UNIT-9 Binary I/O ,Recursion and Generics

(Chapter-17) Text I/O, binary I/O, Binary I/O classes, Object I/o, Random Access files, **(Chapter-18)** Problem solving using Recursion, Recursive Helper methods, Tail Recursion, **(Chapter-19)** Defining Generic classes and interfaces, Generic methods, Raw types and backward compatibility, wildcard Generic types, Erasure and Restrictions on Generics.

- Java provides many classes for performing text I/O and binary 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.
- In general, these can be classified as input classes and output classes.
- An input class contains the methods to read data, and an output class contains the methods to write data.
- The **java.io** package contains all the classes required for input and output operations.

File I/O [Covered in Unit-6]

In File IO , We have already seen that,

- **PrintWriter** is an example of an output class, and **Scanner** is an example of an input class.
- Text data are read using the **Scanner** class and written using the **PrintWriter** class.

The following code creates an input object for the file temp.txt and **reads** data from the file.

```
Scanner input = new Scanner(new File("temp.txt"));
System.out.println(input.nextLine());
```

You can create an object using the **PrintWriter** class as follows:

```
PrintWriter output = new PrintWriter("temp.txt");
```

You can now invoke the print method on the object to write a string to the file.

```
output.print("Java 101");
```

The next statement closes the file.

```
output.close();
```

9.1 Java I/O Streams [Text I/O, binary I/O, Binary I/O classes, Object I/o]

In Java, streams are the sequence of data that are read from the source and written to the destination.

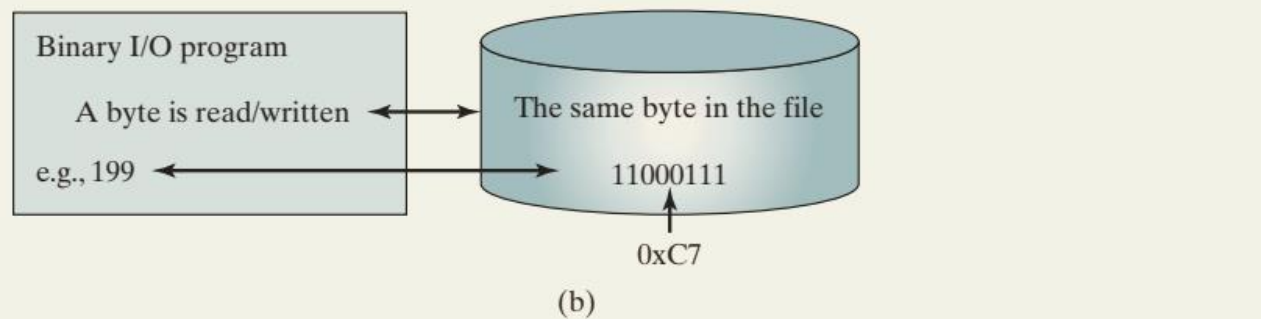
Types of Streams

Depending upon the data a stream holds, it can be classified into:

- Byte Stream (Binary IO)
- Character Stream (Text IO)

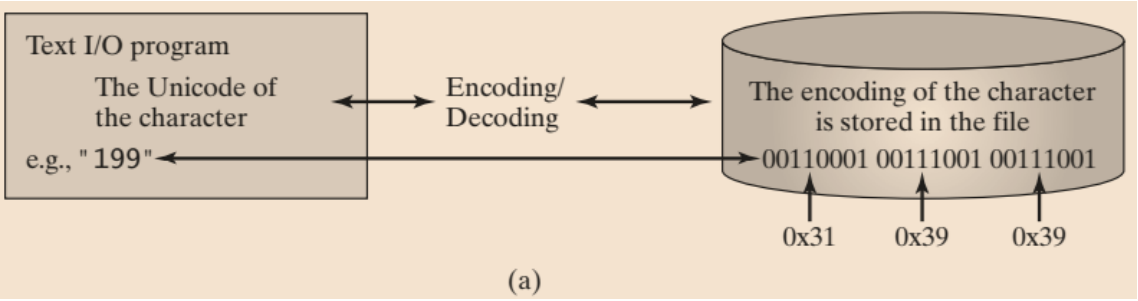
Byte Stream

- Byte stream is used to read and write a single byte (8 bits) of data.
- All byte stream classes are derived from base abstract classes called InputStream and OutputStream.

