



OOP CONCEPTS

Feb, 2013

Presented By : JSAG

Agenda

2

- ▣ Object and Class
- ▣ Methods and Attributes
- ▣ Abstraction
- ▣ Encapsulation, information hiding
- ▣ Coupling, Cohesion
- ▣ Inheritance
- ▣ Polymorphism
- ▣ Class diagram



Objects and Classes

What is an Object?

4

- ❑ In some way everything can be an object.
- ❑ In general, an **object** is a person, place, thing, event, or concept.
- ❑ Because different people have different perceptions of the same object, what an object is depends upon the point of view of the observer.
 - ▣ “ ... for there is nothing either good or bad, but thinking makes it so.”
 - ▣ That is, we describe an object on the basis of the features and behaviors that are important or relevant

Types of Objects

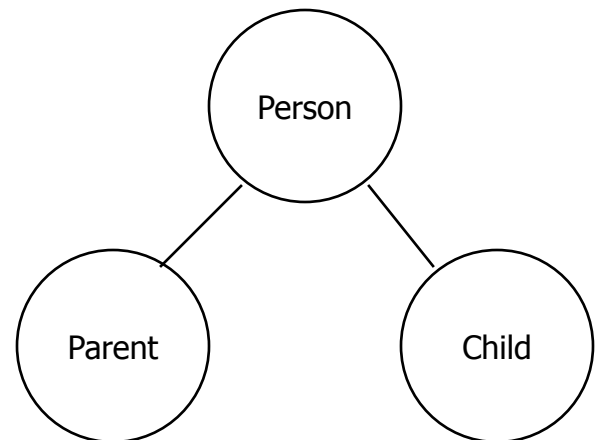
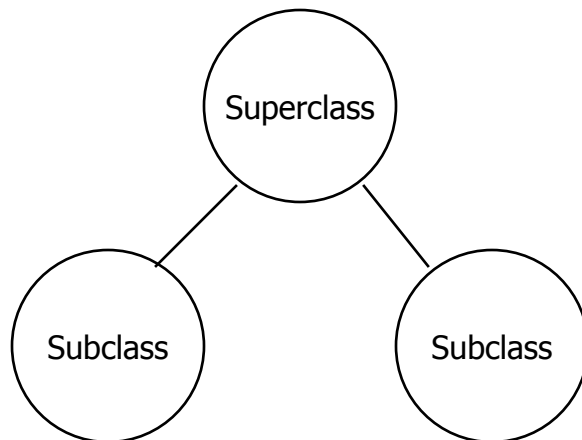
5

- Objects are usually classified as
 - ▣ Objects representing physical things
 - e.g. students, furniture, buildings, classrooms
 - ▣ Objects representing concepts
 - e.g., courses, departments, loan

Class

6

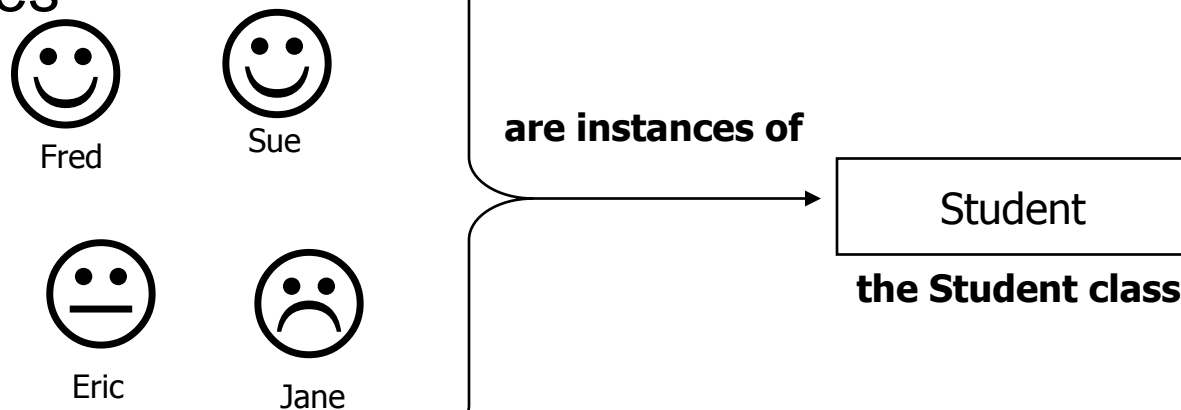
- We classify similar objects as a type of thing in order to categorize them and make inferences about their attributes and behavior.
 - ▣ To a child, mommy, daddy, big brother, and aunt Jane are all classified together as people with relatively similar attributes (head, nose, arms) and behaviors (smile, hug, run, etc).
- For this reason, a type of object is referred to as a **class**.
 - ▣ A general type of thing is a **superclass**, a special type of thing is a **subclass**.



Class and objects

7

- A class is thus a type of thing, and all specific things that fit the general definition of the class are things that belong to the class.
 - ▣ A class is a "blueprint" or description for the objects.
- An object is a specific **instance** of a class.
 - ▣ Objects are thus **instantiated** (created/defined) from classes



Class Members

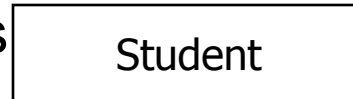
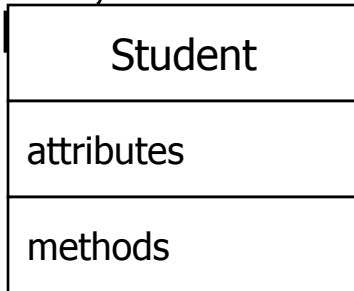
8

- ❑ Attributes/Data
 - ▣ the things classes "know" (nouns)
 - ▣ a piece of data or information
 - ▣ sometimes called **data members** or member variables or fields
- ❑ Methods/Operations/Behaviors
 - ▣ the things classes "do" (verbs)
 - ▣ These define the operations you can perform for the class.
 - ▣ Similar to procedures/functions in other programming languages.

Naming classes

9

- ❑ Class names are typically nouns.
 - ▣ The name of a class generally should be one or two words.
- ❑ Class names are almost always singular (Student, not Students).
 - ▣ In the real world, you would say “I am a student,” not “I am a students”
- ❑ In UML, classes are modeled as a rectangle that lists its attributes and methods.



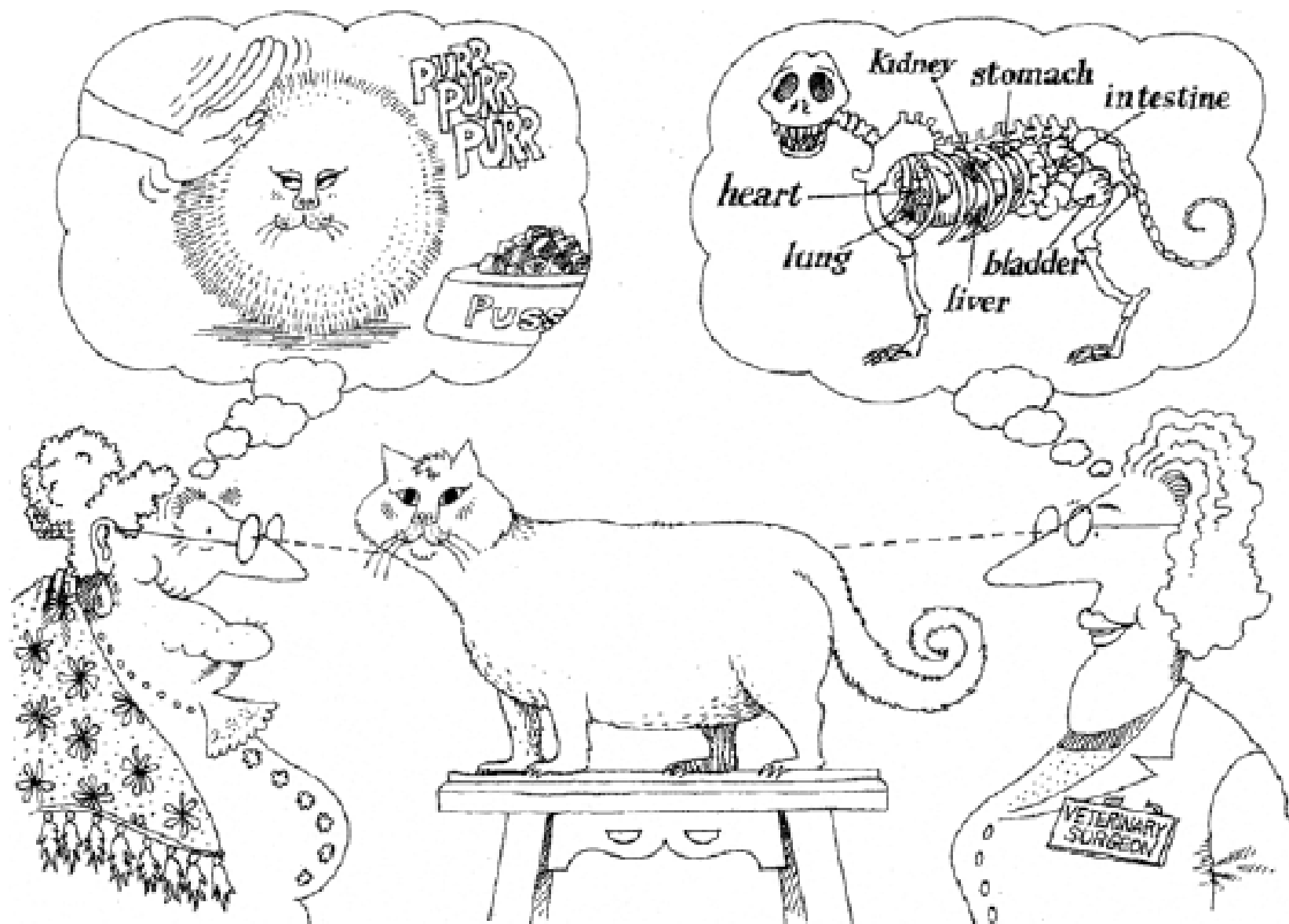
UML (Unified Modeling Language) is a diagramming notation for describing software systems.

