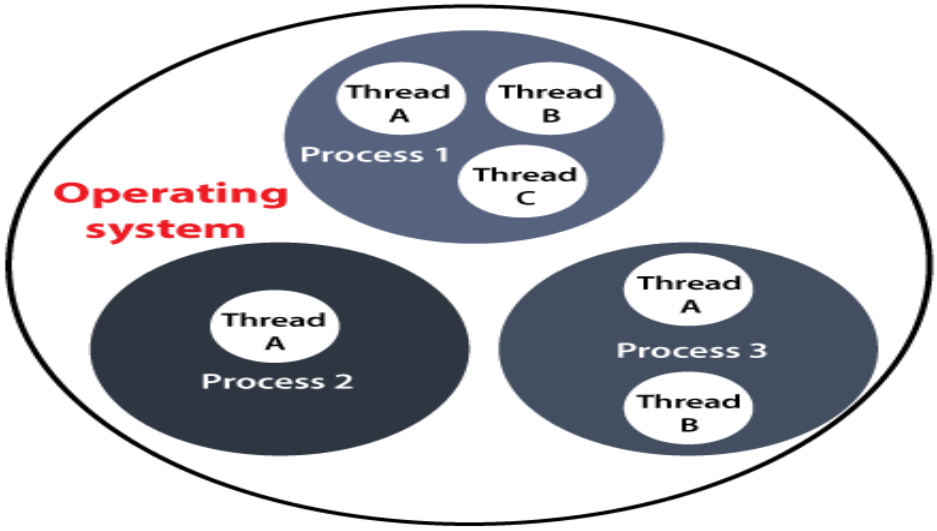
Unit 4 Threads and packages

❖ Thread

Before introducing the thread concept, we were unable to run more than one task in parallel. It was a **drawback**, and to remove that drawback, Thread Concept was introduced. In Java, a thread refers to the **smallest unit of execution** within a process. Threads allow **concurrent execution** of tasks.

A Thread is a very **light-weighted process**, or we can say the smallest part of the process that allows a program to operate more efficiently by running multiple tasks simultaneously.

In order to perform **complicated tasks in the background**, we used the Thread concept in Java. In a program or process, all the threads have their own separate path for execution, so each thread of a process is **independent**.



Another benefit of using **thread** is that if a thread gets an exception or an error at the time of its execution, it doesn't affect the execution of the other threads. All the threads share a common memory and have their own stack, local variables and program counter. When multiple threads are executed in parallel at the same time, this process is known as **Multithreading**.

The Concept Of Multitasking

To help users Operating System accommodates users the privilege of multitasking, where users can perform multiple actions simultaneously on the machine. This Multitasking can be enabled in two ways:

1. **Process-Based Multitasking**
2. **Thread-Based Multitasking**

1. Process-Based Multitasking (Multiprocessing)

- Each process has an **address in memory**. In other words, each process allocates a **separate memory** area.
- A process is **heavyweight.**
- Cost of communication between the process is **high**.
- Switching from one process to another requires **some time for saving and loading registers, memory maps, updating lists, etc.**

2. Thread-Based Multitasking

As we discussed above Threads are provided with lightweight nature and share the same address space, and the cost of communication between threads is also low.

Note: At least one process is required for each thread

Advantages of Java Multithreading (Example in NB)

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.

2) You can **perform** many **operations together,** so it **saves time**.

3) Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

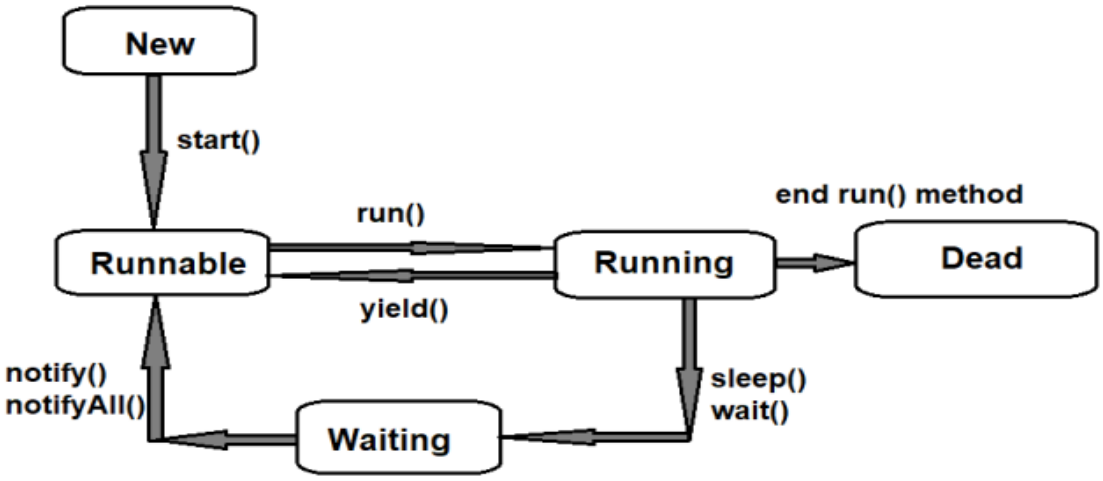
❖ Life cycle of a Thread (Thread States)

In Java, a thread always exists in any one of the following states. These states are:

1. New (Born)

Active (Running $ Runnable)

2. Runnable (Ready)
3. Running (Execution)
4. Blocked / Waiting
5. Terminated (Dead)



Life Cycle of Thread in Java

New: Whenever a new thread is created, it is always in the new state. For a thread in the new state, the code has not been run yet and thus its execution has not begun.

Active: When a thread invokes the start() method, it moves from the new state to the active state. The active state contains two states within it: one is **runnable**, and the other is **running**.

- **Runnable:** A thread, that is ready to run is then moved to the runnable state. In the runnable state, the thread may be running or may be ready to run at any given instant of time. It is the duty of the thread scheduler to provide the thread time to run.
  A program implementing multithreading acquires a fixed slice of time to each individual thread. Each and every thread runs for a short span of time and when that allocated time slice is over, the thread voluntarily gives up the CPU to the other thread, so that the other threads can also run for their slice of time.
  Whenever such a scenario occurs, all those threads that are willing to run, waiting for their turn to run lie in the runnable state. In the runnable state, there is a queue where the threads lie.
- **Running:** When the thread gets the CPU, it moves from the runnable to the running state. Generally, the most common change in the state of a thread is from runnable to running and again back to runnable.

Blocked or Waiting: Whenever a thread is inactive for a span of time (not permanently) then, either the thread is in the blocked state or is in the waiting state.

For example, a thread (let's say its name is A) may want to print some data from the printer. However, at the same time, the other thread (let's say its name is B) is using the

printer to print some data. Therefore, thread A has to wait for thread B to use the printer. Thus, thread A is in the blocked state. A thread in the blocked state is unable to perform any execution and thus never consume any cycle of the Central Processing Unit (CPU). Hence, we can say that thread A remains idle until the thread scheduler reactivates thread A, which is in the waiting or blocked state.

Timed Waiting: Sometimes, waiting for leads to starvation. For example, a thread (its name is A) has entered the critical section of a code and is not willing to leave that critical section. In such a scenario, another thread (its name is B) has to wait forever, which leads to starvation. To avoid such scenario, a timed waiting state is given to thread B. Thus, thread lies in the waiting state for a specific span of time, and not forever. A real example of timed waiting is when we invoke the sleep() method on a specific thread. The sleep() method puts the thread in the timed wait state. After the time runs out, the thread wakes up and start its execution from when it has left earlier.

Terminated: A thread reaches the termination state because of the following reasons:

- When a thread has finished its job, then it exits or terminates normally.
- Abnormal termination**:** It occurs when some unusual events such as an unhandled exception or segmentation fault.