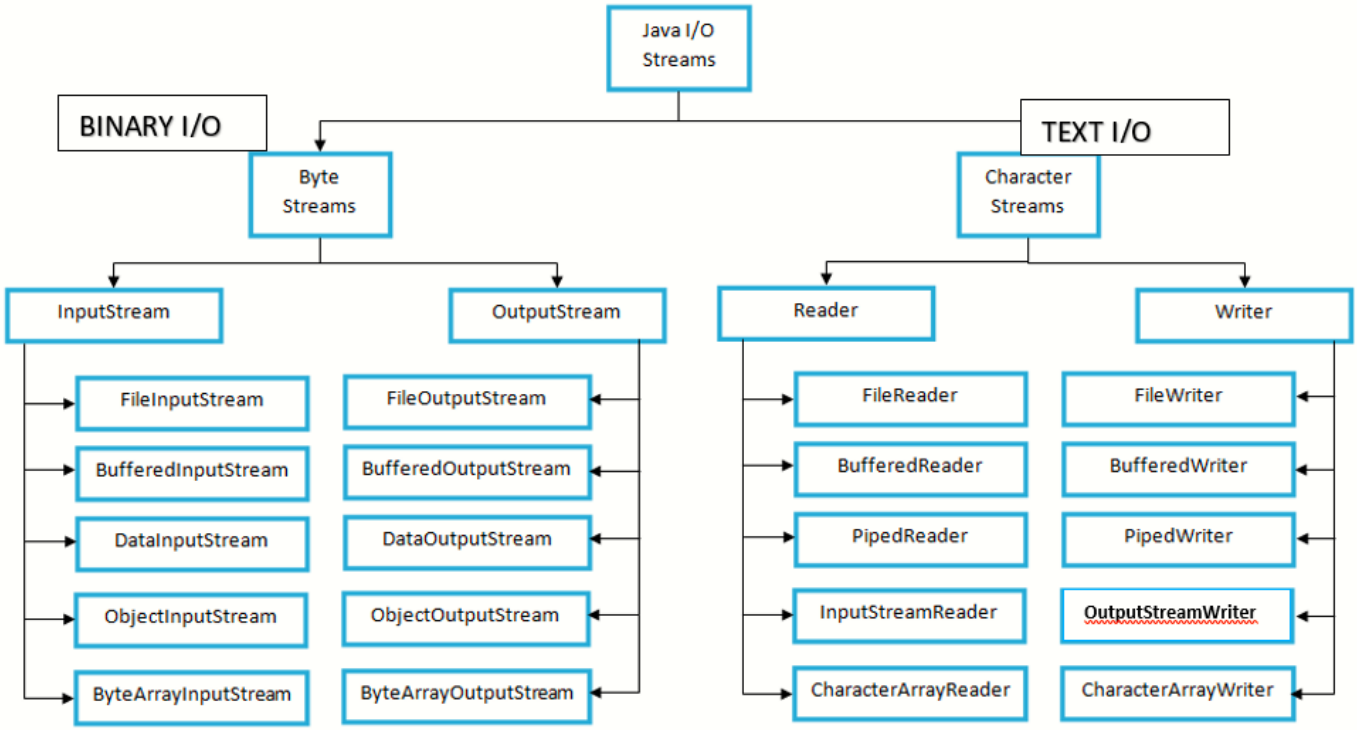


Character Stream

- Character stream is used to read and write a single character of data.
- All the character stream classes are derived from base abstract classes Reader and Writer.



Java I/O Streams Hierarchy



Character Stream	Byte Stream
A character stream will access a file character by character.	A byte stream access the file byte by byte.
These handle data in 16 bit Unicode. The character stream classes read/write 16-bit characters.	These handle data in bytes (8 bits) i.e., the byte stream classes read/write data of 8 bits.
Using these you can read and write text data only.	Using these you can store characters, videos, audios, images etc.
A character stream needs to be given the file's encoding in order to work properly.	byte stream do not use any encoding.

Byte Streams (Binary I/O)

- Java byte streams are used to perform input and output of 8-bit bytes.
- The most frequently used classes are, **FileInputStream** and **FileOutputStream**.
- The java byte stream is defined by two abstract classes, **InputStream** and **OutputStream**.
- The InputStream class used for byte stream based input operations
- The OutputStream class used for byte stream based output operations.
- All the methods in the binary I/O classes are declared to throw **java.io.IOException** or a subclass of java.io.IOException.

InputStream class

The InputStream class has defined as an abstract class, and it has the following methods which have implemented by its concrete classes.

