Java Programming

UNIT - II

by

Mr. K. Srikar Goud

Asst. Professor

Department of Information Technology

Packages in Java

- Introduction
- It is necessary in software development to create several classes and interfaces.
- After creating these classes and interfaces, it is better if they are divided into some groups depending on their relationship.
- Thus, the classes and interfaces which handle similar or same task are put into the same directory.
- This directory or folder is also called a package.

Packages in Java

- Package:

- A package represents a directory that contains related group of classes and interfaces.

- For example, when we write statements like:
  Import java.io.*;

- We are importing classes of java.io package.
- Here, java is a directory name and io is another sub directory within it.
- And the '*' represents all the classes and interfaces of that io sub directory.

Packages in Java

- Package:
- Package in java can be categorized in two form, built-in package and user-defined package.
- There are many built-in packages such as
- java,
- lang,
- awt,
- javax,
- swing,
- net,
- io,
- util,
- sql etc.

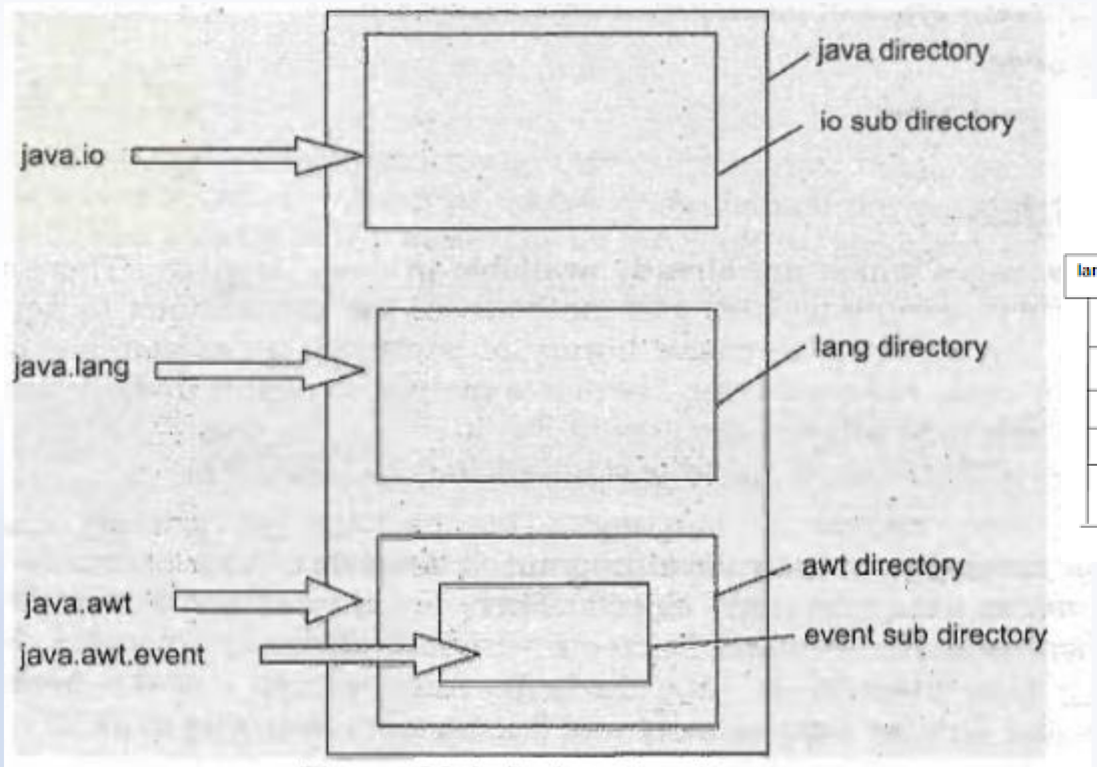
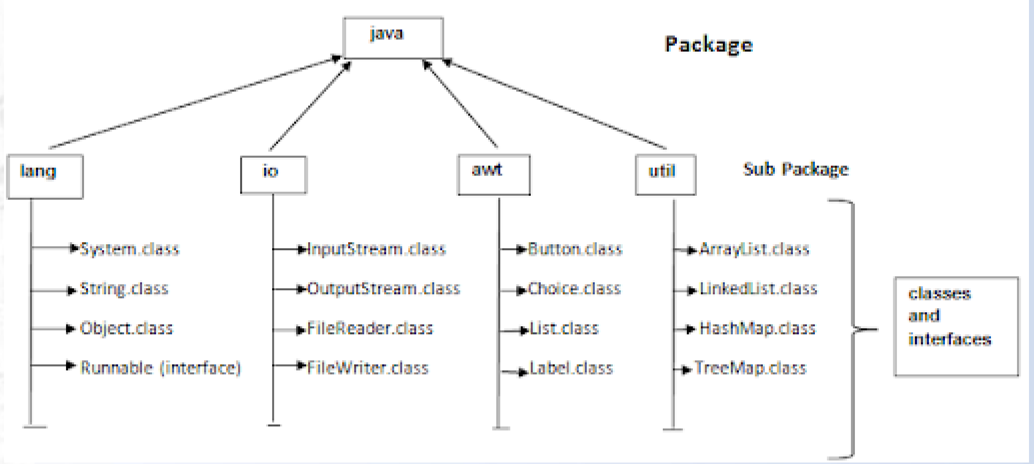


Figure 20.1 Package is a directory



Packages in Java

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) Java package removes naming collision

Packages in Java

- Creating Package:

- The **package** **keyword** is used to create a package in java.

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

Packages in Java

- Compiling Package:

- If you are not using any IDE, you need to follow the **syntax** given below:

- javac -d directory javafilename

For **example**

```
javac -d . Simple.java
```

- The -d switch specifies the destination where to put the generated class file.
- You can use any directory name like /home (in case of Linux),
- d:/abc (in case of windows) etc.
- If you want to keep the package within the same directory, you can use . (dot).

Packages in Java

- Running Package:

- You need to use fully qualified name e.g. mypack.Simple etc to run the class.

To Compile: javac -d . Simple.java

To Run: java mypack.Simple

- The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.

Packages in Java

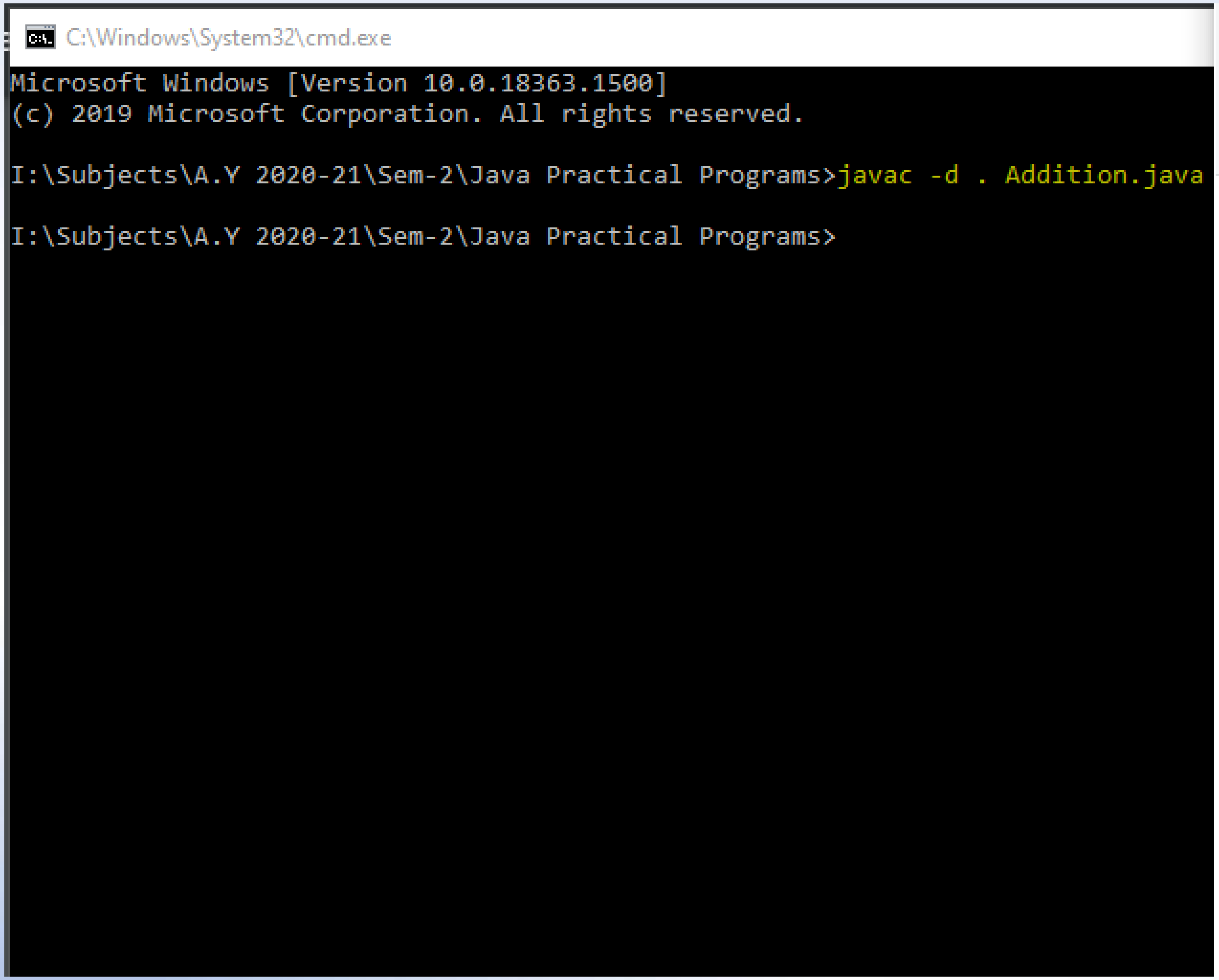
- Example Program

//Write a program to create a package with the name pack and store Addition class in it.

