

CSC584 Enterprise Programming

MOHD HANAPI ABDUL LATIF

PROGRAMMING CHAPTER 1 – REVIEW OF OBJECT ORIENTED
PROGRAMMING

A solid orange horizontal bar spanning the width of the slide at the bottom.

Chapter 1 Outline

- **Review of Object Oriented Programming**
- a) Object Oriented Programming Concepts
 - Objects, classes, packages
- **b) Inheritance & Polymorphism concepts**
 - Inheriting instances fields and methods
 - Method overriding
 - Access levels –public, protected, private
 - Abstract super classes and methods
 - Interface

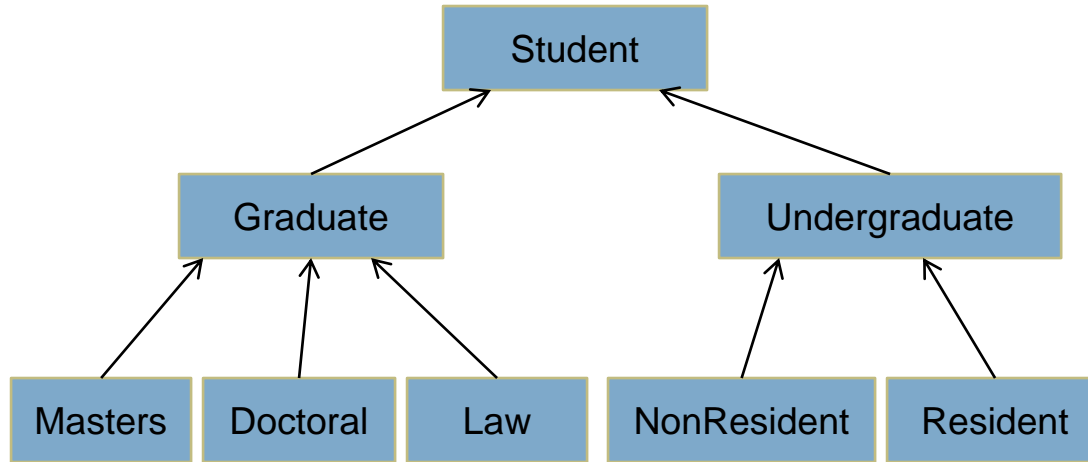
Inheritance

Inheritance

- Inheritance enables a class to inherit the methods and fields of another class.
- A new class (subclass) is derived from existing class (superclass).
- Subclass inherits attributes & behavior from superclass.
- Advantage:
 - code/software reusability
 - save time and cost.

Inheritance

- Hierarchical Diagram



Superclass

- A class that is used as a base for inheritance.
- It is also called a base class or parent class.
- Every class has only one direct superclass (single inheritance)

Subclass

- More specialized group of objects.
- Automatically **inherits** methods and fields of superclass.
- Add methods and fields of its own.
- Become a candidate to be a superclass for some future subclass.

Inheritance Concept - Terminology

Terminology	Description
Superclass	the class that is inherited from (the parent class)
Subclass	the class that does the inheriting (the child class)
Extends	in Java the keyword extends means that a class will inherit from another class
Overload	a method is overloaded if there are two or more methods with the same name in a class. Each overloaded method has a different set of parameters.
Override	a method is overridden if there is a method in the subclass that has the same name and the same set of parameters. The superclass method is then NOT inherited.

Type of Inheritance

- **Single inheritance**
 - Inherits from one superclass.
- **Multiple inheritance**
 - Inherits from multiple superclasses.
- Java only permits single inheritance.

Inheritance examples

Super Class	Sub Class
Student	GraduateStudent UndergraduateStudent
Shape	Circle Triangle Rectangle
Employee	FullTime PartTime
BankAccount	CheckingAccount SavingsAccount

Defining Classes with Inheritance

Case Study:

- Suppose we want implement a class worker that contains both **FullTime** and **PartTime** employees.
- Each employee's record will contain his or her name, and the employee id.
- The formula for determining the salary is different for full-time employees than for part-time employees.

Modeling Employee Class

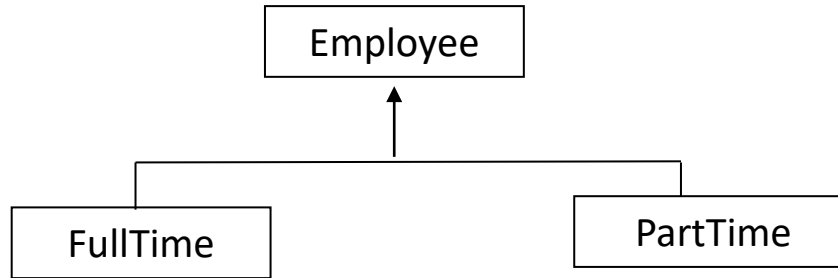
- We can model the employees by using classes that are related in an inheritance hierarchy.

Classes for the Class Worker

- For the Class Worker sample, we design three classes:
 - Employee
 - FullTime
 - PartTime
- The Employee class will incorporate behavior and data common to both **FullTime** and **PartTime** objects
- The **FullTime** class and the **PartTime** class will each contain behaviors and data specific to their respective objects.