S.No.	Method with Description
1	**int available():** It returns the number of bytes that can be read from the input stream.
2	**int read():** It reads the next byte from the input stream.
3	**int read(byte[] b):** It reads a chunk of bytes from the input stream and store them in its byte array, b.
4	**void close():** It closes the input stream and also frees any resources connected with this input stream.

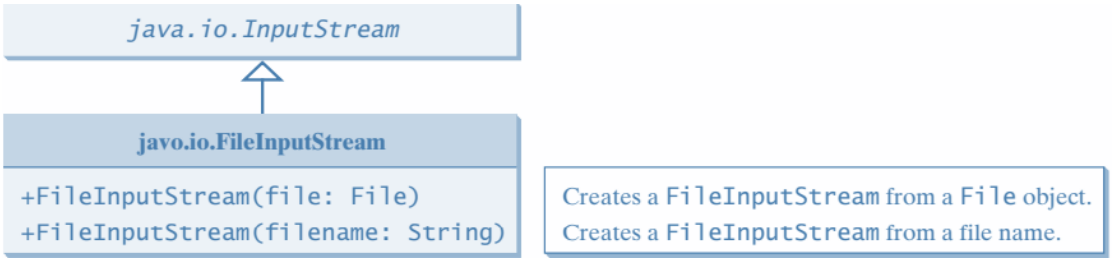
OutputStream class

The OutputStream class has defined as an abstract class, and it has the following methods which have implemented by its concrete classes.

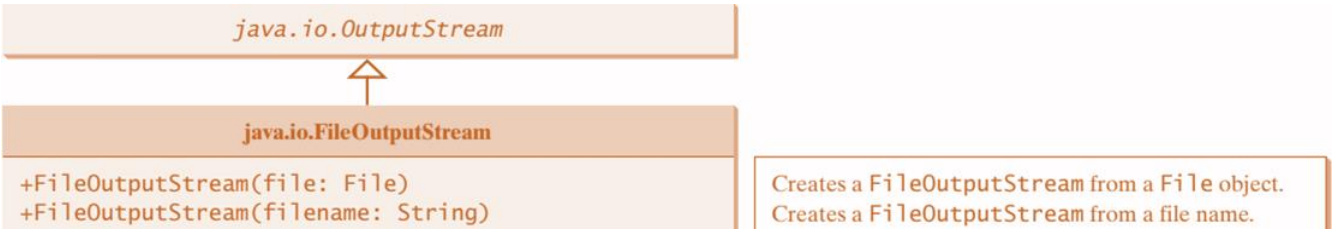
S.No.	Method with Description
1	**void write(int n):** It writes byte(contained in an int) to the output stream.
2	**void write(byte[] b):** It writes a whole byte array(b) to the output stream.
3	**void flush():** It flushes the output steam by forcing out buffered bytes to be written out.
4	**void close():** It closes the output stream and also frees any resources connected with this output stream.

❖ FileInputStream/FileOutputStream

- FileInputStream/FileOutputStream is for reading/writing bytes from/to files.
- All the methods in these classes are inherited from InputStream and OutputStream.
- FileInputStream inputs a stream of bytes from a file



- FileOutputStream outputs a stream of bytes to a file.



- Almost all the methods in the I/O classes throw java.io.IOException. Therefore, you have to declare to throw java.io.IOException in the method or place the code in a trycatch block

Example: Demonstrate FileInputStream and FileOutputStream by copying content of one file to another.

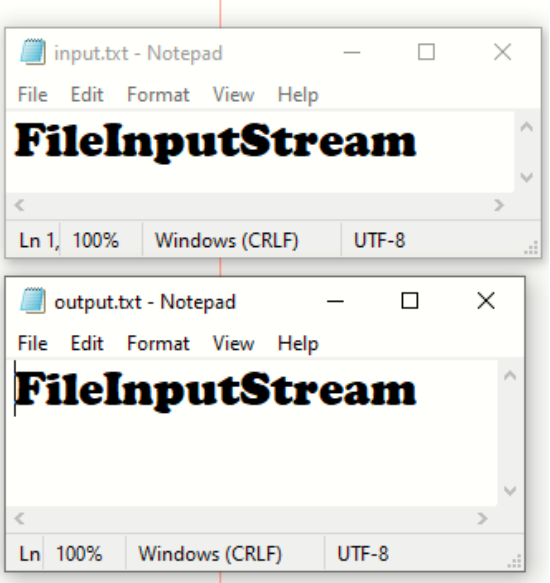
```
import java.io.*;
public class Main {

   public static void main(String args[]) throws IOException {
      FileInputStream in = null;
      FileOutputStream out = null;

      try {
         in = new FileInputStream("E:\\input.txt");
         out = new FileOutputStream("E:\\output.txt");

         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      }finally {
         if (in != null) {
            in.close();
         }
         if (out != null) {
            out.close();
         }

      }
   }
}
```



When a stream is no longer needed, always close it using the close() method or automatically close it using a try-with-resource statement.
Not closing streams may cause data corruption in the output file, or other programming errors.

