CMSC 132: Object-Oriented Programming II

Inheritance
Inheritance

• Classes can be *derived* from other classes, thereby *inheriting* fields and methods from those classes.

• A class that is derived from another class is called a *subclass* (also a *derived class*, *extended class*, or *child class*).

• The class from which the subclass is derived is called a *superclass* (also a *base class* or a *parent class*).

• Derived (Child) class can be base (parent) class
Inheritance

**Motivation**: In real life objects have a hierarchical structure:

![Diagram showing inheritance hierarchy: Shape (Circle, Triangle, Rectangle) and further subdivisions such as Right-Triangle, Equilateral-Triangle, Square.](image-url)
Inheritance

- Define a general class
- Later, define specialized classes based on the general class
- These specialized classes inherit properties from the general class
Inheritance cont.

- What are some properties of a Person?
  - name, height, weight, age

- How about a Student?
  - ID, major, gpa

- Does a Student have a name, height, weight, and age?
  - Student inherits these properties from Person
is-a relationship

- This inheritance relationship is known as an is-a relationship
  - A Grad student is a Student
  - A Student is a Person.

- Is a Person a Student? – Not necessarily!
Why inheritance is useful

- Enables you to define shared properties and actions once
- Derived classes can perform the same actions as base classes without having to redefine the actions
- If desired, the actions can be redefined – method overriding
public class Person {
    private String name;
    public Person(){
        name = "noname";
    }
    public Person(String name){
        this.name = name;
    }
    public void setName(String newName){
        name = newName;
    }
    public String getName(){
        return name;
    }
    @Override
    public String toString(){
        return "Name:" + name;
    }
}

<table>
<thead>
<tr>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>-name</td>
</tr>
<tr>
<td>+Person()</td>
</tr>
<tr>
<td>+Person(String name):void</td>
</tr>
<tr>
<td>+setName(String name) : void</td>
</tr>
<tr>
<td>+getName() : String</td>
</tr>
</tbody>
</table>
public class Student extends Person{
    private int id;
    public Student() {
        id = 0;
    }
    public Student(String name, int id) {
        super(name);
        this.id = id;
    }
    public void setID(int idNumber) {
        id = idNumber;
    }
    public intgetID(){
        return id;
    }
    @Override
    public String toString(){
        return "Id:"+ id +"\tName:" +
               getName();
    }
}

CMSC 132 Summer 2017
Dissecting the Student Class

- **Extends**: To specify that Student is a *derived class* (subclass) of Person we add the descriptor “extends” to the class definition:
  - `public class Student extends Person { ... }

- Notice that a Student class
  - Inherits everything from the Person class
  - A Student IS-A Person (wherever a Person is needed, we can use a Student).

- `super( )`: When initializing a new Student object, we need to initialize its *base class* (or superclass). This is done by calling `super( ... )`. For example, `super( name)` invokes the constructor `Person( name)`
  - `super( ... )` must be the *first statement* of your constructor
  - If you **do not** call `super( )`, Java will automatically invoke the base class’s **default constructor**
  - What if the base class’s default constructor is **undefined**? Error
  - You must use “`super( ... )`”, not “`Person( ... )`”.

CMSC 132 Summer 2017
Memory Layout and Initialization Order

- When you create a new derived class object:
  - Java allocates space for both the base class instance variables and the derived class variables
  - Java initializes the base class variables first, and then initializes the derived class variables

Example:

Person ted = new Person( "Ted Goodman");
Student bob = new Student( "Bob Goodstudent", 100);
Inheritance

- **Inheritance**: Since Student is derived from Person, a Student object can invoke any of the Person methods, it *inherits* them

  ```java
  Student bob = new Student( "Bob Goodstudent", 100);
  String bobsName = bob.getName( ) ;
  bob.setName( "Robert Goodstudent" );
  System.out.println( "Bob's new info: " + bob.toString( ) );
  ```

- **A Student “is a” Person**:
  - By inheritance a Student object is also a Person object. We can use a Student reference anywhere that a Person reference is needed

    ```java
    Person robert = bob;  // Okay: A Student is a Person
    ```

  - We cannot reverse this. (A Person need not be a Student.)

    ```java
    Student bob2 = robert;  // Error! Cannot convert Person to Student
    ```
Overriding Methods

- **New Methods**: A derived class can define *entirely new* instance variables and new methods (e.g. gpa and getGpa())
- **Overriding**: A derived class can also *redefine existing* methods

```java
public class Person {
    ...
    public String toString() { ... }
}
public class Student extends Person {
    ...
    public String toString() { ... }
}
Student bob = new Student( "Bob Goodstudent", 100);
System.out.println("Bob's info: " + bob);
```

Since bob is of type Student, this invokes the Student toString()
Overriding and Overloading

- Don’t confuse method **overriding** with method **overloading**.
  
  **Overriding**: occurs when a derived class defines a method with the **same name** and **parameters** as the base class.

  **Overloading**: occurs when two or more methods have the **same name**, but have **different parameters** (different signature).

**Example:**

```java
public class Person {
    public void setName(String n) { name = n; }
    ...
}

public class Faculty extends Person {
    public void setName(String n) {
        super.setName("The Evil Professor " + n);
    }
    public void setName(String first, String last) {
        super.setName(first + " " + last);
    }
}
```

The base class defines a method `setName( )`

Overriding: Same name and parameters; different definition.

