Xebia

OOPS

OOPS CONCEPT

- There are 2 famous programming models
    1. pops
    2.oops

1.Procedure oriented programming model

- The main building blocks of this model is function
- Any function can access any data
- Programming is more complex

Object oriented programming model

- The main building blocks of this model is Object
- Programming is more simple
- Concept of encapsulation and data security code reusability
- adding more function is not difficult
- Objects are independent of each other

Difference between oops and pops

oops

- Object are main building blocks
- Modification is easy because Object are independent of each other
- It provide access modifier so more secure
- Ex: c language

pops

- Function are main building block
- Modification is difficult because functions are dependent on each other
- It is not provide any access modifier so less secure
- Ex : java and c++

4 Main Pillars

- Encapsulation:
  - Bundling data and methods into a single unit (class).
  - Restricting direct access to data and providing controlled access through methods.
- Inheritance:
  - Mechanism for creating a new class (subclass) by inheriting properties and behaviors from an existing class (superclass).
  - Promoting code reuse and establishing a hierarchical relationship.
- Polymorphism:
  - Ability to use objects of different classes through a common interface.
  - Allows methods to have multiple implementations based on the context.
- Abstraction:
  - Simplifying complex systems by modeling essential properties and behaviors.
  - Focusing on "what" an object does rather than "how" it does it.
  - Creating abstract classes and interfaces to define contracts for concrete classes.

Inheritance

- Part of Object-Oriented Programming (OOP).
- Subclass inherits attributes and methods from a superclass.
- Promotes code reuse and hierarchical relationships.
- extends keyword indicates inheritance.
- Subclass can override inherited methods with @Override.
- Java supports single inheritance for classes but multiple inheritance through interfaces.
- Access modifiers control visibility of inherited members.
- super keyword is used to call superclass's constructor or access its members.
- Forms an "is-a" relationship, building class hierarchies.
- Enhances code organization, abstraction, and extensibility.

Types of Inheritance

Single Inheritance:

Subclass inherits from only one superclass.

Multiple Inheritance (Through Interfaces):

A class can inherit from multiple interfaces.

Multilevel Inheritance:

Subclass inherits from another subclass, forming a chain.

Hierarchical Inheritance:

Multiple subclasses inherit from a single superclass.

Hybrid Inheritance (Combination):

Combines multiple types of inheritance within a single program.

1.single Inheritance

- This is simple one inheriting property from one class to other class

Demo :

```
class A{
        int i=10;
        }
class B extends A{
        System.out.println(i);
        }
```

2.Multilevel Inheritance

- One super class for each sub class

Demo :

```
class A{
        int i=10;
        }
    class B extends A{
        System.out.println(i);
        }
    class C extends B{}
```

3.Multiple Inheritance

- Multiple super class for each single subclass

Demo :

```
class A{
        int i=10;
        }
    class B extends A{
        System.out.println(i);
        }
    class C extends A ,B{}
```