A terminated thread means the thread is no more in the system.

How to create a thread in Java ?

There are two ways to create a thread:

1. By extending **Thread class**
2. By implementing **Runnable interface.**

Thread class:

A thread can be created by extending the **java.lang.Thread** class.

Thread class provide constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

A thread is created by extending the Thread class and overriding its **run()** method. The run() method contains the code that will be executed by the thread.

```
public class MyThread extends Thread {
   public void run() {
      // Code to be executed by the thread
   }
}
```

After defining your Thread subclass, you can create an instance of it and start the execution of the thread using the **start()** method. This method looks out for the run() method and begins executing the body of the run() method.

MyThread obj = new **MyThread**();

obj.**start();**

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r,String name)

Commonly used methods of Thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread. JVM calls the run() method on the thread.
3. **public void stop():** is used to stop the thread(depricated).
4. **Public static void sleep(long miliseconds):** Causes the currently executing thread to sleep for the specified number of milliseconds.
5. **public void join():** waits for a thread to **die**.
6. **public void join(long miliseconds):** waits for a thread to **die** for the specified miliseconds.
7. **public int getPriority():** returns the priority of the thread.
8. **public void setPriority(int priority):** changes the priority of the thread.
9. **public String getName():** returns the name of the thread.
10. **public void setName(String name):** changes the name of the thread.
11. **public int getId():** returns the id of the thread.
12. **public boolean isAlive():** tests if the thread is alive.
13. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.

Exam :

```
class Multi extends Thread{
        public void run(){
        System.out.println("thread is running...");
        }
        public static void main(String args[]){
                Multi t1=new Multi();
                t1.start();
        }
 }
```

Output:

```
thread is running...
```

Thread(String Name)

We can directly use the Thread class to spawn new threads using the constructors defined above.

```
public class MyThread1  {
        public static void main(String args[])  {
                Thread t= new Thread("My first thread");
                t.start();
                String str = t.getName();
                System.out.println(str);
        }
 }
```

Output:

```
My first thread
```

Thread(Runnable r, String name)

```
public class MyThread2 implements Runnable  {
        public void run() {
                System.out.println("Now the thread is running ...");
        }
        public static void main(String argvs[])  {
                Runnable r1 = new MyThread2();
                Thread th1 = new Thread(r1, "My new thread");
                th1.start();
                String str = th1.getName();
                System.out.println(str);
        }
}
```

Output:

```
My new thread
Now the thread is running ...
```

Runnable interface:

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface has an undefined method **run()** with void as return type, and it takes in no arguments.

~ public void run(): is used to perform action for a thread.

Steps to create thread using Runnable :

- Create a class that implements the Runnable interface. This class must have a run() method, which is the code that the thread will execute.
- Create an instance of the Runnable class.
- Create an instance of the Thread class, passing in the Runnable instance as the constructor parameter.
- Call the start() method on the Thread instance.

If you are not extending the Thread class, your class object would **not be treated** as a thread object, so you need to **explicitly create the Thread class object**. We are passing the object of your class that implements Runnable so that your class run() method may execute.

Exam :

```
class Multi3 implements Runnable{
        public void run(){
                System.out.println("thread is running...");
        }
        public static void main(String args[]){
                Multi3 m1=new Multi3();
                Thread t1 =new Thread(m1);   // Using the constructor Thread(Runnable r)
                t1.start();
        }
 }
```

Output:

```
thread is running...
```

❖ Thread Priority

Each thread has a priority. Priorities are represented by a number between **1** and **10**. In most cases, the thread scheduler schedules the threads according to their priority. But it is not guaranteed because it depends on JVM specification that which scheduling it chooses. Not only JVM a Java programmer can also assign the priorities of a thread explicitly in a Java program.

Setter & Getter Method of Thread Priority

public final int getPriority():

This method returns the priority of the given thread.

public final void setPriority(int newPriority):

This method updates or assign the priority of the thread to newPriority. The method throws **IllegalArgumentException** if the value newPriority goes out of the range, which is 1 (minimum) to 10 (maximum).

