Class Hierarchy

Super-classes and Sub-classes:

- **Inheritance**: a common form of re-use in Object-oriented programming.
- **Hierarchies**: an arrangement of classes with a common functionality.
- **Superclass**: defines instance variables and methods common to all the classes lower in the hierarchy.
- **Sub-class**: a class that descends from, or inherits from, a superclass.
  - Can inherit instance variables.
  - Can inherit methods.
  - Or both!

Super-classes and Sub-classes (continued):

- Consider a class that represents an employee. What instance variables?
  - Employee number
  - Salary
  - Name
  - Address
  - …
- Can represent this using UML (Unified Modeling Language) diagram:

```
<table>
<thead>
<tr>
<th>Employee</th>
</tr>
</thead>
<tbody>
<tr>
<td>-empNumber: long</td>
</tr>
<tr>
<td>-empSalary: double</td>
</tr>
<tr>
<td>+getEmployeeNum: long</td>
</tr>
<tr>
<td>+setEmployeeNum(long num): void</td>
</tr>
<tr>
<td>+getSalary: double</td>
</tr>
<tr>
<td>-setSalary(double value): void</td>
</tr>
</tbody>
</table>
```

Name of the class

instance variables

methods
Super-classes and Sub-classes (continued):

- May have different types of employees. For example, at UA, there are different types of faculty.
  - Tenure track: Assistant Professor, Associate Professor, Professor, Regents Professor.
  - Teaching: Instructor, Lecturer, Sr Lecturer
  - Coach: …
  - …
- But
  - all are employees
  - all have employee number
  - all have salary.
- Sub-classes!

Super-classes and Sub-classes (continued):

- But, the UA has employees that are not faculty. How to represent them?
  - Divide employees into faculty and staff.
  - Then, have sub-classes for faculty.
Super-classes and Sub-classes (continued):

- Superclass can have multiple subclasses.
- Subclasses can be superclasses of other subclasses.
- Subclass **inherits** from its superclass.
  - Instance variables
  - Methods
- Subclass inherits from **exactly one** superclass.

- What is at the top of the hierarchy?
  - **Object**
  - See Java API for `java.lang.Object`

Super-classes and Sub-classes (continued):

- Can write code that is common to multiple classes once and reuse it in subclasses.
  - Employee class:
    - All employees pay income tax, and Social Security tax.
    - All employees have sick leave.
    - All employees have health insurance.
    - The methods for managing taxes, sick leave, and health insurance are the same.
  - Staff
    - Have vacation.
    - (Faculty do not have vacation.)
Super-classes and Sub-classes (continued):

- Syntax: use `extends` keyword in the definition of the class.

  ```java
  public class Staff extends Employee
  { ...
  ```

- Again, a class may have **exactly ONE** super-class.
  - Use `extends` at most once for each class.
  - If `extends` is not present, does the class have a super-class?

Inheritance:

- Methods that are in the super-class can be used by the sub-class.
- Instance variables in the super-class are generally declared to be private.

- Example, creating a teaching faculty:
  - employee number 1234567, salary 83000
  - rank "Senior Lecturer", contractLength 3

  ```java
  public Teaching(long number, double pay,
  String title, int length)
  { ...
  ```

  - Can use the `setRank()` and `setContractLength()` methods to initialize the two instance variables that are in `Teaching`.
  - But, what about the super-class? How does it get created? How do `empNumber` and `empSalary` get their values?
Inheritance (continued):

• Answer: call the constructor of the super-class:

```java
class Teaching
{
  public Teaching(long number, double pay,
                  String title, int length)
  {
    super(number, pay); // done first!
    setRank(title);
    setContractLength(length);
  }
}
```

• The call to `super()` goes one-level up the class hierarchy.

• Special note: The call to `super()` must be the first statement!
  
  - I.e., in the example above, the calls to `setRank()` and `setContractLength()` must come after the call to `super()`!

Inheritance (continued):

• What happens if there is not call to the super-class constructor?
  
  - For example, `Card.java` extends `Object`, but you did not call `Object`'s constructor.
  
  - If the constructor for a class does not include a call to `super`, then the Java compiler will put in a call to `super()`
    
    - This is the no-parameter constructor for the super class.
    
    - This is why it is generally a good practice to have a no-parameter constructor for classes you write.
    
