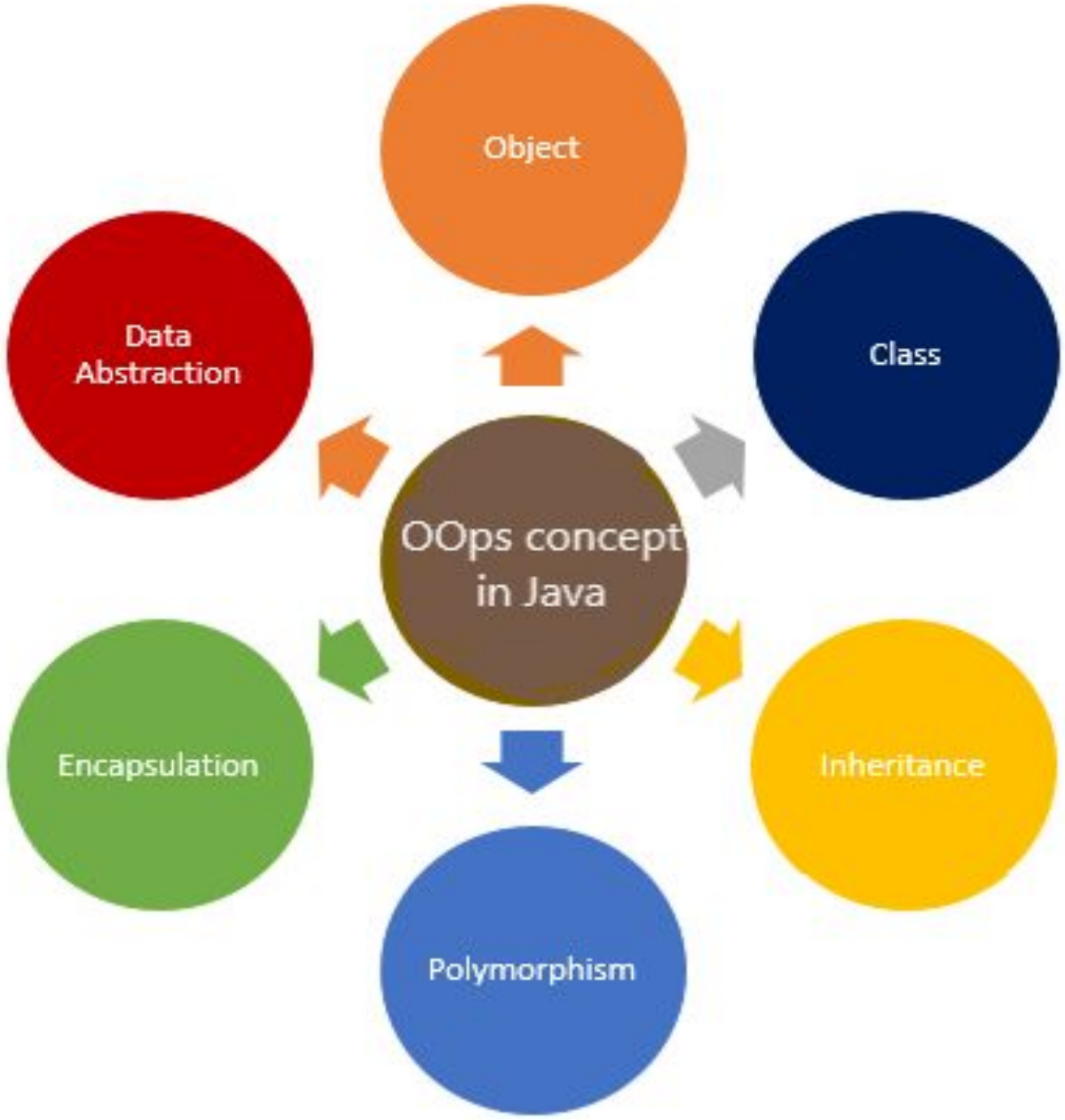


CS20004: Object Oriented Programming using Java

Lec-14

In this Discussion . . .

- Namespace Management
- Package
  - Package Definition
  - Packages and Directories
  - Package Hierarchy and Package Finding
  - Importing of Packages
  - Accessing Packages from other Packages
  - Subpackage
- References

Namespace Management

- Classes written so far all belong to a single name space: a unique name has to be chosen for each class to avoid name collision
- Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is a **package**.

Package

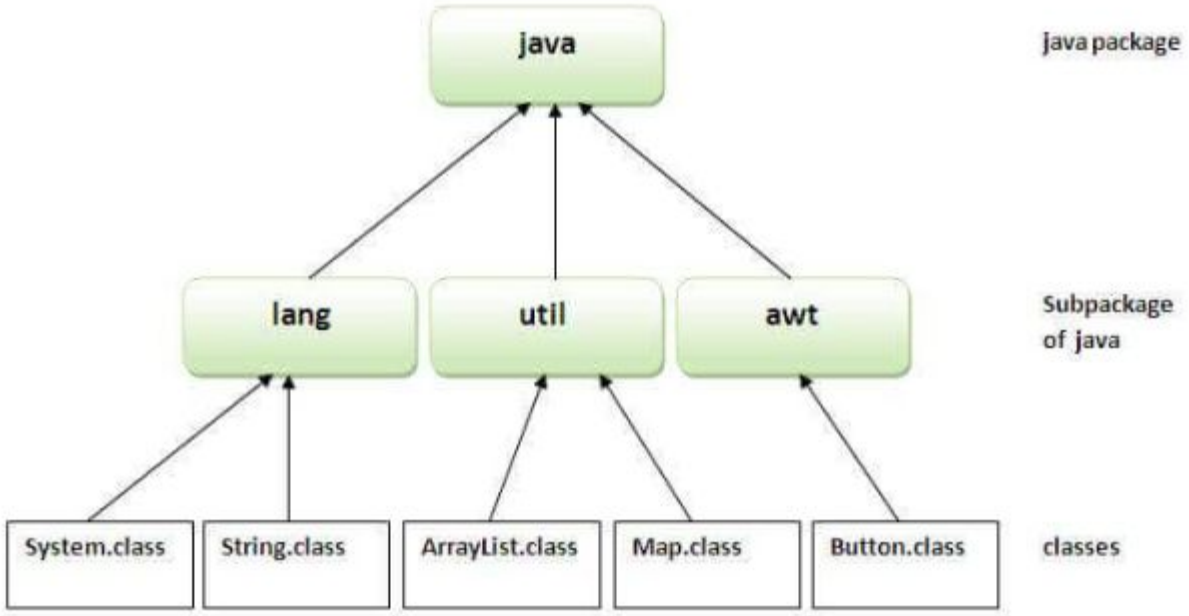
- A package is both a naming and a visibility control mechanism:
  - divides the name space into disjoint subsets:
    - It is possible to define classes within a package that are not accessible by code outside the package
  - controls the visibility of classes and their members:
    - It is possible to define class members that are only exposed to other members of the same package
- Same-package classes may have an intimate knowledge of each other, but not expose that knowledge to other packages

