JAVA Packages.

Packages are containers for classes. They are used to keep the class name space compartmentalized. For example, a package allows you to create a class named List, which you can store in your own package without concern that it will collide with some other class named List stored elsewhere. Packages are stored in a hierarchial manner and are explicitly imported into new class definitions.

The package is both a naming and a visibility control mechanism.

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 case is significant, and the directory name must match the package name exactly.

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

    package java.awt.image;

needs to be stored in java\awt\image in a Windows environment. Be sure to choose your package names carefully.

You cannot rename a package without renaming the directory in which the classes are stored.

Let us understand the "import" statement.

Say you created two package namely package com.kunal.packages.a; & package com.kunal.packages.b;

```
① package com.kunal.packages.b;
  public class Message {
     psvm {

     }
     public static void message(){
         sout("This course is awesome");
     }
  }
```

* If you make it private, it can't be accessed in another package.

In second package you write,

```
package com.kunal.packages.a;
import static com.kunal.packages.b. Message.message;
public class Greeting {
    psvm{
        sout ("Hello world");
        message();
    }
}
```

this will be printed.

message(); ← when you press Alt + Enter

Clicking Ctrl + Click will take you to the package from where it was imported.

O/P:

```
Hello World
This course is awesome.
```

i.e: It imported from some other Java class and package.
How?

Q How does java run-time system know where to look for packages that you create?
- First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found.
- Second, you can specify a directory path or paths by setting the CLASSPATH environmental variable.
- Third, you can use the -classpath option with java and javac to specify the path to your classes.

NOTE When a package is imported, only those items within the package declared as public will be available to non-subclasses in the importing code.

⇒ "static" in Java

Say you create two Java classes in the same package staticExample.

1st Java Class

```
package staticExample;

public class Human {
        int age;
        String name;
        int salary;
        boolean married;

        public Human(int age, String name, int salary, boolean married){
            this.age=age;
            this.name=name;
            this.salary=salary;
            this.married=married;
        }
}
```

2nd Java class

```
package staticExample;

public class Main {
    psvm {
        Human kunal = new Human(22, "Kunal", 10000, false);
        Human rahul = new Human(34, "Rahul", 15000, true);
        sout (kunal.name);
    }
}
```

Both the class are in same package so to import Human you do not need to write import statement

O/P: Kunal

We are all humans and each of us have ~~all~~ the above traits i.e. age, name, salary, married status but values are different.

Q what if there was some sort of trait/property that is common to all human beings?
=> Yes, population

For each and every human, population will remain same b/c it is a fact.

So, the property is not related to any object, are cla static variables or methods.

Ex. ~~&~~ if you add the following in Human class
~~long population~~;
static long population;

& add ~~this~~ Human.population += 1; in public Human

& ~~we~~ In 2nd Java class, you ~~write~~ add,

```
sout(Human.population);
sout(Human.population);
```

o/p: 2
2

| Even if you write kunal.population or ~~rahul~~.population, it will give 2 but static is not depended on objects so we write name of the class.

=> To sum up the above things in more theoratical way:
When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object. You can declare both methods and variables to be static. The most common

example of a static member is main().
main() is declared as static because it must be called before any object exist. static method is Java is a method which belongs to the class and not the object.

→ A static method can access only static data. It cannot access non-static data (instance variable).

For example.

```
void greeting (){
    sout("Hello world");
}
psvm{
    greeting(); ← this will give you an error
}
```

b/c psvm(static) it can't access non-static method. If you wrote, static void greeting(), it would work.

can't access non-static in static member.

This is because non-static member belongs to an instance/object. It's meaningless without somehow resolving which instance of a class you are talking about. In a static context, you don't have an instance, that's why you can't access a non-static member without explicitly mentioning an object reference.

→ You can access static member in non-static.

✱ ⇒ In fact, you can access a non-static member in a static context by specifying the object reference explicitly.

Java class name

```
Ex  psvm {
      Main obj = new Main();
      obj.greeting();
    }
```

```
void greeting () {
        sout ("Hello   world");
}
```

Now, it works.
In short, you cannot access non-static stuff without referencing their instances in a static context.

→ A static method can call only other static methods and cannot call a non-static method from it.
→ A static method can be accessed directly by the class name and doesn't need any object.
→ A static method cannot refer to "this" or "super" keywords in any anyway.
⇒ Initialising static variable.
* If you need to do computation in order to initialize your static variables, you can declare a static block that gets executed exactly once, when the class is first loaded.

Ex
```
class UseStatic {
    static int a=3;
    static int b;
    static void meth(int x){
        sout("x= "+x);
        sout("a= "+a);
        sout("b= "+b);
    }
    static{
        sout("static block initialized.");
        b=a* 4;
    }
    psum{
        meth(42);
    }
}
```

As soon as the UseStatic class is loaded, all of the static statements are run. First, a is set to 3, then the static block executes, which prints a message and then initializes b to a*4 or 12. Then main() is called which calls meth(), passing 42 to x. The three println() statements refer to the static variables a and b as well as the local variable x.

Here is the output of the program:

```
Static block initialized. x = 42.
a=3
b=12.
```

NOTE: 1) main method is static, since it must be accessible for an application to run, before any instantiation takes place.

2) Only nested classes can be static. (Outside class can't be static)

3) Static inner classes can have static variables.

You can't override the inherited static methods, as in Java overriding take place by resolving the type of object at run-time and not compile time, and then calling the respective method.

Static methods are class level methods, so it is always resolved during compile time.

Static INTERFACE METHODS are not inherited by either an implementing class or a sub-interface.

NOTE:
```
public class Static {
    //class Test //ERROR
    static class Test{
        String name;

        public Test(String name){
            this.name=name;
        }
    }
    psvm{
        Test a=new Test("Kunal");
        Test b= new Test("Rahul");

        sout(a.name); //Kunal
        sout(b.name); //Rahul
    }
}
```

Because:
The static keyword may modify the declaration of a member type C within the body of a non-inner class or interface T. Its effect is to declare that C is not an inner class. Just as a static method of T has no current instance of T in its body, C also has no current instance T, nor does it have any lexically enclosing instances.

Here, test does not have any instance of its outer class Static. Neither does main.

But main & Test can have instances of each other.