -- removes all the elements in the collection, setting it back to an empty state.
- boolean remove(Object target) -- searches for and removes the first instance of the target if found. Returns true if an element is found and removed.

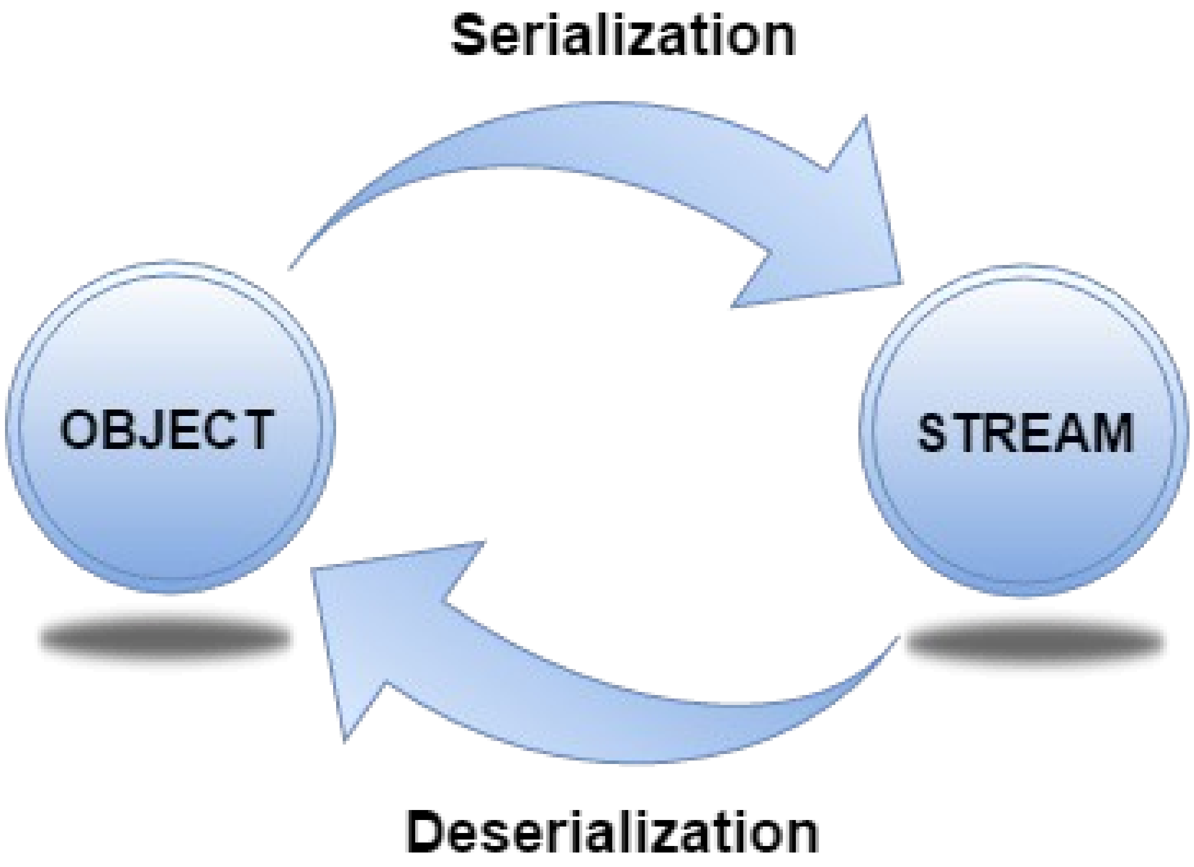
Collection Utility Methods

- boolean removeAll(Collection coll) -- removes from the receiver collection all of the elements which appear in the given collection (returns true if changed).
- boolean addAll(Collection coll) -- adds to the receiver collection all of the elements in the given collection (returns true if changed).
- boolean retainAll(Collection coll) -- retains in the receiver collection only the elements which also appear in the given collection.
- Object[] toArray() -- builds and returns an Object[] array containing the elements from the collection.
- For List alone
- int indexOf(Object target) – returns the int index of the first appearance of target in the list, or -1 if not found.
- List subList(int fromIndex, int toIndex) -- returns a new list that represents the part of the original list between fromIndex up to but not including toIndex.

Serialization and Deserialization in Java

- **Serialization in Java is a mechanism of writing the state of an object into a byte stream.**
  - It is mainly used in Hibernate, RMI, JPA, EJB and JMS technologies.
- **The reverse operation of serialization is called deserialization.**
- **Advantages of Java Serialization**
  - It is mainly used to travel object's state on the network (which is known as marshaling).
  - To save/persist state of an object.

Serialization and Deserialization in Java



java.io.Serializable interface

- Serializable is a marker interface (has no data member and method). It is used to "mark" Java classes so that objects of these classes may get the certain capability.
- It must be implemented by the class whose object you want to persist.
- The String class and all the wrapper classes implement the java.io.Serializable interface by default.

java.io.Serializable interface

- The ObjectOutputStream class contains writeObject() method for serializing an Object.

```
public final void writeObject(Object obj)
                throws IOException
```

- The ObjectInputStream class contains readObject() method for deserializing an object.

```
public final Object readObject()
              throws IOException,
            ClassNotFoundException
```

Points to remember

- 1. If a parent class has implemented Serializable interface then child class doesn’t need to implement it but vice-versa is not true.
- 2. Only non-static data members are saved via Serialization process.
- 3. Static data members and transient data members are not saved via Serialization process. So, if you don’t want to save value of a non-static data member then make it transient.
- 4. Constructor of object is never called when an object is deserialized.
- 5. Associated objects must be implementing Serializable interface.