Abstraction

Abstraction

11

- An object is thus an **abstraction**.
- An object is an “abstraction of something in the problem domain, reflecting the capabilities of the system to keep information about it, interact with it, or both.” (Coad and Yourdon)



Abstraction

13

- An **abstraction** is a form of representation that includes only what is useful or interesting from a particular viewpoint.
 - ▣ e.g., a map is an abstract representation, since no map shows every detail of the territory it covers

Encapsulation

Encapsulation

15

- Perhaps the most important principle in OO is that of encapsulation (also known as information hiding or implementation hiding).
 - ▣ This is the principle of separating the implementation of an class from its interface and hiding the implementation from its clients.
 - ▣ Thus, someone who uses a software object will have knowledge of what it can do, but will have no knowledge of how it does it.

Encapsulation

16

□ Benefits

▣ Maximizes maintainability

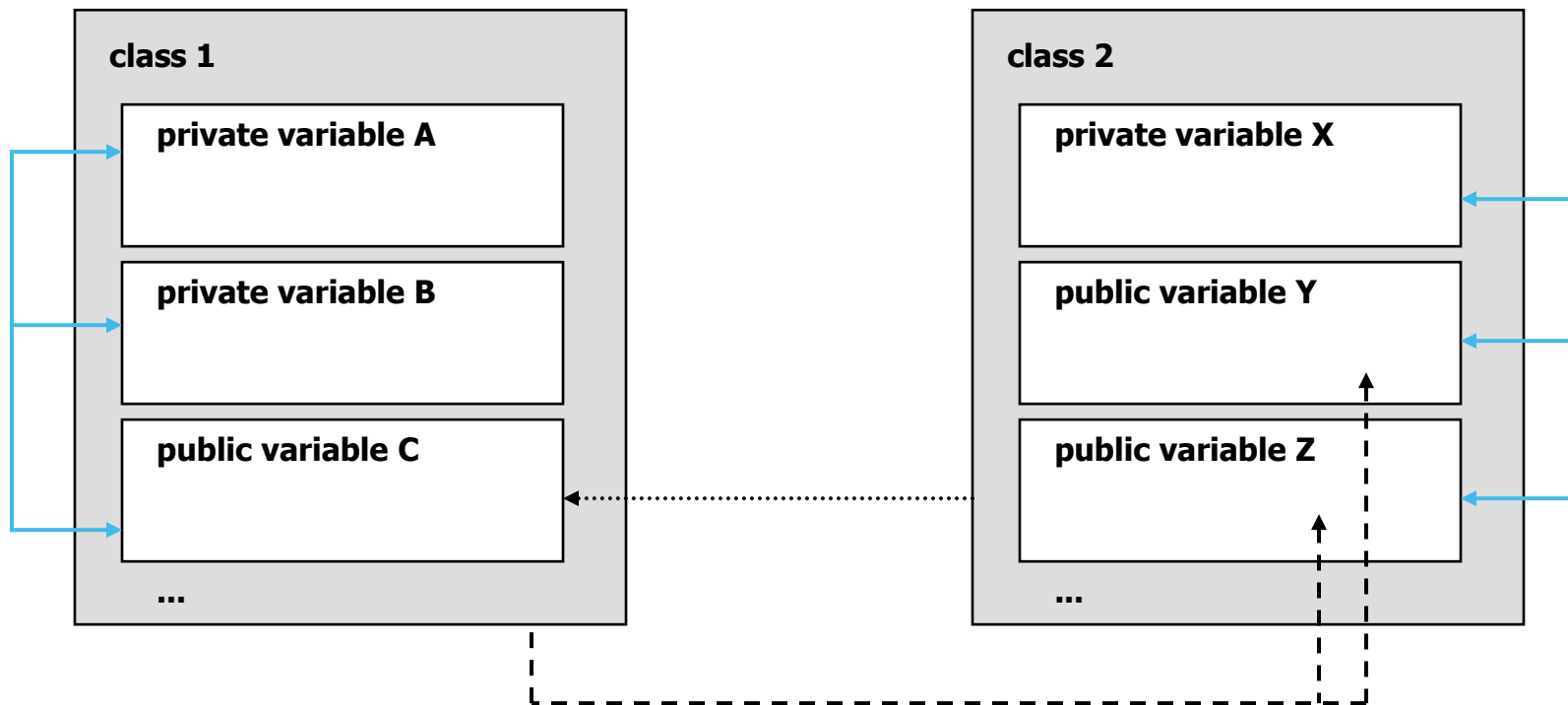
- making changes to the implementation of a well encapsulated class should have no impact on rest of system.

▣ Decouples content of information from its form of representation

- thus the user of an object will not become tied to format of the information.

Encapsulation

17



Class variables

Variable Scope

19

- Encapsulation is implemented through **access protection**.
 - ▣ Every class, data member and method in a Java program is defined as either `public`, `private`, `protected` or `unspecified`.

Attribute

No access attribute

`public`

`private`

`protected`

Permitted Access

From any class in the same package

From any class anywhere

No access from outside class

From any class in the same package and from any subclass anywhere

Scope of Member Variables

20

```
class ScopeClass
{
    private int aPrivate = 1;
    int aUnknown = 2;
    public int aPublic = 3;
}
```

```
class ScopeTest
{
    public static void main(String[] args)
    {
        ScopeClass aScope = new ScopeClass();

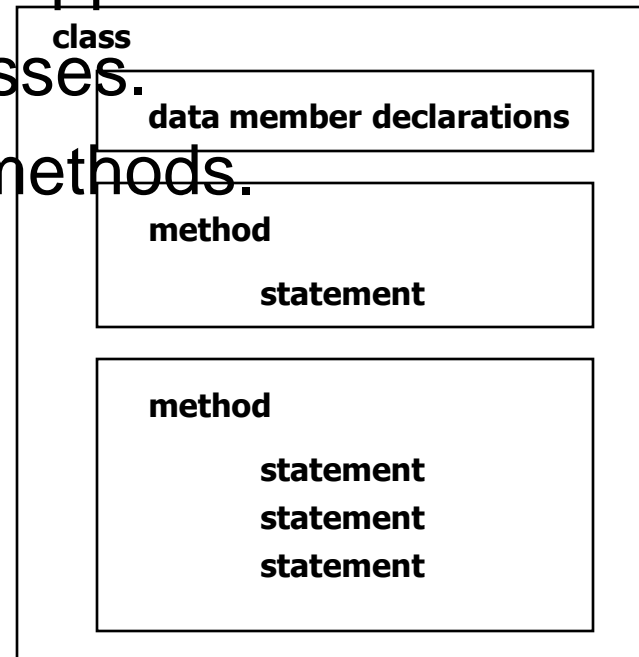
        System.out.println("Scope Test");
        System.out.println("aPrivate=" + aScope.aPrivate);
        System.out.println("aUnknown=" + aScope.aUnknown);
        System.out.println("aPublic=" + aScope.aPublic);
    }
}
```

Generates a compile error

Review: Classes

21

- ❑ Classes are the most fundamental structural element in Java
 - ▣ Every program has at least one class defined.
 - ▣ Data member declarations appear within classes.
 - ▣ Methods appear within classes.
 - ▣ statements appear within methods.



Class Definitions

22

class definition

data member declaration

method definition

method implementation

method definition

method implementation

method definition

method implementation

...

```
public class Rectangle {
```

```
    public double lowerLeftX, lowerLeftY;  
    public double upperRightX, lowerRightY;
```

```
    public double width() {  
        return upperRightX - lowerLeftX;  
    }
```

```
    public double height() {  
        return upperRightY - lowerLeftY;  
    }
```

```
    public double area() {  
        return width() * height();  
    }
```

```
}
```

Call or **invoking** the
width and *height* methods
defined above.

