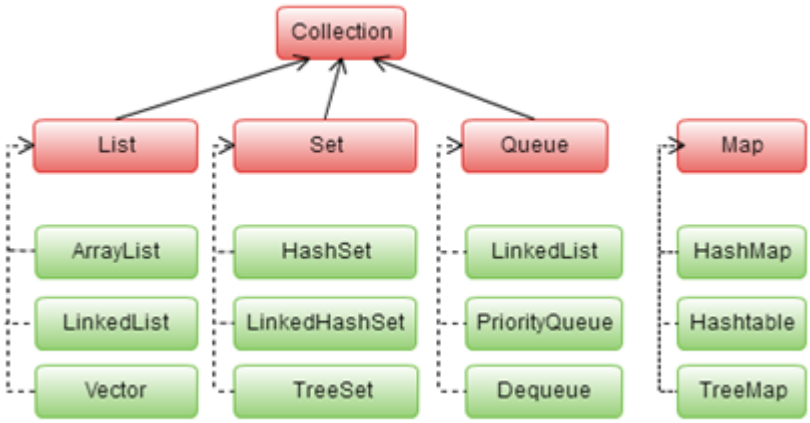
Collection Framework:



Collection Interface

1. **List:** Ordered collection supporting duplicate elements (e.g., ArrayList, LinkedList).
2. **Set:** Unordered collection with no duplicate elements (e.g., HashSet, TreeSet).
3. **Queue:** first in first out (e.g., LinkedList, PriorityQueue).

Collection classes:

1. **ArrayList:** Resizable array implementation of the List interface.
2. **LinkedList:** Doubly linked list implementation of the List interface.
3. **HashSet:** Implementation of the Set interface using a hash table for storage.
4. **TreeSet:** Implementation of the Set interface using a self-balancing binary search tree.
5. **PriorityQueue:** Implementation of the Queue interface based on priority heap.

Map Interface

1. **Map:** Key-value pair collection (e.g., HashMap, TreeMap).

Map Classes:

1. **HashMap:** Implementation of the Map interface using a hash table for storage.

2. **TreeMap:** Implementation of the Map interface using a self-balancing binary search tree.

List

List<String> list = Arrays.asList(array); // convert array to list

Method	Description	Example Code
boolean add(E element)	Adds the specified element at the end of the list.	list.add("Hello"); // Adds "Hello" to the list
void add(int index, E element)	Inserts the specified element at the specified position in the list.	list.add(1, "World"); // Adds "World" at index 1
boolean addAll(Collection<? extends E> c)	Adds all elements of the specified collection to the end of the list.	list.addAll(anotherList); // Adds all elements of anotherList
void clear()	Removes all elements from the list.	list.clear(); // Removes all elements from the list
boolean contains(Object o)	Checks if the list contains the specified element.	boolean contains = list.contains("Hello"); // Checks if "Hello" is in the list
E get(int index)	Returns the element at the specified position.	String element = list.get(0); // Retrieves element at index 0
int indexOf(Object o)	Returns the index of the first occurrence of the specified element in the list.	int index = list.indexOf("World"); // Gets index of "World"
boolean isEmpty()	Checks if the list is empty.	boolean empty = list.isEmpty(); // Checks if the list is empty
Iterator<E> iterator()	Returns an iterator over the elements in the list.	Iterator<String> iter = list.iterator(); // Iterates through elements
int lastIndexOf(Object o)	Returns the index of the last occurrence of the specified element in the list.	int lastIndex = list.lastIndexOf("World"); // Gets last index of "World"

Method	Description	Example Code
E remove(int index)	Removes the element at the specified position.	list.remove(1); // Removes element at index 1
boolean remove(Object o)	Removes the first occurrence of the specified element from the list.	boolean removed = list.remove("Hello"); // Removes "Hello" from the list
boolean removeAll(Collection<?> c)	Removes all elements from the list that are present in the specified collection.	boolean removedAll = list.removeAll(anotherList); // Removes all elements present in anotherList
boolean retainAll(Collection<?> c)	Retains only the elements in the list that are present in the specified collection.	boolean retainedAll = list.retainAll(anotherList); // Retains only elements present in anotherList
E set(int index, E element)	Replaces the element at the specified position in the list with the specified element.	String replaced = list.set(1, "Universe"); // Replaces element at index 1 with "Universe"
int size()	Returns the number of elements in the list.	int size = list.size(); // Gets the size of the list
Object[] toArray()	Returns an array containing all the elements in the list.	Object[] array = list.toArray(); // Converts list to an array

Queue

Method	Description	Example Code
boolean add(E element)	Inserts the specified element into the queue. Throws an exception if the operation fails.	queue.add("First"); // Adds "First" to the queue
boolean offer(E element)	Inserts the specified element into the queue. Returns true if successful, false otherwise.	boolean offered = queue.offer("Second"); // Offers "Second" to the queue
E remove()	Removes and returns the head of the queue. Throws an exception if the queue is empty.	String element = queue.remove(); // Removes and returns head of the queue
E poll()	Removes and returns the head of the queue. Returns null if the queue is empty.	String polledElement = queue.poll(); // Polls and returns head of the queue
E element()	Retrieves, but does not remove, the head of the queue. Throws an exception if the queue is empty.	String head = queue.element(); // Retrieves head of the queue without removal
E peek()	Retrieves, but does not remove, the head of the queue. Returns null if the queue is empty.	String peekedElement = queue.peek(); // Peeks head of the queue without removal

Method	Description	Example Code
boolean addAll(Collection<? extends E> c)	Adds all elements of the specified collection to the queue.	queue.addAll(anotherQueue); // Adds all elements of anotherQueue to the queue
void clear()	Removes all elements from the queue.	queue.clear(); // Removes all elements from the queue
boolean contains(Object o)	Checks if the queue contains the specified element.	boolean contains = queue.contains("Element"); // Checks if "Element" is in the queue
boolean containsAll(Collection<?> c)	Checks if the queue contains all elements of the specified collection.	boolean containsAll = queue.containsAll(anotherQueue); // Checks if all elements of anotherQueue are in the queue
boolean isEmpty()	Checks if the queue is empty.	boolean empty = queue.isEmpty(); // Checks if the queue is empty
Iterator<E> iterator()	Returns an iterator over the elements in the queue.	Iterator<String> iter = queue.iterator(); // Iterates through elements
int size()	Returns the number of elements in the queue.	int size = queue.size(); // Gets the size of the queue

Set

Method	Description	Example Code
boolean add(E element)	Adds the specified element to the set. Returns true if this set did not already contain the specified element.	set.add("Element");
boolean addAll(Collection<? extends E> c)	Adds all elements of the specified collection to the set.	set.addAll(anotherSet);
void clear()	Removes all elements from the set.	set.clear();
boolean contains(Object o)	Checks if the set contains the specified element.	set.contains("Element");
boolean containsAll(Collection<?> c)	Checks if the set contains all elements of the specified collection.	set.containsAll(anotherSet);
boolean isEmpty()	Checks if the set is empty.	set.isEmpty();
Iterator<E> iterator()	Returns an iterator over the elements in the set.	Iterator<String> iter = set.iterator();
boolean remove(Object o)	Removes the specified element from the set. Returns true if the set contained the specified element.	set.remove("Element");

Method	Description	Example Code
boolean removeAll(Collection<?> c)	Removes all elements from the set that are present in the specified collection.	set.removeAll(anotherSet);
boolean retainAll(Collection<?> c)	Retains only the elements in the set that are present in the specified collection.	set.retainAll(anotherSet);
int size()	Returns the number of elements in the set.	set.size();

Map

```
Iterator i = <map>.entrySet().iterator();
```

https://www.geeksforgeeks.org/sorting-a-hashmap-according-to-values/

Method	Description	Example Code
V put(K key, V value)	Associates the specified value with the specified key in the map.	map.put("key", "value"); // Adds a key-value pair to the map
void putAll(Map<? extends K, ? extends V> m)	Copies all of the mappings from the specified map to this map.	map.putAll(anotherMap); // Copies mappings from another map to this map
V get(Object key)	Returns the value to which the specified key is mapped, or null if the key is not present.	V value = map.get("key"); // Retrieves value associated with the key
boolean containsKey(Object key)	Checks if the map contains the specified key.	map.containsKey("key"); // Checks if the map contains the key
boolean containsValue(Object value)	Checks if the map contains the specified value.	map.containsValue("value"); // Checks if the map contains the value

Method	Description	Example Code
V remove(Object key)	Removes the mapping for the specified key from the map if present.	V removedValue = map.remove("key"); // Removes mapping associated with the key
void clear()	Removes all mappings from the map.	map.clear(); // Clears all mappings in the map
boolean isEmpty()	Checks if the map is empty.	map.isEmpty(); // Checks if the map is empty
int size()	Returns the number of key-value mappings in the map.	map.size(); // Gets the number of mappings in the map
Set<K> keySet()	Returns a set view of the keys contained in the map.	Set<K> keys = map.keySet(); // Retrieves keys in the map
Collection<V> values()	Returns a collection view of the values contained in the map.	Collection<V> values = map.values(); // Retrieves values in the map

Comparator/Comparable support :- TreeSet, TreeMap, PriorityQueue

REGEX:

Pattern	Description
^	Matches the start of a string.
$	Matches the end of a string.
.	Matches any single character.
[xyz]	Matches one character from x, y, or z.
[^xyz]	Matches one character that is NOT x, y, or z.
[a-z]	Matches one character in the range a through z.
*	Matches 0 or more occurrences of the preceding element.
+	Matches one or more occurrences of the preceding element.
?	Matches 0 or 1 occurrence of the preceding element.
{x}, {x,}, {x, y}	Matches exactly x times, at least x times, x but no more than y times.
sub1 | sub2	Matches either substring #1 or substring #2.
\b	Word boundary: finds word within spaces, parentheses, period.

Pattern	Description
\d	Any single digit.
\s	Whitespace character (space, tab, newline).
\w	Single word character: A-Za-z0-9_.

Pattern	Matches
^$	Empty lines.
^.$	Line with exactly one character.
[A-Z]	Any single capital letter.
[0-9]	Any single digit.
[0-9][a-z]	Any digit followed by any lowercase letter.
[0-9][^a-z]*	Any digit followed by anything but a lowercase letter.
\?\.	Question mark followed by period.
.*	Any string, including a blank string.
".*"	Any string, including the null string.
".+"	Any string, not including the null string.
".?"	Zero or one character.
(\?){2}	Two question marks.
(Barfy){2,4}	Two, three, or four occurrences of "Barfy".
[a-z]{3}	Three lowercase letters.
\s[01]+\s	Binary data: one or more 0 or 1 surrounded by spaces.
[\+\-]?[0-9]*\.[0-9]*	Floating point numbers.

