

CSE 143



Object & Class Relationships – Inheritance

Reading: Ch. 9, 14

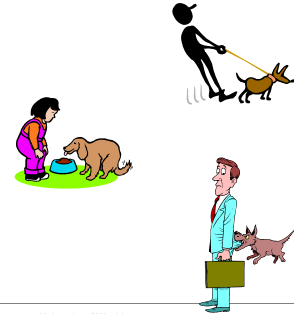
9/29/2004

(c) 2001-4, University of Washington

02-1

Relationships Between Real Things

- Man walks dog
- Dog strains at leash
- Dog wears collar
- Man wears hat
- Girl feeds dog
- Girl watches dog
- Dog eats food
- Man holds briefcase
- Dog bites man



9/29/2004

(c) 2001-4, University of Washington

02-2

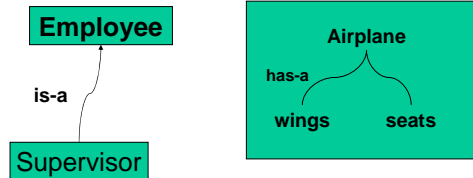
Common Relationship Patterns

- A few types of relationships occur extremely often
 - **IS-A**: a supervisor is an employee (and a taxpayer and a sister and a skier and ...)
 - **HAS-A**: An airplane has seats (and lights and wings and engines and...)
- These are so important and common that programming languages have special features to model them
 - Some of these you know (maybe without knowing you know)
 - Some of them we'll cover carefully in this course, starting now, with **inheritance**

9/29/2004

(c) 2001-4, University of Washington

02-3



9/29/2004

(c) 2001-4, University of Washington

02-4

Composition: "has a"

- Classes and objects can be related in several ways
- One way: *composition, aggregation, or reference*
- Dog has-a owner, dog has-a age, dog has-a name, etc.
- In java: one object refers to another object
 - via an instance variable

```
public class Dog {
    private String name; // this dog's name
    private int age; // this dog's age
    private Person owner; // this dog's owner
    private Dog mother, father; // this dog's parents
    private Color coatColor; // etc, etc.
}
```



- Composition: One can think of the dog as "composed" of various objects

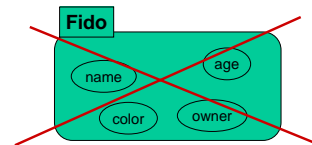
9/29/2004

(c) 2001-4, University of Washington

02-5

Picturing the Relationships

- Dog Fido; //might be 6 years old, brown, owned by Marge, etc.
- Dog Apollo; //might be 2 years old, no owner, etc.
- In Java, it is a mistake to think of the parts of an object as being "inside" the whole.



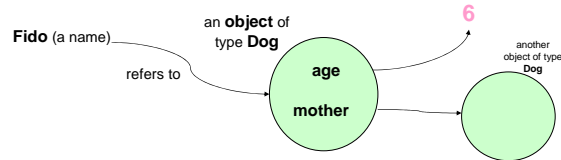
9/29/2004

(c) 2001-4, University of Washington

02-6

Drawing Names and Objects

- Names and objects
 - Very different things!
- In general, names refer to objects
 - Objects can *refer* to other objects using instance variable names



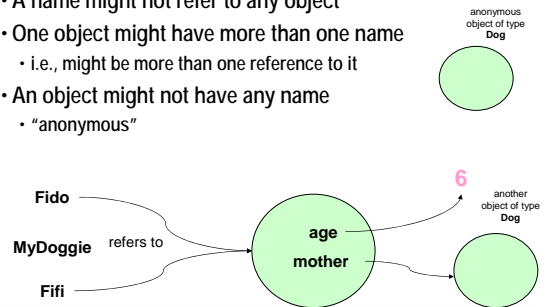
9/29/2004

(c) 2001-4, University of Washington

02-7

Drawing Names and Objects

- A name might not refer to any object
- One object might have more than one name
 - i.e., might be more than one reference to it
- An object might not have any name
 - "anonymous"



9/29/2004

(c) 2001-4, University of Washington

02-8

Specialization - "is a"

- Specialization relations can form *classification hierarchies*
 - cats and dogs are special kinds of mammals;
 - mammals and birds are special kinds of animals;
 - animals and plants are special kinds of living things
 - lines and triangles are special kinds of polygons;
 - rectangles, ovals, and polygons are special kinds of shapes
- Keep in mind: Specialization is not the same as composition
 - A cat "is-a" animal vs. a cat "has-a" owner

9/29/2004

(c) 2001-4, University of Washington

02-9

"is-a" in Programming

- Classes (and interfaces) can be related via *specialization*
 - one class/interface is a *special kind of* another class/interface
 - Rectangle class is a kind of Shape
- The general mechanism for representing "is-a" is *inheritance*

9/29/2004

(c) 2001-4, University of Washington

02-10

Inheritance

- Java provides direct support for "is-a" relations
 - likewise C++, C#, and other object-oriented languages
- Class *inheritance*
 - one class can *inherit from* another class, meaning that it's is a special kind of the other
- Terminology
 - Original class is called the *base class* or *superclass*
 - Specializing class is called the *derived class* or *subclass*