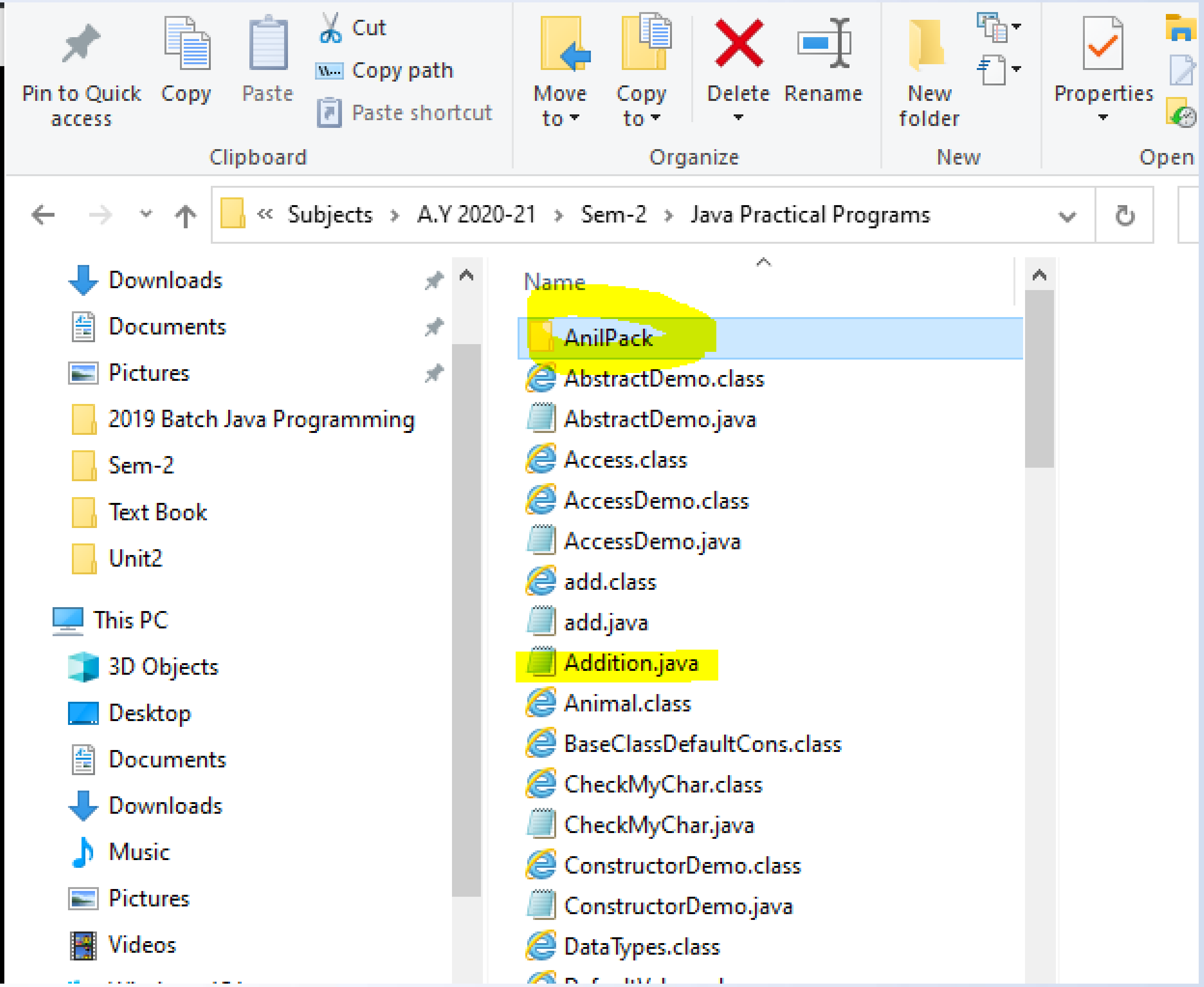
```
package AnilPack;  // AnilPack is the package name
class Addition {
            private double d1, d2;
            public Addition (double a, double b) {
            d1 = a;
            d2 = b;
            }
            public void sum(){
            System.out.println("Sum is : " +(d1+d2));
            }
}
```

Packages in Java

- Running Package:



```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.18363.1500]
(c) 2019 Microsoft Corporation. All rights reserved.

I:\Subjects\A.Y 2020-21\Sem-2\Java Practical Programs>javac -d . Addition.java

I:\Subjects\A.Y 2020-21\Sem-2\Java Practical Programs>
```



AnilPack
AbstractDemo.class
AbstractDemo.java
Access.class
AccessDemo.class
AccessDemo.java
add.class
add.java
Addition.java
Animal.class
BaseClassDefaultCons.class
CheckMyChar.class
CheckMyChar.java
ConstructorDemo.class
ConstructorDemo.java
DataTypes.class

Packages in Java

- Running Package:

```
To Compile: javac -d . Addition.java
To Run: java AnilPack.Addition
```

- The preceding command means create a package (-d) in the current directory (.) and store Addition.class flie there in the package.
- The package name is specified in the program as AnilPack
- So the Java compiler creates a directory in I:\Subjects\A.Y 2020-21\Sem-2\Java Practical Programs with the name as AnilPack and stores Addition.class there.

- So, our package with Addition class is ready.

Packages in Java

- The next step is to use the Addition class and sum () method in a program.
- For this purpose, we write another class UseAnilPack as shown in Program
- In this program, we can refer to the Addition class of package pack using membership operator as,

```
AnilPack.Addition
```

- Now, to create an object to Addition class, we can write as:

```
AnilPack.Addition obj  = new AnilPack.Addition(10, 15.5);
```

Packages in Java

- Write a program which depicts how to use the Addition class of package pack

```
class UseAnilPack {
public static void main(String args[]) {
AnilPack.Addition obj = new AnilPack.Addition(10, 10.5);
obj.sum();
}
}
```

Packages in Java

- Write a program which depicts how to use the Addition class of package pack

```
class use {
public static void main(String args[]) {
AnilPack.Addition obj = new AnilPack.Addition(10, 10.5);
obj.sum();
}
}
```

Everytime we refer to a class of a package, we should write the package name before the class name as AnilPack.Addition in the preceding program. This is inconvenient for the programmer.

Packages in Java

To overcome this, we can use import statement only once in the beginning of the program, as:

```
import AnilPack.Addition;
```

Once the import statement is written as shown earlier, we need not to use the package name before class name in the rest of the program and we can create the object to Addition class, in a normal way as:

```
Addition obj  = new Addition (10, 10.5);
```

Packages in Java

```
Import AnilPack.Addition;

class UseAnilPack {
public static void main(String args[]) {
Addition obj = new Addition(10, 10.5);
obj.sum();
}
}
```

Packages in Java

Write a program to add another class Subtraction to the same package AnilPack.

```
package AnilPack;
public class Subtraction {
        public static double sub(double a, double b){
        return (a - b);
        }
}
```

Packages in Java

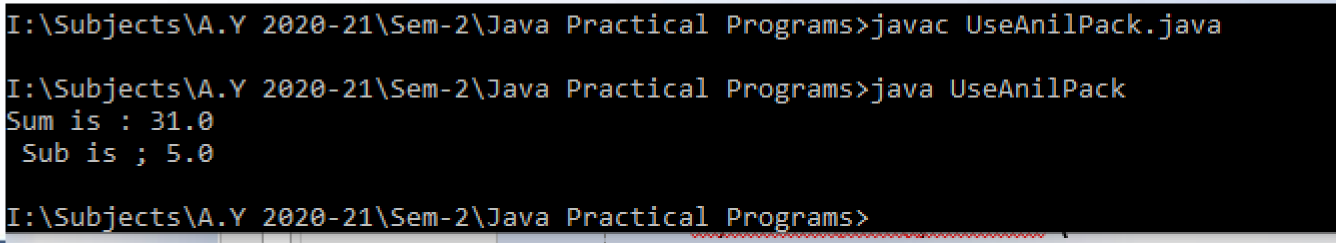
Write a program to add another class Subtraction to the same package AnilPack.

```
package AnilPack;
public class Subtraction {
        public static double sub(double a, double b){
        return (a - b);
        }
}
```

Packages in Java

Let us make a program using both the Addition and Subtraction classes of the package AnilPack

```
import AnilPack.Addition;
import AnilPack.Subtraction;
class UseAnilPack {
public static void main(String args[]){
      Addition obj = new Addition ( 10.5,  20.5);
      obj.sum();

      double res = Subtraction.sub(15 , 10);
      System.out.println (" Sub is : " + res);
}
}
```

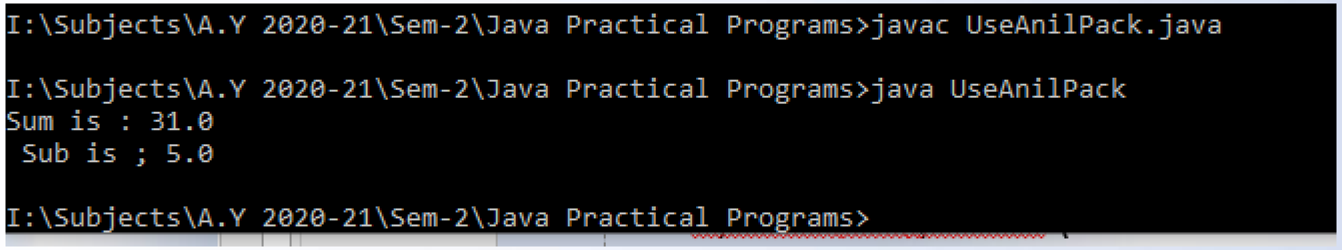


```
I:\Subjects\A.Y 2020-21\Sem-2\Java Practical Programs>javac UseAnilPack.java

I:\Subjects\A.Y 2020-21\Sem-2\Java Practical Programs>java UseAnilPack
Sum is : 31.0
 Sub is ; 5.0

I:\Subjects\A.Y 2020-21\Sem-2\Java Practical Programs>
```

Packages in Java

Final Program

```
import AnilPack.*;
class UseAnilPack {
public static void main(String args[]){
Addition obj = new Addition ( 10.5,  20.5);
obj.sum();

double res = Subtraction.sub(15 , 10);
System.out.println (" Sub is : " + res);
}
}
```



```
I:\Subjects\A.Y 2020-21\Sem-2\Java Practical Programs>javac UseAnilPack.java

I:\Subjects\A.Y 2020-21\Sem-2\Java Practical Programs>java UseAnilPack
Sum is : 31.0
 Sub is ; 5.0

I:\Subjects\A.Y 2020-21\Sem-2\Java Practical Programs>
```

Packages in Java

CLASSPATH

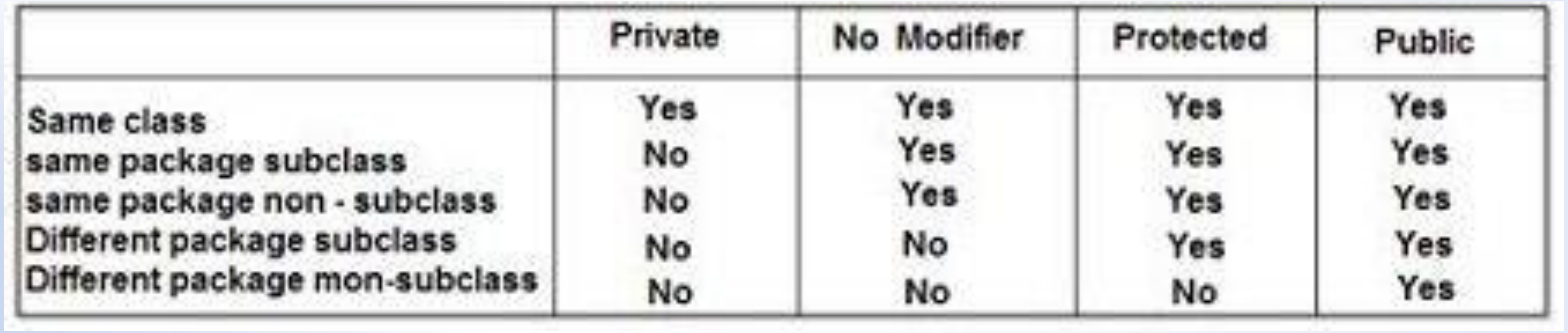
Of course, the package AnilPack is available in the current directory.
If the package is not available the current directory, then what happens? Suppose our program is running in C: \ and the package AnilPack is available in the directory D: \sub.
In this case, the compiler should be given information regarding the package location by mentioning the directory name of the package in class path.