Overloading: Same name, but different parameters.
Quiz 1: Output of following program

class Test {
    int i;
}
class Main {
    public static void main(String args[]){
        Test t;
        System.out.println(t.i);
    }
}

A. 0
B. garbage value
C. compiler error
D. runtime error
Quiz 1: Output of following program

class Test {
    int i;
}
class Main {
    public static void main(String args[]){
        Test t;
        System.out.println(t.i);
    }
}

A. 0
B. garbage value
C. compiler error: variable not initialized
D. runtime error
Quiz 2: Output of following program

class Test {
    int i;
}
class Main {
    public static void main(String args[]) {
        Test t = null;
        System.out.println(t.i);
    }
}

A. 0
B. garbage value
C. compiler error
D. runtime error
Quiz 2: Output of following program

class Test {
    int i;
}
class Main {
    public static void main(String args[]) {
        Test t = null;
        System.out.println(t.i);
    }
}

A. 0
B. garbage value
C. compiler error
D. runtime error: Null pointer exception
Quiz 3: Output of following program

```java
class Base{
    void display() {System.out.print("Base ");}
}
class Child extends Base{
    void display(){System.out.print("Child ");}
}
Base b= new Base();
Child c = new Child ();
Base ref = b;
ref.display();
ref = c;
ref.display();
```

A. Compilation error
B. Base Child
C. Child Base
D. Runtime error
Quiz 3: Output of following program

class Base{
    void display() {System.out.print("Base ");}
}
class Child extends Base{
    void display(){System.out.print("Child ");}
}
Base b= new Base();
Child c = new Child ();
Base ref = b;
ref.display();
ref = c;
ref.display();

A. Compilation error
B. Base Child
C. Child Base
D. Runtime error
Quiz 4: Output of following program

class Test{
    int a = 1;
    int b = 2;
    Test func(Test obj){
        Test obj3 = new Test();
        obj3 = obj;
        obj3.a = obj.a++ + ++obj.b;
        obj.b = obj.b;
        return obj3;
    }
    main(){
        Test obj1 = new Test();
        Test obj2 = obj1.func(obj1);
        System.out.print(obj1.a + "," + obj1.b + "",");
        System.out.print(obj2.a + "," + obj2.b + "",");
    }
}

A. 1,2,4,3,  
B. 4,3 4,3  
C. Error
Quiz 4: Output of following program

class Test{
    int a = 1;
    int b = 2;

    Test func(Test obj){
        Test x = new Test();
        x = obj;
        x.a = obj.a++ + ++obj.b;
        obj.b = obj.b;
        return x;
    }

    main(){
        Test obj1 = new Test();
        Test obj2 = obj1.func(obj1);
        System.out.print(obj1.a + "," + obj1.b + ",");
        System.out.print(obj2.a + "," + obj2.b + ",");
    }
}

A. 1,2,4,3,
B. 4,3 4,3
C. Error
Overriding Variables: Shadowing

• We can override methods, can we override instance variables too?
• **Answer:** Yes, it is possible, but **not recommended**
  • Overriding an instance variable is called **shadowing**, because it makes the base instance variables of the base class inaccessible. (We can still access it explicitly using `super.varName`).

```java
public class Person {
    public class Staff extends Person {
        String name;
        // ...
        // name refers to Staff’s name
    }
}
```

• This can be **confusing** to readers, since they may not have noticed that you redefined name. Better to just pick a new variable name.
Shadowing example

class Base {
    public int x;
    public Base(){x = 10;}
    public void foo(){return x;}
}

class Derived extends Base {
    public int x;
    public Derived(){ x = 20;}
    public void foo(){return (x + "\t" + super.x);}
}

Derived d = new Derived();
d.foo();
Shadowing example

class Base {
    public int x;
    public Base(){x = 10;}
    public void foo(){return x;}
}

class Derived extends Base {
    public int x;
    public Derived(){ x = 20;}
    public void foo(){return (x + "\t" + super.x);}
}

Derived d = new Derived();
Base b = d;
b.foo();
Shadowing example

class Base {
    public int x;
    public Base(){x = 10;}
    public void foo(){return x;}
}

class Derived extends Base {
    public int x;
    public Derived(){ x = 20;}
    public void foo(){return (x + \"\t\" + super.x);}
}

Derived d = new Derived();
Base b = d;
d.x;
b.x;
super and this

- **super**: refers to the base class object
  - We can invoke any base class constructor using `super( ... )`.
  - We can access data and methods in the base class (Person) through `super`. E.g., `toString()` and `equals()` invoke the corresponding methods from the Person base class, using `super.toString()` and `super.equals()`.

- **this**: refers to the current object
  - We can refer to our own data and methods using “this.” but this usually is not needed
  - We can invoke any of our own constructors using `this( ... )`. As with the super constructor, this can only be done within a constructor, and must be the first statement of the constructor. Example:

```java
public Fraction(int n) {
    this(n,1);
}
```
Memory Layout

class Base{
    private int a;
    protected int b;
    protected int c;
    protected void m1(){}
    public void m2(){}
}

class Child extends Base{
    private int d;
    public void m1(){}
    public void m3(){}
}

The Java Virtual Machine does not mandate any particular internal structure for objects.
class Base{
    private int a;
    protected int b;
    protected int c;
    protected void m1(){}
    public void m2(){}
}

class Child extends Base{
    private int d;
    public void m1(){}
    public void m3(){}
}
Memory Layout

class Base{
    private int a;
    protected int b;
    protected int c;
    protected void m1(){}
    public void m2(){}
}

class Child extends Base{
    private int d;
    public void m1(){}
    public void m3(){}
}
Memory Layout

class Base{
    private int a;
    protected int b;
    protected int c;
    protected void m1(){}
    public void m2(){}
}

class Child extends Base{
    private int d;
    public void m1(){}
    public void m3(){}
}

Each class has one vtable.

All objects of the this class shares the vtable.