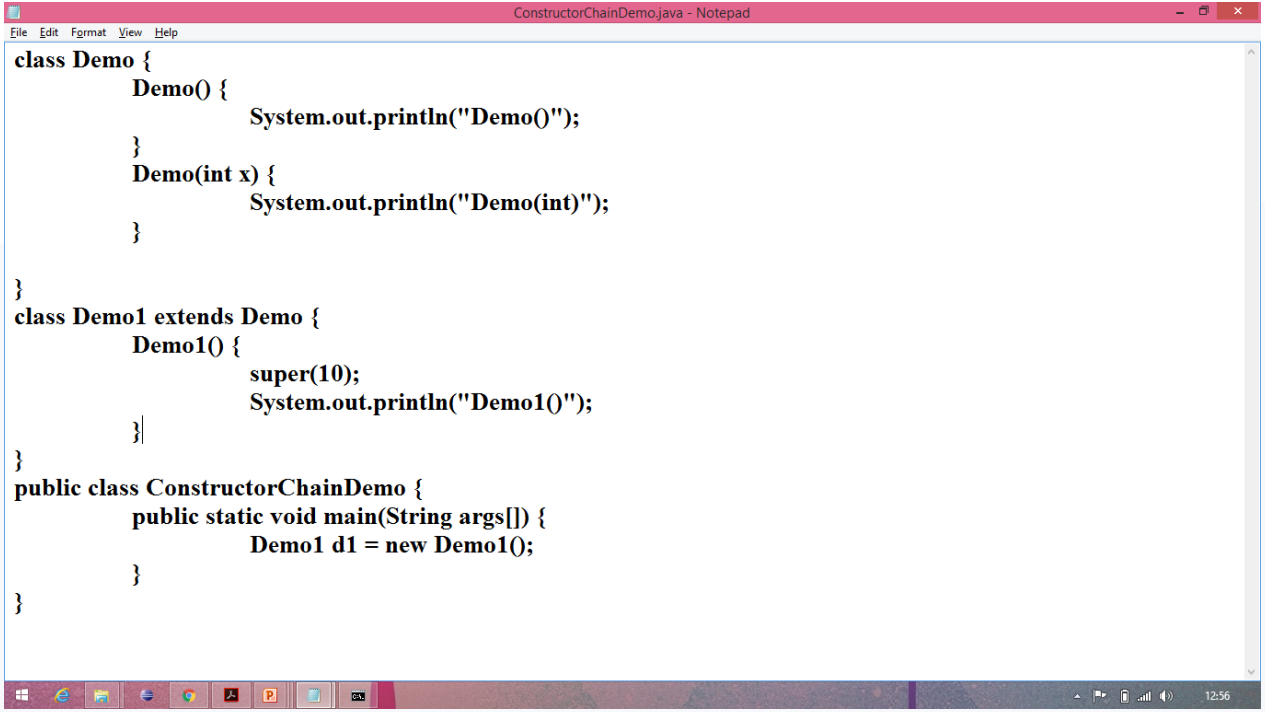
4.Hierarchal Inheritance

- Multiple subclass for super class

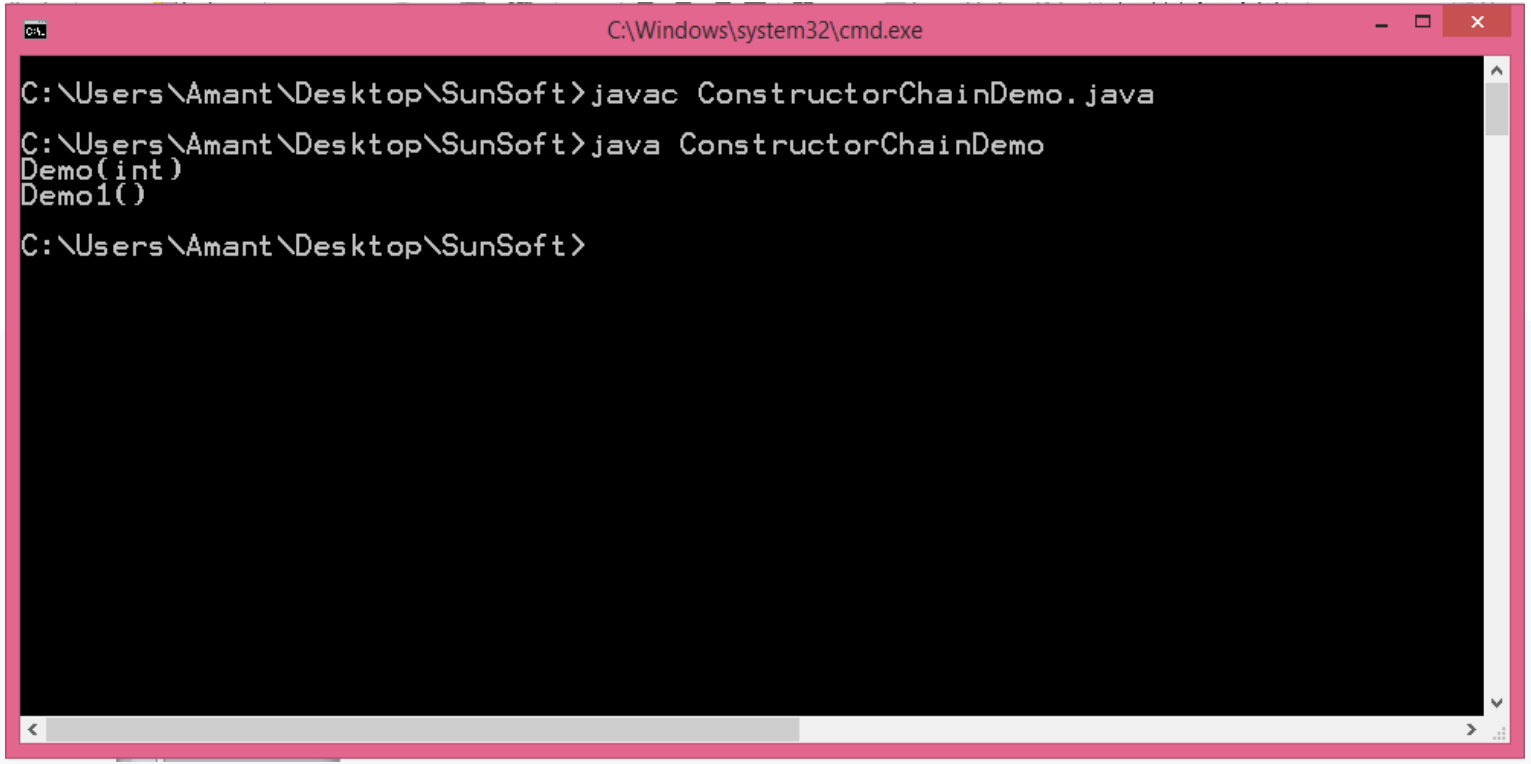
Demo :

```
class A{
                int i=10;
        }
class B extends A
        {
                System.out.println(i);
        }
class C extends A {}
```