    - If/when a sub-class is created for your class, you can
Inheritance (continued):

- Consider `toString()` for the `Teaching` class:
  - Want to print the `rank` and `contractLength`
  - AND, want to print the `empNumber` and `empSalary`.
  - How?

```java
public String toString() {
    String temp;
    temp = "Rank: " + getRank() + " Contract: " + getContractLength() + " Employee Number: " + getEmpNumber() + " Salary: $" + getEmpSalary();
    return temp;
}
```

- Consider the `Hand` class from Program 3. What is the super class for `Hand`?

```java
import java.util.Random;

public class HandExample1 {
    public static void main(String[] args) {
        Random shuffle;
        long seed;
        Object cards;
        cards = new Object();
        System.out.println(cards);
        cards = new Object();
        System.out.println(cards);
    } // main
} // HandExample1
```

- What gets printed? Yes, the `Object` class has a `toString()` method. What does it print?
Inheritance (continued):

- Declare cards to be `Object`, but invoke the constructor for `Hand`:

```java
import java.util.Random;

public class HandExample2
{
    public static void main( String [] args )
    {
        Random shuffle;
        long seed;

        Object cards;

        seed = 17;
        shuffle = new Random( seed );
        cards = new Hand( shuffle );
        System.out.printf("For seed = %d, the hand contains:\n", seed);
        System.out.println(cards);
    } // main
} // HandExample2
```

- What gets printed? Which `toString()` method gets invoked: the one for `Object` or the one for `Hand`?

Casting:

- The seed 4 generates a `Hand` that contains a pair. Can we find the pair?

```java
import java.util.Random;

public class HandExample3
{
    public static void main( String [] args )
    {
        long seed = 4;
        Random shuffle = new Random( seed );

        Object cards;

        cards = new Hand( shuffle );
        System.out.printf("For seed = %d, the hand contains:\n", seed);
        System.out.println( cards );

        if ( cards.pair() )
        {
            System.out.println("Found a pair");
        }
        else
        {
            System.out.println("No pair found");
        }
    } // main
} // HandExample3
```

This works (compiles and runs). Uses the `toString()` method from the `Hand` class. Why?

This does not work. The compiler says:

```
HandExample3.java:16: error: cannot find symbol
    if ( cards.pair() )
                      ^
  symbol:   method pair()
location: variable cards of type Object
```

```java
HandExample3.java:16: error: cannot find symbol
    if ( cards.pair() )
                      ^
  symbol:   method pair()
location: variable cards of type Object
```
Casting (continued):

- Since cards was created using the constructor for Hand:
  - We can try to cast cards to be a Hand. I.e.,
    ```java
    if ( (Hand)cards.pair() )
        System.out.println("Found a pair");
    else
        System.out.println("No pair found");
    ```
  - But, the compiler then says:
    ```java
    HandExample3.java:16: error: cannot find symbol
       if ( (Hand)cards.pair() )
          ^
    symbol:   method pair()
    location: variable cards of type Object
    HandExample3.java:16: error: incompatible types
       if ( (Hand)cards.pair() )
          ^
    required: boolean
    found:    Hand
    2 errors
    ```
  - We now have two errors instead of one -- Argh!
  - The compiler still considers the call to `pair()` to be invalid, since cards is of type Object.
  - Why did the cast not work?

Casting (continued):

- The compiler is looking for a method named `pair()` in `Object` and does this before applying the cast.
- Need to change the order!
  - We can try to cast cards to be a Hand. I.e.,
    ```java
    if ( ((Hand)cards).pair() )
        System.out.println("Found a pair");
    else
        System.out.println("No pair found");
    ```
  - Tell the compiler to cast `cards` to be of type `Hand`, and then look for the `pair` method.
  - The compiler will now use the `pair` method from `Hand`. 
Inheriting members:

- **private** Members:
  - Are not inherited (but are a part of the subclass).
  - A *Teaching* object cannot directly access:
    ```java
    private long empNumber;
    ```
  - Would use the `setEmpNumber` and `getEmpNumber` methods.
  - These methods would be **public**.

- This simplifies maintenance because the *Employee* class enforces the data validation rules for `empNumber`.

