Programming and Software Development
COMP90041

Lecture 8

Inheritance

Dr. Thomas Christy
thomas.christy@unimelb.edu.au

Abstract Datatypes

- An abstract data type (ADT) is a type that is defined in terms of its operations and their semantics (meaning)
- The focus of an ADT is on its interface — its publicly visible part
- Clients (users) of a class view it as its interface
- The interface hides the complexity of the implementation from clients
- ADTs are one of the central concepts of object oriented programming

Interface Perspective

- But an ADT also needs an implementation
- The ADT's implementors view it as its implementation
- The interface hides the complexity of all the ADT's uses from its implementors
- An ADT's interface insulates client from implementor, allowing them to work independently
  - As long as client follows ADT interface, he may use the ADT any way he likes
  - As long as the implementor maintains the ADT interface, she may modify the implementation any way she likes
- A Java class is well-suited to defining an ADT

Design by Contract

- ADT's interface specifies semantics of operations: *"if you supply inputs that meet these conditions, I will supply an output that meets those conditions"*
- Think of an interface as a contract between implementor and client
  - Class clients must supply inputs that meet the contract
  - Class implementor expects (should verify) inputs that meet the contract, and must supply outputs that do
- This leads to design by contract emphasising not just the types of operation inputs and outputs, but their preconditions and postconditions

Inheritance

- The second central concept of object oriented programming is inheritance
- Inheritance allows a derived class to be defined by specifying only how it differs from its base class
- Base class also called superclass or parent class; derived class also called subclass or child class
- Parts of the derived class that are the same as in the base class need not be mentioned again
- Called implementation inheritance because implementation as well as interface is inherited

Declaration

- Put `extends BaseClass` in declaration of derived class to declare inheritance, *e.g.*:

```
public class LostPerson extends Person {
    private String lastSeenLocation;
    private Time lastSeenTime;
    ⋮
```

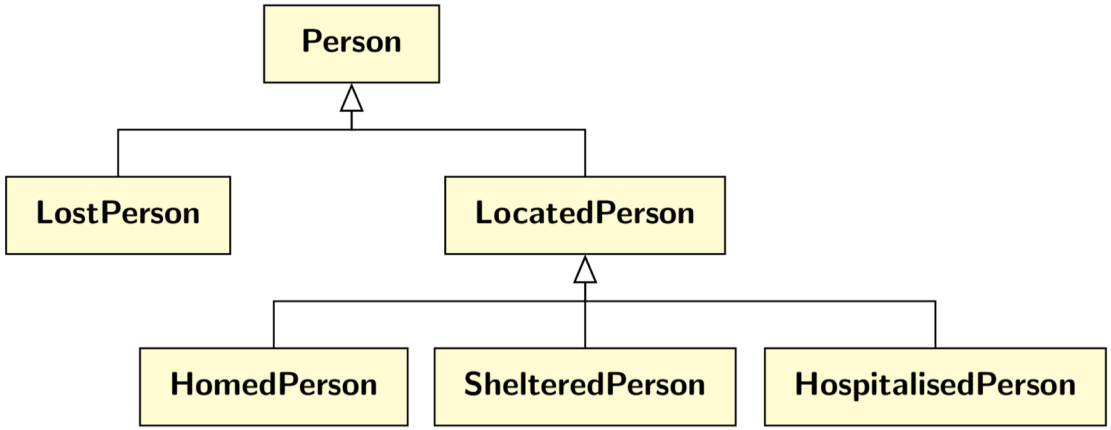
- This `LostPerson` class inherits all the instance variables and methods of the `Person` class
- . . . and adds its own
- No need to mention inherited instance variables and methods

Liskov Substitution Principle

- In Java, every instance of the derived class is also an instance of the base class
- *E.g.*, every `LostPerson` is a `Person`
- Things you can do with a `LostPerson` object:
  - Store it in a variable of type `Person`;
  - Pass it as a message argument of type `Person`;
  - Send it any message understood by a `Person`
- Liskov Substitution Principle (LSP) says *it must be possible to substitute an instance of the derived class anywhere the base class could be used*
- Be sure you design and implement derived classes so this is true

Taxonomy

- Each class can be extended by any number of classes
- Java is a single inheritance language: each class can extend only one class
- UML class diagram shows inheritance hierarchy: hollow-headed arrows point to base classes



- Each class inherits from all its ancestors; members are inherited by all descendants

