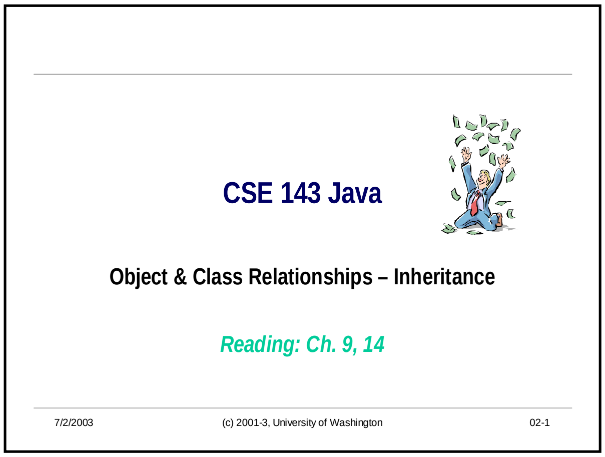
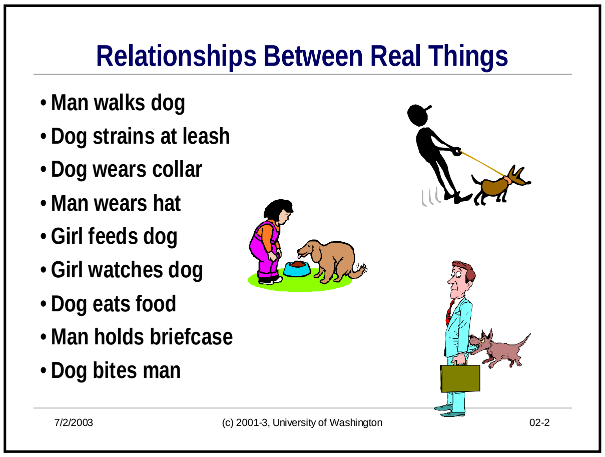


CSE 143 Java

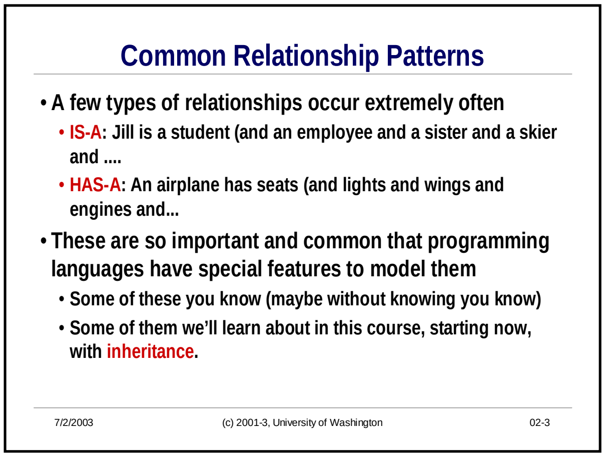
Object & Class Relationships – Inheritance

Reading: Ch. 9, 14



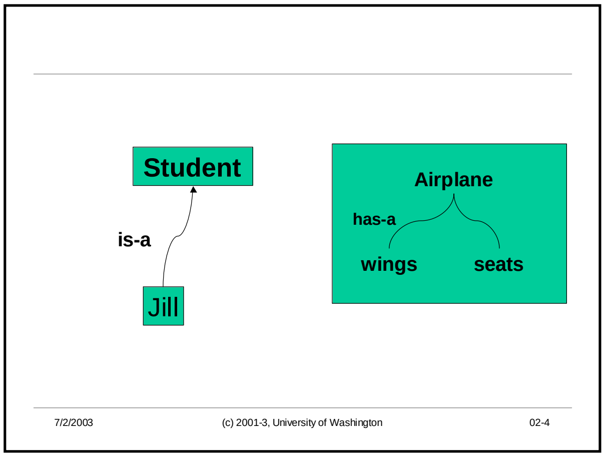
Relationships Between Real Things

- 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



Common Relationship Patterns

- A few types of relationships occur extremely often
  - IS-A: Jill is a student (and an employee 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 learn about in this course, starting now, with inheritance.