Example: Using binary I/O to write ten byte values from 1 to 10 to a file named temp.dat (binary file) and read them back from the file.

```java
import java.io.*;

public class TestFileStream {
public static void main(String[] args) throws IOException {
try (
// Create an output stream to the file
FileOutputStream output = new FileOutputStream("temp.dat");
) {
// Output values to the file
for (int i = 1; i <= 10; i++)
output.write(i);
}

try (
// Create an input stream for the file
FileInputStream input = new FileInputStream("temp.dat");
) {
// Read values from the file
int value;
while ((value = input.read()) != -1)
System.out.print(value + " ");
}
}
}
```

```
1 2 3 4 5 6 7 8 9 10
```

❖ FilterInputStream/FilterOutputStream

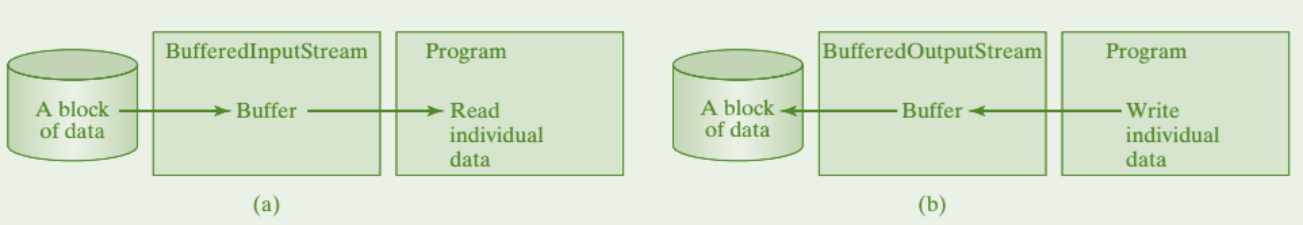
- FilterInputStream and FilterOutputStream are the base classes for filtering data.
- When you need to process primitive numeric types, use DataInputStream and DataOutputStream to filter bytes.

❖ DataInputStream/DataOutputStream

- DataInputStream reads bytes from the stream and converts them into appropriate primitive-type values or strings.
- DataOutputStream converts primitive-type values or strings into bytes and outputs the bytes to the stream.
- You have to read data in the same order and format in which they are stored.
- For example, since names are written in UTF-8 using writeUTF, you must read names using readUTF.
- If you keep reading data at the end of an InputStream, an **EOFException** will occur. This exception can be used to detect the end of a file

❖ BufferedInputStream/BufferedOutputStream

- BufferedInputStream/BufferedOutputStream can be used to speed up input and output by reducing the number of disk reads and writes.
- Using BufferedInputStream, the whole block of data on the disk is read into the buffer in the memory once.
- Buffer I/O places data in a buffer for fast processing
- You can wrap a BufferedInputStream/BufferedOutputStream on any InputStream/OutputStream using the constructors shown in following figures.
- If no buffer size is specified, the default size is 512 bytes.



❖ **ObjectInputStream/ObjectOutputStream** classes can be used to read/write **serializable** objects.

- They enables you to perform I/O for objects in addition to primitive-type values and strings.
- Since ObjectInputStream/ ObjectOutputStream contains all the functions of DataInputStream/ DataOutputStream, you can replace DataInputStream/DataOutputStream completely with ObjectInputStream/ObjectOutputStream.
- ObjectInputStream can read objects, primitive-type values, and strings.
- ObjectOutputStream can write objects, primitive-type values, and strings.

Example: Using binary I/O to write byte values to a file named File.txt and read them back from the file.