Example codes regex:

```
import java.util.regex.*;

public class MatchesExample {
    public static void main(String[] args) {
        String text = "Hello, world!";
        String regex = "Hello.*"; // Pattern to match strings starting with "Hello"

        Pattern pattern = Pattern.compile(regex);
        Matcher matcher = pattern.matcher(text);

        if (matcher.matches()) {
            System.out.println("The entire string matches the pattern.");
        } else {
            System.out.println("The entire string does not match the pattern.");
```

Output:
The entire string matches the pattern.

```
        }
    }
}
```

```
import java.util.regex.*;

public class FindExample {
    public static void main(String[] args) {
        String text = "The cat sat on the mat. cat";
        String regex = "\\bcat\\b"; // Pattern to find the word "cat"

        Pattern pattern = Pattern.compile(regex);
        Matcher matcher = pattern.matcher(text);

        while (matcher.find()) {
            System.out.println("Found match at index " + matcher.start());
        }
    }
}
```

Output:

```
Found match at index 4
Found match at index 21
```

Enum

```
package Learn;

enum Status {
        // named constants
        failed, success, pending
}
enum Laptop {
        // named constants and its price
        macbook(1000), xps(2000), surface(1500), abc; // objects
        private int price;
        // object without price can not be created if constructor without param given below is not written.
        private Laptop(){ // constructor
                this.price = 400;
        }
        private Laptop(int price){ // constructor
                this.price = price;
        }
        public int getPrice() {
                return price;
        }
        public void setPrice(int price) {
                this.price = price;
        }

}
```

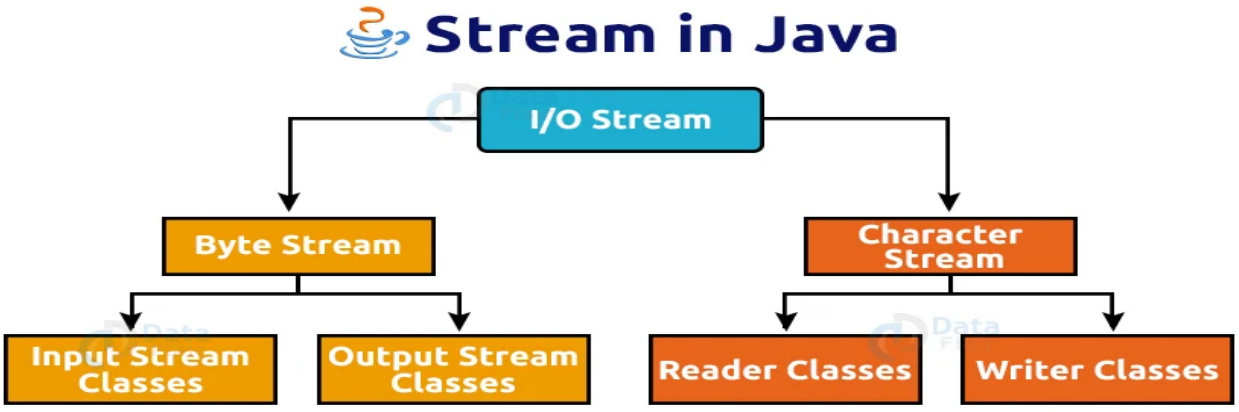
```java
public class Enum {
	public static void main(String[] args) {
		// Status is class and everything inside it, is a class
		Status s = Status.failed;
		System.out.println(s);
		// prints index of status.
		System.out.println(s.ordinal());
		// to get all status
		Status[] sarray = Status.values();

		for (Status st : sarray) {
			System.out.println(st + " : " + st.ordinal());
		}

		// error
		// Status s1 = "vf";
		Status statusCheck = Status.failed;
		switch (statusCheck) {
		// statusCheck is type of 'Status' so no need to write Status.pending in case
		// but in case of if-else Status.<state> is required.
		case pending:
			System.out.println("--Pending");
			break;
		case failed:
			System.out.println("--Failed");
			break;
		case success:
			System.out.println("--Success");
			break;
		}
		if(statusCheck.equals(Status.pending)) {
			System.out.println("--Pending");

		}
		else if(statusCheck.equals(Status.failed)) {
			System.out.println("--Failed");

		}

		// enum is class but it can not be extend by other class.
		// enum in java extends Enum class.
		System.out.println(s.getClass());

		System.out.println(s.getClass().getSuperclass());


		// laptop as class
		Laptop l = Laptop.macbook;

		// we can change value of price.
		System.out.println(l.getPrice());
		l.setPrice(3000);
		System.out.println(l.getPrice());
		for (Laptop lap : Laptop.values()) {
			System.out.println(lap + " : " + lap.getPrice());
```

```
                }

        }
}
```

output:

```
failed
0
failed : 0
success : 1
pending : 2
--Failed
--Failed
class Learn.Status
class java.lang.Enum
1000
3000
macbook : 3000
xps : 2000
surface : 1500
abc : 400
```

IO Java:



Byte Stream: Byte streams in Java operate directly with *bytes* of data. Used for binary data (like images, audio).

InputStreamClasses: Reads binary data from a source.

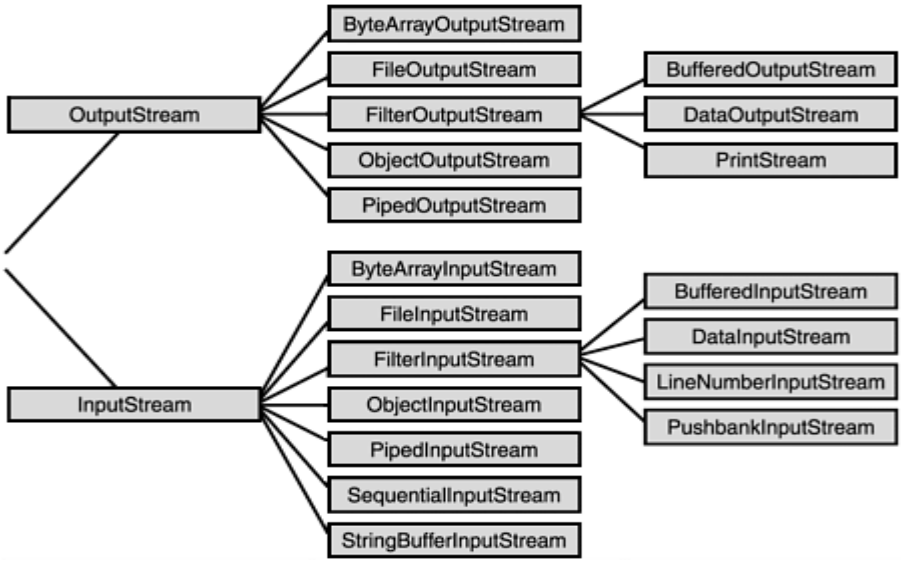
OutputStreamClasses: Writes binary data to a destination.

Character Stream: Character streams in Java handle IO operations with *characters*. Used for text-based data (like strings, documents).

ReaderClasses: Reads characters from a source.

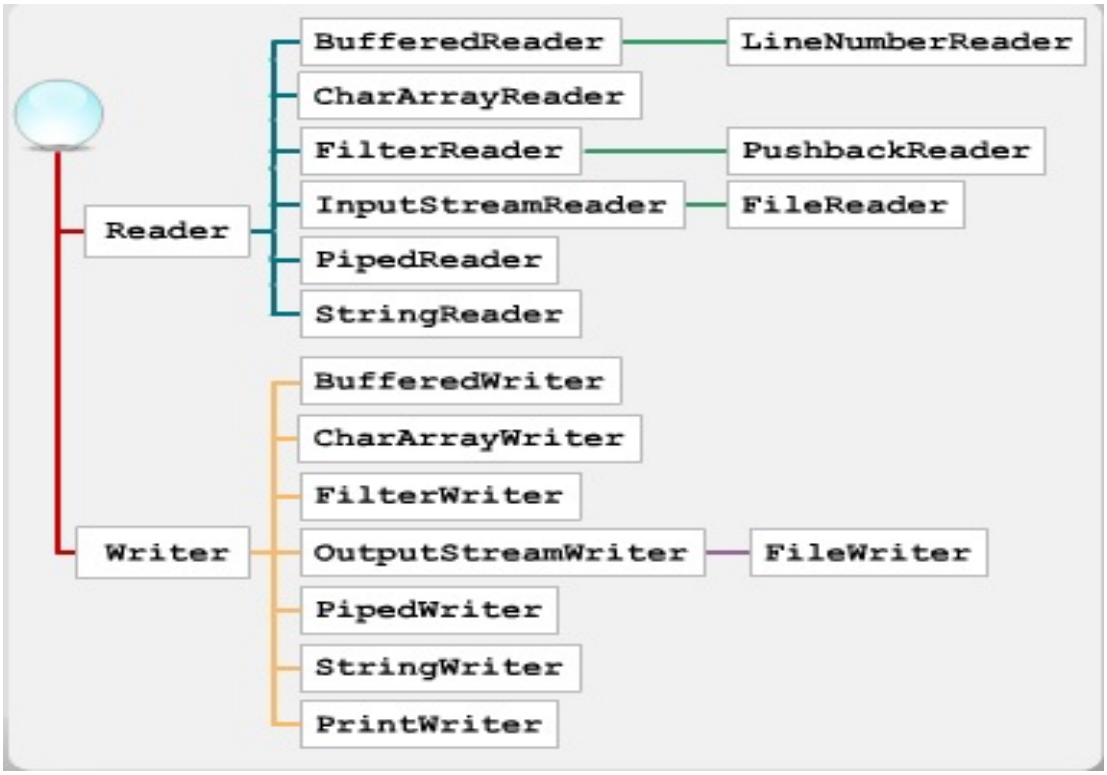
WriterClasses: Writes characters to a destination.

Byte Stream:



1. FileInputStream: Reads bytes from a file.
2. BufferedInputStream: Buffers input from another stream for improved efficiency.
3. ByteArrayInputStream: Reads bytes from a byte array.
4. ObjectInputStream: Reads Java objects from an underlying input stream.
5. DataInputStream: Reads primitive data types from an underlying input stream.
6. FileOutputStream: Writes bytes to a file.
7. BufferedOutputStream: Buffers output to another stream for improved efficiency.
8. ByteArrayOutputStream: Writes bytes to a byte array.
9. ObjectOutputStream: Writes Java objects to an underlying output stream.
10. PrintStream: Provides convenient methods for writing formatted data to an output stream.
11. DataOutputStream: Writes primitive data types to an underlying output stream.

Character Stream:



1. FileReader: Reads characters from a file.
2. BufferedReader: Buffers input from another reader for efficiency.
3. CharArrayReader: Reads characters from a character array.
4. StringReader: Reads characters from a string.
5. InputStreamReader: Adapts an input stream into a reader.
6. FileWriter: Writes characters to a file.
7. BufferedWriter: Buffers output to another writer for efficiency.
8. CharArrayWriter: Writes characters to a character array.
9. StringWriter: Writes characters to a string.
10. OutputStreamWriter: Adapts an output stream into a writer.
11. PrintWriter: Provides convenient methods for writing formatted text to an output stream.

