GIA DINH
UNIVERSITY

Chapter 3

P.I.E triangle

Objectives

- *Polymorphism*
- *Inheritance*
- *Encapsulation*
- *Abstract*

Inheritance

Inheritance

- In family, the properties of parents are often inherited by their children.
- Inheritance is an important and integral part of object-oriented programming.
- Inheritance enables you to define a very general class first and then define more specialized classes later
- Superclass, base class, parent class ≠ subclass, derived class, child class

Creating subclass

- Syntax:

```
class subclass_name extends superclass_name {
        …
    }
```

- Ex: class SinhVien extends ConNguoi{

```
        ….
    }
```

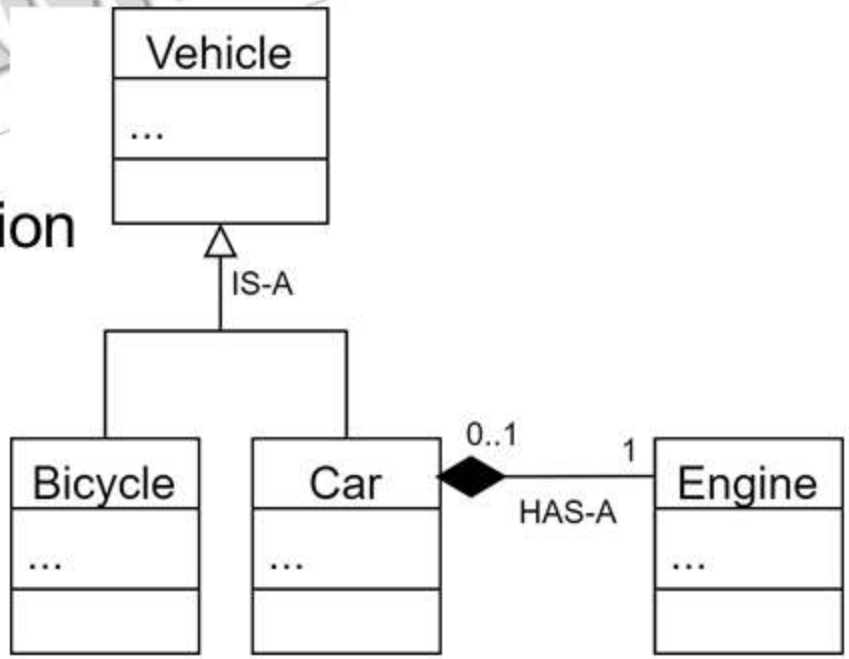
IS-A vs HAS-A Relationship

- IS-A relationship: Generalization *(inheritance)*
  - "A IS-A B type of thing"
  - Ex: Car IS-A Vehicle (but Vehicle is not a Car)

```
class Car extends Vehicle { //Inheritance
        …
}
```

- HAS-A relationship: Composition
  - "A HAS-A B field of a type"
  - Ex: Car HAS-A Engine

```
class Engine {
        …
}
```



Method overriding

- Method overriding: When a subclass defines a new method having the same signature as the method in the superclass

Ex:

```
public class Calculation{
        ...
        int sum(){
                return 5;
        }
}
```

```
public class Arithmetic extends Calculation{
        ...
        int sum(){
                return 10;
        }
}
```

Method overloading

- Several methods, having the same name but different parameters
- Ex:

```
public class OverloadClass{
        int sum(){
                return 4;
        }

        float sum(int a,int b){
                return 7;
        }
}
```

no parameter

2 parameters

Keyword: new, this, super

- new: Create a new object
  - Syntax: ClassName objectVariable = new ClassName();
  - Ex:
    - Calculation t1=new Calculation();
    - Calculation t2=new Arithmetic();
    - Arithmetic c=new Calculation(); //error, why?
- this: to refer to the current object in memory
  - Syntax: this.variable *or*
    this(var*1*,var2,…,var*n*);
  - Ex: this.a=a; // recommend
    this(a,b,c); // rarely
- super: to refer to the object of superclass
  - Syntax: super.variable *or*
    super(var*1*,var2,…,var*n*);
  - Ex: super.a=a; // rarely
    super(a,b,c); // recommend

```
public class Calculation{
   ...
   int sum(){
            return 5;
   }
}
```

```
public class Arithmetic extends Calculation{
   ...
   int sum(){
            return 10;
   }
}
```