3 constants defined in Thread class:

1. public static int MIN_PRIORITY
2. public static int NORM_PRIORITY
3. public static int MAX_PRIORITY

Default priority of a thread is **5 (NORM_PRIORITY)**. The value of **MIN_PRIORITY** is **1** and the value of **MAX_PRIORITY** is **10**.

```
class MyThread extends Thread {
   public MyThread(String name) {
      super(name);
   }

   public void run() {
      System.out.println("Thread: " + getName() + ", Priority: " + getPriority());
   }
}

public class Main {
   public static void main(String[] args) {
      MyThread thread1 = new MyThread("Thread 1");
      MyThread thread2 = new MyThread("Thread 2");

      // Set priorities
      thread1.setPriority(Thread.MIN_PRIORITY);
      thread2.setPriority(Thread.MAX_PRIORITY);

      // Start threads
      thread1.start();
      thread2.start();
   }
}
```

In this example, **thread1** is set to the minimum priority, and **thread2** is set to the maximum priority. When you run this code, the output might vary depending on the system, but generally, you'll see that **thread2** tends to run before **thread1** because it has a higher priority.

❖ Inter-thread Communication

Inter-thread communication or **Co-operation** is all about allowing synchronized threads to communicate with each other. Inter-thread communication is also known as Cooperation in Java.

Inter-thread communication is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed. It is implemented by following methods of **Object class**:

- wait()
- notify()
- notifyAll()

1) wait() method

The wait() method causes current thread to **release the lock** and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has finished.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

Method	Description
public final void wait() throws InterruptedException	It waits until object is notified.
public final void wait(long timeout) throws InterruptedException	It waits for the specified amount of time.

2) notify() method

This method is used to send the notification to one of the waiting thread, so that thread enters into a running state and execute the remaining task.

This method wakeup a single thread into the active state that acts on the common object.

Syntax: **public final void** notify()

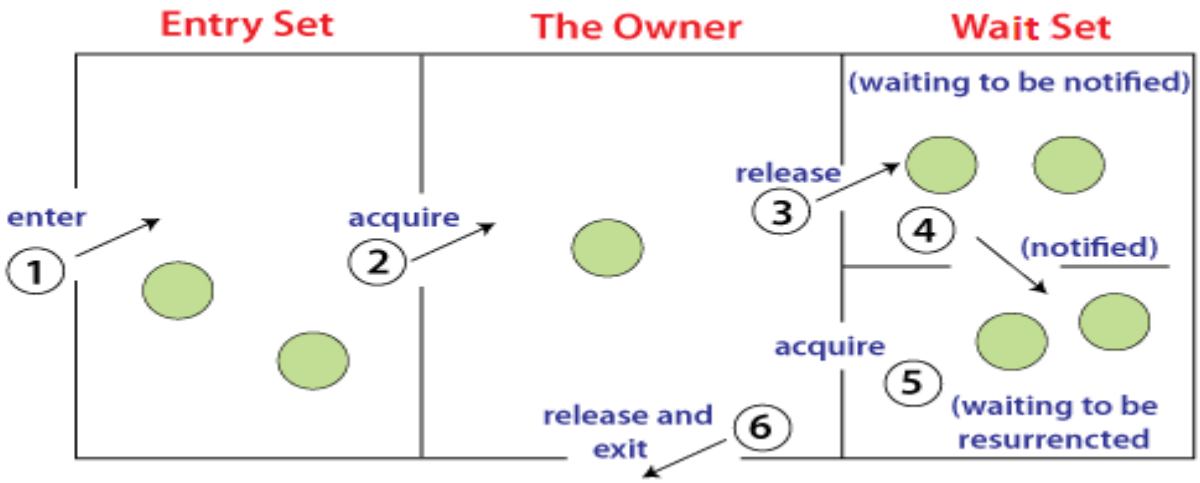
3) notifyAll() method

This method is used to send the notification to all the waiting threads, so that all thread enters into the running state and execute simultaneously.