Package

- A **java package** is a group of similar types of classes, interfaces and sub-packages.
- Package in java can be categorized in two forms:
  - 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.

Advantages of using Packages

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



Package Definition

- Creating a package is quite easy:
  - The **package** keyword is used to create a package in java.

Syntax:

```
package mypackage;
```

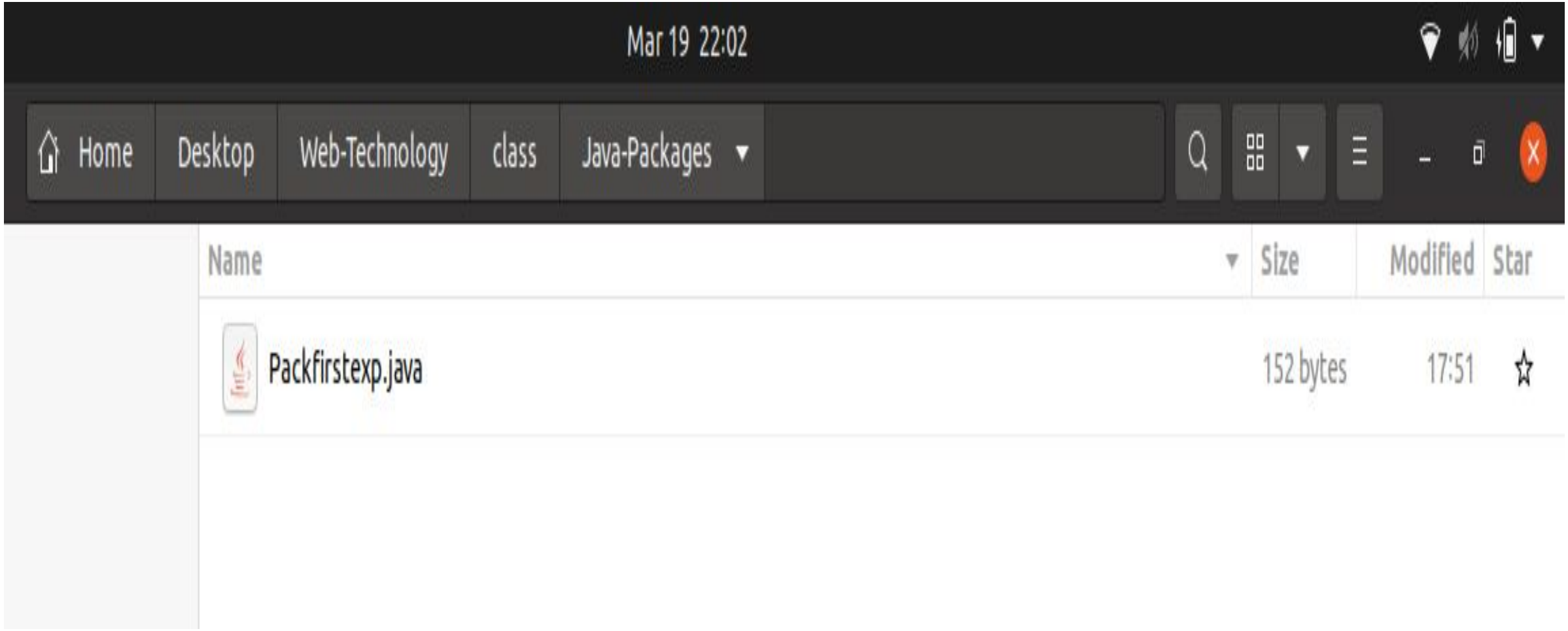
Package Example

- The **package** keyword is used to create a package in java.
- The package statement creates a name space where classes are stored
- When the package statement is omitted, class names are put into the default package which has no name

A Sample Program

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

Package Example



Compiling the java package

- If we are not using any IDE, then we need to follow the given syntax:

```
javac -d directory java-filename
```