An inheritance Diagram

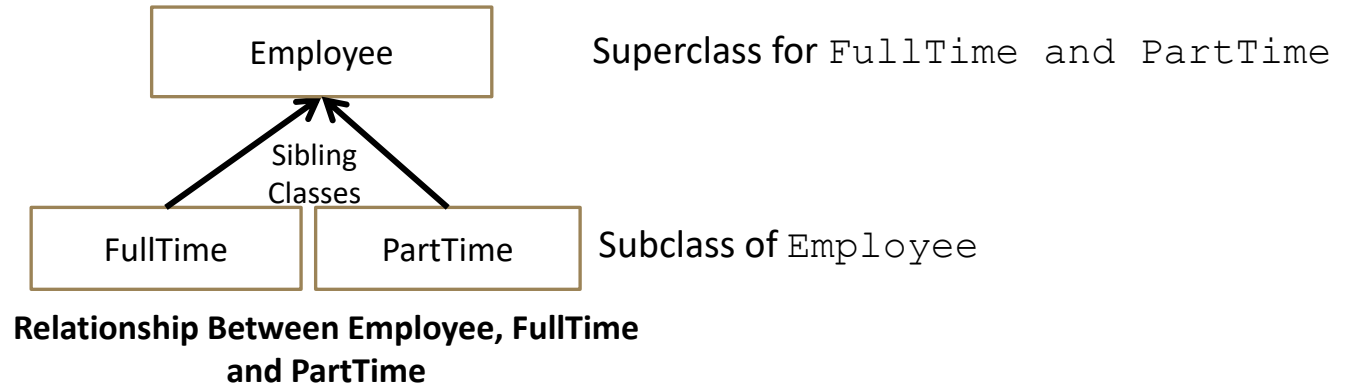
Every class extends the Object class either directly or indirectly.



FullTime and **PartTime** are subclasses that extends the superclass **Employee**

Superclass / subclass

- The inheritance relationship is also referred to as **is-a** relationship (is-a link)
- In the example, a FullTime is a Employee
- Every class that does not specifically inherit another class is a subclass of the class `Object`



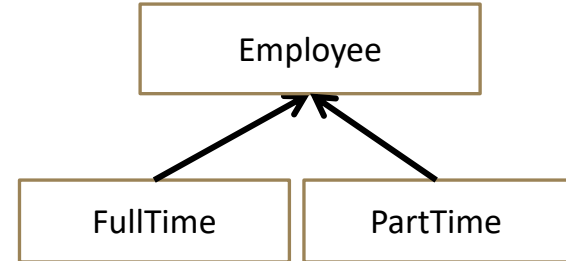
Inheriting – object instantiation

VALID DECLARATION

Employee wrk = new Employee(); ✓
Or
Employee wrk = new FullTime(); ✓
Or
Employee wrk = new PartTime(); ✓
Or
FullTime wrk = new FullTime(); ✓
Or
PartTime wrk = new PartTime(); ✓

INVALID DECLARATION

FullTime wrk1, wrk2;
wrk1 = new Employee(); ✗
wrk2 = new PartTime(); ✗
FullTime wrk = new PartTime(); ✗
PartTime wrk = new FullTime(); ✗



Inheritance (Sample Program)

```
public class Employee {  
    // Data members  
    private String name;  
    private long empId;  
  
    // Default constructor  
    public Employee() {  
        name = "noname";  
        empId = 0;  
    }  
    // Normal constructor  
    public Employee(String n, long id) {  
        name = n;  
        empId = id;  
    }  
    //Accessors  
    public String getName() { return name;}  
    public long getEmpId() { return empId;}  
  
    //Returns the string representation of the object.  
    public String toString(){  
        return "employee name= " +name + "Emp id= " + empId;  
    }  
}
```

```
public class FullTime extends Employee {  
    // Data members  
    private double base;  
    private double allow;  
  
    // Accessors  
    public double getBase() { return base; }  
    public double getAllow(){ return allow; }  
  
    public double calSalary(){  
        double salary;  
        salary = base + allow;  
        return salary;  
    }  
}
```