Constructor chaining in inheritance:



Constructor chaining in inheritance (contd.):



Polymorphism

- Allows objects of different classes to be treated as objects of a common superclass.
- Enables flexibility in method calls with multiple implementations.
- Achieved through method overriding and interfaces.
- @Override annotation indicates method overriding.
- Promotes generic and flexible code.
- Enhances code extensibility and adaptability.
- Used with interfaces to create a common contract.
- Utilizes inheritance and interfaces in Java.
- Basis for dynamic method dispatch at runtime.

Type of Polymorphism

- **Compile-time Polymorphism (Static Binding or Early Binding):**
  - Determined at compile time.
  - Method or operation decided based on method signatures and argument types.
  - Examples include method overloading and operator overloading.
- **Runtime Polymorphism (Dynamic Binding or Late Binding):**
  - Determined at runtime.
  - Method execution based on the actual object's type.
  - Achieved through method overriding in inheritance and interfaces.
  - Provides flexibility and adaptability in method calls.

Method Overloading & Overriding

Method Overloading:

- Compile-time polymorphism (static polymorphism).
- Multiple methods with the same name but different parameters.
- Parameters must differ in number, type, or order.
- Return type alone does not differentiate overloaded methods.
- Enhances code readability and provides multiple method signatures.

Method Overriding:

- Runtime polymorphism (dynamic polymorphism).
- Occurs when a subclass provides its own implementation for a superclass's method.
- Overriding method must have the same name, return type, and parameter list.
- Indicates specialization or customization of inherited behavior.
- Achieved using @Override annotation in Java.

Method Overriding

- Overriding is the process of writing same method name with same signature in subclass

Demo :

```
class A
{
        void add(){}
}
class B extends A
{
        void add(){}
}
```

Method Overloading

- Overloading is the process of writing same method name with different parameter in same class

Demo ;

```
class A{
        void add(){}
        void add(int a){}
        }
```

Abstraction

- Part of Object-Oriented Programming (OOP).
- Simplifies complex reality by modeling classes based on essential properties and behaviors.
- Focuses on "what" an object does rather than "how" it does it.
- Achieved through abstract classes and interfaces.
- Abstract classes cannot be instantiated and may contain abstract methods.
- Interfaces define a contract for classes to implement.
- Promotes code reusability, maintainability, and higher levels of abstraction.
- Useful for designing frameworks and APIs.
- Encourages modular design and separation of concerns.

