MODULE 3
Streamlining I/O and Exploring Collections Framework

Java IO:

- I/O basics
- Reading and writing console input
- The PrintWriter class
- Reading and writing files.

Collections:

- Overview, Recent changes to collections,
- Collection interfaces-

1. List interface,
2. Set interface,
3. Queue interface
4. Dequeue interface,

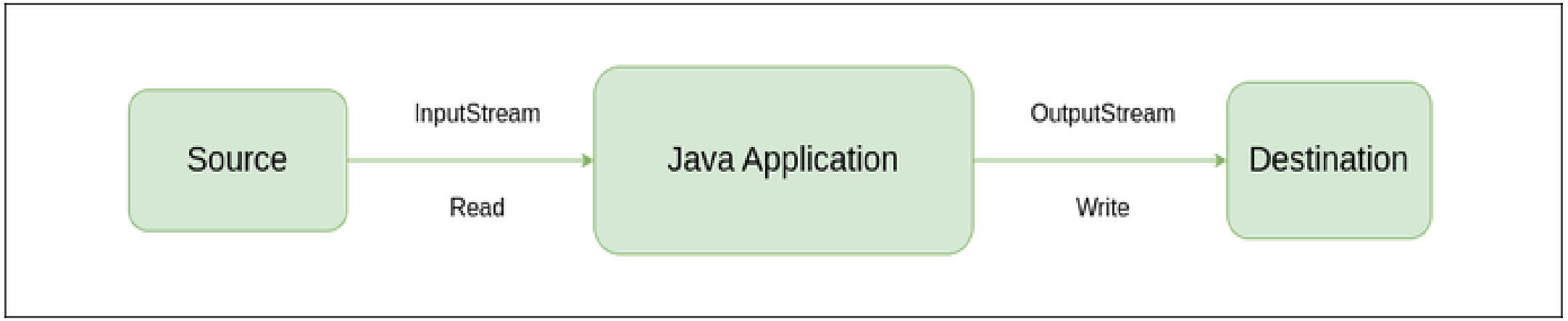
- Collection classes

1. ArrayList class
2. LinkedList class
3. TreeSet class
4. PriorityQueue class and Array Dequeue class

- Accessing a Collections via an iterator

I/O basics:

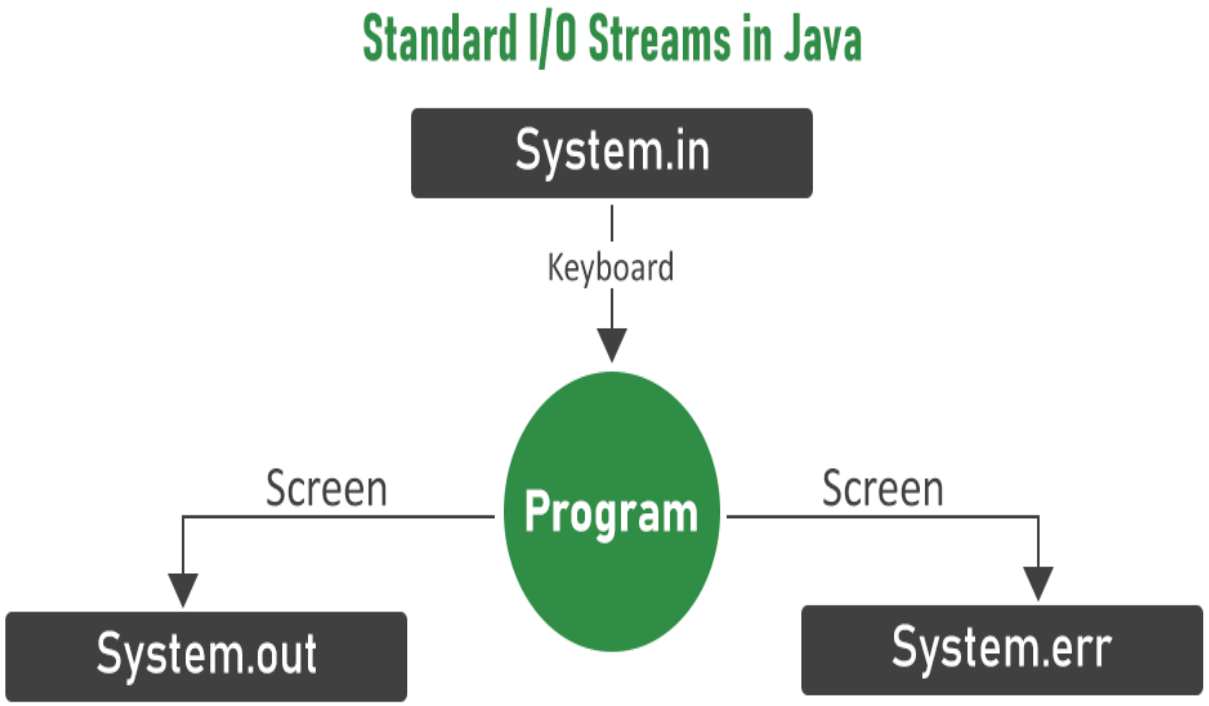
- ***Java I/O (Input/Output) provides a way to read data from input sources (like keyboard, files, or network) and write data to output destinations (like screen, files, or network).***
- In simple words, we can say that the Java IO helps the users to take the input and produce output based on that input.
- ***The java.io package contains all the classes required for input and output operations***. We can perform **file handling in Java** by Java I/O API.



Stream :

- A stream can be defined as a sequence of data.
- Java I/O stream is the flow of data that you can either read from, or you can write to.
- An I/O stream represents an input source or an output destination.
- A stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays.
- The stream in the java.io package supports many data such as primitives, object, localized characters, etc.
- Java uses streams to perform read and write operations in file permanently. **Java I/O stream** is also called as **File Handling**, or **File I/O.**

Standard Streams / Predefined Streams



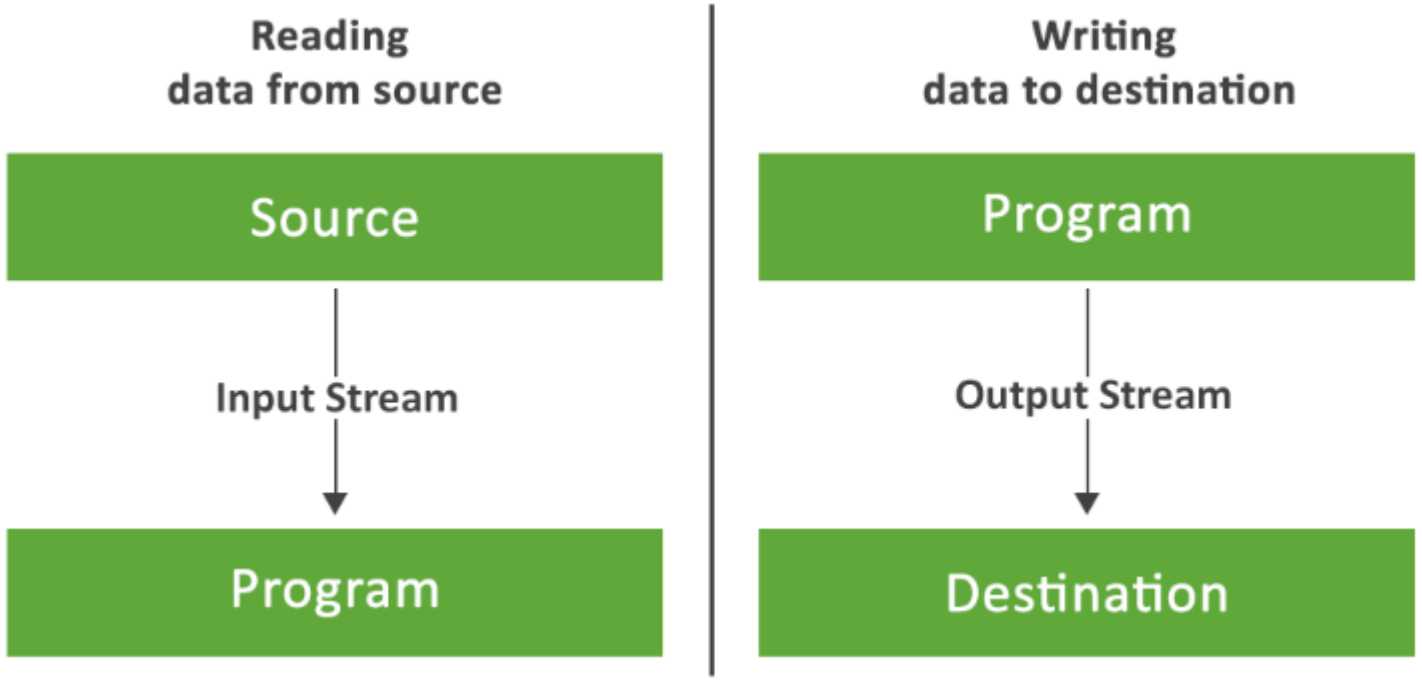
Standard Input − This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as **System.in**.

Standard Output − This is used to output the data produced by the user's program and usually a computer screen is used for standard output stream and represented as **System.out**.

Standard Error − This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as **System.err**.

TYPES OF STREAM:

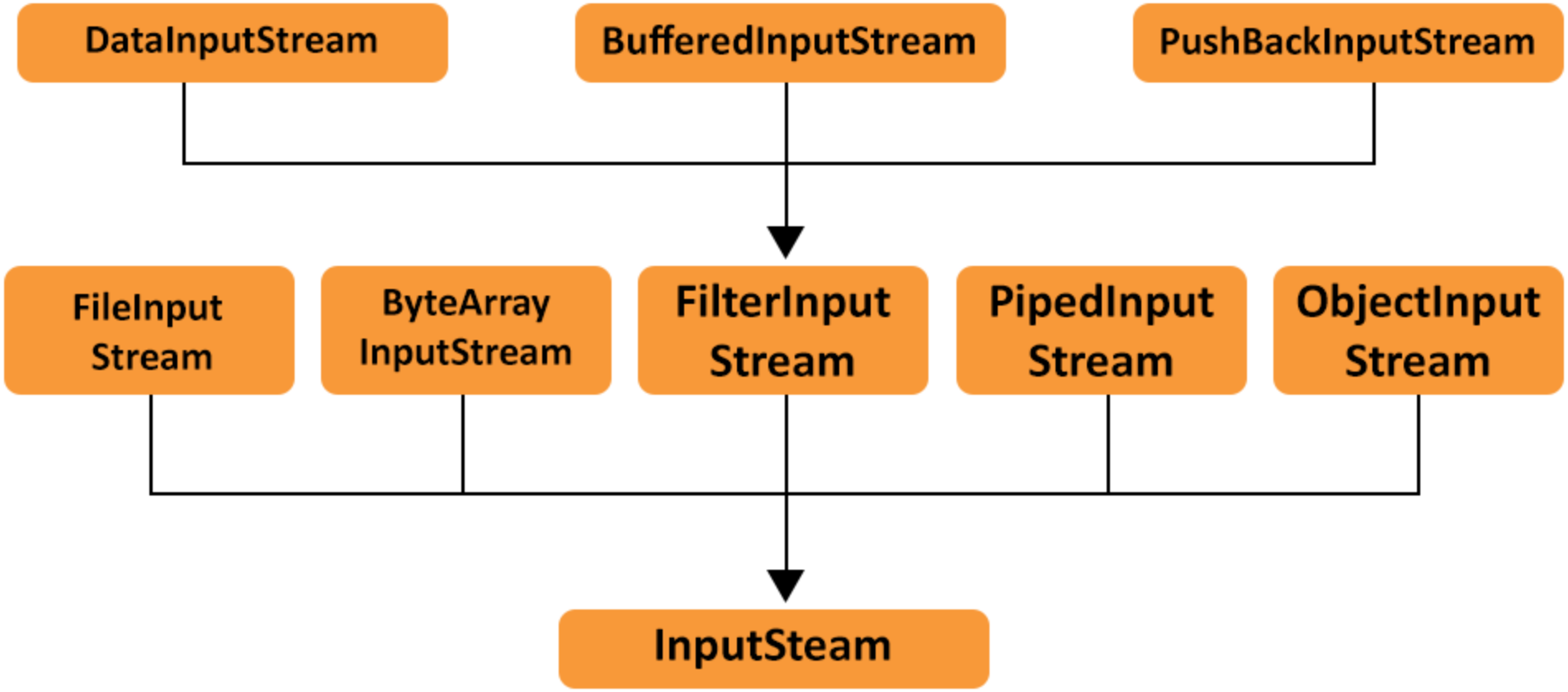
- **Depending on the type of operations**, streams can be divided into two primary classes:



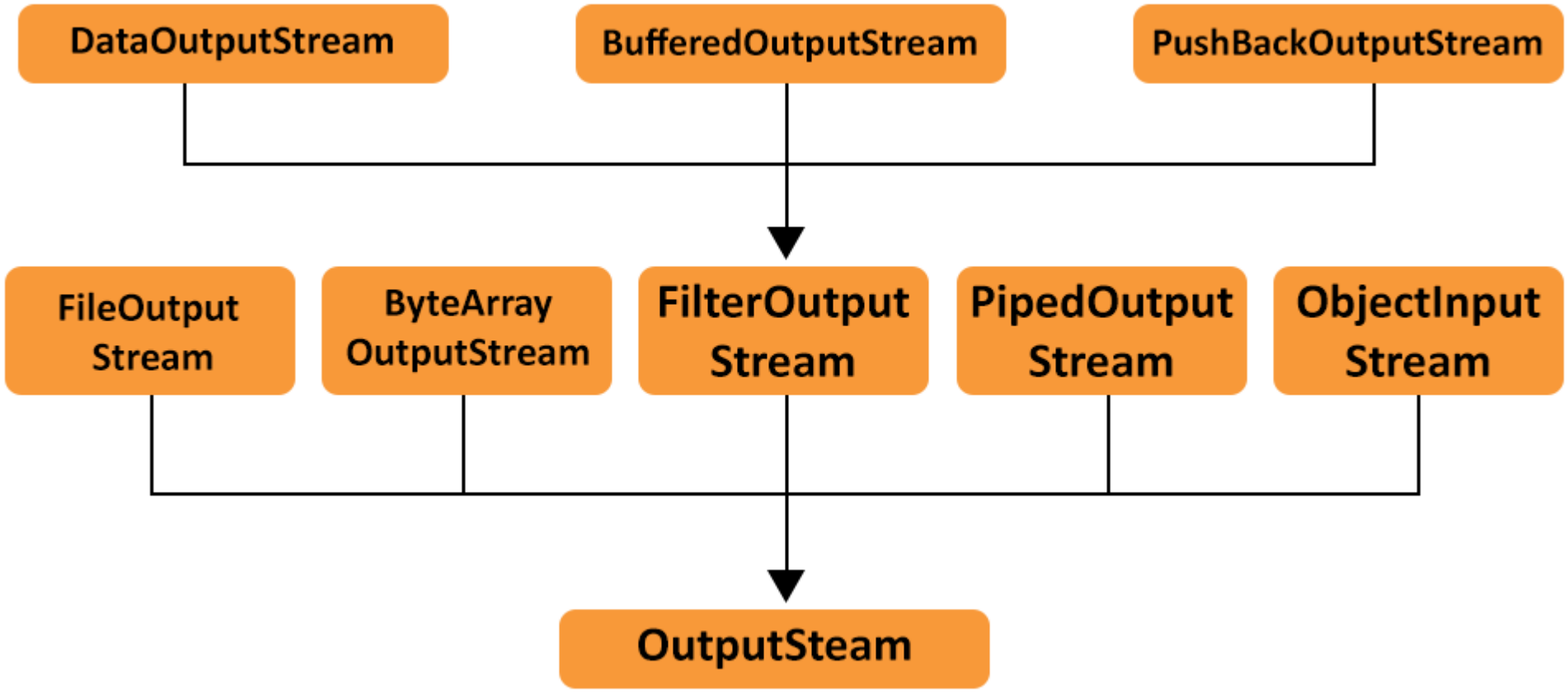
InputStream − The InputStream is used to read data from various sources like keyboard, file, network, etc.

OutputStream − The OutputStream is used for writing data to various output devices like monitor, file, network, etc.