We can Replace **DataOutputSream/DataInputStream** with BufferedInputStream/BufferedOutputStream **OR** ObjectInputStream/ObjectOutputStream in following code

```
1  package javaiostream;
2  import java.io.DataInputStream;
3  import java.io.DataOutputStream;
4  import java.io.File;
5  import java.io.FileInputStream;
6  import java.io.FileOutputStream;
7  import java.io.IOException;
8  import java.io.InputStream;
9  import java.io.OutputStream;
10 public class JavaIOStream {
11     public static void main(String[] args) throws IOException {
12         // Create New File in the same Directory
13         File f = new File("File.txt");
14         f.createNewFile();
15         // Write data to File.txt
16         OutputStream out = new FileOutputStream(f);
           DataOutputStream output = new DataOutputStream(out);
18         String s = "Hello";
19         byte[] ob = s.getBytes();
20         output.write(ob);
21         output.flush();
22         output.close();
23         // Read data from File.txt
24         InputStream in = new FileInputStream(f);
25         DataInputStream input = new DataInputStream(in);
26         byte[] ib = new byte[input.available()];
27         input.read(ib);
28         for (byte bt : ib) {
29             char c = (char) bt;
30             System.out.print(c);
31         }
32         System.out.println();
33     }
34 }
```

```
Output - JavaIOStream (run)
run:
Hello
BUILD SUCCESSFUL (total time: 0 seconds)
```

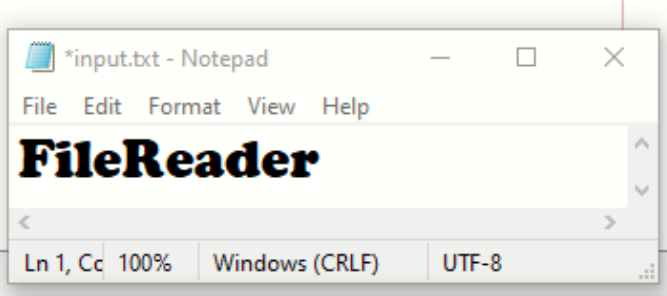
Character Streams (Text I/O)

- Java Character streams are used to perform input and output for 16-bit unicode.
- Though there are many classes related to character streams but the most frequently used classes are, FileReader and FileWriter.
- Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

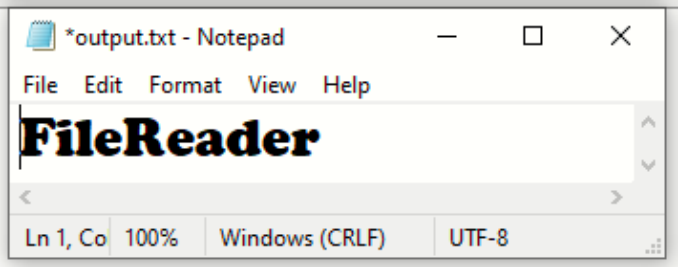
We can re-write the example of FileInputStream/ FileOutputStream by replacing them with FileReader/FileWriter,

Example: copy an input file (having unicode characters) into an output file

```java
package javaiostream;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
public class FileReaderDemo  {
    public static void main(String[] args) throws IOException {
        FileReader in = new FileReader("E:\\input.txt");
        FileWriter out = new FileWriter("E:\\output.txt");
        int c;
        while ((c = in.read()) != -1) {
            out.write(c);
        }
            in.close();
            out.close();
    }
}
```




```
Output - JavaIOStream (run)
run:
BUILD SUCCESSFUL (total time: 0 seconds)
```



Modify above program to use <u>BufferReader</u> as Follows:

```java
public class CopyContentEx {
    public static void main(String args[]) throws IOException {
      BufferedReader in = null;
      BufferedWriter out = null;

      try {
         in = new BufferedReader(new FileReader("E:\\input.txt"));
         out = new BufferedWriter(new FileWriter("E:\\output.txt"));
```

Modify above program to use InputStreamReader/InputStreamWriter as Follows:

```
1  package javaiostream;
2  import java.io.FileInputStream;
3  import java.io.FileNotFoundException;
4  import java.io.FileOutputStream;
5  import java.io.IOException;
6  import java.io.InputStreamReader;
7  import java.io.OutputStreamWriter;
8  public class InOutStreamReadWrite  {
9      public static void main(String[] args) throws FileNotFoundException,IOException {
           InputStreamReader in = new InputStreamReader(new FileInputStream("E:\\input.txt"));
11         OutputStreamWriter out = new OutputStreamWriter(new FileOutputStream("E:\\output.txt"));
12         int c;
13         while ((c = in.read()) != -1) {
14            out.write(c);
15         }
16            in.close();
17            out.close();
18     }
19 }
```

input.txt - Notepad

File Edit Format View Help

InputStreamReader

Ln 1, Cc 100% Windows (CRLF) UTF-8

output.txt - Notepad

File Edit Format View Help

InputStreamReader

Ln 1, Co 100% Windows (CRLF) UTF-8

Output - JavaIOStream (run)

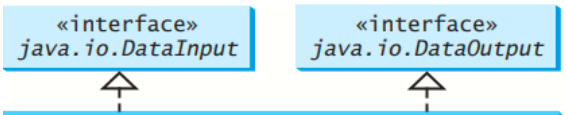
```
run:
BUILD SUCCESSFUL (total time: 1 second)
```

9.2 Random Access files

Java provides the **RandomAccessFile** class to allow data to be read from and written to at any locations in the file.