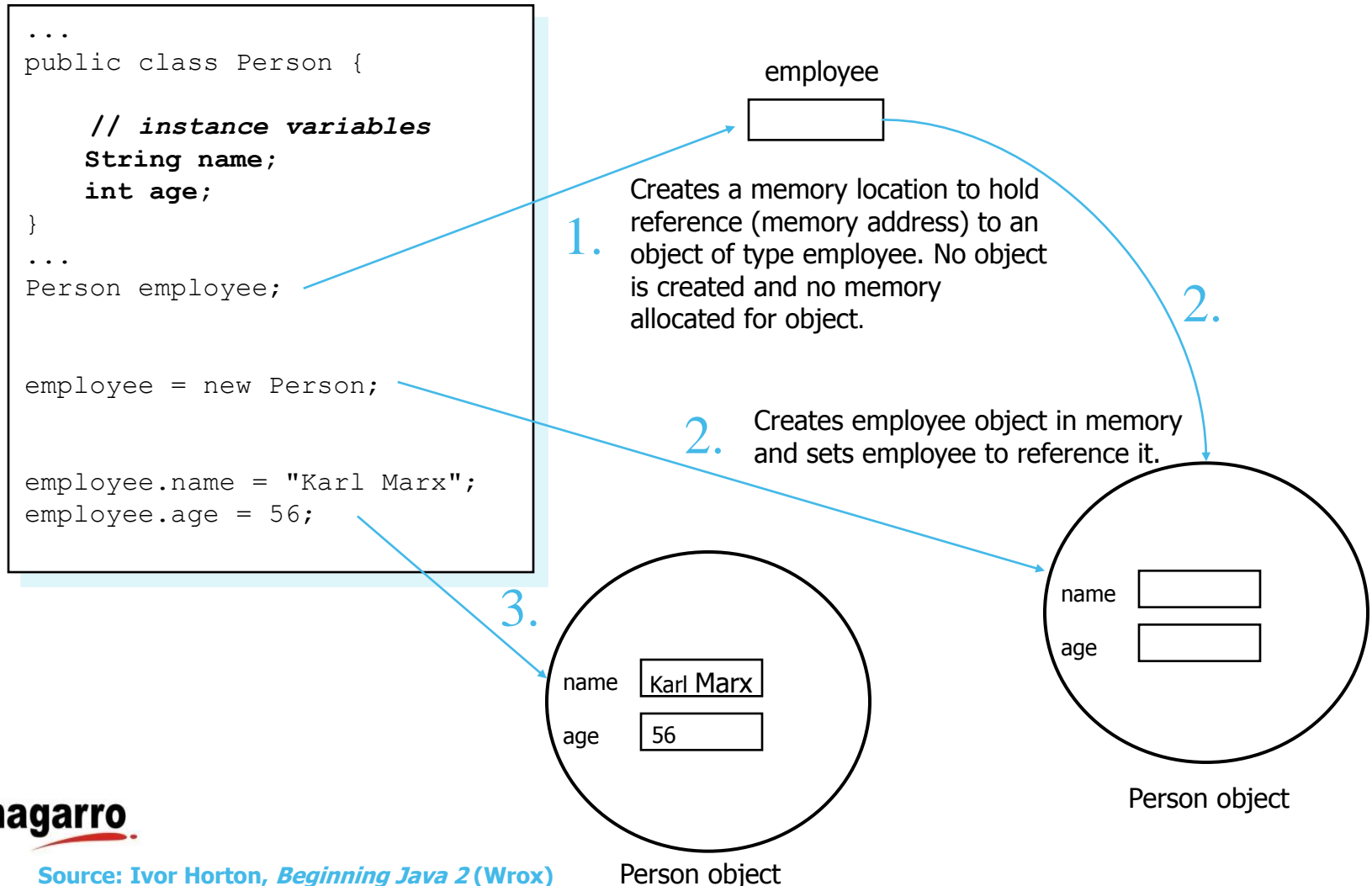
Types of members

23

- Two types of members (data members and methods):
 - ▣ **static** or **class members**
 - a member that is shared by all objects of that class
 - it is called a class member because it belongs to the class, not to a given object.
 - It is called a static member because it is declared with the `static` keyword.
 - ▣ **instance members**
 - Each instance/object of the class will have a copy of each instance member.

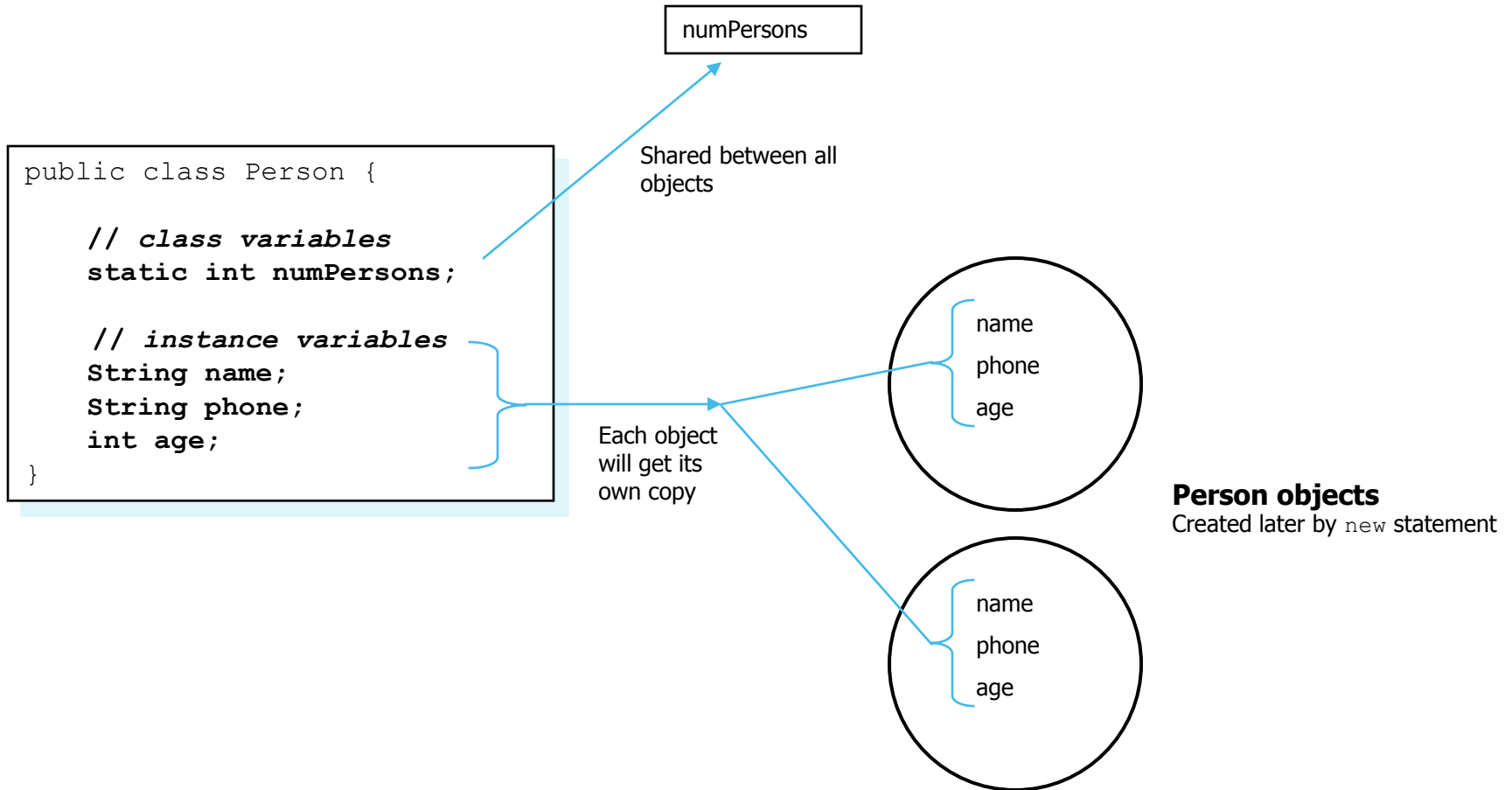
Instance variables in a class definition

24



Members in a class definition

25



Instance variables in a class definition

26

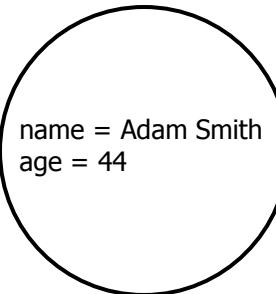
```
...  
public class Person {  
  
    // instance variables  
    String name;  
    int age;  
}  
...  
Person employee = new Person;  
Person boss = new Person;  
  
employee.name = "Karl Marx";  
employee.age = 56;  
  
boss.name = "Adam Smith";  
boss.age = 44;
```

Creates memory location to reference object **and** creates employee object in memory and sets employee to reference it.

employee



Person objects



boss

Some Scope Combinations

27

- ❑ public static
 - ▣ Static variable, that is shared between all instances, and that is available to other classes.
- ❑ private static
 - ▣ Static variable, that is shared between all instances, and that only available within its class.
- ❑ public
 - ▣ Instance variable (each instance will have a copy), that is available to other classes.
- ❑ private
 - ▣ Instance variable (each instance will have a copy), that is only available within its class.
- ❑ final
 - ▣ Indicates that the value in the variable can not be changed (i.e., it is a constant).

