CS100M

Introduction to Computer Programming

Spring 2004
Types

Announcements

- GDIAC (The Game Design Initiative at Cornell)
- Open House:
  - Wed, May 12
  - 3:30-6:30
  - Upson 315, 319
  - Course info? CIS 300
- Final exam info
  - see Final Exam link on course website
  - early review: see leftover questions (arrays, inheritance, sorting, lists) on old Prelim3s
  - I'll also post more questions (1 or more old finals)

Motivation

- Problem solving
  - non-OOP
  - OOP
- Redunancy?
  - related classes that repeat code
  - want to avoid copying code
- Inheritance and subtyping to the rescue!

Type Taxonomy

- Thing
  - Place
    - ?
    - ?
  - Creature
    - ?
    - ?
- Ideas:
  - looking for classifications of classes
  - identify "higher" classifications of "lower" classes
  - high:
    - more general
    - can be expressed in many ways
  - low:
    - more specific, additional features not seen in general
    - cannot be used to classify other classes so easily

Why is this cool?

```
public class Test1 {
   public static void main(String[] args) {
      Person[] p = { new Student(123456),
                     new Faculty(123456) };
      for (int i=0; i<p.length; i++)
         System.out.println(p[i].getID());
   }
}

class Person {
   protected int id;
   public Person(int id) {this.id = id;}
   public int getID() { return id; }
}
class Student extends Person {
   public Student(int id) { super(id); }
}
class Faculty extends Person {
   public Faculty(int id) { super(id); this.id *= 10; }
}
```


Inheritance

- Terms:
  - ***inheritance***: code "copied" from one class to another
  - ***extensibility***: extend behavior of a class to another
  - ***code reuse***: copy code from one class to another
  - ***subtyping***: generalize notion of types (things can be other things)
  - more terms coming...
- Picture:

conceptual classes&code

Super/Sub Types

- Classifiction of type:
  - ***Supertype***, ***superclass***, ***base class***
  - ***Subtype***, ***subclass***, ***derived class***
- Relationship:
  - supertype variable can get value of that type or a subtype eg) `Animal x = new Platypus();`
  - need syntax to tell Java that `Animal` and `Platypus` are related
  - subtype variable get supertype value? need syntax (cast!)
- Three mechanisms for relating types in Java:
  - primitives—promotion for some!
  - inheritance—extending a class
  - interface—specifying the type, methods, constants for a class, but not the bodies of the methods

Primitives

- Not really part of inheritance, but helps
- Compare `double`s, `int`s, and `char`s
  - what's the supermost supertype?
  - what's the submost subtype?
  - what's the visualization?
- Relating supertypes and subtypes:
  - `double` can get `int`?
  - what happens to the `int` value?
- How to go in reverse?
  - can an `int` get a double?
  - what's the mechanism?
- Why does all of this work?
  - examples of the types are related (built-in)!

Example

```
public class Primitives {
  public static void main(String[] args) {

    double d;
    int i;

    d = 7.2; // ok? why?
    d = 7;   // ok? why?

    i = 7; // ok?
    d = i; // ok?
    // System.out.println(d); // output?

    // i = d; // is this is bad ... why?
  }
}
```


Inheritance: Intro

- OOP rem:
  - you define the types!
  - collect data and ops in one place (the class, also the type)
- To relate types:
  - find the nouns that will become classes
  - see if the classes are related somehow
  - connect the classes with new syntax and rules
- Syntax glimpse:

```
class Coin { } // most general
class Penny extends Coin { } // specific
class Dime extends Coin { }
class CanadianPenny extends Penny { }
```

- So... **`class Sub extends Super { }`**
- "Mostest generalest" class of all time?
- There's much more to come next lecture...

Inheritance Continued

- Upcasting:
  - supertype variable can store subtype reference
  - why? more general thing can be represented as more specific thing
  - eg: Human can be a Man, Human can be a Woman
  - eg: Coin can be a Penny, Coin can be a Dime
- Code:

```
class Human {}
class Woman extends Human {}
class Man extends Human {}
Human h1 = new Human(); // OK
Human h2 = new Man();   // OK
Human h3 = new Woman(); // OK
```


Upcasting

- Upcasting Syntax:
  `Supertype var = new Subtype(...)`
- Type on LHS:
  - variable is supertype
  - ref must be that supertype
- Type of object:
  - object still has its own type and knows its own type
  - useful for accessing methods!
- Type of RHS:
  - promotion: Java checks if object type extends the LHS supertype
  - if so, Java declares the value of the whole RHS as the supertype, which means the LHS matches the type
  - object's known type is NOT changed

Demo of Upcasting

```
class Human {}
class Woman extends Human {}
class Man extends Human {}

class Human {}
class Woman extends Human {}
class Man extends Human {}

public class UpCast {
  public static void main(String[] args) {
    Human h1 = new Human(); // OK
    Human h2 = new Man();   // OK
    Human h3 = new Woman(); // OK
    System.out.println(h2);
    System.out.println(new Woman() instanceof Human);
    System.out.println(h3 instanceof Human);
  }
}
```


Downcasting

- Can't always make a specific thing into a general thing
  - which of these is OK?
    A Dog is a Creature. A Creature is a Dog.
  - maybe the Creature in question happens to be a Dog.
  - need to provide more information assist!
- Syntax:
  `Sub var = (Sub) new Super(...)`
  - how to remember? **`int i = (int) 7.7`**
  - downcasting is not always legal
- Pattern:
  - upcasting is always legal for inheritance relationship
  - so, can use superclass variables to store "very sub" subclass objects, which can be used in "mid sub" refs
  - see next page...

Downcast Example

```
public class DownCast {
  public static void main(String[] args) {
    Coin c = new SteelPenny();
    Penny p = (Penny) c;
    System.out.println(p);
  }
}

class Coin { }
class Penny extends Coin {}
class SteelPenny extends Penny { }
```


More to Inheritance

- Still need to explain
  - how to automatically copy code
  - how to use privacy modifiers
  - how to override methods
  - how to chain constructors
  - design issues
  - all next lectures
- Back to types...
  - can you extend more than one class to share types?
    ```
    class Transgendered extends Man, Woman {}
    ```
  - sorry, *no* multiple inheritance (so, example above is bad!)
  - there's a workaround...

Interfaces

- ***Interface***: many uses and meanings
  - "sparse class" (constants, method headers)
  - specification to be implemented by a class
  - definition of a type (don't have to worry about class)
- Syntax:

```
interface ISomething {
   constants
   methodheaders
}
class C implements I1, I2, ... { ... }
```


Why Useful?

- Some rules for class:
  - class that implements an interface must define all the methods of the interface
  - why useful for developers? keeps consistent methods!
- Treating interface as a type

```
IName var = new something()
```

  - the object must implement the interface
  - if you say `var.method(...)`, the method header must be in the interface and implemented in the class
- Some interfaces are built-in:
  - `java.lang`: `Comparable` defines a `compareTo` method (see OOP lecture)
  - `java.util`: `Collection` has many data structure methods

Interface Example

```
public class Interfaces {
  public static void main(String[] args) {
    Coin[] c = {new Penny(), new Dime(), new Dime() };
    int pocket = 0;
    for (int i=0; i<c.length; i++)
      pocket += c[i].getValue();
    System.out.println(pocket);
  }
}

interface Coin {
  public int getValue();
}

class Penny implements Coin {
  public int getValue() { return 1; }
}

class Dime implements Coin {
  public int getValue() { return 10; }
}
```