Member Accessibility

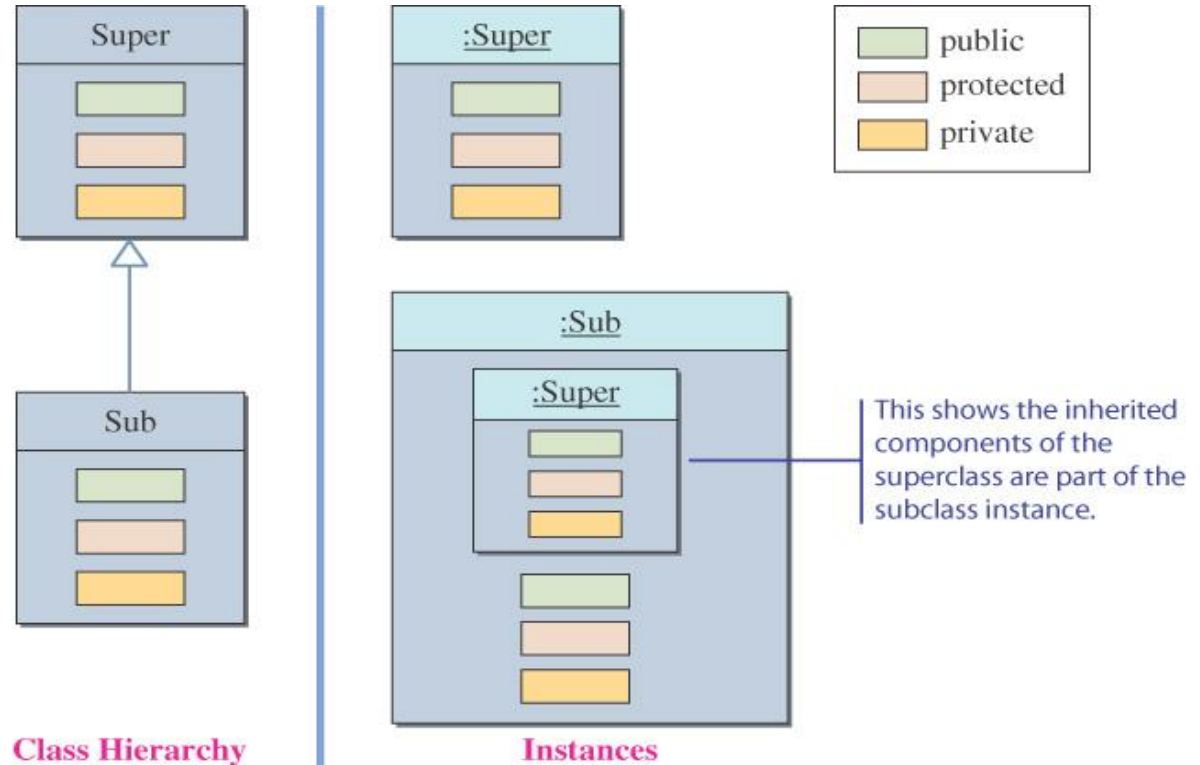
- Java has four (4) levels of controlling access to fields, methods, and classes:
- **public** access
 - Can be accessed by methods of all classes
- **protected** access
 - Can be accessed only by methods that belong to the same class or to the descendant classes
- **private** access
 - Can be accessed only by the methods of their own class

Member Accessibility (cont'd)

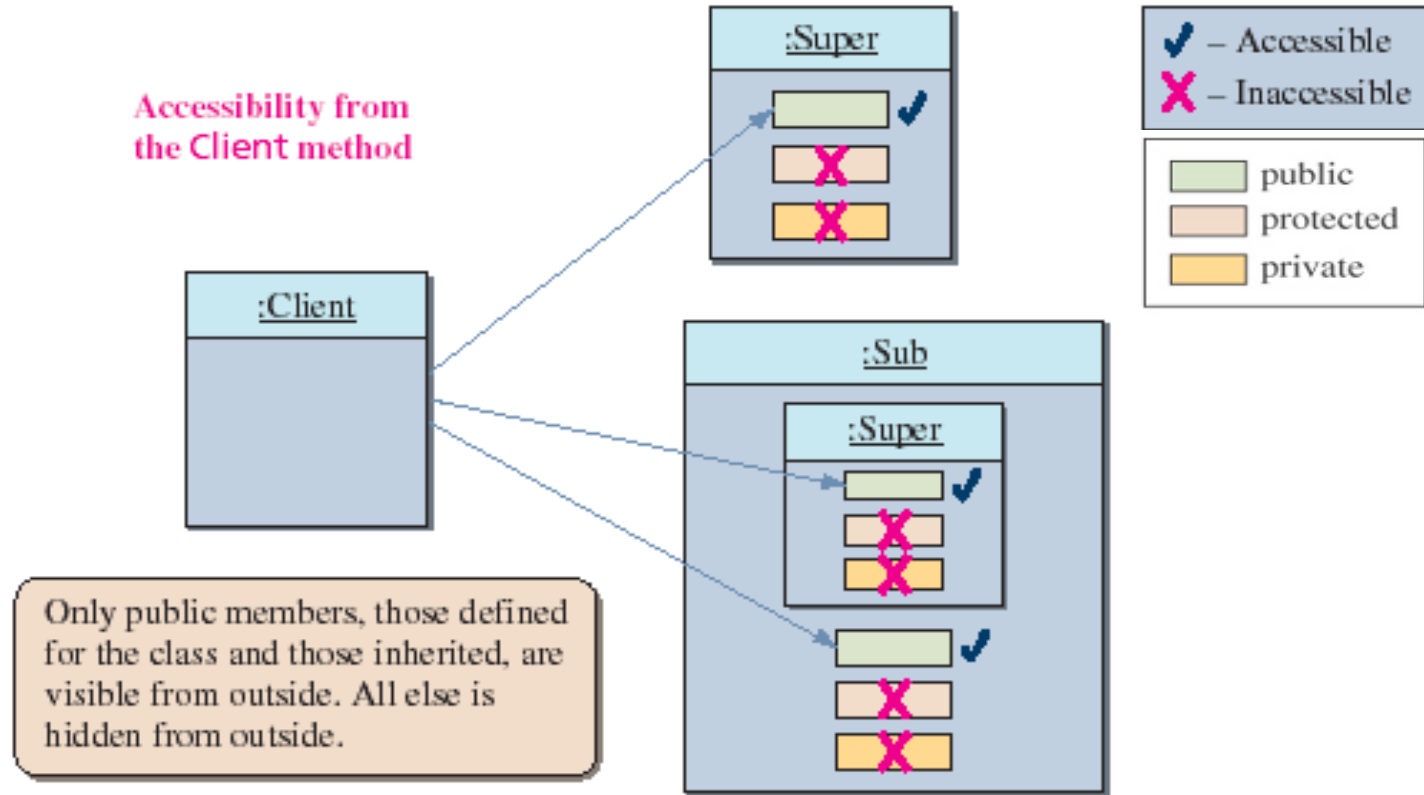
- **package** access
 - The **default**, when no access modifier is given.
 - Can be accessed by all classes in the same package.
- The methods of a sub classes objects can access both the **public** and **protected** members of super class.
- The methods of a sub classes objects cannot access the **private** members of super class.