For Ex-

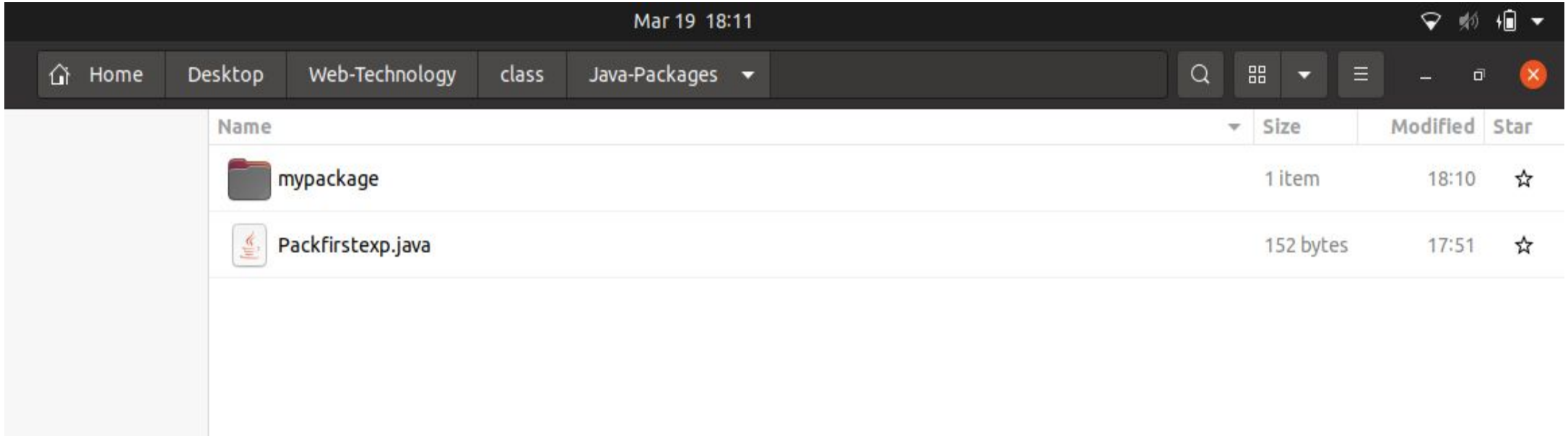
```
javac -d . Packfirstexp.java
```

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

Compiling the java package

- If we are not using any IDE, then we need to follow the given syntax:

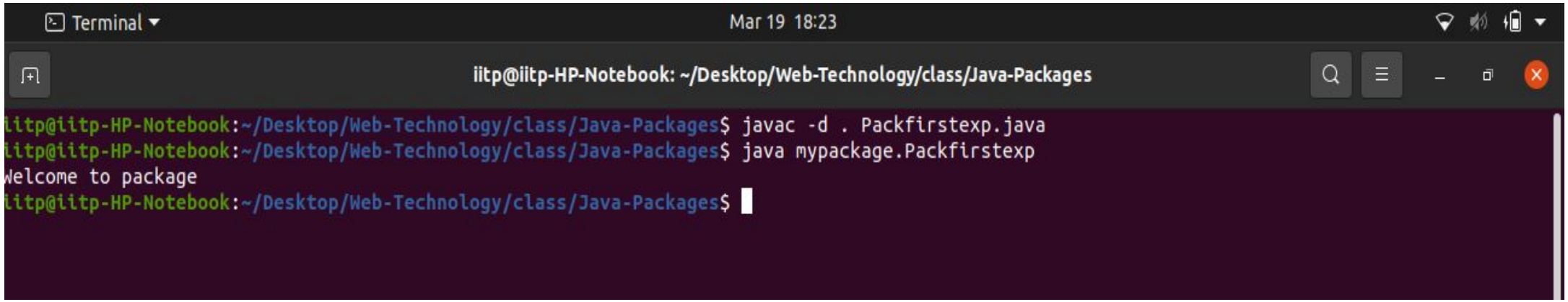
```
javac -d . Packfirstexp.java
```



Running the java package program

- We need to use fully qualified name e.g. mypackage.Packfirstexp to run the class using the given syntax:

```
java myPackage.Packfirstexp
```



Packages and Directories

- Java uses file system directories to store packages

  package myPackage;

  class MyClass1 {...}

  class MyClass2 {...}

- The bytecode files MyClass1.class and MyClass2.class must be stored in a directory **myPackage.**
- **Case is significant! Directory names must match package names exactly.**

Package Hierarchy and Package Finding

- To create a package hierarchy, separate each package name with a dot:

  package myPackage1.myPackage2.myPackage3;

- A package hierarchy must be reflected in the file system of your java development system
- ***You cannot rename a package without renaming its directory!***
- As packages are stored in directories, how does the Java run-time system know where to look for packages?
  - **The current directory is the default start point-** if packages are stored in the current directory or sub-directories, they will be found
  - **Specify a directory path or paths** by setting the CLASSPATH environment variable

Package Hierarchy and Package Finding

- CLASSPATH - environment variable that points to the root directory of the system's package hierarchy
- A package hierarchy must be reflected in the file system of your java development system
- Several root directories may be specified in CLASSPATH, e.g. the current directory and the *C:\ myJava directory:* ;*C:\ myJava*
- Java will search for the required packages by looking up subsequent directories described in the CLASSPATH variable
- In order for a program to find myPackage, one of the following must be true:
  - *program is executed from the directory immediately above myPackage (the parent of myPackage directory)*
  - *CLASSPATH must be set to include the path to myPackage*

Package Hierarchy and Package Finding

```
package MyPack;

class Balance
{
	String name;
	double bal;
	Balance(String n, double b)
	{
		name = n;
		bal = b;
	}
	void show()
	{
		if(bal < 0)
		{
			System.out.println("--> >");
		}
		System.out.println(name+":$"+bal);
	}
}
```

```
class AccountBalance
{
	public static void main(String args[])
	{
		Balance current[] = new Balance[3];
		current[0] = new Balance("Sourajit
Behera",257.32);
		current[1] = new Balance("Student 1", 157.02);
		current[2] = new Balance("Student 2",-12.3);
		for(int i = 0; i<3;i++)
		{
			current[i].show();
		}
	}
}
```

Package Hierarchy and Package Finding

- **Save, Compile, and Execute**
  - call the file AccountBalance.java
  - save the file in the directory MyPack
  - compile; AccountBalance.class should be also in MyPack
  - set access to MyPack in CLASSPATH variable, or make the parent of MyPack your current directory
  - **run**: **java MyPack.AccountBalance**
  - **You need to be in the directory above MyPack when executing this command**
  - *The .class filename must be qualified with its package name*

Importing of Packages

- Since classes within packages must be fully-qualified with their package names, it would be tedious to always type long dot-separated names
- The import statement allows to use classes or whole packages directly
- **Importing of a concrete class:**
  - *import myPackage1.myPackage2.myClass;*
- **Importing of all classes within a package:**
  - *import myPackage1.myPackage2.*;*

How to access packages from another packages

- There are three ways to access the package from outside the package:

  a. import package.*;

  b. import package.classname;

  c. fully qualified name.

Using packagename.*

- If we 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.