Various InputStream Classes



Various OutputStream Classes



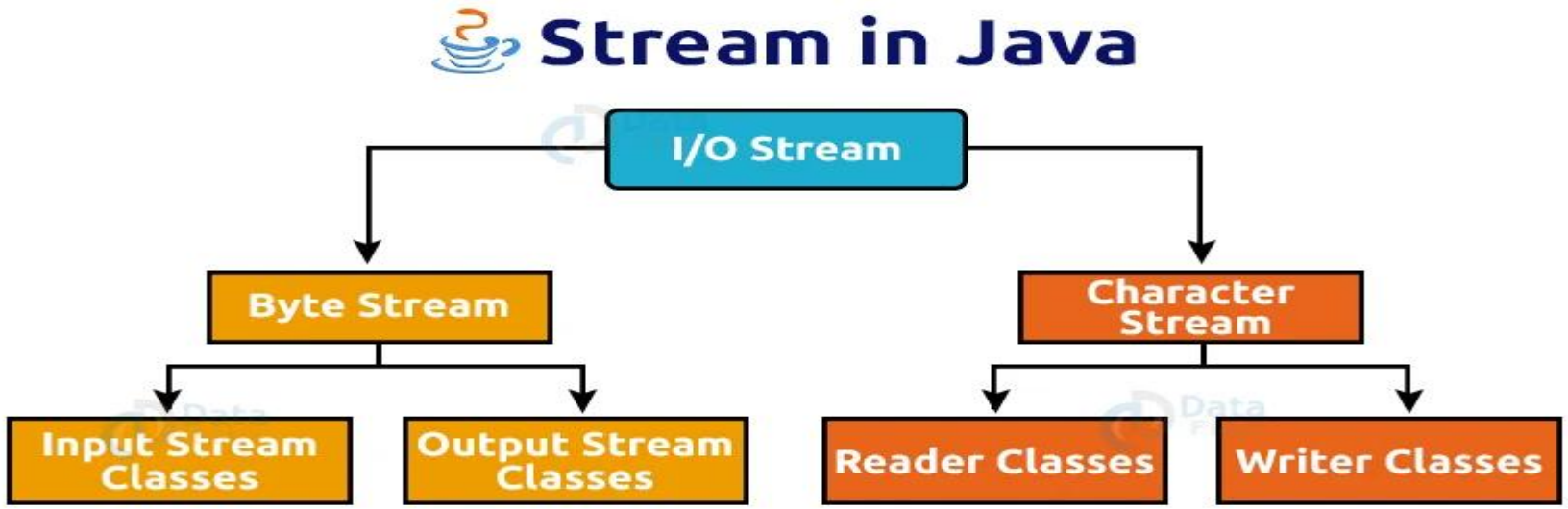
STREAM BASED ON THE TYPE OF DATA:

•Java supports Input and Output(I/O) operations with characters/text and binary stream.

• Java I/O model is highly flexible so that it can accommodate data sources like Files, Arrays, piped streams, etc.

• Since, ASCII (8 bit) character encoding was not sufficient to cover all possible character sets, Java uses 16 bit Unicode to represent characters.

• In Java, text(collection of characters) is represented as two-byte UNICODE characters.

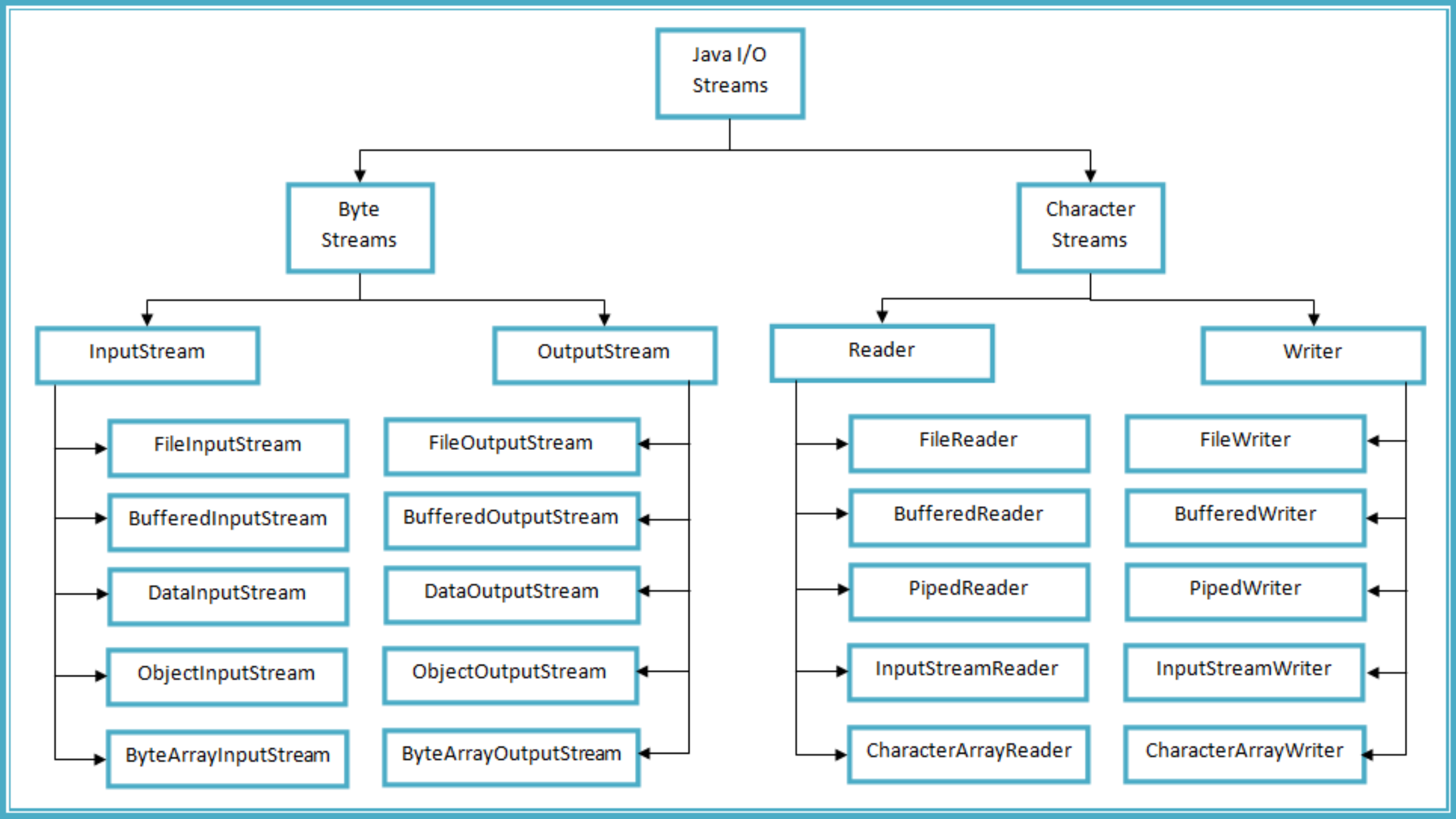


ByteStream:

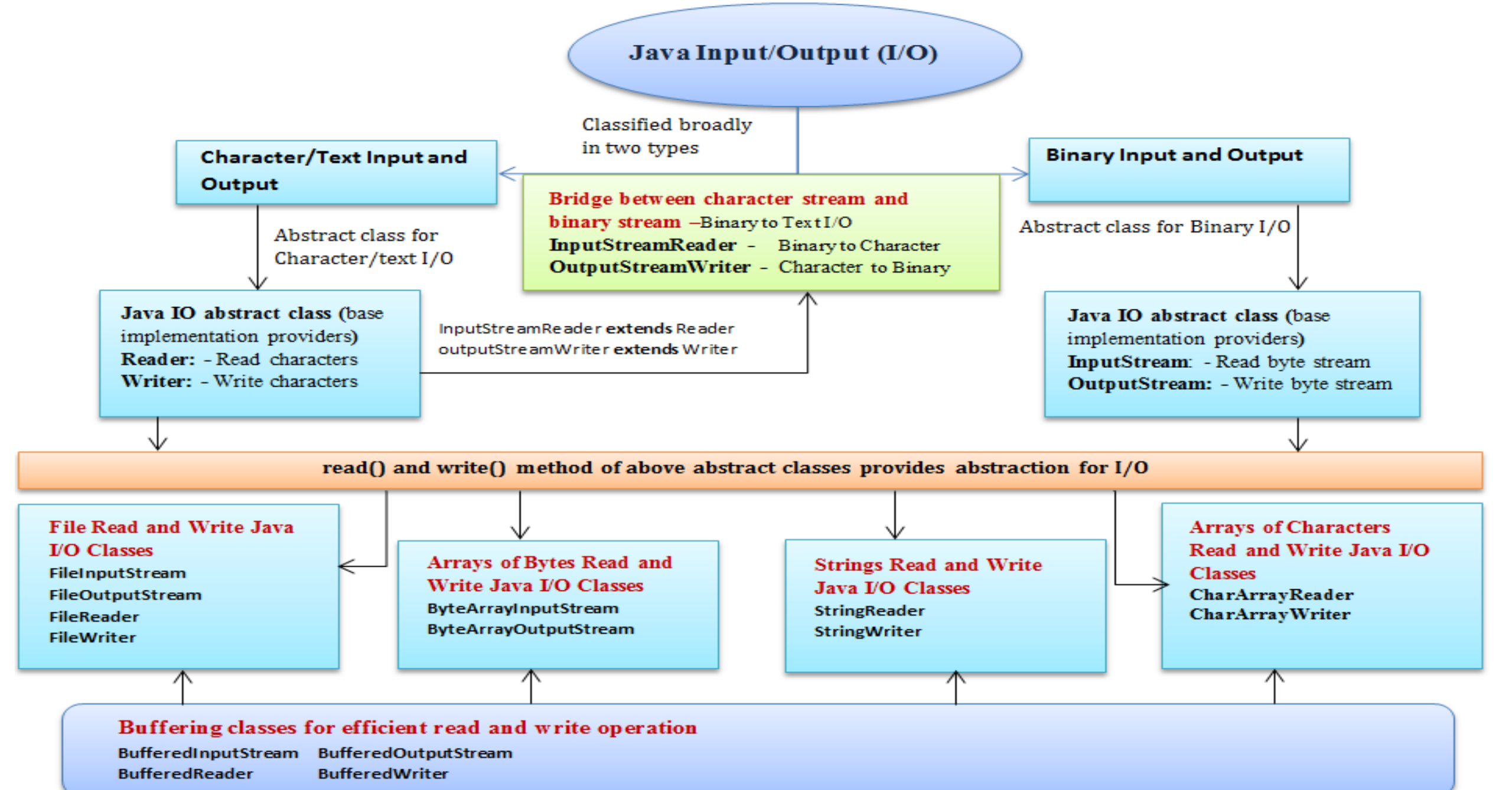
- This is used to process data byte by byte (8 bits) i.e. used to read or write byte data.
- Though it has many classes, the ***FileInputStream and the FileOutputStream*** are the most popular ones.
- ***The FileInputStream is used to read from the source and FileOutputStream is used to write to the destination.***

CharacterStream:

- In Java, characters are stored using Unicode conventions. Character stream automatically allows us to read/write data character by character.
- Though it has many classes, the ***FileReader and the FileWriter*** are the most popular ones.
- ***FileReader and FileWriter are character streams used to read from the source and write to the destination respectively*.**


Java I/O Streams
Byte Streams
Character Streams
InputStream
OutputStream
Reader
Writer
FileInputStream
BufferedInputStream
DataInputStream
ObjectInputStream
ByteArrayInputStream
FileOutputStream
BufferedOutputStream
DataOutputStream
ObjectOutputStream
ByteArrayOutputStream
FileReader
BufferedReader
PipedReader
InputStreamReader
CharacterArrayReader
FileWriter
BufferedWriter
PipedWriter
InputStreamWriter
CharacterArrayWriter

Java I/O and classes involved to support both text I/O and binary I/O



- For handling **character I/O Java provides Reader and Writer abstract class** and for **byte stream, InputStream and OutputStream -** which provides flexible read() and write() methods and gives an abstraction capability to subclasses.
- For **File related operation** (character and byte stream read/write)**-** Java provides concrete classes like FileReader, FileWriter, FileInputStream and FileOutputStream.
- For handling **byte array/character array-** ByteArrayInputStream, ByteArrayOutputStream / CharArrayReader, CharArrayWriter
- For **String object -** read and write operation is carried out with StringReader and StringWriter.

Byte Streams:

Java byte streams are used to perform input and output of 8-bit bytes.

The most frequently used classes are FileInputStream and FileOutputStream.

Common Methods in InputStream

Method Signature	Meaning / Description
int read()	Reads **one byte** of data from the input stream. Returns the byte read, or -1 if end of stream is reached.
int read(byte[] b)	Reads **bytes into the array** b. Returns the number of bytes read, or -1 if end of stream.
int read(byte[] b, int off, int len)	Reads up to len bytes into b starting at offset off. Returns number of bytes read or -1.
long skip(long n)	Skips over and discards n bytes of data from the input stream. Returns actual number of bytes skipped.
int available()	Returns an estimate of the number of bytes that can be read (or skipped) without blocking.
void close()	Closes the input stream and releases any system resources associated with it.
synchronized void mark(int readlimit)	Marks the current position in the stream. You can return to this position later using reset().
synchronized void reset()	Resets the stream to the most recent mark. If mark is not set or not supported, throws IOException.

Common Methods in OutputStream

Method Signature	Meaning / Description
void write(int b)	Writes the **lowest 8 bits** of the given integer b (i.e., a single byte) to the output stream.
void write(byte[] b)	Writes **all bytes** from the byte array b to the output stream.
void write(byte[] b, int off, int len)	Writes len bytes from the byte array b, starting at offset off, to the output stream.
void flush()	Forces any buffered output bytes to be written out. Useful for flushing data to the destination before closing.
void close()	Closes the output stream and releases any system resources associated with it.

Example: Program to copy an input file into an output file

```
import java.io.*;
public class CopyFile {
public static void main(String args[]) throws IOException {
FileInputStream in = null;
FileOutputStream out = null;
try {
in = new FileInputStream("input.txt");
out = new FileOutputStream("output.txt");
int c;
while ((c = in.read()) != -1) {
out.write(c);
}
 }finally {
if (in != null) {
in.close();
 }
if (out != null) {
out.close();
} }} }
```

Character Streams

- Java **Byte** streams are used to perform input and output of 8-bit bytes, whereas Java **Character** streams are used to perform input and output for 16-bit unicode.
- 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 above example, which makes the use of these two classes to copy an input file (having unicode characters) into an output file −

```
import java.io.*;

public class CopyFile {

public static void main(String args[]) throws IOException {

FileReader in = null;

FileWriter out = null;
```

```
try {
 in = new FileReader("input.txt");
out = new FileWriter("output.txt");
 int c;
while ((c = in.read()) != -1) {
 out.write(c);
} }
finally {
if (in != null) {
 in.close();
 }
if (out != null) {
 out.close();
} } } }
```

Reading and writing console input/output in Java

Java provides several ways to read input from the console:

1. **Using Scanner class (Recommended for most use cases)**
2. **Using BufferedReader with InputStreamReader**

In Java, both Scanner and BufferedReader are used to read input, but they serve different purposes based on the use case.

- Scanner is ideal for reading user input from the console when working with simple data types like integers, strings, or tokens. It provides convenient methods like nextInt(), nextLine(), and nextDouble() for parsing input directly, making it beginner-friendly and suitable for interactive console applications.
- On the other hand, ***BufferedReader is preferred when performance matters***, especially ***for reading large text data from files or streams*** efficiently. It reads input line by line using the readLine() method and is ***commonly paired with InputStreamReader for console input or FileReader for file input.***

1.Scanner Class in Java

- Scanner is a class in java.util package used for obtaining the input of the primitive types like int, double, etc. and strings.
- It is the easiest way to read input in a Java program, though not very efficient if you want an input method for scenarios where time is a constraint like in competitive programming.

Commonly Used Methods in Scanner

Method	Description
next()	Reads a single word (until space).
nextLine()	Reads the entire line including spaces.
nextInt()	Reads an integer value.
nextDouble()	Reads a double value.
nextFloat()	Reads a float value.
nextLong()	Reads a long value.
nextBoolean()	Reads a boolean (true or false).
hasNext()	Returns true if there is another token.
hasNextInt()	Checks if the next input is an integer.
close()	Closes the scanner and releases resources.

- To read a single character, we use next().charAt(0). next() function returns the next token/word in the input as a string and charAt(0) function returns the first character in that string.

```
// Java program to read data of various types using Scanner class.
import java.util.Scanner;
public class ScannerDemo1
{
public static void main(String[] args)
{
// Declare the object and initialize with predefined standard input object
Scanner sc = new Scanner(System.in);

// String input
String name = sc.nextLine();

// Character input
char gender = sc.next().charAt(0);

// Numerical data input byte, short and float can be read  using similar-named functions.

 int age = sc.nextInt();
long mobileNo = sc.nextLong();
double cgpa = sc.nextDouble();

// Print the values to check if the input was correctly obtained.

 System.out.println("Name: "+name);

System.out.println("Gender: "+gender);

System.out.println("Age: "+age);

System.out.println("Mobile Number: "+mobileNo);

 System.out.println("CGPA: "+cgpa);
}
}
```

Array elements With Scanner

```
import java.util.Arrays;

import java.util.Scanner;

public class ArrayReadingWithScanner {  public static void main(String args[]) {
Scanner s = new Scanner(System.in);
System.out.println("Enter the length of the array:");
int length = s.nextInt();

int [] myArray = new int[length];

System.out.println("Enter the elements of the array:");
for(int i=0; i<length; i++ ) {  myArray[i] = s.nextInt();
}
System.out.println(Arrays.toString(myArray));
}
}
```

2. Reading Console Input Using BufferedReader

Console input means reading data entered by the user during the program's execution — usually through the **keyboard**.

Java provides several ways to read input. One ***efficient way is using the BufferedReader*** class

•It **reads text efficiently**. Reading or writing one character or byte at a time is slow.

•**A buffer allows the program to read or write larger chunks of data at once.**

•This reduces the number of times the program accesses the disk or network — making it **faster and more efficient**.

•It supports **reading a full line at a time**.

