Lecture 38: More about Interfaces

Last time:
1. Safe downcasting
2. `equals` reconsidered
3. Copying and cloning
4. Composition

Today:
1. Overloading and overriding revisited
2. Interfaces and class inheritance
3. Interface hierarchies
Inheritance Recap

- All objects are derived (directly or indirectly) from `Object`.
- Late binding and inheritance allows you to create polymorphic variables.
- When a method in a base class is not provided, the method and class are said to be `abstract`. Abstract methods may be implemented in (concrete) derived classes.
- Run-time information about class / type information of objects can be obtained using `getClass()` method and `instanceof` operator.
- **Upcasting** of object to superclass type is always safe and done automatically.
- **Downcasting** may not be safe, depending on actual class object belongs to at run-time. Safe downcasting can be done using `instanceof`.
- In presence of inheritance, issues arise in definitions of `equals()`, copying, because parameter may belong to subclass of argument type (e.g. in `Person` copy constructor, parameter may be `Student` rather than `Person`).
- For `equals()`, make argument type `Object` and check for class membership using `getClass()`.
- To copy objects, use `cloning`: embed information about copying in objects rather than classes by implementing `clone()` method to perform copying.
- **Composition** is another way to implement one class based on another. It is different from inheritance: an object in a class created with composition “has-a” instance of an object in the earlier class, as opposed to having an “is-a” relationship.
Overloading and Overriding Redux

- **Overloading**
  - Different methods can have same name; in this case, method name is **overloaded**
  - Overloaded methods must have different argument lists (different # of args, or different types)
  - Overloading can occur due to inheritance!
    - Recall old (inferior) treatment of equality in `Person` / `Student`
      ```java
      public class Person {
        ...
        public boolean equals (Person p) { ... }
      }
      public class Student {
        ...
        public boolean equals (Student p) { .... }
      }
      ``
    - **Student** has two (actually, three!) overloaded versions of **equals()**
      - Student version
      - Person version
      - Object version

- **Overriding**
  - A subclass can redefine an existing method from a superclass
  - When overriding a method the subclass method prototype must match **exactly the prototype** of the superclass (same name, same return type, same arguments). Otherwise: overloading!
  - You may change access specifier (public, private, protected), but derived classes cannot decrease the visibility.
Example

```java
public class Base {
    protected void m (int x) { ... }
}

public class Derived extends Base {
    public void m (int x) { ... }
    public int m (int x) { ... }
    public void m (double d) { ... }
}

// The following appears in the same package as above
Base b = new Base();
Base d = new Derived();
Derived e = new Derived();
b.m (5);
d.m (6);
d.m (7.0);
e.m (8.0);
```

- **Overriding:** with increased visibility
- **Error!** duplicate method declaration
- **Overloading**

Error! Since `d` is declared `Base`, the compiler looks for `Base:m(double)`. Doesn't exist! So this does not make it past the compiler, even though `Derived:m(double)` is defined!
Is `Object` Abstract?

- No!
- You can create instance of `Object` using
  ```java
  new Object o = new Object();
  ```
- But there’s not a lot you can do with them
Recall Interfaces

- Interfaces contain lists of method prototypes
- Example from Lecture #23:
  ```java
  public interface UMStudent {
    public void goToClass();
    public void study();
    public void add(int a, int b);
    public String getName();
  }
  ```
- Classes can be indicated as implementing interfaces
  ```java
  public class CSMajor implements UMStudent {
    ...
  }
  ```
  - To satisfy Java compiler, CSMajor must provide implementations of goToClass(), study(), etc.
- Interfaces can be used as types, and thus to support polymorphism:
  ```java
  public void psychoAnalyze(UMStudent student) { ... }
  ```
- From last time: interfaces are similar to, but different from, abstract classes
  - Abstract classes can contain abstract, concrete methods
  - Classes can implement multiple interfaces, but inherit (directly) from only one class
Main Uses of Interfaces

- API for classes
- Polymorphism
- “Faking multiple inheritance”
- Specifying sets of symbolic constants
“Multiple Inheritance”? 

- Intuitively useful to be able to inherit from multiple classes (multiple inheritance)

- But Java does not allow this
Why Does Java Disallow Multiple Inheritance?

- Semantic difficulties!
- Consider `StudentAthlete`
  - Objects would get name field from `Student`
  - Objects would also get name field from `Athlete`
  - Duplicate fields: what to do?
- Some languages (e.g. C++) do allow multiple inheritance
Can We Achieve Some of Benefits of Multiple Inheritance in Java?

- Yes, using interfaces + inheritance
  - Idea: use inheritance for one of inherited classes, interfaces for others
  - Interfaces ensure that relevant methods are implemented
- Example
  ```java
  public class Person { ... }

  public class Student extends Person { ... }

  public interface Athlete {
      public String getSport ();
      public void setSport (String sport);
  }

  public class StudentAthlete extends Student implements Athlete {
      ...
  }
  ```
- Objects of type `StudentAthlete` “are” `Student`
- They also can be wherever objects matching `Athlete` are required
Interfaces and Constants

- Interfaces can also contain `public final static` variables
- Sometimes interfaces are used to provide consistent definitions for constants throughout an application
- Example

```java
public interface Months {
    public final static int JANUARY = 1;
    public final static int FEBRUARY = 2;
    public final static int MARCH = 3;
    ...
    public final static int DECEMBER = 12;
}

public class MonthDemo implements Months {
    public static void main( String[ ] args ) {
        System.out.println( "March is month number " + MARCH );
    }
}

Because MonthDemo implements Months, it has access to the constants
```
Interface Hierarchies

- Inheritance may also be used to build new interfaces from previous ones
- A subinterface inherits all method / constant declarations from its base interface
- A subinterface may also introduce new methods / constants
- E.g. recall `Iterator<T>` interface

```java
public interface Iterator<T> {
    boolean hasNext( ); // any more items?
    T next( ); // return the next item
    void remove( ); // remove the current item
}
```

We can define a new, bidirectional iterator interface using inheritance

```java
public interface BidirectionalIterator<T> extends Iterator<T> { // any prior items?
    T previous(); // return the previous item
}
```