Byte File I/O -(FileInputStream and FileOutputStream) -unbuffered -read/write one byte per call	**Character File I/O** -(FileReader and FileWriter) -unbuffered -read/write one char(=2 byte) per call

try (FileOutputStream out = new FileOutputStream(filename);
FileInputStream in = new FileInputStream("src/mycode.java")) {
int c;
while ((c = in.read()) != -1) {
out.write(c);
}
in.close();
out.close();
} catch (FileNotFoundException e) {
System.out.println(e.getMessage());
} catch (IOException e) {
System.out.println(e.getMessage());
}
	try (FileWriter out = new FileWriter(filename);
	FileReader in = new FileReader("src/mycode.java")) {
	int c;
	while ((c = in.read()) != -1) {
	out.write(c);
	}
	in.close();
	out.close();
	} catch (FileNotFoundException e) {
	System.out.println(e.getMessage());
	} catch (IOException e) {
	System.out.println(e.getMessage());
	}

Buffered means the JVM reads more data from a source(file, socket...) than it currently needs, so that the next read(s) will be faster – it's already in memory
Unbuffered means each read or write request is handled directly by the operating system. So, much slower
The difference might not be noticeable in small data writes. However, in scenarios with large amounts of data, the buffered approach tends to be more efficient as it reduces the number of actual write operations by storing data temporarily in memory before writing to the file.

Wrapping File I/O

Wrapping file IO involves enhancing basic file IO operations by using wrapper classes that add additional functionality or improve performance.
These wrapper classes are built on top of fundamental file IO classes (FileInputStream, FileOutputStream, FileReader, FileWriter, etc.) to provide enhanced capabilities.

Advantage:

Functionality Extension: Wrappers add functionality that the base classes may lack, such as buffering, character encoding, or specialized handling.
Performance Improvement: Wrappers can improve performance by reducing the number of actual IO operations or by optimizing data handling.

Eg.

A) FileInputStream fileInputStream = new FileInputStream("example.txt")
BufferedInputStream bufferedInputStream = new BufferedInputStream(fileInputStream)
-> In this example, **BufferedInputStream** wraps around **FileInputStream** to add buffering capabilities. The bufferedInputStream can be used for more efficient reading from the file.
-> The wrapper class uses an overloaded constructor that accepts an instance of the base class to utilize its functionality while adding its own features or improvements (like buffering in this case). Here, BufferedInputStream's constructor takes an InputStream as a parameter, allowing it to wrap around any class that extends InputStream, such as FileInputStream.

B) FileReader fileReader = new FileReader("example.txt");
BufferedReader bufferedReader = new BufferedReader(fileReader);
-> The constructor of BufferedReader takes a Reader (or its subclass) as a parameter, allowing it to wrap around any class that extends Reader, such as FileReader.
-> Wrapping FileReader with BufferedReader allows for enhanced character-based IO functionality, especially in terms of buffering and reading lines efficiently.

```
public static String read(String filename) {
    StringBuilder sb = new StringBuilder();
    try (BufferedReader input = new
BufferedReader(new FileReader(filename))) {
        String s;
        while ((s = input.readLine()) != null) {
            sb.append(s).append("\n");
        }
    } catch (IOException e) {
        System.out.println(e.getMessage());
    }
    return sb.toString();
}
```

-The method readLine() is part of the BufferedReader class.
-The FileReader class does not directly provide a method for reading whole lines of text like readLine() in BufferedReader. (Advantage of Wrapping)

Scanner Class

-> Scanner class is not part of the I/O library (it's in java.util).

-> It offers a versatile way to parse data from various sources, including input streams and files.

Eg.

```
-Scanner keyboard = new Scanner(System.in) // wraps the standard inputStream
-Scanner fileScanner = new Scanner(new File("example.txt")) // wraps file obj
-Scanner readerScanner = new Scanner(new FileReader("example.txt")); //    wraps FileReader obj
-Scanner bufferedScanner = new Scanner(new BufferedReader(new FileReader("example.txt"))); // wraps BufferedReader obj
```

```
public static void readInts(String filename) {
    try (Scanner scanner = new Scanner(new BufferedReader(new FileReader(filename)))) {
        while (scanner.hasNext()) {
            intList.add(scanner.nextInt());
        }
    } catch (IOException e) {
        System.out.println(e.getMessage());
    }
}
```

PrintWriter

-> PrintWriter is specifically designed for writing text, unlike OutputStream or Writer, which deal with bytes or characters.

-> PrintWriter has several constructors that allow it to be used with different types of output destinations such as files, output streams, or writers.

```
try (PrintWriter writer = new PrintWriter("output.txt")) {
    writer.println("Hello, PrintWriter!");
    writer.printf("Formatted: %d, %s%n", 42, "value");  // formatted text
} catch (FileNotFoundException e) {
    e.printStackTrace();
}
```

Output.txt

```
Hello, PrintWriter!
Formatted: 42, value
```

-> System.out and System.err are PrintStream objects, but basically have the same formatting capabilities

-> System.out (Standard Output) is normally the screen but can be redirected at the OS level such as a file or another output stream.

-> System.err (Standard Error)is also normally the screen, can also be redirected, and is used for things like traceback stacks. Some apps redirect this to a file so that the user doesn't see it (normally), but it's there for developers to help debug

```
public static void writeInts(String filename) {
    try (PrintWriter printWriter = new PrintWriter(
            new BufferedWriter(new FileWriter(filename)))) {
        for (Integer i : intList) {
            printWriter.println(i);
        }
    } catch (IOException e) {
        System.err.println(e.getMessage());
    }
}
```

InputStreamReader wrapping a ByteArrayInputStream to interpret some UTF-8 characters

```
import java.io.*;
import java.nio.charset.StandardCharsets;

public class ByteReader {
    public static void main(String[] args) {
        String s = "\u0041\u0042\u0043\u0044\n" +
                "\u0905\u0906\u0907\u0908\n" +
                "\u266A\u266B\n" +
                "\u263A\n" +
                "\u0627\u0628\u0629\u062A\n" +
                "\u4E00\u4E8C\u4E09\u4E96";
        byte[] codes = s.getBytes(); //load s as raw bytes

        // Open InputStreamReader with a character decoder as per UTF-8
        try (InputStreamReader isr = new InputStreamReader(new
ByteArrayInputStream(codes),
                StandardCharsets.UTF_8)) {
```

```
            int c;
            while ((c = isr.read()) != -1)
                System.out.print((char) c); //will print according to settings in
IDE
        } catch (IOException e1) {
            e1.printStackTrace();
        }
    }
}
```

Binary Data

Object Stream:

These are typically used to store entire objects to a file.
When you serialize an object (convert it into a sequence of bytes), you can write it to a file using object streams.
This method is useful for storing complex data structures where you want to maintain the entire object's integrity.(When you store an entire object using serialization, you're preserving the entire state and structure of that object. This means all the data within the object, along with its relationships and hierarchy, are maintained.)
It allows easy reading and writing of objects but might not be the most efficient method for large datasets or records due to the overhead of storing complete objects.
The ObjectInputStream and ObjectOutputStream allow entire objects to be read/written, instead of reading/writing their fields and using setters/getters
To do this, a class must implement the Serializable interface. This interface has no methods, it just identifies the class as being serializable.
Then use writeObject(yourObject) and (YourClass)readObject(yourObject) to write/read the binary (not text) file

```
import java.io.*;

// Sample object class
class Student implements Serializable {
    String name;
    int age;

    public Student(String name, int age) {
```

```
        this.name = name;
        this.age = age;
    }
}

public class ObjectStreamExample {
    public static void main(String[] args) {
        // Writing objects to a file using ObjectOutputStream
        try (ObjectOutputStream outputStream = new ObjectOutputStream(new
FileOutputStream("students.dat"))) {
            // Creating and writing objects to the file
            Student student1 = new Student("Alice", 20);
            Student student2 = new Student("Bob", 22);

            outputStream.writeObject(student1);
            outputStream.writeObject(student2);
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Reading objects from the file using ObjectInputStream
        try (ObjectInputStream inputStream = new ObjectInputStream(new
FileInputStream("students.dat"))) {
            // Reading objects from the file
            Student student1 = (Student) inputStream.readObject();
            Student student2 = (Student) inputStream.readObject();

            System.out.println(student1.name + " " + student1.age);
            System.out.println(student2.name + " " + student2.age);
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

ObjectInputStream doesn't have a way to peek ahead or say "end of file", so using a loop to read a file can stop when ... •

- readObject( ) throws an exception – this isn't the right way to handle normal processing
- the available( ) method in FileInputStream returns a value less than zero –you have to declare a variable of that type instead of just wrapping it. This is the better solution.

```
import java.io.*;

class Student implements Serializable {
    String name;
```

```
    int age;

    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

public class ObjectStreamExample {
    public static void main(String[] args) {
        // Writing objects to a file using ObjectOutputStream
        try (ObjectOutputStream outputStream = new ObjectOutputStream(new
FileOutputStream("students.dat"))) {
            // Creating and writing objects to the file
            Student student1 = new Student("Alice", 20);
            Student student2 = new Student("Bob", 22);

            outputStream.writeObject(student1);
            outputStream.writeObject(student2);
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Reading objects from the file using FileInputStream and ObjectInputStream
        try (FileInputStream fileInputStream = new FileInputStream("students.dat");
             ObjectInputStream inputStream = new ObjectInputStream(fileInputStream)) {

            // Loop through the file and read each serialized object until no more objects are
available
            while (fileInputStream.available() > 0) {
                Student student = (Student) inputStream.readObject();
                System.out.println(student.name + " " + student.age);
            }
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Random Access Files:

These store records in a binary format within a file.
Unlike object streams where you write entire objects, random access files allow you to read or write data at any specific position within the file.
This method is efficient for fixed-size records or data where you need direct access to specific parts without necessarily reading the entire file.

It's commonly used in situations where you want to access data in a non-sequential manner, like databases or when dealing with large datasets.
RandomAccessFile allows you to move the file pointer around at will with the seek(byteNumber) method – but you need to know what argument to use as the byte number.