Student

is-a

Jill

Airplane

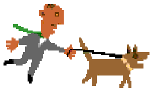
has-a

wings

seats

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

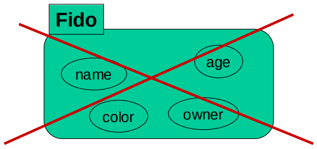


```
public class Dog {
    private String name;/          // this dog's name
    private int  age;              //this dog's age
    private Person owner;          // this dog's owner
    private Dog mother,  father;   // this dog's parents
    private Color coatColor;       //etc, etc.
}
```

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

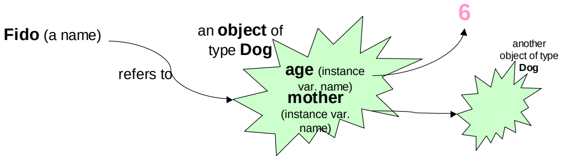
Picturing the Relationships

- 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.



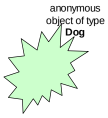
Drawing Names and Objects

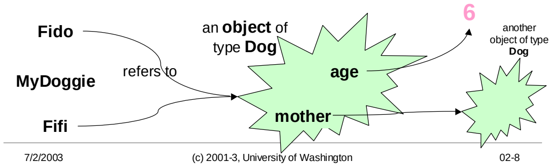
- Names and objects
  - Very different things!
- In general, names refer to objects
  - Objects can *refer* to other objects using instance variable names

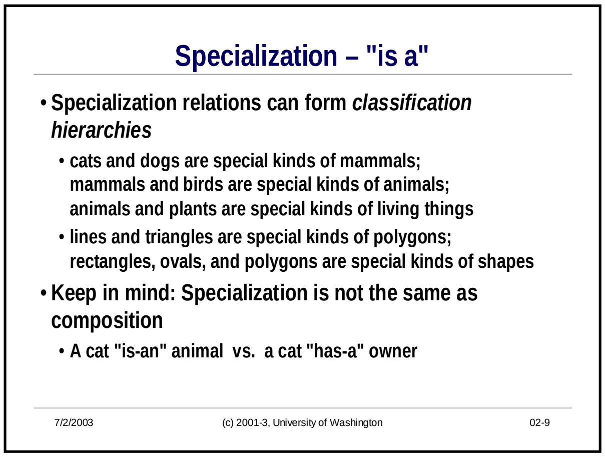


Drawing Names and Objects

- 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”

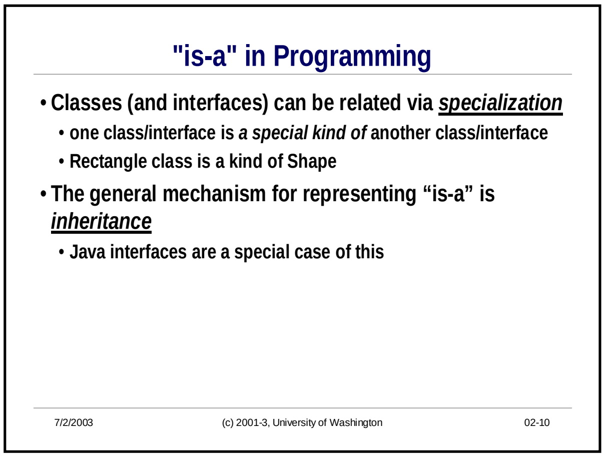






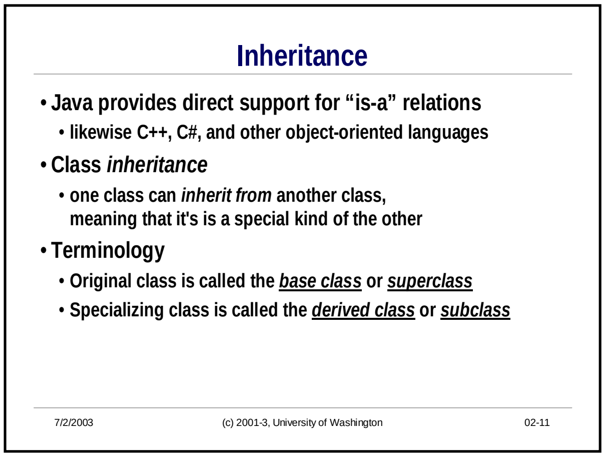
Specialization – "is a"

