CSE 1720

Lecture 8

Inheritance, II

Announcements:

- Lectures 7-10 assigned reading: Ch 9, JBA
- Preparation for Labtes#2:
  - you will be asked to construct a generic collection of `Shape` objects
  - you will be asked to iterate over the elements of the collection
  - you will be asked to implement a condition on the basis of the type of the elements
    - e.g., iterate over the Shapes, draw each one as an unfilled shape except the Ellipse2D objects, which should be drawn as filled

Goals/To do:

- Good practices for the declaration and instantiation of objects within a class hierarchy
- Take advantage of polymorphism when desiging apps
- Create, modify, and iterate over a collection of Shapes; use services of `Graphics2D` for manipulating and/or operating upon the shape objects

Goals/To understand:

- understand a class in terms of its position within a hierarchy
- understand the `Object` class in terms of its position at the top of the class hierarchy
- recognize and understand subclass features from their APIs
- distinguish between early and late binding
- understand and distinguish among non-primitive types defined by: *classes*, *abstract classes* and *interfaces*.
- understand *generic collections*

Recap

- the methods defined in a child class fall under the following categories:
  - new methods; methods defined in the child class only and not defined in parent
  - inherited methods; methods defined in the parent class and thus also available to child instances
  - overridden methods; methods defined in the parent class and also defined in the child class; the child class provides another version of the method functionality that overrides the parent's method functionality

Binding

- Binding refers to the process of *resolving* an expression.
- *resolve* ≅ locate the referent of an identifier
  - recall an **identifier** means a variable, method or class name
  - the **referent** means the thing that the identifier stands for
    - the referent of a variable is its value
    - the referent of a class name is a class definition (the class body)
    - the referent of a method signature is a method body (the method name alone is insufficient)

Early Binding

sec 3.1.3,
sec 9.2.2

- consider the case of the statement
  `r.m(…)`
- the compiler needs to *resolve* this expression:
  - what is the declared type of r? **this is the referent class**
  - what is the signature of the invoked method?
  - find the signature in the **referent class definition**
    - same method name
    - same number of parameters
    - parameters of the same type or higher in the hierarchy
  - if multiple target methods found, choose the method that requires *the least amount of promotion*
  - ***generate bytecode, stipulate the signature in the bytecode***

Late Binding

sec 9.2.2

- consider the case of the bytecode corresponding to the statement:
  `r.m(…)`
- the VM needs to *resolve* this expression:
  - what is the class of the object to which r refers? **this is the referent class**
  - what does the bytecode say is the signature of the invoked method?
  - find this signature in the referent class definition
    - the VM *will always find a matching method*
    - the compiler's early binding ensures that at least one matching method can always be found (that of the parent class)
    - VM needs to resolve parameters and pass them to the matching method

Early Binding (Compiler)	**Late Binding (VM)**
– what is the declared type of `r`?	– what is the class of the object to which r refers?
– what is the signature of the invoked method?	– what does the bytecode say is the signature of the invoked method?
– is the target method found in the type definition (class definition)? • if multiple targets methods found, choose the method that requires the least amount of promotion	– **find that target method** in the class definition, resolve parameters and pass them to the matching method

generate bytecode, stipulate the signature in the bytecode

Exercise

- let's revisit an example that shows how early/late binding can resolve to different referents
- `L08App01`

Exercise

- let's consider the case of the method `isSimilar(CreditCard)`, which is defined in `CreditCard`
- it determines similarity simply on the basis of card balance (not name, number, issue or expiry dates)
- `L08App02`

- let's consider a situation in which early binding requires promotion
- `L08App03`

Exercise

- let's consider the case of the method `isSimilar(RewardCard)`, which is defined in `RewardCard`
- it determines similarity simply on the basis of card balance and points balance (not name, number, issue or expiry dates)
- `L08App04`

- let's consider a situation in which early binding **does not** require promotion, but late binding does
- `L08App05`

Abstract classes and Interfaces

- abstract classes and interfaces **define types**
- abstract classes and interfaces **are not allowed to have constructors**; only their children classes
- for an instance, need to use:
  - a factory method
  - constructor from a child class

Abstract classes vs. Interfaces

- methods:
  - an abstract class can implement methods which can then be inherited by all children classes
  - interfaces can stipulate method, but each child class provide its own implementation
- parenthood:
  - a child class can have only one parent (either an abstract class or regular class), as defined by “extends”
  - a child class can implement as many interfaces as needed

Revisit Graphics2D Drawing

java.awt.geom

Class Ellipse2D.Double

```
java.lang.Object
 └ java.awt.geom.RectangularShape
    └ java.awt.geom.Ellipse2D
       └ java.awt.geom.Ellipse2D.Double
```

All Implemented Interfaces:
Shape, Serializable, Cloneable

Any of these declarations are syntactically correct. Design-wise, which would be best?

```
Ellipse2D.Double ellipse = new Ellipse2D.Double(DIM / 2, DIM / 2,
                                                DIM / 10, DIM / 10);

Ellipse2D ellipse = new Ellipse2D.Double(DIM / 2, DIM / 2,
                                         DIM / 10, DIM / 10);

RectangularShape ellipse = new Ellipse2D.Double(DIM / 2, DIM / 2,
                                                DIM / 10, DIM / 10);

Shape ellipse = new Ellipse2D.Double(DIM / 2, DIM / 2,
                                     DIM / 10, DIM / 10);
```


Graphics2D Drawing

abstract void	draw(Shape s) Strokes the outline of a Shape using the settings of the current Graphics2D context.
abstract void	fill(Shape s) Fills the interior of a Shape using the settings of the Graphics2D context.

- *Design-wise, declare variables as “high up” in the class hierarchy as possible*
- *abstract away as much detail as possible*
- *this will serve to modularize your app*

Generic Collections

- we will now start the topic of *generic collections*
- ref: 9.3.3, which we will cover in more detail on Tuesday
- approach:
  - review the concept of a *collection*
  - introduce the concept of a *generic collection* (aka a *parameterized collection*)
  - examine two examples of parameterized collections

The Forest Gump way of defining a collection

A collection **is** what a collection **does.**

Does it have elements that I can traverse?
Does it let me add elements?
Does it let me remove elements?
Does it tell me its size?

Then it is a collection.*

*a collection does a few other things, but we will talk about these later

Collections, Generics

- The hypothetical class `Bag<T>` defines a *generic* collection (aka a *parameterized* type).
- `Bag` is **generically** a collection
  - as such, it has elements
  - what is the **type** of these elements?
    with generic collections, one commits to the element type at the time of instantiation

Collections, Generics Example

- the following uses a hypothetical class `Bag<T>` to define a *generic* collection

```
Bag<CreditCard> bag = new Bag<CreditCard>();
```

  - we commit to the elements of bag being `CreditCard` objects
  - the signature of the add method becomes **`add(CreditCard)`**
  - the generic signature of the add method was **`add(T)`**

- Here are some actual declarations/instantiations:

```
Set<CreditCard> set = new HashSet<CreditCard>();
List<CreditCard> list = new ArrayList<CreditCard>();
```


Collections, Generics Example

- Here are some actual declarations/instantiations:

```
Set<Shape> set = new HashSet<Shape>();
List<Shape> list = new ArrayList<Shape>();
```

- What types of objects could we add to these collections?
- eg. `L08App06`
- For the labtest, you will be asked to construct a generic collection of Shape objects and iterate over them. You will be asked to implement conditional behaviour (such as draw only the Ellipse2D objects, or draw some objects as filled and others an unfilled)