All of the streams you have used so far are known as read-only or write-only streams. These streams are called sequential streams. A file that is opened using a sequential stream is called a sequential-access file. The contents of a sequential-access file cannot be updated.

Java provides the RandomAccessFile class to allow data to be read from and written to at any locations in a file. A file that is opened using the RandomAccessFile class is known as a random-access file.



java.io.RandomAccessFile

```
+RandomAccessFile(file: File, mode:
  String)
+RandomAccessFile(name: String,
  mode: String)
+close(): void
+getFilePointer(): long

+length(): long
+read(): int
+read(b: byte[]): int
+read(b: byte[], off: int, len: int): int
+seek(pos: long): void

+setLength(newLength: long): void
+skipBytes(int n): int
+write(b: byte[]): void

+write(b: byte[], off: int, len: int):
  void
```

Creates a `RandomAccessFile` stream with the specified `File` object and mode.	
Creates a `RandomAccessFile` stream with the specified file name string and mode.	
Closes the stream and releases the resource associated with it.	
Returns the offset, in bytes, from the beginning of the file to where the next `read` or `write` occurs.	
Returns the number of bytes in this file.	
Reads a byte of data from this file and returns –1 at the end of stream.	
Reads up to `b.length` bytes of data from this file into an array of bytes.	
Reads up to `len` bytes of data from this file into an array of bytes.	
Sets the offset (in bytes specified in `pos`) from the beginning of the stream to where the next `read` or `write` occurs.	
Sets a new length for this file.	
Skips over `n` bytes of input.	
Writes `b.length` bytes from the specified byte array to this file, starting at the current file pointer.	
Writes `len` bytes from the specified byte array, starting at offset `off`, to this file.	

If the file is not intended to be modified, open it with the r mode. This prevents unintentional modification of the file.

```
import java.io.*;

public class Main {
public static void main(String[] args) throws IOException {
try ( // Create a random access file
RandomAccessFile inout = new RandomAccessFile("inout.dat", "rw");
) {
// Clear the file to destroy the old contents if exists
inout.setLength(0);

// Write new integers to the file
for (int i = 0; i < 200; i++)
inout.writeInt(i);

// Display the current length of the file
System.out.println("Current file length is " + inout.length());

// Retrieve the first number
inout.seek(0); // Move the file pointer to the beginning
System.out.println("The first number is " + inout.readInt());

// Retrieve the second number
inout.seek(1 * 4); // Move the file pointer to the second number
System.out.println("The second number is " + inout.readInt());

// Retrieve the tenth number
inout.seek(9 * 4); // Move the file pointer to the tenth number
System.out.println("The tenth number is " + inout.readInt());

// Modify the eleventh number
inout.writeInt(555);

// Append a new number
inout.seek(inout.length()); // Move the file pointer to the end
inout.writeInt(999);

// Display the new length
System.out.println("The new length is " + inout.length());

// Retrieve the new eleventh number
inout.seek(10 * 4); // Move the file pointer to the eleventh number
System.out.println("The eleventh number is " + inout.readInt());
}
}
}
```

```
Current file length is 800
The first number is 0
The second number is 1
The tenth number is 9
The new length is 804
The eleventh number is 555
```

9.4 Defining Generic classes and interfaces

GENERICS

- Generics enable you to detect errors at compile time rather than at runtime.
- The motivation for using Java generics is to detect errors at compile time.
- Generics means parameterized types. The idea is to allow type (Integer, String, … etc., and user-defined types) to be a parameter to methods, classes, and interfaces.
- Using Generics, it is possible to create classes that work with different data types.
- An entity such as class, interface, or method that operates on a parameterized type is a generic entity.
- **Generic** types must be **reference** types.

For example, prior to JDK 1.5 the **java.lang.Comparable** interface was defined as shown in Figure **a**, but since JDK 1.5 it is modified as shown in Figure **b**.

```
package java.lang;

public interface Comparable {
  public int compareTo(Object o)
}
```

(a) Prior to JDK 1.5

```
package java.lang;

public interface Comparable<T> {
  public int compareTo(T o)
}
```

(b) JDK 1.5

Why Generics?

