1. Strings in Java

In Java, a String represents a sequence of characters. Strings in Java are objects of the String class, which is immutable—once created, their values cannot be changed.

String Handling Functions

Java provides several methods to manipulate strings. Here are some of the most commonly used methods:

a. String Creation

```
String str1 = "Hello";              // Using string literal
String str2 = new String("World");    // Using new keyword
```

b. Common String Methods

Method	Description	Example
length()	Returns the length of the string.	str.length();
charAt(index)	Returns the character at the specified index.	str.charAt(0); // 'H'
substring(start, end)	Returns a substring from start index to end-1 index.	str.substring(0, 2); // "He"
toUpperCase()	Converts the string to uppercase.	str.toUpperCase(); // "HELLO"
toLowerCase()	Converts the string to lowercase.	str.toLowerCase(); // "hello"
concat(str)	Concatenates two strings.	str1.concat(str2); // "HelloWorld"
equals(str)	Checks if two strings are equal (case-sensitive).	str1.equals("Hello"); // true
equalsIgnoreCase(str)	Checks if two strings are equal, ignoring case.	str1.equalsIgnoreCase("hello"); // true
replace(oldChar, newChar)	Replaces all occurrences of a character.	str.replace('H', 'Y'); // "Yello"
trim()	Removes leading and trailing whitespaces.	" Hello ".trim(); // “Hello"

Example:

```
public class Main {
    public static void main(String[] args) {
        String str = "Hello, World!";
        System.out.println("Length: " + str.length());
        System.out.println("Character at index 1: " + str.charAt(1));
        System.out.println("Substring (0,5): " + str.substring(0, 5));
        System.out.println("Uppercase: " + str.toUpperCase());
    }
}
```

StringBuilder and StringBuffer

StringBuilder and StringBuffer are mutable versions of the String class. They allow modifications of the content.

StringBuffer is synchronized (thread-safe), while StringBuilder is not.

Example:

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(" World");
System.out.println(sb); // Output: Hello World
```

2. Packages in Java

A package is a namespace that organizes a set of related classes and interfaces. Packages prevent name conflicts and can contain multiple classes, interfaces, and sub-packages.

Creating and Using Packages

To declare a class in a package, use the package keyword at the top of the file.

Example of Creating a Package:

```
package mypackage;

public class MyClass {
    public void display() {
        System.out.println("Hello from MyClass in mypackage.");
    }
}
```

Importing Packages

You can import a package using the import keyword. If you want to import a specific class, use:

```
import mypackage.MyClass;
```

To import all classes from a package:

```
import mypackage.*;
```

Example:

```
import mypackage.MyClass;

public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj.display();
    }
}
```

Classpath

Classpath is the path where the Java runtime system looks for class files. You can set the classpath using the CLASSPATH environment variable or by using the -classpath or -cp option when running a program.

Example of setting classpath:

```
java -cp /path/to/package mypackage.MyClass
```

3. Interfaces in Java

An interface in Java is a blueprint of a class. It has abstract methods (without implementations) and constants. Interfaces are used to achieve abstraction and multiple inheritance in Java.

Differences Between Classes and Interfaces

Aspect	Class	Interface
Implementation	Contains method implementations.	Only method declarations (until Java 8)
Inheritance	A class can inherit from one superclass.	A class can implement multiple interfaces.
Keywords	Declared using class keyword.	Declared using interface keyword.
Variables	Can have instance variables.	Can only have public, static, final variables.
Access Specifier	Can be private, protected, public.	All methods are public and abstract by default.

Implementing an Interface

A class implements an interface using the implements keyword. The class must provide concrete implementations for all the abstract methods in the interface.

Example:

```
interface Animal {
    void sound();  // abstract method
}


class Dog implements Animal {
    public void sound() {
        System.out.println("Dog barks.");
    }
}


public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.sound();
    }
}
```

Applying Interface

Multiple interfaces can be implemented by a class, allowing Java to achieve multiple inheritance in a way that avoids the problems faced by multiple class inheritance.

Example of Multiple Interfaces:

```
interface Animal {
    void sound();
}

interface Movable {
    void move();
}

class Dog implements Animal, Movable {
    public void sound() {
        System.out.println("Dog barks.");
    }

    public void move() {
        System.out.println("Dog runs.");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.sound();
        dog.move();
    }
}
```

Default and Static Methods in Interfaces (Since Java 8)

Default Methods: Interfaces can have default implementations of methods.

Static Methods: Static methods can be called on the interface itself.

Example:

```
interface Animal {
    default void sleep() {
        System.out.println("Animal is sleeping.");
    }

    static void eat() {
        System.out.println("Animal is eating.");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal.eat();  // Calling static method
    }
}
```

4. Enumerations in Java

Enumerations (enums) are a special class type in Java that represents a group of constants. Enums provide type safety and are useful when working with predefined sets of values.

Declaring an Enum

Example:

```
enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
```

```
public class Main {
    public static void main(String[] args) {
        Day day = Day.MONDAY;
        System.out.println("Day: " + day);
    }
}
```

Enum Methods

Enums come with several built-in methods:

values(): Returns an array of all enum constants.

ordinal(): Returns the position of the enum constant.

valueOf(): Converts a string to the corresponding enum constant.

Example:

```
public class Main {
    public static void main(String[] args) {
        Day[] days = Day.values();
        for (Day day : days) {
            System.out.println(day + " at position " + day.ordinal());
        }
    }
}
```

Using Enums in Switch Statements

Enums can also be used in switch statements for better readability.

Example:

```
public class Main {
    enum Day {
        MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
    }

    public static void main(String[] args) {
        Day day = Day.FRIDAY;

        switch (day) {
            case MONDAY:
                System.out.println("Start of the work week!");
                break;
            case FRIDAY:
                System.out.println("Weekend is almost here!");
                break;
            case SUNDAY:
                System.out.println("Rest day.");
                break;
            default:
                System.out.println("Middle of the week.");
                break;
        }
    }
}
```

Summary:

Strings: Immutable objects with methods for string manipulation.

Packages: Used for organizing classes and preventing name conflicts. Can be imported using the import keyword.

Classpath: The path that defines where Java looks for classes during execution.

Interfaces: Abstract contracts that define methods but do not provide implementations, allowing for multiple inheritance.

Enumerations: Used to define a collection of constants, often used for representing fixed sets of values.