Inheritance and Member Accessibility

- Visual representation of inheritance to illustrate data member accessibility.



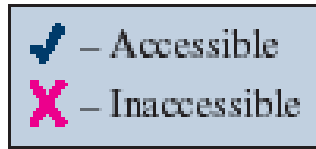
The Effect of Three Visibility Modifiers



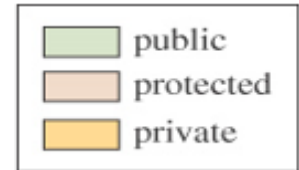
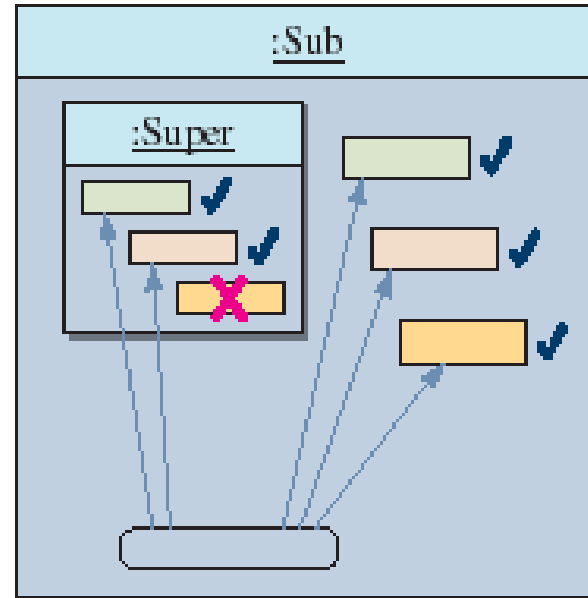
Accessibility of Super from Sub

- Everything except the private members of the Super class is visible from a method of the Subclass.

Accessibility from a method of the Sub class



From a method of Sub, everything is visible except the private members of its superclass.



Inheritance and Constructors

- Unlike members of a superclass, constructors of a superclass are ***not inherited*** by its subclasses.
- You must define a constructor for a class or use the default constructor added by the compiler.
- The statement `super () ;` calls the superclass's constructor.
- A constructor of the base class can be invoked within the definition of a derived class constructor.
- **super keyword** MUST BE THE FIRST statement in constructor

Using Methods of the Superclass in a Subclass

- Subclass also inherits the methods of the superclass.
- The subclass can give some of its methods the same signature as given by the super class.
- Example:
 - *Superclass* contains a method: `toString()`, that returns the values of the data members of *superclass*.
 - *Subclass* contains data members in addition to data members inherited from the *superclass*.
 - A method is include in the subclass that returns the data members of subclass
 - This method can be named in any kind of name.
 - However, in the class *subclass*, this method also can be named as `toString()` (same name used by *super class*)
 - This is called **overriding** the method of the superclass.

Using Methods of the Superclass in a Subclass

- To override a public method of the superclass in the subclass, the corresponding method in the subclass must have same:
 - Name
 - Type
 - Formal parameter list
- If the corresponding method in the *superclass* and the *subclass* has the **same name but different parameter lists**, this method is called **overloading** in the subclass (is allowed).
- When *override* or *overload* a method of the *superclass* in the *subclass*, you must know **how to specify a call to the method** of the *superclass* that has the same name as that used by a method of the subclass.

```
g id){
```

```
urn name;}
```

```
rn empId;}
```

tion of the object.

```
+name + "Emp id= " + empId;
```

```
public class FullTime extends Employee {
    // Data members
    private double base;
    private double allow;

    public FullTime(){
        super(); // call superclass default constructor
        base = 0;
        allow = 0;
    }

    public FullTime(String nm, long id, double base, double allow){
        super(nm, id); // call superclass normal constructor
        this.base = base;
        this.allow = allow;
    }
    // Returns the string representation of the object
    public String toString(){
        return "FullTime[super= " + super.toString() +
            "], base sal: " + base + " Allow: " + allow;
    }
}
```