Example

```
import java.io.*;
class Emp implements Serializable {
   transient int a;
   transient String name;
   static int b;
   int age;
public Emp(String name, int age, int a, int b) {
      this.name = name;
      this.age = age;
      this.a = a;
      this.b = b; }   }
```

Example

```
public class SerialDemo1{
public static void printdata(Emp object1)   {
      System.out.println("name = " + object1.name);
      System.out.println("age = " + object1.age);
      System.out.println("a = " + object1.a);
      System.out.println("b = " + object1.b); }
public static void main(String[] args) {
      Emp object = new Emp("VIT", 20, 2, 1000);
      String filename = "cse1007.txt";
      try {
         FileOutputStream file = new FileOutputStream(filename);
         ObjectOutputStream out = new ObjectOutputStream(file);
```

Example

```
out.writeObject(object);
      out.close();
      file.close();
      System.out.println("Object has been serialized\n"   + "Data before Deserialization.");
      printdata(object);
      object.b = 2000; }
    catch (IOException ex) {  System.out.println("IOException is caught"); }
     object = null;
    try {
     FileInputStream file = new FileInputStream(filename);
      ObjectInputStream in = new ObjectInputStream(file);
      object = (Emp)in.readObject();
```

Example

```
      in.close();

      file.close();

      System.out.println("Object has been deserialized\n"  + "Data after
Deserialization.");

      printdata(object);

    }

     catch (IOException ex) {

      System.out.println("IOException is caught");

    }

    catch (ClassNotFoundException ex) {

      System.out.println("ClassNotFoundException is caught");

    }   } }
```

Java Lambda Expressions

- Lambda expression is a new and important feature of Java which was included in Java SE 8. It provides a clear and concise way to represent one method interface using an expression.
- It is very useful in collection library. It helps to iterate, filter and extract data from collection.
- The Lambda expression is used to provide the implementation of an interface which has functional interface. It saves a lot of code. In case of lambda expression, we don't need to define the method again for providing the implementation. Here, we just write the implementation code.

Java Lambda Expressions

- **Java lambda expression is treated as a function, so compiler does not create .class file.**
- **Functional Interface**
  - Lambda expression provides implementation of functional interface. An interface which has only one abstract method is called functional interface. Java provides an anotation @FunctionalInterface, which is used to declare an interface as functional interface.

Java Lambda Expressions

- **Why use Lambda Expression**
  - To provide the implementation of Functional interface.
  - Less coding.
- **Java Lambda Expression Syntax**
  - (argument-list) -> {body}
- **Java lambda expression is consisted of three components.**
  - 1) Argument-list: It can be empty or non-empty as well.
  - 2) Arrow-token: It is used to link arguments-list and body of expression.
  - 3) Body: It contains expressions and statements for lambda expression.

Example

- interface Drawable{ public void draw(); }
- public class LambdaExpressionExample {
-     public static void main(String[] args) {
-         int width=10;
  - //without lambda, Drawable implementation using anonymous class
-         Drawable d=new Drawable() {
-      public void draw(){System.out.println("Drawing "+width);} };
-         d.draw();   } }

Example

- @FunctionalInterface //It is optional
- interface Drawable{ public void draw(); }
- public class LambdaExpressionExample2 {
- public static void main(String[] args) {
- int width=10; //with lambda
- Drawable d2=()-> { System.out.println("Drawing "+width); };
- d2.draw(); } }

Lambda Expression Exp: No Parameter

```
  interface Sayable {  public String say(); }
  public class LambdaExpDemo2{
  public static void main(String[] args)
{
     Sayable s=()->{  return "I have nothing to say.";  };
     System.out.println(s.say());
  }
  }
```

Lambda Expression Exp: Single Parameter

```
interface Sayable{   public String say(String name);  }
  public class LambdaExpDemo3{
   public static void main(String[] args) {
      // Lambda expression with single parameter.
      Sayable s1=(name)->{  return "Hello, "+name;  };
      System.out.println(s1.say("Sonoo"));
      // You can omit function parentheses
      Sayable s2= name ->{  return "Hello, "+name;};
      System.out.println(s2.say("Sonoo"));
   } }
```

Lambda Expression Exp: Multiple Parameters

```
•   interface Addable{   int add(int a,int b);   }
•   public class LambdaExpDemo4{
•       public static void main(String[] args) {
•           // Multiple parameters in lambda expression
•           Addable ad1=(a,b)->(a+b);
•           System.out.println(ad1.add(10,20));
•           // Multiple parameters with data type in lambda expression
•           Addable ad2=(int a,int b)->(a+b);
•           System.out.println(ad2.add(100,200));
•       }    }
```

Lambda Expression Exp: with List

- import java.util.*;
- public class LambdaExpDemo5{
-     public static void main(String[] args) {
-         List<String> list=new ArrayList<String>();
-         list.add("ankit");
-         list.add("mayank");
-         list.add("irfan");
-         list.add("jai");
-         list.forEach((n)->System.out.println(n));
-     }   }

References

- https://docs.oracle.com/javase/tutorial/
- https://www.javatpoint.com/features-of-java
- http://www.java2novice.com/java-fundamentals/
- http://archive.oreilly.com/oreillyschool/courses/java2/
- https://docs.oracle.com/javase/7/docs/api/
- https://www.javatpoint.com/
- https://beginnersbook.com/java-tutorial-for-beginners-with-examples/
- https://www.javaworld.com/
- https://www.geeksforgeeks.org/

Thank You!