Class path represents an Operating System environment variable which stores active directory path such that all the flies in those directories are available to any programs in the system
Generally, it is written in all capital letters as CLASSPATH.

Packages in Java

Access Protection in Packages

- Access modifiers define the scope of the class and its members (data and methods).
- For example, private members are accessible within the same class members (methods).
- Java provides many levels of security that provides the visibility of members (variables and methods) within the classes, subclasses, and packages.
- The three main access modifiers *private, public* and *protected* provides a range of ways to access required by these categories



	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
same package subclass	No	Yes	Yes	Yes
same package non - subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package mon-subclass	No	No	No	Yes

Interface in Java

- Introduction
- An Interface is a specification of method prototypes. This means, only method name written in the interface without method bodies.
- An interface will have 0 or more abstract methods which are all public and abstract by default
- An interface can have variables which are public static and final by default. This means variables of interfaces are constants
- None of the methods in interface can be private, protected or static
- All the methods of interface should be implemented in its implementation classes
- Interface reference can refer to the objects of its implemented classes.
- When an interface is written, any third party vendor can provide implemented classes.
- An interface can extend another interface.
- An interface cannot implement another interface
- A class can implement (not extend) multiple interfaces

Interface in Java

- Introduction

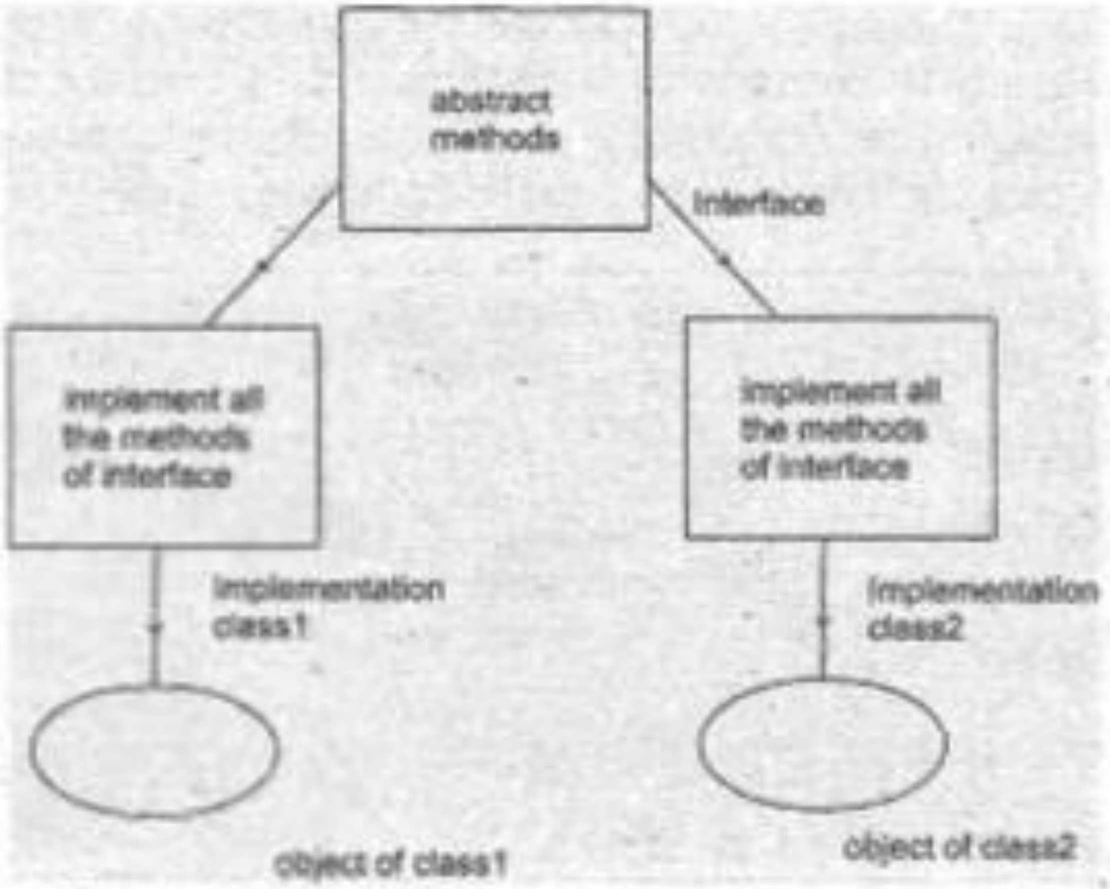


Figure 19.1 Interface and its implementation classes

Interface in Java

- Interface Example

```
interface Test1{
          public void display();
}

class TestDemo implements Test1{
     public void display() {
     System.out.println("Interface Demo");
     }
}
public class InterfaceDemo2 {
public static void main(String[] args) {
     TestDemo t = new TestDemo();
     t.display();
}
}
```

Interface in Java

- Interface Example

```
interface Test1{
            public void display();
}

class TestDemo implements Test1{
      public void display() {
      System.out.println("Interface Demo");
      }
}
public class InterfaceDemo2 {
public static void main(String[] args) {
      TestDemo t = new TestDemo();
      t.display();
}
}
```

Interface in Java

- Nested Interface Example

```
interface OuterInterface{
        void outerMethod();

        interface InnerInterface{
                void innerMethod();
        }
}

class OnlyOuter implements OuterInterface{
        public void outerMethod() {
                System.out.println("This is OuterInterface method");
        }
}

class OnlyInner implements OuterInterface.InnerInterface{
        public void innerMethod() {
                System.out.println("This is InnerInterface method");
        }
}

public class NestedInterfaceExample {

        public static void main(String[] args) {
                OnlyOuter obj_1 = new OnlyOuter();
                OnlyInner obj_2 = new OnlyInner();

                obj_1.outerMethod();
                obj_2.innerMethod();
        }
}
```

Interface in Java

- Nested Interface Example

```
class OuterClass{

interface InnerInterface{
void innerMethod();
}
}

class ImplementingClass implements OuterClass.InnerInterface{
public void innerMethod() {
System.out.println("This is InnerInterface method");
}
}

public class InterfaceInsideClass {
public static void main(String[] args) {
ImplementingClass obj = new ImplementingClass();

obj.innerMethod();
}
}
```

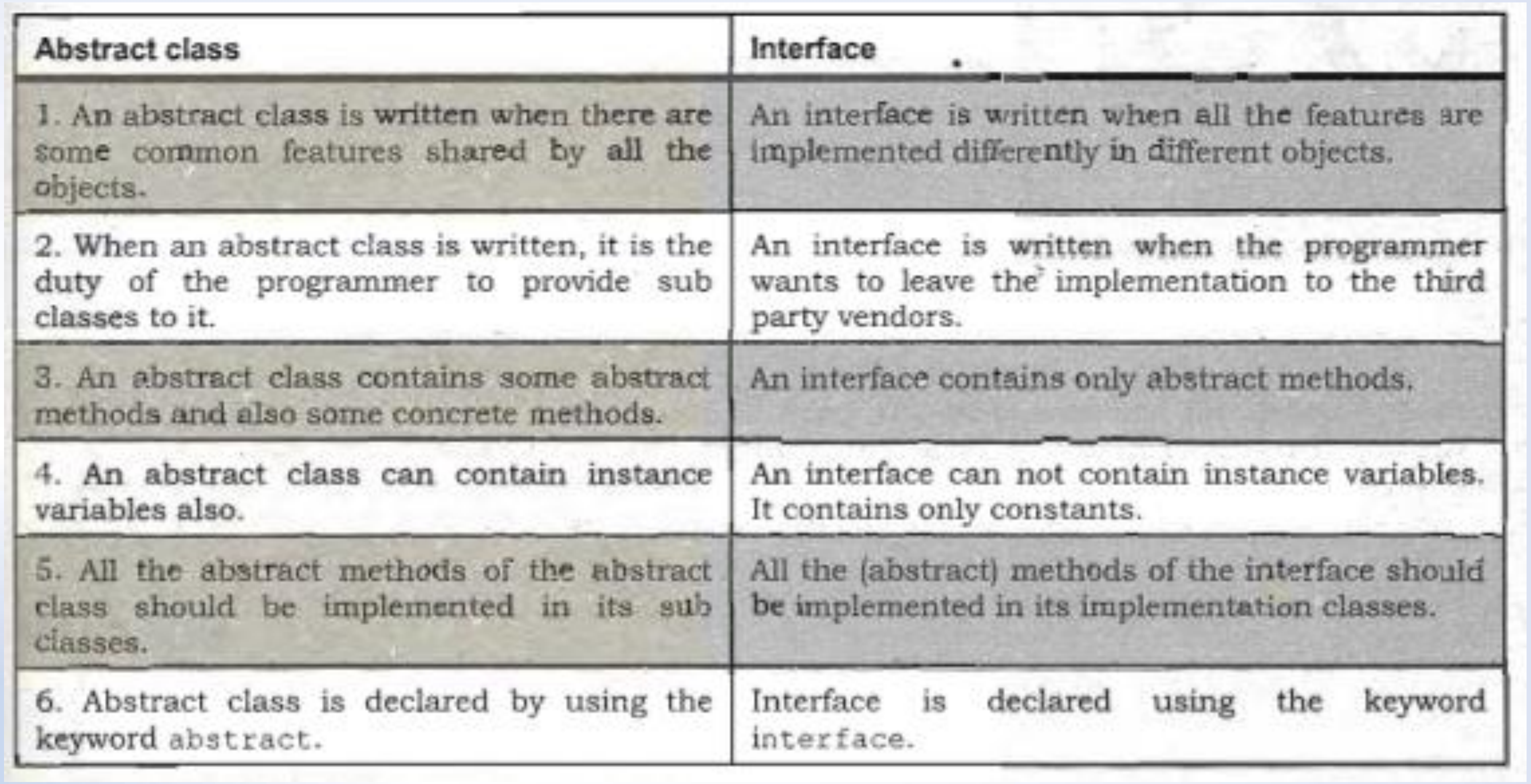
Interface in Java

- Extending Interface Example

```
interface AB {
  void funcA();
}
interface BC extends AB {
  void funcB();
}
class CD implements BC {
  public void funcA() {
    System.out.println("This is funcA");
  }
  public void funcB() {
    System.out.println("This is funcB");
  }
}
public class InterfaceExtendDemo {
public static void main(String[] args) {
CD obj = new CD();
    obj.funcA();
    obj.funcB();
}
}
```

Interface in Java

- Introduction



Abstract class	Interface
1. An abstract class is written when there are some common features shared by all the objects.	An interface is written when all the features are implemented differently in different objects.
2. When an abstract class is written, it is the duty of the programmer to provide sub classes to it.	An interface is written when the programmer wants to leave the implementation to the third party vendors.
3. An abstract class contains some abstract methods and also some concrete methods.	An interface contains only abstract methods.
4. An abstract class can contain instance variables also.	An interface can not contain instance variables. It contains only constants.
5. All the abstract methods of the abstract class should be implemented in its sub classes.	All the (abstract) methods of the interface should be implemented in its implementation classes.
6. Abstract class is declared by using the keyword `abstract`.	Interface is declared using the keyword `interface`.