Constructors and Inheritance (Sample Application Program)

```
/* Author : Hanapi
   Purpose: The main program test the Superclass Employee and Subclass FullTime */
import java.util.*;
public class Worker {
    public static void main (String args[]) {
        Scanner inp = new Scanner(System.in);
        System.out.println("Enter Number of employee: ");
        int n = inp.nextInt();

        // declare array of FullTime objects
        FullTime[] ft = new FullTime[n];
        for (int i=0 ; i < ft.length; i++) {
            // input data for the employees
            System.out.println(" Enter name, id, salary, allowance : ");
            String name = inp.next();
            long id = inp.nextLong();
            double bs = inp.nextDouble();
            double all = inp.nextDouble();

            // create and store data for objects
            ft[i] = new FullTime(name, id, bs, all);
        }

        for (int i=0 ; i < ft.length; i++) {
            System.out.println("The detail of Fulltime emp : " ft[i].toString());
            System.out.println(" The salary : " + ft[i].calSalary());
        }
    }
}
```

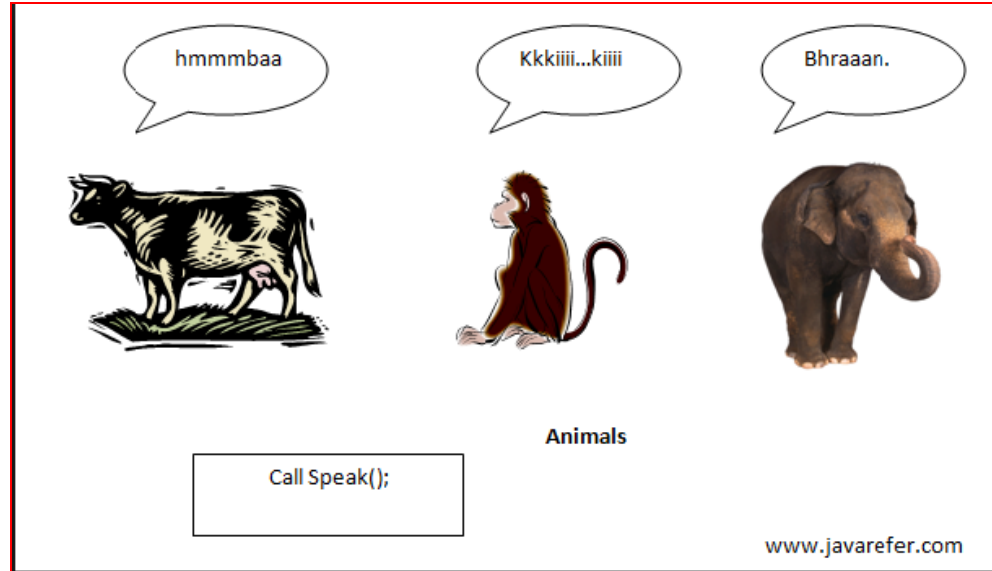
Polymorphism

Polymorphism

- **Polymorphism** means “having many forms”.
- **Polymorphism** is the ability for objects of different classes related by inheritance to respond differently to the same method call.
- A **polymorphic** program is a program that allows a single variable to refer to objects from different classes.

Polymorphism

For example: Given a super class `Animals`, polymorphism enables the programmer to define different `speak()` methods for any sub classes such as `Cow`, `Monkey`, `Elephant`. Calling the `speak()` method of the sub classes objects will return the correct results.



Polymorphism

- In OOP, **polymorphism** allows a single variable to refer to objects from different subclasses in the same inheritance hierarchy. When the variable is used with the notation object reference.method() to invoke a method, exactly which method is run depends on the object that the variable object reference currently refers to and not the object references itself.

Note:

- In **inheritance**, designing a **generic super class** is important. **Separate arrays are used for each sub class.**
- In **polymorphism**, designing a **generic abstract super class** is important. A **common super class array is used for sub classes objects.** Instead of using separate arrays for each sub classes, polymorphism simplifies this task by allowing us to maintain a single array only. Even if we later add new subclasses under the super class, we will still use the same single array.

Polymorphism examples

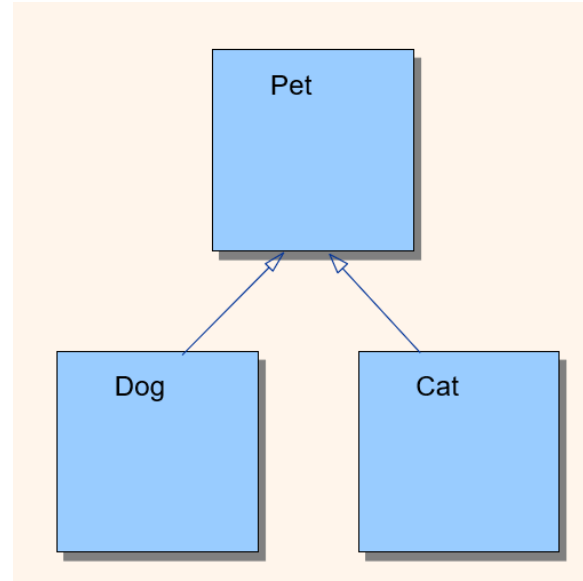
- if Cat and Dog are subclasses of Pet, then the following statements are valid:

```
Pet myPet;
```

```
myPet = new Dog(); //legal
```

```
. . .
```

```
myPet = new Cat(); //legal
```



Polymorphism examples(cont'd)

- if UndergraduateStudent and GraduateStudent classes are subclasses of Student, then the following statements are valid.

