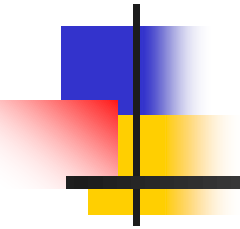
Interfaces - Abstract class - Packages



Object Oriented Programming with Java

Chapter 4

FPTU Da Nang - IT Department

Fpt University

Content

- Abstract method and abstract class
- Design & implementation of interface
- Interface concept Package concept
- Package definition and usage
- Common modifier keywords

abstract class–abstract method

- abstract method is a method that does NOT define a body
  - public abstract void doSomething();
- abstract class is declared using the keyword *abstract and* contains abstract methods
  - public abstract class Menu {..
    public abstract void execute(int n);
    }
- Important note
  - Can NOT create the instance of abstract class using **new** if NO overriding the abstract method is supplied
  - Abstract class along with interface and sub-class are 3 levels of inheritance in Java

Interface

- Interface is a blueprint – template for a class, for new reference type.
  - unified programming interface
  - support team-work
  - allow multiple inheritance in Java
- Syntax :

```
public interface InterfaceName {
    // static final
    // abstract method;
    // default method;
}
```

  - static final : sharing constants for all classes of an application.
  - abstract method: method with no body, will be defined by the classes that implement the interface
  - default function declared with the *default* keyword at the beginning of the method signature
  - All member of interface are public by default

Interface..

```
public interface Shape {
        double Pi=3.1416;
        public double perimeter();
        public double area();
        public default void display(){
                System.out.println("This is a shape");
        }
}
```

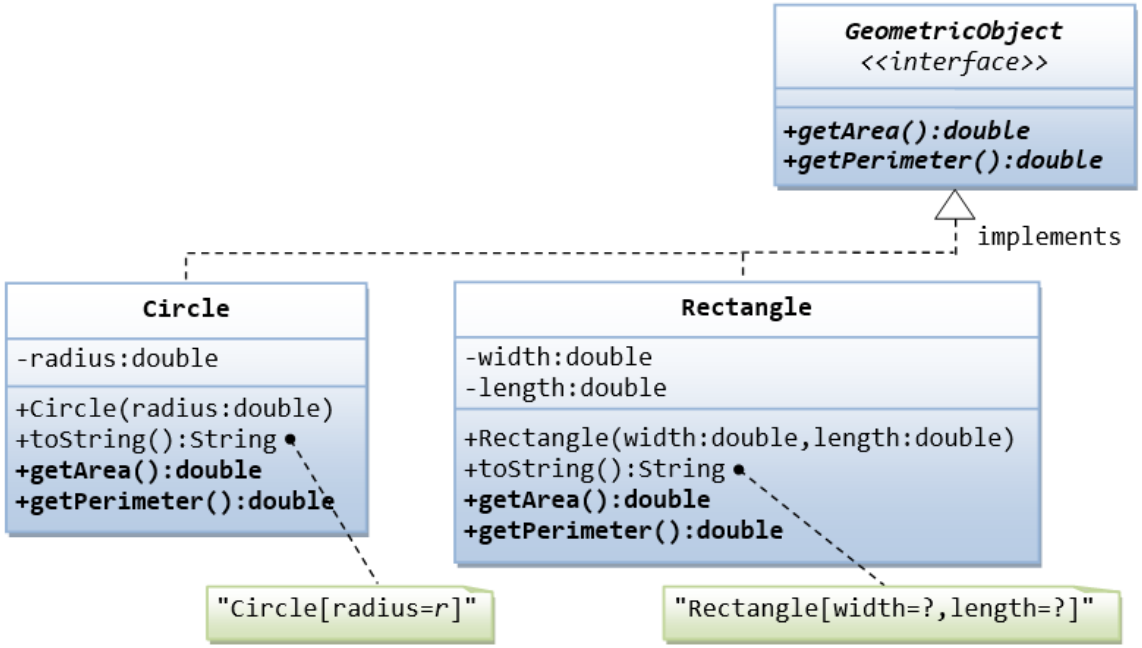
- Important note when design and implement an interface
  - An interface can inherit one interface or multiple interfaces
    - public interface MouseAction extends MouseListener, MouseMotionListener
  - All members of interface are default accessed by public
  - The variable declarations in the interface are static and final – so they must be initialized
  - Final methods are not allowed

Implementation class

- Syntax
  - Using keywords **implements**
  - public class Rectangle **implements** Shape {
    private double length, width;
    public double area() { return length*width; };
    public double perimeter() { return 2*(length+width);}
    }
- Important notes:
  - implementing class must define all abstract methods of interface.
  - A class can inherit only 1 base class **but** can implement many interfaces: multiple inheritance
    public class Student extends Person implements Comparable, Serializable{ …}
  - interface type variable can take the instance of the corresponding implementing class
    - Shape cn= new Rectangle();

Interface Demo

- Write an interface called GeometricObject, which contains 2 abstract methods: getArea() and getPerimeter(), as shown in the class diagram. Also write an implementation class called Circle. Mark all the overridden methods with annotation @Override.



