Object Oriented Building Blocks

- Inheritance
- Polymorphism
- Dynamic Binding
- Interfaces
- Event Listener Model

Why Inheritance?

- A primary goal of OO Design is **reuse**
  - Use software that already exists and works correctly
  - Speed up design - don't re-invent
  - Improves quality of software
  - Improves maintenance of software
- Aggregation of component classes into composite classes is a form of reuse
- But how do we reuse a class that does mostly what we need, but not quite enough?
  - Or does some things differently from what we need?
- Use **inheritance**!
What is Inheritance?

- In aggregation, one class **contains** other classes as parts
  - Called the "**has-a**" relationship
  - The container class **has** the components as parts
- In inheritance, one class **extends** the functionality of another class
  - Existing class serves as **base**
  - Inheriting class is **derived** from base
  - Called the "**is-a**" relationship
  - The derived class **is** an extension of the base
- Other terminology for inheritance
  - Inherited class is **subtype** of **supertype**
  - Inherited class is **child** class of **parent** class

Benefits of Inheritance

- Everything that is true for the base is true for the derived (extended) class
  - All behavior of the base is inherited by the derived class
  - E.g., if the base class has a method, then the derived class has the same method – without having to code it again
  - Wherever an object of the base type can be used, an object of the derived type may also be used
    - Since the derived object "is-a" base object
  - But the derived type extends the base type
    - Additional behavior (methods) may be added
    - Some inherited behavior may be overridden
  - Inheritance is a form of specialization or customization
    - Can design common abstractions and specialize them
Example of Inheritance

- A company has various workers
  - All are Employees – this makes a good base class
  - Some are Hourly, but they are still Employees
  - Others are Salaried, but still Employees
  - Some are Managers, but still Salaried and still Employees
  - Some Managers are company Officers (but still Managers, and still Salaried, and still Employees)
- Inheritance relationships form a hierarchy
- A good design captures common behaviors at the right level in the hierarchy, e.g.,
  - All Employees get paid
  - But pay calculation is done differently for Hourly vs. Salaried
  - Additional features for some classes – Officers get stock options

Inheritance Hierarchy

- UML like diagram
Another Example of Inheritance

- Video game
  - Base class Character with properties like location, ability to move, be targeted, etc.
  - Inheritance hierarchy of specializations of Character

```
public class ClubCard extends TuneCard {
    // That's all it takes to specify inheritance

    // The addition of extends TuneCard means that everything about class TuneCard also works for ClubCard
    // A ClubCard is a TuneCard
    // Where ever a TuneCard object could be used, a ClubCard object could be used as well

    // Since ClubCard is a class, it could also be a base for another class definition to extend
```

Java Syntax for Inheritance
Inheriting Methods

- All public methods of a base class are inherited by the derived class
  - They may be called for a derived object
  - Private methods of the base are still private

```java
public class ClubCard extends TunesCard {
    ... // Additional methods, data for club members
}
```

```java
TunesCard card = new ClubCard();
```

```java
card.addValue(10);
```

Constructors

- Constructors are **not** inherited, but
  - Initializing a subclass requires proper initialization of superclass
  - The constructor of the derived class automatically calls the constructor of the base
  - We can also control the base constructor explicitly:
    ```java
    public class Derived extends Base {
        Derived() {
            super();
        ...
        //Initialization specific to Derived
    }
    ```

- Arguments to a base constructor may be given if needed
  ```java
  super(117, x); // Base expects int and double
  ```
- If no arguments, then base must have void constructor
Overriding Methods

- Methods are automatically inherited by the sub class.
- The sub class may define additional methods with different signatures.
- The sub class may override an inherited method's definition by defining the same method.
  - Signature must match exactly to override inherited definition.
  - Even though inherited method is overridden, the super class's method may be called explicitly with the qualification `super`.

```java
class Base {
    void doStuff() {
    }
}
class Derived extends Base {
    void doStuff() {
        doSomeDerivedStuff(); // pre-conditions
        super.doStuff(); // Now do the Base stuff
        . . . // post-conditions
    }
}
```