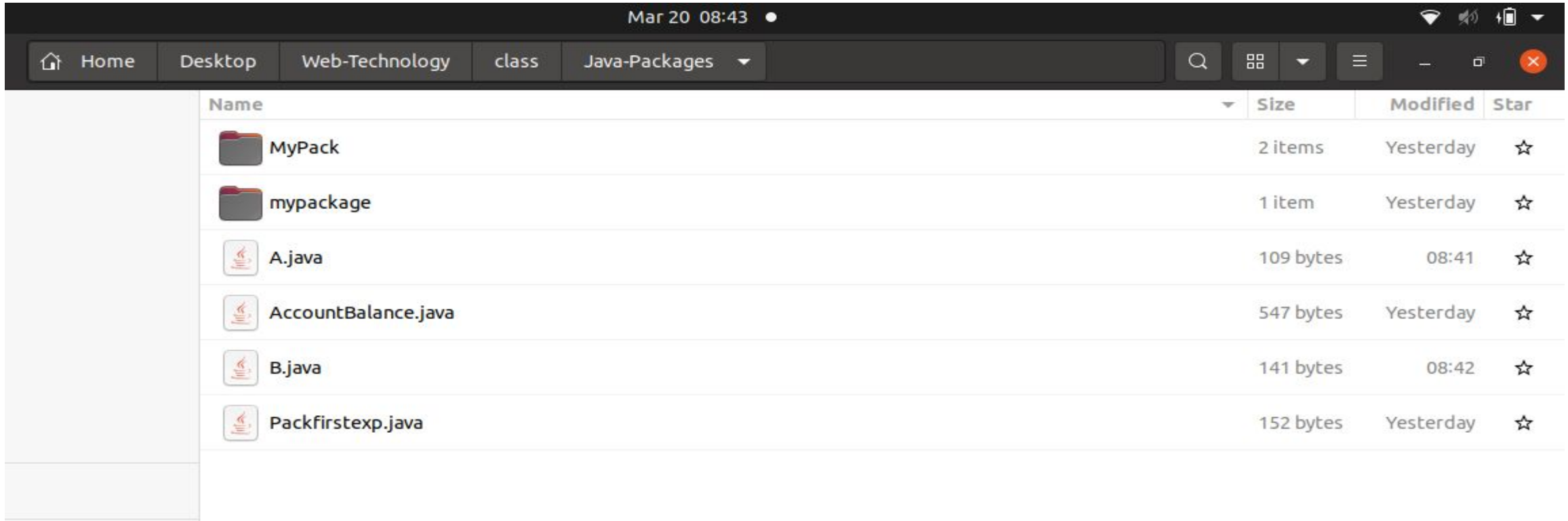
```
package pack1;
public class A
{
  public void msg()
  {
        System.out.println("Inside pack1");
  }
}
```

```
package pack2;
import pack1.*;

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

Using packagename.*

- If we 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.



Using packagename.*

- If we use **package.*** then all the classes and interfaces of this package will be accessible but not subpackages.
- Directly trying to compile the class with main function, i.e., B.java will result in an error which can be shown as below:

```
iitp@iitp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Packages$ javac -d . B.java
B.java:2: error: package pack1 does not exist
import pack1.*;
^
B.java:8: error: cannot find symbol
                A obj = new A();
                ^
  symbol:   class A
  location: class B
B.java:8: error: cannot find symbol
                A obj = new A();
                            ^
  symbol:   class A
  location: class B
3 errors
```

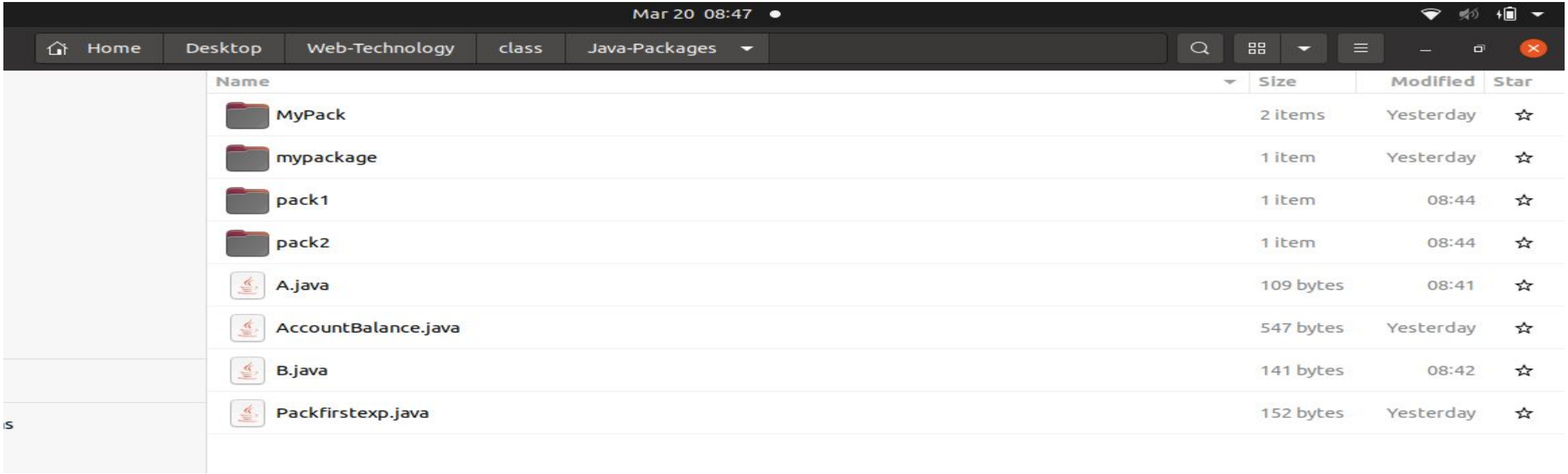
Using packagename.*

- If we use **package.*** then all the classes and interfaces of this package will be accessible but not subpackages.
- Thus, the correct order of compiling the java files comprises first dealing with A.java followed by then with B.java

```
itp@itp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Packages$
itp@itp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Packages$ javac -d . A.java
itp@itp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Packages$ javac -d . B.java
```

Using packagename.*

- If we use **package.*** then all the classes and interfaces of this package will be accessible but not subpackages.
- Thus, the correct order of compiling the java files comprises first dealing with A.java followed by then with B.java



Using packagename.*

- If we use **package.*** then all the classes and interfaces of this package will be accessible but not subpackages.
- Thus, we need to run the java file with the main function (i.e., B.class) but with the full package name.