Interface in Java

- Summary of Interfaces

  - 🔔 An interface is a container of abstract methods and static final variables.
  - 🔔 An interface, implemented by a class. (**class implements interface**).
  - 🔔 An interface may extend another interface. (**Interface extends Interface**).
  - 🔔 An interface never implements another interface, or class.
  - 🔔 A class may implement any number of interfaces.
  - 🔔 We can not instantiate an interface.
  - 🔔 Specifying the keyword abstract for interface methods is optional, it automatically added.
  - 🔔 All the members of an interface are public by default.

Stream based I/O (java.io) in Java

- **Stream based I/O** (java.io)

- **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.

Stream based I/O (java.io) in Java

- **Stream based I/O** (java.io)

Stream

- A stream is a sequence of data.
- In Java, a stream is composed of bytes.
- A stream can be associated with a file, an Internet resource (Ex: a socket), to a pipe or a memory buffer.

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

see the code to print **output and an error** message to the console.

Stream based I/O (java.io) in Java

- **Stream based I/O** (java.io)

- import java.io.*;

OutputStream

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

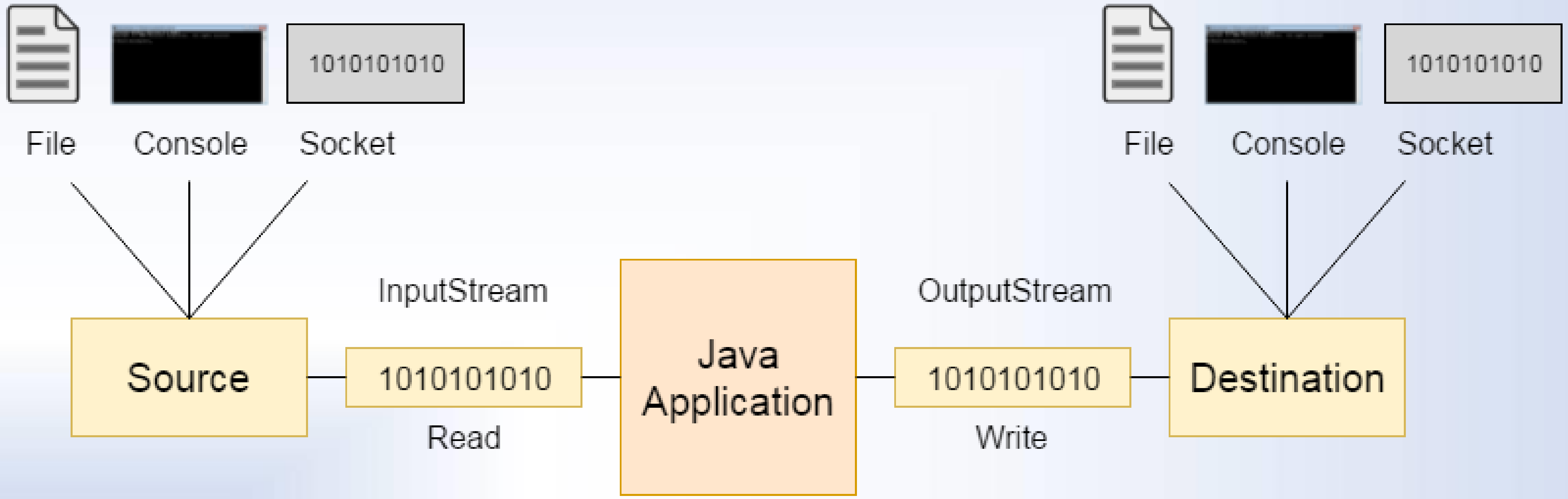
InputStream

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

- InputStream and OutputStream classes are operated on bytes for reading and writing respectively
- Classes Reader and Writer are operated on characters for reading and writing respectively.

Stream based I/O (java.io) in Java

- **Stream based I/O** (java.io)



Stream based I/O (java.io) in Java

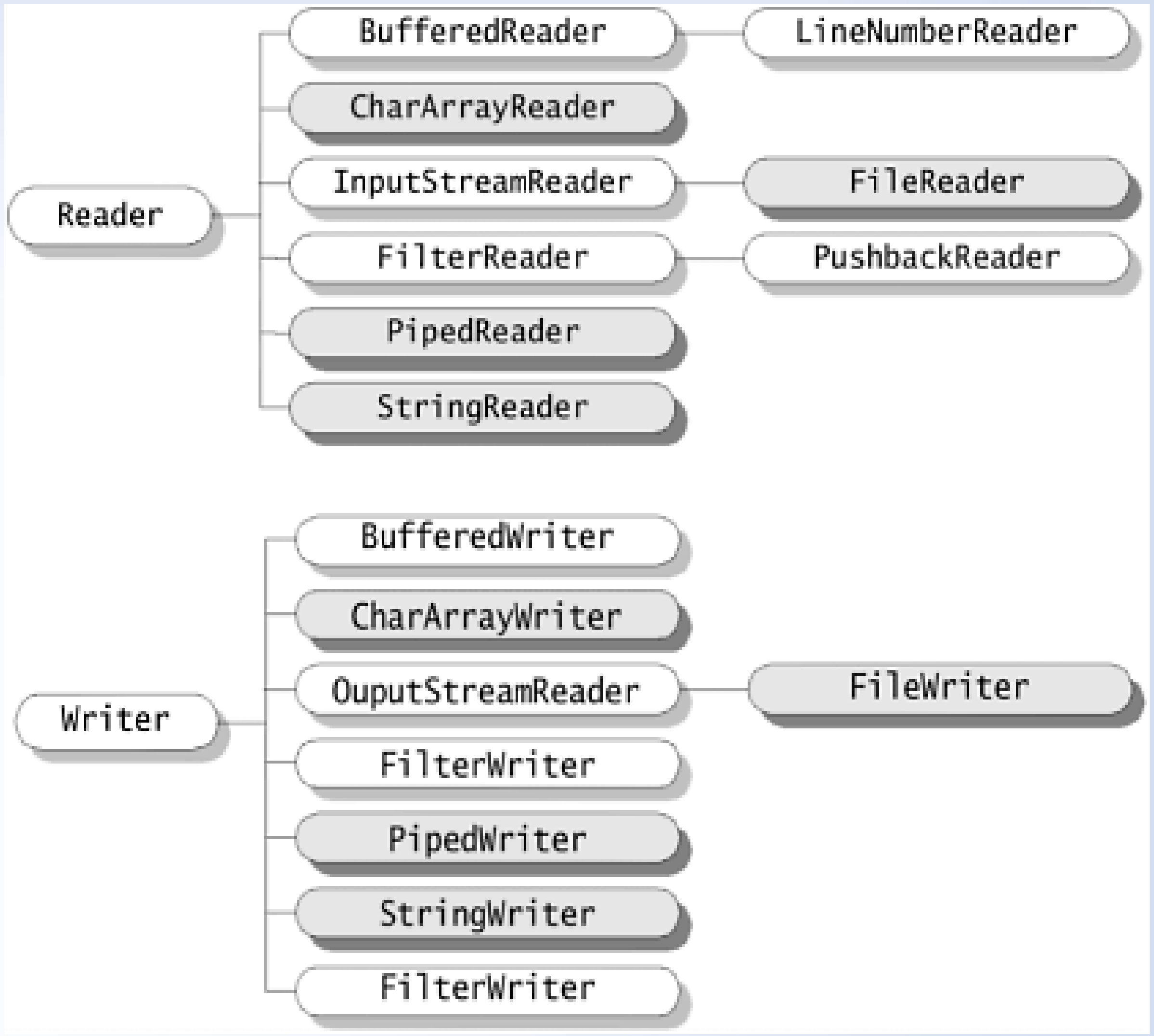
- **Stream based I/O** (java.io)

Character Streams

Reader and Writer are abstract super-classes For streaming 16-bit character input and outputs, respectively.

Methods of these classes throw the IOException exception under error Condition

All the methods in Writer class have return Type void



Stream based I/O (java.io) in Java

- **Stream based I/O** (java.io)