Inner class

- Inner class: nest classes (a class within a class).
- Purpose: group classes → code more readable and maintainable
- Ex:

```
class OuterClass {
  short a = 1;

  class InnerClass {
    short b = 2;
  }
}
```

```
public class Main {
  public static void main(String[] args) {
    OuterClass outerClass = new OuterClass();
    OuterClass.InnerClass innerClass =
outerClass.new InnerClass();
    System.out.println(outerClass.x + innerClass.y);
  }
}
```

- ***Note:*** *an inner class can be* ***private*** *or* ***protected***

Polymorphism

Polymorphism

- Polymorphism means “many forms”
- Polymorphism uses those methods to perform different tasks

```
class Animal {
  public void animalSound() {
    System.out.println("The animal makes a sound");
  }
}
```

```
class Pig extends Animal {
  public void animalSound() {
    System.out.print("The pig says: ột ột");
  }
}
```

```
class Rooster extends Animal {
  public void animalSound() {
    System.out.print("The rooster says: ò ó o");
  }
}
```

```
class Main {
  public static void main(String[] args) {
    Animal myAnimal = new Animal();
    Animal myPig = new Pig();
    Animal myRooster = new Rooster();
    myAnimal.animalSound();
    myPig.animalSound();
    myRooster.animalSound();
  }
}
```

Constructor order

```
public class PhuongTienGiaoThong {
  public PhuongTienGiaoThong() {
    System.out.println("Đây là
PhuongTienGiaoThong");
  }
}
```

```
public class
PhuongTienGiaoThongDuongBo
extends PhuongTienGiaoThong {
    PhuongTienGiaoThongDuongBo() {
    System.out.println("Đây là
PhuongTienGiaoThongDuongBo");
    }
}
```

```
public class XeHoi extends
PhuongTienGiaoThongDuongBo {
   XeHoi(){
   System.out.println("Đây là XeHoi");
   }
}
```

```
public class Test{
   public static void main(String[] args) {
 System.out.println("Phương thức main: ");
   XeHoi xh=new XeHoi();
    System.out.println("Kết thúc kiểm tra lớp
XeHoi");
    }
}
```

Packages and access modifiers

Packages

- Package: group of classes and interfaces
  - Contents more sub packages or zero
  - Fully qualified name:

    packagename.subpackagename.class

    Ex: hospital.doctor

  → Directory: hospital\doctor
  - Syntax creating: package <packagename>

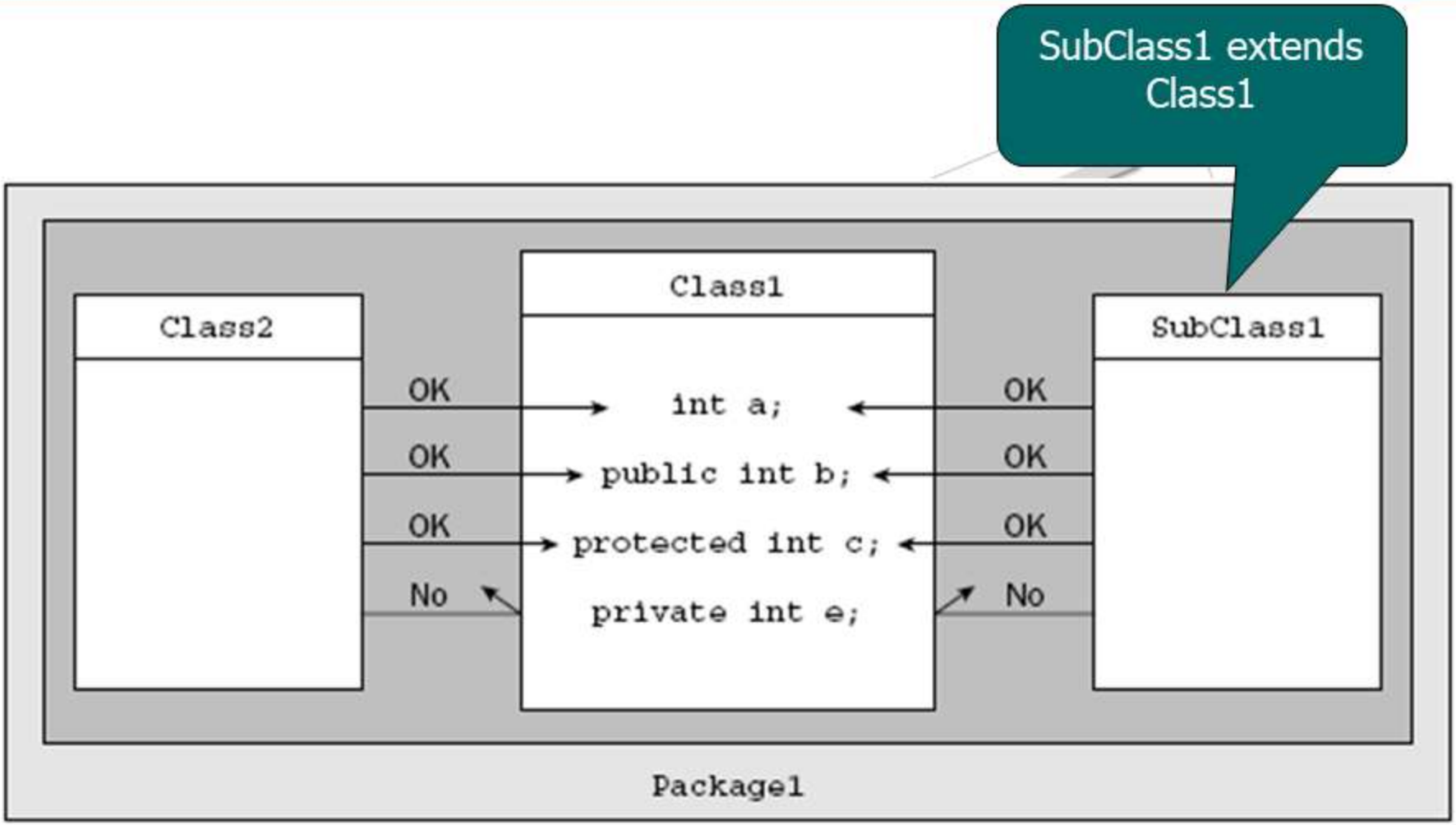
    Ex: package hospital
  - Syntax using: import packagename.class;

    Ex: import hospital;