Some Scope Combinations

```
class ScopeClass {  
    // data members  
    private static int aPrivateStatic = 1;  
    public static int aPublicStatic = 2;  
    private final static int ONECONSTANT = 10;  
    public final static int TWOCONSTANT = 20;  
    private int aPrivateInstance = 3;  
    public int aPublicInstance = 4;  
}
```

```
class ScopeTest {  
    public static void main(String[] args) {  
        ScopeClass xScope = new ScopeClass();  
        ScopeClass yScope = new ScopeClass();  
  
        xScope.aPublicInstance = 77;  
        yScope.aPublicInstance = 44;  
        xScope.aPublicStatic = 2000;  
        yScope.aPublicStatic = 4000;  
  
        System.out.println(xScope.aPublicInstance);  
        System.out.println(xScope.aPublicStatic);  
  
        System.out.println(yScope.aPublicInstance);  
        System.out.println(yScope.aPublicStatic);  
  
        System.out.println(ScopeClass.aPublicStatic);  
        System.out.println(ScopeClass.TWOCONSTANT);  
    }  
}
```

aPrivateStatic = 1

aPublicStatic = 2 ⇒ 2000 ⇒ 4000

ONECONSTANT = 10

TWOCONSTANT = 20

xScope

aPublicInstance=77
aPrivateInstance=3

yScope

aPublicInstance=44
aPrivateInstance=3

Accessing data members directly

29

```
public class Employee {  
  
    // data members  
    public String name;  
    public double salary;  
    public static int numEmployees;  
  
}
```

```
...  
Employee boss = new Employee("Randy");  
  
    boss.salary = 50000.0  
    Employee.numEmployees = 44;  
...
```

Is this allowed?

Accessing data members directly

30

- ❑ Direct variable access of data members from other classes is allowed unless the variable is `private`.
 - ▣ e.g., `boss.salary`
- ❑ However, it is **strongly** discouraged.
 - ▣ With direct access, the class loses its ability to control what values go into an data member (i.e., no longer do we have encapsulation).
 - e.g., in previous example, user could enter a negative value for salary.

Accessors and Mutators

31

- In **Java**, provide **accessor and mutator methods** for retrieving data values and for setting data values

- e.g., `getSalary()` and `setSalary()`

- To prevent outside direct variable access, use the `private` modifier

```
private double salary;

public void setSalary(double newSalary) {
    salary = newSalary;
}

public double getSalary() {
    return salary;
}
```

- Mutators
whole package

is is their

```
public void setSalary(double newSalary) {
    if (newSalary >= 0)
        salary = newSalary;
    else
        throw new Exception("Illegal salary in setSalary");
}
```

Constructors

Constructors

33

- When an object of a class is created (via `new`), a special method called a **constructor** is always invoked.
 - ▣ You can supply your own constructor, or let the compiler supply one (that does nothing).
 - ▣ Constructors are often used to initialize the instance variables for that class.
- Two characteristics:
 - ▣ constructor method has same name as the class.
 - ▣ Constructor method never returns a value and must not have a return type specified (not even `void`).

Constructors

34

```
public class Person {  
  
    // data members  
    public String name;  
    public String phone;  
    public int age;  
  
    // class constructor  
    public Person()  
    {  
        age = -1;  
    }  
  
    // class methods  
  
    // instance methods  
}
```

```
...  
Person employee = new Person();  
  
Person boss = new Person();  
...
```

Each time a Person object is created, the class constructor is called (and as a result, employee._age and boss._age is initialized to -1)

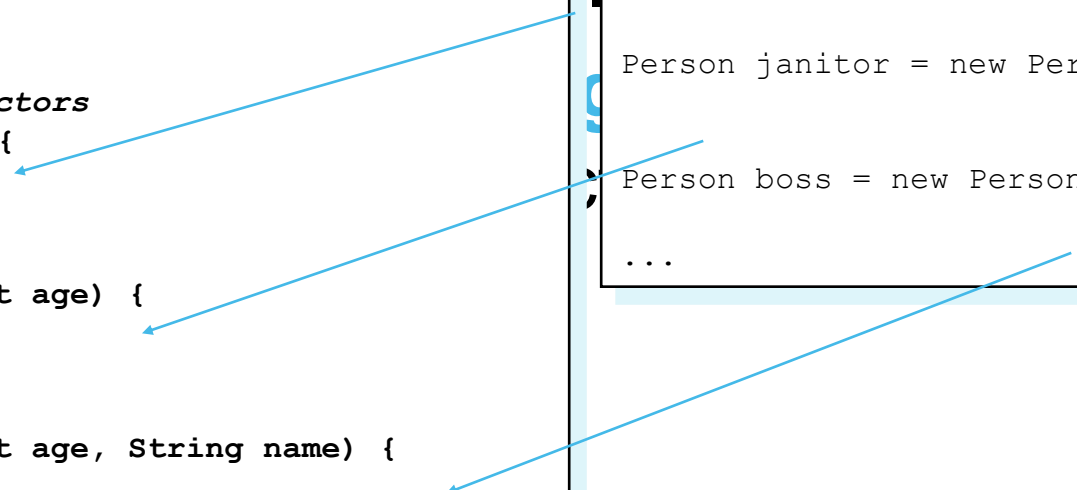
Multiple Constructors

35

- ❑ You can supply more than one constructor method to a class.

```
public class Person {  
  
    // instance variables  
    String _name;  
    int age;  
  
    // class constructors  
    public Person() {  
        age = -1;  
    }  
  
    public Person(int age) {  
        age = newAge;  
    }  
  
    public Person(int age, String name) {  
        age = age;  
        name = name;  
    }  
}
```

```
...  
Person employee = new Person;  
  
Person janitor = new Person(45);  
  
Person boss = new Person(30, "Fred");  
  
...
```



Method Overloading

36

- ❑ There are situations where you need a method that does similar things but accept different lists of arguments.
- ❑ Java allow you to **overload** a method definition so that the method can accept a different number of arguments.
 - ▣ That is, they allow multiple methods to have the same name, as long as those methods have different lists of arguments.
 - ▣ It makes a method more flexible

```
class Calculator {  
    public int add(int a, int b) {  
        return a + b;  
    }  
    public double add(double a, double b) {  
        return a + b;  
    }  
}
```

```
...  
Calculator calc = new Calculator();  
int a = calc(3,5);  
double b = calc(34.56,342.45);  
...
```

- ❑ Method overloading is an important part of object-oriented development
 - ▣ Subclasses can overload methods defined in a superclass.

Coupling and Cohesion

Coupling

38

- Coupling is the degree to which one class knows about another class.
- If the only knowledge that class A has about class B, is what class B has exposed through its interface, then class A and class B are said to be loosely coupled.
- Benefits
 - ▣ Ease of maintenance
 - ▣ Ease of enhancement

Coupling

39

```
class CalculateTaxes {  
    float rate;  
    float doIndia() {  
        TaxRatesInIndia str = new  
        TaxRatesInIndia();  
        rate = str.salesRate; // ouch  
        // this should be a method  
        call:  
        // rate =  
        str.getSalesTaxRates("CO"  
        );  
        // do stuff with rate  
    }  
}
```

```
class TaxRatesInIndia {  
    public float salesRate; // should  
    be private  
    public float adjustedSalesRate;  
    // should be private  
  
    public float  
    getSalesTaxRates(String  
    region) {  
        salesRate = new  
        CalculateTaxes().doIndia(); //  
        ouch again!  
        // do country-based calculations  
        return adjustedSalesRate;  
    }  
}
```

Cohesion

40

- ❑ Cohesion is all about how a single class is designed.
- ❑ A class should have a single, well-focused purpose, resulting in high cohesion
- ❑ Benefits
 - ▣ Classes are much easier to maintain (and less frequently changed)
 - ▣ Classes with a well-focused purpose tend to be more reusable than other classes

Cohesion

41

```
class SalesReport {  
    void connectToDb(){ }  
    void generateSalesReport() { }  
    void saveAsFile() { }  
    void print() { }  
}
```

```
class SalesReport {  
    Options getReportingOptions() { }  
    void generateSalesReport(Options o) { }  
}
```

```
class ConnectToDb {  
    DBconnection getDb() { }  
}
```

```
class PrintStuff {  
    PrintOptions getPrintOptions() { }  
}
```

```
class FileSaver {  
    SaveOptions getFileSaveOptions() { }  
}
```

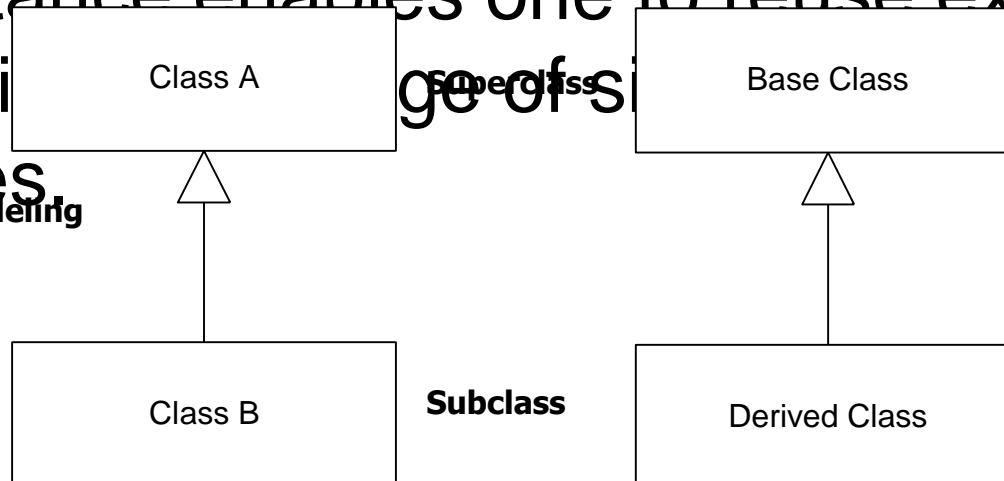
Inheritance

Inheritance

43

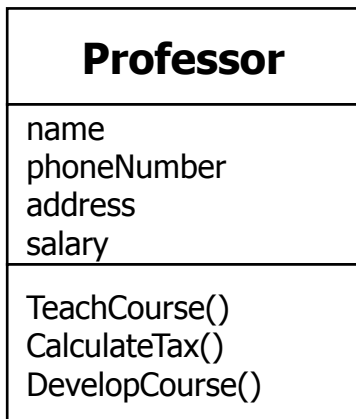
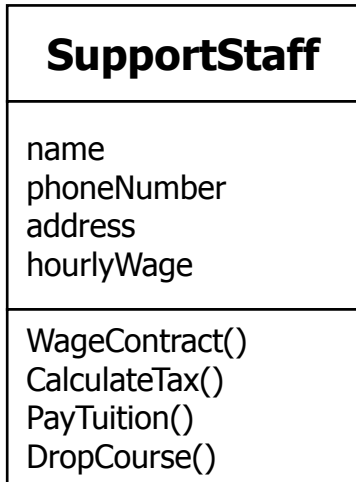
- **Inheritance** is the facility by which a class A has implicitly defined upon it each of the attributes and operations of another class B.
 - A is a **superclass** of B
 - B is a **subclass** of A
- Inheritance enables one to reuse existing code by taking advantage of similarities between classes.