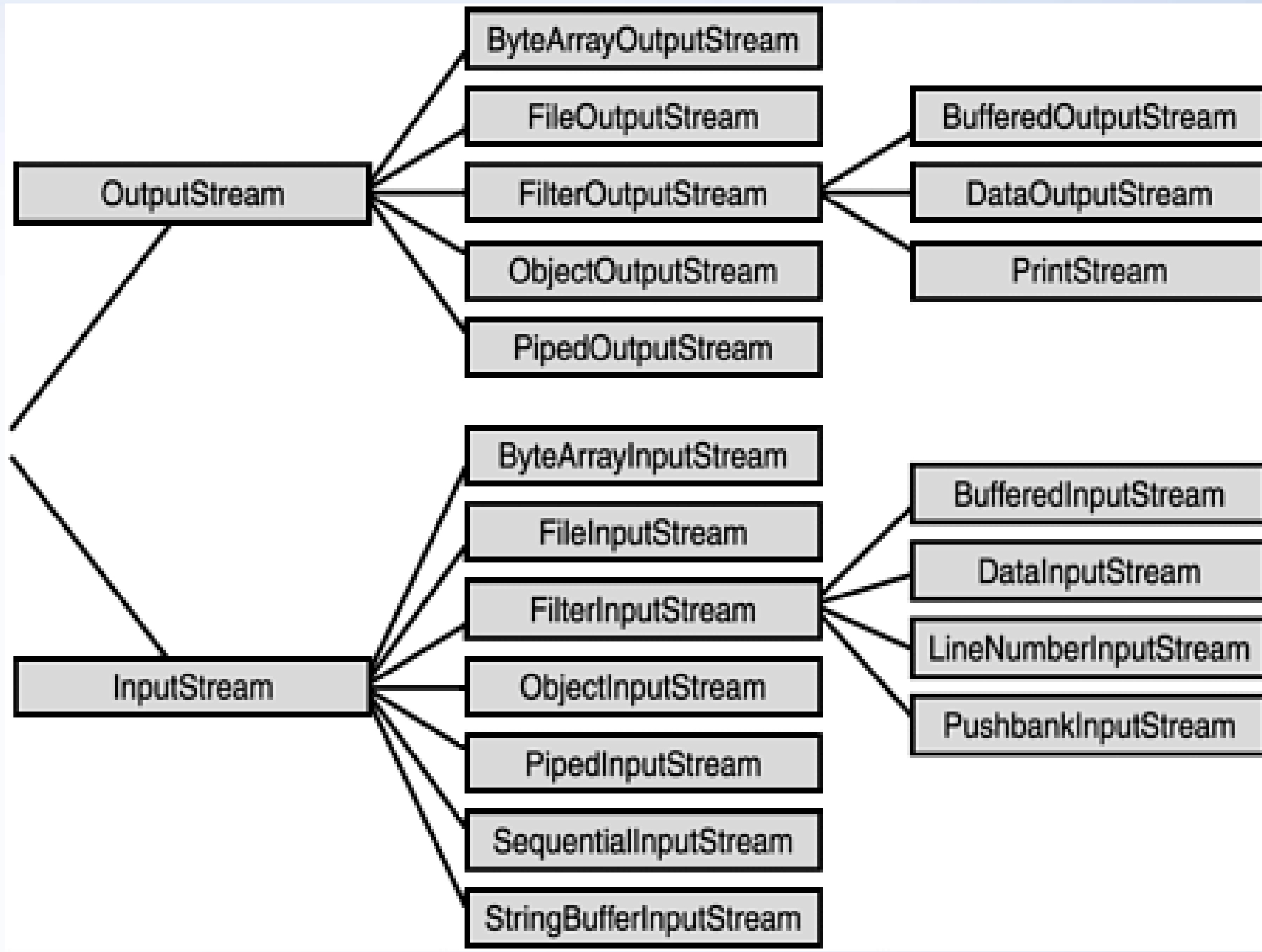
Byte Streams

Byte streams are used in program to read and write 8-bit bytes.

InputStream and OutputStream are abstract super-classes of all byte streams that have sequential in nature.

InputStream and OutputStream provides Application Program Interface (API) and partial implementation for input stream and Output streams.

InputStream and OutputStream are inherited From Object Class.



Stream based I/O (java.io) in Java

- **Stream based I/O** (java.io)

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.

Stream based I/O (java.io) in Java

- **Stream based I/O** (java.io)

OutputStream class

Useful methods of OutputStream

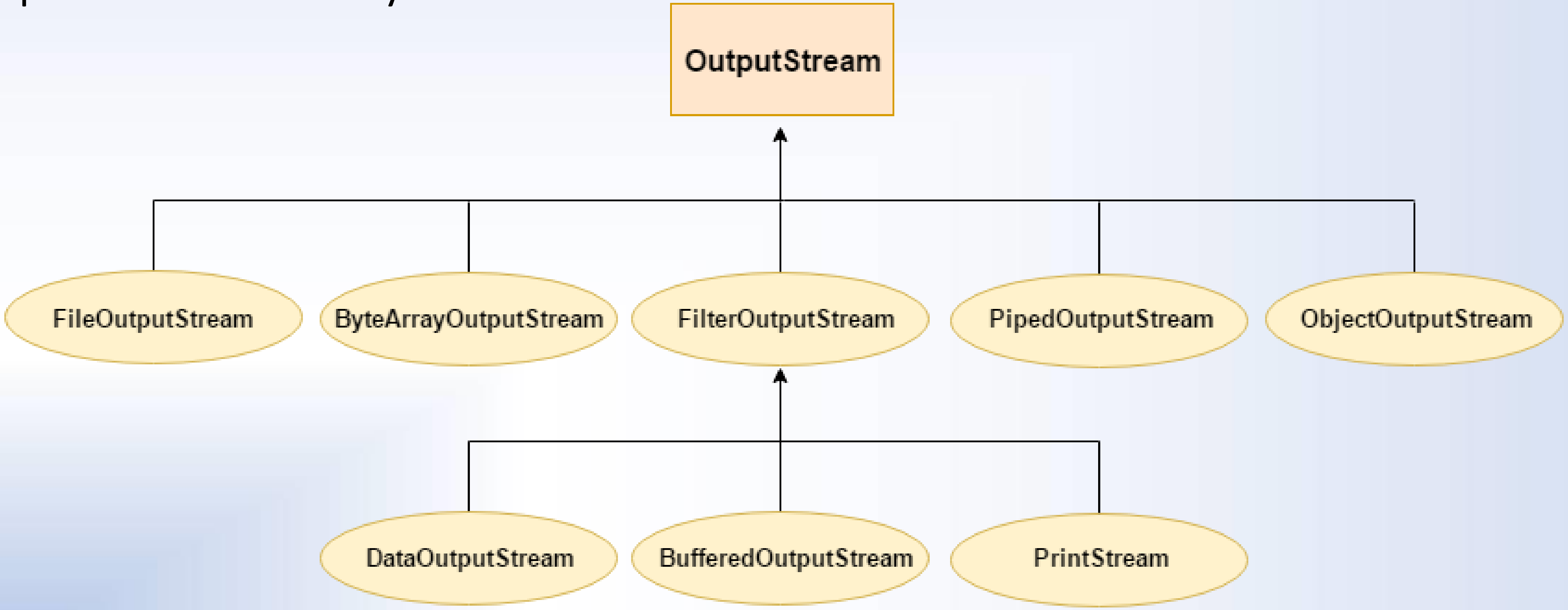
Method	Description
1) public void write(int)throws IOException	is used to write a byte to the current output stream.
2) public void write(byte[])throws IOException	is used to write an array of byte to the current output stream.
3) public void flush()throws IOException	flushes the current output stream.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

Stream based I/O (java.io) in Java

- **Stream based I/O** (java.io)

OutputStream class

OutputStream Hierarchy



Stream based I/O (java.io) in Java

- **Stream based I/O** (java.io)

InputStream class

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

Stream based I/O (java.io) in Java

- **Stream based I/O** (java.io)

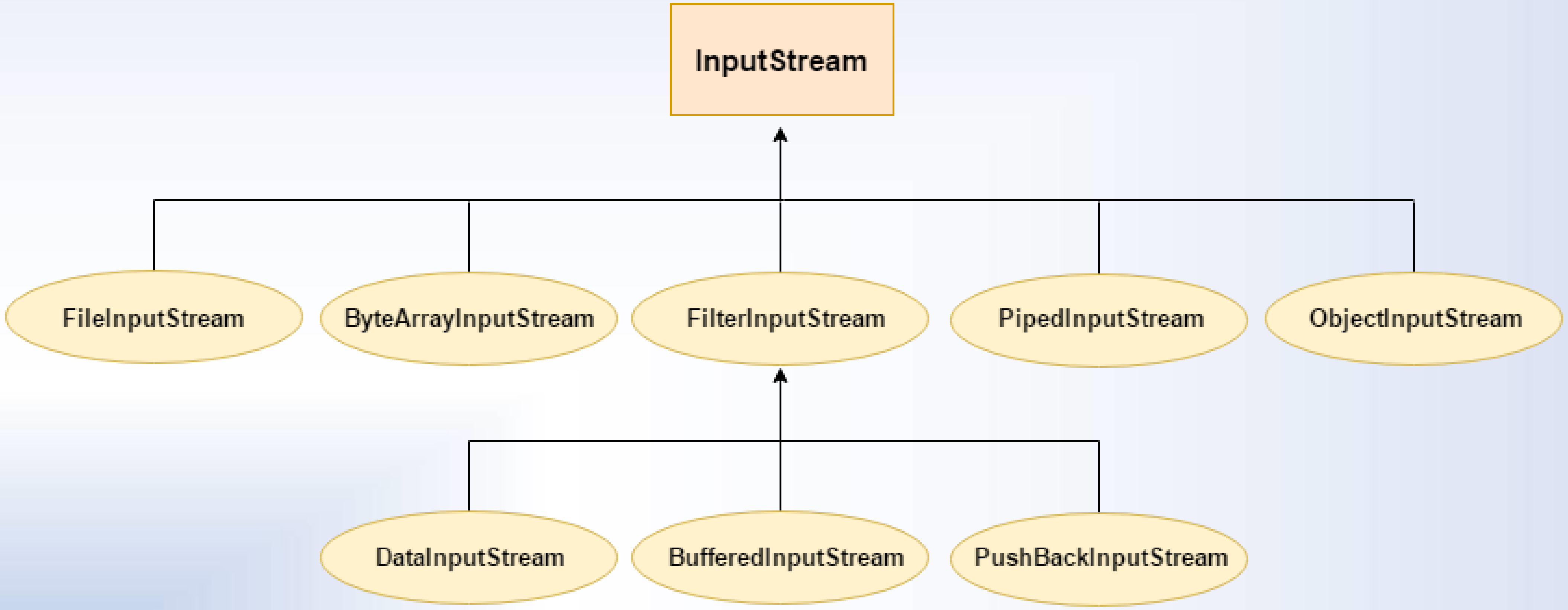
Useful methods of InputStream

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.