```
iitp@iitp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Packages$
iitp@iitp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Packages$ javac -d . A.java
iitp@iitp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Packages$ javac -d . B.java
iitp@iitp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Packages$ java pack2.B
Inside pack1
iitp@iitp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Packages$
```

Using packagename.classname

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

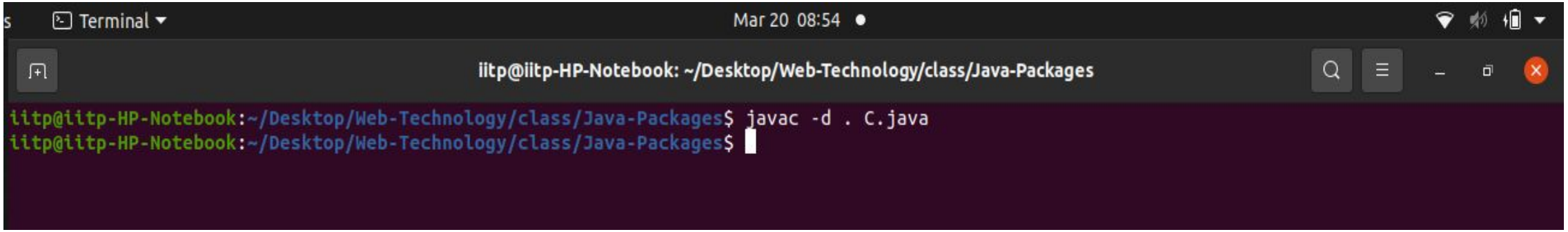
```
package pack3;
public class C
{
	public void msg()
	{
		System.out.println("Inside the pack3 package");
	}
}
```

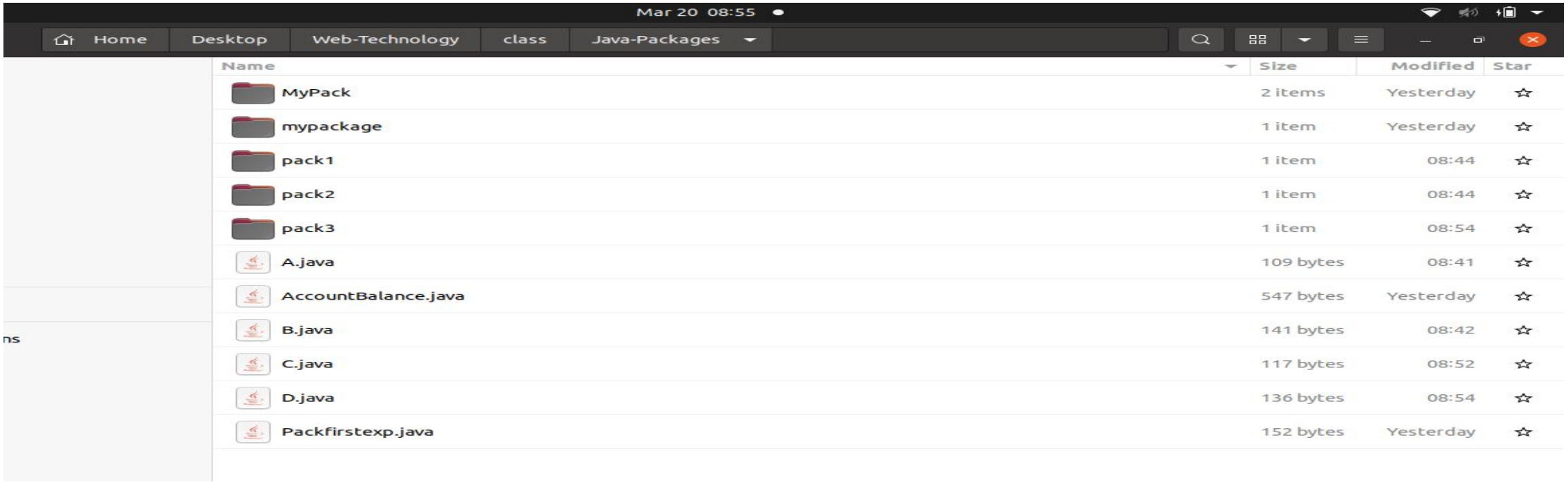
```
package pack4;
import pack3.C;