- seek(0) takes you back to the beginning of the file
- That's why you don't do this with text files(each line or record might have a different length because the content within them varies. When you move the file pointer to a specific byte position using seek(), it might not align with the start of a record, especially if the records have varying lengths.)

you usually open this type of file for "rw" access

```
import java.io.*;

public class RandomAccessFileExample {
    public static void main(String[] args) {
        try (RandomAccessFile file = new RandomAccessFile("records.dat", "rw")) {
            // Writing records to the file
            // Assuming each record has a fixed size of 20 bytes (for simplicity)
            String record1 = "Record 1 data";
            String record2 = "Record 2 data";

            file.writeBytes(String.format("%-20s", record1));
            file.writeBytes(String.format("%-20s", record2));

            // Reading specific records from the file
            int recordSize = 20;
            file.seek(recordSize * 1); // Move to the second record (index starts at 0)
            byte[] buffer = new byte[recordSize];
            file.read(buffer);

            String retrievedRecord = new String(buffer);
            System.out.println("Retrieved Record: " + retrievedRecord.trim()); // Trim to remove
whitespace padding

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
public class RandomAccessStuff {
    public static void main(String[] args) {
        // Array of double values
        double[] data = {2.7, -10.12, 9.5, 2.87};

        try (RandomAccessFile raf = new RandomAccessFile("binary.bin", "rw")) {
```

```
            // Writing double values to the file
            for (double d : data) {
                raf.writeDouble(d); // Write out the data
            }

            long b = raf.length() / 8; // Each double is 8 bytes
            raf.seek(2 * b); // Seek to record #2 (each double is treated as one record)

            // Overwriting a specific record at position 2 with a new double value
            raf.writeDouble(200.45);

            raf.seek(0); // Seek to the beginning of the file

            // Reading double values from the file back into the data array
            for (int i = 0; i < 4; i++) {
                data[i] = raf.readDouble();
            }

            raf.close(); // Close the RandomAccessFile
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }

        // Outputting the updated data array
        for (double d : data) {
            System.out.println("d = " + d);
        }
    }
}
```

Paths and Files:

The java.nio.file package in Java, introduced in Java 7, provides an improved and more versatile way to perform file I/O operations compared to the traditional java.io package.

Paths:

The Paths class is used to create instances of Path, which represents the location of a file or directory in the file system.
It provides methods to create a Path object from a string representation of a file path or URI.
Paths are platform-independent, making it easier to work with file paths across different operating systems.

```
import java.nio.file.Path;
```

```
import java.nio.file.Paths;

public class PathExample {
    public static void main(String[] args) {
        // Creating a Path object
        Path path = Paths.get("folder", "file.txt");
        // Or
        // Path path = Paths.get("/home/user/folder/file.txt"); // Absolute path
        System.out.println(path.toAbsolutePath()); // Absolute path to file.txt
    }
}
```

Files:

The Files class provides various static methods for performing operations on files, such as reading/writing, copying, moving, deleting, and checking file attributes.
It complements the Path class and works with Path objects to perform file-related operations.

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.io.IOException;

public class FileOperations {
    public static void main(String[] args) {
        // Creating a Path object
        Path path = Paths.get("folder", "file.txt");

        try {
            // Checking if a file exists
            boolean exists = Files.exists(path);

            // Creating a new file
            Path newFile = Files.createFile(path);

            // Copying a file
            Path destination = Paths.get("new_folder", "copied_file.txt");
            Files.copy(path, destination);

            // Deleting a file
            Files.delete(newFile);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.List;

public class FileMethodsExample {
    public static void main(String[] args) {
        // Creating a file object
        File file = new File("example.txt");

        try {
            // Getting absolute path
            String absolutePath = file.getAbsolutePath();
            System.out.println("Absolute Path: " + absolutePath);

            // Checking if file exists
            boolean doesExist = file.exists();
            System.out.println("File exists: " + doesExist);

            // Creating a directory
            File directory = new File("new_directory");
            boolean isCreated = directory.mkdir();
            System.out.println("Directory created: " + isCreated);

            // Getting file size
            Path filePath = file.toPath();
            long fileSize = Files.size(filePath);
            System.out.println("File size: " + fileSize + " bytes");

            // Reading all lines from a file
            List<String> lines = Files.readAllLines(filePath);
            System.out.println("Lines read: " + lines);

            // Renaming the file
            File newFile = new File("renamed_example.txt");
            boolean isRenamed = file.renameTo(newFile);
            System.out.println("File renamed: " + isRenamed);

            // Deleting the file
            boolean isDeleted = Files.deleteIfExists(newFile.toPath());
            System.out.println("File deleted: " + isDeleted);

            // Deleting the directory created
            boolean isDirDeleted = directory.delete();
```

```
            System.out.println("Directory deleted: " + isDirDeleted);

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

java.io.File Methods	Description
createNewFile()	Creates a new empty file if it doesn't exist.
mkdir()	Creates the directory specified by the abstract pathname.
mkdirs()	Creates the directory specified by the abstract pathname, including any necessary but nonexistent parent directories.
delete()	Deletes the file or directory.
exists()	Checks if the file or directory exists.
isFile()	Checks if the abstract pathname denotes a file.
isDirectory()	Checks if the abstract pathname denotes a directory.
getName()	Returns the name of the file or directory.
getPath()	Returns the pathname string of this abstract pathname.
getAbsolutePath()	Returns the absolute pathname string of this abstract pathname.
lastModified()	Returns the time when the file was last modified.

```
import java.io.File;
import java.io.IOException;

public class FileMethodsExample {
    public static void main(String[] args) {
        // Creating a File object
        File file = new File("example.txt");

        try {
            // createNewFile() - Creates a new empty file if it doesn't exist
            boolean createdNewFile = file.createNewFile();
            System.out.println("createNewFile() - File created: " + createdNewFile);

            // mkdir() - Creates the directory specified by the abstract pathname
            File directory = new File("new_directory");
            boolean dirCreated = directory.mkdir();
            System.out.println("mkdir() - Directory created: " + dirCreated);

            // mkdirs() - Creates the directory and its parent directories if they don't exist
```

```
            File nestedDirectory = new File("parent_directory/nested_directory");
            boolean nestedDirCreated = nestedDirectory.mkdirs();
            System.out.println("mkdirs() - Nested Directory created: " + nestedDirCreated);

            // delete() - Deletes the file or directory
            boolean deleted = file.delete();
            System.out.println("delete() - File deleted: " + deleted);

            // exists() - Checks if the file or directory exists
            boolean doesExist = file.exists();
            System.out.println("exists() - File exists: " + doesExist);

            // isFile() - Checks if the abstract pathname denotes a file
            boolean isFile = file.isFile();
            System.out.println("isFile() - Is it a file: " + isFile);

            // isDirectory() - Checks if the abstract pathname denotes a directory
            boolean isDir = directory.isDirectory();
            System.out.println("isDirectory() - Is it a directory: " + isDir);

            // getName() - Returns the name of the file or directory
            String name = file.getName();
            System.out.println("getName() - Name: " + name);

            // getPath() - Returns the pathname string of this abstract pathname
            String path = file.getPath();
            System.out.println("getPath() - Path: " + path);

            // getAbsolutePath() - Returns the absolute pathname string of this abstract
pathname
            String absolutePath = file.getAbsolutePath();
            System.out.println("getAbsolutePath() - Absolute Path: " + absolutePath);

            // lastModified() - Returns the time when the file was last modified
            long lastModified = file.lastModified();
            System.out.println("lastModified() - Last Modified: " + lastModified);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

java.nio.file.Files Methods	Description
createDirectory(Path path)	Creates a directory.
createDirectories(Path dir)	Creates the directory and its parent directories if they don't exist.
createFile(Path path, FileAttribute<?>... attrs)	Creates a new file.
delete(Path path)	Deletes a file or empty directory.
move(Path source, Path target, CopyOption... options)	Moves or renames a file or directory to a target file or directory.
exists(Path path, LinkOption... options)	Checks if the file or directory exists.
isRegularFile(Path path, LinkOption... options)	Checks if the path is a regular file.
isDirectory(Path path, LinkOption... options)	Checks if the path is a directory.
isReadable(Path path)	Checks if the file is readable.
isWritable(Path path)	Checks if the file is writable.
readAttributes(Path path, String attributes, LinkOption... options)	Reads the attributes of a file as a bulk operation.
readAllBytes(Path path)	Reads all bytes from a file.
readAllLines(Path path)	Reads all lines from a file.
isExecutable(Path path)	Checks if the file is executable.

```
import java.io.IOException;
import java.nio.file.*;
import java.util.List;

public class FilesMethodsExample {
    public static void main(String[] args) {
        // Define a path for the file/directory operations
        Path path = Paths.get("example.txt");

        try {
            // createDirectory(Path path) - Creates a directory
            Files.createDirectory(path);
            System.out.println("createDirectory() - Directory created");
```

```
        // createDirectories(Path dir) - Creates the directory and its parent directories if they
don't exist
        Path nestedPath = Paths.get("parent_directory/nested_directory");
        Files.createDirectories(nestedPath);
        System.out.println("createDirectories() - Nested Directory created");

        // createFile(Path path, FileAttribute<?>... attrs) - Creates a new file
        Path filePath = Paths.get("newfile.txt");
        Files.createFile(filePath);
        System.out.println("createFile() - New file created");

        // delete(Path path) - Deletes a file or empty directory
        Files.delete(filePath);
        System.out.println("delete() - File deleted");

        // move(Path source, Path target, CopyOption... options) - Moves or renames a file or
directory
        Path sourcePath = Paths.get("source.txt");
        Path targetPath = Paths.get("target.txt");
        Files.move(sourcePath, targetPath, StandardCopyOption.REPLACE_EXISTING);
        System.out.println("move() - File moved");

        // exists(Path path, LinkOption... options) - Checks if the file or directory exists
        boolean exists = Files.exists(targetPath);
        System.out.println("exists() - File exists: " + exists);

        // isRegularFile(Path path, LinkOption... options) - Checks if the path is a regular file
        boolean isRegularFile = Files.isRegularFile(targetPath);
        System.out.println("isRegularFile() - Is it a regular file: " + isRegularFile);

        // isDirectory(Path path, LinkOption... options) - Checks if the path is a directory
        boolean isDirectory = Files.isDirectory(targetPath);
        System.out.println("isDirectory() - Is it a directory: " + isDirectory);

        // isReadable(Path path) - Checks if the file is readable
        boolean isReadable = Files.isReadable(targetPath);
        System.out.println("isReadable() - Is it readable: " + isReadable);

        // isWritable(Path path) - Checks if the file is writable
        boolean isWritable = Files.isWritable(targetPath);
        System.out.println("isWritable() - Is it writable: " + isWritable);

        // readAttributes(Path path, String attributes, LinkOption... options) - Reads the
attributes of a file as a bulk operation
        BasicFileAttributes attr = Files.readAttributes(targetPath, BasicFileAttributes.class);
```

```
            System.out.println("readAttributes() - Creation Time: " + attr.creationTime());

            // readAllBytes(Path path) - Reads all bytes from a file
            byte[] fileBytes = Files.readAllBytes(targetPath);
            System.out.println("readAllBytes() - Bytes read: " + fileBytes.length);

            // readAllLines(Path path) - Reads all lines from a file
            List<String> lines = Files.readAllLines(targetPath);
            System.out.println("readAllLines() - Lines read: " + lines.size());

