COMPUTER SCIENCE

B. Sc. II (CBCS) Semester-IV 2023-2024

2CS2 : RDBMS and Core Java

Unit-VI : Strings & Packages

PROF. V. V. AGARKAR

Assistant Professor & Head

Department of Computer Science

Shri. D. M. Burungale Science & Arts College, Shegaon, Dist. Buldana

Unit–VI

String & Packages: String: String operation, String comparison, Searching and modifying string, StringBuffer, Wrapper classes, **Packages:** Package concept, Defining Package, organizing classes and interfaces in packages, making jar files for library packages, Java In-built Package.

String

Strings are used for storing text. In Java, a String is a fundamental data type used to represent and manipulate text. `String` is a class defined in the Java standard library (`java.lang.String`) and represents text rather than a numeric value. In Java, a string is a sequence of characters. For example, "hello" is a string containing a sequence of characters 'h', 'e', 'l', 'l', and 'o'. Double quotes are used to represent a string in Java.

In Java, basically string is an object that represents a sequence of characters. When a string is created in Java, it actually creates an object of type `String`. Strings in Java are ***immutable***, meaning once a string object is created, its state cannot be changed. In Java, strings are implemented using two classes either `String` or `StringBuffer`. `String` is an immutable class and `StringBuffer` is a mutable class.

A `String` in Java has several key characteristics:

- **Immutability**: Once a `String` object is created, its value cannot be changed. If you modify a String, a new `String` object is created in memory.
- **Stored as Object**: String in Java is not a primitive data type like `int` or `double`, and it's a class, and each `String` is an instance of this class.
- **Literal Creation**: A `String` can be created simply by assigning a string literal, like `String s = "Hello";`
- **Unicode Support**: Java `String` supports Unicode, making it capable of representing a wide range of characters and symbols.

String Operations

The `java.lang.String` class provides many useful methods to perform operations on strings. Here's are some common operations:

1) Creation of String object

There are two ways to create String object in Java:

1. By `String` literal
2. By `new` keyword

1) String Literal

Java String literal is created by using double quotes. For Example:

```
String s = "welcome";
```

2) By `new` keyword

```
String s = new String("Welcome");
```

- **String Constructors**

The `String` class supports several constructors.

a) To create an empty `String`, the default constructor is called as follows:

```
String s = new String();
```

will create an instance of `String` with no characters in it.

b) To create a `String` initialized by an array of characters, use the constructor shown here:

```
String(char chars[])
```

Here is an example:

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
```

This constructor initializes `s` with the string "`abc`".

2) `length` method

The `length()` method can be used to find the length of a string. This method returns the number of characters in the string. For example:

```
String str = "Hello, World!";
int len = str.length();
```

In this example, `str.length()` returns 13 because there are 13 characters in the string "Hello, World!".

3) `valueOf()` method

This method is used to convert different types of values into their corresponding string representation. That is, this method takes different types of parameters (such as `int`, `double`, `char`, etc.) and converts them into their string representation.

```
String.valueOf(parameter)
```

For example:

```
int num = 42;
String str = String.valueOf(num);
```

In the above code, the `String.valueOf()` converts `42` into the string "`42`".

4) `charAt()` method

The Java String `charAt()` method is used to retrieve the character at a specified index within a string. The index is zero-based, i.e. the first character is at index 0, the second character is at index 1, and so on. It has this general form:

```
char charAt(int index)
```

Here, *index* is the index of the character to be retrieved. The `charAt()` returns the character at the specified index.

For example,

```
String str = "Hello, World!";
char ch1 = str.charAt(0);      // Retrieves the first character ('H')
char ch2 = str.charAt(5);  // Retrieves the sixth character (',')
```

In this example, `str.charAt(0)` returns the first character of the string "`Hello, World!`", which is '`H`', and `str.charAt(5)` returns the sixth character, which is '`,`'.

• String Comparison

The String class includes several methods that compare strings or substrings within strings. Some of them are given below:

1) `equals()`

The Java string `equals()` method is used to compare two strings for equality. It returns *true* if the two strings have the same sequence of characters, and *false* otherwise. The comparison is ***case-sensitive***. It has this general form:

```
str1.equals(str2)
```

Here, `str1` and `str2` both are the strings or string objects which are to be compared.

Example :