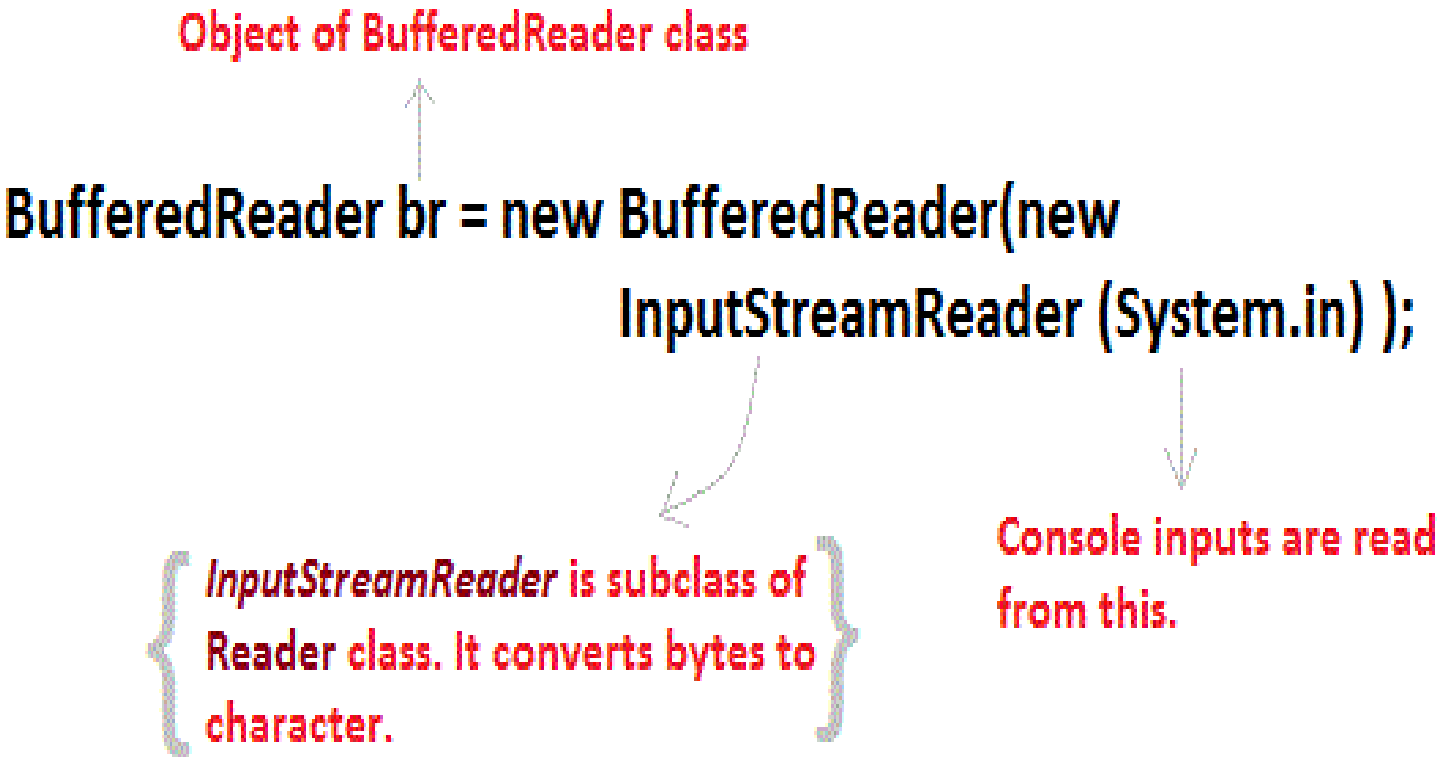
•It uses an internal **buffer** to speed up reading (that’s why it’s called *Buffered*Reader).

•A **buffer** is a **temporary memory storage area** used to hold data while it's being transferred between two places — like **between a program and a file**, or **between a program and the keyboard**.

How to use:

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

How It Works:



- **System.in** is a **byte stream** (reads raw bytes from the keyboard).
- **InputStreamReader** converts those **bytes into characters**.
- **BufferedReader** wraps around it and provides **line-by-line reading** using a **buffer**.

Common Methods in BufferedReader

Method Signature	Description
String readLine()	Reads a **line of text**. Returns null if end of stream is reached.
int read()	Reads a **single character**. Returns the character as an int, or -1 if end of stream.
int read(char[] cbuf, int off, int len)	Reads characters into a **portion of a character array**. Returns the number of characters read, or -1.
boolean ready()	Returns true if the stream is ready to be read without blocking.
void close()	Closes the stream and releases any system resources.

Note:

- You need to handle or declare IOException when using BufferedReader.
- BufferedReader reads **strings only**, so if you need numbers, you'll have to convert manually using Integer.parseInt(), Double.parseDouble(), etc.

Reading Characters

- ***read() method is used with BufferedReader object to read characters***. As this function returns integer type value has we need to use typecasting to convert it into **char** type.

method signature:

int **read()** throws **IOException**

- Below is a simple example explaining character input.

```
class CharRead {
public static void main( String args[]) {
BufferedReader br = new Bufferedreader(new InputstreamReader(System.in));
 char c = (char)br.read(); //Reading character
 }}
```

- Each time that read( ) is called, it reads a character from the input stream and returns it as an integer value. It returns 1– when the end of the stream is encountered.

- **The following program demonstrates read( ) by reading characters from the console until the user types a "q."**
- Notice that any I/O exceptions that might be generated are simply thrown out of **main( )**.
- Such an approach is common when reading from the console, but you can handle these types of errors yourself, if you chose.

// Use a BufferedReader to read characters from the console.

```
import java.io.*;
class BRRead {
public static void main(String args[]) throws IOException
{
char c;
BufferedReader br = new Buffered Reader (new InputStreamReader (System.in));
System.out.println("Enter characters, 'q' to quit.");
// read characters
do {
c = (char) br.read();
System.out.println(c);
} while(c != 'q');    }            }
```

Reading Strings

To read a string from the keyboard, use the version of **readLine( )** that is a member of the **BufferedReader** class. Its general form is shown here:

String readLine( ) throws IOException

As you can see, it returns a **String** object.

The following program demonstrates **BufferedReader** and the **readLine( )** method; the program reads and displays lines of text .

// Read a string from console using a BufferedReader.

```
 import java.io.*;
  public class ReadLineExample {
    public static void main(String[] args) {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        try {
            System.out.print("Enter a line of text: ");
            String text = br.readLine(); // reads a line of text
            System.out.println("You entered: " + text);
        } catch (IOException e) {
            System.out.println("Error reading input: " + e.getMessage());
        }
    }
}
```

Reading Multiple Lines from User Input (Console):

```
import java.io.*;
public class MultiLineInput {
    public static void main(String[] args) {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Enter multiple lines of text (type 'exit' to stop):");
        try {
            String line;
            while (true) {
                line = br.readLine();  // Read a line
                if (line.equalsIgnoreCase("exit")) {
                    break;  // Exit when user types "exit"
                }
                System.out.println("You entered: " + line);
            }
        } catch (IOException e) {
            System.out.println("Error reading input: " + e.getMessage());
        }  }   }
```

Writing Console Output in Java

In Java, writing output to the console is done using the System.out stream. This is a standard output stream that allows printing text or data to the console window.

Common Methods:

Method	Description
System.out.print()	Prints text **without** a newline at the end
System.out.println()	Prints text **with** a newline at the end
System.out.printf()	Prints formatted output (like printf in C)

Reading and Writing Files

File handling is an essential aspect of programming, and Java provides numerous libraries for handling files.

The ***java.io package is used for file handling in Java***. It provides various classes for handling files, such as ***File, FileReader, FileWriter, FileInputStream, and FileOutputStream***.

File operations in Java

The following are the several operations that can be performed on a file in Java :

- **Create a File**
- **Read from a File**
- **Write to a File**
- **Delete a File**

Java File Class Methods

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

CREATING A FILE

- In order to create a file in Java, you can use the **createNewFile() method**.
- If the file is successfully created, it will return a Boolean value true and false if the file already exists.

```
import java.io.File;
import java.io.IOException;
public class CreateFileExample {
    public static void main(String[] args) {
        // Create a File object
        File file = new File("newfile.txt");// Constructor takes String (filename) as an argument
        try {
            // Create the file if it doesn't already exist
            if (file.createNewFile()) {
                System.out.println("File created: " + file.getName());
            } else {
                System.out.println("File already exists.");
            }
        } catch (IOException e) {
            System.out.println("An error occurred: " + e.getMessage());
        }    }  }
```

Reading and Writing Files Using FileReader and FileWriter

The **FileReader** and **FileWriter** classes in Java are designed to handle reading from and writing to files that contain character data. They are part of the **java.io** package and are commonly used for text file operations.

FileReader and FileWriter Methods:

Class	Method	Description
FileReader	read()	Reads a single character from the file. It returns the Unicode value (int) of the character. Returns -1 if the end of the file is reached.
	close()	Closes the FileReader and releases system resources.
FileWriter	write(int c)	Writes a single character to the file.
	write(String str)	Writes a string to the file.
	write(char[] cbuf)	Writes an array of characters to the file.
	close()	Closes the FileWriter and releases system resources.

Reading Files using FileReader

The **FileReader** class is used for reading files character by character. It's typically used to read text files.

Constructor:

1. FileReader(String fileName):

•This constructor creates a FileReader using a **file name as a string**. Creates a FileReader object to read from the file specified by the given file name. You simply pass the name or path of the file as a string, and it opens that file for reading.

Example:

```
FileReader fr = new FileReader("sample.txt");
```

2.FileReader(File file):

•Creates a FileReader object to read from the specified File object.

•This constructor accepts a File object as an argument. You first create a File object, then pass it to FileReader.

Example:

```
    // Create a File object
File file = new File("sample.txt");
// Pass the File object to FileReader
FileReader fr = new FileReader(file);
```

Example of Reading a File:

```
import java.io.FileReader;
import java.io.IOException;
public class ReadFileExample {
    public static void main(String[] args) {
        FileReader reader = null;
        try {
            // Create a FileReader object
            reader = new FileReader("example.txt");
            int character;
            // Read each character until the end of the file (-1)
            while ((character = reader.read()) != -1) {
                System.out.print((char) character);  // Print the character
            }
        } catch (IOException e) {
            System.out.println("An error occurred: " + e.getMessage());
        } finally {
              if (reader != null) {
                    reader.close();  // Always close the reader to free resources
} } }
```

Writing Files using FileWriter

The **FileWriter** class is used for writing character data to a file. If the file does not exist, it will be created automatically. If the file exists, it will be overwritten unless you specify the append mode.

Constructor:

•**FileWriter(String fileName)**: Creates a FileWriter object to write to the file specified by the given file name.

•**FileWriter(File file)**: Creates a FileWriter object to write to the file specified by the File object.

```
// Step 1: Create a File object
File file = new File("output.txt");
// Step 2: Create FileWriter using the File object
FileWriter writer = new FileWriter(file);
```

•**FileWriter(String fileName, boolean append)**: Creates a FileWriter object to write to the file, with an option to **append** data to the file (if append is true) rather than overwriting it.

Example of Writing to a File:

```
import java.io.*;
public class CreateAndWriteFile {
    public static void main(String[] args) {
        try {
            File file = new File("example.txt");
            if (file.createNewFile()) {
                System.out.println("File created: " + file.getName());
            } else {
                System.out.println("File already exists.");
            }
            FileWriter writer = new FileWriter(file);  // This will overwrite existing content
            writer.write("This is the first line in the file.\n");
            writer.write("This file was created and written using Java.\n");
            writer.close();
            System.out.println("Successfully wrote to the file.");
        } catch (IOException e) {
            System.out.println("An error occurred: " + e.getMessage());
        }
    } }
```

Deleting a File

```
import java.io.*;
public class DeleteOp {
public static void main(String[] args)
{
File Obj = new File("myfile.txt");
if (Obj.delete()) {
System.out.println("The deleted file is : "+ Obj.getName());
}
else {
System.out.println("Failed in deleting the file.");
}
}
}
```

Reading and Writing Text Files Using BufferedReader with FileReader and BufferedWriter with FileWriter in Java

These are **character-based** I/O classes in Java used to read and write **text files** efficiently.

•BufferedReader: Reads text from a character-input stream, buffering characters for efficient reading.

•BufferedWriter: Writes text to a character-output stream, buffering characters to provide efficient writing.

•This combination is suitable when you're working with plain text files (like .txt, .csv, .log) and need to read or write large amounts of text data, BufferedReader and BufferedWriter provide efficient I/O operations by buffering the data in memory.

Basic Syntax:

```
BufferedReader br = new BufferedReader(new FileReader("input.txt"));
BufferedWriter bw = new BufferedWriter(new FileWriter("output.txt"));
```

Note:

•BufferedReader + FileReader is used to read the **input file** line by line.

•BufferedWriter + FileWriter is used to **write** each line to a new file.

Write a Java program that reads and displays the content of a text file. The program should prompt the user for the file name, open the file, and print its contents line by line.

```
import java.io.*;
import java.util.Scanner;
public class DisplayFileContent {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the file name: ");
        String fileName = scanner.nextLine();  // Input from user
        BufferedReader reader = null;
        try {
            reader = new BufferedReader(new FileReader(fileName));
            String line;
            // Read and print each line
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
        } catch (FileNotFoundException e) {
            System.out.println("File not found");
        } catch (IOException e) {
            System.out.println("An error occurred while reading the file.");
        }
        scanner.close();
    }
}
```

Reading and Writing Files Using FileInputStream and FileOutputStream:

•Both classes are part of the **java.io package**.

•They are used for reading and writing data in the form of **raw bytes** (not characters).

•Suitable for **binary files** like images, audio, PDF, and also text files (if you manually convert bytes to characters).

FileInputStream (for Reading)

FileInputStream is used to read data **from a file** byte-by-byte.

Common Constructors:

Constructor	Description
FileInputStream(String fileName)	Opens a file using its name.
FileInputStream(File file)	Opens a file using a File object.

Common Methods:

Method	Description
int read()	Reads one byte at a time; returns -1 at EOF.
int read(byte[] b)	Reads bytes into an array.
int read(byte[] b, int off, int len)	Reads up to len bytes into b starting at off.
void close()	Closes the input stream.

FileOutputStream (for Writing)

FileOutputStream is used to **write bytes to a file**.

Common Constructors:

Constructor	Description
FileOutputStream(String fileName)	Opens or creates a file with given name.
FileOutputStream(File file)	Uses a File object.
FileOutputStream(String fileName, boolean append)	If append = true, it appends to the file instead of overwriting.

Common Methods:

Method	Description
void write(int b)	Writes a single byte.
void write(byte[] b)	Writes all bytes from the array.
void write(byte[] b, int off, int len)	Writes part of the byte array.
void close()	Closes the output stream.

Example: Program to copy an input file into an output file

```
import java.io.*;
public class CopyFile {
public static void main(String args[]) throws IOException {
FileInputStream in = null;
FileOutputStream out = null;
try {
in = new FileInputStream("input.txt");
out = new FileOutputStream("output.txt");
int c;
while ((c = in.read()) != -1) {
out.write(c);
}
 }finally {
if (in != null) {
in.close();
 }
if (out != null) {
out.close();
} }} }
```

The PrintWriter class

- The PrintWriter class of the java.io package can be used to write output data in a commonly readable form (text).
- The PrintWriter class in Java is a part of the java.io package and is used for writing formatted text to an output stream (like a file, console, or other destinations).
- It extends the abstract class Writer.

- Unlike other writers, PrintWriter converts the primitive data(int, float, char, etc.) into the text format. It then writes that formatted data to the writer.
- Also, the PrintWriter class does not throw any input/output exception. Instead, we need to use the checkError() method to find any error in it.

Constructors of PrintWriter

1. PrintWriter(File file)
2. PrintWriter(File file, String charsetName)
3. PrintWriter(OutputStream out)
4. PrintWriter(OutputStream out, boolean autoFlush)
5. PrintWriter(Writer out)
6. PrintWriter(Writer out, boolean autoFlush)

Common Methods:

Method	Description
print()	Writes data without a newline. Overloaded for various types.
println()	Writes data followed by a newline.
printf()	Supports formatted output like in C.
flush()	Forces any buffered output to be written immediately.
close()	Closes the stream.
checkError()	Checks if an error has occurred during writing.

Example: Writing to a File using PrintWriter

```
import java.io.*;
public class PrintWriterExample {
    public static void main(String[] args) throws IOException {
        PrintWriter pw = new PrintWriter(new FileWriter("output.txt"));
        pw.println("Hello, PrintWriter!");
        pw.printf("Formatted number: %.2f%n", 12.3456);
        pw.close();
    }
}
```

Writing Console Output Using PrintWriter

```
import java.io.PrintWriter;
public class ConsoleOutputWithPrintWriter {
    public static void main(String[] args) {
        // Create PrintWriter that writes to the console
        PrintWriter pw = new PrintWriter(System.out, true); // autoFlush set to true
        // Output messages
        pw.println("This is printed using PrintWriter.");
        pw.printf("Formatted output: %d + %d = %d%n", 10, 20, 10 + 20);
        pw.println("End of PrintWriter output.");
    }
}
```

Difference Between PrintWriter and System.out

Feature	PrintWriter	System.out
Class	java.io.PrintWriter	java.io.PrintStream
Output Destination	Can write to files, sockets, streams, etc.	Generally prints to the console (standard output)
AutoFlush Support	Optional via constructor	AutoFlush on println, printf, etc.
Formatting Support	print(), println(), printf()	Same, but less flexible in terms of redirection
Exception Handling	Does not throw IOException	Also does not throw, but based on underlying stream
Customization	Highly customizable	Mostly fixed to standard output (console)

Java Collections Framework

Collections in java:

A collection in Java is a container object that is used for storing multiple homogeneous and heterogeneous, duplicate, and unique elements without any size limitation.