The UML modelling notation for inheritance

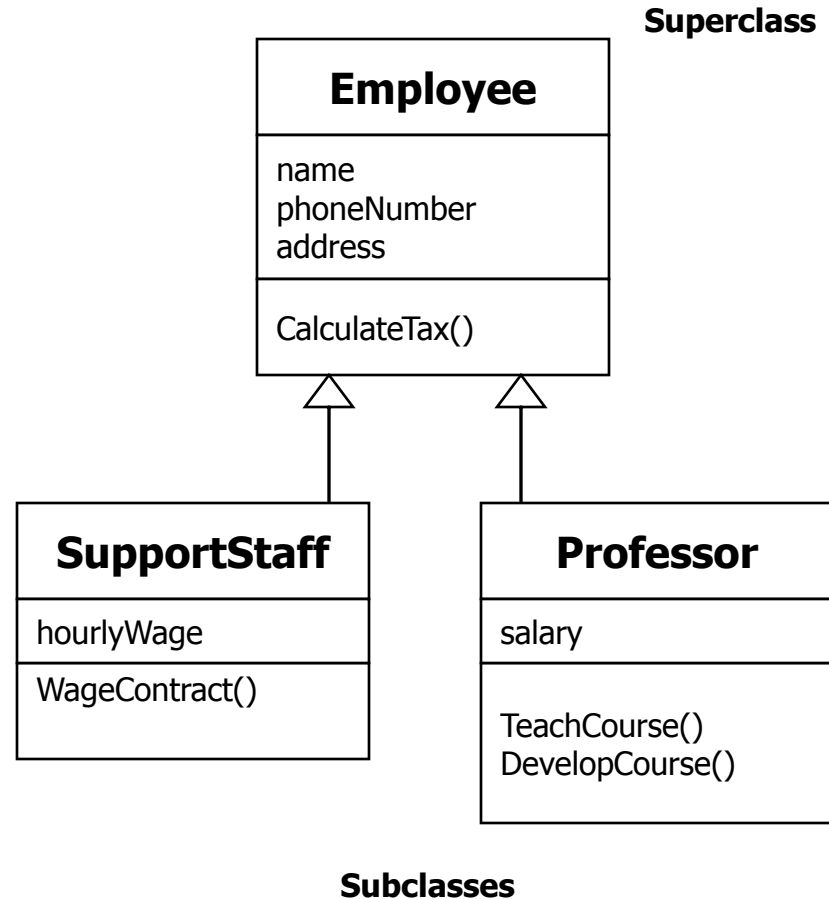


Inheritance

44



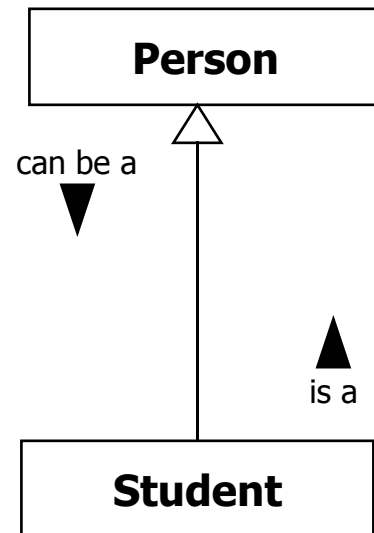
OR



Both the SupportStaff and Professor classes inherit the methods and attributes of the Employee class.

Inheritance

45





A subclass may inherit the structure and behavior of its superclass.

Inheritance

47

- Inheritance is the representation of an **is a**, **is like**, **is kind of** relationship between two classes.
 - e.g., a Student **is a** Person, a Professor **is kind of** a Person
 - **is a** relationships are best.
- With inheritance, you define a new class that encapsulates the similarities between two classes
 - e.g., for Student and Professor, the new class was Person, which contains the attributes and methods contained in both.

Inheritance Tips

48

- Look for similarities
 - ▣ whenever you have similarities between two or more classes (either attributes or methods), you probably have an opportunity for inheritance.
- Look for existing classes
 - ▣ when you create a new class, you may already have an existing class to which it is similar. Perhaps you can inherit from this existing class and code just the differences.
- Follow the sentence rule
 - ▣ It should make sense to use the classes in one of the sentences:
"a X **is a** Y", "a X **is a kind of** Y", or "a X **is like a** Y"

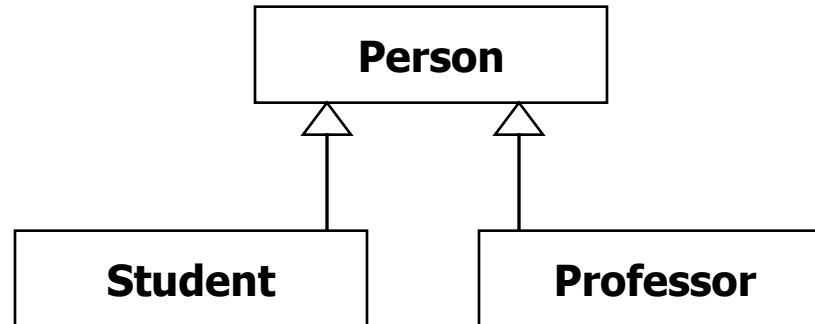
Inheritance Tips

49

- ❑ Avoid implementation inheritance
 - i.e., don't apply inheritance just to use a feature of another class if the sentence rule does not apply.
 - Be cautious of "is like a" applications of inheritance (more on this later).
- ❑ Inherit everything
 - the subclass should inherit all features of its superclass.
 - i.e., you shouldn't say "X is a Y, except for ..."

Inheritance in Java

50



```
public class Person
{
    // data members and methods here
}
```

```
public class Student extends Person
{
}
```

```
public class Professor extends Person
{
}
```

```
public class Employee {
    private String employeeName;
    private String employeeDept;
    private int yearsService;

    public Employee(String name, String dept, int years) {
        employeeName=name; employeeDept=dept; yearsService=years;
    }

    public String getName() { return employeeName; }
    public String getDept() { return employeeDept; }
    public int getYearsService() { return yearsService; }
    public double getHolidays() { return 20.0; }
}
```

```
public class Technician extends Employee {
    private int serviceCalls;

    public Technician(String name, String dept, int years) {
        super(name, dept, years);
    }

    public int getServiceCalls() { return serviceCalls; }
}
```

```
public class Manager extends Employee {

    public Manager(String name, String dept, int years) {
        super(name, dept, years);
    }

    public double getHolidays() { return 30.0; }
}
```

