The Movie Class

Symphony Class
Classes and Objects

Java (and C++) classes correspond to categories; they are like blueprints or templates for creating instances (objects) that correspond to individuals.

Class definitions can include class and instance variables/constants plus class and instance methods. Each can have “public” or “private” access. For example,

class Symphony
{
    private int music, playing, conducting; // instance variables
    private static int numberOfSymphonies = 0; // class variable
    public Symphony (int m, int p, int c) // constructor
    {  music = m;  playing = p;  conducting = c;  numberOfSymphonies++; }
    public static int  getNumberOfSymphonies ()
    {   return numberOfSymphonies;   } // class method
    public double rating (double scale)
    {   return ( music + playing + conducting) * scale;  } // instance method
}
Objects as Biological Cells

See Taylor, David: Object Technology (2nd edition)

Cell Hierarchies and Inheritance
What is Inheritance?

**inheritance**: A way to form new classes based on existing classes, taking on their attributes/behavior.
- a way to group related classes
- a way to share code between ≥ 2 classes

**One class can extend another**, absorbing its data/behavior.
- *superclass*: The parent class being extended.
- *subclass*: The child class that extends the superclass and inherits its behavior.
- Subclass gets a copy of every field and method from superclass

Inheritance Example #1
Abstract Classes (a Preview)

Animal
   - Coldblood
   - Mammal
      - Fish
      - Hippo
      - Canine
      - Feline
         - Wolf
         - Dog
         - Tiger
         - Cat
         - Lion

Why Use Inheritance? An Example

Movie
   - rating()
   - acting
   - directing
   - script
   - name
   - timeInMinutes
   - getHours()
   - getName()
   - getMinutes()

Symphony
   - rating()
   - music
   - playing
   - conducting
   - name
   - timeInMinutes
   - getHours()
   - getName()
   - getMinutes()
Inheritance Example, continued

Suppose both the Movie class and the Symphony class each contain 
timelnMinutes and name instance variables, along with companion methods getMinutes(), setMinutes(), etc.

The problem is that these instance variables and instance methods are exact 
duplicates in both classes. Maintaining multiple copies makes software development and maintenance difficult as you try to correct bugs, add features, improve performance, change behavior, etc.

Instead: tie together classes in hierarchies such that instance features declared in one class automatically appear in instances belonging to another; this allows one to create "natural" category hierarchies by creating a subclass to direct-superclass relationship, using the pattern:

    class SubclassName extends SuperclassName {
        ...
    }

Movies and Symphonies are Attractions

For example,

    public class Movie2 extends Attraction {
        ...
    }

One can say that movies and symphonies are both "attractions," and define both instance variables and appropriate getter/setter methods just in this class:

    public class Attraction
    {
        protected String name;
        protected int timeInMinutes;
        public Attraction () { timeInMinutes = 75;}
        public int getHours () { ???? }
        public void setMinutes(int d) {minutes=d; }
    }

A class will inherit public and protected instance variables and methods from all of its superclasses.

See .java files Attraction, Movie2, Symphony2, and DemonstrateInheritance
Extending the Attraction Class

Shadowing

A class can have only one immediate superclass.

When a subclass-superclass chain contains multiple instance methods with the same name, argument number, and argument types, the one closest to the target instance in the subclass-superclass chain is the one executed. All others are shadowed / overridden. Consider

- public class StevenSpielbergMovie extends Movie2
  {
    public int rating() { return 10 + acting + script; }
  }

- StevenSpielbergMovie m = new StevenSpielbergMovie();
  int i = m.rating(); // which rating method is invoked?
Abstract Classes

To prevent the creation of instances of the *Attraction* class, define it as *abstract*.

You can, however, create *Attraction* variables and assign to them either *Movie2* or *Symphony2* instances.

WHY is that legal?

When you mark a class with the *final* keyword, such a class cannot be extended.

All classes form an inverted tree with the *Object* class at the root. *Final* classes appear as leaves.

It's All So ... final

A final variable means you *cannot change* its value; this includes parameters as well as non-static variables.

A final method means you *cannot override* the method.

A final class means you *cannot extend* the class (i.e., you can't make a subclass)
Abstract Methods

Problem identified:

- `Attraction x = new Movie2();
  System.out.println (x.rating()); // bug`

We need to define method `rating()` for Attraction

- Undesirable, since this `rating()` method is never called.

- Fortunately, Java allows you to define `rating()` as an abstract method of the abstract `Attraction` class: public abstract int rating();

- Once you have defined an abstract method, Java forces you to define corresponding non-abstract methods in certain sub-classes of the abstract class.

Scoping Visibility Rules

- Class
  - public
  - protected
  - default
  - private

- Class within same Package
  - public
  - protected
  - default

- Subclass within Package sees
  - public
  - protected
  - default

- Subclass outside of Package sees
  - public
  - protected

- Class outside of Package sees
  - public
Access Specifiers

- A **public** feature (data or method) can be accessed outside the class definition. A public class can be accessed outside the package in which it's declared.

- A **protected** feature can be accessed only within the class definition in which it appears, within other classes in the same package, or within the definition of subclasses.

- A **private** feature can be accessed only within the class definition in which it appears.

- A **default** feature can be accessed by subclasses and by other classes in the same package.

Placing a Piece on a Chessboard
How a Bishop Attacks

For convenience, define

- columnDiff = pieceColumn - indexColumn;
- rowDiff = pieceRow - indexRow;

Then the diagonal \ satisfies ...

And the diagonal / satisfies ...

For other squares,

```java
if ( columnDiff == rowDiff )
    System.out.print(" B");
else
    System.out.print(" W");
```

How a Knight Attacks

Again we calculate

- columnDiff = pieceColumn - indexColumn;
- rowDiff = pieceRow - indexRow;

Now the condition for printing a * is

```java
if ( columnDiff + rowDiff == 3 )
    System.out.print(" *");
```
Constructors and Inheritance

A child class can add new data and methods, but can also **override** methods defined in the parent class (so long as the name, return type and arguments match).

When you want a constructor to hand one or more arguments to another constructor in the direct superclass, you put as the FIRST statement in the subclass constructor a statement consisting of

```
super (argument list for parent's constructor) ;
```

If, on the other hand, you want a constructor to call another constructor IN THE SAME CLASS explicitly:

```
this (argument list for other constructor) ;
```

The super Constructor

A call to the base class constructor can never use the name of the base class, but uses the keyword **super** instead

A call to **super** must always be the first action taken in a constructor definition

An instance variable cannot be used as an argument to **super**

See files **C.java** and **SuperDemo.java**
The super Constructor, cont’d.

- If a derived class constructor does not include an invocation of super, then the no-argument constructor of the base class will automatically be invoked.
  - This can result in an error if the base class has not defined a no-argument constructor.
- Since the inherited instance variables should be initialized, and the base class constructor is designed to do that, then an explicit call to super should always be used.
- NOTE: SUPER CAN BE USED TO INVOKE ORDINARY (non-constructor) METHODS IN A SUPERCLASS.

The this Constructor

- Within the definition of a constructor for a class, this can be used as a name for invoking another constructor in the same class.
  - The same restrictions on how to use a call to super apply to the this constructor.
- If it is necessary to include a call to both super and this, the call using this must be made first, and then the constructor that is called must call super as its first action.