Stream based I/O (java.io) in Java

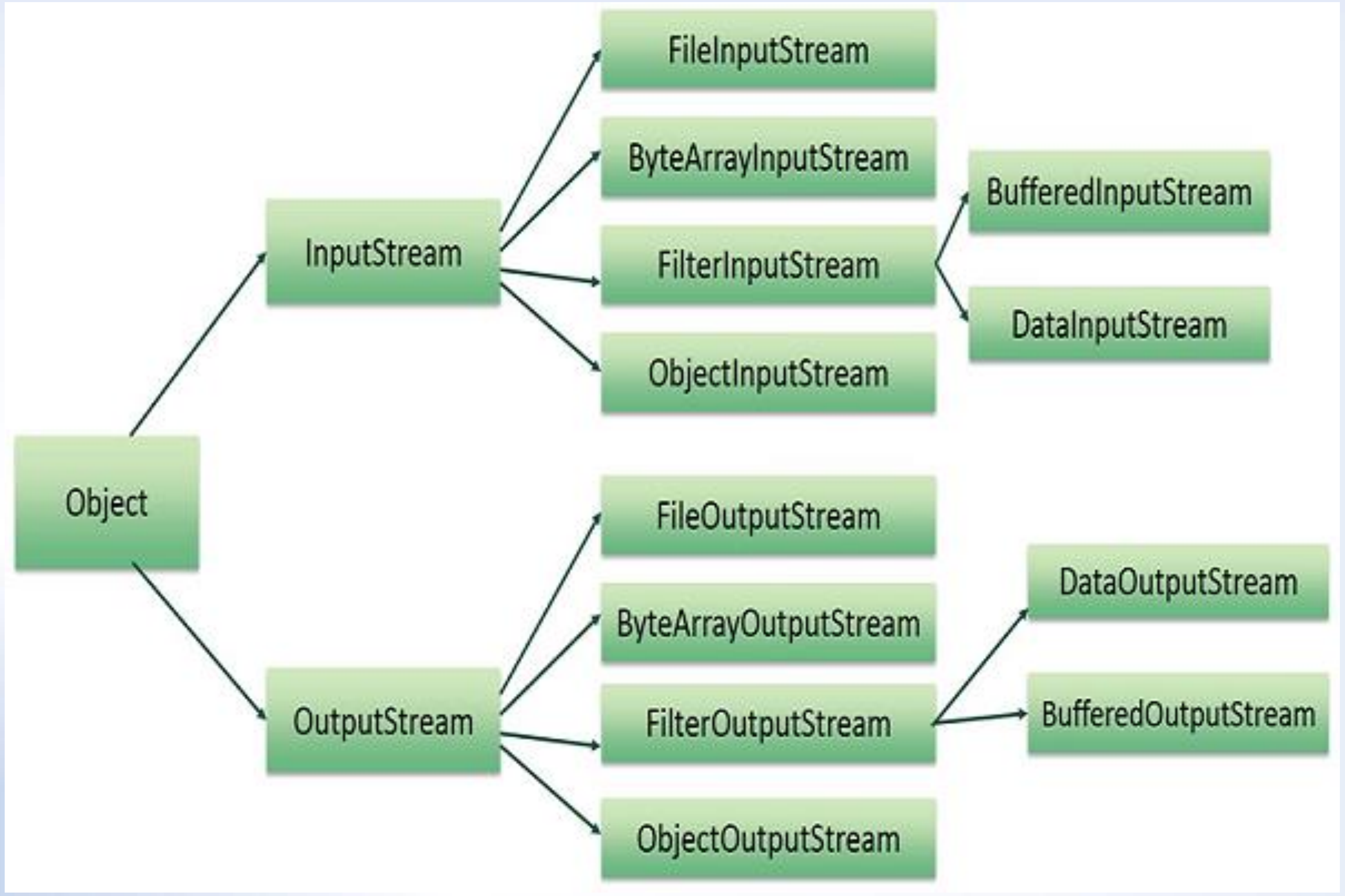
- **Stream based I/O** (java.io)

InputStream Hierarchy



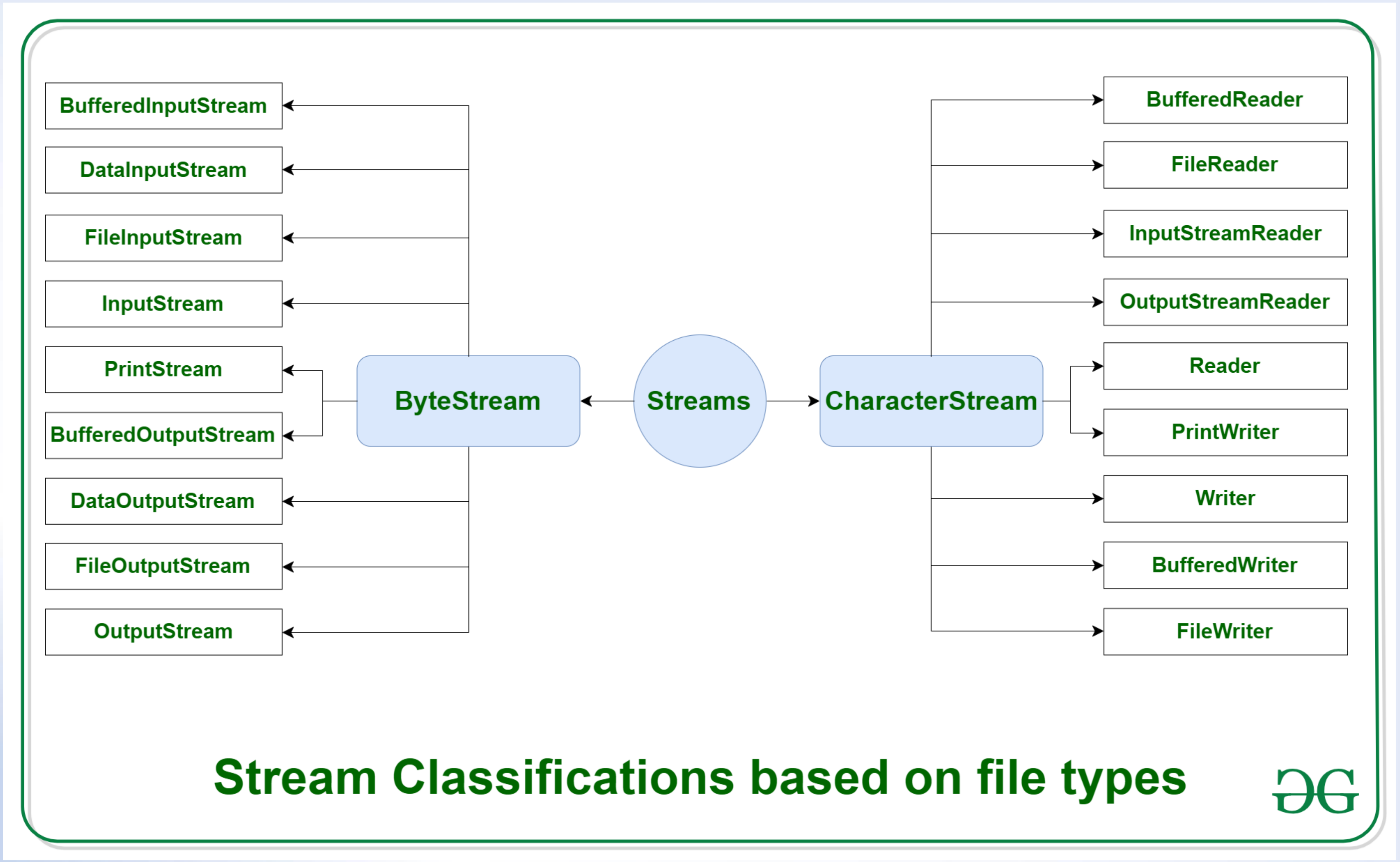
Stream based I/O (java.io) in Java

- **Stream based I/O** (java.io)



Stream based I/O (java.io) in Java

- **Stream based I/O** (java.io)



Stream based I/O (java.io) in Java

- **Stream based I/O** (java.io)
- Reading console Input and Writing Console Output

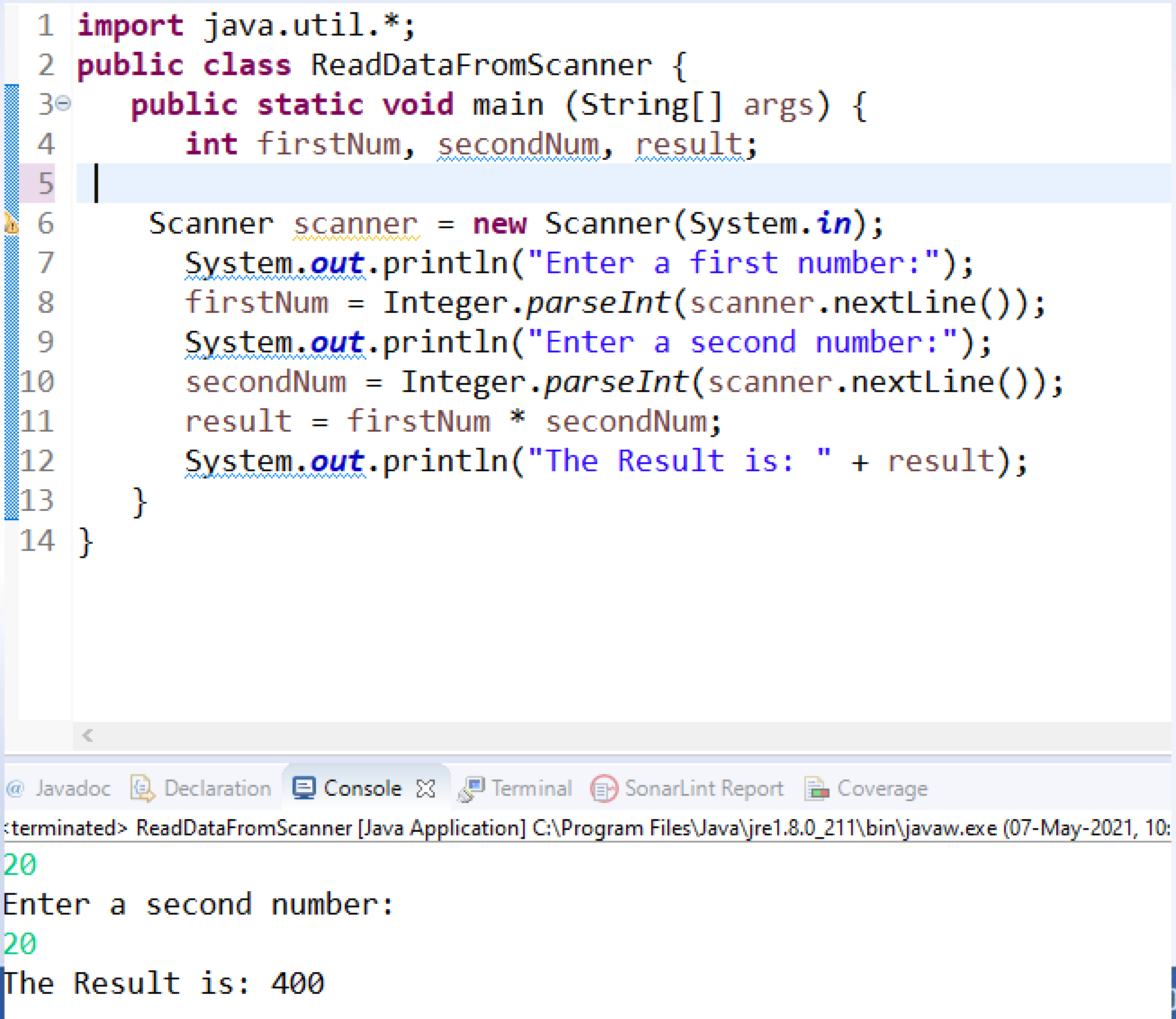
Stream based I/O (java.io) in Java

- Reading console Input and Writing Console Output Using BufferedReader

```
import java.io.*;
public class ReadDataFromInput {
  public static void main (String[] args) {
            int firstNum, secondNum, result;
            BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
            try {
                      System.out.println("Enter a first number:");
                      firstNum = Integer.parseInt(br.readLine());
                      System.out.println("Enter a second number:");
                      secondNum = Integer.parseInt(br.readLine());
                      result = firstNum * secondNum;
                      System.out.println("The Result is: " + result);
                    } catch (IOException ioe) {
                      System.out.println(ioe);
                    }

                            float f;
                            String str;
                            try {
                        f = Float.parseFloat(br.readLine());
                        System.out.println(f);
                        str = br.readLine();
                        System.out.println(str);
                        } catch (NumberFormatException | IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                        }
  }
}
```