            // isExecutable(Path path) - Checks if the file is executable
            boolean isExecutable = Files.isExecutable(targetPath);
            System.out.println("isExecutable() - Is it executable: " + isExecutable);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

java.nio.file.Path Method	Description
getFileName()	Retrieves the name of the file or directory denoted by this path as a Path object.
getParent()	Returns the parent path, or null if this path does not have a parent.
getRoot()	Returns the root component of this path as a Path object, or null if this path does not have a root component.
toAbsolutePath()	Returns the absolute path representation of this path.
resolve(Path other)	Resolves the given path against this path.
normalize()	Normalizes this path.
subpath(int beginIndex, int endIndex)	Returns a relative Path that is a subsequence of the name elements of this path.
startsWith(Path other)	Tests if this path starts with the given path.
endsWith(Path other)	Tests if this path ends with the given path.
relativize(Path other)	Constructs a relative path between this path and a given path.
toFile()	Returns a File object representing this path if this path represents an absolute path.

```
import java.nio.file.*;

public class PathMethodsExample {
    public static void main(String[] args) {
```

```
        // Create a Path object
        Path path = Paths.get("/Users/username/Documents/testfile.txt");

        // getFileName() - Retrieves the name of the file or directory denoted by this path as a
Path object
        Path fileName = path.getFileName();
        System.out.println("getFileName() - File Name: " + fileName);

        // getParent() - Returns the parent path, or null if this path does not have a parent
        Path parent = path.getParent();
        System.out.println("getParent() - Parent Path: " + parent);

        // getRoot() - Returns the root component of this path as a Path object, or null if this
path does not have a root component
        Path root = path.getRoot();
        System.out.println("getRoot() - Root Path: " + root);

        // toAbsolutePath() - Returns the absolute path representation of this path
        Path absolutePath = path.toAbsolutePath();
        System.out.println("toAbsolutePath() - Absolute Path: " + absolutePath);

        // resolve(Path other) - Resolves the given path against this path
        Path resolvedPath = path.resolve(Paths.get("subfolder/file.txt"));
        System.out.println("resolve() - Resolved Path: " + resolvedPath);

        // normalize() - Normalizes this path
        Path normalizedPath = Paths.get("/test/../dir").normalize();
        System.out.println("normalize() - Normalized Path: " + normalizedPath);

        // subpath(int beginIndex, int endIndex) - Returns a relative Path that is a subsequence
of the name elements of this path
        Path subPath = path.subpath(1, 3);
        System.out.println("subpath() - Sub Path: " + subPath);

        // startsWith(Path other) - Tests if this path starts with the given path
        boolean startsWithPath = path.startsWith(Paths.get("/Users"));
        System.out.println("startsWith() - Starts with '/Users': " + startsWithPath);

        // endsWith(Path other) - Tests if this path ends with the given path
        boolean endsWithPath = path.endsWith(Paths.get("testfile.txt"));
        System.out.println("endsWith() - Ends with 'testfile.txt': " + endsWithPath);

        // relativize(Path other) - Constructs a relative path between this path and a given path
        Path relativePath = Paths.get("/home").relativize(Paths.get("/home/user/docs"));
        System.out.println("relativize() - Relative Path: " + relativePath);
```

```
        // toFile() - Returns a File object representing this path if this path represents an
absolute path
        try {
            File file = path.toFile();
            System.out.println("toFile() - File representation: " + file);
        } catch (UnsupportedOperationException e) {
            System.out.println("toFile() - Conversion not supported for this path");
        }
    }
}
```

Class Relationships

Relationship	Description	Example	Code

Inheritance	"**Is-a**" relationship where one class inherits properties and behaviors from another class.	Car is a Vehicle	public class Vehicle {
			  // properties and methods common to all vehicles
			}
			
			public class Car extends Vehicle {
			  // additional properties and methods specific to cars
			}

Composition (Strong)	"**Has-a**" relationship where one class contains an instance of another class as part of its attributes. It is a part-whole relationship: the whole contains component parts.	Car has an Engine (**when car is scrapped, engine is usually**	public class Engine {
			  // properties and methods related to the engine
			}
			
			public class Car {
			  private Engine engine;
			  public Car() { this.engine = **new Engine();** }
			  // other properties and methods

Relationship	Description	Example	Code
		disposed of as well	) }

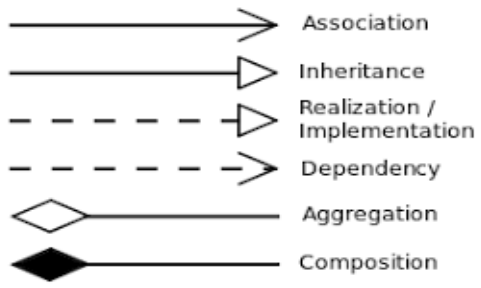
Association(“**uses**”)	Relationship between two separate classes, indicating how they are related or interact with each other. It is another way one class can use the services of another: some third party (a class, or the main program) passes an object of one class type as a parameter to a method of a object of another class type	Person is associated with Address	public class Person {
			  // properties and methods of a person
			  public Person(String name, **Address address**)
			  { this.name = name; this.address = address; }
			}
			
			public class Address {
			  // properties and methods related to an address
			}
			
			public class PersonAddress {
			  private Address address = new Address();
			  private Person person = new Person(address);
			
			  // methods to associate a person with an address
			}

Aggregation (Weak)(“**has copies of**”)	Aggregation is also about one object containing others, but the idea here is a data structure: multiple objects of the same type	Department has Employees (but **employees can exist independently**)	public class Department {
			  private String name;
			  private List<Employee> emp;
			  // Other attributes
			  // Constructor, getters, setters, etc.
			}
			
			public class Employee {
			  private String name;
			
			  // Constructor, getters, setters, etc.
			}

dependency (“temporary use”)	A dependency relationship is when one class needs the services of another class, but doesn't need to contain it permanently		// Dependency example
			public class A {
			  public void doSomething(B objB) {
			  // Class A depends on class B by using it as a parameter
			  objB.performAction();
			  }
			}
			
			public class B {
			  public void performAction() {
			  // Some action performed by class B
			  }
			}

Relationship	Description	Example	Code

			// Usage
			A objA = new A();
			B objB = new B();
			objA.doSomething(objB); // Class A depends on class B by using it as a method parameter



Difference b/w Composition and Aggregation:

Composition	Aggregation

public class Engine {
}
 	
public class Car {
  private Engine engine;
 	
  public Car() {
  this.engine = new Engine();
  }
}
	public class Department {
	}
	
	public class University {
	  private List<Department> departments;
	
	  public University() {
	  this.departments = new ArrayList<>(); /
	  }
	
	  public void addDepartment(Department department) {
	  departments.add(department);
	  }
	}

In this example, the Car class contains an Engine object as a part of its composition. The Car class owns the Engine, and when a Car object is created, it also creates an	In this example, the University class has a list of Department objects as part of its aggregation. The University class has a relationship with Departments, but the Department objects can

Engine object as part of its initialization. The Engine object's existence is tied directly to the Car.	exist independently. They can be created, modified, or destroyed without directly impacting the existence of the University.

Dependency	Association
Indicates that one class uses another class	Represents a direct relationship between classes

```
// Dependency example
public class A {
    public void doSomething(B objB) {
        // Class A depends on class B by using it as a parameter
        objB.performAction();
    }
}

public class B {
    public void performAction() {
        // Some action performed by class B
    }
}

// Usage
A objA = new A();
B objB = new B();
objA.doSomething(objB); // Class A depends on class B by using it as a method parameter
```

```
public class Person {
    // properties and methods of a person
}

public class Address {
    // properties and methods related to an address
}

public class PersonAddress {
    private Person person;
    private Address address;
    // methods to associate a person with an address
}
```

Class Inheritance

```
package Learn;

// can have multiple children
class Vehicle { // parent/super/base class
        private String brand;
        private int year;

        public Vehicle(String brand, int year) {
                this.brand = brand;
                this.year = year;
        }

        public void startEngine() {
                System.out.println("The engine is started.: parent class");
        }
}

// if other class extends Car then Vehicle becomes grandparent
class Car extends Vehicle { // child/sub/derived class
        // Car is a specialized type of Vehicle
        private int numberOfDoors;
```

```
        public Car(String brand, int year, int numberOfDoors) {
                super(brand, year);
                this.numberOfDoors = numberOfDoors;
        }

        // Method overriding
        public void startEngine() {
                System.out.println("The engine is started.: child class");
        }

        public void honkHorn() {
                System.out.println("Honk! Honk!");
        }
}

class Person {
        // Person has a Vehicle
        private Vehicle vehicle;

        public Person(Vehicle vehicle) {
                this.vehicle = vehicle;
        }

        public void drive() {
                System.out.println("The person is driving a car.");
        }
}

public class Inheritance {
        // Need?
        // Person has a Vehicle -> 'has' => Composition
        // Car is a Vehicle -> 'is' => Inheritance
        public static void main(String[] args) {

                // Creating a Car object
                Car myCar = new Car("Toyota", 2023, 4);

                // Creating a Person object with a Car
                Person person = new Person(myCar);

                // Demonstrating "is-a" relationship
                myCar.startEngine(); // Inherited method from Vehicle
                myCar.honkHorn(); // Car-specific method

                // Demonstrating "has-a" relationship
                person.drive(); // Person has a Car

                // Dog extends Animal -> Single Inheritance
                // Labrador extends Dog, Dog extends Animal -> multilevel inheritance

                // multiple Inheritance??? -> NOT possible in java -> Ambiguity problem
                // reason: Suppose [class A extends B,C] and there is method called hello() in
                // both B and C then which one will be executed using object of A.
```

```
		// Method overriding
		System.out.println("*****Method Overriding*****");
		Vehicle v = new Vehicle("my branch", 2);
		v.startEngine();
		// first JVM searches for method in child class, if found then execute otherwise
		// check in parent class.
		myCar.startEngine();
	}
}
```

```
package Learn;

//every class in java extends 'Object' class by default even if you don't mention as below.
class A extends Object {
	private int ina;

	public A() {
		// following super method exists in first line of every constructor in java by
		// default.
		// super(); // this refers to the constructor of Object class
		System.out.println("Inside A class: 0 param");
	}

	public A(int ina) {
		System.out.println("Inside A class: 1 param");
		this.ina = ina;
	}

	public int getIna() {
		return ina;
	}

	public void setIna(int ina) {
		this.ina = ina;
	}

}

// B class is not not extending 'Object' class, we have defined A class explicitly.
// B extends A and A extends Object -> this is called multilevel inheritance.
class B extends A {
	private int inb;

	public B() {
		super(); // this method exists in first line of every constructor in java by default.
		System.out.println("Inside B class: 0 param");
		// super();  // error: Constructor call must be the first statement in a constructor
	}

	public B(int ina) {
		// by default super() will be called, that is default constructor of class A, if
		// you want to call overloaded constructor of A then explicitly define as below.
		super(ina);
```

```
                System.out.println("Inside B class: 1 param");
        }

        public B(int ina, int inb) {
                // if you want to call another constructor of same class then use this keyword
                // as below.
                this(ina);
                System.out.println("Inside B class: 2 param");
                this.inb = inb;
        }

        public int getInb() {
                return inb;
        }

        public void setInb(int inb) {
                this.inb = inb;
        }

}

public class This_and_Super {
        public static void main(String[] args) {
                // constructor of both class will be called

                System.out.println("**********");
                B b = new B(1, 2);
                System.out.println("**********");
                B b1 = new B(5);
                System.out.println("**********");
                B b2 = new B();
                System.out.println("Inside B class");
                System.out.println(b.getIna());
                System.out.println(b.getInb());

        }
}
```

```
public class Parent {
   public Parent(int value) {
      this.value = value;
   }

}
public class Child extends
Parent {
   public Child() {
      do some more code;
   }
}
```