Invokes constructor
from superclass

overrides

Interfaces

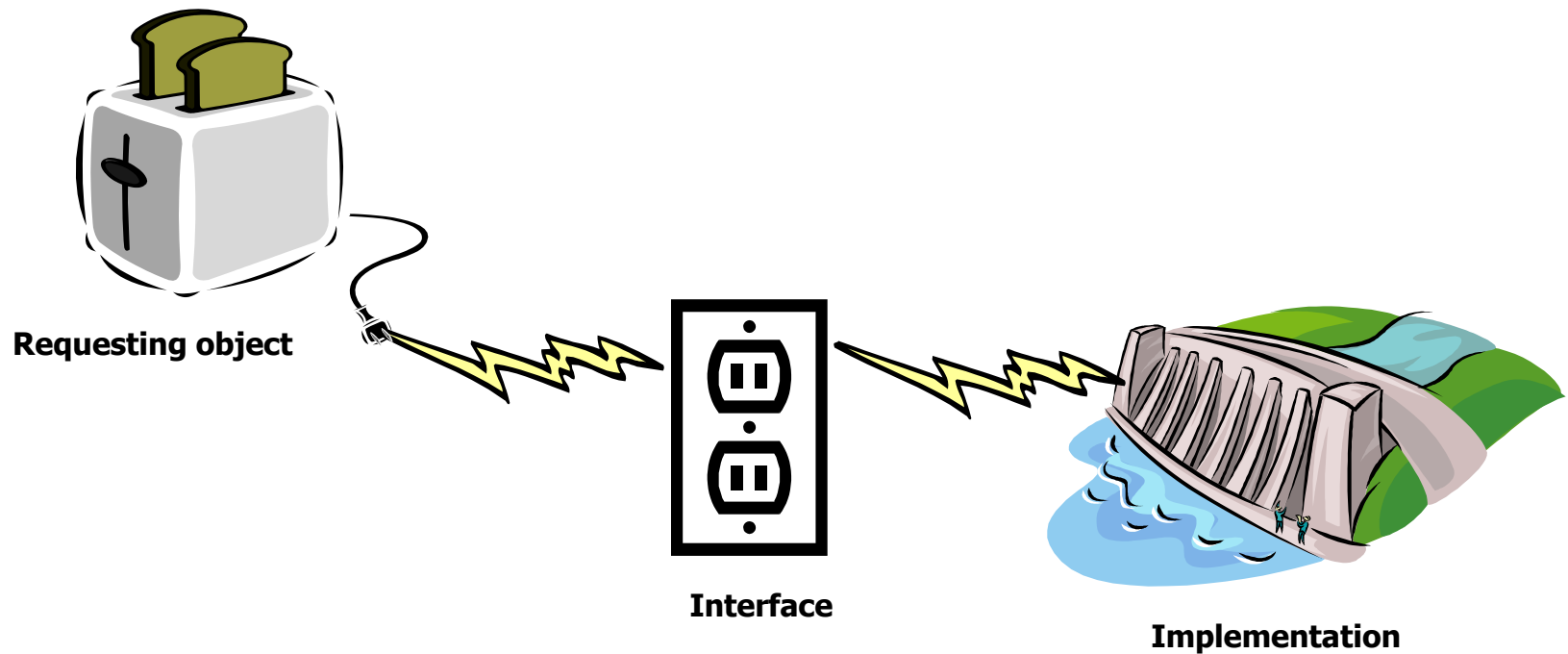
Interfaces

53

- The attributes and methods that a class provides to other classes constitute the class's **interface**.
- The interface thus completely describes how users of that class interact with the class.
 - ▣ In Java methods and attributes are designated using either the `public`, `private` or `protected` keyword.

Real-world Interface & Implementation

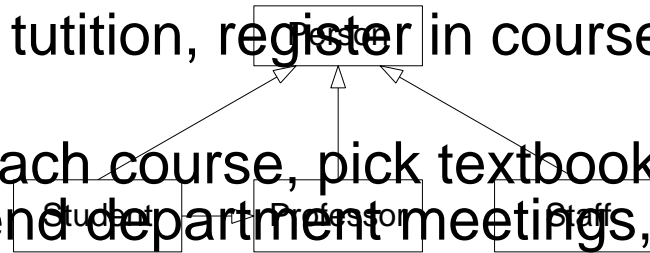
54



Interface Example

55

- ❑ A university system has a Student, a Professor, and a SupportStaff classes.
 - ▣ Students pay tuition, register in courses, receive grades, etc
 - ▣ Professors teach course, pick textbooks, supervise students, attend department meetings, pay union dues, etc
 - ▣ Support Staff answer phones, clean the floors, fix computers, etc.
- ❑ All three are subtypes of the Person supertype.
 - ▣ Persons have names, addresses, birthdays, etc
- ❑ However, it is possible that a Student may teach a course (and then pick textbooks), but not supervise other students, attend department meetings, or pay dues.



This isn't right: No multiple inheritance allowed

As well, this isn't right because a Student is not always a type of Professor

Implementing Interfaces (Java)

```
public interface Teacher
{
    public void teachCourse();
    public void pickTextBook();
}
```

```
class Professor extends Person implements Teacher
{
    public void teachCourse() {
        ...
    }
    public void pickTextbook() {
        ...
    }
    // other methods here
    ...
}
```

```
class Student extends Person implements Teacher
{
    public void teachCourse() {
        ...
    }
    public void pickTextbook() {
        ...
    }
    // other methods here
    ...
}
```


Two Types of Inheritance

57

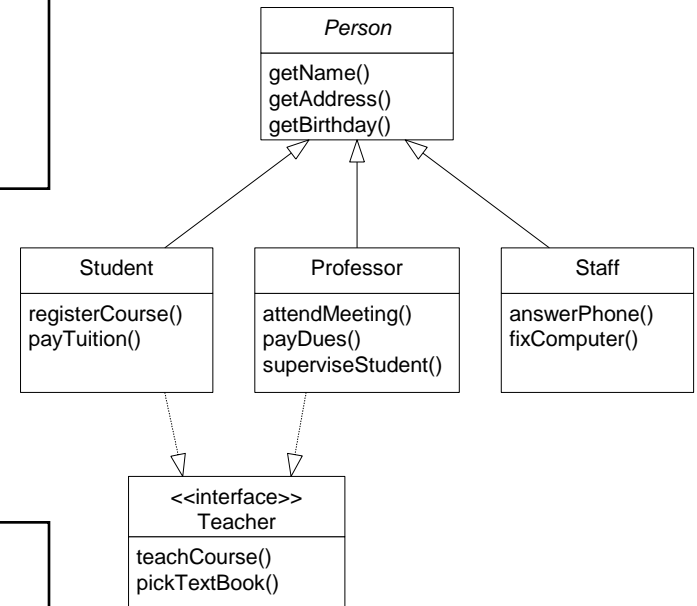
- There are two types of inheritance:
 - ▣ Implementation inheritance
 - ▣ Interface inheritance

Two Types of Inheritance

58

```
...  
Person s = new Professor();  
Person p = new Student();  
...  
System.out.println( s.getName() );  
System.out.println( p.getName() );  
...
```

Implementation inheritance



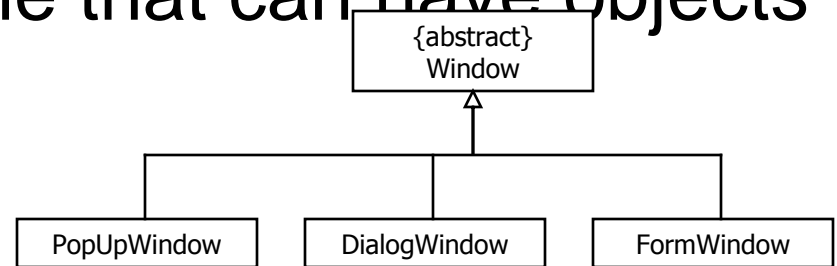
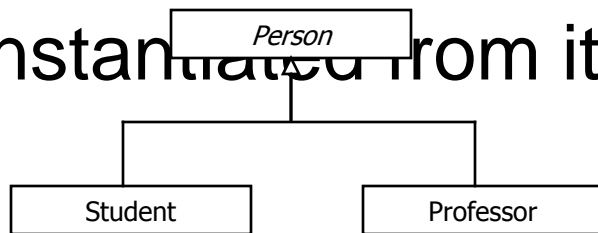
```
...  
Teacher t = new Professor();  
Teacher t = new Student();  
...  
s.teachCourse();  
t.teachCourse();  
...
```

Abstract and Concrete Classes

Abstract and Concrete classes

60

- An **abstract class** is one that can not have objects instantiated from it.
 - ▣ In diagrams, an abstract class is indicated via italics
 - ▣ When italics are impractical (e.g., whiteboards, blackboards), then use the stereotype {abstract}
- A **concrete class** is one that can have objects instantiated from it.



Abstract and Concrete classes

61

- Abstract classes allow you to do either or both:
 - ▣ Specify the required **interface** of concrete classes that will be implemented by the concrete subclasses.
 - ▣ Specify the **implementation** of common functionality (i.e., implement members).

Abstract and Concrete classes

62

- For example, let's imagine that for some health management software for a veterinarian, we've come up with the following **concrete** objects, which will correspond to the actual animals coming into the vets. Now try to group them under one or more **abstract** classes.

Dog

Cat

Mouse

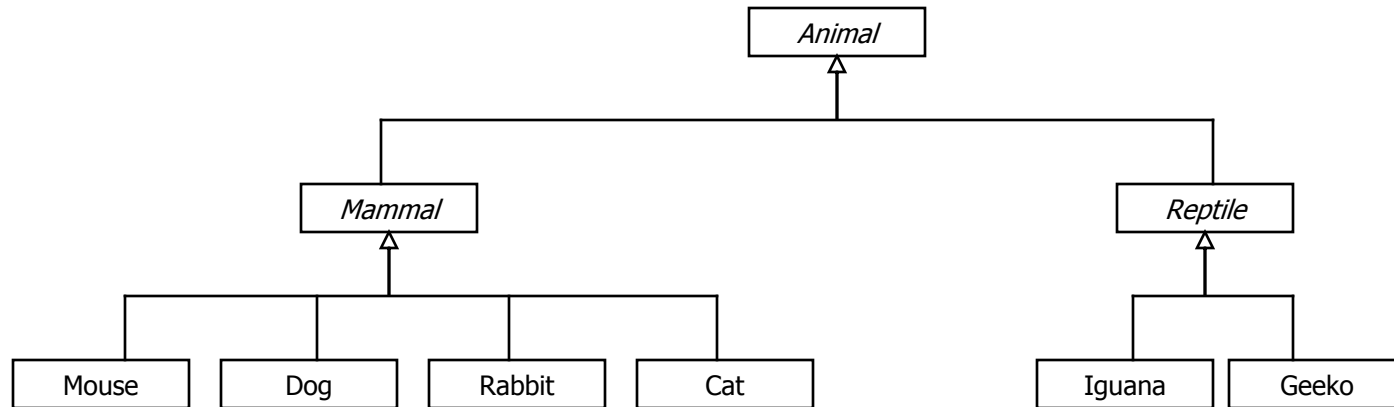
Rabbit

Iguana

Geeko

Abstract and Concrete classes example

63



Mammal and *Animal* are abstract classes, since it wouldn't be a mammal that comes in to get treated, but a concrete kind of mammal, such as a dog or cat. That is, it is unlikely that the veterinarian wouldn't know what kind of mammal or animal he or she was treating.