class D
{
	public static void main(String args[])
	{
		C obj = new C();
		obj.msg();
	}
}
```

Using packagename.classname

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

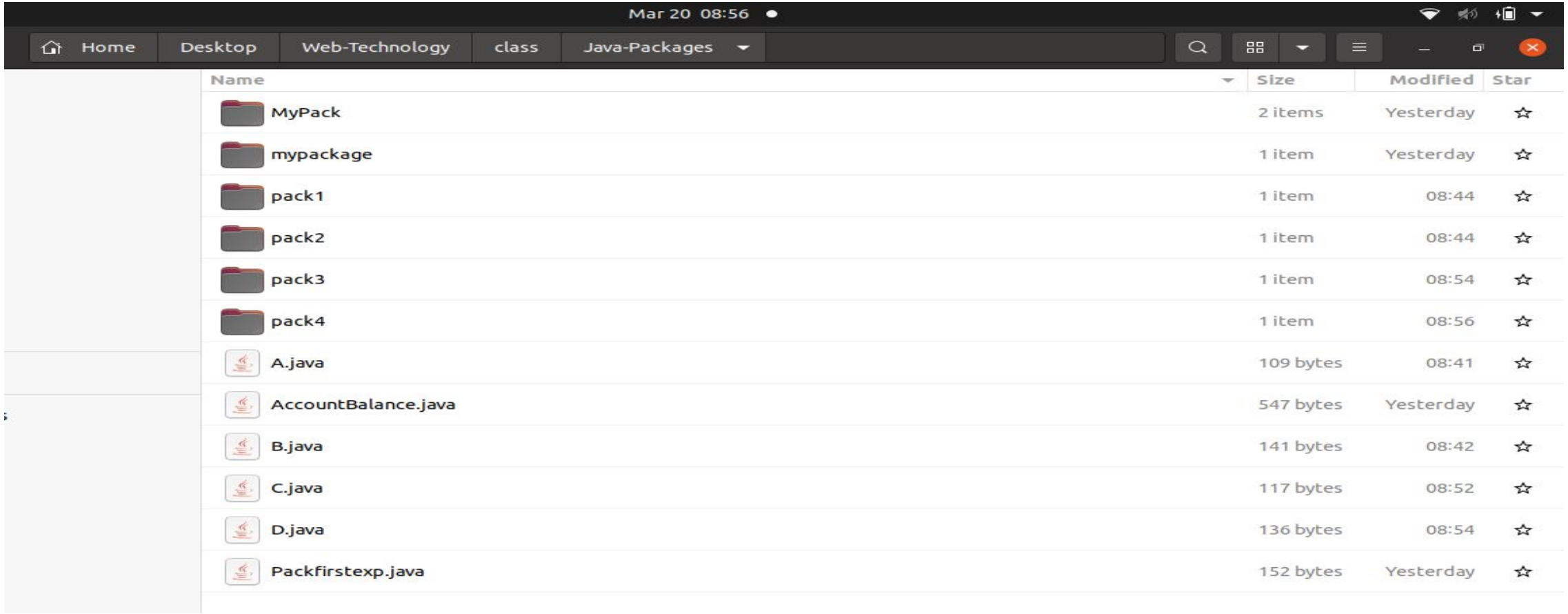




Using packagename.classname

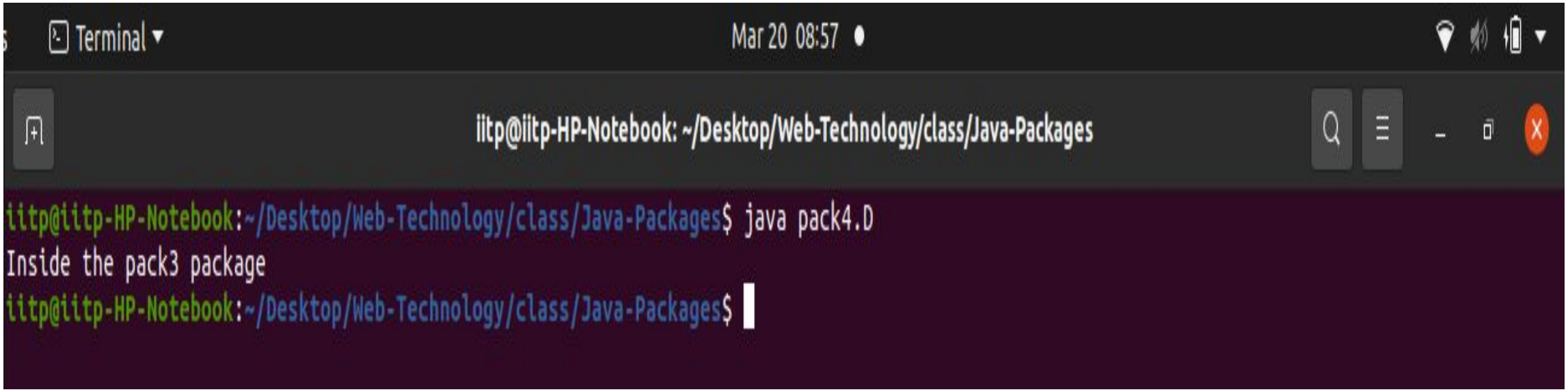
- If you import **package.classname** then only declared class of this package will be accessible.

```
iitp@iitp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Packages$ javac -d . D.java
iitp@iitp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Packages$
```



Using packagename.classname

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



Using fully qualified name

- If we use the fully qualified name then only declared class of the package will be accessible.
- In this case there is no need to import. But we need to use fully qualified name every time when we 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.

Using fully qualified name

- If we use the fully qualified name then only declared class of the package will be accessible.

package pack5;
public class E{
  public void msg()
  {
  System.out.println("Inside the pack5 package");
  }
}
	package pack6;
	class F
	{
	  public static void main(String args[])
	  {
	  pack5.E obj = new pack5.E();//using fully qualified name
	  obj.msg();
	  }
	}

Using fully qualified name

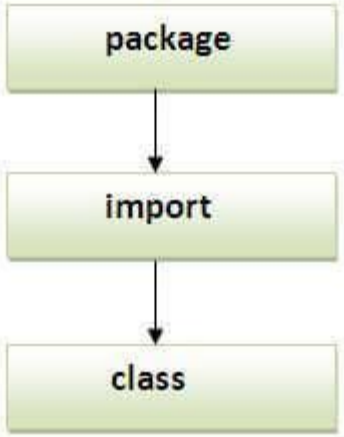
- If we use the fully qualified name then only declared class of the package will be accessible.