Child causes a compiler error: no default constructor

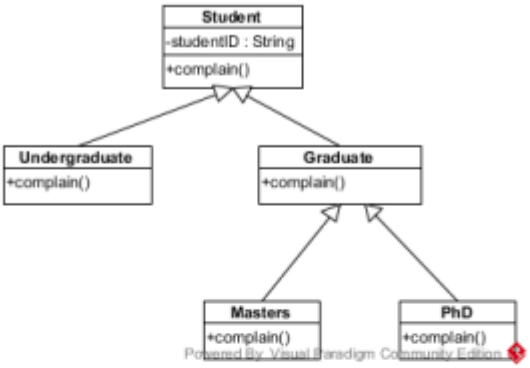
Polymorphism

```
package Learn;

class PolyParent {
	public void show() {
		System.out.println("*parent*");
	}
}

class PolyChild1 extends PolyParent {

	// overriding
	public void show() {
		System.out.println("*child*");
	}

	// overloading
	public void show(int a) {
		System.out.println("*overloading*");
	}
}

public class Polymorphism {
	// Polymorphism => many + behavior
	// two type: compile time(early binding) & run time(late binding)

	// overloading -> part of compile time polymorphism
	// overriding, dynamic method dispatch -> part of run time polymorphism

	public static void main(String[] args) {
		PolyChild1 c1 = new PolyChild1();
		c1.show();
		// PolyChild1 is PolyParent but vice versa is not true.
		// PolyChild1 c2 = new PolyParent();
		// reference variable of super class and object of sub class
		PolyParent c2 = new PolyChild1();
		c2.show();

		// dynamic method dispatch -> below thing is called dynamic method dispatch.
		PolyParent c3 = new PolyParent();
		c3.show();
		// following line only works when you have inheritance(PolyChild1 is extending
		// PolyParent).
		c3 = new PolyChild1(); // reference variable in main stack will update its address from
PolyParent's
                                                                // object address to PolyChild1's object
address
```

```
            c3.show();
        }
}
```

```
PhD phd = new PhD();
phd.complain(); // Phd::complain
phd.super.complain(); // ERROR

// inside class Phd:
public void complain() {
    // OK: Graduate::complain
    super.complain();
    // ERROR: Student::complain
    super.super.complain();
}
```

Constructor




```
PhD phd = new PhD();
```

-> calls PhD constructor
-> calls Graduate constructor
-> calls Student constructor
-> allocates studentID and sets it to null

	Private	Protected	Public	Default
Same class	Yes	Yes	Yes	Yes
Same package subclass	NO	Yes	Yes	Yes
Same package non-subclass	NO	Yes	Yes	Yes
Different package subclass	NO	Yes	Yes	NO
Different package non-subclass	NO	NO	Yes	NO

Abstact class

```
package Learn;


//A class with an abstract method must be marked as abstract
```

```java
// An abstract class is not required to have any abstract methods
abstract class absClass {
	public void m1() {
		// not sure how to implement?? -> no code written here

		// this is method declaration
		// if you don't know implementation then instead of declaring method, define it
		// like m3.
	}

	public void m2() {
		System.out.println("method2");
	}

	// I'm giving abstract idea of feature, you have to define it when you extend.
	public abstract void m3();

	public abstract void m4();

}

abstract class myAbs extends absClass {

	// when you extend abstract class, you have to implement all the abstract method
	// of base class -> if you don't want that then define this class also as
	// abstract class
	@Override
	public void m3() {
		// TODO Auto-generated method stub
		System.out.println("method3");

	}

}

class myAbs2 extends myAbs { // concrete class -> its object can be created and extends abstract class
	@Override
	public void m4() {
		// TODO Auto-generated method stub
		System.out.println("method4");

	}

}

public class AbstactKeyword {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

		// object of abstract class can not be created.
//		absClass ab = new absClass();
//		ab.m1();
//		ab.m2();
		absClass abs = new myAbs2();
```

```
                abs.m3();
                myAbs2 mab = new myAbs2();
                mab.m1();
                mab.m2();

                // nested abstract classes are also possible (class -extends--> abstract class
                // -extends--> abstract class)
        }

}
```

Encapsulation

-> implies restricting access to the data to ensure its integrity, also known as data hiding
-> only allow controlled changes to the data
-> Breaking encapsulation occurs whenever you give a "back door" way to change a class's data directly from outside the class without using a method to access it

- you make data public or package private
- you return a reference to member data (not a primitive)
- you retain a reference outside (like the main program) to data being set as member data

Shallow copy vs Deep copy

A shallow copy creates a new object but copies only the references to the original objects. It doesn't create copies of the internal objects. If the original object contains references to other objects, the shallow copy will reference the same objects.
A deep copy, on the other hand, creates a new object and recursively copies all the objects that are referenced by the original object. It creates new instances of the internal objects, ensuring that changes in the copied object do not affect the original.

-> data member : private + getter setter + no return of reference variable
    -> if you want to use parent class member data directly in child class then declare them as protected
-> methods:
    -> private if it is helper methods or utility functions that should not be called from outside othewise public/protected/.. based on requirement(should be accessible outside)

-> constructor: generally public but in some design patterns like factory we keep it private.
-> class: either public or package

Final Keyword

```
package Learn;
class  Cal1{
	public void show() {
		System.out.println("show");
	}
	public final void add(int a, int b) {
		System.out.println("add: "+ (a+b));
	}
}
class advCal1 extends Cal1 {
	public void show() {
		System.out.println("show");
	}

	// we can not override final methods
//	public void add(int a, int b) {
//		System.out.println("add: "+ (a+b));
//	}
}
final class  Cal2{
	public void show() {
		System.out.println("show");
	}
	public final void add(int a, int b) {
		System.out.println("add: "+ (a+b));
	}
}
//final class can not be inherited.
//class advCal2 extends Cal2 {
//
//}
public class Final {
	// final keyword can be used with variable/method/class.
	public static void main(String[] a) {
		final int i = 9;
		// final variable values can not be changed.
		// i=10;
		Cal1 c = new Cal1();
		c.add(1,2);
	}
}
```

Immutable Data: Overloaded constructor is one-time setter+ variable : private and final + no setter
Useful for data that shouldn't change, like a student's id

Static

```
package Learn;

class Mobile {
	// instance variables -> stored inside heap of particular object
	String brand;
	int price;
	static String name; // it belongs to class not object


	public Mobile() {
		System.out.println("**********Inside constructor**********");
		brand = "";
		price = 200;
		name = "phone";
	}


	// instance method
	public void show() {
		System.out.println("Brand ::" + brand);
		System.out.println("price ::" + price);
		System.out.println("name ::" + name);
		System.out.println();
	}

	public static void show1() {
		// non static variables can not be used inside static method
		// because when you call Mobile.show1(), JVM won't figure out which branch and
		// price variable to use.
		// System.out.println("Brand ::" + brand);
		// System.out.println("price ::" + price);
		System.out.println("name(show1) ::" + name);
		System.out.println();
	}

	// how to use non-static variables inside static method
	public static void show2(Mobile obj) {
		System.out.println("Brand ::" + obj.brand);
		System.out.println("price ::" + obj.price);
		System.out.println("name ::" + name);
		System.out.println();
	}

}
public class Static {

	// why main method is static?
	// ->if main is non-static then we can not call it without object of
	// Static(class name in which main is defined)
	// class define above it. and main is starting point of execution so, if we need
	// to create object of Static class to call it, where can we create without
	// starting point? kind of deadlock situation if main is non-static.
```

```
	public static void main(String[] args) throws ClassNotFoundException{
		// TODO Auto-generated method stub
		// for obj1 and obj2 both space will be allocated in heap.
		// static -> if any instance variable is static then that will be stored as
		// common between all objects inside heap and will be shared by all objects.
		// for obj1 and obj2 JVM will create two different objects with brand and price
		// variable but name variable will be stored as common between both the objects
		// as it is static
		Mobile obj1 = new Mobile();
		obj1.brand = "Apple";
		obj1.price = 1500;
		obj1.name = "smart phone";

		Mobile obj2 = new Mobile();
		obj2.brand = "Apple1";
		obj2.price = 1501;
		obj2.name = "smart phone 2"; // changing name variable of obj2 will change value for obj1 also

		// as name is static we can call it with class name also.
		Mobile.name = "class name";
		obj1.show();
		obj2.show();

		obj1.show1();
		// static method call
		Mobile.show1();
		Mobile.show2(obj1);

	}

}
```

Casting

```
package Learn;

class base_class {
	public void show_base() {
		System.out.println("show base");
	}

}
class child_class extends base_class {
	public void show_child() {
		System.out.println("show child");
	}

}
public class Casting {
	public static void main(String[] args) {
		double d = 4.5;
		int i = (int) d; // type casting
		System.out.println(i); // data loss
```

```
                base_class base = new base_class();
                base.show_base();
                // error
                // base.show_child();
                child_class child = new child_class();
                child.show_base();
                child.show_child();
                base_class base_child = (base_class) new child_class(); // upcasting -> implicite
                base_child.show_base();
                // error
                // base_child.show_child();
                child_class child_base =(child_class) base_child; // downcasting
                // now show_child can be called.
                child_base.show_child();
        }
}
```

Not all methods can be overridden:

private – just not allowed: automatically final
final – because that's what final means
static – actually, a child can have a static method of the same name, but it's not dynamically bound – it's a new method

