Enumerated Types

- An enumerated type is a type whose values are all specific constants
- New feature in Java 5
- (Simplest) form:
- `enum typeName {value1, value2,...};`
- Place this at top level inside a class, or in a file by itself, named `typeName.java`
- Convention: values are spelled all uppercase, words separated by underscores
- *E.g.*:

```
enum DayOfWeek {SUNDAY, MONDAY, TUESDAY,
        WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
};
```

Using Enumerated Types

- With `enum` declaration, *`typeName`* is a valid type
- Each *`value`* is a constant referred to as *`typeName.value`*
- Use `==` and `!=` to compare `enum` values for equality
- *E.g.*:

```
public static boolean isWeekend(DayOfWeek day) {
    return day == DayOfWeek.SATURDAY
        || day == DayOfWeek.SUNDAY;
}
```

Enumerated Types Methods

- Java implements `enum`s as classes
- They automatically have useful public methods:
  - `String toString()` returns value as a `String`
  - `static` *`type`* `valueOf(String)` returns enum value of string (Note `static`; error if no exact match)
  - `int ordinal()` returns 0 for the first value, 1 for the second, *etc.*.
  - `static` *`type`*`[] values()` Returns an array of all the enum values for the type, in order
- `values` is very useful, as it allows you to loop over all values of an enum type

`equals` and `compareTo`

- Every class type *t*, including `enum` types, defines:
  - `boolean equals(`*t*`)` returns `true` iff the object is "the same" as the argument *t*
  - `int compareTo(`*t*`)` returns a negative number if the object is "less than" the argument *t*, zero if the same, and positive if "greater"
- For classes, the programmer must define these
- It's up to you to decide what "the same", "less" and "greater" mean for your class
- For `enum` types, they are defined automatically; "less than" means appearing earlier in the declaration
- Usually must use `a.equals(b)`, not `a==b`, to test objects for equality; enum types are an exception
- Not defined for arrays, only classes and enums

Enumerated Type Example

```
public class DayOfWeekTest {
    enum DayOfWeek {SUNDAY, MONDAY, TUESDAY,
        WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
    };

    public static void main(String[] args) {
        DayOfWeek d = DayOfWeek.valueOf(args[0]);
        System.out.println(d + ": " +
                (d.ordinal()+1) + "th day of week");
        int next = d.ordinal() + 1;
        next %= DayOfWeek.values().length;
        d = DayOfWeek.values()[next];
        System.out.println("Next day is " + d);
    }
}
```

Summary

- An array is a collection of any number of values of the same type (can be primitive or object type)
- `baseType[]` is type of array of `baseType` elements
- `new baseType[size]` returns a fresh array of `size` elements of `baseType`
- Access/set elements using `array[index]` syntax; `array[0]` is first element
- `array.length` is number of elements in array
- `for(type name : array)` iterates over `array`
- `enum type {value1,...};` declares an enumerated type