Comparable interface

- Comparable is an interface for comparing two objects of a class by overriding the method compareTo()
  - public int compareTo(Object other); define the logical comparision between 2 objects(this and other), return:
    + positive number if this>other
    + ==0 if this=other
    + negative number if this<other
  - public class Student implements Comparable<Student>{

    private String id, name;
    public int compareTo(Student s){
    return this.name.compareTo(s.name); }
  - Arrays.sort(m); sort array m in ascending order with array elements comparable

Functional interface – lambda expr.

- Functional interface is an interface with only one abstract method for doing something..
  - Comparable, Comparator, Predicate, Runnable
- **Predicate** interface declares **test()** method to check whether an object satisfies a certain condition or not.

  + public boolean test(Object a);
- **Predicate** interface: effectively used in searching by different criteria without writing multiple search functions -> by using lambda expressions, no need to code implementation class.

Lambda expression

- Lambda expressions are similar to methods; lists the arguments (left side) and the body of the method (right side) that handles those arguments
  - (lambdaParameters) ->{ lambdaBody}
    (int x, int y) -> x+y
- For example, this search function when called will be passed a lambda expression containing the search criteria
  - public ArrayList search(Predicate<Student> p) {
    if(p.test(element)) addElementToSearchResult();
    }
- Search by name starting with S and find birthday before X
  - search(Student s -> s.getName().startWith("S"))
    search(Student s -> s.getDob().before(X))

Fpt University

Lambda expression demo

```
public class Contact {

    private String name;
    private String email;
    private String phone;

    public Contact() {
    }

    public Contact(String name, String email, Strin
        this.name = name;
        this.email = email;
        this.phone = phone;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getPhone() {
        return phone;
    }

    public void setPhone(String phone) {
        this.phone = phone;
    }

    @Override
    public String toString() {
        return "Contact{" + "name=" + name + ", email=" + email +
    }
}
```

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

/**
 *
 * @author Ly Quynh Tran
 */
//public interface Predicate<T>
//{
//    boolean test(T t);
//}
public class ContactList {

    private List<Contact> contacts = new ArrayList<>();

    public ContactList() {
        contacts.add(new Contact("Mr A", "a@y.c", "555-1212"));
        contacts.add(new Contact("Mr B", null, null));
        contacts.add(new Contact("Mr C", "c@y.c", null));
    }

    public List<Contact> filterContact(Predicate<Contact> condition) {
        List<Contact> filteredContacts = new ArrayList<>();
        for (Contact contact : contacts) {
            if (condition.test(contact)) {
                filteredContacts.add(contact);
            }
        }
        return filteredContacts;
    }
}
```

```
public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        ContactList list = new ContactList();
        List<Contact> contactsWithoutPhone = list.filterContact(contact-> contact.getPhone()==null);
        System.out.println(""+contactsWithoutPhone);
        List<Contact> contactsWithoutEmail = list.filterContact(contact-> contact.getEmail()==null);
        System.out.println(""+contactsWithoutEmail);
    }

}
```

```
run:
[Contact{name=Mr B, email=null, phone=null}, Contact{name=Mr C, email=c@y.c, phone=null}]
[Contact{name=Mr B, email=null, phone=null}]
BUILD SUCCESSFUL (total time: 0 seconds)
```

Packages

- Package in Java is a folder contain a group of classes, sub packages and interfaces. Packages are used for:
  - Extend class naming space, prevent naming conflicts.
    - For example there can be two classes with name Employee in two packages, college.staff.cse.Employee and college.staff.ee.Employee
    - Full class name including package name java.io.Reader
    - Undeclared package: unnamed package – current directory

Packages

- Defining the scope of access
  - Classes in the same package can access each other without declaring access scope
- Using keyword **import** for referencing classes from other packages.
  - When not declaring **import,** specify the package name before the class name
    java.util.Date d= new java.util.Date();

Organize Packages

- To organize the package, we make the classes become the members of package
  - First, declare package before creating the class