The **Object** is the superclass of all other classes, and Object reference can refer to any object.
These features lack type safety.
Generics add that type of safety feature. We will discuss that type of safety feature in later examples.

Collection without Generics

```
import java.util.ArrayList;
 import java.util.List;
  public class Main {
     public static void main(String[] args) {
          List list = new ArrayList<>();
          list.add("NLJIET");
          list.add(1);
          list.add(2);
          list.add(4.5);
          System.out.println(list);
     }
}
```

[NLJIET, 1, 2, 4.5]

Collection with Generics – to achieve type safety

```
import java.util.ArrayList;
 import java.util.List;
 public class Main {
     public static void main(String[] args) {
          List<Integer> list = new ArrayList<>();
          list.add("NLJIET");
          list.add(1);
          list.add(2);
          list.add(4.5);
          System.out.println(list);
     }
```

```
}
```

```
Exception in thread "main" java.lang.RuntimeException: Uncompilable code - incompatible types: java.lang.String cannot be converted to java.lang.Integer
        at Main.main(Main.java:1)
C:\Users\Pooja\AppData\Local\NetBeans\Cache\12.6\executor-snippets\run.xml:111: The following error occurred while executing this line:
C:\Users\Pooja\AppData\Local\NetBeans\Cache\12.6\executor-snippets\run.xml:68: Java returned: 1
BUILD FAILED (total time: 1 second)
```

NOTE:

1. **List<Integer> in generics forces LIST elements to be of the same kind of class(*Integer* only)**
2. **primitive data types i.e *int* not allowed in generics**

Collection with Generics – integers only

```
import java.util.ArrayList;
 import java.util.List;
 public class Main {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        list.add(1);
        list.add(2);
        System.out.println(list);
    }
}
```

```
[1, 2]
```

Type Parameters in Java Generics

The type parameters naming conventions are important to learn generics thoroughly. The common type parameters are as follows:

- T – Type
- E – Element
- K – Key
- N – Number
- V – Value

Generic Classes: A generic class is implemented exactly like a non-generic class. The only difference is that it contains a type parameter section. There can be more than one type of parameter, separated by a comma. The classes, which accept one or more parameters, are known as parameterized classes or parameterized types.

```
public class Main {
    public static void main(String[] args) {
        Genericc<Integer> obj = new Genericc<>();
        obj.value=10;
        obj.show();
    }
}
class Genericc<T>
{
    T value;
    void show(){
        System.out.println(value);
    }
}
```

NOTE:

1. **Integer can be replaced with any type – Float, Double, Boolean, Character, String etc [T extends Number]**
2. **Generic "T" supports user defined classes as generic types too.**

Generic Method:

Generic Java method takes a parameter and returns some value after performing a task. It is exactly like a normal function, however, a generic method has type parameters that are cited by actual type. This allows the generic method to be used in a more general way. The compiler takes care of the type of safety which enables programmers to code easily since they do not have to perform long, individual type castings.

Sometimes we don't want the whole class to be parameterized, in that case, we can create java generics method. Since the constructor is a special kind of method, we can use generics type in constructors too.
We can also write generic functions that can be called with different types of arguments based on the type of arguments passed to the generic method. The compiler handles each method.
Here is a class showing an example of a java generic method.

// Java program to show working of user defined generic functions

```
class Test {
      // A Generic method example
      static <T> void genericPrint(T element)
      {
            System.out.println(element.getClass().getName() + " = " + element);
      }

      public static void main(String[] args)
      {
            genericPrint (11);
            genericPrint ("GTU");
            genericPrint (1.0);
      }
}
```

Output

```
java.lang.Integer = 11
java.lang.String = GTU
java.lang.Double = 1.0
```

Generic Interface:

Comparable interface (refer collections) is a great example of Generics in interfaces and it's written as:

```
package java.lang;
import java.util.*;

public interface Comparable<T> {
  public int compareTo(T o);
}
```

In similar way, we can create generic interfaces in java. We can also have multiple type parameters as in Map interface. Again we can provide parameterized value to a parameterized type also, for example new `HashMap<String, List<String>>();` is valid.

Advantages of Generics

Programs that use Generics has got many benefits over non-generic code.
1. Code Reuse: We can write a method/class/interface once and use it for any type we want.
2. Type Safety: Generics make errors to appear compile time than at run time (It's always better to know problems in your code at compile time rather than making your code fail at run time). Suppose you want to create an ArrayList that store name of students, and if by mistake the programmer adds an integer object instead of a string, the compiler allows it. But, when we retrieve this data from ArrayList, it causes problems at runtime.