Interface

```
package Learn;



abstract class abstractClass {

        // abstract class can have (0 or more) abstract method + (0 or more) normal
        // method.
        public abstract void m1();

        public void m2() {
                System.out.println("method2");
        }
}
interface myInterface {
        // ->If in abstract class all the methods are abstract then that can also be
        // defined by interface.
        // ->Interface is not a Class, while abstract class is a Class
        // -> by default every method of an interface is 'public abstract'
        public abstract void m1();

        public void m2();

        void m3();

        // interface can have variables, by default they are 'final and static'
```

```
	// reason for this is, we don't create instance of interface and when we
	// implement interface, we override methods and not variables. so, If it is not
	// final and static then we never get a chance to assign values to it.
	int i = 44;

	// following will give an error, because final variables should be assigned some
	// value.
	// int j;
}
interface otherInterface{
	void m4();
}
// Like classes, Interface can extend other Interface("extends")
interface hello extends otherInterface
{
	void m5();
}
// Unlike classes, multiple inheritance is also possible
interface multipleInheritance extends otherInterface,myInterface
{
	void m6();
}
// all methods of interface should be overridden if class is not an abstract class.
class classInterface implements myInterface, otherInterface {

	@Override
	public void m1() {
		// TODO Auto-generated method stub
		System.out.println("m1");

	}

	@Override
	public void m2() {
		// TODO Auto-generated method stub
		System.out.println("m2");

	}

	@Override
	public void m3() {
		// TODO Auto-generated method stub
		System.out.println("m3");

	}

	@Override
	public void m4() {
		// TODO Auto-generated method stub
		System.out.println("m4");

	}

}
```

```
public class Interface {
	public static void main(String[] args) {
		myInterface interfaceAnonymousInnerClass = new myInterface() {
			@Override
			public void m3() {
				// TODO Auto-generated method stub
				System.out.println("anonymous Inner m3");

			}

			@Override
			public void m2() {
				// TODO Auto-generated method stub
				System.out.println("anonymous Inner m2");

			}

			@Override
			public void m1() {
				// TODO Auto-generated method stub
				System.out.println("anonymous Inner m1");
			}
		};
		interfaceAnonymousInnerClass.m1();
		interfaceAnonymousInnerClass.m2();
		interfaceAnonymousInnerClass.m3();
		classInterface cin = new classInterface();
		cin.m1();
		cin.m2();
		cin.m3();
		// access interface variables
		System.out.println(myInterface.i);
		// interface variables are final, so, can not be changed.
		// myInterface.i = 3;

		// note: class can implements more than one interface at a time
		// note: interface can extends other interface same as classes.

		// interface -extends--> interface
		// class -implements--> interface
		// class -extends--> class

		otherInterface other = new classInterface();
		other.m4(); // m1, m2 and m3 can not be called by this object.
		myInterface my = (myInterface)other;
		my.m1(); // m4 can not be called by this object.
		my.m2();
		my.m3();

	}
}
```

Comparable, Comparator

Comparable	Comparator	Annonymous Inner class

import java.util.*;
class Student implements Comparable<Student> {
  private int id;
  private String name;
  public Student(int id, String name) {
  this.id = id;
  this.name = name;
  }
 	
  @Override
  public int compareTo(Student otherStudent) {
  return Integer.compare(this.id, otherStudent.getId());
  }
 	
  public static void main(String[] args) {
  List<Student> students = new ArrayList<>();
  students.add(new Student(3, "Alice"));
  students.add(new Student(1, "Bob"));
  students.add(new Student(2, "Charlie"));
 	
  Collections.sort(students); // Uses Comparable
 	
  }
}
	package Learn;
	class Student {
	  private final int id;
	  private final String name;
	  public Student(int id, String name) {
	  this.id = id;
	  this.name = name;
	  }
	}
	
	class StudentIdComparator implements Comparator<Student> {
	  @Override
	  public int compare(Student student1, Student student2) {
	  return Integer.compare(student1.getId(), student2.getId());
	  }
	}
	
	public class Interface {
	  public static void main(String[] args) {
	  List<Student> students = new ArrayList<>();
	  students.add(new Student(3, "Alice"));
	  students.add(new Student(1, "Bob"));
	  students.add(new Student(2, "Charlie"));
	
	  StudentIdComparator idComparator = new StudentIdComparator();
	
	  Collections.sort(students, idComparator);
	  }
	}
		import java.util.*;
		class Student {
		  private int id;
		  private String name;
		  public Student(int id, String name) {
		  this.id = id;
		  this.name = name;
		  }
		
		  public static void main(String[] args) {
		  List<Student> students = new ArrayList<>();
		  students.add(new Student(3, "Alice"));
		  students.add(new Student(1, "Bob"));
		  students.add(new Student(2, "Charlie"));
		
		  Comparator<Student> idComparator = new Comparator<Student>() {
		  @Override
		  public int compare(Student s1, Student s2) {
		  return Integer.compare(s1.getId(), s2.getId());
		  }
		  };
		
		  students.sort(idComparator);
		
		  for (Student student : students) {
		  System.out.println("ID: " + student.getId() + ", Name: " + student.getName());
		  }
		  }
		}

Dependency Injection

```
package Learn;

interface Shape {
        void draw();
}

// This class Drawing has a field named shape of type Shape, which can be set using
// constructor injection (in the constructor) or setter injection (using a setter method).
// This flexibility allows the Drawing class to work with any object that implements the Shape
// interface. For instance, if you have classes like Circle, Square, or Triangle that implement
// the Shape interface, you can create an instance of Drawing and pass any of these objects to it
class Drawing {
        private Shape shape;

        // Constructor injection
        public Drawing(Shape shape) {
```

```
                this.shape = shape;
        }

        // Setter injection
        public void setShape(Shape shape) {
                this.shape = shape;
        }

        public void drawShape() {
                if (shape != null) {
                        shape.draw();
                } else {
                        System.out.println("No shape to draw!");
                }
        }
}

class Circle implements Shape {
        @Override
        public void draw() {
                System.out.println("Drawing Circle");
        }
}

class Square implements Shape {
        @Override
        public void draw() {
                System.out.println("Drawing Square");
        }
}

public class DependencyInjection {

        public static void main(String[] args) {
                Drawing drawing = new Drawing(new Circle());
                drawing.drawShape(); // Output: Drawing Circle

                drawing.setShape(new Square());
                drawing.drawShape(); // Output: Drawing Square
        }

}
```

Reflection

```
package Learn;
```

```
import java.lang.reflect.Field;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.lang.reflect.Type;

class Cat {
	private final String name;
	private int age;
	public Cat() {
		this.name = "final";

   }
	public Cat(String name, int age) {
		this.name = name;
		this.age = age;
	}

	public int getAge() {
		return age;
	}

	public void setAge(int age) {
		this.age = age;
	}

	public String getName() {
		return name;
	}

	public void publicMethod() {
		System.out.println("Public method without param");
	}

	public void publicMethodWithParam(int a, String st) {
		System.out.println("Public method with param");
	}

	private void privateMethod() {
		System.out.println("Private method");
	}

	public static void publicStaticMethod() {
		System.out.println("Public static method");
	}

	private static void privateStaticMethod() {
		System.out.println("Private static method");
	}
}

public class Reflection {

	public static void main(String[] args) {
		// reflection is a feature that allows you to inspect and manipulate classes,
		// interfaces, fields, methods, and constructors at runtime. It provides a way
```

```
		// to examine and modify the structure and behavior of classes, even if you
		// don't have access to their source code.
//		Cat myCat = new Cat("HiCat", 1);
		try {
			// Dynamically load the class using Class.forName()
			Class cat = Class.forName("Learn.Cat");

			// Instantiate the class using newInstance()
			Cat myCat = (Cat)cat.getDeclaredConstructor().newInstance();
//			Cat c = new Cat();
			// Cast the instance to your class type and invoke a method
			if (myCat instanceof Cat) {
				Cat obj = (Cat) myCat;
				obj.publicMethod();
			}

			Field[] fields = cat.getDeclaredFields();
			for (Field field : fields) {
				System.out.println(field.getName());
			}
			System.out.println();

			// name can not be changed as it is final and private.
			// myCat.name = "changeName";
			// But with reflection without making any changes to actual code of Cat class,
			// we can force java to change the name of Cat object
			System.out.println("Before: " + myCat.getName());

			for (Field field : fields) {
				if (field.getName().equals("name")) {
					field.setAccessible(true);
					try {
						field.set(myCat, "newName");
					} catch (IllegalArgumentException | IllegalAccessException e) {
						e.printStackTrace();
					}
				}
			}
			// we changed final and private variable
			System.out.println("After: " + myCat.getName());

			System.out.println();

			// same as fields, we can also get all the method details of the class.
			Method[] methods = cat.getDeclaredMethods();
			for (Method method : methods) {
				System.out.println(method.getName());
			}
			System.out.println();

			// we can trigger any method using reflection
			for (Method method : methods) {
				// invoke method without parameter
				if (method.getName().equals("publicMethod")) {
					try {
```

```
                                        method.invoke(myCat);
                                } catch (IllegalAccessException | IllegalArgumentException |
InvocationTargetException e) {
                                        e.printStackTrace();
                                }
                        }

                        // invoke method with parameter
                        if (method.getName().equals("publicMethodWithParam")) {
                                try {
                                        method.invoke(myCat, 1, "myString");
                                        for(Type t: method.getParameterTypes()) {
                                                System.out.println("--"+t);
                                                System.out.println(method.getReturnType( ));

                                        }
                                } catch (IllegalAccessException | IllegalArgumentException |
InvocationTargetException e) {
                                        e.printStackTrace();
                                }
                        }

                        // invoke method private method
                        if (method.getName().equals("privateMethod")) {
                                method.setAccessible(true);
                                try {
                                        // private method can to called using reflection
                                        method.invoke(myCat);
                                        // private method can not be called as below
                                        // myCat.privateMethod();
                                } catch (IllegalAccessException | IllegalArgumentException |
InvocationTargetException e) {
                                        e.printStackTrace();
                                }
                        }

                        // invoke static method
                        if (method.getName().equals("publicStaticMethod")) {
                                try {
                                        // we don't need class object to call static method, so pass
null
                                        method.invoke(null);
                                } catch (IllegalAccessException | IllegalArgumentException |
InvocationTargetException e) {
                                        e.printStackTrace();
                                }
                        }

                        if (method.getName().equals("privateStaticMethod")) {
                                method.setAccessible(true);
                                try {
                                        // we don't need class object to call static method, so pass
null
                                        method.invoke(null);
                                } catch (IllegalAccessException | IllegalArgumentException |
```

```
InvocationTargetException e) {
                                                e.printStackTrace();
                                        }
                                }
                        }
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }

}
```

Generics

```
package Learn;

class Box<T> {
        private T data;

        public  void setData(T data) {
                this.data = data;
        }

        public T getData() {
                return data;
        }

        public static <T> void doTheThing(T parameter) {
                // This method can perform actions on the parameter of type T
                // But it's not immediately clear what methods can be called on T
                // due to the generic nature of T.
                System.out.println("Doing the thing with: " + parameter);
        }
}

// The issue here is that within the Container2 class, dataItem is of type T, which could be any class.
// If doTheThing() is not a method that exists on every possible type that could be substituted for T,
// calling dataItem.doTheThing() will cause a compilation error because the compiler cannot guarantee that
// this method exists for all potential types.
class Container2<T> {
        private T dataItem;

        public Container2(T dataItem) {
                this.dataItem = dataItem;
        }

        public void doSomething() {
                // This attempts to call a method on dataItem, but it's generic, so we can't be
                // sure
                // if the method exists on every possible type T could be substituted with.
                // This line will cause a compilation error.
```

```
                // dataItem.doTheThing();
        }

        public T getDataItem() {
                return dataItem;
        }
}

interface Doable {
        void doTheThing();
}

class Container<T extends Doable> {
        private T dataItem;

        public Container(T dataItem) {
                this.dataItem = dataItem;
        }

        public void doSomething() {
                // Now, because T extends Doable, we can be sure that any object
                // assigned to dataItem will have the doTheThing() method.
                dataItem.doTheThing();
        }

        public T getDataItem() {
                return dataItem;
        }
}

//Sample classes implementing the Doable interface
class DoableClass implements Doable {
        @Override
        public void doTheThing() {
                System.out.println("Doing the thing in DoableClass");
        }
}

class AnotherDoableClass implements Doable {
        @Override
        public void doTheThing() {
                System.out.println("Doing the thing in AnotherDoableClass");
        }
}

public class Generics {
        public static void main(String[] args) {
                Box<String> str = new Box();
                Box<Integer> i = new Box();

                str.setData("hi");
                i.setData(2);
                // Attempting to call methods on the generic parameter
      // Uncommenting this will result in a compilation error
      // i.doTheThing("hi").getClass(); // This won't compile
```

```
		// Creating a Container with a class that implements Doable interface
		DoableClass doableObj = new DoableClass();
		Container<DoableClass> doableContainer = new Container<>(doableObj);
		doableContainer.doSomething(); // Calls doTheThing() method

		// Creating a Container with a class that doesn't implement Doable interface
		// This will result in a compilation error
		// UnDoableClass unDoableObj = new UnDoableClass();
		// Container<UnDoableClass> unDoableContainer = new Container<>(unDoableObj);

		// You can also have other classes implementing Doable interface
		AnotherDoableClass anotherDoableObj = new AnotherDoableClass();
		Container<AnotherDoableClass> anotherDoableContainer = new
Container<>(anotherDoableObj);
		anotherDoableContainer.doSomething(); // Calls doTheThing() method

	}
}
```

