UNIT 4
PACKAGES, MULTITHREADING

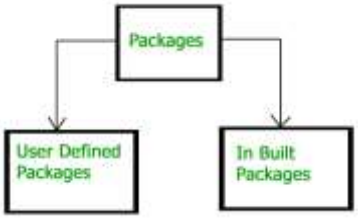
PACKAGES

A java package is a group of similar types of classes, interfaces and sub-packages.
Package in java can be categorized in two form, built-in package and user-defined package.
There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.
Here, we will have the detailed learning of creating and using user-defined packages.

Advantage of Java Package

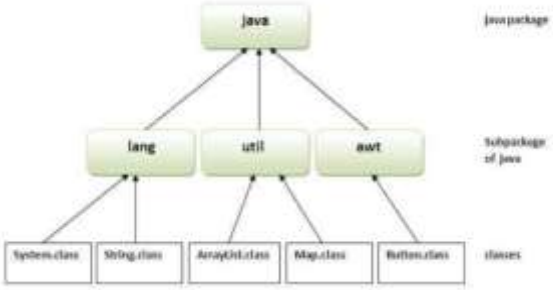
1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
2) Java package provides access protection.
3) Java package removes naming collision.

Types of Packages



Built-in Packages

These packages consist of a large number of classes which are a part of Java API.Some of the commonly used built-in packages are:



- **java.lang**: Contains language support classes(e.g classes which defines primitive data types, math operations). This package is automatically imported.
- **java.io**: Contains classes for supporting input / output operations.
- **java.util**: Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.
- **java.applet**: Contains classes for creating Applets.
- **java.awt**: Contain classes for implementing the components for graphical user interfaces (like button , ;menus etc).

User defined Packages

A Java package is a group of similar types of classes, interfaces and sub-packages. A package, in the context of computing and software development, typically refers to a collection of files and resources bundled together for distribution, installation, and use. Packages are often used to organize and distribute software libraries, frameworks, applications, or modules. They typically contain executable code, configuration files, documentation, and other necessary resources. In various programming languages and environments, such as Python, Java, and JavaScript, packages are used to manage dependencies, facilitate code reuse, and streamline development processes. They are often distributed via package managers, which automate the process of downloading, installing, and updating software components. Packages are used for:

- Preventing naming conflicts. For example, there can be two classes with the name Employee in two college.staff.ee.Employee packages, college.staff.cse.Employee and
- Making searching/locating and usage of classes, interfaces, enumerations and annotations easier
- Providing controlled access: protected and default have package-level access control. A protected member is accessible by classes in the same package and its subclasses. A default member (without any access specifier) is accessible by classes in the same package only.
- Packages can be considered as data encapsulation (or data-hiding)

Creating Packages

- Step 1: Simply include a package command that has the first statement in the Java source file. Any class you declare within that file will belong to the specified package.
  Syntax:
  package packagename;
  Example:
  package mypack;
- Step 2: Next define the class that is to be put in the package and declare it as public.
- Step 3: Now store the classname.java file in the directory having the name same as package name.
- Step 4: The file is to be compiled, which creates class file in the directory java also supports the package hierarchy, which allows to group of related classes into a package and then group related packages into a larger package. We can achieve this by specifying multiple names in a package statement, separated by a dot (.).

Syntax:
package outerpackagename.innerpackagename;
Example:
package dilkk.parni;

Accessing package: A Java system package can be accessed either using a fully qualified classname or using an import statement. We generally use import statements.
Syntax:
import package.classname;
or
import package.*;

Java program on creating a package

```
package myPack;
class Folder
{
        public static void main(String args[])
        {
                System.out.println("Welcome to package");
        }
}
```

To compile: javac –d . ClassName.java
The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

```
E:\JAVA\ANU\Anitha>javac -d . Folder.java

E:\JAVA\ANU\Anitha>java myPack.Folder
Welcome to package
```

Accessing package from another package:

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

1) **Using packagename.***

- If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.
- The import keyword is used to make the classes and interface of another package accessible to the current package.

```
package pack;
public class A
{
        public void msg()
        {
                System.out.println("using packageName.*");
        }
}
```

```
package newPack;
import pack.*;
class B
{
        public static void main(String args[])
        {
                A a=new A();
                a.msg();
        }
}
```

```
E:\JAVA\ANU\Anitha>javac -d . A.java

E:\JAVA\ANU\Anitha>javac -d . B.java

E:\JAVA\ANU\Anitha>java newPack.B
using packageName.*
```

2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

```
package pack;
public class A
{
        public void msg()
        {
                System.out.println("using packageName.class ");
        }
}
```

```
package newPack;
import pack.A;
class B
{
        public static void main(String args[])
        {
                A a=new A();
                a.msg();
        }
}
```

```
E:\JAVA\ANU\Anitha>javac -d . A.java

E:\JAVA\ANU\Anitha>javac -d . B.java

E:\JAVA\ANU\Anitha>java newPack.B
using packageName.class
```

3) Using fully qualified name

- If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.
- It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

```
package pack;
public class A
{
        public void msg()
        {
                System.out.println("using Fully qualified name ");
        }
}
```

```
package newPack;
class B
{
        public static void main(String args[])
        {
                pack.A a=new pack.A();
                a.msg();
        }
}
```

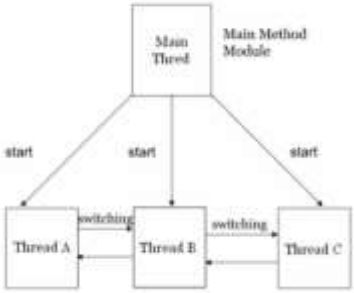
```
E:\JAVA\ANU\Anitha>javac -d . A.java

E:\JAVA\ANU\Anitha>javac -d . B.java

E:\JAVA\ANU\Anitha>java newPack.B
using Fully qualified name
```

MULTITHREADING

- Multithreading in Java is a process of executing multiple threads simultaneously.
- A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.
- However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.
- Java Multithreading is mostly used in games, animation, etc.