Consider the example of a piggy bank. We all had it during our childhood where we used to store our coins. This piggy bank is called a Collection and the coins are nothing but objects.

Technically, a collection is an object or a container that stores a group of other objects.

Java Collections Framework :

The Java Collections Framework is a collection of interfaces and classes, that provides easy management of a group of objects.

This framework has several useful classes which have tons of useful functions which makes a programmer task super easy.

Need for Collections Framework in Java

- In Java programming, managing groups of related objects efficiently is a common requirement.
- **Initially, Java developers used arrays to store and manipulate multiple elements**. However, arrays have several limitations.
- ***They are of fixed size***, meaning once an array is created, its length cannot be changed. This makes it unsuitable for applications where the number of elements can vary at runtime.
- ***Arrays also do not provide built-in methods for common operations like sorting, searching, or inserting elements at arbitrary positions,*** requiring the programmer to write extra code for these tasks.
- Furthermore, arrays store elements in a linear and sequential way, and **do not support advanced data structures like sets, maps, queues, or stacks out of the box.**

- To address these shortcomings, Java introduced the **Collections Framework** in version 1.2.
- **Collections Framework** is a well-designed set of interfaces and classes that ***provide ready-to-use data structures and algorithms***. It includes interfaces such as List, Set, Queue, and Map, and their various implementations like ArrayList, LinkedList, HashSet, TreeSet, PriorityQueue, HashMap, and TreeMap.
- ***These collections are more flexible than arrays, as they can grow and shrink dynamically***. This makes collections more flexible, especially when the volume of data is unknown at the beginning of program execution.
- Additionally, the ***Java Collections Framework supports both homogeneous and heterogeneous data types.***
- Moreover, ***with the introduction of generics in Java 5,*** the Collections Framework became type-safe. This means ***developers can specify the type of objects a collection will hold***, and the compiler will enforce this, reducing the chance of runtime errors caused by incorrect type casting. (e.g., ***List<String> can hold only String objects***), collections without generics (***List<Object> can store different types of objects, like integers, strings, and even custom objects, in the same container***).

Advantage of Collections Framework in Java

•Dynamic Size

Collections can grow or shrink dynamically, unlike arrays which have a fixed size.

•Built-in Data Structures

Provides ready-to-use implementations of common data structures like List, Set, Map, Queue, Stack, etc.

•Code Reusability

Interfaces and standard methods allow for reusable code across different collection types.

•Consistent API Design

Most collection classes follow a consistent method pattern (e.g., add(), remove(), contains(), etc.), making them easy to learn and use.

•Generic Support

Generics ensure type safety by allowing collections to hold only specified types of objects, reducing runtime type errors.

•Utility Methods

The Collections class provides helpful static methods for sorting, searching, reversing, shuffling, and more.

- **Efficient Performance**

  Collection classes are optimized for performance and support faster data operations compared to manual implementations.
- **Saves Development Time**

  Developers don’t need to write complex data structures like linked lists, hash tables, or trees from scratch. The Collections Framework provides ready-made, reliable implementations that can be used directly.
- Built-in collections are well-tested and optimized. Using them reduces the risk of bugs that might occur when manually implementing data structures or logic.
- **Flexible Element Handling**

  List allows duplicates and maintains order.

  Set stores only unique elements.

  Map stores key-value pairs with unique keys.

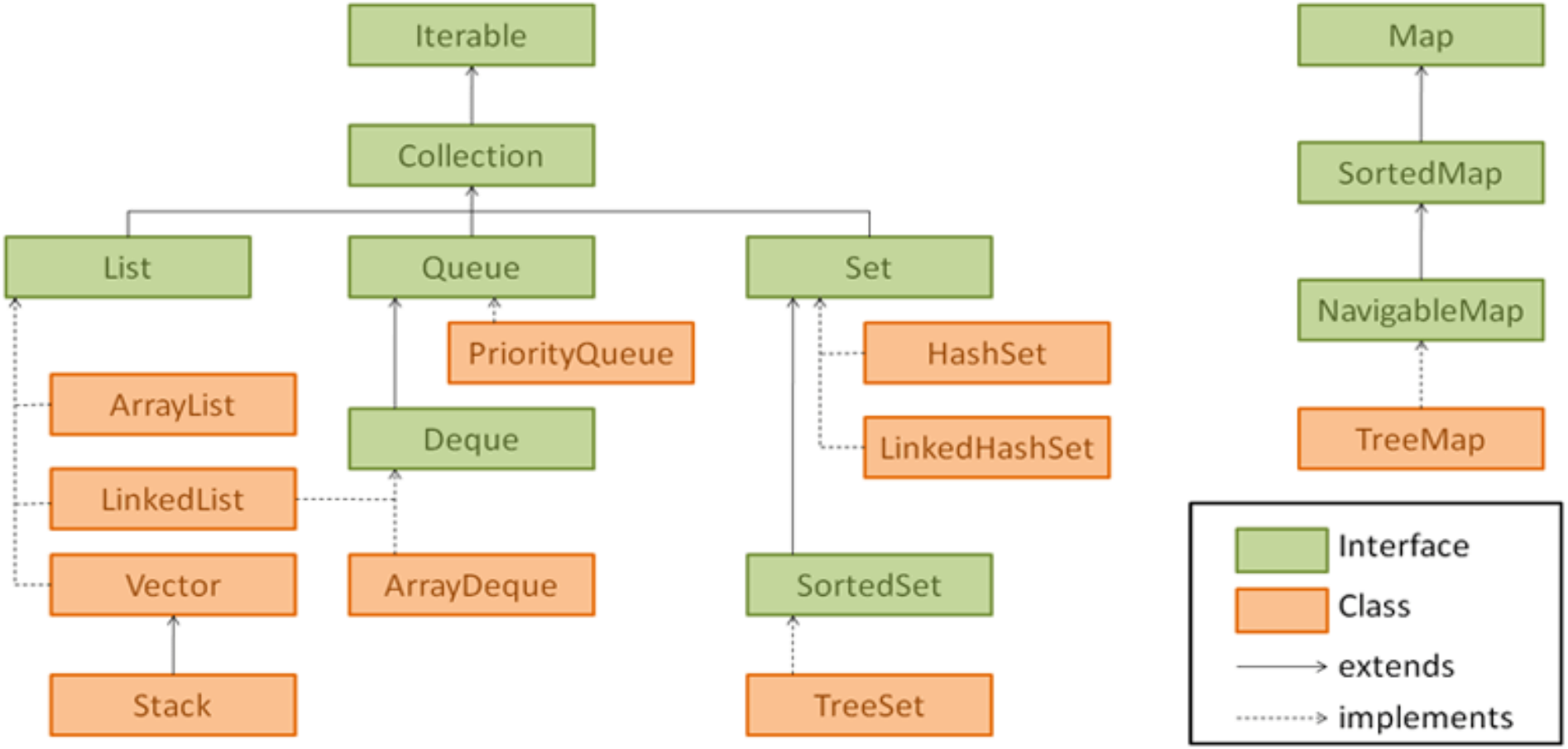
Collection Hierarchy in Java

Collections framework in Java supports two types of containers:

One for storing a collection of elements (objects), that is simply called a collection.

The other, for storing key/value pairs, which is called a map.

- The utility package***, java.util contains all the classes and interfaces that are required by the collection***



The Collection Interfaces

1. Iterable interface

The iterable interface is the root interface for the entire collection framework. The collection interface extends the iterable interface. Therefore, inherently, *all the interfaces and classes implements Iterable interface.*

The main functionality of this interface is to provide an iterator for the collections. It has only one method.

```
public interface Iterable<T> {
    Iterator<T> iterator();
}
```

The iterator() method ***returns an Iterator object that enables you to iterate through the collection's elements one by one.***

Iterator:

Iterator is an interface in the Java Collection Framework, located in the java.util package.

This Iterator provides the **mechanism** to access elements in the collection

It provides methods to **sequentially access elements** of a collection (like ArrayList, HashSet, etc.) one at a time. The interface defines three main methods: hasNext(), next(), and optionally remove() to traverse the collection.

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();  // Optional method
}
```

Methods:

hasNext(): Checks if there are more elements to iterate.

next(): Returns the next element in the iteration. Initially, the iterator object points just before the first element of the collection. Each subsequent call to next() advances the iterator to the next element in the collection.

remove(): Optionally removes the last element returned by next().

How it Works:

- Each collection class like ArrayList, HashSet, and LinkedList has a private inner class (e.g., Itr, HashIterator, ListItr) that implements the Iterator interface.
- This inner class contains the logic for traversing through the elements of that specific collection.
- The **inner class** is responsible for implementing the methods defined by the Iterator interface (hasNext(), next(), and optionally remove()).
- When you call iterator() on a collection, the collection (like ArrayList) returns an **instance of an inner class** that implements the Iterator interface. Using this instance we can call hasNext(), next() methods to traverse the collection.

Example:

```
ArrayList<String> list = new ArrayList<>();
list.add("A");
list.add("B");
list.add("C");
// Getting the iterator object from the list
Iterator<String> iterator = list.iterator();
// Using the iterator methods to traverse the collection
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
```

Explanation:

ArrayList<String> list = new ArrayList<>();

Iterator<String> iterator = list.iterator();

list.iterator()

When you call iterator() on a collection like ArrayList, it returns an instance of an **inner class.** Internally, ArrayList has an **inner class** (e.g., ArrayList.Itr) that implements the Iterator interface.

- This instance is **assigned to an Iterator reference variable** — which is an example of **upcasting**.

  Iterator<String> iterator = **list.iterator()**;

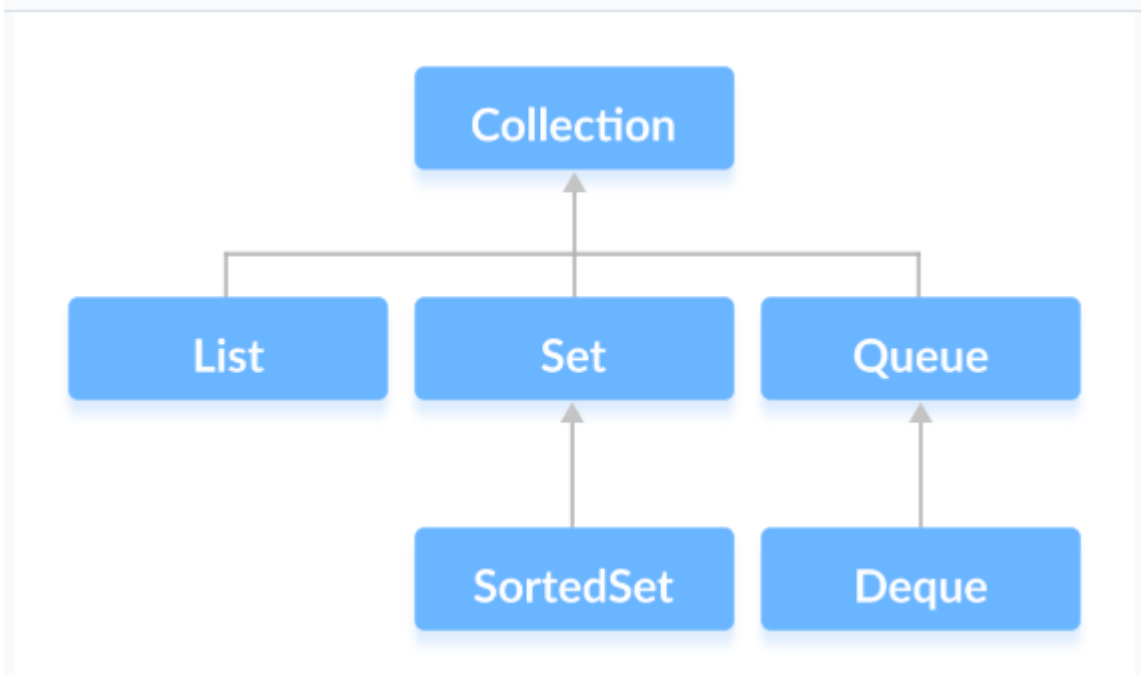
- Upcasting is when a subclass (or a class that implements an interface) object is **assigned to a reference variable of its superclass or interface type**.
- Using this reference(iterator object), you can call the iterator methods to traverse the collection. The iterator object points just before the first element of the collection

  iterator.hasNext();

  iterator.next();

Collection Interface in Java

- The basic interface of the collections framework is the ***Collection interface which is the root interface of all collections in the API*** (Application Programming Interface).
- It is placed at the top of the collection hierarchy in java. ***It provides the basic operations for adding and removing elements in the collection.***
- ***Collection interface extends the Iterable interface.*** The iterable interface has only one method called iterator(). The function of the iterator method is to return the iterator object. Using this iterator object, we can iterate over the elements of the collection.
- ***List, Queue, and Set are three interfaces which extends the Collection interface***. A map is not inherited by Collection interface.



The Collection interface is not directly implemented by any class. However, it is implemented indirectly via its subtypes or subinterfaces like List, Queue, and Set.

For Example:

- The HashSet class implements the Set interface which is a sub interface of the Collection interface.
- The ArrayList class implements the List interface which is a sub interface of the Collection Interface.

The Methods Defined by Collection interface

Method	Description
boolean add(Object obj)	Adds obj to the invoking collection. Returns true if obj was added to the collection. Returns false if obj is already a member of the collection, or if the collection does not allow duplicates.
boolean addAll(Collection c)	Adds all the elements of c to the invoking collection. Returns true if the operation succeeded (i.e., the elements were added). Otherwise, returns false.
void clear()	Removes all elements from the invoking collection.
boolean contains(Object obj)	Returns true if obj is an element of the invoking collection. Otherwise, returns false.
boolean containsAll(Collection c)	Returns true if the invoking collection contains all elements of c. Otherwise, returns false.
boolean equals(Object obj)	Returns true if the invoking collection and obj are equal. Otherwise, returns false.
int hashCode()	Returns the hash code for the invoking collection.

Method	Description
boolean isEmpty()	Returns true if the invoking collection is empty. Otherwise, returns false.
Iterator iterator()	Returns an iterator for the invoking collection.
boolean remove(Object obj)	Removes one instance of obj from the invoking collection. Returns true if the element was removed. Otherwise, returns false.
boolean removeAll(Collection c)	Removes all elements of c from the invoking collection. Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false.
boolean retainAll(Collection c)	Removes all elements from the invoking collection except those in c. Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false.
int size()	Returns the number of elements held in the invoking collection.
Object[] toArray()	Returns an array that contains all the elements stored in the invoking collection. The array elements are copies of the collection elements.

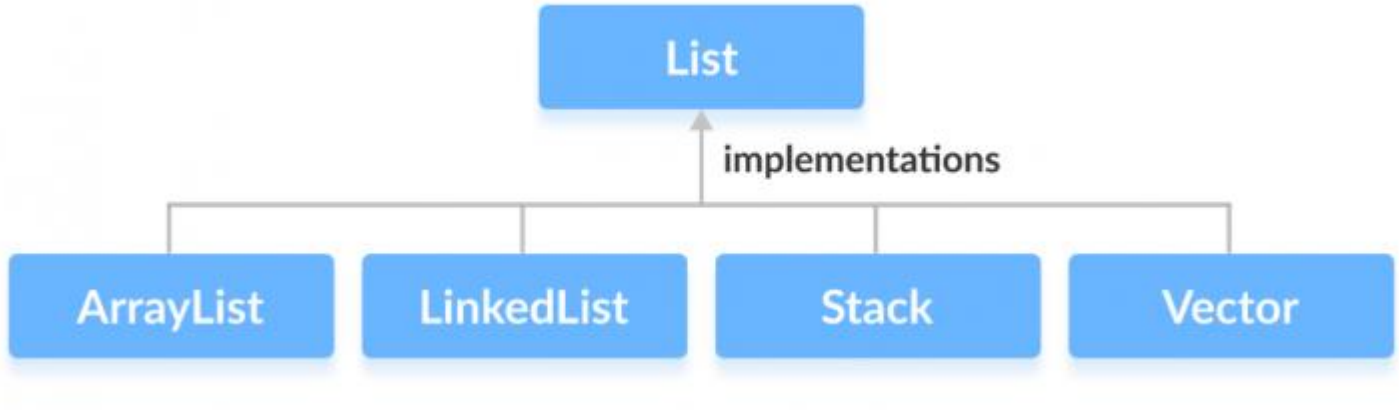
Method	Description
Object[] toArray(Object array[])	Returns an array containing only those collection elements whose type matches that of array. The array elements are copies of the collection elements.

List Interface

➢ List interface represents a ***collection of elements whose elements are arranged sequentially ordered***.

Key Characteristics Of The List Interface

1. **Ordered Collection**: Lists preserve the order of elements, ensuring that elements are stored in the sequence in which they were added.
2. **Duplicates Allowed**: Lists permit duplicate elements. Multiple occurrences of the same element can coexist in a list.
3. **Indexed Access**: Elements can be retrieved and modified using their index positions.
4. **Dynamic Size**: Lists can dynamically grow or shrink as elements are added or removed.
5. ArrayList, Vector, Stack and LinkedList are three concrete subclasses that implement the list interface.



How to Create List in Java?

To create a list in Java, we can use one of its concrete subclasses: ArrayList, LinkedList, Vector and Stack.

List p = new ArrayList();
List q = new LinkedList();
List r = new Vector();
List s = new Stack();

How to Create Generic List Object in Java

After the introduction of ***Generic in Java 1.5, we can restrict the type of object that can be stored in the list***. The general syntax for creating a list of objects with a generic type parameter is as follows:

List<data type> list = new ArrayList<data type>(); // General syntax.

For example:

// Creating a list of objects of String type using ArrayList.

a. List<String> list = new ArrayList<String>();

// Creating a list of objects of Integer type using LinkedList.

b. List<Integer> list = new LinkedList<Integer>();

// Creating a list of objects of String type using LinkedList.

c. **List<String> list1 = new LinkedList<String>();**

// Create a list of objects of type obj using ArrayList class.

d. **List<obj> list = new ArrayList<obj>()**;

For example:

List<Book> list=new ArrayList<Book>(); // Book is the type of object.

2. Starting from Java 1.7, we can use a diamond operator.

a. **List<String> str = new ArrayList<>();**

b. List<Integer> list = new LinkedList<>();

Java List Methods

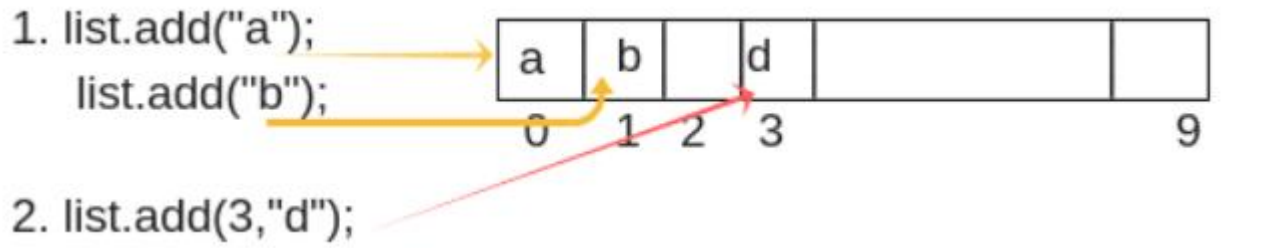
Java List interface provides 10 specific and useful methods in addition to the 15 methods specified by the collection interface that it extends. These methods can be used to initialize a list in java. They are as follows:

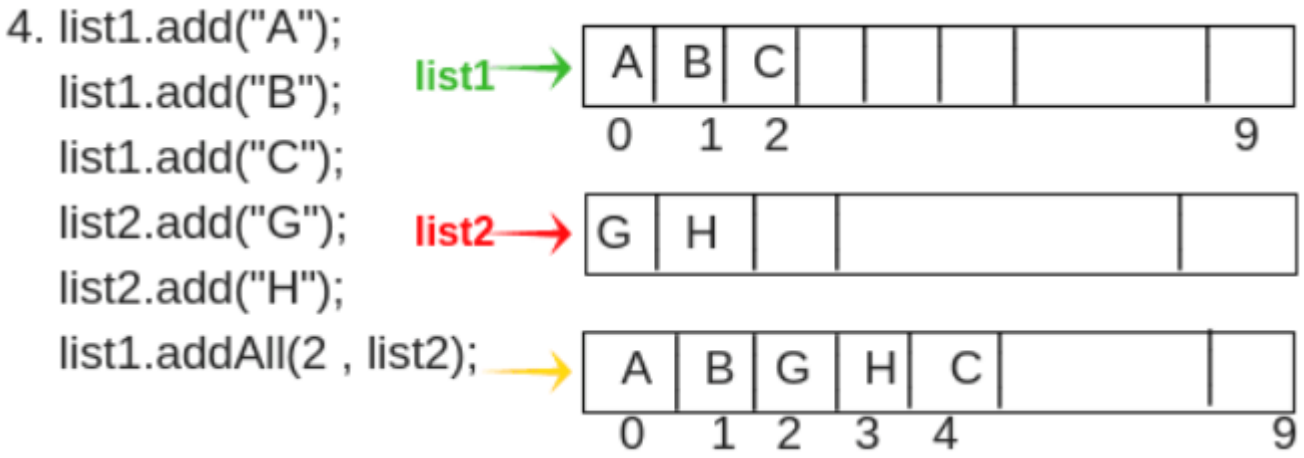
Method	Description
boolean add(E e)	Appends the specified element to the end of the list
void add(int index, E element)	Inserts the element at the specified position
boolean addAll(Collection c)	Appends all elements from the specified collection to the end of the list
boolean addAll(int index, Collection c)	Inserts all elements from the collection starting at the specified index
E get(int index)	Returns the element at the specified position
E set(int index, E element)	Replaces the element at the specified index
E remove(int index)	Removes the element at the specified index
boolean remove(Object o)	Removes the first occurrence of the specified element

int size()	Returns the number of elements in the list
boolean isEmpty()	Returns `true` if the list contains no elements
boolean contains(Object o)	Returns `true` if the list contains the specified element
int indexOf(Object o)	Returns the index of the first occurrence of the specified element
int lastIndexOf(Object o)	Returns the index of the last occurrence of the specified element
void clear()	Removes all elements from the list
List<E> subList(int fromIndex, int toIndex)	Returns a portion of the list between the specified indexes

Example:

List<String> list=new ArrayList<>();





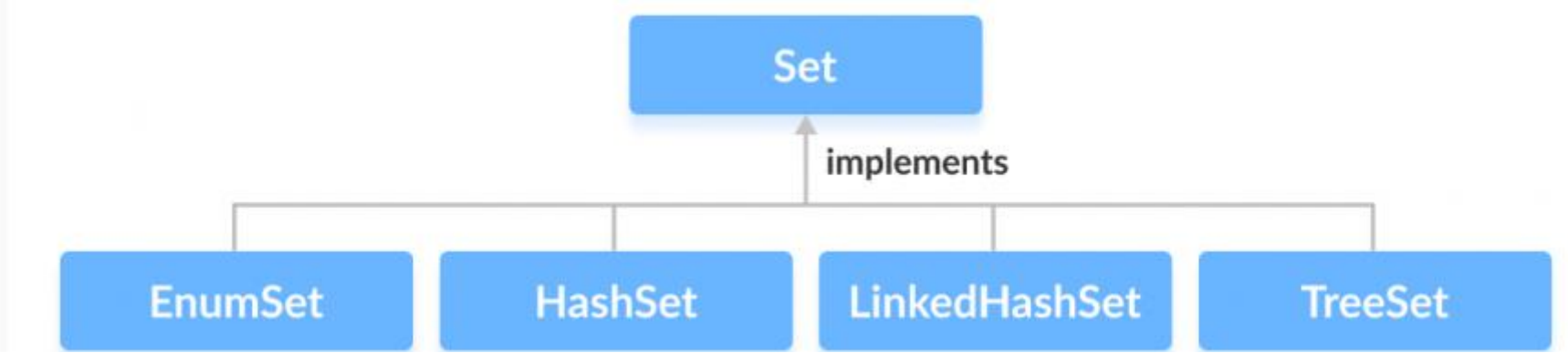
Set Interface

➢ Set ***represents a collection that does not allow duplicate elements***. This is useful when you want to store unique values.

Key Characteristics of Set:

1. **No Duplicates:** The most defining feature of a Set is that doesn't allow duplicates. Each element in a Set is unique. If you try to add an element that is already present in the Set, the operation will not change the Set and typically returns false.
2. **Unordered Collection:** Generally, sets do not guarantee the order of their elements (except for certain implementations like `LinkedHashSet` and `TreeSet`)
3. **Single Null Value:** Most Set implementations allow at least one null element, though certain implementations (like TreeSet) do not allow null elements and will throw a NullPointerException.

- HashSet, LinkedHashSet, TreeSet classes implements the set interface and SortedSet interface extends a Set interface.



- It can be iterated by using Iterator but cannot be iterated using ListIterator.
- Set interface extends the Collection interface. So it inherits all Collection methods (like add(), remove(), contains(), etc.).

Set Interface Operations

The Java Set interface allows us to perform basic mathematical set operations like union, intersection, and subset.

1. **Union:** to get the union of two sets x and y, we can use x.addAll(y).
2. **Intersection:** to get the intersection of two sets x and y, we can use x.retainAll(y).
3. **Subset:** to check if x is a subset of y, we can use y.containsAll(x).

Creating Set Objects

- To create a Set object in Java, you can use one of the classes that implement the Set interface, such as HashSet, LinkedHashSet, or TreeSet.

```
Set<String> set = new HashSet<>();
Set<Integer> set = new LinkedHashSet<>();
Set<String> set = new TreeSet<>();
```

SortedSet Interface in Java

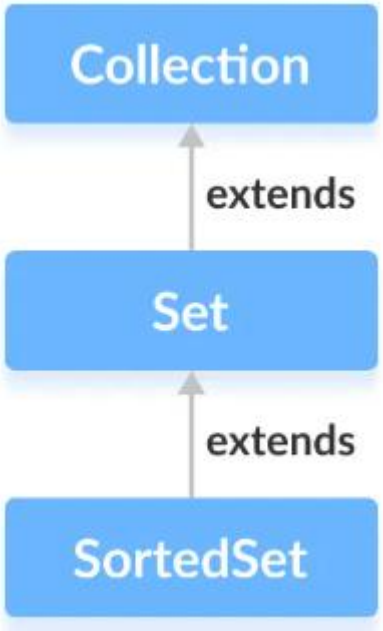
- The SortedSet interface extends Set and represents a **set that maintains its elements in ascending order**. It is part of the java.util package.

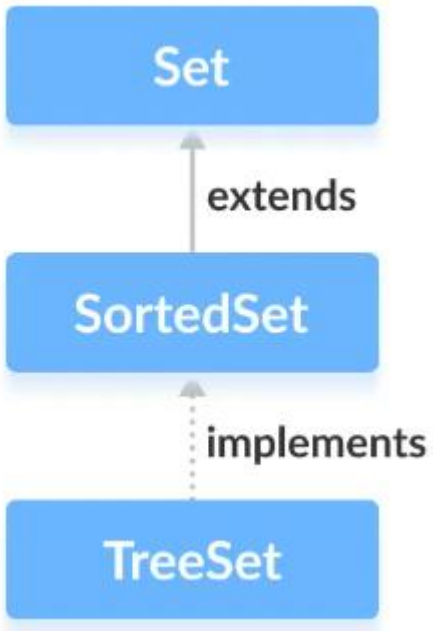
Key Characteristics of SortedSet:

•**Sorted**: Elements are automatically sorted either by natural ordering or using a specified Comparator.

•**No duplicates**: Like all sets, duplicates are not allowed.

- TreeSet implements the sorted interface.



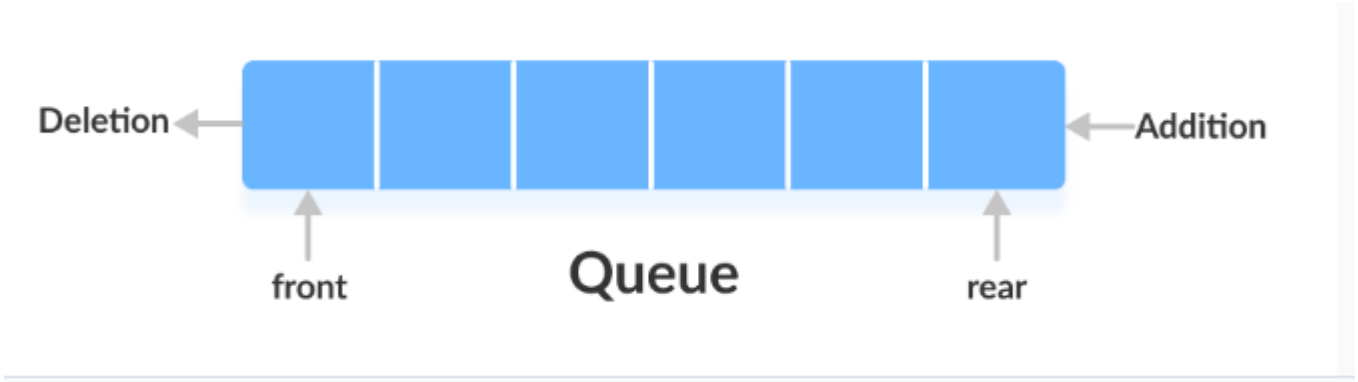


SortedSet Methods

Method	Description
Comparator<? super E> comparator()	Returns the comparator used to order the set, or null if it uses **natural ordering**.
E first()	Returns the **first (lowest)** element in the set.
E last()	Returns the **last (highest)** element in the set.
SortedSet<E> headSet(E toElement)	Returns a **view of the portion** of the set **less than toElement** (exclusive).
SortedSet<E> tailSet(E fromElement)	Returns a view of the set **greater than or equal to fromElement** (inclusive).
SortedSet<E> subSet(E fromElement, E toElement)	Returns a view of the set **from fromElement (inclusive) to toElement (exclusive)**.

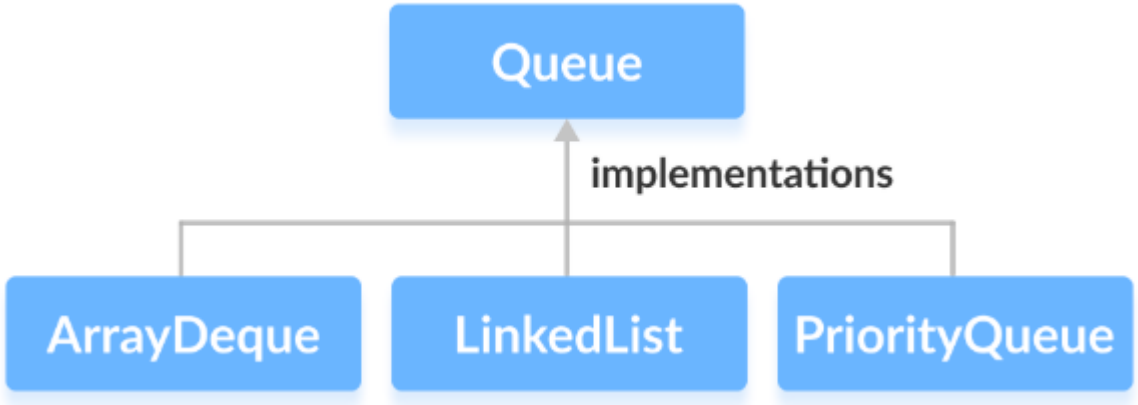
Queue Interface

1. A queue is an ordered of the homogeneous group of elements in which new elements are added at one end(rear) and elements are removed from the other end(front). In queues, elements are stored and accessed in First In, First Out manner

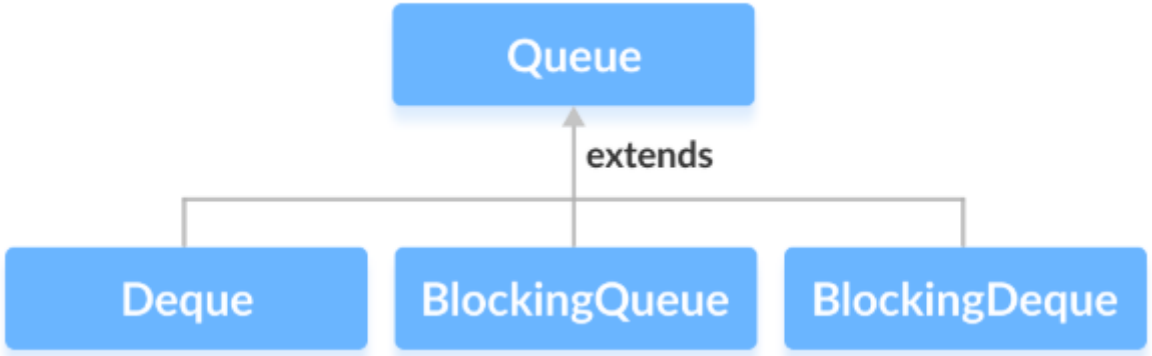


2. This interface represents a special type of list whose elements are removed only from the head.

Classes that Implement Queue



Interfaces that extend Queue



How to use Queue?

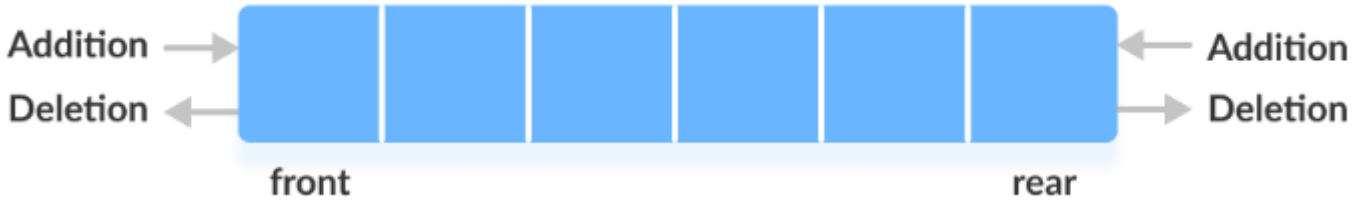
```
// LinkedList implementation of Queue