This method wakes up all the waiting threads that act on the common objects.

Syntax: **public final void** notifyAll()

Understanding the process of inter-thread communication



Example

```
class Customer{
	int amount=10000;

	synchronized void withdraw(int amount){
		System.out.println("going to withdraw...");
		if(this.amount<amount){
			System.out.println("Less balance; waiting for deposit...");
			try{
				wait();
			}
			catch(Exception e){}
		}
		this.amount-=amount;
		System.out.println("withdraw completed...");
	}

	synchronized void deposit(int amount){
		System.out.println("going to deposit...");
		this.amount+=amount;
		System.out.println("deposit completed... ");
		notify();
	}
}
class Test{
	public static void main(String args[]){
		final Customer c=new Customer();
		new Thread(){
			public void run(){
				c.withdraw(15000);
```

```
                }
            }.start();
            new Thread(){
                public void run(){
                    c.deposit(10000);
                }
            }.start();
        }
}
```

Output:

```
going to withdraw...
Less balance; waiting for deposit...
going to deposit...
deposit completed...
withdraw completed
```

❖ Difference between wait and sleep

wait()	sleep()
The wait() method releases the lock.	The sleep() method doesn't release the lock.
It is a method of Object class	It is a method of Thread class
It is the non-static method	It is the static method
It should be notified by notify() or notifyAll() methods	After the specified amount of time, sleep is completed.

❖ Deadlock in Java

Deadlock in Java is a part of **multithreading**. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.

Synchronized keyword is used to make the class or method thread-safe which means only one thread can have lock of synchronized method and use it, other threads have to wait till the lock releases and anyone of them acquire that lock. It is important to use if our program is running in multi-threaded environment where two or more threads execute simultaneously. But sometimes it also causes a problem which is called **Deadlock**.

Figure - 1

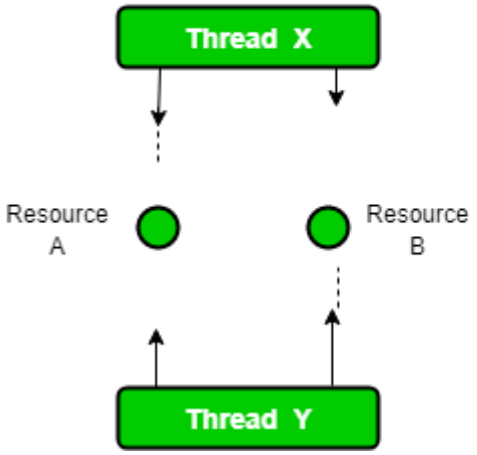
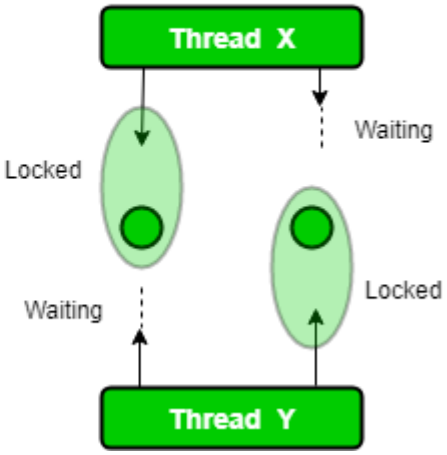


Figure - 2



How to Avoid Deadlock in Java?

Deadlocks cannot be completely resolved. But we can avoid them by following basic rules mentioned below:

1. **Avoid Nested Locks**: We must avoid giving locks to multiple threads, this is the main reason for a deadlock condition. It normally happens when you give locks to multiple threads.
2. **Avoid Unnecessary Locks**: The locks should be given to the important threads. Giving locks to the unnecessary threads that cause the deadlock condition.
3. **Using Thread Join**: A deadlock usually happens when one thread is waiting for the other to finish.