Overriding

- If a class defines a method with the same signature as an ancestor, its definition overrides the ancestor's
- Base or derived class's definition is used depending on class of object
- *E.g.*, `Person` class's `toString` method just shows name and age; `LostPerson`'s `toString`:

```
public String toString() {
    return getName() + ", age " +
           getAge() + ", last seen " +
           lastSeenTime + " at " +
           lastSeenLocation;
}
```

Using Overridden Methods

- Need to use `getName()` and `getAge()` because `name` and `age` instance variables are private
- Would be better to use the overridden `toString()` method: works even if we modify superclass
- We can: inside a method, use `super.`*`methodName`*`(`*`args...`*`)` to invoke the overridden method

```
public String toString() {
    return super.toString() +
           ", last seen " + lastSeenTime +
           " at " + lastSeenLocation;
}
```

Late Binding

- In Java we can always use a `LostPerson` where a `Person` is expected:

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

- Which `toString` method is used?
  - `LostPerson`'s because that's what `p` actually is?
  - `Person`'s because that's what `p` is declared to be?
- For Java, it's always based on object's actual type
- This is called late binding or dynamic binding, because compiler defers decision to runtime
- No late binding for `static` members (because there is no `this` object on which to base the decision)
- But `static` members are still Inherited

Overriding vs Overloading

- Overloading and overriding are completely different
- If signature of method in derived class is the same as method in base class, it's overriding
- If method name is the same but signature is different, it's overloading
- When sending message to instance of derived class:
  - With overloading, you can access both methods, depending on number and types of arguments
  - With overriding, you can only access the overriding method
- You usually want overriding

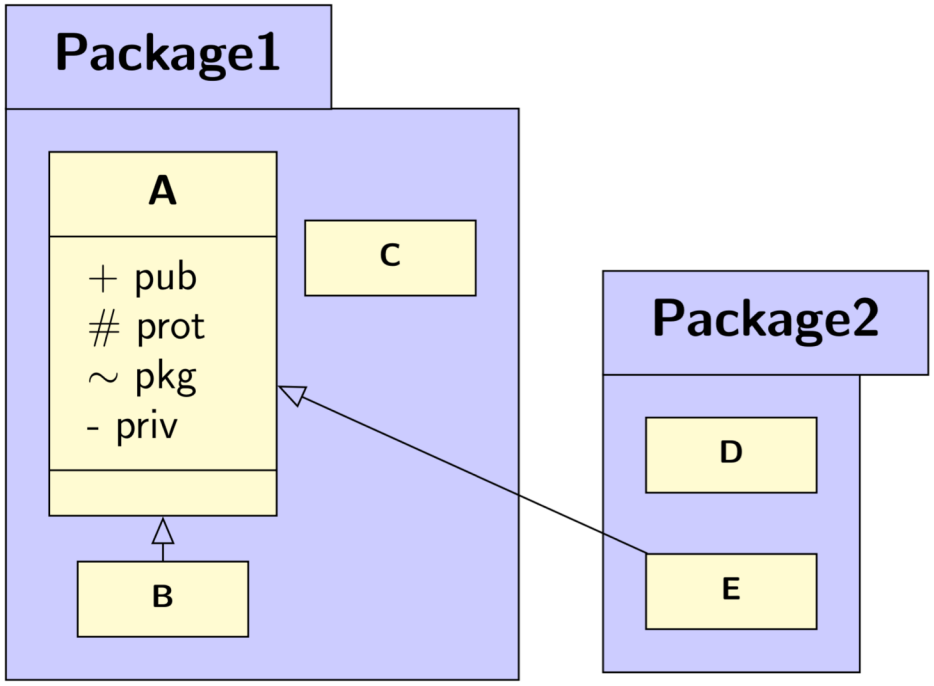
Inheritance and Visibility

- Occasionally it's useful to allow methods in derived classes to access base class instance variables
- Protected visibility allows this
- Form: `protected` *`type instanceVar`*`;`
- Alternative to `public` and `private`
- Name is a misnomer: `protected` instance variables are not well very protected
- To access a protected instance variable, you just need to create a subclass
- Can also declare methods `protected`, which may be more useful

Packages and Visibility

- A Java package is a collection of classes
- Class declares package with: `package` *`pkgname`*`;`
- Package classes all in same folder/directory
- Protected members are also visible in all classes in the same package
- Fourth visibility is called default or friendly or package visibility
- This means visible in any class in the same package
- Declare package visibility by not using any visibility keyword (no `public`, `private`, or `protected`)
- In order of least visibility to most:
  `private` < default < `protected` < `public`

Packages and Visibility



A sees pub, prot, pkg, priv
B sees pub, prot, pkg,
C sees pub, prot, pkg,
D sees pub
E sees pub, prot

UML legend

Class parts:

Class
instance variables. . .
methods. . .