Raw types and backward compatibility

A *raw type* is the name of a generic class or interface without any type arguments. For example, given the generic `Box` class:

```
public class Box<T> {
    public void set(T t) { /* ... */ }
    // ...
}
```

To create a parameterized type of `Box<T>`, you supply an actual type argument for the formal type parameter `T`:

```
Box<Integer> intBox = new Box<>();
```

If the actual type argument is omitted, you create a raw type of `Box<T>`:

```
Box rawBox = new Box();
```

Therefore, `Box` is the raw type of the generic type `Box<T>`. However, a non-generic class or interface type is *not* a raw type.

Raw types show up in legacy code because lots of API classes (such as the `Collections` classes) were not generic prior to JDK 5.0. When using raw types, you essentially get pre-generics behavior — a `Box` gives you `Object`s. For backward compatibility, assigning a parameterized type to its raw type is allowed:

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox;               // OK
But if you assign a raw type to a parameterized type, you get a warning:
Box rawBox = new Box();           // rawBox is a raw type of Box<T>
Box<Integer> intBox = rawBox;     // warning: unchecked conversion
```

Upper Bounded Wildcards

You can use an upper bounded wildcard to relax the restrictions on a variable. For example, say you want to write a method that works on `List<Integer>`, `List<Double>`, *and* `List<Number>`; you can achieve this by using an upper bounded wildcard.

The `sumOfList` method returns the sum of the numbers in a list:

```
public static double sumOfList(List<? extends Number> list) {
    double s = 0.0;
    for (Number n : list)
        s += n.doubleValue();
    return s;
}
```

The following code, using a list of `Integer` objects, prints `sum = 6.0`:

```
List<Integer> li = Arrays.asList(1, 2, 3);
System.out.println("sum = " + sumOfList(li));
A list of Double values can use the same sumOfList method. The following
code prints sum = 7.0:
List<Double> ld = Arrays.asList(1.2, 2.3, 3.5);
System.out.println("sum = " + sumOfList(ld));
```

Lower Bounded Wildcards

The Upper Bounded Wildcards section shows that an upper bounded wildcard restricts the unknown type to be a specific type or a subtype of that type and is represented using the `extends` keyword. In a similar way, a *lower bounded* wildcard restricts the unknown type to be a specific type or a *super type* of that type.
A lower bounded wildcard is expressed using the wildcard character ('`?`'), following by the `super` keyword, followed by its *lower bound*: `<? super A>`.

Note: You can specify an upper bound for a wildcard, or you can specify a lower bound, but you cannot specify both.

To write the method that works on lists of Integer and the supertypes of Integer, such as Integer, Number, and Object, you would specify List<? super Integer>. The term List<Integer> is more restrictive than List<? super Integer> because the former matches a list of type Integer only, whereas the latter matches a list of any type that is a supertype of Integer.
The following code adds the numbers 1 through 10 to the end of a list:

```
public static void addNumbers(List<? super Integer> list) {
    for (int i = 1; i <= 10; i++) {
        list.add(i);
    }
}
```

What Is Type Erasure?

Type erasure can be explained as the process of enforcing type constraints only at compile time and discarding the element type information at runtime.

For example:

```
public static  <E> boolean containsElement(E [] elements, E element){
    for (E e : elements){
        if(e.equals(element)){
            return true;
        }
    }
    return false;
}
```

The compiler replaces the unbound type *E* with an actual type of *Object*:

```
public static  boolean containsElement(Object [] elements, Object element){
    for (Object e : elements){
        if(e.equals(element)){
            return true;
        }
    }
    return false;
}
```

Therefore the compiler ensures type safety of our code and prevents runtime errors.

Restrictions on Generics

To use Java generics effectively, you must consider the following restrictions:

- Cannot Instantiate Generic Types with Primitive Types

class Pair<K, V>

When creating a Pair object, you cannot substitute a primitive type for the type parameter K or V:

```
Pair<int, char> p = new Pair<>(8, 'a'); // compile-time error
```

- Cannot Create Instances of Type Parameters

```
public static <E> void append(List<E> list) {
    E elem = new E();  // compile-time error
```

- Cannot Declare Static Fields Whose Types are Type Parameters

```
public class MobileDevice<T> {
    private static T os;
```

- Cannot Use *instanceof* With Parameterized Types

```
if (list instanceof ArrayList<Integer>) {  // compile-time error
```