Example :

```
public class TestDeadlockExample1 {
 public static void main(String[] args) {
        final String resource1 = "ratan jaiswal";
        final String resource2 = "vimal jaiswal";
        // t1 tries to lock resource1 then resource2
        Thread t1 = new Thread() {
         public void run() {
                synchronized (resource1) {
                        System.out.println("Thread 1: locked resource 1");
                        try { Thread.sleep(100);} catch (Exception e) {}
                        synchronized (resource2) {
                                System.out.println("Thread 1: locked resource 2");
                        }
                }
         }
        };
  // t2 tries to lock resource2 then resource1
```

```
        Thread t2 = new Thread() {
          public void run() {
            synchronized (resource2) {
                        System.out.println("Thread 2: locked resource 2");

                        try { Thread.sleep(100);} catch (Exception e) {}
                        synchronized (resource1) {
                                System.out.println("Thread 2: locked resource 1");
                        }
            }
          }
          };
          t1.start();
          t2.start();
      } }
```

Output:

```
Thread 1: locked resource 1
Thread 2: locked resource 2
```

What is Package? Explain access specifier with java package also explain types of package.

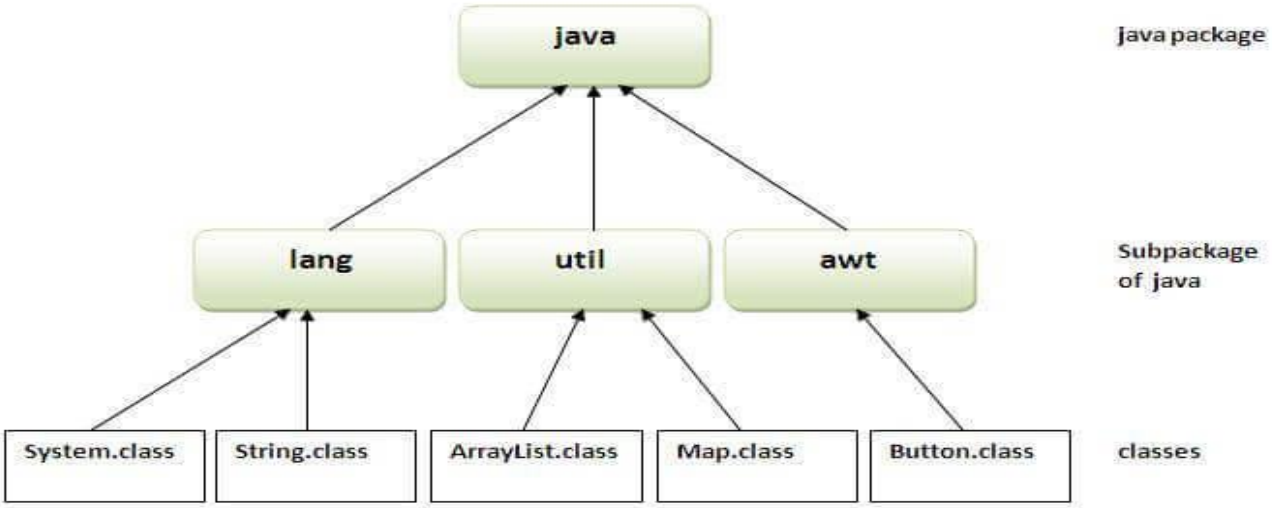
A **java package** is a group of similar types of classes, interfaces and sub-packages. Every class is a part of a certain package. We can reuse existing classes from the packages as many times as we need them in our program. package can be defined using the keyword **package**.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming conflicts.



Package Access

----ACCESS CONTROL IN JAVA----

Packages are divided into two parts:

- **Built-in packages:** In java, we already have various pre-defined packages and these packages contain large numbers of classes and interfaces that we used in java are known as Built-in packages.
- **User-defined packages:** As the name suggests user-defined packages are a package that is defined by the user or programmer.