9/29/2004

(c) 2001-4, University of Washington

02-11

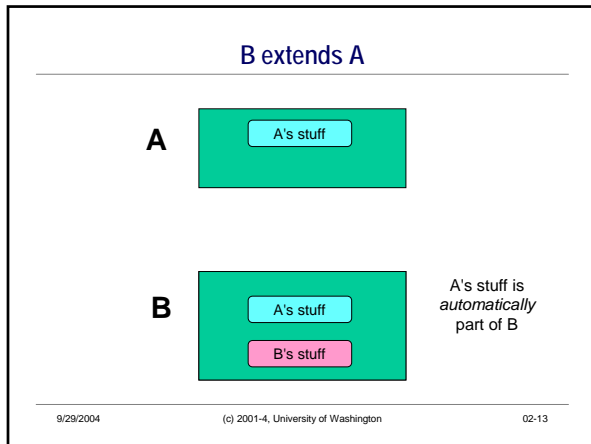
Inheritance: The Main Programming Facts

- Subclass *inherits* all instance variables and methods of the inherited class
 - All instance variables and methods of the superclass are *automatically* part of the subclass
 - Constructors are a special case (later)
- Subclass can *add* additional methods and instance variables
- Subclass can provide *different versions* of inherited methods

9/29/2004

(c) 2001-4, University of Washington

02-12



- ### Design Example: Employee Database
- Suppose we want to generalize our Employee example to handle a more realistic situation
 - Application domain – kinds of employees
 - Hourly
 - Exempt
 - Boss
- 9/29/2004 (c) 2001-4, University of Washington 02-14

- ### Design Process – Step 1
- Think up a class to model each “kind” of thing
- 9/29/2004 (c) 2001-4, University of Washington 02-15

- ### Design Process – Step 2
- Identify state/properties of each kind of thing
- 9/29/2004 (c) 2001-4, University of Washington 02-16

- ### Design Process – Step 3
- Identify actions (behaviors) that each kind of thing can do
- 9/29/2004 (c) 2001-4, University of Washington 02-17

- ### Key Observation
- Many kinds of employees share common properties and actions
 - We can factor common properties into a base class and use inheritance to create variations for specific classes
- 9/29/2004 (c) 2001-4, University of Washington 02-18

Generic Employees

```
/** Representation of a generic employee. */
public class Employee {
    // instance variables
    private String name; // employee name
    private int id; // employee id number
    /** Construct a new employee with the give name and id number... */
    public Employee(String name, int id) {
        this.name = name;
        this.id = id;
    }
    /** Return the name of this employee */
    public String getName() { return name; }
    ...
    /** Return the pay earned by this employee */
    public double getPay() { return 0.0; } // ???
    ...
}
```

9/29/2004

(c) 2001-4, University of Washington

02-19

Specific Kinds of Employees

• Hourly Employee

```
public class HourlyEmployee
    extends Employee {
    // additional instance variables
    private double hours; // hours worked
    private double hourlyPay; // pay rate

    /** Return pay earned */
    public double getPay() {
        return hours * hourlyPay;
    }
    ...
}
```

• Exempt Employee

```
public class HourlyEmployee
    extends Employee {
    // additional instance variable
    private double salary; // weekly pay

    /** Return pay earned */
    public double getPay() {
        return salary;
    }
    ...
}
```

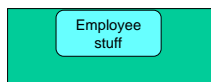
9/29/2004

(c) 2001-4, University of Washington

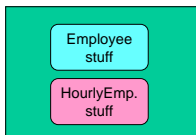
02-20

In Pictures

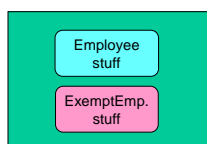
Employee



HourlyEmployee



ExemptEmployee



9/29/2004

(c) 2001-4, University of Washington

02-21

More Java

If class D extends B (inherits from) B...

- Class D inherits all methods and fields from class B
- But... "all" is too strong
 - constructors are *not* inherited (but there is a way to use superclass constructors during object creation)
 - same is true of static methods and static fields (although these static members are still available in inherited part of the object – technicalities we will look at later)
- Class D may contain additional (new) methods and fields
- But has no way to delete any

9/29/2004

(c) 2001-4, University of Washington

02-22

Never to be Forgotten

If class D extends (inherits from) B...