Queue<String> animal1 = new LinkedList<>();

// Array implementation of Queue

Queue<String> animal2 = new ArrayDeque<>();

// Priority Queue implementation of Queue

Queue<String> animal3 = new PriorityQueue<>();
```

Methods in Queue Interface

Type	Method	Description
Insertion	add(E e)	Inserts the specified element into the queue; If the task is successful, add() returns true, if not it throws an exception.
	offer(E e)	Inserts the specified element into the queue. If the task is successful, offer() returns true, if not it returns false.
Removal	remove()	Removes and returns head; throws NoSuchElementException if empty.
	poll()	Removes and returns head; returns null if empty.
Examine	element()	Returns head without removing; throws NoSuchElementException if empty.
	peek()	Returns head without removing; returns null if empty.

Deque Interface

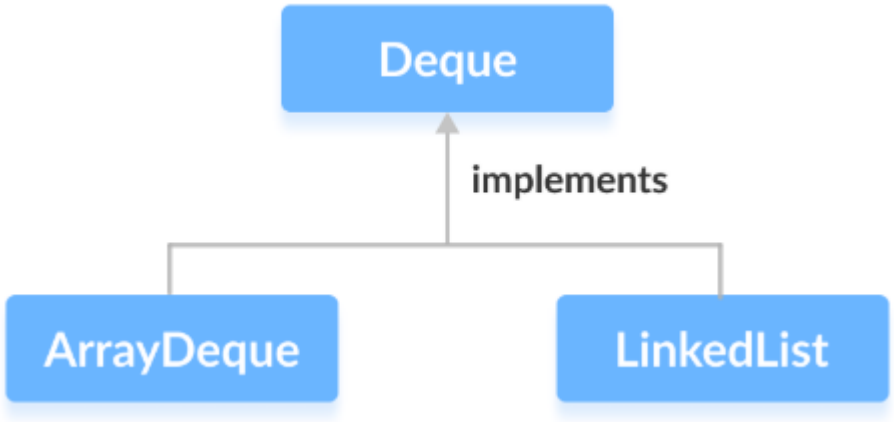
- The Deque interface in Java (short for **Double-Ended Queue**) is part of the **java.util** package and extends the Queue interface.
- Dequeue interface that represents a linear collection that supports element insertion and removal at both ends.



The Deque interface provides more functionality compared to the Queue interface, making it a versatile data structure for various use cases.

Classes that implement Deque

In order to use the functionalities of the Deque interface, we need to use classes that implement it:



How to use Deque?

In Java, we must import the java.util.Deque package to use Deque.

// Array implementation of Deque

Deque<String> animal1 = new ArrayDeque<>();

// LinkedList implementation of Deque

Deque<String> animal2 = new LinkedList<>();

Key Features

The Deque interface, introduced in Java 6, extends the Queue interface and provides the following features:

1.Bidirectional operations: The Deque interface allows elements to be added or removed from both ends, enabling it to function as a queue or a stack, depending on the requirements of the application.

2.Null elements: Similar to most collection interfaces in Java, the Deque interface allows null elements.

3.Capacity-restricted implementations.

Methods of Deque Interface :

Insertion Methods

Method	Description
addFirst(E e)	Inserts element at the **front**. Throws exception if fails.
addLast(E e)	Inserts element at the **end**. Throws exception if fails.
offerFirst(E e)	Inserts element at front. Returns false if fails.
offerLast(E e)	Inserts element at end. Returns false if fails.

Removal Methods

Method	Description
removeFirst()	Removes and returns first element. Throws NoSuchElementException if empty.
removeLast()	Removes and returns last element. Throws NoSuchElementException if empty.
pollFirst()	Removes and returns first element, or null if empty.
pollLast()	Removes and returns last element, or null if empty.

Retrieval (Peek) Methods

Method	Description
getFirst()	Retrieves but doesn't remove first element. Throws exception if empty.
getLast()	Retrieves but doesn't remove last element. Throws exception if empty.
peekFirst()	Retrieves first element, or null if empty.
peekLast()	Retrieves last element, or null if empty.

Stack-Implementation Methods (LIFO)

Method	Description
push(E e)	Adds element to front (like a stack push).
pop()	Removes and returns front element (like a stack pop).
peek()	Retrieves front element (like a stack peek).

Collection Classes in Java

The collections classes implement the collection interfaces. Some of the classes provide full implementations that can be used as it is. Others are abstract that provide basic implementations that can be used to create concrete collections. A brief overview of each concrete collection class is given below.

1. AbstractCollection: It implements most of the collection interface. It is a superclass for all of the concrete collection classes.

2. AbstractList: It extends AbstractCollection and implements most of the List interface.

3. AbstractQueue: It extends AbstractCollection and implements the queue interface.

4. AbstractSequentialList: It extends AbstractList and uses sequential order to access elements.

5. AbstractSet: Extends AbstractCollection and implements most of the set interface.

6. ArrayList: It implements a dynamic array by extending AbstractList.

7. LinkedHashSet: Extends HashSet to allow insertion-order iterations.

8. LinkedList: Implements a linked list by extending AbstractSequentialList.

9. PriorityQueue: Extends AbstractQueue to support a priority-based queue.

10. TreeSet: Extends AbstractSet and implements the SortedSet interface.

ArrayList:

- ArrayList in Java is a ***resizable array that can grow or shrink in the memory whenever needed***. It is dynamically created with an initial capacity.
- It means that ***if the initial capacity of the array is exceeded, a new array with larger capacity is created automatically and all the elements from the current array are copied to the new array.***
- ***Elements in ArrayList are placed according to the zero-based index***. That is the first element will be placed at 0 index and the last element at index (n-1) where n is the size of ArrayList.

Key Features

1. **Resizable:** Unlike arrays, ArrayList can grow and shrink in size dynamically.
2. **Indexed:** Elements can be accessed by their index, similar to arrays.
3. **Duplicate Elements:** Allows duplicate elements.
4. **Null Elements:** Can contain null elements.
5. **Non-Synchronized:** Not synchronized, which means it is not thread-safe unless explicitly synchronized.
6. **Random Access:** ArrayList implements random access because it uses an index-based structure. Therefore, we can get, set, insert, and remove elements of the array list from any arbitrary position.
7. **Implements List Interface:** Provides all methods defined by the List interface.

Constructors

ArrayList has the constructors shown here:

1. ArrayList( )
2. ArrayList(Collection *c*)
3. ArrayList(int *capacity*)

- The first constructor builds an empty array list.
- The second constructor builds an array list that is initialized with the elements of the collection *c*.
- The third constructor builds an array list that has the specified initial *capacity*. The capacity is the size of the underlying array that is used to store the elements. The capacity grows automatically as elements are added to an array list.

Create an Arraylist

```
import java.util.ArrayList; // import the ArrayList class
ArrayList<String> cars = new ArrayList<String>(); // Create an ArrayList object
```

Add Items

```
import java.util.ArrayList;
public class Main {
public static void main(String[] args) {
ArrayList<String> cars = new ArrayList<String>();
cars.add("Volvo");
cars.add("BMW");
cars.add("Ford");
cars.add("Mazda");
System.out.println(cars);    // Printss the Collection
}
}
```

You can also add an item at a specified position by referring to the index number:

```
cars.add(0, "Mazda"); // Insert element at the beginning of the list (0)
```

Access an Item

• To access an element in the ArrayList, use the get() method and refer to the index number:

```
cars.get(0);
```

Change an Item

• To modify an element, use the set() method and refer to the index number:

```
cars.set(0, "Opel");
```

Remove an Item

```
cars.remove(0);
```

To remove all the elements in the ArrayList, use the clear() method:

```
cars.clear();
```

ArrayList Size

• To find out how many elements an ArrayList have, use the size() method:

```
cars.size();
```

TASK

- Create an arraylist
- Now add the elements- C, A ,E ,D,B,F
- Display the size of the list after adding an element
- Add element A2 at position 2
- Display the contents of the arraylist
- Remove ‘F’
- Remove the element added at the 2nd index.
- Display the contents of the arraylist and size of the arraylist after deletion.

```
import java.util.ArrayList;

public class ArrayListOperations {
    public static void main(String[] args) {
        // Create an ArrayList and add elements to it
        ArrayList<String> list = new ArrayList<>();
        list.add("C");
        list.add("A");
        list.add("E");
        list.add("D");
        list.add("B");
        list.add("F");

        // Display the size of the list after adding elements
        System.out.println("Size of the list after adding elements: " + list.size());
```

```
// Add element A2 at position 2
        list.add(2, "A2");

        // Display the contents of the ArrayList
        System.out.println("Contents of the ArrayList after adding A2: " + list);

        // Remove element 'F'
        list.remove("F");

        // Remove element at index 2 (which is now "A2")
        list.remove(2);

        // Display the contents and size of the ArrayList after deletions
        System.out.println("Contents of the ArrayList after deletions: " + list);
        System.out.println("Size of the list after deletions: " + list.size());
    }
}
```

Tasks:

1.Create an empty list to store book titles.

2.Add the following books to the list:

"Java", "Python", "C++", "Data Structures", "Algorithms", "Web Development".

3.Display the list of books.

4.Insert "Operating Systems" at the beginning and "Computer Networks" at the 3rd index.

5.Replace "C++" with "Advanced C++".

6.Remove "Python" from the list by value.

7.Remove the book at index 4.

8.Check if "Machine Learning" is in the list.

9.Retrieve and display the book at index 2.

10.Display the index of "Data Structures".

11.Get the total number of books in the list.

12.Create a new list of books issued and copy the first 3 books from the main list to it.

13.Clear all elements from the issued list.

How do we manually increase or decrease current capacity of ArrayList?

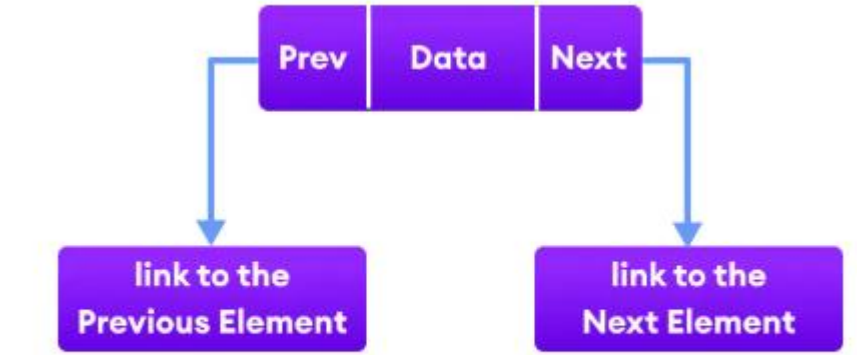
1.ensureCapacity(): This method is used to increase the current capacity of ArrayList. Since the capacity of an array list is automatically increased when we add more elements. But ***to increase manually, ensureCapacity() method of ArrayList class is used.***

2. trimTosize(): The trimTosize() method is used to ***trim the capacity of ArrayList to the current size of ArrayList.***

```
ArrayList<String> list = new ArrayList<String>();
// Here, list can hold 10 elements.(Default initial capacity).
list.ensureCapacity(20);
// Now it can hold 20 elements.
list.trimTosize();
```

LinkedList Class in Java

- The LinkedList class of the Java collections framework provides the functionality of the linked list data structure (doubly linkedlist).
- In Java, the *LinkedList class* is a doubly-linked list implementation of List and Deque interfaces. It also implements all optional list operations and permits all elements (including null).



Java Doubly LinkedList

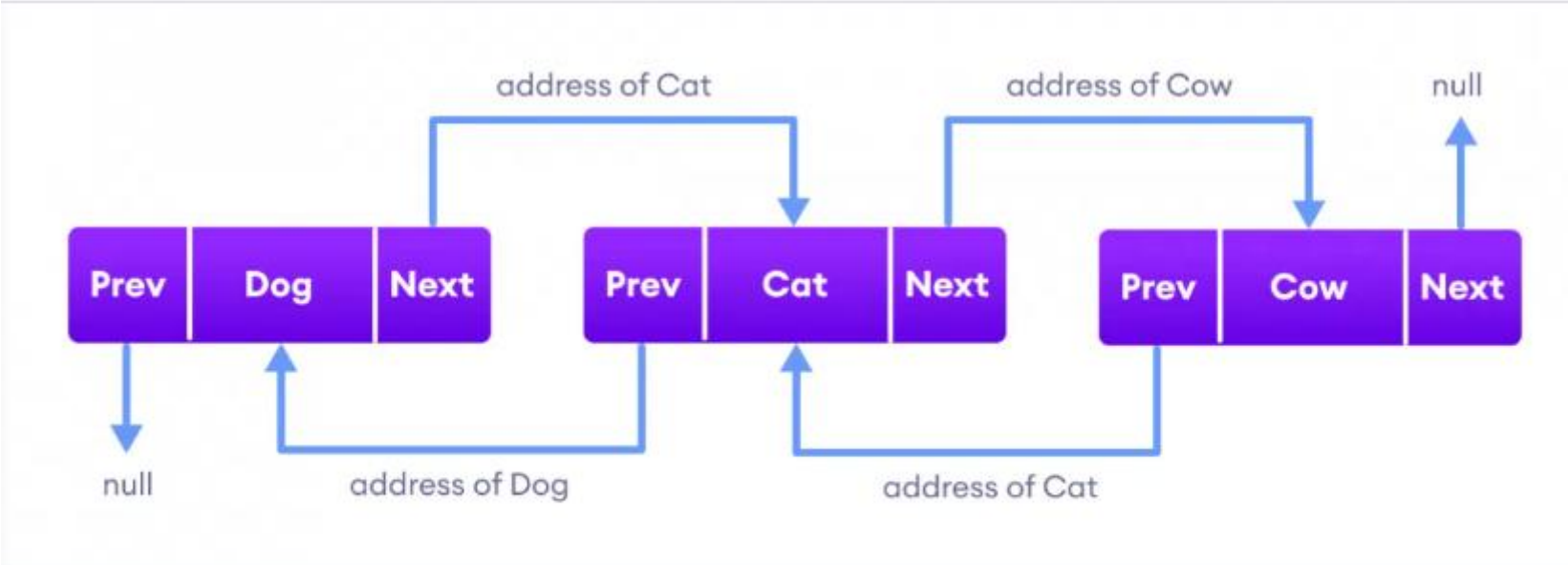
Each element in a linked list is known as a **node**. It consists of 3 fields:

•**Prev** - stores an address of the previous element in the list. It is null for the first element

•**Next** - stores an address of the next element in the list. It is null for the last element

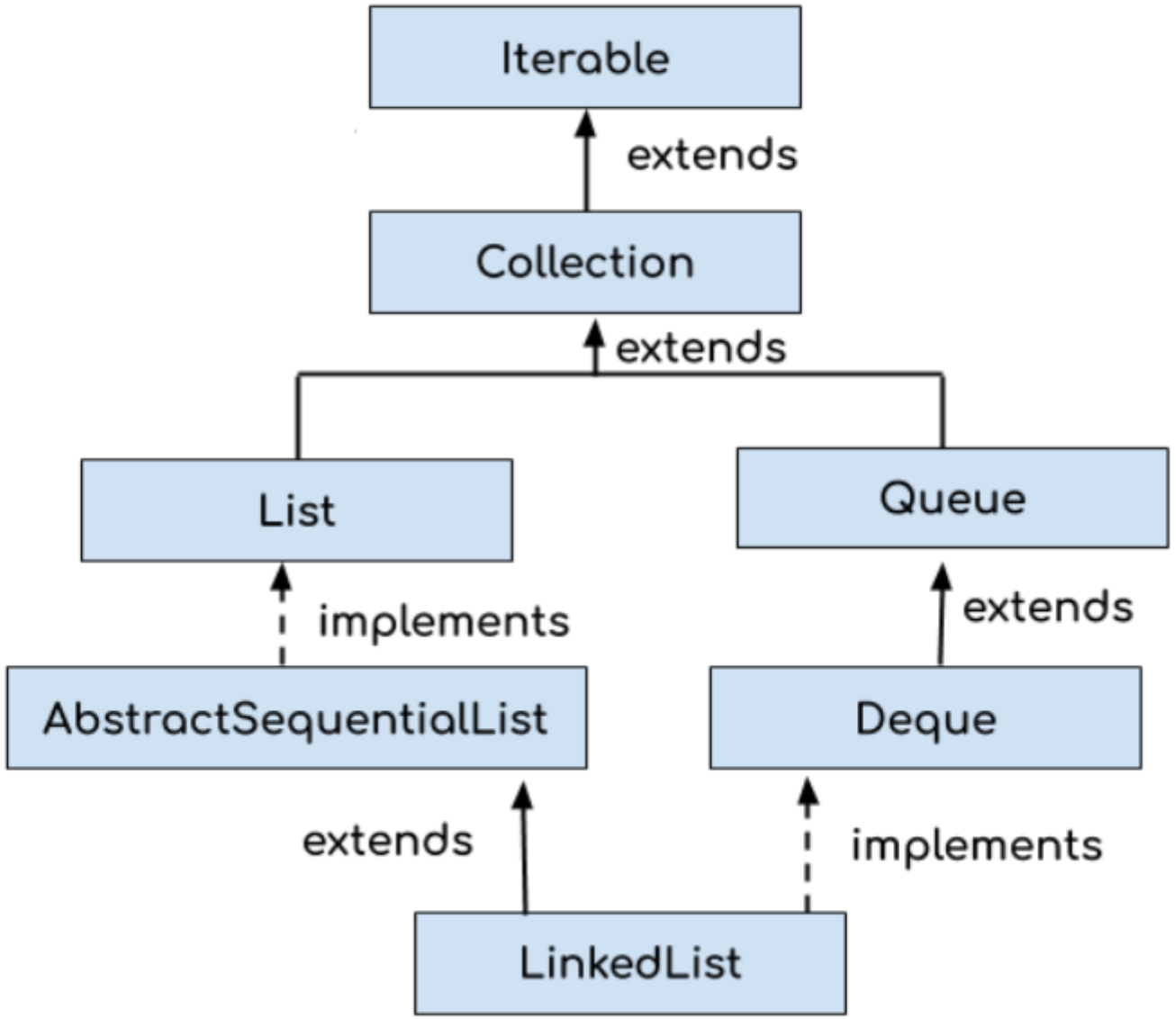
•**Data** - stores the actual data

Elements in linked lists are not stored in sequence. Instead, they are scattered and connected through links (**Prev** and **Next**).



Java LinkedList Implementation

Hierarchy of LinkedList class in Java



Features of LinkedList class

The main features of the Java LinkedList class are as follows:

1. The underlying data structure of LinkedList is a doubly LinkedList data structure. It is another concrete implementation of the List interface like an array list.
2. Implements Queue and Deque interfaces. Therefore, It can also be used as a Queue, Deque, or Stack
3. Java LinkedList class allows storing duplicate elements.
4. Null elements can be added to the linked list.
5. Java LinkedList is not synchronized. So, multiple threads can access the same LinkedList object at the same time. Therefore, It is not thread-safe.
6. Insertion and removal of elements in the LinkedList are fast because, in the linked list, there is no shifting of elements after each adding and removal. The only reference for next and previous elements changed.
7. LinkedList is the best choice if your frequent operation is insertion or deletion in the middle.
8. Java LinkedList does not implement random access interface, so, we can access elements in sequential order only. So, the element cannot be accessed (getting) randomly. To access the given element, we have to traverse from the beginning or ending to reach elements in the LinkedList.
9. We can use ListIterator to iterate elements of the list.
10. The LinkedList can be used as a "stack". It has pop() and push() methods which make it function as a stack.

Constructors of LinkedList Class

LinkedList has the two constructors shown here:

1.LinkedList( ) — Builds an empty linked list

2.LinkedList(Collection c) — Builds a linked list that is initialized with the elements of the collection c.

Creating LinkedList:

We can create an empty linked list object for storing String type elements (objects) as:

```
LinkedList<String> llist = new LinkedList<String>(); // An empty list.
```

There’s a variety of built-in methods that you can use when working with your linked list.

Commonly used LinkedList Methods are:

- **add(E e)** — This method adds elements to the linked list one after the other
- **add(int index, E element)** — This method adds the specified element at the specified position in this list
- **addFirst(E e)** — This method adds the specified element at the beginning of this list
- **addLast(E e)** — This method adds the specified element to the end of this list
- **get(int index):** This method returns the element at the specified position in this list
- **set(int index, E element):** This method replaces the element at the specified position in this list with the specified element
- **remove(int index):** This method removes the element at the specified position in this list
- **removeFirst():** This method removes and returns the first element from this list
- **removeLast():** This method removes and returns the last element from this list

```
package MyPackage;
import java.util.LinkedList;
import java.util.ListIterator;
public class linkedlist {
public static void main(String args[]) {
/* Linked List Declaration */
LinkedList<String> list = new LinkedList<String>();
/*add(String Item) is used for adding* the Items to the
linked list*/
list.add("Java");
list.add("Python");
list.add("Scala");
list.add("Swift");
System.out.println("Linked List Content: " +list);
/*Add Items at specified position*/
list.add(2, "JavaScript");
list.add(3, "Kotlin");
System.out.println("list Content after editing:" +list);
/*Add First and Last Item*/
list.addFirst("First Course");
list.addLast("Last Course");
System.out.println("list Content after add:" +list);
/*Get and set Items in the list*/
Object firstvar = list.get(0);
System.out.println("First Item: " +firstvar);
list.set(0, "Java9");
System.out.println("list Content after updating first
Item: " +list);
```

```
/* Remove from a position*/
list.remove(1);
list.remove(2);
System.out.println("LinkedList after deletion of Item in 2nd and 3rd position " +list);
/*Remove first and last Item*/
list.removeFirst();
list.removeLast();
System.out.println("Final Content after removing first and last Item: "+list);
/*Iterating the linked list*/
ListIterator<String> itrator = l_list.listIterator();
System.out.println("List displayed using iterator:");
while (itrator.hasNext())
 {
System.out.println(itrator.next());
}
}
}
```

What is best case scenario to use LinkedList in Java application?

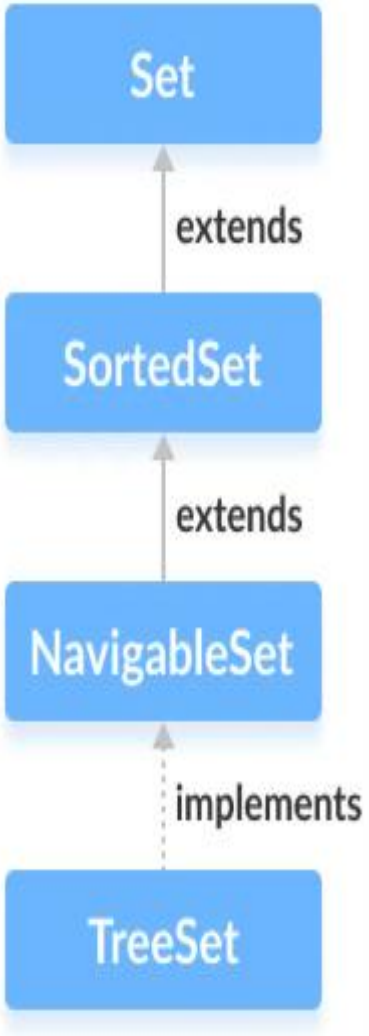
- Java LinkedList is the best choice to use when your frequent operation is adding or removing elements in the middle of the list because the adding and removing of elements in the linked list is faster as compared to ArrayList.
- Let's take a realtime scenario to understand this concept.
- Suppose there are 100 elements in the ArrayList. If we remove the 50th element from the ArrayList, 51st element will go to 50th position, 52nd element to 51st position, and similarly for other elements that will consume a lot of time for shifting. Due to which the manipulation will be slow in ArrayList.
- But in the case of linked list, if we remove 50^{th} element from the linked list, no shifting of elements will take place after removal. Only the reference of the next and previous node will change.
- Moreover, LinkedList can be used when we need a stack (LIFO) or queue (FIFO) data structure by allowing duplicates.

What is worst case scenario to use LinkedList?

- Java LinkedList is the worst choice to use when your frequent operation is retrieval (getting) of elements from the linked list because retrieval of elements is very slow in the LinkedList as compared to ArrayList.
- Since LinkedList does not implement Random Access Interface. Therefore, an element cannot be accessed (getting) randomly. We will have to traverse from the beginning or ending to reach elements in the linked list.
- ArrayList is the best choice to use for getting elements from the list because ArrayList implements random access interface. So, we can get an element from the array list very fast from any arbitrary position

TreeSet in Java

- A **TreeSet in Java** is another important implementation of the Set interface.
- Java TreeSet implements SortedSet interface. It is a collection for storing a set of unique elements (objects) according to their natural ordering.
- It creates a sorted collection that uses a tree structure for the storage of elements or objects. In simple words, elements are kept in sorted, ascending order in the tree set.
- In Java TreeSet, access and retrieval of elements are quite fast because of using tree structure. Therefore, TreeSet is an excellent choice for quick and fast access to large amounts of sorted data. The only restriction with using tree set is that we cannot add duplicate elements in the tree set.



Features of TreeSet class in Java

1. TreeSet contains unique elements similar to the Java HashSet. It does not allow the addition of a duplicate element.
2. The access and retrieval times are quite fast.
3. TreeSet does not allow inserting null element.
4. TreeSet class is non-synchronized. That means it is not thread-safe.
5. TreeSet maintains the ascending order. When we add elements into the collection in any order, the values are automatically presented in sorted, ascending order.
6. Java TreeSet internally uses a TreeMap for storing elements.

How to Create TreeSet in Java?

TreeSet has the following constructors. We can create a TreeSet instance by using one of its four constructors.

1. **TreeSet( ):** This default constructor creates an empty TreeSet that will be sorted in ascending order according to the natural order of its elements.

2. **TreeSet(Collection c):** It creates a tree set and adds the elements from a collection c according to the natural ordering of its elements. All the elements added into the new set must implement Comparable interface.

3. **TreeSet(Comparator comp):** This constructor creates an empty tree set that will be sorted according to the comparator specified by comp.

   All elements in the tree set are compared mutually by the specified comparator. A comparator is an interface that is used to implement the ordering of data.

4. **TreeSet(SortedSet s):** This constructor creates a tree set that contains the elements of sorted set s.

How to Sort TreeSet in Java | Ordering of Elements in TreeSet

A TreeSet in Java determines the order of elements in either of two ways:

1. A tree set sorts the natural ordering of elements when it implements java.lang.Comparable interface. The ordering produced by comparable interface is called natural ordering. The syntax is given below:

```
public interface Comparable {

        public int compareTo(Object o); // Abstract method.

}
```

- The compareTo() method of this interface is implemented by TreeSet class to compare the current element with element passed in as a parameter for the order.
- If the element argument is less than the current element, the method returns +ve integer, zero if they are equal, or a -ve integer if element argument is greater.

How to Sort TreeSet in Java | Ordering of Elements in TreeSet: There are two ways:

1. TreeSet with Comparable (Natural Ordering)

- A tree set sorts the natural ordering of elements when it implements java.lang.Comparable interface. The ordering produced by comparable interface is called natural ordering.
- When a TreeSet implement the Comparable interface, the elements are sorted based on the **natural ordering** defined by the compareTo() method.**compareTo()** defines how objects of the class should be compared by default.
- In this case, all instances of the TreeSet will sort using the same sorting logic (the one defined in compareTo()).
- The syntax is given below:

```
public interface Comparable {
        public int compareTo(Object o); // Abstract method.
}
```

- The compareTo() method of this interface is implemented by TreeSet class to compare the current element with element passed in as a parameter for the order.
- If the element argument is less than the current element, the method returns +ve integer, zero if they are equal, or a -ve integer if element argument is greater.

```
import java.util.TreeSet;
class Person implements Comparable<Person> {
    String name;
    int age;
Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    // Implement compareTo for natural ordering (by age)
    @Override
    public int compareTo(Person other) {
        return Integer.compare(this.age, other.age);
        // Ascending order by age
    }
    // Override toString for easy printing
    @Override
    public String toString() {
        return name + " (" + age + ")";
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        TreeSet<Person> people = new TreeSet<>();

        people.add(new Person("Alice", 30));
        people.add(new Person("Bob", 25));
        people.add(new Person("Charlie", 35));

        // Print TreeSet
        System.out.println("People sorted by age (natural
ordering):");
        for (Person p : people) {
            System.out.println(p);
        }
    }
}
```

2. TreeSet with Comparator (Custom Sorting)

TreeSet in Java also determines the order of elements by implementing the Comparator interface. This technique is used when TreeSet class needs to impose a different sorting algorithm regardless of the natural ordering of elements.

•Using a Comparator allows you to implement different sorting strategies for different usecases, providing **flexibility**.

Scenario: Employee Sorting Based on Different Criteria

Suppose an Employee class has: name (String), age (int), salary (double), experience (int in years)

Use Cases

1. Sort employees by salary

2. Sort employees by age

3. Sort employees by experience

You can achieve this by:

•Creating different Comparator implementations i.e. If you want to sort the Employee in multiple ways (e.g., by salary in one case, by age in another), you need different Comparator implementations.

•Using them with different TreeSet objects as needed

The Comparator's **compare()** method defines how two objects should be compared according to your custom logic.

Syntax:

```
public interface Comparator {

        public int Compare(Element e1, Element e2); // Abstract method.

}
```

It compares **two different objects**, e1 and e2.

Return Value	Meaning
0	e1 is equal to e2
< 0	e1 is less than e2 (comes first)
> 0	e1 is greater than e2

```
import java.util.*;

// Employee class
class Employee {
    String name;
    int age;
    double salary;
    int exp;

 Employee(String name, int age, double salary, int exp) {
        this.name = name;
        this.age = age;
        this.salary = salary;
        this.exp = exp;
    }

    public String toString() {
        return  name + " - Age: " + age + ", Salary: " + salary + ",
Exp: " + exp;
    }
}
```

```
// Comparator to sort by Salary
class SalaryComparator implements Comparator<Employee> {
    public int compare(Employee e1, Employee e2) {
        return Double.compare(e1.salary, e2.salary);
    }
}

// Comparator to sort by Age
class AgeComparator implements Comparator<Employee> {
    public int compare(Employee e1, Employee e2) {
        return Integer.compare(e1.age, e2.age);
    }
}

// Comparator to sort by Experience
class ExperienceComparator implements
Comparator<Employee> {
    public int compare(Employee e1, Employee e2) {
        return Integer.compare(e1.experience, e2.experience);
    }
}
```

```
public class TreeSetComparatorDemo {

    public static void main(String[] args) {

        Employee e1 = new Employee("Alice", 30, 70000, 5);

        Employee e2 = new Employee("Bob", 28, 65000, 4);

        Employee e3 = new Employee("Charlie", 35, 80000, 8);

TreeSet<Employee> salarySet = new TreeSet<>(new
SalaryComparator());

        salarySet.add(e1);

        salarySet.add(e2);

        salarySet.add(e3);

        System.out.println("Sorted by Salary:");

        System.out.println(salarySet);
```

```
 // Sort by Age
        TreeSet<Employee> ageSet = new TreeSet<>(new
AgeComparator());


        ageSet.add(e1);
        ageSet.add(e2);
        ageSet.add(e3);
        System.out.println("\nSorted by Age:");

        System.out.println(ageSet);


        // Sort by Experience


        TreeSet<Employee> expSet = new TreeSet<>(new
ExperienceComparator());


        expSet.add(e1);
        expSet.add(e2);
        expSet.add(e3);
        System.out.println("\nSorted by Experience:");
        System.out.println(expSet);


    }
}
```

ArrayDeque in Java

Java ArrayDeque class implements double-ended queue interface to support the addition of elements from both sides of queue. It also implements queue interface to support the first in first out data structure.

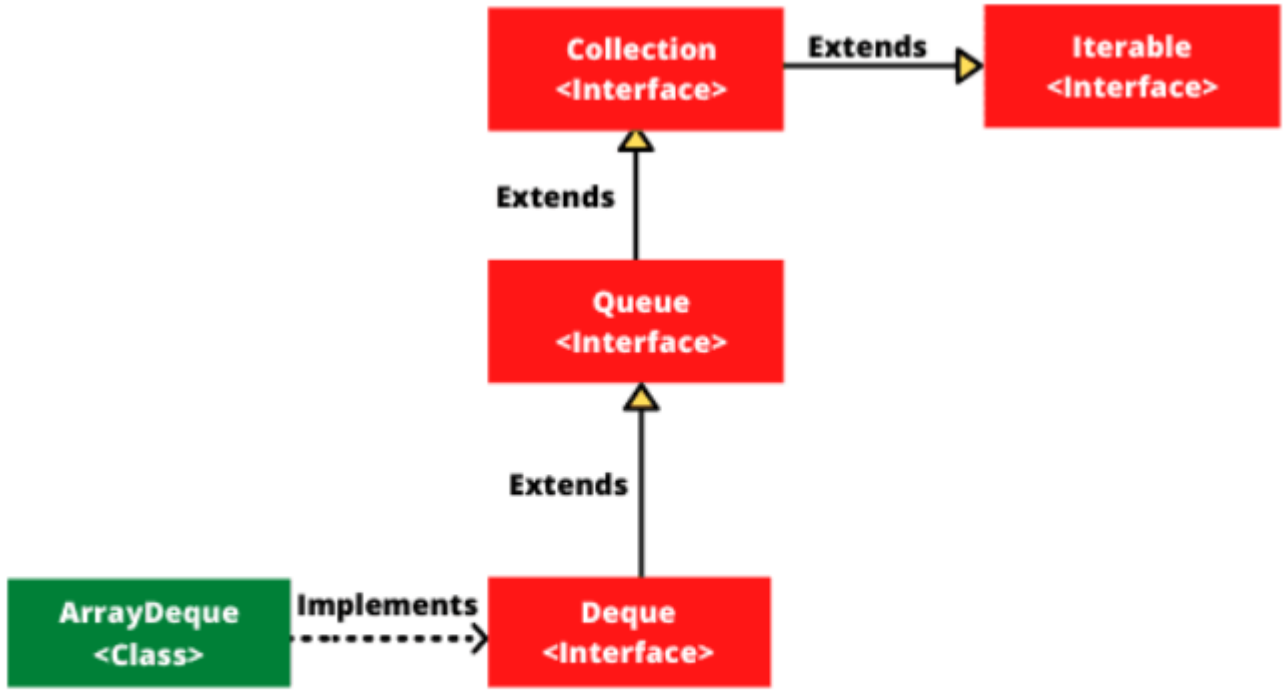


Fig: Hierarchy diagram of ArrayDeque in Java

Features of ArrayDeque

There are several important features of array deck that should be kept in mind. They are as:

1. Java ArrayDeque class provides a resizable array implementation of Deque interface.

2. It has no capacity restrictions. So, it can grow according to the need to handle elements added to the collection.

3. Array deque is not synchronized. That means it is not thread-safe. Multiple threads can access the same ArrayDeque object at the same time.

4. Null elements are restricted in the ArrayDeque.

5. ArrayDeque class performs faster operations than Stack when used as a stack.

6. ArrayDeque class performs faster operations than LinkedList when used as a queue.

Constructors of ArrayDeque Class in Java

1. ArrayDeque(): This constructor creates an empty array deque with starting capacity of 16 elements. The general syntax to create array deque object is as follows:

```
ArrayDeque<E> dq = new ArrayDeque<E>();
```

2.ArrayDeque(int numElements): This constructor creates an empty array deque with the specified initial capacity sufficient to hold elements.