Advantages of Java Multithreading

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.

Main Thread

- The main thread in Java is a crucial component of any Java program. The thread is automatically created when a Java program starts, and maintains the main() method of the application. The main() method, which serves as the program's entrance point, is the initial method utilised at the start of the programme.
- The main thread works as a single thread, which means that each line of code is executed sequentially. Other names for this thread are "main application thread" and "main thread".
- Any additional threads that the programme may require must be created and started by the main thread. It is also in charge of organising how these threads are carried out and making sure they function effectively.

static Thread.currentThread():

The currentThread() method of thread class is used to return a reference to the currently executing thread object.

```
class EgThread
{
        public static void main(String args[])
        {
                Thread t=Thread.currentThread();
                System.out.println(t);
                t.setName("myfirstThread");
                System.out.println(t);
                System.out.println(t.getPriority());
                t.setPriority(10);
                System.out.println(t.getPriority());
        }
}
```

```
E:\JAVA\ANU>javac EgThread.java

E:\JAVA\ANU>java EgThread
Thread[#1,main,5,main]
Thread[#1,myfirstThread,5,main]
5
10
```

Thread:

- A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.
- Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.

Creating Threads

A thread can programmatically be created by:

- Implementing the java.lang.Runnable interface.
- Extending the java.lang.Thread class.

By implementing Runnable interface

The Runnable interface contains the run() method that is required for implementing the threads in our program. To do this we must perform the following steps:

- Declare a class as implementing the Runnable interface.
- Implement the run() method
- Create a Thread by defining an object that is instantiated from this "runnable" class as the target of the thread
- Call the thread's start() method to run the thread

```
class ExThrd implements Runnable
{
        public void run()
        {
                System.out.println("Thread is running");
        }

        public static void main(String args[])
        {
                ExThrd e=new ExThrd();
                Thread t=new Thread(e);
                t.start();
        }
}
```

```
E:\JAVA\ANU>javac ExThrd.java

E:\JAVA\ANU>java ExThrd
Thread is running
```

By extending Thread class

We can make our thread by extending the Thread class of java.lang.Thread class. This gives us access to all the methods of the Thread. It includes the following steps:

- Declare the class as Extending the Thread class.
- Override the "run()" method that is responsible for running the thread.
- Create a thread and call the "start()" method to instantiate the Thread Execution.

```
class EgThrd extends Thread
{
        public void run()
        {
                System.out.println("Thread is running");
        }

        public static void main(String args[])
        {
                EgThrd e=new EgThrd();
                e.start();
        }
}
```

```
E:\JAVA\ANU>javac EgThrd.java

E:\JAVA\ANU>java EgThrd
Thread is running
```

Life of a Thraed: (read from text book pg no :225)

Thread Methods: (read from text book pg no :232)

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. Note that 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

- **getPriority():** The java.lang.Thread.getPriority() method returns the priority of the given thread.
- **setPriority(int newPriority):** The java.lang.Thread.setPriority() 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:

- public static int MIN_PRIORITY
- public static int NORM_PRIORITY
- 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 EgThread extends Thread
{
        public void run()
        {
                System.out.println("thread is running");
        }
        public static void main(String args[])
        {
                EgThread t1=new EgThread();
                EgThread t2=new EgThread();
                EgThread t3=new EgThread();

                System.out.println(t1.getPriority());
                System.out.println(t2.getPriority());
                System.out.println(t3.getPriority());

                t1.setPriority(1);
                t2.setPriority(10);
                t3.setPriority(6);

                System.out.println(t1.getName());
                System.out.println(t2.getName());
                System.out.println(t3.getName());

                System.out.println(t1.getPriority());
                System.out.println(t2.getPriority());
                System.out.println(t3.getPriority());

        }
}
```

```
E:\JAVA\ANU>javac EgThread.java

E:\JAVA\ANU>java EgThread
5
5
5
Thread-0
Thread-1
Thread-2
1
10
6
```

Synchronization

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization.

The key to synchronization is the concept of the monitor (also called a semaphore). A monitor is an object that is used as a mutually exclusive lock, or mutex. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These other threads are said to be waiting for the monitor. A thread that owns a monitor can renter the same monitor if it so desires.

```
class Counter
{
	int count;
	synchronized public void inc()
	{
		count++;
	}
}
class MyThread1 extends Thread
{
	Counter c;
	MyThread1(Counter c)
	{
		this.c=c;
	}
	public void run()
	{
		for(int i=1;i<=1000;i++)
		{
			c.inc();
		}
	}
}
class MyThread2 extends Thread
{
	Counter c;
	MyThread2(Counter c)
	{
		this.c=c;
	}
	public void run()
	{
		for(int i=1;i<=1000;i++)
		{
			c.inc();
		}
	}
}
public class Synch
{
	public static void main(String args[])
	{
		Counter c=new Counter();
		MyThread1 t1=new MyThread1(c);
		MyThread2 t2=new MyThread2(c);

		t1.start();
		t2.start();

		try{Thread.sleep(100);}catch(Exception e){System.out.println(e);}
		System.out.println("main : "+c.count);
	}
}
```

```
E:\JAVA\ANU>javac Synch.java

E:\JAVA\ANU>java Synch
main : 2000
```