CSC 143 Java

Class Relationships and Inheritance

Common Relationship Patterns

- A few types of relationships occur extremely often
  - IS – A: a supervisor is an employee, a Chevrolet Camaro is a car
  - HAS – A: An airplane has seats (and wings, and …)
- These are so important and common that programming languages have special features to model them

Composition: HAS - A

- Object *composition, aggregation,* or *reference*: instance variables that refer to other objects
  - Simple example: objects representing people

```
public class Person {
    // instance variables
    private String name;        // this person's name
    private Person mother;      // this person's mother
}
```


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
- Specialization is not the same as composition ("has-a")
  - A truck "is-a" vehicle vs. a truck "has-a" door
- Often, classes related this way have some behavior or state in common.

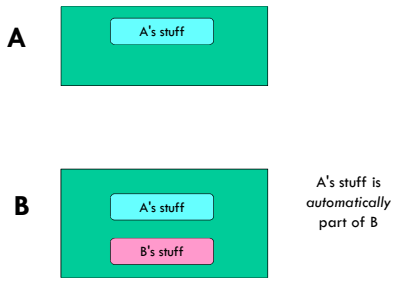
IS – A in Programming: Inheritance

- Java (and C++, and many languages) provide direct support for "IS - A": **class inheritance**
- Idea: define a new class as an extension or specialization of an existing class
- Key concept for object-oriented programming: reusing software may shorten development cycle and reduce bugs.

Vocabulary and Principles

- Terminology:
  - Original class is called the **base class** or **super class**
  - Specialized class is called the **derived class** or **sub class**
- Derived class *inherits* all instance variables and methods of the inherited class
  - All instance variables and methods of the superclass are *automatically* part of the subclass
  - There are some special cases (discussed later)
- Derived class can *add* additional methods and instance variables
- Derived class can provide *different versions* of inherited methods

B extends A



UML Diagrams

- Unified Modeling Language
- Different diagram categories for different views of the system
  - Use-case diagrams
  - Class diagrams
  - Object diagrams
  - Sequence diagrams
  - State diagrams
  - Etc…
- What is the UML class diagram for HAS-A? IS-A?

Design Example: Employee Database

- Suppose we want to generalize an Employee class to handle a more realistic situation
- Application domain – kinds of employees
  - Hourly
  - Exempt
  - Boss
- Initial Design Process
  - Step 1: think up class to model each thing
  - Step 2: identify state/properties of each thing
  - Step 3: identify actions/responsibilities of each thing

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

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 hoursWorked;
 private double payRate;

public void setHours(double hrs) {
    hoursWorked = hrs;
}
…
}
```

- Exempt Employee

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

…
}
```

What does the UML class diagram look like?

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
  - private data is hidden from derived class implementation
    but can access through get/set methods from base class (if they exist!)
- Class D may contain additional (new) methods and fields
  - But has no way to delete any

Derived Class Constructors

How do we implement HourlyEmployee constructor?

```
public HourlyEmployee(String name, int id, double pay) {...}
```

Need to initialize instance variables that have been inherited.

BUT, they are private, so derived class code cannot access them.

Any thoughts?

Member Access in Subclasses

- **public:** accessible anywhere the class can be accessed
- **private:** accessible only inside the same class
  - Does *not* include subclasses – derived classes have no special permissions

- A new mode: ***protected***
  accessible inside the defining class and all its subclasses
  - Use protected for "internal" things that subclasses also are intended to access
  - Consider this carefully – why?

Super

- If a subclass constructor wants to run a superclass constructor, it can do that using the syntax
  super(<possibly empty list of argument expressions>)
  as the first thing in the subclass constructor's body
- Example:

```
public HourlyEmployee(String name, int id, double pay) {
  super(name, id);
  payRate = pay;
  hoursWorked = 0.0;
 }
```

Let's write the constructor for ExemptEmployee

Constructor Rules

- Rule 1: If you do not write any constructor in a class, Java assumes there is a zero-argument, empty one
  ClassName( ) { }
  - If you write any constructor, Java does not make this assumption
- Rule 2: If you do not write super(…) as the first line of an extended class constructor, the compiler will assume the constructor starts with a call to super();
- Rule 3: When an extended class object is constructed, there must be a constructor in the parent class whose parameter list matches the explicit or implicit call to super(…)
- Corollary: a constructor is always called at each level of the inheritance chain when an object is created

Overloaded Constructors and this

- Classes often have several related Constructors
  - Common pattern: some provide explicit parameters while others assume default values
- this(…) can be used at the beginning of a constructor to execute another constructor in the same class
  - Syntax similar to super
  - Can have other statements in the constructor following the "this" call
  - Good practice – can provide a single implementation of code common to both constructors

Overriding Methods

- If class D extends B, class D may provide alternative implementations for methods it would otherwise inherit from B
- Overriding: replacing an inherited method in a subclass

```
class One {
    public int method(String arg1, double arg2) { … }
}
class Two extends One {
    public int method(String arg1, double arg2) { … }
}
```

Method name, parameter lists, and return must match *exactly* (number and types)

Let's override getPay() for the two subclasses

Another use for Super

- calling super(…) from the constructor of the derived class to execute a constructor of the base class.
- Another use : in any subclass, super.method(args) can be used to call the version of the method in the superclass, even if it has been overridden
  - Can be done anywhere in the code – does not need to be at the beginning of the calling method, as for constructors
  - Often used to create "wrapper" methods

```
/** Return the pay of this manager.  Managers receive a 20% bonus */
public double getPay() {
    double basePay = super.getPay();
    return basePay * 1.2;
}
```

  - Question: what if we had written "this.getPay( )" instead?

Client Code

- Employee e = new Employee("Quynh", 123);
  Draw the picture. What methods can we call?
- HourlyEmployee h = new HourlyEmployee("Jacob", 555, 8.50);
  Draw the picture. What methods can we call?
- e = h;
  Is this legal? What's going on here?

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

Polymorphism

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

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

- Polymorphic methods can be *reused* for many types

Static and Dynamic Types

- With polymorphism, we can distinguish between
  - Static type: the declared type of the reference variable. Used by the compiler to check syntax.
  - Dynamic type: the run-time type of the object the variable currently refers to (can change as program executes)

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 (actual) type of the object

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
  - Ensures 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

Related Dynamic Dispatch Topics

Object

- toString()
- Equals()

instanceof: <object> instanceof <classOrInterface>

- common use: checking types of generic objects before casting

```
if (otherObject instanceof Blob) {
    Blob bob = (Blob) otherObject;
    ....
}
```


What about getPay( )?

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

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

- But no implementation really makes sense. Let's address that next…

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 { … }
```

- abstract classes cannot be instantiated
  - Because they are missing implementations of one or more methods

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; }
}
```

  - extended classes can be instantiated
- 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*