```
ArrayDeque<String> dq = new ArrayDeque<String>(5);
```

3. ArrayDeque(Collection c): This constructor creates an array deque that is initialized with elements of collection c. If c contains null reference then NullPointerException will be thrown. The general syntax to create ArrayDeque instance with the specified collection is given below:

```
ArrayDeque<E> dq = new ArrayDeque<E>(Collection c);
```

Methods of ArrayDeque in Java

ArrayDeque in Java adds no methods of its own. All methods are inherited by Deque, Queue, and Collection interface. The important methods are given below:

1. boolean add(Object o): It inserts the specified element to the end of deque.

2. void addFirst(Object o): It inserts the specified element to the front of the deque.

3. void addLast(Object o): It inserts the specified element to the last of deque.

4. void clear(): This method is used to remove all elements from deque.

5. Object clone(): It is used to get a copy of ArrayDeque instance.

6. boolean contains(Object element): It returns true if the deque contains the specified element.

7. Object element(): This method is used to retrieve an element from the head of deque. It does not remove element.

8. Object getFirst(): This method is used to retrieve the first element of deque. It does not remove element.

9. Object getLast(): This method is used to retrieve the last element of deque. It does not remove element.

10. boolean isEmpty(): The isEmpty() method returns true if deque has no elements. 11. boolean offer(Object element): It inserts the specified element at the end of deque.

11. boolean offerFirst(Object element): It inserts the specified element at the front of deque.

12. boolean offerLast(Object element): It inserts the specified element at the end of deque.

13. Object peek(): The peek() method retrieves element from the head of deque but does not remove. If the deque is empty, it returns null element.

14. Object peekFirst(): The peekFirst() method retrieves the first element from the head of deque but does not remove it. It returns null element if the deque is empty.

15. Object peekLast(): The peekLast() method retrieves the last element from deque but does not remove it. It returns null element if the deque is empty.

16. Object poll(): The poll() method retrieves and removes the element from the head of deque. It returns null element if the deque is empty.

17. Object pollFirst(): The pollFirst() method retrieves and removes the first element of deque. It returns null element if the deque is empty.

18. Object pollLast(): The pollLast() method retrieves and removes the last element of deque. It returns null element if the deque is empty.

19. Object pop(): The pop() method is used to pop an element from the stack represented by the deque.

20. Object push(Object element): The push() method is used to push an element from the stack represented by the deque.

21. Object remove(): The remove() method is used to retrieve and remove an element from the deque.

22. boolean removeFirst(): This method is used to retrieve and remove the first element from the deque.

23. boolean removeFirstOccurrence(Object o): This method removes the last occurrence of the specified element from the deque.

24. int size(): It returns the number of elements in the deque.

25. Object toArray(): It returns an array that contains all elements in the deque in the proper sequence from first to last element.

26. Iterator iterator(): It returns an iterator over elements from the deque.

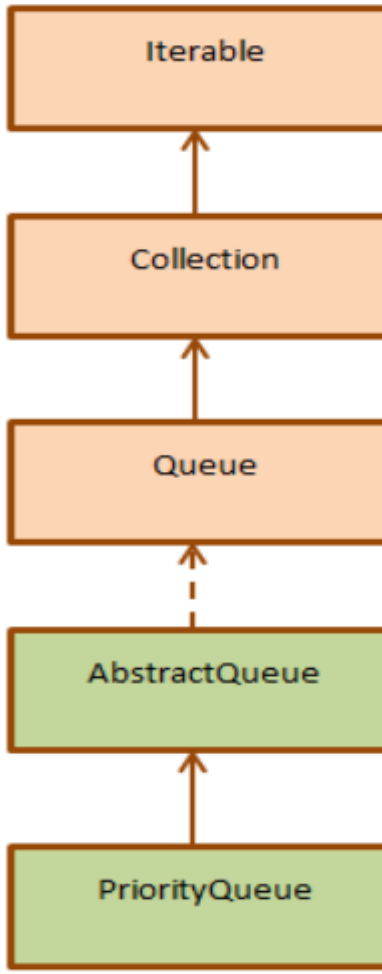
27. Iterator descendingIterator(): It returns an iterator over elements in reverse sequential order from the deque.

PriorityQueue Class

• PriorityQueue is also class that is defined in the collection framework that gives us a way for processing the objects on the basis of priority.

It is already described that the insertion and deletion of objects follows FIFO pattern in the Java queue.

However, sometimes the elements of the queue are needed to be processed according to the priority, that's where a PriorityQueue comes into action.



Real- World Scenario Example:

In a software development company, reported issues (bugs or feature requests) have different priorities:

High (app crash, security vulnerability)

Medium (performance lag)

Low (UI alignment)

- priority (1 = High, 2 = Medium, 3 = Low)
- Developers must resolve issues based on **priority**, not order of arrival. PriorityQueue ensures that **issues are addressed first** .

PriorityQueue Class Declaration

• The declaration for java.util.PriorityQueue class is as follows:

public class PriorityQueue<E> **extends** AbstractQueue<E> i**mplements** Serializable

Feature of the PriorityQueue:

- PriorityQueue does not allow null values.
- For non-comparable objects, we cannot create a priority queue.
- PriorityQueue inherits from the classes like AbstractQueue, AbstractCollection, Collection, and Object.
- The head or front of the queue contains the least element as per the natural ordering.
- Priority Queue implementation is not thread-safe. Thus if we desire synchronized access, we should use the PriorityBlockingQueue.

Method	Method Prototype	Description
Add	boolean add(E e)	Adds element e to the PriorityQueue.
Clear	void clear()	Clears the PriorityQueue by deleting all the elements.
comparator	Comparatorcomparator()	Returns a custom comparator used for the ordering of elements in the Queue.
contains	boolean contains(Object o)	Checks if the PriorityQueue contains the given element o. Returns true if yes.
iterator	Iterator< E >iterator()	Method to get an iterator for the given PriorityQueue.
offer	boolean offer(E e)	Insert given element e to the PriorityQueue.
peek	E peek()	Returns the head of the queue without deleting the element.
poll	E poll()	Removes and returns the head of the queue. Returns null if the queue is empty.
remove	boolean remove(Object o)	Removes an instance of a given element o if it is present in the queue.
size	int size()	Returns the size or number of elements in this PriorityQueue.
toArray	Object[] toArray()	Returns an array representation of the given PriorityQueue.
toArray	T[] toArray(T[] a)	Returns an array representation for the given Priority Queue with the same runtime type as the specified array a.

Constructors:

Constructor Prototype	Description
PriorityQueue()	Creates an empty priority queue with **default initial capacity (11)** and orders elements according to their **natural ordering** (i.e., the element class must implement Comparable).
PriorityQueue(Collection< ? extends E > c)	Creates a PriorityQueue object with initial elements from given collection c.
PriorityQueue(int initialCapacity)	Creates a PriorityQueue object with the given 'initialCapacity'. Elements are ordered as per natural ordering.
PriorityQueue(int initialCapacity, Comparator < ? super E > comparator)	Creates a PriorityQueue object with the given 'initialCapacity'. The elements are ordered according to the given comparator.

How is priority assigned?

- **When new elements are inserted into the *PriorityQueue*, they are ordered** either by their **natural ordering** (if they implement Comparable) or **by a custom comparator** provided during the creation of the PriorityQueue.

1. Natural Ordering

• If the elements (E) stored in the PriorityQueue implement the Comparable interface, the priority queue uses the compareTo() method to determine the order of elements.

The compareTo method establishes the natural ordering of the elements. For example, if you have a PriorityQueue of Integer objects, they are ordered numerically from smallest to largest.

PriorityQueue<Integer> pq = new PriorityQueue<>();

Creates an empty priority queue with **default initial capacity (11)** and orders elements according to their **natural ordering** (i.e., the element class must implement Comparable). This creates a **min-heap** where: The **smallest element** is always at the front.

Example:

```
import java.util.PriorityQueue;
public class NaturalOrderExample {
    public static void main(String[] args) {
        PriorityQueue<Integer> pq = new PriorityQueue<>();// Natural Ordering

        pq.add(40);
        pq.add(10);
        pq.add(30);
        pq.add(20);

        while (!pq.isEmpty()) {
            System.out.println(pq.poll()); // prints in ascending order
        }
    }
}
```

2. Custom Comparator

PriorityQueue elements are naturally ordered.

• Alternatively, you can specify a custom comparator when creating the PriorityQueue to define a non-natural ordering of elements.

This is done by providing an instance of Comparator<E> in the constructor of PriorityQueue.

Syntax:

PriorityQueue<E> pq = new PriorityQueue<>(Comparator<? super E> comparator);

Example 1:

PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Comparator.reverseOrder());

Comparator.reverseOrder():

•This is a **static method** from the Comparator interface. It returns a **comparator** that reverses the natural ordering of elements.

•For integers, it turns ascending order (1, 2, 3) into descending (3, 2, 1).

Example 2:

we can also create our own comparator class that implements the Comparator interface

With Initial Capacity and Custom Comparator:

PriorityQueue(int initialCapacity, Comparator < ? super E > comparator)

```
import java.util.PriorityQueue;
import java.util.Comparator;
class Main {
    public static void main(String[] args) {
        // Creating a priority queue with custom comparator (descending order)
        PriorityQueue<Integer> numbers = new PriorityQueue<>(new CustomComparator());
        numbers.add(4);
        numbers.add(2);
        numbers.add(1);
        numbers.add(3);
        System.out.println("PriorityQueue (Max-Heap): " + numbers);
        // Printing elements in priority order
        System.out.print("Polled in order: ");
        while (!numbers.isEmpty()) {
            System.out.print(numbers.poll() + " ");
        } }
}
class CustomComparator implements Comparator<Integer> {
    @Override
    public int compare(Integer number1, Integer number2) {
        return number2.compareTo(number1);  // Reverse the order
    }
}
```

Accessing Collections via an Iterator

- Often, you will want to cycle through the elements in a collection. For example, you might want to display each element.
- By far, the easiest way to do this is to employ an *iterator,* an object that implements either the **Iterator** or the **ListIterator** interface.
- **Iterator** enables you to cycle through a collection, obtaining or removing elements.
- **ListIterator** extends **Iterator** to allow bidirectional traversal of a list, and the modification of elements.

1. Using an Iterator

• Each of the collection classes provides an **iterator( )** method that returns an iterator to the start of the collection. By using this iterator object, you can access each element in the collection, one element at a time.

• In general, to use an iterator to cycle through the contents of a collection, follow these steps:

1. Obtain an iterator to the start of the collection by calling the collection's **iterator( )** method.

2. Set up a loop that makes a call to **hasNext( )**. Have the loop iterate as long as **hasNext( )** returns **true**.

3. Within the loop, obtain each element by calling **next( )**.

```
public interface Iterator<E> {

   boolean hasNext();

   E next();

   void remove();  // Optional method

}
```

Method	Description
boolean hasNext( )	Returns **true** if there are more elements. Otherwise, returns **false**.
Object next( )	Returns the next element. Throws **NoSuchElementException** if there is not a next element.
void remove( )	Removes the current element. Throws **IllegalStateException** if an attempt is made to call **remove( )** that is not preceded by a call to **next( )**.

Table 15-4. *The Methods Declared by* Iterator

Example:

```
import java.util.ArrayList;
import java.util.Iterator;
public class IteratorExample {
    public static void main(String[] args) {
        // Creating an ArrayList of Strings
        ArrayList<String> list = new ArrayList<>();
        list.add("A");
        list.add("B");
        list.add("C");

        // Getting the iterator object from the list
        Iterator<String> iterator = list.iterator();

        // Using the iterator to traverse the collection
        System.out.println("Elements in the list:");
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
}
```

2. ListIterator

For collections that implement **List**, you can also obtain an iterator by calling **ListIterator**.

As explained, a list iterator gives you the ability to access the collection in either the forward or backward direction and lets you modify an element. Otherwise, **ListIterator** is used just like **Iterator**.

Method	Description
boolean hasNext()	Returns true if there are more elements when moving forward.
E next()	Returns the next element in the list and moves the cursor forward.
boolean hasPrevious()	Returns true if there are elements when moving backward.
E previous()	Returns the previous element and moves the cursor backward.
int nextIndex()	Returns the index of the next element.
int previousIndex()	Returns the index of the previous element.
void remove()	Removes the last element returned by next() or previous().
void set(E e)	Replaces the last element returned with the specified element.
void add(E e)	Inserts the specified element into the list at the current position.

```
import java.util.*;
public class ListIteratorDemo {
    public static void main(String[] args) {
        List<String> fruits = new ArrayList<>(Arrays.asList("Apple", "Banana", "Cherry"));
        // Create a ListIterator
        ListIterator<String> iterator = fruits.listIterator();
        System.out.println("Forward Traversal:");
        while (iterator.hasNext()) {
            String fruit = iterator.next();
            System.out.println(fruit);
            // Replace "Banana" with "Mango"
            if (fruit.equals("Banana")) {
                iterator.set("Mango");
            }
        }
        System.out.println("\nBackward Traversal:");
        while (iterator.hasPrevious()) {
            System.out.println(iterator.previous());
        }
        System.out.println("\nFinal List: " + fruits);
    }
}
```

Collections Class

Key Points about Collections Class

•It is a **utility class**.

•Contains **static methods** only.

•Used for **sorting**, **searching**, **reversing**, **shuffling**, **min/max operations**, **synchronization**, etc., on collections.

•Primarily used with **List**, **Set**, **Map**, and **Collection** types.

Commonly Used Methods in Collections Class

1.sort(List<T> list): Sorts the list in ascending (natural) order.

Example:

```
List<Integer> list = Arrays.asList(5, 3, 9, 1);
Collections.sort(list);
System.out.println(list); // Output: [1, 3, 5, 9]
```

2. sort(List<T> list, Comparator<? super T> c): Sorts the list using a custom comparator.

```
List<String> names = Arrays.asList("Charlie", "Alice", "Bob");
Collections.sort(names, Comparator.reverseOrder());
System.out.println(names); // Output: [Charlie, Bob, Alice]
```

3.reverse(List<?> list): Reverses the order of elements in the list.

```
List<String> list = Arrays.asList("A", "B", "C");
Collections.reverse(list);
System.out.println(list); // Output: [C, B, A]
```

4. shuffle(List<?> list): Randomly shuffles the list elements.

```
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);
Collections.shuffle(list);
System.out.println(list); // Output: [random order]
```

5. swap(List<?> list, int i, int j): Swaps two elements at the given positions.

```
List<String> list = new ArrayList<>(Arrays.asList("A", "B", "C"));
Collections.swap(list, 0, 2);
System.out.println(list); // Output: [C, B, A]
```

6. max(Collection<? extends T> coll) : Returns the maximum element.

```
List<Integer> list = Arrays.asList(10, 20, 5);
System.out.println(Collections.max(list)); // Output: 20
```

7. min(Collection<? extends T> coll): Returns the minimum element.

```
List<Integer> list = Arrays.asList(10, 20, 5);
System.out.println(Collections.min(list)); // Output: 5
```

8. binarySearch(List<T> list, T key): Searches for an element in a **sorted** list.

```
List<Integer> list = Arrays.asList(1, 3, 5, 7);
System.out.println(Collections.binarySearch(list, 5)); // 2
```

9. fill(List<? super T> list, T obj) : Fills the list with the specified object.

```
List<String> list = new ArrayList<>(Arrays.asList("A", "B", "C"));
Collections.fill(list, "X");
System.out.println(list); // Output: [X, X, X]
```

10. replaceAll(List<T> list, T oldVal, T newVal): Replaces all occurrences of oldVal with newVal.

```
List<String> list = new ArrayList<>(Arrays.asList("apple", "banana", "apple"));
Collections.replaceAll(list, "apple", "orange");
System.out.println(list); // Output: [orange, banana, orange]
```

11. frequency(Collection<?> c, Object o): Counts how many times o appears in collection c.

```
List<String> list = Arrays.asList("A", "B", "A", "C", "A");
int freq = Collections.frequency(list, "A");
System.out.println(freq); // Output: 3
```

12. disjoint(Collection<?> c1, Collection<?> c2): Returns true if there are no common elements

```
List<Integer> list1 = Arrays.asList(1, 2, 3);
List<Integer> list2 = Arrays.asList(4, 5, 6);
System.out.println(Collections.disjoint(list1, list2)); // Output: true
```