Abstraction Demo

```
class A{
        abstract void add();
        }
class B extends A{
        void add(){}
        }
public class M{
        public static void main(String[] args){
//A a=new A();
B b=new B();
        }
}
```

Abstract Class and Method

Abstract Class:

- Cannot be instantiated.
- Serves as a blueprint for other classes.
- Contains a mix of abstract and concrete methods.
- Declared using the abstract keyword.
- Subclasses must implement all abstract methods.
- Can have constructors, fields, and non-abstract methods.

Abstract Method:

- Declared without an implementation (no method body).
- Defined in an abstract class.
- Intended to be implemented by concrete subclasses.
- Declared using the abstract keyword.
- Forms a contract for concrete subclasses to implement.
- No method body, just method signature followed by a semicolon.

Interfaces

- Interfaces define a contract of methods that implementing classes must adhere to.
- An interface contains method signatures without method bodies.
- Declared using the interface keyword.
- A class can implement multiple interfaces.
- Implemented using the implements keyword.
- All methods declared in an interface must be implemented in the implementing class.
- Promote code flexibility and modularity.
- Used to create APIs and ensure certain functionalities are provided.
- Facilitate multiple inheritance in Java.
- Examples include Serializable, Comparable, and Runnable.

Demo for Interface

```
interface I{
        public static final Int i=10;
                public abstract void add();
                }
class A implements I{
        public void add(){}
        }
```

- interface can not have constructer
- interface can extends another interface
- class can implements interface

Marker Interface

- The interface which does not contain any thing or empty
- it is just instruction to java

  ex : cloneable interface , serializable interface

Non access modifier static abstract final

1. **static**

- static is key word applicable for variable method
- static member will load in memory when class is loading in to the memory
- static member can be access with class name without creating an Object of the class
- instance variable can not be declare in static member

Static variable

- static variables are also called as ‘Class variables’.
- static variables are declared inside a class with ‘static’ keyword, but outside any block like methods or constructor.
- static variables have only one copy throughout the class regardless of number of object created.
- static variables are stored in static memory.
- It is mostly used to declare constants.
- Default values of static variables are same as instance variables.
- static variables can be invoked by calling the class name.
  - className.variable_name

Differences between abstract class and interface

abstract class

- This can have abstract as well as normal method
- variable can be any instance, static final
- Constructor are present in abstract classes
- abstract class does not support multiple inheritance
- 100% abstraction is not possible

interface

- This will have only abstract method
- variable are static final only
- Constructor are absent in interface
- interface support multiple inheritance
- 100% abstraction can be achieved

Packages in Java

- **Definition:**
- **A package is a collection of classes, interfaces, sub packages and annotations.**

- **Uses:**
- Code reusability.
- To avoid naming collision i.e. to maintain more than one class with same but different functions.
- **Root package:**
- J2SE java
- J2EE javax

Packages in Java(Contd.)

create userdefined package:

1. To create a userdefined package we must write “package package_name” in our code.
2. Then during compilation we must use “-d . “.

Syntax:

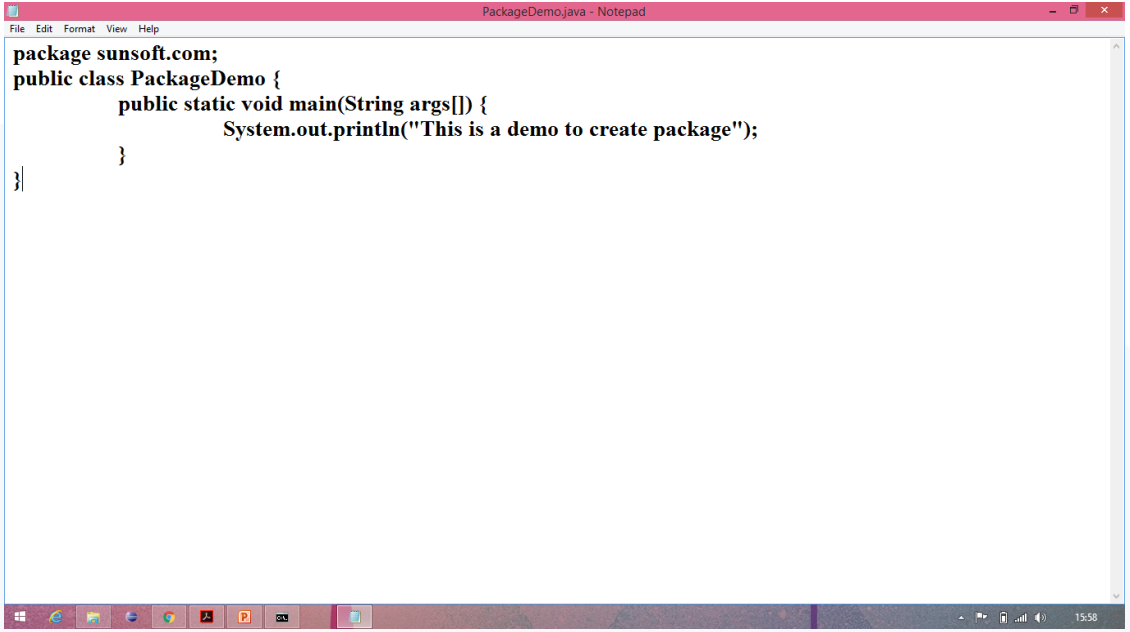
javac –d . FileName

Note: While creating a package we must follow the order given below.

package **import** **class**

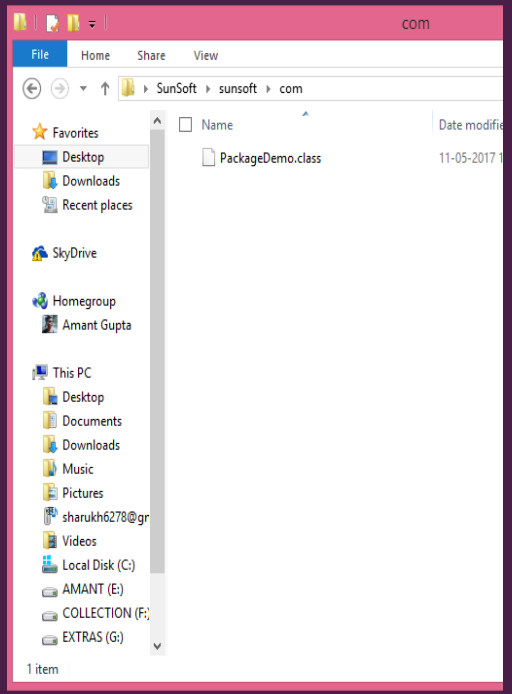
Packages in Java (contd.)

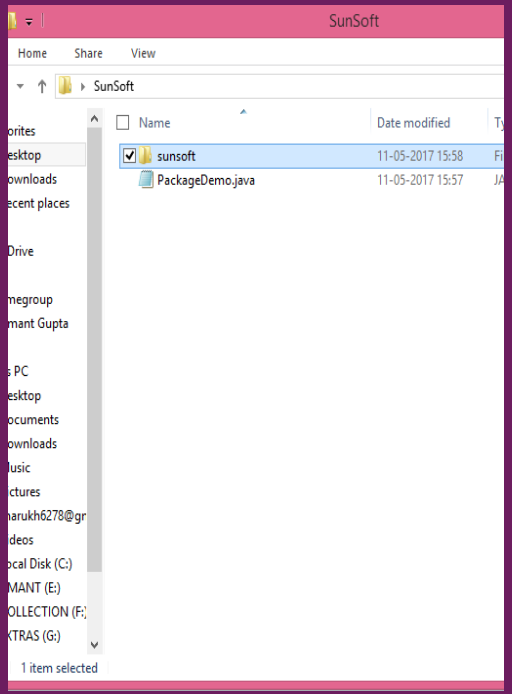
- create userdefined package demo:

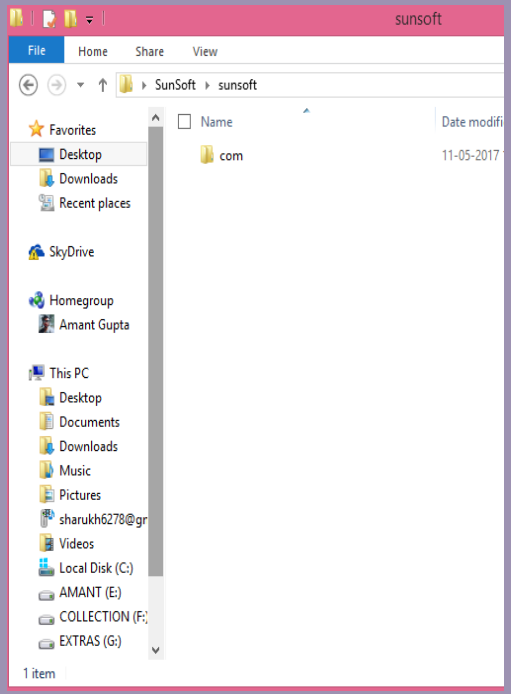


Packages in Java (contd.)

create userdefined package demo:

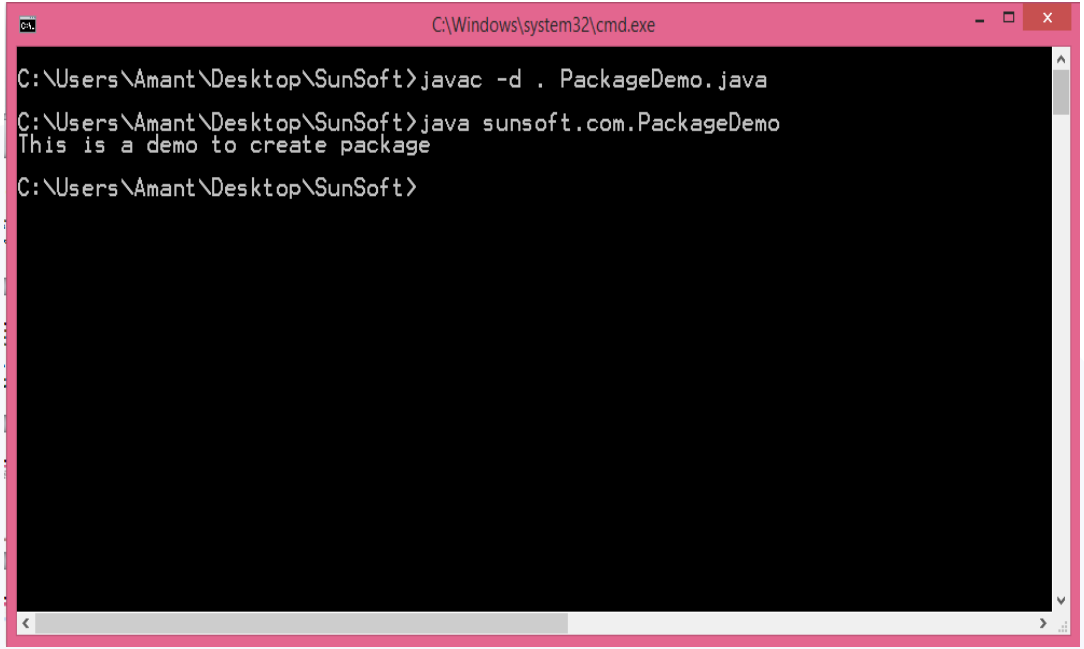






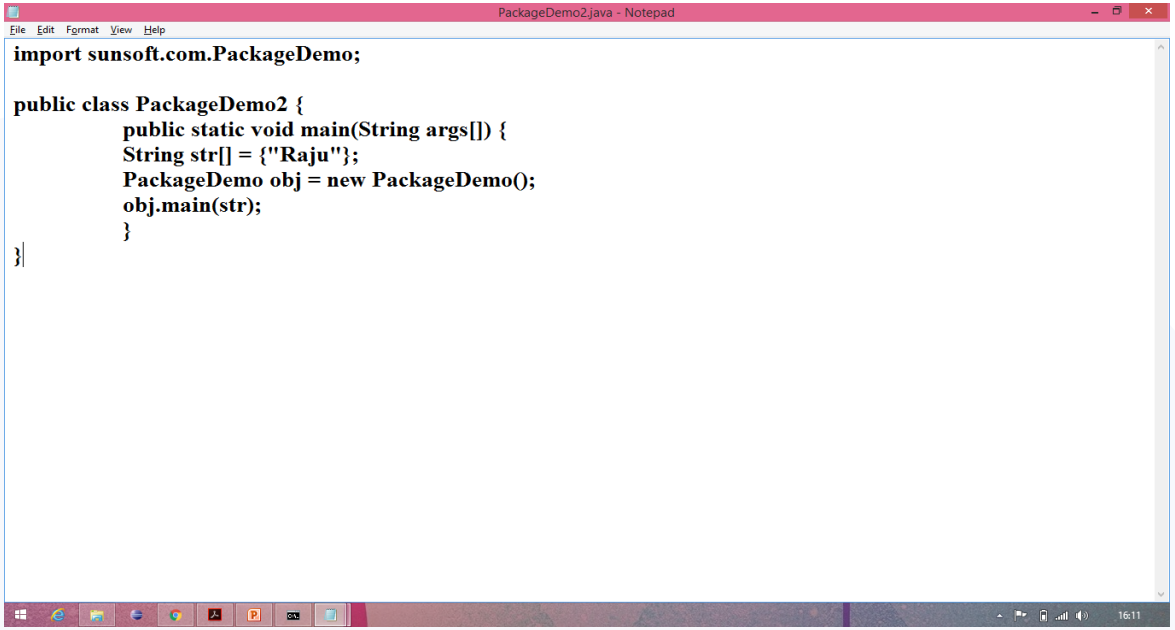
Packages in Java (contd.)

- **create userdefined package demo:**



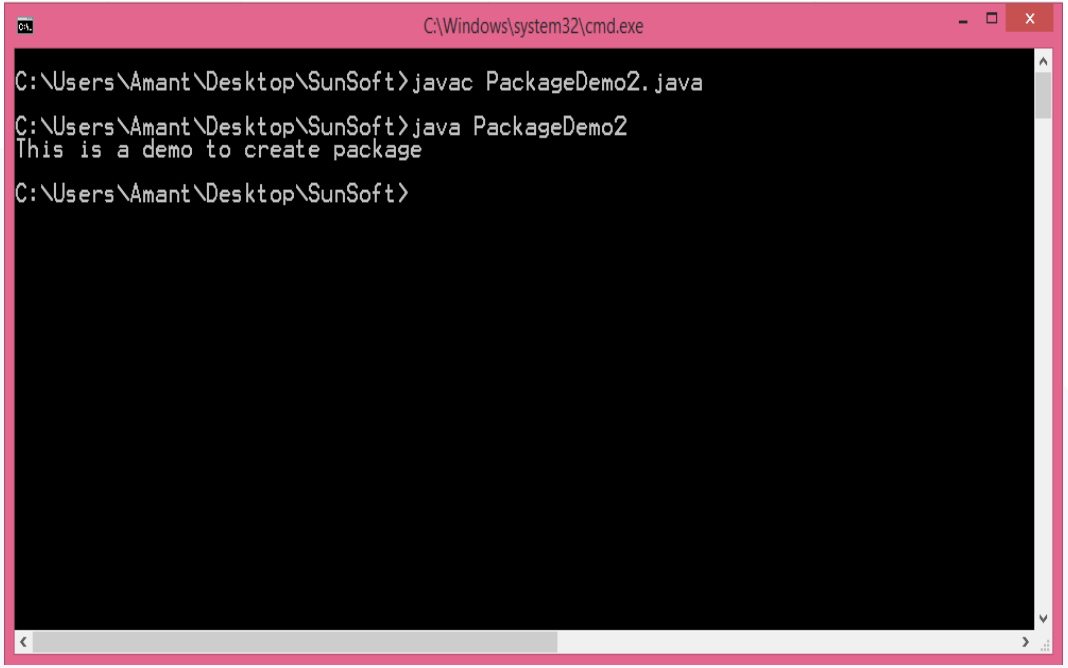
Packages in Java (contd.)

- **Importing the userdefined packages:**



Packages in Java (contd.)

- **Importing the userdefined packages:**



Encapsulation

- Encapsulation is an Object-Oriented Programming (OOP) concept that involves bundling data (attributes or fields) and methods (functions) into a single unit called a class, with the purpose of enhancing security and maintainability by controlling data access through methods.

xebia.com