- 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-an" animal vs. a cat "has-a" owner



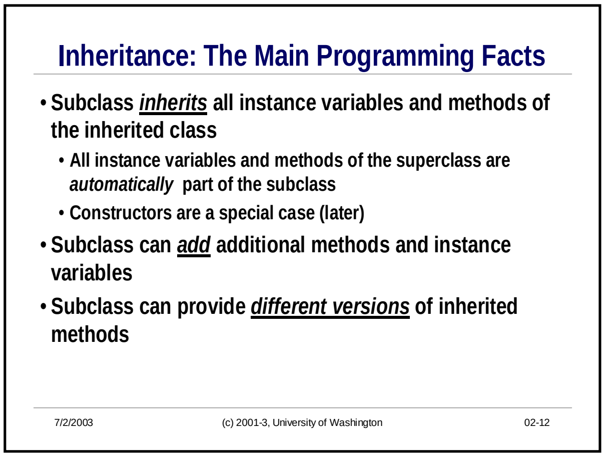
"is-a" in Programming

- 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***
  - Java interfaces are a special case of this



Inheritance

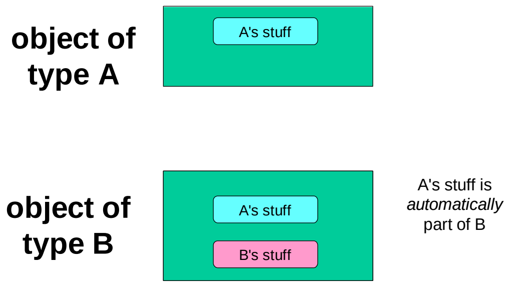
- Java provides direct support for "is-a" relations
  - likewise C++, C#, and other object-oriented languages
- Class *inheritance*
  - one class can *inherit from* another class,
    meaning that it's is a special kind of the other
- Terminology
  - Original class is called the ***base class*** or ***superclass***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



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

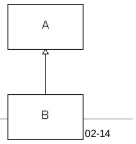
B extends A



A's stuff is *automatically* part of B

Drawing Classes

- Classes and interfaces are generally drawn as rectangles
  - In UML-style pictures, put the label inside the rectangle, near the top
- When there is a relationship between the classes, you can draw a line between the rectangles
  - Lines may have arrowheads or other decorations to indicate additional relationship information



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

Design Process – Step 1

- Think up a class to model each “kind” of thing

Design Process – Step 2

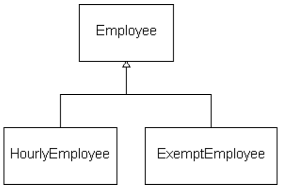
- Identify state/properties of each kind of thing

Design Process – Step 3

- Identify actions (behaviors) that each kind of thing can do

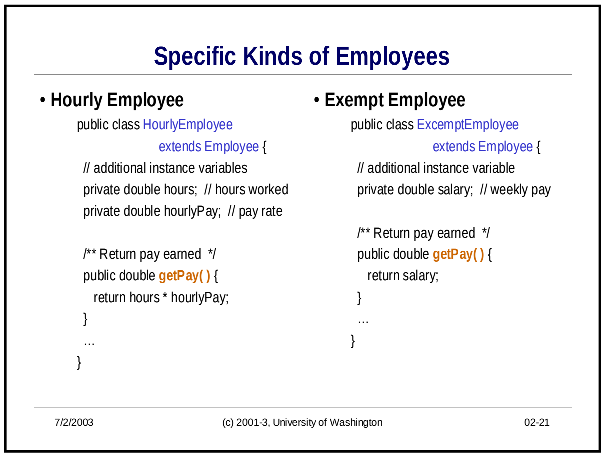
Key Observation

- Many kinds of employees share *common* properties and actions
- We can factor common elements into a base
- Use inheritance to create variations for specific classes