```
String s1 = "Hello";
String s2 = "Hello";
String s3 = "hello";
      s1.equals(s2)
      s1.equals(s3)
```

In this example, `s1.equals(s2)` returns `true` because both strings contain the same sequence of characters ("`Hello`"). However, `s1.equals(s3)` returns `false` because the characters are not the same due to case difference ("`Hello`" vs "`hello`").

2) `equalsIgnoreCase()`

The java `equalsIgnoreCase()` method is similar to the `equals()` method, but it performs a ***case-insensitive*** comparison between two strings. It checks if two strings are equal, disregarding differences in case. It has this general form:

```
str1.equalsIgnoreCase(str2)
```

Here, `str1` and `str2` both are the strings or string objects which are to be compared.

Example :

```
String s1 = "Hello";
String s2 = "Hello";
String s3 = "hello";
s1.equalsIgnoreCase(s2)
s1.equalsIgnoreCase(s3)
```

In this example, `s1.equals(s2)` returns `true` because both strings contain the same sequence of characters ("`Hello`"). Also, `s1.equals(s3)` returns `true` because both strings contain the same sequence of characters ("`Hello`" and "`hello`") ignoring case.

3) `startsWith()`

Java String `startsWith()` method is used to check whether a string starts with a specific prefix. This method returns ***true*** if the string begins with the specified prefix; otherwise, it returns ***false***. It has two variants and has the following general forms:

```
boolean startsWith(String str)
```

```
boolean startsWith(String str, int fromindex)
```

Here, ***str*** is the String being tested and ***fromindex*** specifies the index into the invoking string at which point the search will begin.

For example:

```
String str= "Hello World";
System.out.println("Starts with: " + str.startsWith("Hello"));
System.out.println("Starts with: " + str.startsWith("Wo"));
System.out.println("Starts with: " + str.startsWith("World", 6));
```

In this example, the first returns ***true*** for "Hello", the second returns ***false*** for "Wo" and the third returns ***true*** for "World" searches from sixth character position.

4) `endsWith()`

Java String `endsWith()` method is used to check whether a string ends with a specific suffix. This method returns ***true*** if the string ends with the specified suffix; otherwise, it returns ***false***. It has the following general form:

```
boolean endsWith(String str)
```

Here, *str* is the String being tested.

For example,

```
String str= "Hello World";
System.out.println("Ends with: " + str.endsWith("World"));
System.out.println("Ends with: " + str.endsWith("Hello"));
```

In this example, the `endsWith()` method is used to check if the string `str` ends with the suffixes "World" and "Hello". It returns ***true*** for "World" and ***false*** for "Hello".

5) `compareTo()`

The Java String `compareTo()` method is used to compare two strings lexicographically (alphabetical order). The comparison is based on the Unicode value of each character in the strings and it is ***case-sensitive***. It returns an integer value that indicates whether the string being compared is less than, equal to, or greater than the other string.

The method returns:

- 0 if the two strings are equal.
- A **negative** value if the invoking string is lexicographically less than the string being compared.
- A **positive** value if the invoking string is lexicographically greater than the string being compared.

It has this general form:

```
int compareTo(String str)
```

Here, *str* is the String being compared with the invoking String.

For example,

```
String str1 = "hello";
String str2 = "world";
String str3 = "hello";
System.out.println(str1.compareTo(str2));
System.out.println(str2.compareTo(str1));
System.out.println(str1.compareTo(str3));
```

In this example:

- **str1.compareTo(str2)** returns a negative value because "hello" comes before "world" lexicographically.
- **str2.compareTo(str1)** returns a positive value because "world" comes after "hello" lexicographically.
- **str1.compareTo(str3)** returns 0 because both strings are equal.

6) `compareToIgnoreCase()`

The Java String `compareToIgnoreCase()` method is similar to `compareTo()`, but it performs a ***case-insensitive*** lexicographical comparison of two strings. It's useful when you want to compare strings without considering their case differences.