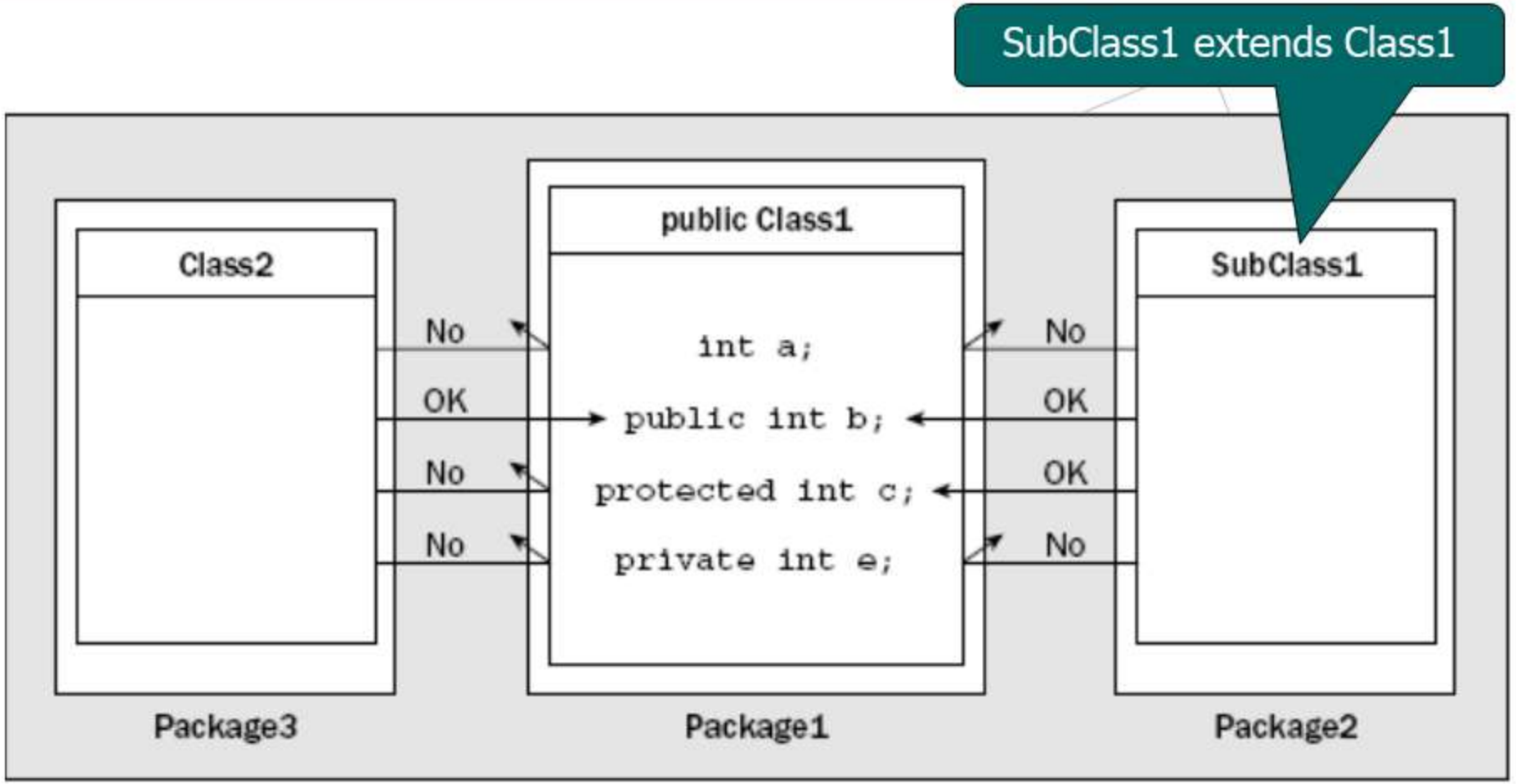
Access modifiers

- Access modifier = Access specifier: controls the access of class and class member
  - public: allows the class to be accessible everywhere
  - protected: accessible only within its own class, package and inheriting classes
  - default: no specific keyword, accessed by any other class in the same package
  - private: accessible only within its own class

Scope


SubClass1 extends Class1
Class1
Class2
SubClass1
OK
OK
OK
No
int a;
public int b;
protected int c;
private int e;
OK
OK
OK
No
Package1

Scope


SubClass1 extends Class1
Class2
public Class1
SubClass1
No
OK
No
No
int a;
public int b;
protected int c;
private int e;
No
OK
OK
No
Package3
Package1
Package2

Scope

Access modifier	Elements visible in			
	Outside the package	Package	Subclass	Class
public	✓	✓	✓	✓
protected	×	✓	✓	✓
default	×	✓	×	✓
private	×	×	×	✓

The scope of access modifiers

Association

Access modifier	Can be applied to				
	Data Field	Method	Constructor	Class	Interface
public	✓	✓	✓	✓	✓
protected	✓	✓	✓	×	×
default	✓	✓	✓	✓	✓
private	✓	✓	✓	×	×

Relationship between access modifiers and elements

Non-Access Modifiers

- For class:
  - final: Cannot be inherited by other classes
  - abstract: cannot be used to create objects
- For attributes and methods (Ref):
  - final: cannot be overridden/modified
  - static: it can be accessed without creating an object of the class
  - abstract: Can only be used on methods in an abstract class
  - transient: be skipped when serializing the object containing them
  - synchronized: Methods can only be accessed by one thread at a time
  - volatile: The value of an attribute is not cached thread-locally, and is always read from the "main memory"

Keyword: final

- 3 types of final
  - **Final variable:** To define a constant identifier, value can not be modified

    Ex: public final int a=19;
  - **Final method:** to prevent a method from being overridden (hidden) in Java subclass.