Stream based I/O (java.io) in Java

- Reading console Input and Writing Console Output Using Scanner Class



```java
import java.util.*;
public class ReadDataFromScanner {
    public static void main (String[] args) {
        int firstNum, secondNum, result;

        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter a first number:");
        firstNum = Integer.parseInt(scanner.nextLine());
        System.out.println("Enter a second number:");
        secondNum = Integer.parseInt(scanner.nextLine());
        result = firstNum * secondNum;
        System.out.println("The Result is: " + result);
    }
}
```

```
<terminated> ReadDataFromScanner [Java Application] C:\Program Files\Java\jre1.8.0_211\bin\javaw.exe (07-May-2021, 10:
20
Enter a second number:
20
The Result is: 400
```

Stream based I/O (java.io) in Java

- Java FileInputStream Example : Reading a Single Character from file

```
import java.io.FileInputStream;
public class FileInputStreamDemo {
    public static void main(String args[]){
        try{
            FileInputStream fin=new FileInputStream("I:\\Subjects\\A.Y 2020-21\\Sem-2\\Java Practical
                                                    Programs\\fileout.txt");
            int i=fin.read();
            System.out.print((char)i);

            fin.close();
        }catch(Exception e){System.out.println(e);}
       }
      }
```

Output: A

Stream based I/O (java.io) in Java

Java FileInputStream example 2: read all characters

```
import java.io.FileInputStream;
public class FileInputStreamDemo1 {
    public static void main(String args[]){
        try{
        FileInputStream fin=new FileInputStream("I:\\Subjects\\A.Y 2020-21\\Sem-2\\Java Practical
Programs\\fileout1.txt");
          int i=0;
          while((i=fin.read())!=-1){
           System.out.print((char)i);
          }
          fin.close();
        }catch(Exception e){System.out.println(e);}
       }
      }
```

Output: Anil Kumar

Stream based I/O (java.io) in Java

Java FileOutputStream Example 1: write byte

```
import java.io.FileOutputStream;
public class FileOutputStreamDemo {
    public static void main(String args[]){
          try{
            FileOutputStream fout=new FileOutputStream("I:\\\\Subjects\\\\A.Y 2020-21\\\\Sem-2\\\\Java
Practical Programs\\\\fileout2.txt");
            fout.write(65);
            fout.close();
            System.out.println("success...");
           }catch(Exception e){System.out.println(e);}
      }
}
```

Output: A

Stream based I/O (java.io) in Java

Java FileOutputStream example 2: write string

```
import java.io.FileOutputStream;
public class FileOutputStreamDemo1 {
    public static void main(String args[]){
          try{
            FileOutputStream fout=new FileOutputStream("I:\\Subjects\\A.Y 2020-21\\Sem-2\\Java Practical
Programs\\fileout3.txt");
            String s="Welcome to Wakeel.";
            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);}
      }
} Output: Welcome to Wakeel.
```

Stream based I/O (java.io) in Java

File class

```
import java.io.*;
public class FileClassDemo {
    public static void main(String[] args) {

        try {
            File file = new File("I:\\Subjects\\A.Y 2020-21\\Sem-2\\Java Practical Programs\\fileout4.txt");
            if (file.createNewFile()) {
                System.out.println("New File is created!");
            } else {
                System.out.println("File already exists.");
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

    }
}
```

Stream based I/O (java.io) in Java

Random access file operations

- This class is used for reading and writing to random access file.
- A random access file behaves like a large array of bytes.
- There is a cursor implied to the array called file pointer, by moving the cursor we do the read write operations. If end-of-file is reached before the desired number of byte has been read than EOFException is thrown.
- It is a type of IOException.

Stream based I/O (java.io) in Java

Random access file operations

```
import java.io.IOException;
import java.io.RandomAccessFile;

public class RandomAccessFileExample {
    static final String FILEPATH ="I:\\Subjects\\A.Y 2020-21\\Sem-2\\Java Practical
Programs\\RandomAccessFile.txt";
    public static void main(String[] args) {
        try {
            System.out.println(new String(readFromFile(FILEPATH, 0, 18)));
            writeToFile(FILEPATH, "I love my country and my people", 31);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

Stream based I/O (java.io) in Java

```
private static byte[] readFromFile(String filePath, int position, int size)
            throws IOException {
        RandomAccessFile file = new RandomAccessFile(filePath, "r");
        file.seek(position);
        byte[] bytes = new byte[size];
        file.read(bytes);
        file.close();
        return bytes;
    }
    private static void writeToFile(String filePath, String data, int position)
            throws IOException {
        RandomAccessFile file = new RandomAccessFile(filePath, "rw");
        file.seek(position);
        file.write(data.getBytes());
        file.close();
    }
}
```

Stream based I/O (java.io) in Java

The Console class

```
import java.io.Console;
import java.io.IOException;
public class IODemo {
			public static void main(String[] args) throws IOException {

						Console console = System.console();
						if(console == null) {
						    System.out.println("Console is not available to current JVM process");
						    return;
						}
						String str = console.readLine("Enter name");
						System.out.println("Str is "+str);

//						Console console = System.console();

						if(console == null) {
						    System.out.println("Console is not available to current JVM process");
						    return;
						}
						char[] password = console.readPassword("Enter the password: ");
						System.out.println("Entered password: " + new String(password));

			}
			}
```

Serialization

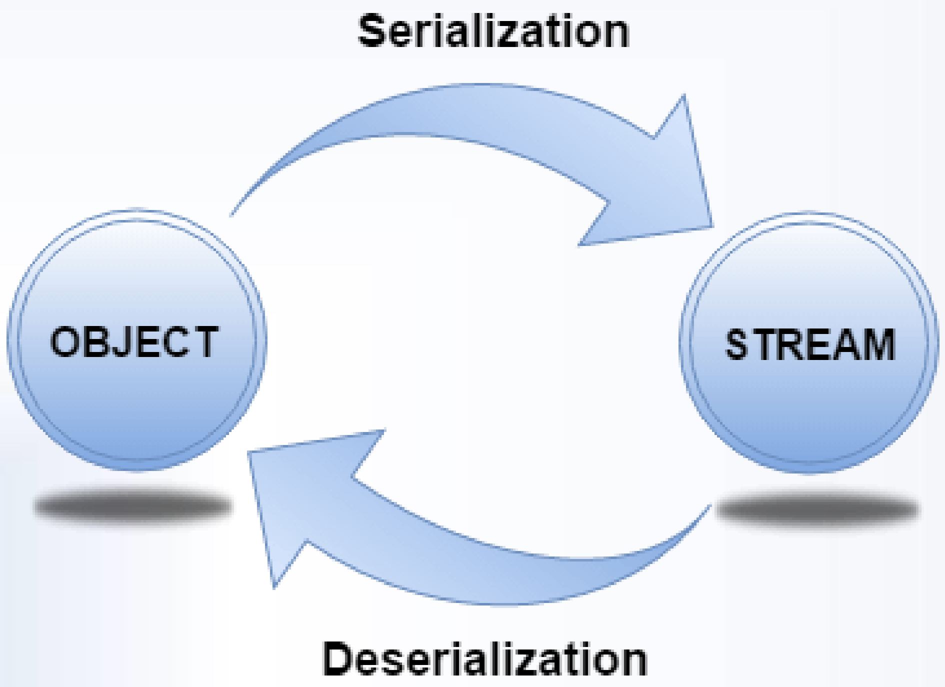
- **Stream based I/O** (java.io)

- **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* where byte-stream is converted into an object.
- The serialization and deserialization process is platform-independent, it means you can serialize an object in a platform and de-serialize in 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.

Serialization

Advantages of Java Serialization

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



Serialization

java.io.Serializable interface

- Serializable is a marker interface (has no data member and method).
- It is used to "mark" Java classes so that the objects of these classes may get a certain capability.
- The Cloneable and Remote are also marker interfaces.
- 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.

Serialization

```
java.io.FileInputStream;
java.io.FileOutputStream;
java.io.IOException;
java.io.ObjectInputStream;
java.io.ObjectOutputStream;
java.io.Serializable;

umbers implements Serializable {

 i ;
 j ;
mbers(int i, int j){
        this.i =i;
        this.j=j;
```

Serialization

```
public class SerializableDemo {

	public static void main(String[] args) throws IOException {
			Numbers n1 = new Numbers(10,20);

			FileOutputStream fos = new FileOutputStream("C:/Users/Anil/Desktop/fos.txt");

			ObjectOutputStream ous = new ObjectOutputStream(fos);
			ous.writeObject(n1.i);
			ous.close();

			FileInputStream fis = new FileInputStream("C:/Users/Anil/Desktop/fos.txt");
			ObjectInputStream ois = new ObjectInputStream(fis);
			Numbers n2 = new Numbers(30, 20);
			System.out.println(n1.i + " hello " + n2.j);
			ois.close();
	}

}
```

Java Enumerations

- Enumerations was added to Java language in JDK5.
- **Enumeration** means a list of named constant.
- In Java, enumeration defines a class type.
- An Enumeration can have constructors, methods and instance variables.
- It is created using **enum** keyword.
- Each enumeration constant is *public*, *static* and *final* by default.
- Even though enumeration defines a class type and have constructors, you do not instantiate an **enum** using **new**.
- Enumeration variables are used and declared in much a same way as you do a primitive variable.

Java Enumerations

- Define and Use an Enumeration

- An enumeration can be defined simply by creating a list of enum variable.
- Let us take an example for list of Subject variable, with different subjects in the list.

```
//Enumeration defined
enum Subject
{
Java, Cpp, C, Dbms
}
```

Java Enumerations

- Define and Use an Enumeration

1. Identifiers Java, Cpp, C and Dbms are called enumeration constants.
   These are public, static and final by default.
2. Variables of Enumeration can be defined directly without any new keyword.
   ```
   Subject sub;
   ```
3. Variables of Enumeration type can have only enumeration constants as value.
   We define an enum variable as
   ```
   enum_variable = enum_type.enum_constant;
    sub = Subject.Java;
   ```

4. Two enumeration constants can be compared for equality by using the = = relational operator.