Visibility:

+ → public
# → protected
~ → package
- → private

Visibility and Overriding

- You cannot override a method giving it less visibility
- LSP requires that a visible base class method must be visible for every descendent class
- You can override a method with a more visible method, though

The `super` Constructor

- Constructors are not inherited, cannot be overriden
- Base class constructor must be run to set up its instance variables (especially if private)
- Constructor chaining: derived class constructor must invoke base class constructor first
- Form: `super(constructor arguments...)`, *e.g.*:

```
public LostPerson(int age, String name,
        String lastLoc, Time lastTime) {
    super(age, name);
    this.lastSeenLocation = lastLoc;
    this.lastSeenTime = lastTime;
}
```

The `this` Constructor

- Sometimes you want to chain to a different overloaded constructor of the same class
- Form: `this(constructor arguments...)`
- Must be first in constructor, in place of `super`
- The constructor chained to will itself chain to super (or another overloaded constructor that will...)

```
public Person(Person orig) {
    this(orig.age, orig.name);
}
```

- Can also chain to constructor that does part of the work, and then do whatever extra is needed

Default Constructor Chaining

- If constructor doesn't begin with `super(...)` or `this(...)`, Java automatically inserts `super();`
- If your class doesn't have a no-argument constructor, this will be an error
- If you write a class without writing any constructor, Java will automatically write a no-argument constructor with body `{super();}`
- But if you write any constructor at all, Java does not give you the free no-argument one

The `Object` Class

- A class that does not declare what class it `extends` automatically extends the `Object` class
- So class hierarchy is a tree (but we don't usually show `Object` class in class diagrams)
- The `Object` class defines a `toString()` method
- . . . and an `equals(Object other)` method
- These are inherited by all classes, but the definitions are not useful
- They should be overriden if they will be used
- `Object` class has no instance variables
- `Object` has a no-argument constructor that does nothing, so default constructor chaining is fine

The `instanceof` Operator

- Usually you use overriding to arrange each method to behave correctly for every descendant of a class
- So most code does not need to worry about which descendant type of a base class an object is
- Occasionally you want a test to see if an object is a descendant of a class
- Form: *`object instanceof ClassName`*
- *E.g.*:

```
if (p instanceof LostPerson) {
    System.out.println(p + " is missing");
}
```

Upcasting and Downcasting

- Derived class may have methods base class does not
- Java will not let you use a method not supported by declared type of an object
- Need to downcast (narrowing cast) to derived class to use derived class-specific methods

```
String lastLoc = "";
if (p instanceof LostPerson) {
    LostPerson lp = (LostPerson) p;
    lastLoc = lp.getLastSeenLocation();
}
```

- Can upcast (widening cast) implicitly or explicitly

```
Person p = (Person) new LostPerson(...);
```

The `getClass` method

- `Object` class also defines a getClass() method
- `ob.getClass()` returns an object that represents the actual class of `ob`
- You can use `==` and `!=` to compare two of these to see if classes are the same
- *E.g.*, `o1.getClass() == o2.getClass()` is true if `o1` and `o2` are actually instances of the same class, not just descendants of some class
- You cannot override `getClass` for your classes, but you don't need to

Defining the `equals` method

- Must override not just overload the `equals` method
- Must have the signature `equals(Object other)` (class of `other` must be `Object`)
- Must check that `other` is not `null`
- Use `getClass` to check objects are the same class
- Must downcast `other` to correct class so you can access members to compare
- For derived class, use `super.equals(other)` to check base class instance variables

Defining the `equals` method

```
public boolean equals(Object other) {
    if (other == null ||
        this.getClass() != other.getClass() ||
        !super.equals(other)) {
        return false;
    }
    LostPerson lp = (LostPerson) other;
    return (this.lastSeenLocation.equals(
                    lp.lastSeenLocation) &&
            this.lastSeenTime.equals(
                    lp.lastSeenTime));
}
```

Summary

- Methods form a contract between client and implementor
- Use `extends` to define class that inherits members from base class
- Instances of derived class are also instances of the base class
  - Design/implement classes so this makes sense
- Derived class can override base class methods
- Late binding: definition for actual class is used
- Members with default visibility accessible from package; `protected` also accessible from subclasses
- Use `super` constructor to chain to base class constructor; `this` to chain to same class

THE UNIVERSITY OF
MELBOURNE