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; }    // ???
   …
 }
```



Specific Kinds of Employees

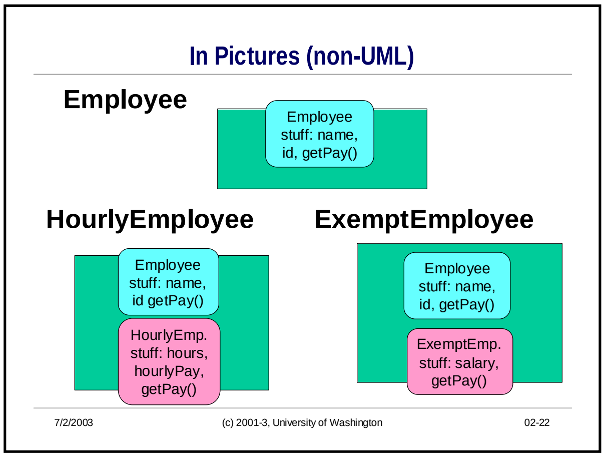
- Hourly Employee

```
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 ExcemptEmployee
            extends Employee {
  // additional instance variable
  private double salary;  // weekly pay

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

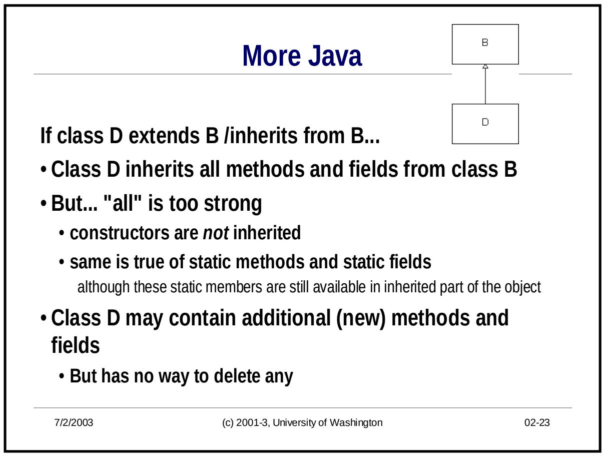
7/2/2003 (c) 2001-3, University of Washington 02-21



In Pictures (non-UML)

Employee
Employee stuff: name, id, getPay()
HourlyEmployee
Employee stuff: name, id getPay()
HourlyEmp. stuff: hours, hourlyPay, getPay()
ExemptEmployee
Employee stuff: name, id, getPay()
ExemptEmp. stuff: salary, getPay()

7/2/2003 (c) 2001-3, University of Washington 02-22



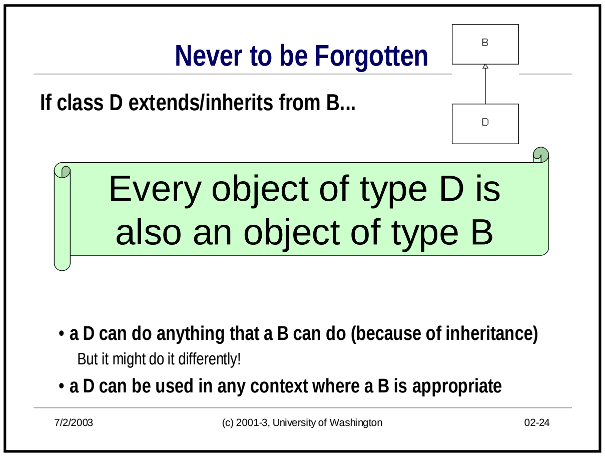
More Java

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
  - same is true of static methods and static fields

    although these static members are still available in inherited part of the object
- Class D may contain additional (new) methods and fields
  - But has no way to delete any

7/2/2003 (c) 2001-3, University of Washington 02-23



Never to be Forgotten

If class D extends/inherits from B...

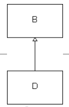
Every object of type D is also an object of type B

- a D can do anything that a B can do (because of inheritance)

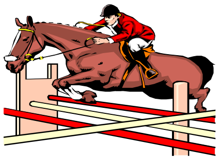
  But it might do it differently!
- a D can be used in any context where a B is appropriate

7/2/2003 (c) 2001-3, University of Washington 02-24

Method Overriding

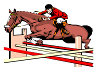


- If class D extends B, class D may provide an *alternative*, *replacement* implementation of any method it would otherwise inherit from B
- The definition in D is said to ***override*** the definition in B
- Example: getPay( )



Peculiarities of Overriding

- An overriding method
  - cannot change the number of arguments
  - cannot change the argument types
  - cannot change the type of the result [why?]
- Can you override an instance variable?
  - The basic answer is “please don’t”
  - You might not get an obvious error if you try it... ask me in person if you're really curious



Polymorphism

- *Polymorphic*: "having many forms“
- Polymorphism is an important feature of object-oriented programming
- Polymorphism comes in several flavors
  - You could say, polymorphism is polymorphic...
- College survival tip: Next time you have to write an essay in a humanities class, use the words “polymorphic” and “polymorphism”. Watch your grade rise!