```
int compareToIgnoreCase(String str)
```

Here, *str* is the String being compared with the invoking String.

The method returns:

- 0 if the two strings are equal (ignoring case).
- A **negative** value if the invoking string is lexicographically less than the string being compared.
- A **positive** value if the invoking string is lexicographically greater than the string being compared.

For example,

```
String str1 = "hello";
String str2 = "world";
String str3 = "HELLO";
System.out.println(str1.compareToIgnoreCase(str2));
System.out.println(str2.compareToIgnoreCase(str1));
System.out.println(str1.compareToIgnoreCase(str3));
```

In this example:

- **str1.compareToIgnoreCase(str2)** returns a negative value because "hello" comes before "world" lexicographically.

- **str2.compareToIgnoreCase(str1)** returns a positive value because "world" comes after "hello" lexicographically.
- **str1.compareToIgnoreCase(str3)** returns 0 because both strings are equal, ignoring case.

• Searching Strings

The String class provides two methods that allow you to search a string for a specified character or substring: `indexOf()` and `lastIndexOf()`.

1) `indexOf()` method

The `indexOf()` method is used to find the index of the first occurrence of a specified substring within a string. It searches the string from the beginning and returns the index of the first occurrence of the specified substring, or –1 if the substring is not found. The general syntax is:

```
indexOf(String substring)
```

Additionally, there's another version of `indexOf()` that takes an additional parameter specifying the starting index from where the search should begin

```
indexOf(String substring, int startindex)
```

Example:

```
String str = "Corporation floor";
str.indexOf("r");
str.indexOf("r", 11);
str.indexOf("or");
str.indexOf("or", 10);
```

In the above example:

- `str.indexOf("r")` returns 2 because the first occurrence of 'r' is at index 2.
- `str.indexOf("r",11)` returns 16 because the first occurrence of 'r' after index 11 is at index 16.
- `str.indexOf("or")` returns 1 because the first occurrence of substring "or" starts at index 1.
- `str.indexOf("or",10)` returns 15 because the first occurrence of substring "or" after index 10 is at index 15.

2) The `lastIndexOf()` method

The `lastIndexOf()` method returns the position of the last occurrence of a given character or substring in a String, and if the given substring is not found it returns –1. The index counter starts from zero. This method has four variants are as follows:

Method	Description
`lastIndexOf(char ch)`	returns index position for the last occurrence of given char value.

`lastIndexOf(char ch, int ToIndex)`	Returns index position for the last occurrence of given char value and search performs within **0th** index to ***ToIndex*** only.
`lastIndexOf(String substring)`	returns index position for the last occurrence of given substring.
`lastIndexOf(String substring, int TOIndex)`	returns last index position of the first character for the given substring and search performs within **0th** index to ***ToIndex*** only.

Examples:

```
String str = "Corporation floor";
str.lastIndexOf('r');
str.lastIndexOf('r', 11);
str.lastIndexOf("or");
str.lastIndexOf("or", 10);
```

In the above example:

- `str.lastIndexOf("r")` returns 16 because the last occurrence of 'r' is at index 16.
- `str. lastIndexOf ("r",11)` returns 5 because the last occurrence of 'r' before index 11 is at index 5.
- `str. lastIndexOf ("or")` returns 15 because the last occurrence of substring "or" starts at index 15.
- `str. lastIndexOf ("or",10)` returns 4 because the last occurrence of substring "or" before the index 10 is at index 4.

• Modifying a String

Because String objects are immutable, whenever you want to modify a String, you must either copy it into a `StringBuffer` or `StringBuilder`, or use one of the following String methods, which will construct a new copy of the string with your modifications complete.

1) **`substring()`**

The `substring()` method in Java is used to extract a part of a string, starting from a specified index to the end of the string or up to a specified ending index. The syntax is as follows:

```
String substring(int startIndex)
String substring(int startIndex, int endIndex)
```

Here,

- **startIndex** is the index from which the substring starts. It is ***inclusive***.
- **endIndex** (Optional) is the index before which the substring ends. It is ***exclusive***. If not specified, the substring extends to the end of the original string.

Example:

```
String s =  "Computer Science";
```

```
System.out.println(s.substring(9));
System.out.println(s.substring(3,6));
```

The first prints `Science` and second prints `put`.

2) **`concat()`**

The Java String `concat()` method is used to concatenate one string to the end of another. It returns a new string that represents the concatenation of the original string with the specified string. The general form is:

```
String concat(String str)
```

`str` is the string to be concatenated to the end of the original string.

For example,

```
String s1 = "Ja";
String s2 = "va";
String s3 = s1.concat(s2);
```

The string `s3` contains the string "Java".

3) **`replace()`**

The `replace()` method in Java is used to replace all occurrences of a specified character or substring within a string with another character or substring. It returns a new string with the replacements made. The syntax is as follows:

```
String replace(char oldChar, char newChar)
String replace(CharSequence target, CharSequence replacement)
```

Here,

- `oldChar`: The character to be replaced.
- `newChar`: The character to replace all occurrences of oldChar.
- `target`: The substring to be replaced.
- `replacement`: The substring that replaces all occurrences of target.

For Example :

```
String str = "Hello World";
String s1 = str.replace('o', 'x');
String s2 = str.replace("World", "Java");
```

In this example, String `s1` contains `"Hellx Wxrld"` after replacing all 'o' by 'x' and String `s2` contains `"Hello Java"` after replacing 'World' by 'Java';

4) **`trim()`**

The `trim()` method in Java is used to remove leading and trailing whitespace (spaces, tabs, and newlines) from a string. It returns a new string with whitespace removed. The syntax is as follows:

```
String trim()
```

Here is an example:

```
String s1 = "          Hello World      ";
String s2 = s1.trim();
```

In this example, trim() method is used to remove leading and trailing whitespace from the string **s1**, resulting in a trimmed string stored in **s2**.

5) **`toLowerCase()`**

The `toLowerCase()` method in Java is used to convert all characters of a string to lowercase. It returns a new string with all characters converted to lowercase. The syntax is as follows:

```
String toLowerCase()
```

Example,

```
String s1 = "COMPUTER";
String s2 = s1.toLowerCase();
```

In this example, the string **s1** contains `"COMPUTER"`, and the `toLowerCase()` method converts all characters of **s1** to lowercase, so string `"computer"` is stored in **s2**.

6) **`toUpperCase()`**

The `toUpperCase()` method in Java is used to convert all characters of a string to uppercase. It returns a new string with all characters converted to uppercase. The syntax is as follows:

```
String toUpperCase()
```

Example,

```
String s1 = "computer";
String s2 = str.toUpperCase();
```

In this example, the string **s1** contains `"computer"`, and the `toUpperCase()` method converts all characters of **s1** to uppercase, so string `"COMPUTER"` is stored in **s2**.

StringBuffer

A `StringBuffer` class is a class in the `java.lang` package and it is used to create mutable (modifiable) strings. The `StringBuffer` class in Java is the same as the `String` class except it is mutable i.e. it can be changed. The `StringBuffer` class allows to modify the contents of a string without creating a new object every time. The string represents fixed-length, immutable character sequences while `StringBuffer` represents growable and writable character sequences. Java `StringBuffer` class is thread-safe i.e. multiple threads cannot access it simultaneously.

Key Features of `StringBuffer`

- **Mutable**: `StringBuffer` provides the flexibility to change the content, making it ideal for scenarios where you have to modify strings frequently.
- **Synchronized**: Being thread-safe, it ensures that only one thread can access the buffer's methods at a time, making it suitable for multi-threaded environments.

- **Performance Efficient**: For repeated string manipulation, using `StringBuffer` can be more efficient than the `String` class.
- **Method Availability**: `StringBuffer` offers several methods to manipulate strings. These include `append()`, `insert()`, `delete()`, `reverse()`, and `replace()`.

Following are some `StringBuffer` methods:

1) `append()`

The `append()` method of Java `StringBuffer` class is used to append the specified string to the end of the `StringBuffer`. The general form is:

```
StringBuffer append(String str)
```

str is the string to be appended to the end of the StringBuffer.

For Example :

```
StringBuffer sb = new StringBuffer("Hello");
sb.append(" World");
System.out.println(sb);
```

It prints "Hello World" i.e. append " World" to the "Hello".

Note that, the `append()` method has several overloaded versions (around 13 variants) to accept different types of data.

2) `insert()`

The `insert()` method of Java in the `StringBuffer` class is used to insert characters or sequences of characters into a `StringBuffer` object at a specified position. The general form is:

```
StringBuffer insert(int position, String str)
```

For Example :

```
StringBuffer sb = new StringBuffer("Hello");
sb.insert(5, " World");
System.out.println(sb);
```

It prints "Hello World" i.e. inserted " World" to the "Hello" string from character position number 5.

Note that, the `insert()` method has several overloaded versions to accept different types of data.

3) `delete()`

The `delete()` method of Java in `StringBuffer` class is used to remove or delete characters from the `StringBuffer` object, starting from a specified index up to a specified ending index. The method returns the string after deleting the characters given by the range mentioned in the parameters. The general syntax is as follows:

```
StringBuffer delete(int startIndex, int endIndex)
```

Here,

- **startIndex** is the index from which the deleting starts. It is ***inclusive***.
- **endIndex** is the index before which the deleting ends. It is ***exclusive***.

Example:

```
StringBuffer sb = new StringBuffer("Hello World");
sb.delete(5, 10);
System.out.println(sb);
```

It prints "Hellod" i.e. after deleting characters " Worl" from position number 5 to 9.

4) `reverse()`

The `reverse()` method of Java in the `StringBuffer` class is used to reverse the characters in the `StringBuffer` object. It does not return anything; instead, it modifies the original `StringBuffer`. This method does not accept any parameters. The general form is:

```
StringBuffer reverse()
```

For Example :

```
StringBuffer sb = new StringBuffer("Hello World");
sb.reverse();
System.out.println(sb);
```

In this example, the `reverse()` method is called on the `StringBuffer` object `sb`, which contains the string "`Hello World`". After calling `reverse()`, the characters in `sb` are reversed, resulting in the string "**`dlroW olleH`**".

Wrapper classes

Java Wrapper classes provide a way to use primitive data types as objects. In Java, primitive data types like `int`, `double`, `float`, etc., are not objects and do not have methods associated with them. Wrapper classes are used to convert primitive data types into objects so that they can be used in Java where objects are required.

Each primitive data type in Java has a corresponding wrapper class. Here are the wrapper classes for some common primitive data types:

Primitive Data Type	Wrapper Class
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean

[**Table :** Primitive data types and their corresponding Wrapper class]

- ***Need and Advantages of Wrapper classes***

Wrapper classes in Java serve several important purposes:

1. **Conversion between primitive types and objects:** Wrapper classes provide a way to convert primitive data types into objects and vice versa. This is particularly useful when working with collections or generics, which only accept objects.
2. **Nullable values:** Wrapper classes allow for null values to be assigned to variables, whereas primitive types cannot hold null values. This is helpful where you need to represent the absence of a value, such as with database queries or user inputs.
3. **Generics:** Wrapper classes are often used with generics because generics in Java cannot directly use primitive types. For example, you cannot create a '`List<int>`'; instead, you would use '`List<Integer>`' where '`Integer`' is the wrapper class for '`int`'.
4. **Standardized behavior and methods:** Wrapper classes provide standardized behavior and methods for working with primitive data types. For example, '`Integer`' provides methods like '`parseInt()`' for converting strings to integers, and '`toString()`' for converting integers to strings.
5. **Collections and algorithms:** Wrapper classes are commonly used in Java collections and algorithms because collections can only hold objects, not primitive types. For example, '`ArrayList<Integer>`' is a common way to store a list of integers in Java.

Java provides automatic conversion between primitive data types and their corresponding wrapper classes, without explicitly calling constructors or methods. The automatic conversion of primitive data type into its corresponding wrapper class is known as ***autoboxing***, for example, `byte` to `Byte`, `char` to `Character`, `int` to `Integer`, etc. The automatic conversion of wrapper type into its corresponding primitive type is known as ***unboxing***, it is the reverse process of autoboxing.

Packages

In Java, packages are a mechanism to organize and manage classes and interfaces. A package is a collection of related classes and interfaces. Packages provide a means to reuse classes defined in one application into another application. A package serves two purposes. First, it provides a mechanism by which related pieces (classes and interfaces) of a program can be organized as a unit. Classes defined within a package must be accessed through their package name. Thus, a package provides a way to name a collection of classes. Second, a package participates in Java's access control mechanism. Classes defined within a package can be made private to that package and not accessible by code outside the package. Thus, the package provides a means by which classes can be encapsulated.

Packages are organized in a hierarchical structure, like the directory of the file system. The periods (dots) are used to separate levels in the package hierarchy. Each package in Java has its unique name and organizes its classes and interfaces into a separate namespace. Following is an example to show hierarchical structure of packages:

```
com
└── example
    └── myproject
        └── MyClass.java
```

Defining a Package

To define a package in Java, a package declaration should be included at the beginning of the Java source file, i.e. simply includes a `package` command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The `package` statement defines a name space in which classes are stored. If you omit the `package` statement, the class names are put into the default package, which has no name. This is the general form of the `package` statement:

```
package pkg;
```

Here, *pkg* is the name of the package.

For example, the following statement creates a package called **MyPackage**.