Built-in packages

Packages that come with JDK or JRD you download are known as built-in packages. The built-in packages have come in the form of JAR files and when we unzip the JAR files we can easily see the packages in JAR files, for example, lang, io, util, SQL, etc. Java provides various built-in packages for example **java.awt**

Examples of Built-in Packages :

- **java.lang**: Contains classes and interfaces that are fundamental to the design of the Java programming language. Classes like String, StringBuffer, System, Math, Integer, etc. are part of this package.
- **java.util**: Contains the collections framework, some internationalization support classes, properties, random number generation classes. Classes like ArrayList, LinkedList, HashMap, Calendar, Date, Time Zone, etc. are part of this package.
- **java.net**: Provides classes for implementing networking applications. Classes like Authenticator, HTTP Cookie, Socket, URL, URLConnection, URLEncoder, URLDecoder, etc. are part of this package.

- **java.io**: Provides classes for system input/output operations. Classes like BufferedReader, BufferedWriter, File, InputStream, OutputStream, PrintStream, Serializable, etc. are part of this package.
- **java.awt**: Contains classes for creating user interfaces and for painting graphics and images. Classes like Button, Color, Event, Font, Graphics, Image, etc. are part of this package.
- **java.sql**: Provides the classes for accessing and processing data stored in a database. Classes like Connection, DriverManager, PreparedStatement, ResultSet, Statement, etc. are part of this package.

User-defined package :

user-defined package means that the package is created by the programmer rather than being part of the core Java language and is a collection of related classes and interfaces that are organized together into a group.

Rules :

❖ **Package Name:**

Names of packages should be written in **lowercase** letters.

Start your package name with your website domain name, but reverse. For instance, if your website is example.com, your package name might start with com.example.

❖ **Package Declaration:**

At the **beginning** of each Java file, include a line that starts with package followed by the name of your package.

This declaration must be the first non-comment statement in the file, if present.

❖ **Directory Structure:**

Arrange your directories to **match** your **package hierarchy**.

Each part of your package name should have its own directory, separated by slashes (/).

❖ **Access Levels:**

By default, classes, interfaces, and members of a package can only be used within the same package.

You can make them accessible outside the package by marking them as **public**.

❖ **Import Package:**

To use classes from your package in other Java files, import the package using the **import statement**.

You can import specific classes or everything in the package using *.

Example :

```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}

//save by B.java
package mypack;
import pack.A;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
Output:Hello
```

Sub package

Creating a subpackage in Java is similar to creating a regular package. You just need to create a directory structure within your existing package directory. Then, you can import classes from the subpackage into another package as you would with any other package.

Create a Subpackage:

Let's assume you have an existing package called com.example, and you want to create a subpackage called mypack within it.

Create a directory named mypack inside the com/example directory.

Your directory structure will look like this:

com

└── example

└── mypack

Write Your Java Classes:

Inside the util directory, create your Java classes.

Make sure to include the package declaration at the beginning of each class file.

For example, you can have a class named MypackageClass:

```
// File: MypackageClass.java

package com.example.mypack;

public class MypackageClass {
```

```
  // Class members and methods
        Public void add(int a,int b){
            Return a+b;
        }
}
```

Importing Subpackage:

To import classes from the subpackage (util) into another package, use the import statement. Specify the hierarchy of classes. we use the dot (.) to specify the package structure in the import statement.

For example, let's say you want to import MypackageClass into a class in the com.example.app package:

```
// File: MyApp.java
package com.example.app;
import com.example.mypack.MypackageClass;


public class MyApp {
   public static void main(String[] args) {
      MypackageClass a = new MypackageClass();


   }
}
```

Package Import

In Java, when you're importing classes from other packages, you have a few options for how you specify the import statement:

Single Type Import: Importing a single class from a specific package.

```
import package_name.ClassName;
import java.util.ArrayList;
```

Wildcard Import: Importing all the classes from a specific package.

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

Static Import: Importing static members (fields or methods) of a class.

```
import static package_name.ClassName.*;
Import static java.lang.Math.*;
```