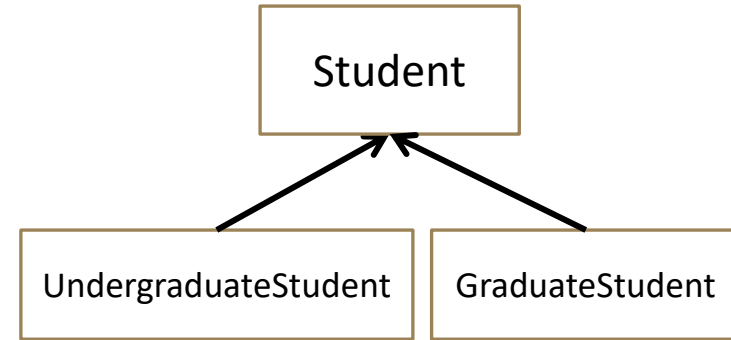
```
// an array of super class reference  
Student roster[] = new Student[40];
```

```
...
```

```
// points to sub class GraduateStudent  
roster[0] = new GraduateStudent();
```

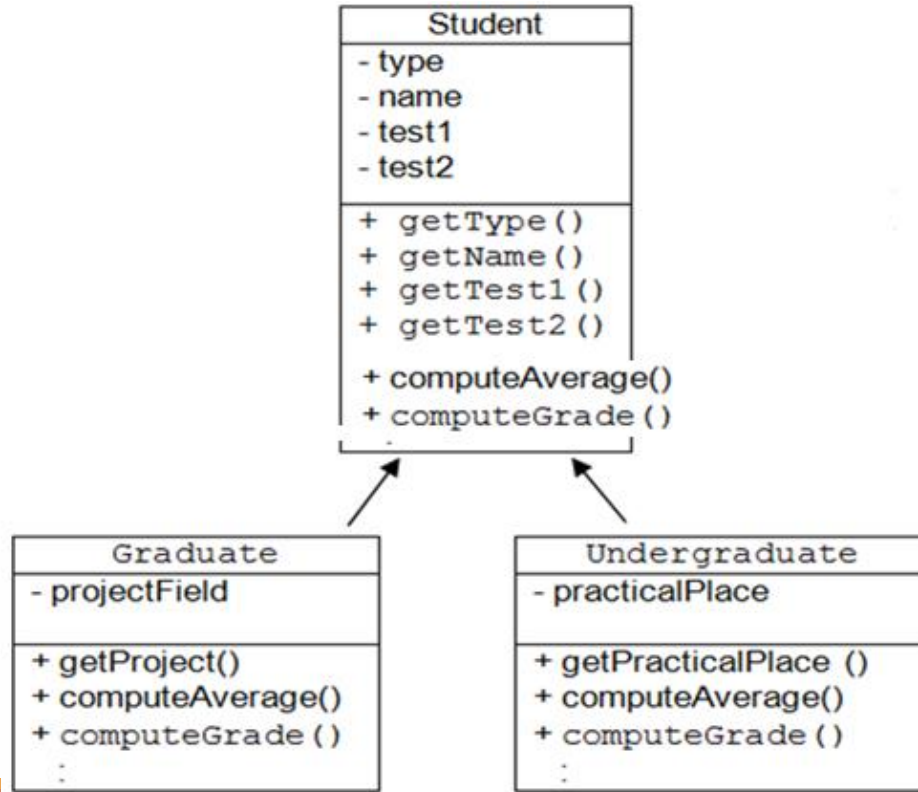
```
// points to sub class undergraduateStudent  
roster[1] = new UndergraduateStudent();
```

```
// points to sub class undergraduateStudent  
roster[2] = new UndergraduateStudent();
```



