

Inheritance

Cmput 114 Section A3, Fall 2005
Lecture 25
Department of Computing Science
University of Alberta

November 24, 2005

About This Lecture

- In this lecture we will learn about inheritance.
- Inheritance provides us with a way of taking advantage of similarities between objects from different classes and building new classes that are extensions of existing classes.

© D. Szafron & C. Jones 1999, 2005



2

Outline

- What is inheritance
- Subclasses and Superclasses
- Type Inheritance
- Method inheritance
- Representation inheritance
- Constructor inheritance
- Why use inheritance
- Polymorphism

© D. Szafron & C. Jones 1999, 2005



3

Inheritance in the Real World

- How is a student like a person?
- Well, every student is a person!
- Students have all of the "properties" of persons, plus some others.
- For example, every person has a name and an age and so does every student.
- However, not every person is a student.
- Every student has a student id and a grade point average, that other persons don't have.

© D. Szafron & C. Jones 1999, 2005



4

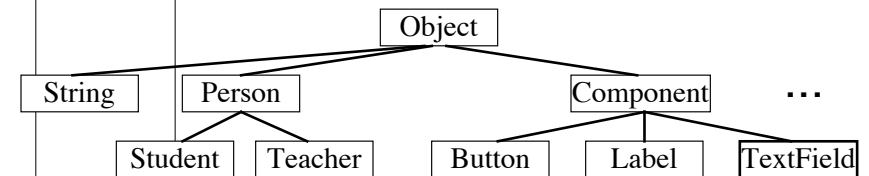
Subclasses and Superclasses

- In Java, we model a person by a Person class.
- In Java, we model a student as by a Student class.
- Since a student is like a person with extra properties, we say the class Student is a subclass of the class Person.
- We also say that Person is a superclass of Student.



The Java Inheritance Tree

- In general, Person can have other subclasses as well, say Teacher.
- We put all the classes in an inheritance tree with class Object as the root.
- We draw the tree with the root at the top.



Type Inheritance 1

- We say that a subclass inherits all of the messages from its superclass.
- Any message that can be sent to an instance of a class can also be sent to an instance of its subclasses.
- However, you can add additional instance messages and static messages to a subclass.



Type Inheritance 2

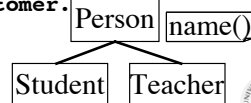
- If you declare the type of a variable to be some class, it can then be bound to an instance of that class or any subclass.
- If the type of a message parameter or the return type of a message is a class, you can use any subclass as well.
- The property of being able to use an instance of a subclass, wherever you can use an instance of a class is called substitutability.



Type Inheritance Example 1

- Assume that we are defining a class called Store.
- Assume that we have already defined a class called Person, with a message called name() and two subclasses: Student and Teacher.
- Assume that we have defined a message in this "Store" class called register that takes a Person as a parameter:

```
public void register(Person aPerson) {
    // Register the given Person as a customer.
```



Type Inheritance Example 2

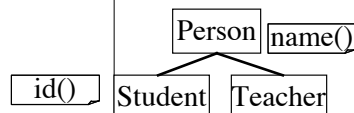
- Here is a method that creates a Person, Student or Teacher customer, depending on a char parameter.

```
public Person createCustomer(char aChar, String nameString){
    Person customer;
    if (aChar == 'T')
        customer = new Teacher(nameString);
    else if (aChar == 'S')
        customer = new Student(nameString);
    else
        customer = new Person(nameString);
    System.out.println("Welcome " + customer.name());
    this.register(customer);
    return customer;
}
```



Compiler Protection

- Substitutability does not work the other way!
- You cannot always use an instance of a superclass, wherever you use an instance of a subclass!
- For example, if a Student subclass adds a message called id(), you cannot send this message to a Person object.
- The compiler prevents this by not allowing the message id() to be sent to a receiver object with declared class Person.



```
Person aPerson;
...
aPerson.id();
```

compile error



Instance and Static Variable (Representation) Inheritance

- In Java, a subclass also inherits all of the instance variables and all of the static variables of its superclass.
- If a variable is private, it cannot be accessed directly in the subclass code (even though it is inherited in the subclass).
- If a variable is declared as protected it can be accessed directly in the subclass code.
- A subclass can also add state by defining additional instance and static variables.



Method (Implementation) Inheritance

- In Java, a subclass also inherits all instance and static methods of its superclass, so they do not have to be re-implemented.
- However, you can also override any method if you want to.
- In addition, you can add some code to an inherited method, using the super object reference.