Every object of type D is also an object of type B

- a D can do anything that a B can do (because of inheritance)
- a D can be used in any context where a B is appropriate

9/29/2004

(c) 2001-4, University of Washington

02-23

Method Overriding

- If class D extends B, class D may provide an *alternative* or *replacement* implementation of any method it would otherwise inherit from B
- The definition in D is said to *override* the definition in B
- An overriding method cannot change the number of arguments or their types, or the type of the result [why?]
 - can only provide a different body (implementation)
- Can you override an instance variable?
 - Not exactly... ask after class if you're really curious

9/29/2004

(c) 2001-4, University of Washington

02-24

Polymorphism

- *Polymorphic*: "having many forms"
- A variable that can refer to objects of different types is said to be *polymorphic*
- Methods with polymorphic arguments are also said to be *polymorphic*

```
public void printPay(Employee e) {  
    System.out.println(e.getPay());  
}
```

- Polymorphic methods can be *reused* for many types

9/29/2004

(c) 2001-4, University of Washington

02-25

Static and Dynamic Types

- With polymorphism, we can distinguish between
 - Static type: the declared type of the variable (never changes)
 - Dynamic type: the run-time class of the object the variable currently refers to (can change as program executes)
 - Legal assignment depends on static type compatibility

9/29/2004

(c) 2001-4, University of Washington

02-26

Static and Dynamic Types

- Which of these are legal? Illegal?
 - Can you fix any of these with casts?
- What are the static and dynamic types of the variables after assignments?

```
HourlyEmployee bart = new HourlyEmployee(...);  
ExemptEmployee homer = new ExemptEmployee(...);  
Employee marge = new Employee(...)  
marge = homer;  
homer = bart;  
homer = marge;
```

9/29/2004

(c) 2001-4, University of Washington

02-27

Dynamic Dispatch

- "Dispatch" refers to the act of actually placing a method in execution at run-time
- When types are static, the compiler knows exactly what method must execute (i.e., which method from which class)
- When types are dynamic... the compiler knows the *name* of the method – but there could be ambiguity about which version of the method will actually be needed at run-time
 - In this case, the decision is deferred until run-time, and we refer to it as *dynamic dispatch*
 - The chosen method is the one matching the dynamic (actual) type of the object

9/29/2004

(c) 2001-4, University of Washington

02-28

Method Lookup: How Dynamic Dispatch Works

- When a message is sent to an object, the right method to run is the one in the *most specific class* that the object is an instance of
 - Makes sure that method overriding always has an effect
- Method lookup (a.k.a. *dynamic dispatch*) algorithm:
 - Start with the actual *run-time class* (*dynamic type*) of the receiver object (not the static type!)
 - Search that class for a matching method
 - If one is found, invoke it
 - Otherwise, go to the superclass, and continue searching
- Example:

```
Employee e = new HourlyEmployee(...)  
System.out.println(e); // HourlyEmployee.toString()  
Employee e = new ExemptEmployee(...)  
System.out.println(e); // ExemptEmployee.toString()
```

9/29/2004

(c) 2001-4, University of Washington

02-29

What about getPay()??

- Got to include it in Employee so polymorphic code can use it (why?)

```
public double getPay(Employee e) {  
    ...  
}
```

- But no implementation really makes sense
- Class Employee doesn't contain "pay" instance variables
- So including an implementation of this in Employee is really bogus

```
/* Return the pay earned by this employee */  
public double getPay() {  
    return 0.0; // ???  
}
```

9/29/2004

(c) 2001-4, University of Washington

02-30

Abstract Methods and Classes

- An **abstract method** is one that is declared but not implemented in a class

```
/** Return the pay earned by this employee */  
public abstract double getPay();
```

- A class that contains any abstract method(s) must itself be declared abstract

```
public abstract class Employee { ... }
```

- Instances of abstract classes cannot be created
 - Usually because they are missing implementations of one or more methods

9/29/2004

(c) 2001-4, University of Washington

02-31

Using Abstract Classes

- An abstract class is intended to be extended
- Extending classes can override abstract methods they inherit to provide actual implementations

```
class HourlyEmployee extends Employee {  
    ...  
    /** Return the pay of this Hourly Employee */  
    public double getPay() { return hoursWorked * payRate; }  
}
```

- Instances of these extended classes can be created
- A class that extends an abstract class without overriding all inherited abstract methods is itself abstract (and can be further extended)
- A class that is not abstract is often called a **concrete class**

9/29/2004

(c) 2001-4, University of Washington

02-32

Class Object

- **Object** is at the root of the Java class hierarchy
- Every class extends Object, either explicitly or implicitly
 - If extends does not appear in a class declaration, "extends Object" is assumed implicitly
 - These are equivalent

```
public class Employee { ... }  
public class Employee extends Object { ... }
```
- Object includes a small number of methods appropriate for all objects – toString, equals, a few others
 - These methods are inherited by all classes, but can be overridden – often necessary or at least a good idea

9/29/2004

(c) 2001-4, University of Washington

02-33

Summary

- Object-oriented programming is hugely important
 - Lots of new concepts and terms
 - Lots of new programming and modeling power
 - Used widely in real programs
- Ideas (so far!)
 - Composition ("has a") vs. specialization ("is a")
 - Inheritance
 - Method overriding
 - Polymorphism, static vs. dynamic types
 - Method lookup, dynamic dispatch
 - Abstract classes and methods

9/29/2004

(c) 2001-4, University of Washington

02-34