Object Reference Polymorphism

- A variable that can refer to objects of different types is said to be *polymorphic*
- Example:

```
Animal  pet;
Dog myDog;
Cat myCat;
```

  - If *Animal* is the superclass of *Dog* and *Cat*, *pet* can refer to either a dog or a cat! In this sense, *pet* is polymorphic

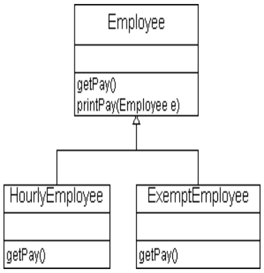
```
myDog = new Dog(“Fido”);
myCat = new Cat(“Mimsy”);
pet = myDog; //legal or illegal?
pet = myCat; //legal or illegal?
myDog = myCat; //legal or illegal?
```

Method Polymorphism

- Methods with polymorphic arguments are also said to be polymorphic

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

- Method printPay can be called with an argument of type HourlyEmployee or of type ExemptEmployee
  - Note that printPay itself is not overriden
  - But it acts differently depending on the dynamic type of e
- Polymorphic methods can be *reused* for many types



HourlyEmployee emp1 = new HourlyEmployee("Cartman");

ExemptEmployee emp2 = new("Kenny");

printPay(emp1);

printPay(emp2);

Static and Dynamic Types

- 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

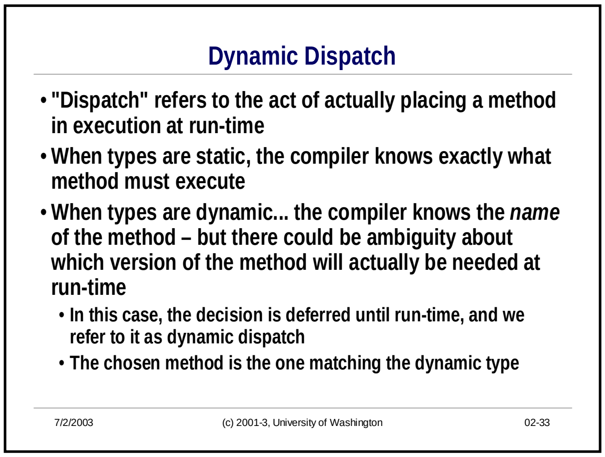
A = B;

- When is such an assignment statement legal?
- It depends on *static* type compatibility
- Does not depend on dynamic type of A and B

Static and Dynamic Types

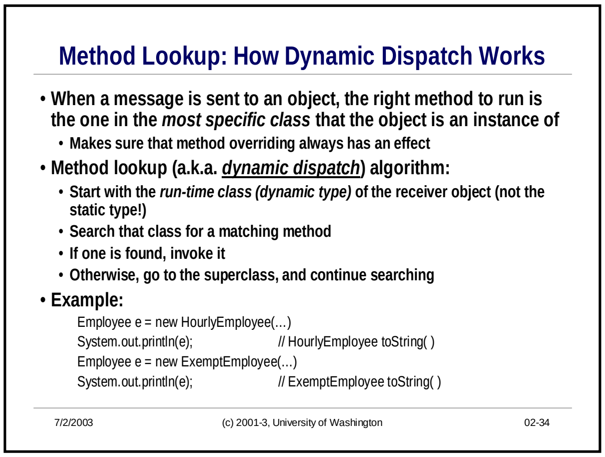
- Which of these are legal? Illegal?
  - Can you fix any of these with casts?
- What are the static and dynamic types of the variables after assignments?

	Static?	Dynamic?
HourlyEmployee bart = new HourlyEmployee(…);		
ExemptEmployee homer = new ExemptEmployee(…);		
Employee marge = new Employee(…)		
marge = homer ;		
homer = bart;		
homer = marge;		



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
- 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 type



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 *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( )
```