Arrays, Inheritance

27 Jan 2011
CMPT166
Dr. Sean Ho
Trinity Western University

TRINITY WESTERN UNIVERSITY

Outline for today

- Unit testing with JUnit
  - FruitStand example
- Arrays
  - Declaring, allocating, initializing
  - Iterating over arrays
- Inheritance
  - “Has a” vs. “Is a” vs. “Is a kind of”
  - Overriding methods
  - Polymorphism

Class design: testbed

- Main class (Student): attribs, methods, constr.
  - **public class Student {**
    - **String name;**
    - **short ID;**
    - **public Student() {…}**
- Testbed class (StudentTest):
  - main() and other methods create instances of Student and call methods:
    - **public class StudentTest {**
      - **public static void main(String args[]) {**
        - **Student s1 = new Student();**
        - **s1.setName(“Joe Smith”);**

Unit testing with JUnit4

- Create a separate class to hold your testcases
    - **import org.junit.Test;**
    - **import static org.junit.Assert.*;**
- Each test case is a method: annotate with @Test
    - Create some objects from your class
    - Call some methods on your objects
    - Make assertions: assertEquals(a, b);
- Run the test cases:
    - In Eclipse: New → JUnit Test Case, and Run
    - org.junit.runner.JUnitCore.runClasses(TestClass1.class);

Arrays in Java

- Aggregate (compound/container) data type
- All entries must have same type
- Size of array is fixed when array is allocated
  - But need not be known at compile-time
  - Arrays can be dynamically created
- Location in memory is usually contiguous
- Index into array using integer indices from 0 up to (size of array)-1
  - Indexing out-of-bounds raises ArrayIndexOutOfBoundsException

Working with arrays

- Declaring arrays:
  - int numApples[]; // or: int[] numApples;
- Allocate array in memory:
  - numApples = new int[10];
- Initializing array entries:
  - numApples[3] = 15;
- Size of array:
  - numApples.length // returns 10

Array initializers and constants

- Initialize an array on one line:
  - int numApples[] = {5, 3, 12, 0, 3};
- Declare constants using the keyword final:
  - final int numApples[] = {5, 3, 12, 0, 3};
  - final float pi = 3.14159265358979323846264;
  - Values cannot be changed
    (even by code in the same class)
  - Initial value must be given in-line with declaration

Multidimensional arrays

- The element type of an array can be any type, including objects, including other arrays:

```
int image[][];
image = new int[width][height];
for (int x=0; x<width; x++)
    for (int y=0; y<width; y++)
        image[x][y] += 10;
```

- Rows may be different lengths:

```
image = new int[width][];
for (int x=0; x<width; x++)
    image[x] = new int[x];          // triangular array
```

Iterating through arrays

- Iterate through an array with a for loop:

```
        for (int idx=0; idx < array.length; idx++)
            sum += array[idx];
```

- Java has an enhancement to the for loop:

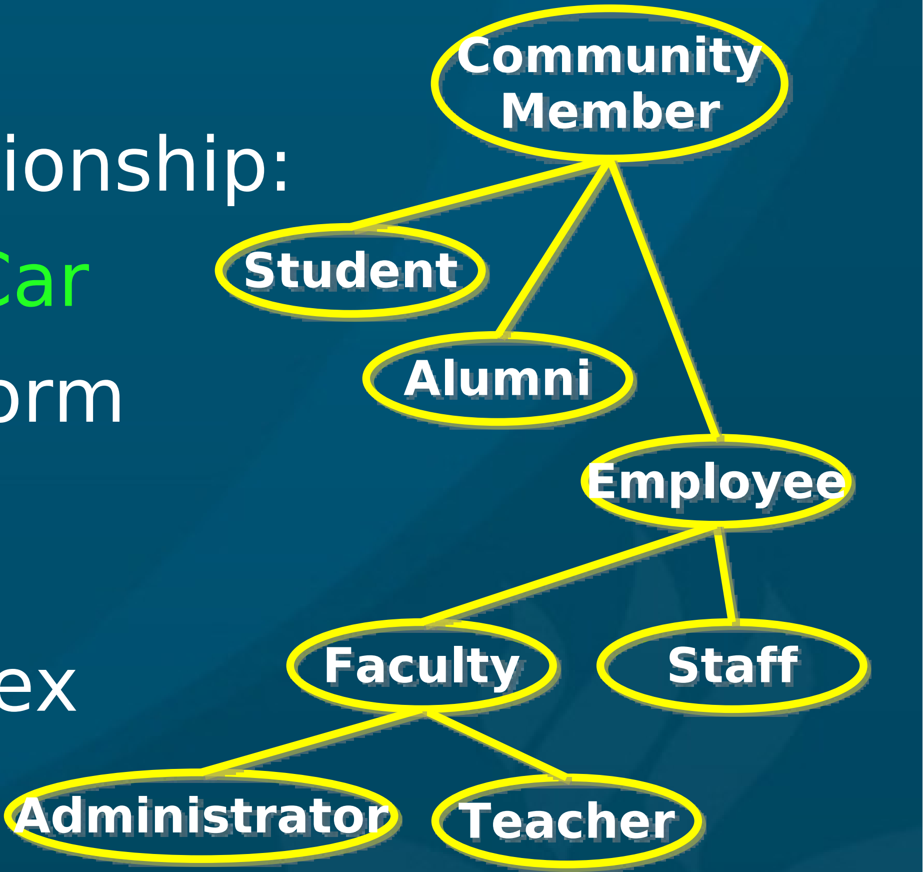
```
        for (int elt : array)
            sum += elt;
```

- But note elt is a copy of each element:
  - Can't use this to modify array:

```
        for (int elt : array)
            elt *= 2;              // doesn't change array!
```

Superclasses and subclasses

- Attribute: "has a" relationship:
  - A Car has a steeringWheel
- Subclass: "is a kind of" relationship:
  - A Convertible is a kind of Car
  - Inheritance relationships form tree-like class hierarchies
  - "extends": more specific, less inclusive, more complex
- Polymorphism: write once
  - changeOil() method works on all Cars, not just Convertibles



Why use inheritance?

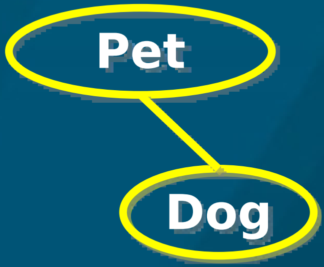
- Reusability
  - Create new classes from existing ones
    - Absorb attributes and behaviours
    - Add new capabilities
- Polymorphism
    - Enable developers to write programs with a general design
    - A single program can handle a variety of existing and future classes
    - Aids in extending program, adding new capabilities

Subclassing in Java

- When declaring a class, indicate its superclass (parent):
    - **public class Dog extends Pet { ....**
  - A Dog is a kind of Pet
  - Inherits everything Pet has
  - Can add Dog-specific attribs/methods
  - Can override general Pet methods with Dog-specific versions

Using subclass instances

- An instance of a subclass can be treated as an instance of the superclass:
  - **Pet fluffy = new Dog();**
  - Cannot do vice-versa:
  - **Dog myDog = new Pet();** **// doesn't work!**



- instanceof checks the class of an object:
  - **if (fluffy instanceof Dog) { ...**
- A superclass reference may be downcast back to the subclass if appropriate:
  - **// this is ok because fluffy is really a Dog**
  - **Dog myDog = (Dog) fluffy;**

Overriding methods

- A subclass can override a method defined by the superclass
  - Every Pet knows how to speak()
  - But Dogs speak() differently from Cats
  - Subclasses override the speak() method
- Late binding: which version of speak() to use?
  - Decided at run-time
- Polymorphism: same code works on several different types, all subclasses of the same parent
- Contrast with overloading (type signature)