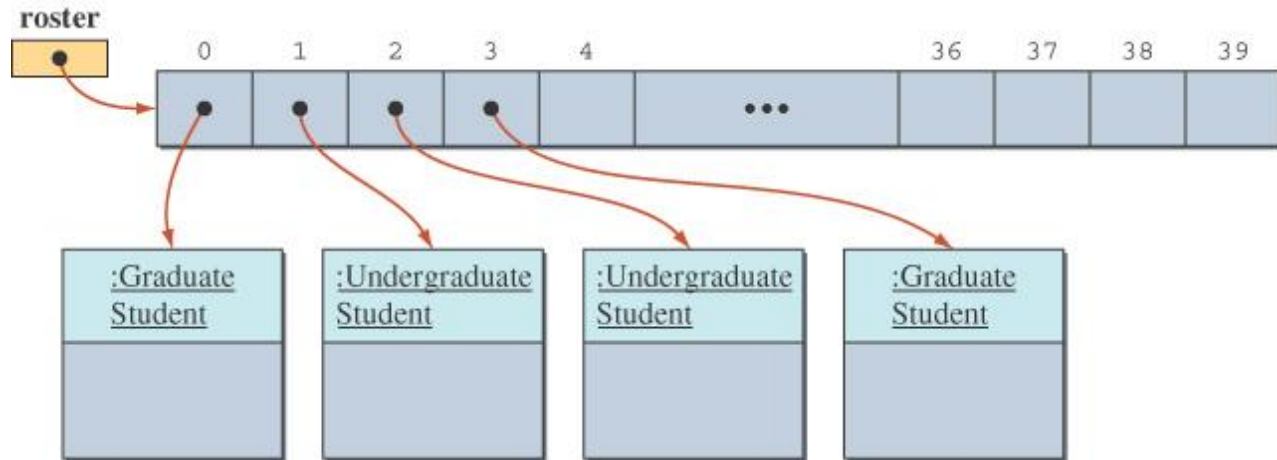
- Note:** We can declare an array of the super class to hold objects of the different sub classes.

Class Diagram of abstract class



Polymorphism examples(cont'd)

- The **roster** array with elements referring to instances of **GraduateStudent** or **UndergraduateStudent** classes.



Any object that belongs in the same hierarchy of sub classes can fit into any slot of the array of the super class as shown in the diagram above

Sample Polymorphic Message

- To compute the course grade using the roster array:

```
for (int i = 0; i < roster.length; i++)  
{  
    roster[i].computeGrade();  
}
```

- If roster[i] refers to a GraduateStudent, then the computeGrade() method of the GraduateStudent class is executed.
- If roster[i] refers to an UndergraduateStudent, then the computeGrade() method of the UndergraduateStudent class is executed.

An Abstract Super Class

- Abstract classes are placeholders that help organize information and provide a base for polymorphic references.
- An *abstract class* is a class.
 - defined with the modifier **abstract** OR
 - that contains at least one abstract method OR
 - that does not provide an implementation of an inherited abstract method.
 - No instances(objects) can be created from an abstract class.

Abstract Method

An *abstract method* is a method:

- with the keyword **abstract**
- that ends with a semicolon instead of a method body.
- that contains no implementation – no body.
- **Note:** Private methods and static methods may not be declared **abstract**.

Rules about abstract classes

- We can create reference to an abstract super class but not instantiate it. It is a special kind of class that is solely meant to be sub classed (inherited form).
- A subclass of an abstract class can be instantiated **if it overrides all of the abstract methods of its super class** by implementing them.
- Then the sub class is called **concrete** class.
- **Note:** An **Overriden** method: a method with the same signature (same method name, same parameters, same return type).

Rules about abstract classes

- If a subclass of an abstract class does not implement **all** of the abstract methods it inherits, that subclass is itself an abstract class.
- The sub class method must define (override) the super class abstract method that is inherited.
- Note:

An abstract class does **have a constructor** that is (implicitly) invoked by its sub classes, but it cannot be invoked directly. Its constructor usually has protected access modifier.

Abstract Class (Example)

```
abstract class A
{
    abstract void callme();
}
class B extends A
{
    void callme()
    {
        System.out.println("this is callme.");
    }
    public static void main(String[] args)
    {
        B b=new B();
        b.callme();
    }
}
```

output: this is callme.

Abstract Class (Example)

```
abstract class Vehicle
{
    public abstract void engine();
}

public class Car extends Vehicle {
    public void engine()
    {
        System.out.println("Car engine");
        //car engine implementation
    }

    public static void main(String args[])
    {
        Vehicle v = new Car();
        v.engine();
    }
}
```

Output: Car engine

Checking Object Type

- To determine the actual class of an object, the `instanceof` operator is used.

Example:

```
// To count the number of Undergraduate Students
```

```
int undergradCount = 0;
for (int i = 0; i < numberOfStudents; i++) {
    if ( roster[i] instanceof UndergraduateStudent) {
        undergradCount++;
    }
}
```