Abstract Methods

64

- ❑ Methods can also be abstract.
- ❑ Java lets you define a method or method without implementing it by declaring the method with the `abstract` modifier.
 - ▣ This is useful for a superclass, in that you want the subclass to inherit the method signature (i.e., the interface), but the subclasses implement the method.
- ❑ Only instance methods can be abstract
 - ▣ That is, static methods cannot be abstract
- ❑ if a class contains an abstract method, then the class itself must also be declared `abstract`.

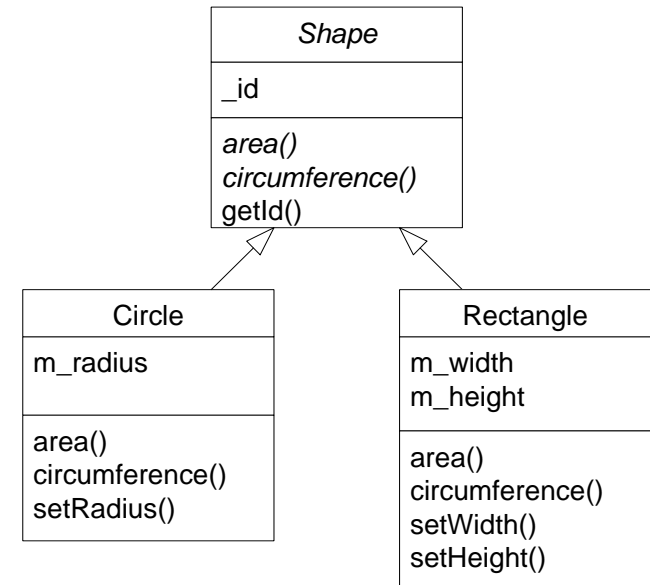
Abstract Class Example (Java)

```
public abstract class Shape {  
    ...  
    public abstract double area();  
    public abstract double circumference();  
    public int getId() { return _id; };  
}
```

Note!

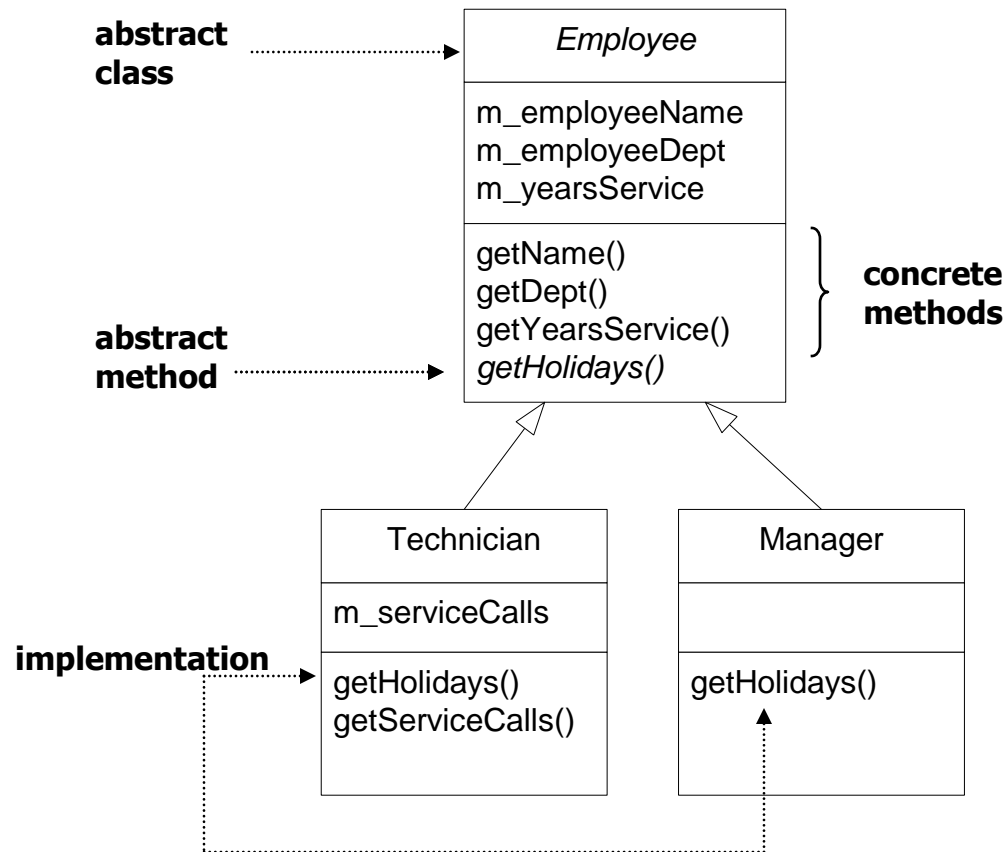
```
public class Circle extends Shape {  
    private double m_radius;  
    ...  
    public double area() {  
        return 3.14159 * m_radius * m_radius;  
    }  
    public double circumference() {  
        return 2.0 * 3.14159 * m_radius;  
    }  
    ...  
}
```

```
public class Rectangle extends Shape {  
    private double m_width, m_height;  
    ...  
    public double area() {  
        return m_width * m_height;  
    }  
    public double circumference() {  
        return 2.0 * (m_width + m_height);  
    }  
    ...  
}
```



Another Abstract Class Example

66



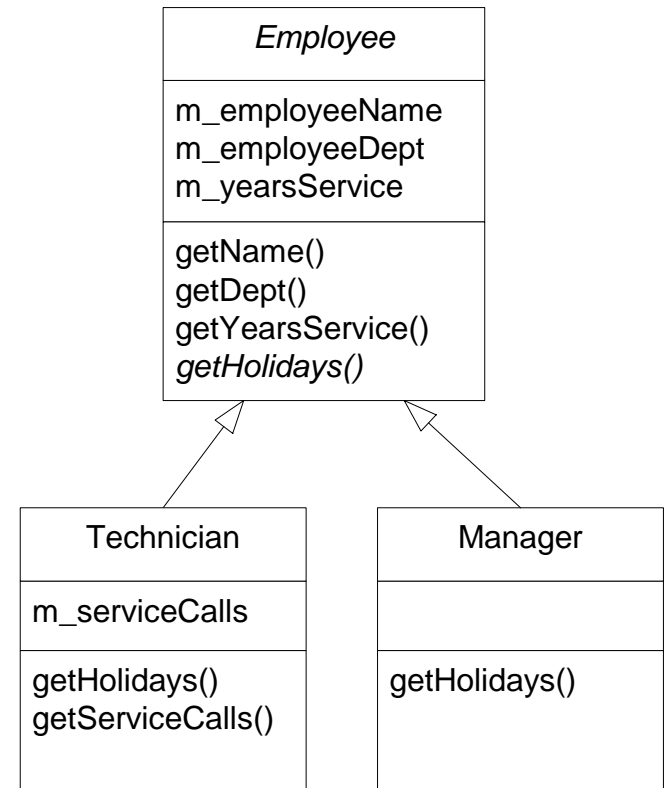
Abstract Class and Method Example

67

```
public abstract class Employee {  
    ...  
    public abstract double getHolidays();  
    ...  
}
```

```
public class Technician extends Employee {  
    ...  
    public getHolidays() { return 15.0; }  
    ...  
}
```

```
public class Manager extends Employee {  
    ...  
    public getHolidays() { return 30.0; }  
    ...  
}
```



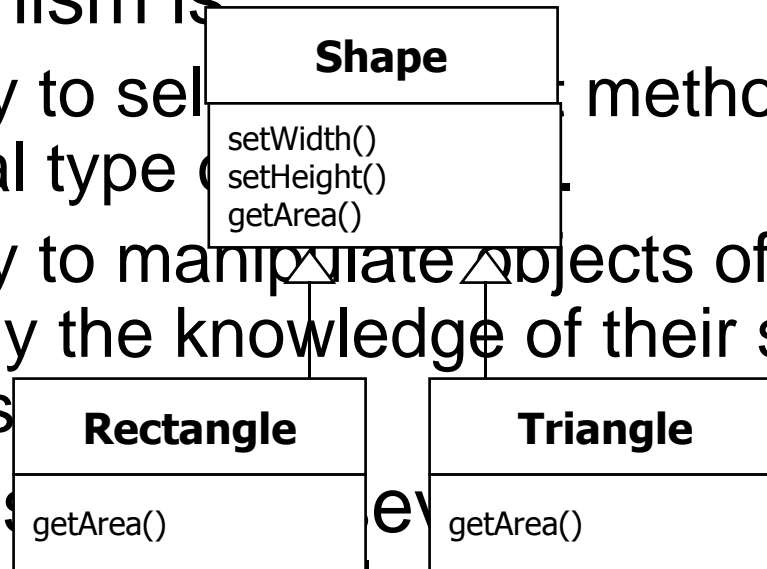
Polymorphism

Polymorphism

69

□ Polymorphism is:

- ▣ the ability to select methods according to the actual type of the object
- ▣ the ability to manipulate objects of distinct classes using only the knowledge of their shared members



- It allows us to use even versions of a method in different classes of a subclass hierarchy and give them all the same name.
- The subclass version of an attribute or method is said to **override** the version from the

nagarro

superclass.