```
Terminal                                   Mar 20 09:06
iitp@iitp-HP-Notebook: ~/Desktop/Web-Technology/class/Java-Packages
iitp@iitp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Packages$ javac -d . E.java
iitp@iitp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Packages$ javac -d . F.java
iitp@iitp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Packages$ java pack6.F
Inside the pack5 package
iitp@iitp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Packages$
```

MyPack	2 items	Yesterday
mypackage	1 item	Yesterday
pack1	1 item	08:44
pack2	1 item	08:44
pack3	1 item	08:54
pack4	1 item	08:56
pack5	1 item	09:06
pack6	1 item	09:06
A.java	109 bytes	08:41
AccountBalance.java	547 bytes	Yesterday
B.java	141 bytes	08:42
C.java	117 bytes	08:52
D.java	136 bytes	08:54
E.java	120 bytes	09:04
F.java	157 bytes	09:05

Using fully qualified name

- If we use the fully qualified name then only declared class of the package will be accessible.
- **Note**:- If we import a package, then all the classes and interface of that package will be imported **excluding** the classes and interfaces of the subpackages. Hence, we need to import the subpackage as well.
- Sequence of the program must be package then import then class.



Subpackage

- Package inside the package is called the **subpackage**. It should be created to **categorize the package further.**
- Let's take an example.
  - Sun Microsystem has defined a package named **java** that contains many classes like System, String, Reader, Writer, Socket etc.
  - These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on.
  - So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.

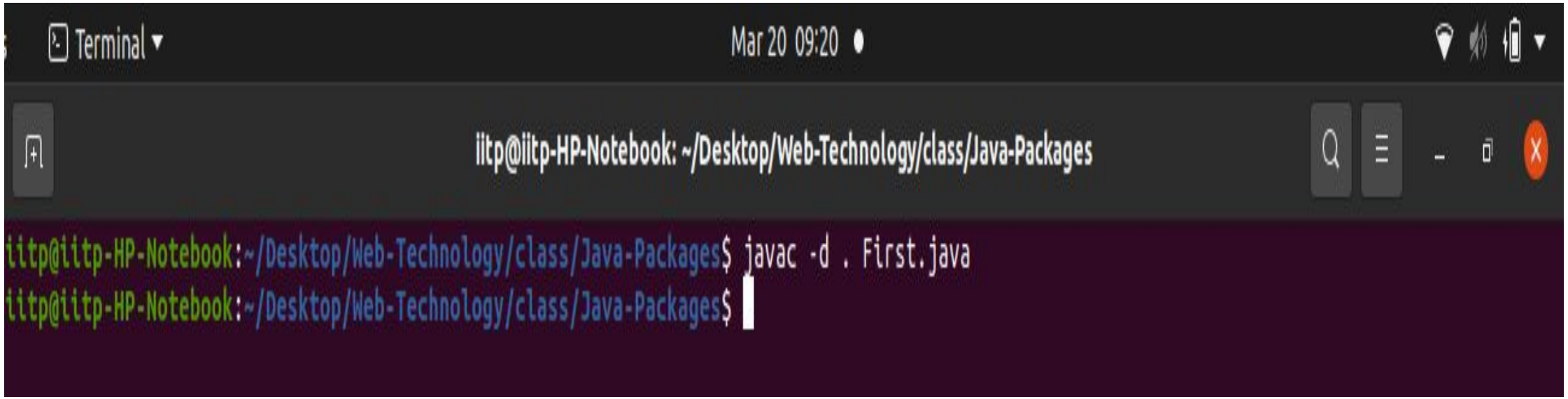
Subpackage

- Package inside the package is called the **subpackage**. It should be created to **categorize the package further.**
- **Note**:- The standard of defining package is **domain.company.package** e.g. com.KIIT.bean or org.sssit.dao.