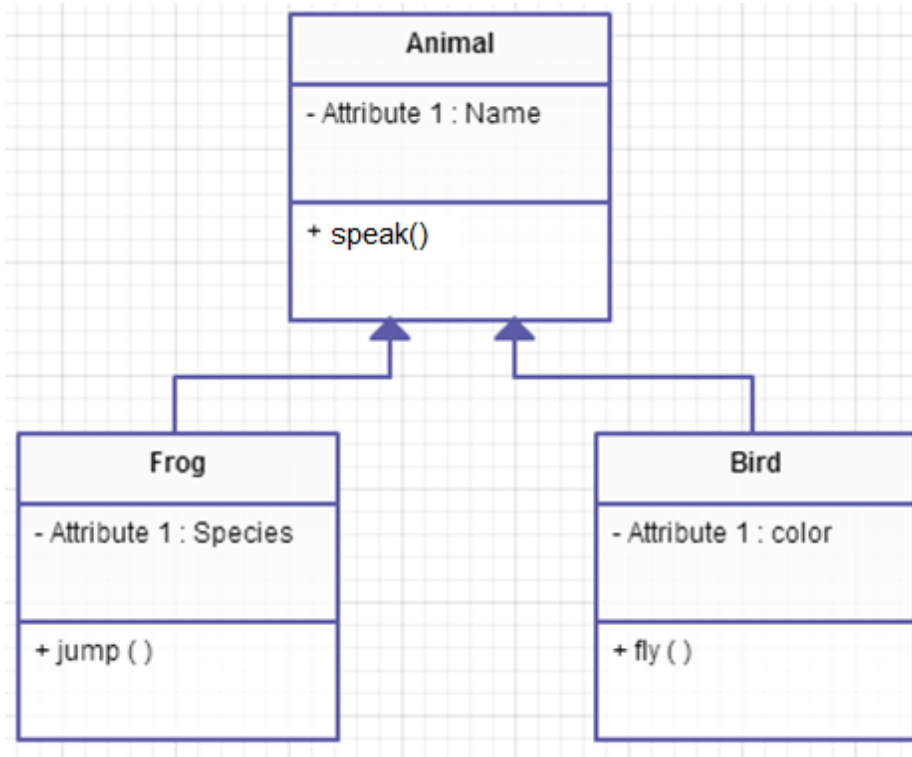
Object Casting

- We will use object casting when we want to **convert object** that we store **in superclass** data type **into the original class** data type.
- We need to convert the type especially when we want **to call method/processor from the original class** which does **not exist in superclass**.

Object Casting: an example

- We have:

Animal a = new Frog();



Object Casting: an example

```
Animal a[] = new Animal[10];
```

```
a[0] = new Frog("fire-bellied", "toad");
```

```
a[1] = new Bird("pigeon", "white");
```

```
a[0].speak(); // valid
```

```
a[1].speak(); // valid
```

```
a[0].jump(); // not valid
```

```
a[1].fly(); // not valid
```

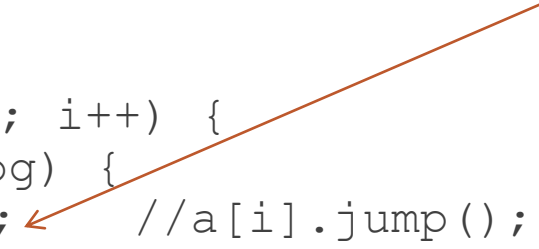
Note:

As you can see, even though a[0] contains object of Frog, we can't call the function jump() because the object is currently stored in Animal data type. Same with a[1]. speak() is valid because it is declared in Animal, so it is guaranteed that all classes that inherit from it will contain the method.

Object Casting: an example

- To **call the specific method in subclass** when the subclass object is stored in superclass data type, we **need to cast the object into subclass** data type.

```
for (int i=0; i < a.length; i++) {  
    if (a[i] instanceof Frog) {  
        Frog f = (Frog)a[i]; //a[i].jump();  
        f.jump(); // valid  
    }  
    else if (a[i] instanceof Bird) {  
        Bird b = (Bird)a[i];  
        b.fly(); // valid  
    }  
}
```



Interfaces

- interface types is used to make code more reusable.
- An interface type is similar to a class, but there are several important differences:
 - All methods in an interface type are abstract; they don't have an implementation.
 - All methods in an interface type are automatically public.
 - An interface type does not have instance fields.

Defining an Interface

```
public interface InterfaceName
{
    //method signatures
}
```

Example:

```
public interface EmpInterface
{
    double calTax();
}
```

Implementing an Interface

```
public class Employee implements EmpInterface {  
    public double calTax()  
    {  
        // method definition  
    }  
}
```

Inheritance versus Interface

- The Java interface is used to share common behavior (only method headers) among the instances of different classes.
- Inheritance is used to share common code (including both data members and methods) among the instances of related classes.
- Use the Java interface to share common behavior.
- Use inheritance to share common code.

Abstract Class versus Interface

Abstract class	Interface
Abstract class is a class which contain one or more abstract methods, which has to be implemented by its sub classes.	Interface is a Java Object containing method declaration but no implementation. The classes which implement the Interfaces must provide the method definition for all the methods.
Abstract class is a Class prefix with an abstract keyword followed by Class definition.	Interface is a pure abstract class which starts with interface keyword.
Abstract class can also contain concrete methods.	Whereas, Interface contains all abstract methods and final variable declarations.
Abstract classes are useful in a situation that Some general methods should be implemented and specialization behavior should be implemented by child classes.	Interfaces are useful in a situation that all properties should be implemented.

References

- Deitel, *Java How To Program*, 5th Edition, Prentice Hall, 2003
- Horstmann, *Java Concepts*, 4th Edition, John Wiley, 2005
- Malik D.S, Nair P.S. *Java Programming: From Problem Analysis To Program Design*. Course Technology 2003.
- Savitch W. *Absolute Java*. 2nd Edition. Addison Wesley 2006.
- Wu, *An Introduction To Object-Oriented Programming With Java*, 4th Edition, McGraw-Hill, 2006