    Ex:

```
public final int TinhTong(){
    ....
}
```
  - **Final class:** A class that cannot be subclassed *(to prevent the modification of the class definitions)*

    Ex:

```
public final class TinhToan(){
    ....
}
```

Keyword: abstract

- abstract class: is a restricted class
  - That cannot be used to create objects
  - To access, it must be inherited from another class (extends keyword)
- abstract method: can only be used in an abstract class
  - It does not have a body
  - The body is provided by the subclass (inherited from)

```
abstract class Animal {
  public abstract void animalSound();
  public void sleep() {
  System.out.print("Zzz");
} }
```

```
class Pig extends Animal {
   public void animalSound() {
     System.out.print("The pig says: ột ột");
   }
}
```

```
class Main {
   public static void main(String[] args) {
   Animal duck= new Animal()// Error
   Pig pig= new Pig();
   pig.animalSound();
   pig.sleep();
 } }
```

Encapsulation

Encapsulation

- To make sure that "sensitive" data is hidden from users
- Declare class variables/attributes as private
- Provide **public get** and **set** methods

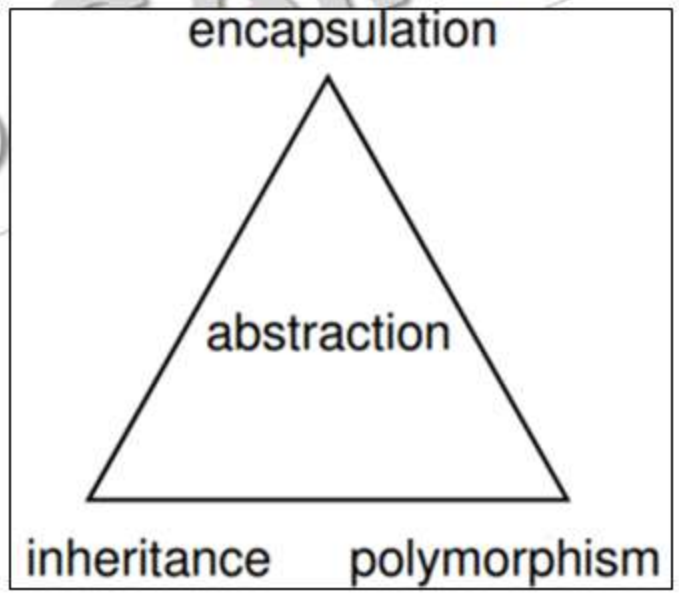
```
public class Person {
   private String name;//cannot access from outside
   public String getName() {
      return name;
   }
   public void setName(String newName) {
      this.name = newName;
 }}
```

```
public class Main {
  public static void main(String[] args) {
    Person p = new Person();
    p.name = "Hoa";  // error
    System.out.println(p.name); // error
    p.setName("Hoa");
    System.out.println(p.getName());
  }}
```

Encapsulation

- Why Encapsulation?
  - Better control of class attributes and methods
  - Class attributes can be made read-only; or write-only
  - Flexible: change one part of the code without affecting other parts
  - Increased security of data

- P.I.E triangle



P.I.E Triangle

Interface

Interface

- Interface
  - is a completely "**abstract class**"
  - must be "implemented" with the **implements** keyword

```
interface Animal {
   public void animalSound();
   public void sleep();
}
```

```
class Pig implements Animal {
   public void animalSound() {
   System.out.println("The pig says: ột ột");
  }
   public void sleep() {
   System.out.println("Zzz");
} }
```

```
class Main {
   public static void main(String[] args) {
      Pig pig = new Pig();
      pig.animalSound();
      pig.sleep();
} }
```

Interface

- Why and When to use Interfaces?
  - To achieve security - hide certain details and only show the important details of an object (interface)
  - Java does not support "multiple inheritance".
  - However, the class can implement "multiple interface" separate them with a comma

```
interface FirstInterface {
  public void myMethod();
}

interface SecondInterface {
  public void myOtherMethod();
}
```

```
class DemoClass implements FirstInterface,
SecondInterface {
  public void myMethod() {
    System.out.println("Some text..");
  }
  public void myOtherMethod() {
    System.out.println("Some other text...");
  }
}
```

Q&A