```
package MyPackage;
```

Java uses file system directories to store packages. For example, the `.class` files for any classes you declare to be part of `MyPackage` must be stored in a directory called `MyPackage`. Remember that the directory name must match the package name exactly.

You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here:

```
package pkg1[.pkg2[.pkg3]];
```

A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as

```
package com.example.MyProject;
```

needs to be stored in `com\example\MyProject` in a Windows environment.

Organizing classes and interfaces in packages

Organizing classes and interfaces in packages helps in maintaining a well-structured and modular codebase. Proper organization promotes code readability, maintainability, and reusability. Following is a typical approach to organize classes and interfaces in packages:

1. **Choose meaningful package names:** Package names should reflect the functionality or purpose of the classes/interfaces contained within them. This helps prevent naming conflicts and makes it easier for developers to understand the purpose of each package. Use lowercase letters for naming packages.
2. **Group related classes/interfaces together:** Place classes and interfaces that are closely related to each other within the same package.
3. **Avoid deep nesting:** Try to avoid excessive levels of nesting within packages. Deeply nested packages can make the codebase harder to navigate and understand.
4. **Use sub-packages carefully:** Sub-packages can be used to further organize classes within a package, but don't overuse them. Use sub-packages only when it significantly improves the organization and clarity of your codebase.
5. **Follow Java naming conventions:** Adhere to Java naming conventions when naming packages, classes, and interfaces. Classes and interfaces should use *camelCase* with the first letter capitalized, while package names should be all lowercase.

6. **Document package-level comments:** Provide package-level documentation comments to describe the purpose and usage of each package. This helps other developers understand the contents of the package without looking into the details.
7. **Keep packages small and focused:** Aim for small, focused packages with a clear purpose. If a package becomes too large, consider refactoring it into smaller, more specialized packages.

Making *jar* files for library packages

In Java, JAR stands for Java ARchive. It is a package file format typically used to combine many Java class files and associated metadata and resources (text, images, etc.) into one file to distribute application software or libraries on the Java platform. In simple words, a JAR file is a file that contains a compressed version of `.class` files, audio files, image files, or directories. We can imagine a `.jar` file as a zipped file `(.zip)`.

A step-by-step procedure to create JAR files for library packages is as follows:

1. Compile Source Code:

- Compile the Java source code files (`.java` files) into bytecode (`.class` files).
- Ensure that you compile all source files that belong to the library package.

2. Create a Manifest File (Optional):

- If your JAR file requires a manifest file (for specifying entry points, main class, etc.), create a text file named **`MANIFEST.MF`**.
- Add necessary entries to the manifest file.

3. Organize Files:

- Organize the compiled class files, resources (if any), and manifest file (if required) into a directory structure that matches the package hierarchy.
- Ensure that the directory structure mirrors the package hierarchy.

4. Create the JAR File:

- Use the ***jar*** command-line tool (provided by the JDK) to create the JAR file.
- The syntax is:

```
jar cf jarfilename inputfiles
```

The options and arguments used in this command are:

- The **c** option indicates to create a JAR file.
- The **f** option indicates that the output go to a file rather than to stdout.
- **jarfilename** is the name of the resulting JAR file. By convention, JAR filenames are given a `.jar` extension, though this is not required.
- The **input-files** is a space-separated list of one or more files that you want to include in your JAR file.

- The command will generate a JAR file and place it in the current directory.

Once the JAR file is created and verified, you can distribute it as a library for others to use. Users can include the JAR file in their Java project's classpaths to use the classes and resources provided by the library.

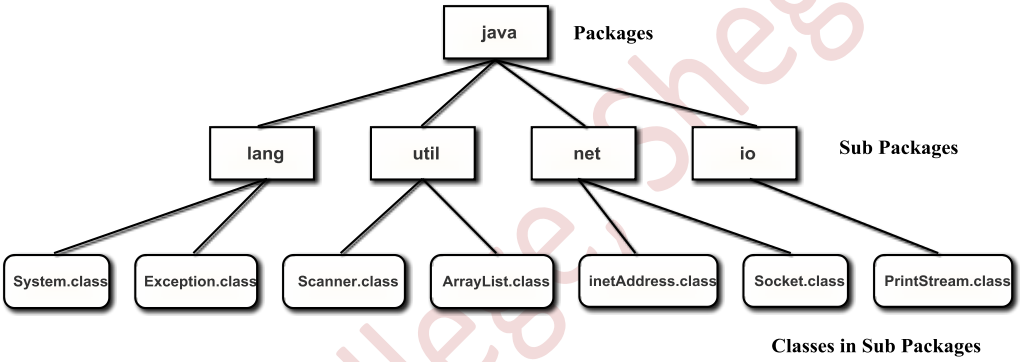
Types of packages in Java

There are two types of packages in java: *Built-in* and *User-Defined packages*.

1) **Built-in package**

The prewritten package in Java is known as *built-in packages*. These built-in packages are those which are supplied as a part of JDK and are automatically available for use in Java programs. Java comes with several built-in packages, which provide a wide range of functionalities to developers. These packages are part of the Java Development Kit (JDK) and are automatically available for use in Java programs.

The Java API is a library of prewritten classes that are free to use, included in the Java Development Environment. The library contains components for managing input, database programming, and much more. The library is divided into **packages** and **classes**. Meaning you can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package. To use a class or a package from the library, you need to use the `import` keyword. There are many predefined packages in java some of them are as follows:



`java.lang`	This package contains language support classes (e.g. classes which defines primitive data types, math operations). This package is automatically imported for each and every java program.
`java.io`	This package contains classes for supporting input / output operations.
`java.util`	This package contains utility classes such as Scanner class, Date Class, class for implement data structure and Collection Framework API etc.
`java.net`	This package contains all the classes for supporting networking operations. This is used for developing client server applications.
`java.text`	This package is used for formatting date and time on day to day business operations.
`java.sql`	This package is used for retrieving the data from database and performing various operations on database.
`java.awt`	This package is an Abstract Windowing Toolkit. It contains classes for implementing the components for graphical user interfaces (like button , menus etc).
`java.applet`	This package contains classes for creating Applets.

There are many more packages .....

2) **User-defined package:**

In Java, user can create his own packages to organize and structure his code. A package that is created by user is called ***user-defined package***. A user defined package is used to group set of classes, interfaces and sub packages which are commonly used at one place. Once the classes are added to the package, user can use them in other Java files by importing them. The user defined package provided access protection and namespace management.

In Java, to declare user defined package, the `package` statement is used as the first non-comment statement in a Java source file. The `package` statement must appear before any `import` statements or class declarations in the file.

Importing Packages

Java includes the `import` statement to bring certain classes, or entire packages, into visibility. Once imported, a class can be referred directly using only its name.

In a Java source file, `import` statements occur immediately following the **package** statement (if it exists) and before any class definitions. The general form is:

```
import pkg1[.pkg2].(classname|*);
```

Here, *`pkg1`* is the name of a top-level package, and *`pkg2`* is the name of a subordinate package inside the outer package separated by a dot (`.`). There is no practical limit on the depth of a package hierarchy, except that imposed by the file system. Finally, you specify either an explicit *`classname`* or a star (*`*`*), which indicates that the Java compiler should import the entire package.

Following examples show both forms in use:

Example 1:

```
import java.util.Scanner;
```

Here:

`java` is a top level package

`util` is a sub package

and `Scanner` is a class which is present in the sub package `util`.

In example-1 only the `Scanner` class from the `java.util` package is imported.

Example 2:

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

In example-2 a whole package is imported. This example will import ALL the classes in the `java.util` package.

■ ■ ■ ■ ■

Sample Questions

1. Explain the following methods with example:
   (i) String.Substring( ) (ii) String.Compare( )
   (iii) String.Concat( ) **6**
2. What is string? Explain any three string comparison functions. **6**
3. Explain following function with syntax and example :
   (i) substring( ) (ii) concat( ) (iii) trim( ) **6**
4. Explain the following:
   (i) substring( ) (ii) replace( ) (iii) concat( ) **6**
5. State string operations in Java with syntax and write a program in Java to demonstrate the use of string comparison. **6**
6. What is a wrapper class? Explain need and advantages of wrapper class. **6**
7. Describe concept of importing package with example. **6**
8. Explain Java-in-built package with example. **6**
9. What is package? How to define a package? Explain with suitable example. **6**
10. Explain the concept of package and write Java In-Built packages. **6**

■■■■■