```
if(sub == Subject.Java) {
   ...
}
```

Java Enumerations

Example of Enumeration

```
enum WeekDays
{
sun, mon, tues, wed, thurs, fri, sat
}

class EnumDemo
{
 public static void main(String args[])
 {
         WeekDays wk; //wk is an enumeration variable of type WeekDays
         wk = WeekDays.sun; //wk can be assigned only the constants defined under enum type Weekdays
         System.out.println("Today is "+wk);
 }
}
```

Java Enumerations

Points to remember about Enumerations

- Enumerations are of class type, and have all the capabilities that a Java class has.
- Enumerations can have Constructors, instance Variables, methods and can even implement Interfaces.
- Enumerations are not instantiated using **new** keyword.
- All Enumerations by default inherit **java.lang.Enum** class.

Auto boxing s

Autoboxing and Unboxing:

- The automatic conversion of primitive data types into its equivalent Wrapper type is known as boxing and opposite operation is known as unboxing.
- This is the new feature of Java5.
- So java programmer doesn't need to write the conversion code.
- int i ;
- Integer a =new Integer(i);

Auto <u>boxing s</u>

<u>Advantage of Autoboxing and Unboxing:</u>

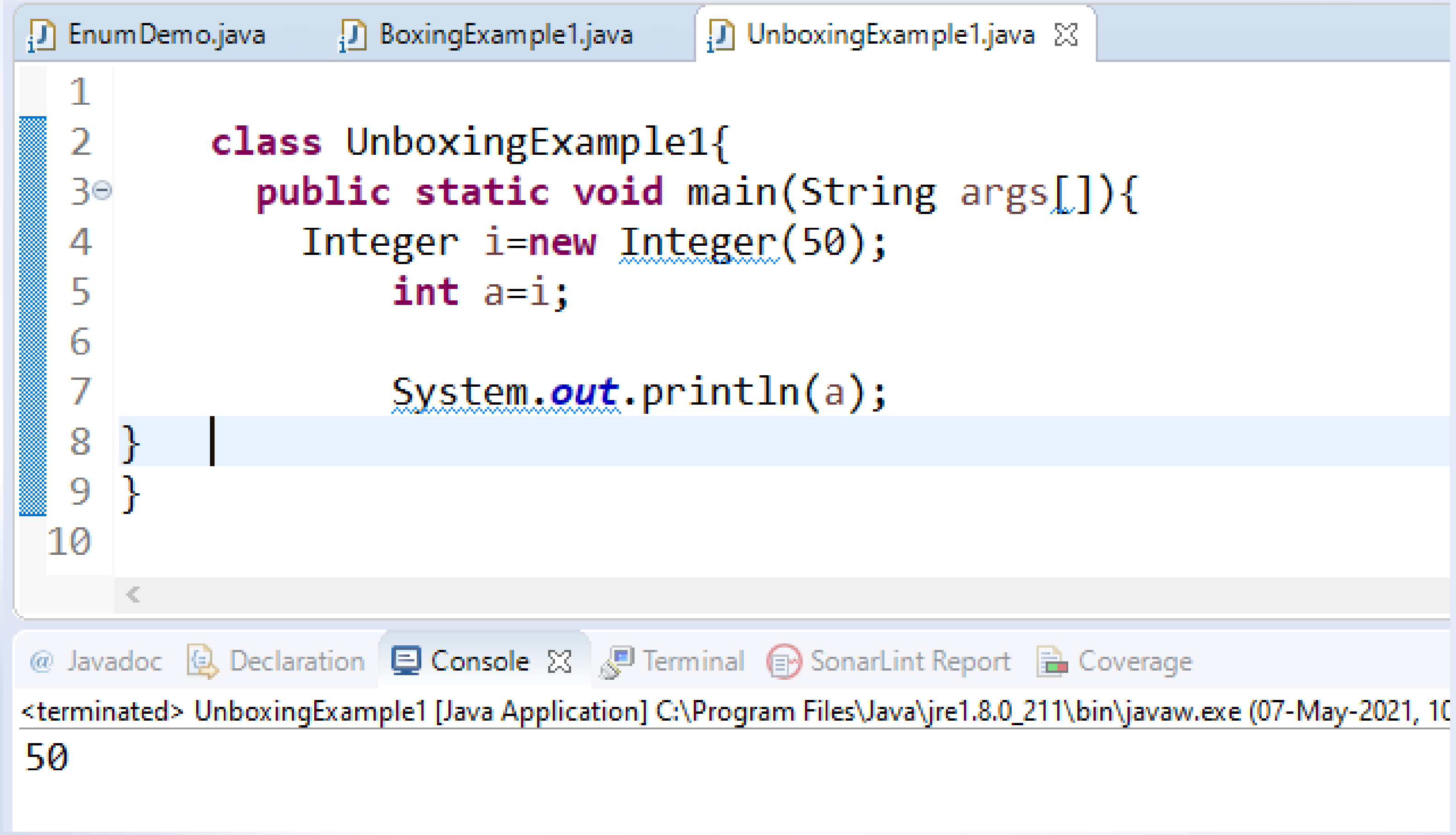
No need of conversion between primitives and Wrappers manually so less coding is required.

Auto boxing s

```
class BoxingExample1{
 public static void main(String args[]){
  int a=50;
      Integer a2=new Integer(a);//Boxing

      Integer a3=5;//Boxing

      System.out.println(a2+" "+a3);
 }
}
```

Auto boxing
s

The automatic conversion of wrapper class type into corresponding primitive type, is known as Unboxing.



```java

    class UnboxingExample1{
      public static void main(String args[]){
        Integer i=new Integer(50);
            int a=i;

            System.out.println(a);
}
}

```

```
<terminated> UnboxingExample1 [Java Application] C:\Program Files\Java\jre1.8.0_211\bin\javaw.exe (07-May-2021, 10
50
```

Generics in Java

- The **Java Generics** programming is introduced in J2SE 5 to deal with type-safe objects.
- It makes the code stable by detecting the bugs at compile time.
- Before generics, we can store any type of objects in the collection, i.e., non-generic.
- Now generics force the java programmer to store a specific type of objects.

Generics in Java

There are mainly 3 advantages of generics. They are as follows:

1) **Type-safety:**
2) **Type casting is not required:**
3) **Compile-Time Checking**

Generics in Java

1) Type-safety: We can hold only a single type of objects in generics. It doesn?t allow to store other objects.

Without Generics, we can store any type of objects.

```
List list = new ArrayList();
list.add(10);
list.add("10");
```

With Generics, it is required to specify the type of object we need to store.

```
List<Integer> list = new ArrayList<Integer>();
list.add(10);
list.add("10");// compile-time error
```

Generics in Java

2) Type casting is not required: There is no need to typecast the object. Before Generics, we need to type cast.

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);//typecasting
```

After Generics, we don't need to typecast the object.

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);
```

Generics in Java

3) Compile-Time Checking: It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

```
List<String> list = new ArrayList<String>();
list.add("hello");
list.add(32);//Compile Time Error
```

Generics in Java

Syntax to use generic collection

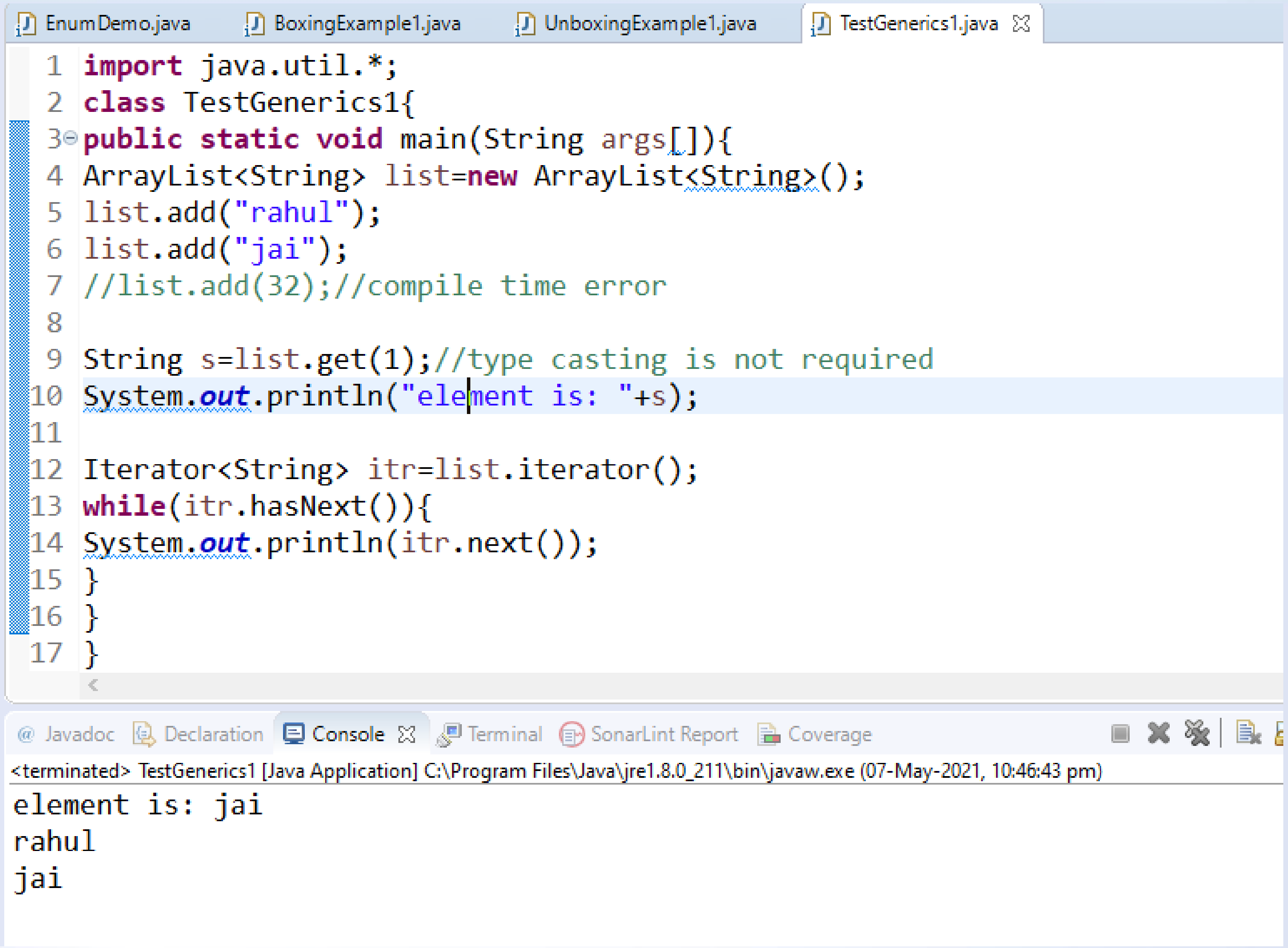
ClassOrInterface<Type>

Example to use Generics in java

ArrayList<String>

Generics in Java

Example of Generic



```java
import java.util.*;
class TestGenerics1{
public static void main(String args[]){
ArrayList<String> list=new ArrayList<String>();
list.add("rahul");
list.add("jai");
//list.add(32);//compile time error

String s=list.get(1);//type casting is not required
System.out.println("element is: "+s);

Iterator<String> itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

```
<terminated> TestGenerics1 [Java Application] C:\Program Files\Java\jre1.8.0_211\bin\javaw.exe (07-May-2021, 10:46:43 pm)
element is: jai
rahul
jai
```