```
package mypackage;
    public class Hello {}
```

  - Then, all the classes created later will store in the folder named package
  - Package names should start with a lowercase letter to distinguish them from class names
- Compile packages
  - The -d parameter.
  - javac – d C:\Temp Hello.java
    - Will automatically create mypackage folder in C:\temp and save the Hello.class code in that folder

Using Packages

- Using import to specify the package that refers to the classes
  - import 1 class of the package :
    - import java.io.Reader;
  - import all class of the package :
    - import java.io.*;
- Components such as class, interface declared **public** will be accessible from outside the package
- java.lang is the default import package for all class

Keyword Modifiers

- Modifiers are keywords that modify declarations

access modifiers	other modifiers
public	**final**
protected	**abstract**
private	**transient**
	native
	synchronized
	static

static

Static

- Used with variables, functions, and blocks indicating shared components in a class
- Static function are not allowed to use 'this'
- Static functions are not allowed to be overriden by non-static functions
- The code placed in the static {code..} block will be executed first when the Java program is run.

final

final

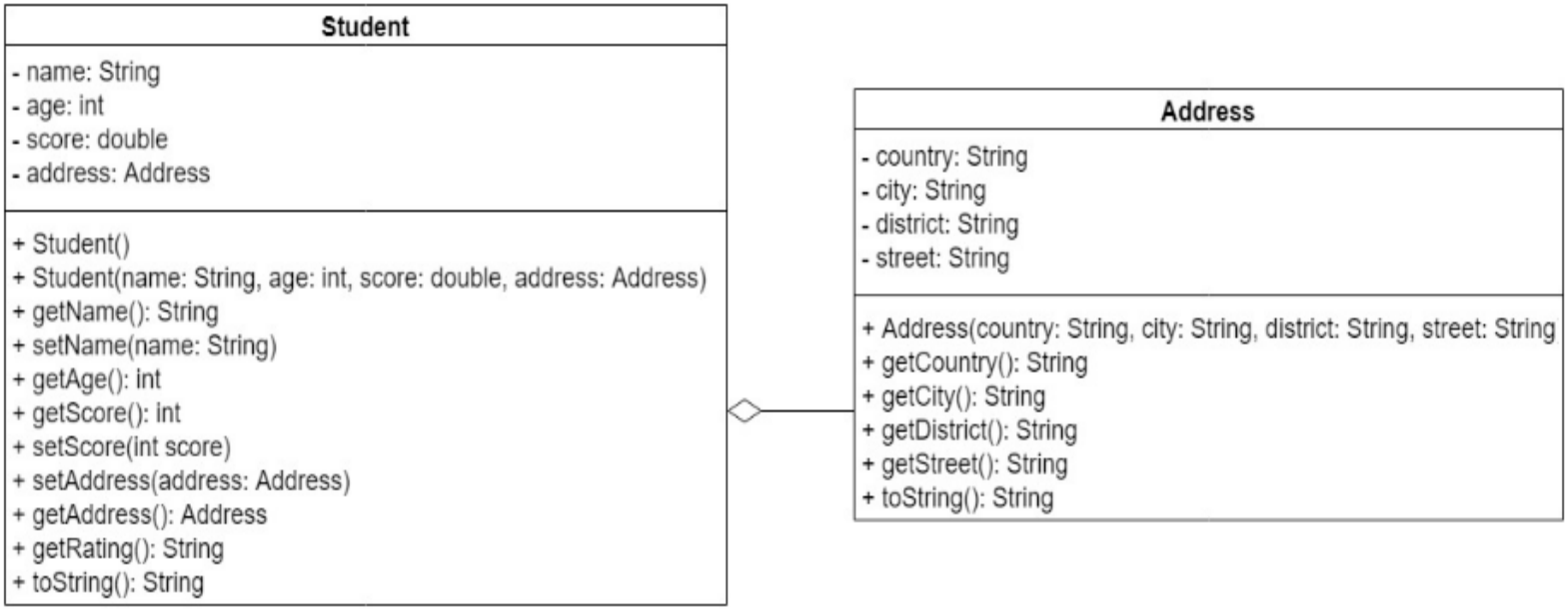
- Apply to class, function, variable
- final class does not allow inheritance
- final variable must be initialized when declared
- final method – not allow overriding
- final objects – the reference to the object is not allowed to change, but the value reference is mutable
- java.lang.Math is the final class

abstract

- abstract
  - Apply to classes and methods
  - Abstract class is an incomplete class
    - There are abstract methods but not defined
    - public abstract int calculate();
  - cannot initialize variables of abstract class using new operator
    - Can be initialized with instance of anonymous class (provide ad-hoc override abstract method)
  - Generates incomplete executable versions & allows partial customization of the class

Fpt University

Exercises



Lab – homework

- Define the interface Menu for user choice..
- Write abstract class TextMenu which implements Menu,
  - override all methods – except execute(..)
  - execute() doing something base on user choice on menu.
- Using TextMenu for Student Management Application
  - Provides tasks for student management: add, update, delete, search..

Constructive Questions

- Give an example of an interface already defined in the JDK library. Explain the use and meaning of that interface
- Why we need interface?
- Define an interface for an electric power consumption management application
- Find the scenarios that need abstract function and abstract class
- Compare inheritance characteristics of sub-class, abstract class and interface.
- Define an interface, then write the abstract class implementing the interface and write a class that extend the abstract class. Choose the free topic

Constructive Questions

- What are the characteristics of variables declared in the interface? Explain why these characteristics are default?
- What is the purpose of package?
- Compare functional interfaces, tag interfaces and interfaces
- How can Arrays.sort(arr) and Collections.sort(list) sort the array arr/ an arbitrary list list?
- Compare the two terms data abstraction and abstract class on the semantics of the word abstract
- Give an example using synchronized keyword and explain its meaning.
- Give an example using transient keyword and explain its meaning.