```
package pack6.core;
class First
{
	public static void main(String args[])
	{
		System.out.println("Hello subpackage");
	}
}
```

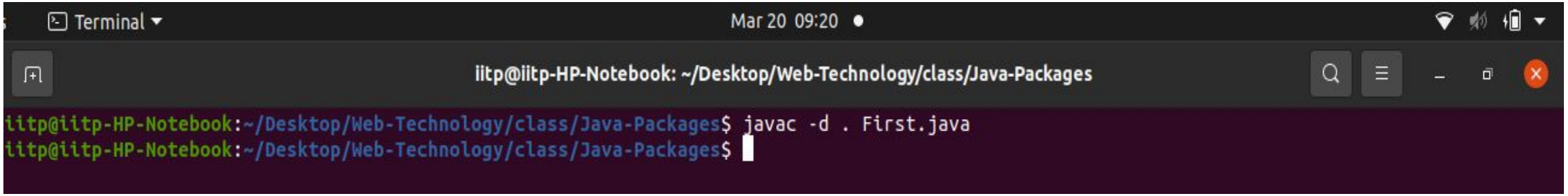
Subpackage

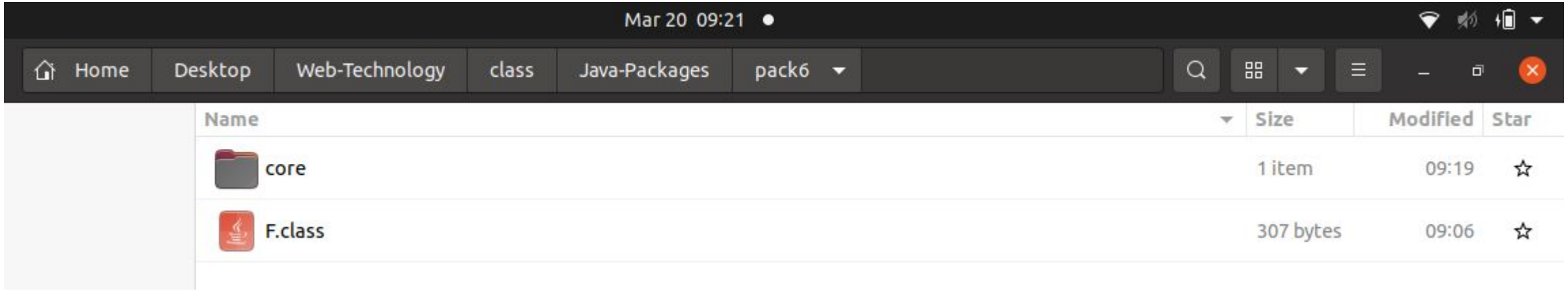
- Package inside the package is called the **subpackage**. It should be created to **categorize the package further.**
- **Note**:- The standard of defining package is **domain.company.package** e.g. com.KIIT.bean or org.sssit.dao.



Subpackage

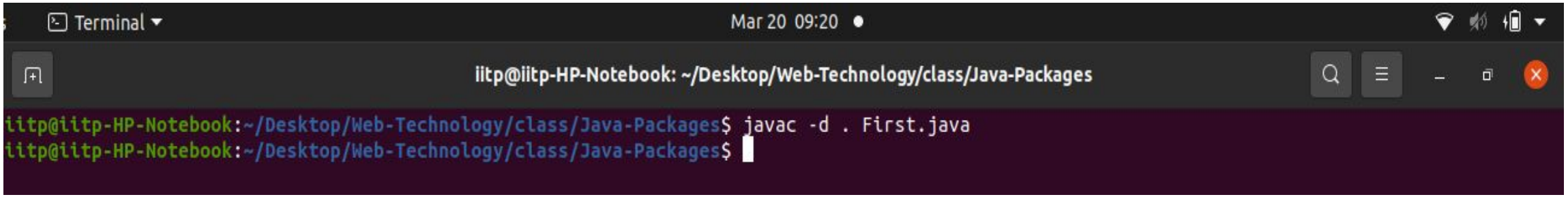
- Package inside the package is called the **subpackage**. It should be created to **categorize the package further.**
- **Note**:- The standard of defining package is **domain.company.package** e.g. com.KIIT.bean or org.sssit.dao.

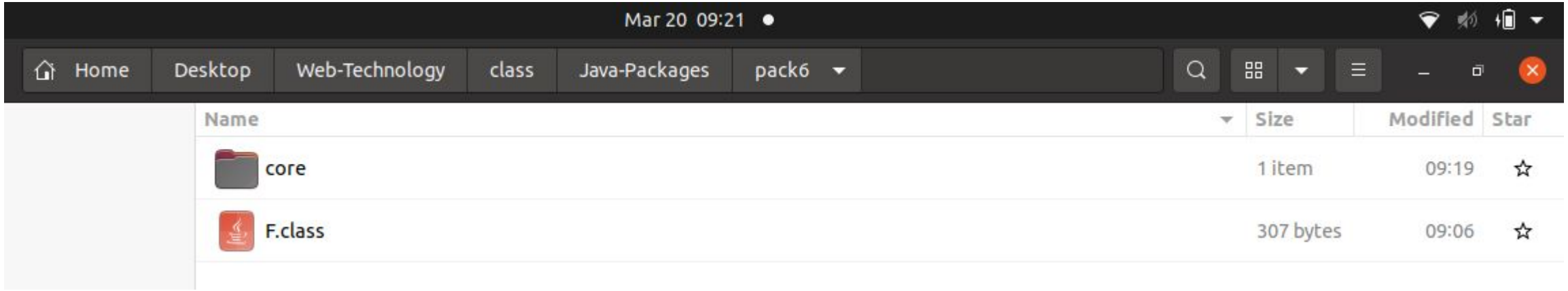


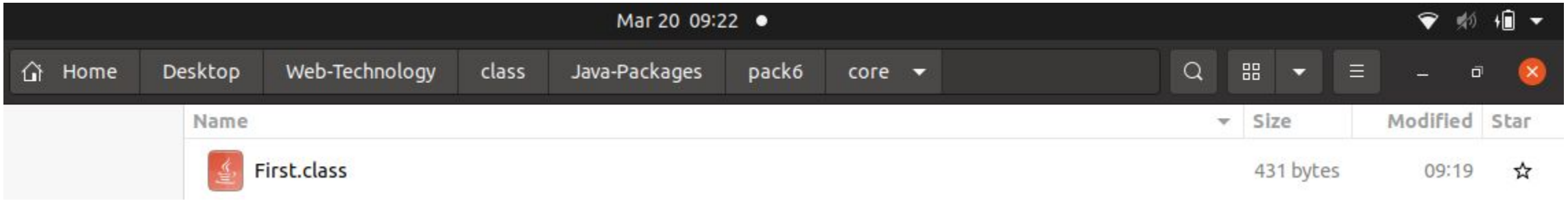


Subpackage

- Package inside the package is called the **subpackage**. It should be created to **categorize the package further.**

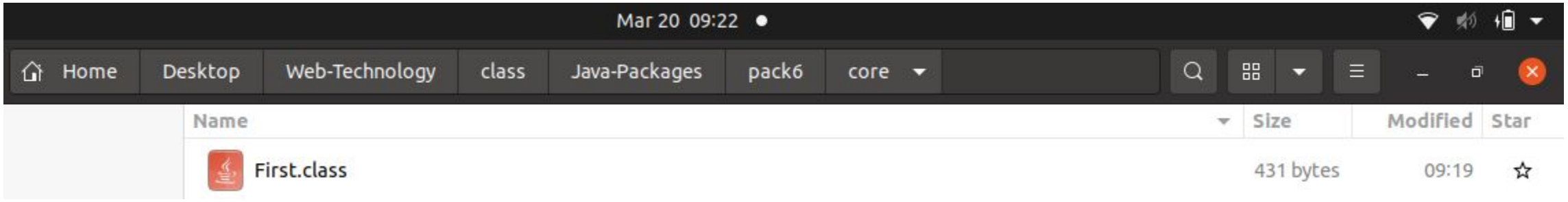


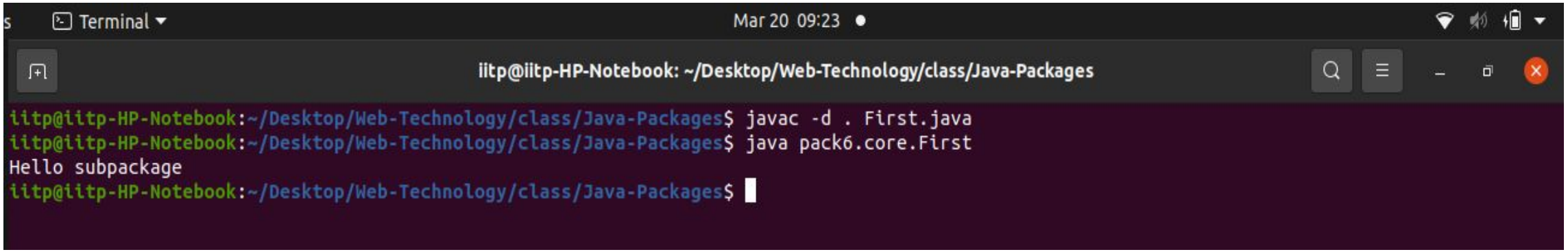




Subpackage

- Package inside the package is called the **subpackage**. It should be created to **categorize the package further.**





How to put two public classes in a package?

- If we want to put two public classes in a package, then have two java source files containing one public class, **but** keep the package name same. For example:

package pack7;
public class G
{
 	
}
	package pack7;
	public class H
	{
	
	}

References

1. https://www.javatpoint.com/package
2.
3.