Inheritance

- UML Diagrams

- What is inheritance in Java?
- Why use inheritance?

UML Diagrams

- UML stands for the Unified Modeling Language
- *UML diagrams* show relationships among classes and objects
- A UML *class diagram* consists of one or more classes, each with sections for the class name, attributes, and methods
- Lines between classes represent *associations* (one object refers to another)
- *Associations* can show *multiplicity* (ratio of object association, e.g., 1-1, 1-2)
A UML class diagram for the *FlipRace* program:

<table>
<thead>
<tr>
<th>FlipRace</th>
<th>Coin</th>
</tr>
</thead>
<tbody>
<tr>
<td>main (args : String[]) : void</td>
<td>face : int</td>
</tr>
</tbody>
</table>

| flip() : void | isHeads() : boolean | toString() : String |

A UML *object diagram* consists of one or more instantiated objects.

It is a snapshot of the objects during an executing program, showing data values.

<table>
<thead>
<tr>
<th>coin1 : Coin</th>
<th>coin2 : Coin</th>
</tr>
</thead>
<tbody>
<tr>
<td>face = 0</td>
<td>face = 1</td>
</tr>
</tbody>
</table>
Encapsulation Review

- Two views of an object:
  - internal - the variables the object holds and the methods that make the object useful within the code of the object
  - external - the services that an object provides and how the object interacts with code outside of the code of the object

- The external view is more restricted and we see an object as an encapsulated entity, providing a set of specific services that define the object's interface.

- The goal is to hide details from the rest of the system and support an abstraction.

Encapsulation Review

- An object should be self-governing.
- Any changes to the object’s state (its variables) should be made only by that object’s methods.
- We should make it difficult, if not impossible, to access an object’s variables other than via its methods.
- The user, or client, of an object can request its services, but it should not have to be aware of how those services are accomplished.
Visibility Review

- In Java, we accomplish encapsulation through the appropriate use of visibility modifiers: public, protected, and private

- Members of a class that are declared with public visibility can be accessed from anywhere

- Public variables violate encapsulation

- Members of a class that are declared with private visibility can only be accessed from inside the class

- Members declared without a visibility modifier have default visibility and can be accessed by any class in the same package

Visibility Modifiers in a Class

<table>
<thead>
<tr>
<th></th>
<th>public</th>
<th>private</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code inside the class</td>
<td>Visible</td>
<td></td>
</tr>
<tr>
<td>Code outside the class</td>
<td>Visible</td>
<td>Invisible - Can't be accessed</td>
</tr>
</tbody>
</table>
Inheritance

- To tailor a derived class, the programmer can add new variables or methods, or can modify the inherited ones.
- *Software reuse* is at the heart of inheritance.
- By using existing software components to create new ones, we capitalize on all the effort that went into the design, implementation, and testing of the existing software.
- Inheritance permits us to factor out *commonality*, both in the design and code.
- Promotes ease of implementation, extension, comprehension.

---

Inheritance

- *Inheritance* allows a software developer to derive a new class from an existing one.
- The existing class is called the *parent class*, or *superclass*, or *base class*.
- The new class is called the *child class* or *subclass*, or *derived* class.
- As the name implies, the child *inherits* characteristics of the parent.
- That is, the child class inherits the methods and data defined for the parent class as if they were defined in the child class.
Inheritance

Inheritance relationships often are shown graphically in a UML class diagram, with an arrow with an open arrowhead pointing to the parent class.

Vehicle

Car

Inheritance should create an is-a relationship, meaning the child is a more specific version of the parent.

Deriving Subclasses

In Java, we use the reserved word `extends` to establish an inheritance relationship.

```java
class Car extends Vehicle {
    // class contents
}
```

- See `Words.java`
- See `Book.java`
- See `Dictionary.java`
Inheriting methods

- webster is a Dictionary object
  - definitionMessage is a method of Dictionary
  - Thus, webster.definitionMessage() works
- But pageMessage is a method of Book
  - However, Dictionary inherits from Book
  - Because of inheritance, a Dictionary is a Book
  - Thus, webster.pageMessage() works

The protected Modifier

- Visibility modifiers determine which class members are inherited and which are not
- Variables and methods declared with public visibility are inherited; those with private visibility are not
- But public variables violate the principle of encapsulation
- There is a third visibility modifier that helps in inheritance situations: protected
The protected Modifier

- The protected modifier allows a member of a base class to be inherited into a child
- Protected visibility provides more encapsulation than public visibility does (see Appendix F)
  - For subclasses, protected is like public
  - For other classes, protected is like private
- Protected is better than public, but does lose some encapsulation
- Protected variables and methods can be shown with a # symbol preceding them in UML diagrams

UML Diagram for Words

```
Book
# pages : int
+ pageMessage() : void

Words
+ main (args : String[]) : void

Dictionary
- definitions : int
+ definitionMessage() : void
```
## Visibility Modifiers in a Class

<table>
<thead>
<tr>
<th>Code inside the class</th>
<th>public</th>
<th>protected</th>
<th>private</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Visible</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code of a child of the class</td>
<td>Visible</td>
<td></td>
<td>Invisible - No access</td>
</tr>
<tr>
<td>Code outside the class</td>
<td>Visible</td>
<td>Invisible - No access</td>
<td></td>
</tr>
</tbody>
</table>