Representation/Implementation on Inheritance - Example 1

```
public class Person {
// Each instance represents a Person.
...
// Instance Variables
protected String    name;
private    Date      birthdate;
...
// Public methods
public void output() {
// Output a representation of myself
    System.out.print("name: " + this.name + " age: ");
    System.out.print(this.age());
}
...
}
```



Representation /Implementation Inheritance - Example 2

```
public class Student extends Person {
// Each instance represents a Student.
...
// Instance Variables
// cannot access birthdate, but can access name
private int    id;

// Public methods
public void output() {
// Output a representation of myself
    super.output();
    System.out.print(" id: ");
    System.out.print(this.id);
}
public int id() {
// return my id.
}
```



Constructor Inheritance and Constructor chaining 1

- Constructors are NOT inherited by a subclass.
- However, if you do not write a "zero" argument constructor, one is created for you (it does nothing).
- If you want to call another constructor in the same subclass, you just use "this()" with the appropriate arguments.
- If you want to call another constructor in the superclass, you just use "super()" with the appropriate arguments.



Constructor Inheritance and Constructor chaining 2

- However, each constructor must “ultimately” call one of the constructors in its superclass.
- This can be done in one of three ways:
 - An explicit call to super() with arguments.
 - A call to another constructor in the subclass using this() with arguments.
 - If neither of these appear as the first statement of the subclass constructor, the compiler inserts an implicit call to the zero argument super constructor super().



Constructors - Example 1

```
public class Person {
    // Each instance represents a Person.
    // Constructors
    public Person() {
        // Set the name "unknown" and birthdate: today
        this.name = "unknown";
        this.birthdate = new Date();
    }

    public Person(String nameString) {
        // Set the given name and birthdate: today
        this(); // do the 0 argument constructor first
        this.name = nameString;
    }
}
```



Constructors - Example 2

```
public class Student extends Person {
    // Each instance represents a Student.
    public Student() {
        // Set the name: "unknown", birthdate: today, id: 0
        this.id = 0; // implicit call to super(); first
    }
    public Student(String nameString) {
        // Set the given name, birthdate: today, id: 0
        super(nameString); // explicit call
        this.id = 0;
    }
    public Student(String nameString, int anInt) {
        // Set the given name and id, birthdate: today
        this(nameString); // or super(nameString)
        this.id = anInt;
    }
}
```



Polymorphism

- When a message expression is evaluated, the class of the receiver object determines which method is called, not the (declared) type of the reference to the receiver.

```
Person aPerson;
aPerson = new Person("Fred");
aPerson.output();
System.out.println();
aPerson = new Student("Fred");
aPerson.output();
```

```
name: Fred age: 0
name: Fred age: 0 id: 0
```



Dynamic Method Binding

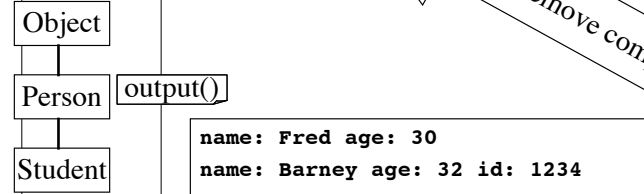
- Can't tell at compile time which instance method to use, the one in subclass or the one in a superclass.

```
Person aPerson;
if ( ...)
    aPerson = new Student();
else
    aPerson = new Person();
aPerson.output();
```



Polymorphism

```
Person aPerson;
...
for (index = 0; index < aVector.size(); index++ {
    aPerson = (Person) aVector.elementAt(index);
    aPerson.output();
    System.out.println();
}
```



Static Method Binding

- The compiler can tell at compile time which static method to use.
- Look at the class referenced in the static method call. If no static method exists for the class, look at the superclass (at compile time).

```
Person.kind(); <-- use static method in Person
Student.kind(); <-- if a static method exists in
the Student class use it, else use a static
method in the Person class.
```



The instanceof operator

- The instanceof operator can be used to check the type of an object at runtime.
- Use it to compare an object against a particular type.
- Returns a boolean:
 - true if the object is of the specified class or subclass
 - false otherwise
- Use it to decide whether an object can respond to a message or not.



instanceof Example

```
Object anObject;
Person aPerson;
...
for (index = 0; index < aVector.size(); index++ {
    anObject = aVector.elementAt(index);
    if (anObject instanceof Person) {
        aPerson = (Person) anObject;
        aPerson.output();
        System.out.println();
    }
    else
        System.out.println(anObject);
}
```

```
name: Fred age: 30
name: Barney age: 32 id: 1234
"This is a String."
```



The final keyword

- If a method has the final keyword, it cannot be overridden.
- For example, Button is a subclass of Component, but the method dispatchEvent() in Component cannot be overridden since it is final.
- A final class cannot be subclassed
- For example, Integer and String are final classes so they cannot have subclasses.