Source: David William Brown, *An Intro to Object-Oriented Analysis* (Wiley, 2002), p. 235

Polymorphism

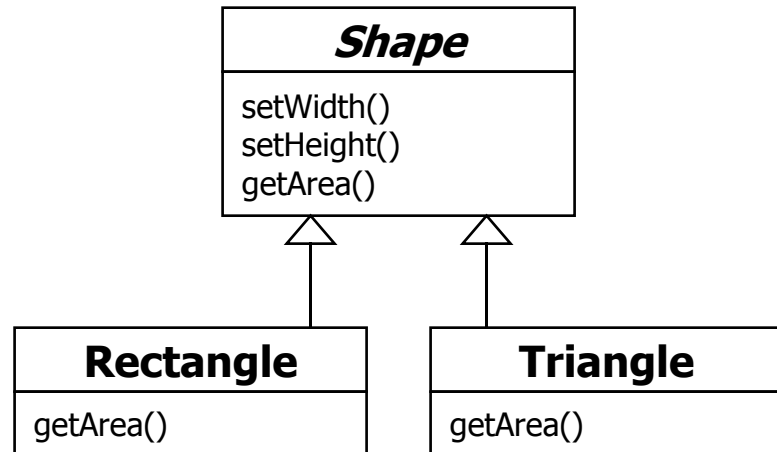
70

```
public class Shape {  
    double width, height;  
    public abstract double getArea();  
  
    public void setWidth(double newWidth) {  
        width = newWidth;  
    }  
    public void setHeight(double newHeight) {  
        height = newHeight;  
    }  
}
```

```
public class Rectangle extends Shape {  
    public double getArea() {  
        return (width * height);  
    }  
}
```

```
public class Triangle extends Shape {  
    public double getArea() {  
        return (0.5 * width * height);  
    }  
}
```

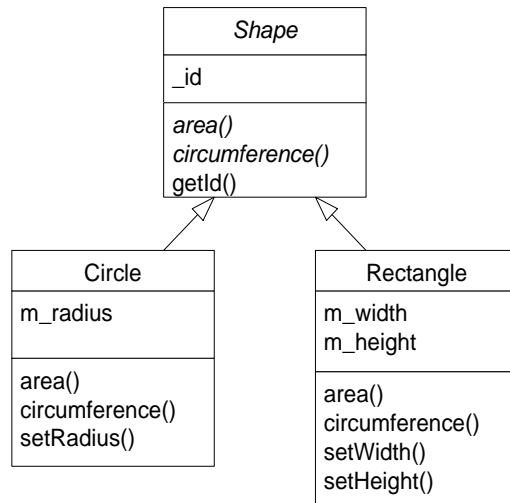
```
...  
Rectangle myRect = new Rectangle();  
myRect.setWidth(5);  
myRect.setHeight(4);  
System.out.println myRect.getArea();  
...
```



Polymorphism and Substitutability

71

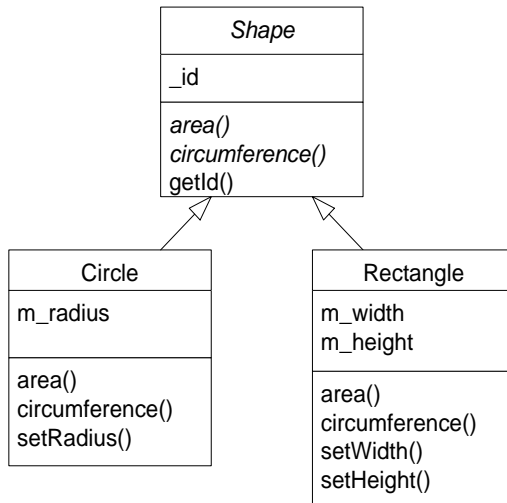
- ❑ Polymorphism allows us to program in a way consistent with LSP.
 - ▣ That is, I write code assuming I have the superclass, it should work with any subclass.



```
...
Shape myRect = new Rectangle();
Shape myCircle = new Circle();
...
myRect.setWidth(50);
myRect.setHeight(100);
myCircle.setRadius(45);
...
System.out.println( myRect.area() );
System.out.println( myCircle.area() );
...
```

Polymorphism and Substitutability

72



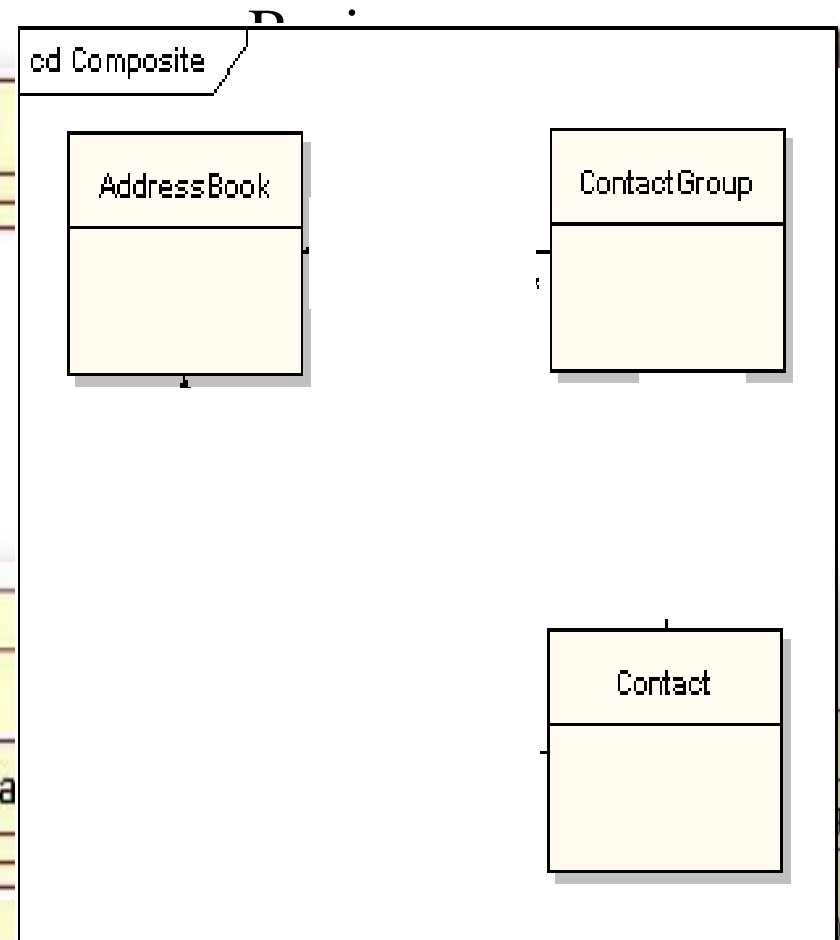
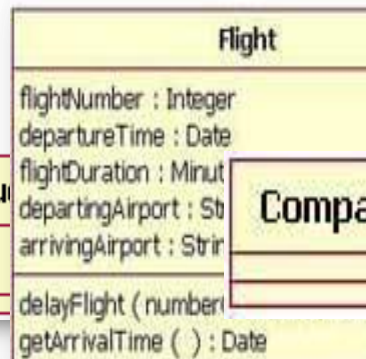
```
...
Rectangle myRect = new Rectangle();
Circle myCircle = new Circle();
...
myRect.setWidth(50);
myRect.setHeight(100);
myCircle.setRadius(45);
...
print(myRect);
print(myCircle);
...

private void print(Shape s)
{
    System.out.println("Id=" + s.getId() );
    System.out.println("Area=" + s.area() );
}
```


Class Diagram

Class Diagram

- Building blocks of any object-orientated system
- Inheritance (is-a relationship)
- Associations (has-a relationship)
- Static view of the system
- Aggregations ("whole to its parts" relationship)
- Compositions



Recap

- ▣ Object and Class
- ▣ Methods and Attributes
- ▣ Abstraction
- ▣ Encapsulation, information hiding
- ▣ Coupling, Cohesion
- ▣ Inheritance
- ▣ Polymorphism
- ▣ Class diagram

Assignment

try this at home





Assignment

77

Assignment: OOP – Billing System

In this assignment, an application 'billing system' of a retail website has to be developed in which different type of customers have different type of discount schemes. And for a given bill, a net payable amount should be generated based on the type of the customer and the product. By the time you have worked your way through the training material mentioned in the course pathway, you will see how OOPs concept can be used to build an application.

As part of your deliverables, you need to provide following:

1. Running application
2. A small write-up explaining the different OOP concept used in this application.
3. Source Code (should be refactored) of the application

References

- ❑ http://en.wikipedia.org/wiki/Object-oriented_programming
- ❑ <http://www.codeproject.com/Articles/22769/Introduction-to-Object-Oriented-Programming-Concep>
- ❑ <http://www.dotnetfunda.com/articles/article1003-basic-concepts-of-oop-encapsulation-and-inheritance.aspx>





Thank You

Questions / Queries / Feedback ?

Send to jsag@nagarro.com