Exception

```
class exceptionDC extends Exception {
	public exceptionDC(String s) {
		super(s);
	}
}

public class ExceptionHandling {
	// compile time error
	// runtime error
	// logical error -> wrong logic/bug
	// exception -> can/cannot be recovered (e.g., ArithmeticException(can or cannot
	// be solved, depends on logic ..)
	// error -> cannot be recovered/handled (e.g., OutOfMemoryError)

	// if some method is throwing exception, while calling that method you have to
	// write try catch or re-throw it

	public static void main(String[] args) {
//		System.out.println(9 / 0);
//		System.out.println(
//					"this will not be executed as above statement is not in try catch and will throw an
exception");
		Pattern p = Pattern.compile("dog");
		String s = "My dog has fleas";
		Matcher m = p.matcher(s);
		if ( m.find() ) {
		System.out.println("Found a match in " + s);
		System.out.println(s.substring(m.start(), m.end()));
		}
		try {
			System.out.println(9 / 1);
		} catch (ArithmeticException e) {
```

```
                        System.out.println("will be executed only if Exception occured");
                } finally {
                        System.out.println("---will always be executed");
                }
                System.out.println("This will be printed as exception is in try catch block");

                // multiple catch -> write specific exceptions first and then generic
                try {
                        int num[] = new int[5];
                        System.out.println(num[5]);

                } catch (ArithmeticException e) {// specific
                        System.out.println("first exception");
//                      System.out.println(e);
                } catch (Exception e) {// generic
                        System.out.println("second exception");
//                      e.printStackTrace();
                }

                // hierarchy
                // object class -> throwable class -> error class, exception class
                // exception class -> runtime exception, SQLexception, IO exception...
                // all runtime exceptions(Arithmetic, arrayIndexOutOfBound, nullPointer..) are
                // unchecked exception

                // checked vs unchecked
                // checked -> need to handle/compiler will ask to handle
                // unchecked -> handling it is not mandatory

                // throw and throws

                try {
                        reThrowingException(); // need to handle as it is throwing exception
                } catch (Exception e) {
                        // TODO Auto-generated catch block
                        System.out.println(e.getMessage());

                }
                handlingException(); // no need to handle already handled

                try {
                        customException();
                } catch (exceptionDC e) {
                        // TODO Auto-generated catch block
                        System.out.println(e.getMessage());
                }

                notHandlingException(); // Arithmetic Exception is unchecked so, it won't ask to handle compulsory
        }

        public static void reThrowingException() throws Exception {
                passingToParentException(); // if you are not handling exception thrown by passingToParentException()
method
                                                                        // here, then need to
                                                                        // re-throw (mandatory in case of
checked exception)(ducking exception)
        }

        public static void handlingException() {
                try {
                        System.out.println(9 / 0);
```

```
                } catch (ArithmeticException e) {
                        System.out.println("divide by zero");
                }
        }

        public static void notHandlingException() {
                System.out.println(9 / 0);
        }

        // we are telling that following method might throw and exception and it needs
        // to be handled from wherever it gets called
        public static void passingToParentException() throws Exception {
                throw new Exception("Hello there!!!!!!!!!!!!!!!!!!!!!!!!!!!!"); // custom message
        }

        // custom Exception
        public static void customException() throws exceptionDC {
                throw new exceptionDC("It's me, exception DC");
        }
}
```

Unchecked	Checked
ArrayIndexOutOfBoundsException	FileNotFoundException
NullPointerException	IOException
ArithmeticException	ClassNotFoundException
IllegalArgumentException	
NumberFormatException	
IndexOutOfBoundsException	

Patterns

Hide object creation logic→ Factory Method
Creating Complex object → Builder
Incompatible interfaces to work together → Adapter
Simplify usage of different subsystems → Facade
Only one instance → Singleton
Different classes and Dynamic change → Strategy
controlled access to the original object/intermediary/adding an extra layer of control or functionality -> Proxy

Factory Design Patterns
https://www.javatpoint.com/factory-method-design-pattern#:~:text=A%20Factory%20Pattern%20or%20Factory,the%20instance%20of%20the%20class.

Strategy Design Pattern
https://www.javatpoint.com/strategy-pattern

Adapter + Facade + Builder + singleton - > mail

MVC
https://www.geeksforgeeks.org/mvc-design-pattern/

Proxy
https://www.javatpoint.com/proxy-pattern

Threading

core -> processing unit

concurrency -> more than one program + context switching + one core
parallel -> two or more program + two or more core

multiprocessing -> more than one core/cpu
multiprogramming -> concurrency
multithreading -> multiprogramming using thread

process -> running program
thread -> independent execution within process(can/cannot share memory)
-> main program is one thread, for more thread? -> create manually

Advantage	**Disadvantage**
speed up task	hard to debug
e.g, downloading different kind of data-> separate logic for each, sorting	shared memory b/w threads? -> consistency issue
	speed up may not be as expected

Amdahl's Law:

$$\text{Speedup} = \frac{1}{(1-P)+\frac{P}{S}}$$

Where:

- Speedup is the theoretical improvement in the execution time of the whole task.
- P is the proportion of the program that can be made parallel.
- S is the speedup of the parallelizable portion.

Ways to create threads:

1) implement runnable
2) extend Thread class
3) thread pool
->create a new thread for each new task + reuse thread
->create a set of threads initially + if #tasks>#threads then queue tasks + reuse thread

```
class myThread implements Runnable {
    private int i;
    public myThread(int i) {
        this.i = i;
```

```
	}

	@Override
	public void run() {
		for (int j = 0; j < 5; j++)
			System.out.println(i);
	}
}

class yourThread extends Thread {
	private int i;
	public yourThread(int i) {
		this.i = i;
	}

	@Override
	public void run() {
		for (int j = 0; j < 5; j++)
			System.out.println(i);
	}

}

class ValueReturningTask implements Callable<Integer> {
	private int a;
	private int b;

	public ValueReturningTask(int a, int b) {
		this.a = a;
		this.b = b;
	}

	@Override
	public Integer call() {
		// Perform some computation here
		for (int j = 0; j < 10; j++)
			System.out.println("#####");
		return a + b; // For instance, adding two numbers
	}
}

public class ThreadConcept {

	public static void main(String[] args) throws InterruptedException {
		// using runnable
		Thread t1 = new Thread(new myThread(1));
		t1.start();
		Thread t2 = new Thread(new myThread(2));
		t2.start();
		// if you remove join-> all four (t1,t2,t3,t4) threads will run in parallel.
		t1.join();
		t2.join();
		// using Thread class
		System.out.println("--------");
		yourThread t3 = new yourThread(3);
```

```
		t3.start();
		yourThread t4 = new yourThread(4);
		t4.start();

		t3.join();
		t4.join();
		System.out.println("--------");
		// Thread pool
		// 1) create a new thread for each new task, and reuse the threads after they
		// complete
		ExecutorService exec1 = Executors.newCachedThreadPool();
		// 2) create a set of threads initially, assign new tasks to threads, reuse as
		// above, but if there are more tasks than threads, the tasks are queued.
		ExecutorService exec2 = Executors.newFixedThreadPool(2);

		exec1.execute(new yourThread(5));
		exec1.execute(new yourThread(6));
		exec1.execute(new yourThread(7));
		exec2.execute(new yourThread(8));
		exec2.execute(new yourThread(9));
		exec2.execute(new yourThread(10));
		exec1.shutdown();
		exec2.shutdown();
		// exec1.execute(t4); // will throw runtime exeption

		System.out.println("--------");
		// Create an ExecutorService
		ExecutorService executorService = Executors.newSingleThreadExecutor();

		// Create a task that returns a value
		Callable<Integer> task = new ValueReturningTask(10, 20);

		// Submit the task to the executor service
		Future<Integer> futureResult = executorService.submit(task);

		// You can perform other operations here while the task is executing
		// asynchronously
		for (int j = 0; j < 10; j++)
			System.out.println("******");

		// Get the result once the task is completed
		try {
			Integer result = futureResult.get(); // This blocks until the result is available
			System.out.println("Result: " + result);
		} catch (InterruptedException | ExecutionException e) {
			e.printStackTrace();
		}

		// Shutdown the executor service
		executorService.shutdown();
	}

}
```

Race condition-> when shared data can become inconsistent (corrupted) due to unprotected updates from multiple threads.
solution: lock data -> only one thread should be able to access **shared** data at a time
-> thread locks data - use data - unlock for other
-> one thread can acquire the lock many times before another thread does

1) Make getters/setters synchronized
2) thread-safe data structure
   -> **java.util.concurrent** contains thread safe data structures like **ArrayBlockingQueue** and **ConcurrentHashMap**
   -> List<E> safeArrayList = Collections.synchronizedList(new ArrayList<E>()); // **wrapping**
3) explicit lock

```
package Learn;

class UnsafeSimpleData { // Wrapper for count
	private int count = 0;

	public synchronized void increment() {
		count++;
	}

	public int getCount() {
		return count;
	}
}

class SharedData implements Runnable {
	private UnsafeSimpleData data;

	public SharedData(UnsafeSimpleData data) {
		this.data = data;
	}

	public void run() {
		// Incrementing the shared integer without proper synchronization
		for (int i = 0; i < 10000; i++)
			data.increment();
	}

	public void print() {

		System.out.println(data.getCount());
	}
}


// using synchronized to avoid race condition
public class ThreadConcept2 {

	public static void main(String[] args) throws InterruptedException {
		UnsafeSimpleData s1 = new UnsafeSimpleData();
		// both Thread objects use s1 - i.e., shared data
		Thread t1 = new Thread(new SharedData(s1));
```

```
                t1.start();
                Thread t2 = new Thread(new SharedData(s1));
                t2.start();
                t1.join();
                t2.join();
                System.out.println(s1.getCount());
        }

}
```