THE UNIVERSITY OF
MELBOURNE

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;
    ⋮
```

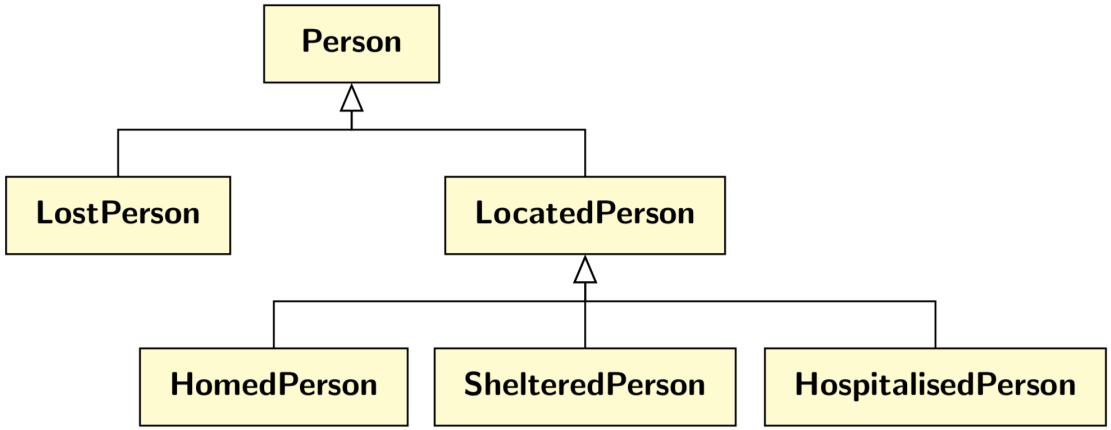
- This `LostPerson` class inherits all the instance variables and methods of the `Person` class
- . . . and adds its own
- No need to mention inherited instance variables and methods

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

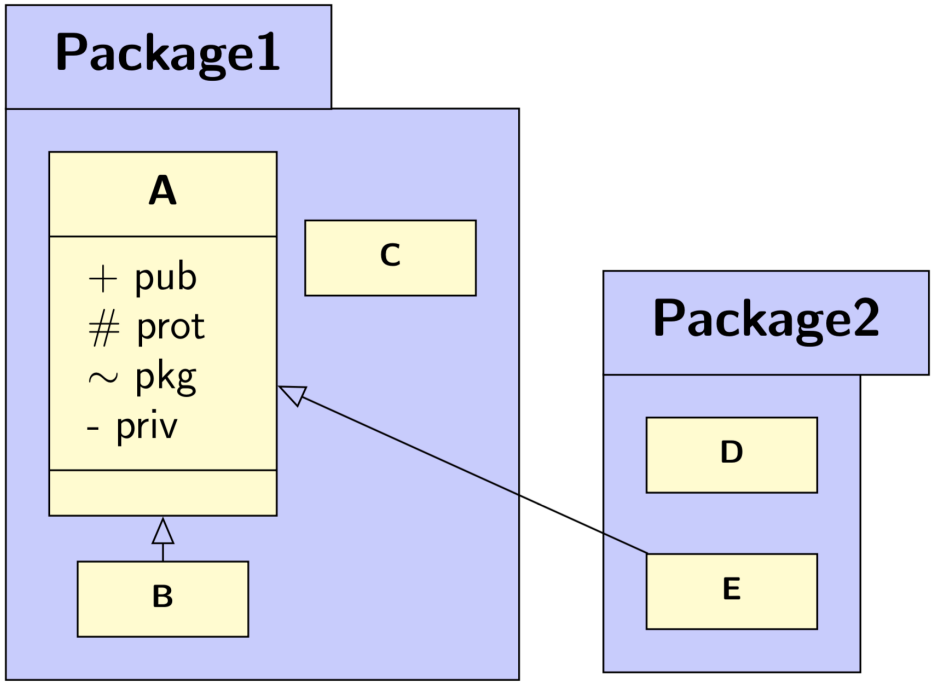
Inheritance and Visibility

- 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`

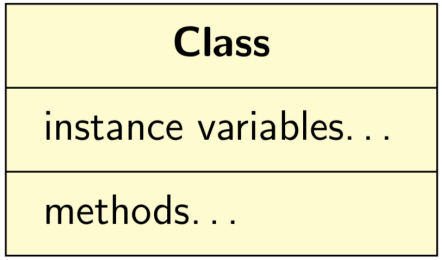
Packages and Visibility



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:



Visibility:

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

Quick Poll

bit.ly/tchristy

qp.unimelb.edu.au/tchristy

Quick Poll

Which classes can access method m?

```
package p1; public class c1 { void m()... }
package p1; public class c2 {...}
package p1; public class c3 extends c1 {...}
package p2; public class c4 {...}
package p2; public class c5 extends c1 {...}
```

A. c1 only
B. c1 and c3 only
C. c1, c2 and c3 only
D. c1, c2, c3 and c5 only
E. All of c1, c2, c3, c4 and c5

Quick Poll

Which classes can access method m?

```
package p1; public class c1 { void m()... }
package p1; public class c2 {...}
package p1; public class c3 extends c1 {...}
package p2; public class c4 {...}
package p2; public class c5 extends c1 {...}
```

A. c1 only
B. c1 and c3 only
C. c1, c2 and c3 only
D. c1, c2, c3 and c5 only
E. All of c1, c2, c3, c4 and c5

Quick Poll

Which classes can access method m?

```
package p1; public class c1 { void m()... }
package p1; public class c2 {...}
package p1; public class c3 extends c1 {...}
package p2; public class c4 {...}
package p2; public class c5 extends c1 {...}
```

A. c1 only
B. c1 and c3 only
C. c1, c2 and c3 only
D. c1, c2, c3 and c5 only
E. All of c1, c2, c3, c4 and c5

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 UNIVERSITY OF
MELBOURNE