Inheriting members:

- **protected** Members:
  - Methods and instance variables defined as protected are **inherited** by subclasses
    - But, are still hidden from client classes.
  - Also, any class in the same package as the superclass can directly access a protected field, even if that class is not a subclass.
    - (Yes, this can get confusing!)
  - If `empNumber` is declared:
    ```java
    protected long empNumber;
    ```
    - Then, the code for *Teaching* can directly use `empNumber`
    - Would not need to use the `setEmpNumber` and `getEmpNumber` methods.

- Disadvantage:
  - Because more than one class can directly access a protected field, the use of `protected` complicates maintenance.
  - Strongly recommend **private**, rather than `protected`, for instance variables!
Inheriting members (continued):

<table>
<thead>
<tr>
<th>Superclass Members</th>
<th>Inherited by subclass?</th>
<th>Directly Accessible by subclass?</th>
<th>Directly Accessible by client of subclass?</th>
</tr>
</thead>
<tbody>
<tr>
<td>public fields</td>
<td>yes</td>
<td>yes, by using field name</td>
<td>yes</td>
</tr>
<tr>
<td>public methods</td>
<td>yes</td>
<td>yes, by calling method from subclass methods</td>
<td>yes</td>
</tr>
<tr>
<td>protected fields</td>
<td>yes</td>
<td>yes, by using field name</td>
<td>no, must call get and set methods</td>
</tr>
<tr>
<td>protected methods</td>
<td>yes</td>
<td>yes, by calling method from subclass methods</td>
<td>no</td>
</tr>
<tr>
<td>private fields</td>
<td>no</td>
<td>no, must call get and set methods</td>
<td>no, must call get and set methods</td>
</tr>
<tr>
<td>private methods</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

Overriding Methods:

- Teaching would have a toString method, and Employee also has a toString method.
  - And, Object has a toString method.
  - Which method is called?
    ```java
    Teaching me = new Teaching(...);
    System.out.println( me );
    ```
    - The toString of Teaching is called.
  - The toString of Teaching can directly access the rank and contractLength fields.
  - But, what if Teaching wants to also print empNumber and empSalary.
    - Could use:
      ```java
      public String toString()
      {
        String temp;
        temp = "Rank: " + getRank() + " " +
                "Contract: " + getContractLength() + " " +
                "Employee Number: " + getEmpNumber() +
                "Salary: $" + getEmpSalary();
        return temp;
      }
      ```
Overriding Methods (continued):

- The `toString` method of Teaching *overrides* the `toString` method of the super class.
- To override a method, you create a method with the same name and the same type and number of arguments.
- The return type should also be the same.

- But, what if we really want to use the `toString` of Teaching and the `toString` of Employee?
  - We can!
  ```java
  public String toString()
  {
      String temp;
      temp = "Rank: " + getRank() + " " +
             "Contract: " + getContractLength() + " " +
             super.toString();
      return temp;
  }
  ```
  - Tells the Java compiler to invoke the `toString` method of Employee.
  - How do we get to the `toString` of Object?
    - We cannot do so from Teaching.
    - But, Employee's `toString` method could invoke Object's `toString`.

abstract Classes and Methods:

- An *abstract* class is one that is not completely implemented.
  - Usually, there is at least one (maybe many) methods declared to be abstract.
  - An abstract method specifies an API, but does not provide an implementation.
    - The return type and the arguments are specified.
    - There is no code for the method.
  - The abstract method is a pattern for a method that a subclass must implement.

  ```java
  public abstract class Figure
  {
      private int x;
      private int y;
      private Color color;
      // Can be constructors, get and set methods for instance variables
      // May not know how to do the drawing, since it may depend on the device
      public abstract void draw( Graphics g );
  }
  ```

  Note: the semi-colon for the method.
  No `{ }`'s for the method.
  It is abstract!
abstract Classes and Methods (continued):

- You can declare variables of type Figure.
- But, you need a sub-class to implement the missing methods.

```java
public class Square extends Figure
{
    private int length;    // Can add more instance variables
    // Can be constructors, get and set methods for instance variables added
    // Can make use of any non-abstract methods from Figure
    // Must provide the code for the abstract draw method.
    public void draw( Graphics g ) {
        // Not declared abstract here!
    } // draw
}

public abstract class Figure
{
    private int x;
    private int y;
    private Color color;
    // Can be constructors, get and
    // set methods for instance variables
    public abstract void draw( Graphics g );
}
```