Inheritance

Cheng, Wei  COMP110-001  June 10, 2014
Today

- Review of Method Overloading
- Inheritance
Overloading

• Using the same method name for two or more methods *within the same class*

• We have seen this for constructors already

• Parameter lists must be different
  – public double average(double n1, double n2)
  – public double average(double n1, double n2, double n3)

• Java knows what to use based on the number and types of the arguments
Method *signature*

- A method’s name and the number and types of its parameters
- *signature* does NOT include return type
- Cannot have two methods with the same signature in the same class
• Overloading and automatic type conversion

• Imagine we have this constructor defined:
  
  ```java
  public Pet(double initialWeight)
  ```

• We create a Pet like this:
  
  ```java
  Pet myPet = new Pet(35);
  ```

• What happens?
• Imagine we have these two constructors defined:
  ```java
  public Pet(int initialAge)
  public Pet(double initialWeight)
  ```

• We create a Pet like this:
  ```java
  Pet myPet = new Pet(35);
  ```

• What happens?
  – We create a pet with age 35, instead of weight 35.0
We have discussed before how classes of objects can have relationships.
• Define a general class
• Later, define specialized classes based on the general class
• These specialized classes *inherit* properties from the general class
Inheritance

• What are some properties of a Person?
  – Name, height, weight, age

• How about a Student?
  – ID, major

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

- This inheritance relationship is known as an *is-a relationship*
- A Doctoral student *is a* Grad student
- A Grad student *is a* Student
- A Student *is a* Person
- Is a Person a Student?
  - Not necessarily!
Our general class is called a **base class**
  - Also called a **parent class** or a **superclass**

Examples:
  - Person, Transportation
A specialized class that inherits properties from a base class is called a **derived class**

– Also called a **child class** or a **subclass**

**Examples:**

– Student *is-a* Person
– Employee *is-a* Person
– Car *is-a* form of Transportation
– Animal *is-a* form of Transportation
Child (derived) classes can be parent (base) classes

- Student is a child class of Person
- Student is also the parent class of Undergrad and Grad

```
Person
  
  Student
    
    Undergrad
    
    Grad
    
    Faculty
    
    Staff

Employee

Masters

Doctoral

Nondegree
```
Why is inheritance 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 – more on this later
public class Person
{
    private String name;
    public Person()
    {
        name = "No name yet";
    }
    public void setName(String newName)
    {
        name = newName;
    }
    public String getName()
    {
        return name;
    }
}
public class Student extends Person {
    private int id;
    public Student() {
        super();
        id = 0;
    }
    public Student(String stdName, int idNumber) {
        setName(stdName);
        setID(idNumber);
    }
    public void setID(int idNumber) {
        id = idNumber;
    }
    public intgetID() {
        return id;
    }
}
The `extends` keyword

```java
public class Derived_Class_Name extends Base_Class_Name {
    Declaration_of_Added_Instance_Variables
    Definitions_of_Added_And_Overridden_Methods
}

public class Student extends Person {
    // stuff goes here
}
```

- A derived (child) class inherits the `public` instance variables and `public` methods of its base (parent) class
private vs. public

- **private** instance variables and **private** methods in the base class are NOT inherited by derived classes

- This would not work:

```java
public Student(String stdName, int idNumber) {
    name = stdName; // ERROR! name is private to Person
    setID(idNumber);
}
```
private vs. public

• **private** instance variables of the base class CAN be accessed by derived classes using the base class’ **public** methods

• This works:

```java
public Student(String stdName, int idNumber)
{
    setName(stdName); // OK! setName is a public method in Person
    setID(idNumber);
}
```
The `super` keyword

- Used to call a constructor of the base class (remember, a base class is also known as a `superclass`)
- More details later
What if the class Person had a method called printInfo?

```java
public class Person {
    // a bunch of other stuff
    // ...
    public void printInfo() {
        System.out.println(name);
    }
}
```
What if the class Student *also* had a method called printInfo?

```java
public class Student extends Person
{
    // a bunch of other stuff
    // ...
    public void printInfo()
    {
        System.out.println("Name: " + getName());
        System.out.println("ID: " + getID());
    }
}
```
Overriding methods

- If Student inherits the printInfo() method and defines its own printInfo() method, it would seem that Student has two methods with the same signature
  - We saw before that this is illegal, so what’s the deal?
Overriding methods

• Java handles this situation as follows:
  – If a derived class defines a method with the same name, number and types of parameters, and return type as a method in the base class, the derived class’ method overrides the base class’ method
  – The method definition in the derived class is the one that is used for objects of the derived class
Both Person and Student have a printInfo() method

Student std = new Student("John Smith", 37183);
std.printInfo(); // calls Student’s printInfo method,
                 // not Person’s

Output would be:
Name: John Smith
ID: 37183
Overriding vs. overloading

• If a derived class defines a method of the same name, same number and types of parameters, and same return type as a base class method, this is **overriding**

• You can still have another method of the *same name* in the same class, as long as its number or types of parameters are different: **overloading**
The class Object

- Every class in Java is derived from the class Object
  - Every class in Java is an Object
• Object has several public methods that are inherited by subclasses

• Two commonly overridden Object methods:
  – toString
  – equals
• There is a version of System.out.println that takes an Object as a parameter. What happens if we do this?

Person p = new Person();
System.out.println(p);

• We get something like:

Person@addbf1

• The class name @ hash code
Every class has a `toString` method, inherited from `Object`:

```java
public String toString()
```

Intent is that `toString` be overridden, so subclasses can return a custom String representation.
When we call System.out.println() on an object...

- the object’s toString method is called
- the String that is returned by the toString method is printed

```java
public class Person {
    private String name;
    public Person(String name) {
        this.name = name;
    }
    public String toString() {
        return "Name: " + name;
    }
}

public class Test {
    public static void main(String[] args) {
        Person per = new Person("Apu");
        System.out.println(per);
    }
}
```

Output:

```
Person@addbf1
Name: Apu
```
What if we have a derived class?

(Assume the Person class has a getName method)

```java
public class Student extends Person {
    private int id;
    public Student(String name, int id) {
        super(name);
        this.id = id;
    }
    public String toString() {
        return "Name: " + getName() + ", ID: " + id;
    }
}
```

```java
public class Test {
    public static void main(String[] args) {
        Student std = new Student("Apu", 17832);
        System.out.println(std);
    }
}
```

Output:
Name: Apu, ID: 17832
What if we have a derived class?

• Would this compile?

```java
public class Test {
    public static void main(String[] args) {
        Person per = new Student("Apu", 17832);
        System.out.println(per);
    }
}
```

• Yes. What is the output?

```
Output:
Name: Apu, ID: 17832
```

• Automatically calls Student’s toString method because `per` is of type Student