Inheritance Example

- Complete example
  - `TunesCard.java`  `ClubCard.java`  `TunesCardTest.java`

- Notes
  - `ClubCard` has additional data and methods.
  - Balance is manipulated using the inherited methods — it is private and cannot be directly accessed by `ClubCard`.
  - The encapsulation of `TunesCard` is preserved.
  - Constructor of `ClubCard` calls constructor of `TunesCard` with name parameter value.
  - `ClubCard` overrides inherited `toString`, but still uses the method from `TunesCard`.
What about protected?

- **protected** is another access modifier which means the same as private, except for subclasses.

<table>
<thead>
<tr>
<th>Access Modifier</th>
<th>Code inside the class</th>
<th>Code of a sub class</th>
<th>Code outside the class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public</td>
<td>Visible</td>
<td>Visible</td>
<td>Visible</td>
</tr>
<tr>
<td>Protected</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Private</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dynamic Binding

- A derived class can override a definition of an inherited method.
- When we have an object of the derived type, the overridden implementation of the method is used.
- What if the derived object is assigned to a base type variable?
  - Is this legal?
    - Answer: Yes, since a derived object "is-a" instance of the base.
  - Which version of the method should be used? The base implementation or the derived?
    - Answer: The right one – since the object is really of the derived type, then that implementation should be used.
- This is called **dynamic binding**.
  - The right method implementation is determined by the actual object type, not just the variable declared type.
  - Dynamic binding supports **polymorphism** – many things that share similarities, but with instances behaving differently.

TunesCardTest2.java
Inheritance in Java

- There is a class `Object` built into Java
  - Every class defined has `Object` as a superclass
  - So everything in Java "is an" `Object`
  - Class `Object` has several methods

  - `boolean equals(Object o)`
    - Determines if two objects are equivalent
    - Inherited definition compares references, returns true if exactly the same object
    - Classes usually override this to define their own notion of equivalence

  - `String toString()`
    - Returns a String that represents the object
    - Inherited definition gives memory location
    - Classes usually override this definition to define their own way of representing the object as a String

  - Dynamic binding essential for these, other methods defined in a subclass
    - We want the correct comparison and representation
    - Want correct method for the data item
    - Even if all we know about the data items is that they are `Object`'s

Abstract Classes

- A class is defined as `abstract` if it is incomplete and can only be used as a base class for inheritance
  - The keyword `abstract` is used in the class definition
  - The class may have abstract methods – method headers only, with no body (also declared with abstract keyword)
  - No concrete instance of an abstract class may be created with `new` (but we may have variables of the abstract type)

- Example: A Vehicle could be an abstract class
  - There is no such thing as just a Vehicle
  - But there are concrete subclasses of Vehicle: Car and Bicycle
  - The abstract base can specify behavior that must be implemented in the subclasses, e.g., steering
  - We could have an array of Vehicles – some of them Cars and some Bicycles
InterfVICES

- An interface is like a class definition
  - It contains only method headers – but no method bodies
  - It may have constants – but no instance variables
  - Defined with keyword interface instead of class

- An interface is used like a base class
  - The keyword implements is required instead of extends
  - The implementing class must provide bodies for methods matching the method headers of the interface

- An interface is a "contract": the implementing class agrees to provide concrete definitions of the methods given in the interface

- A class may implement several different interfaces
  - This is the way that Java provides the effect of multiple inheritance

- An interface cannot be instantiated
  - But you can have a variable of type interface and assign to it a concrete object of a type that implements the interface

The Java API has many interfaces

- Comparable: must implement a compareTo method that allows objects to be compared
- Iterator: must implement methods next, hasNext to iterate over a set
- Many graphical components implement interfaces (ActionListener, etc.)

For example, if we have a list of Comparable objects, we could code a sorting algorithm that uses the compareTo method guaranteed by the interface

- We don't need to know anything else about the objects
- Strings and BigInteger implement Comparable
- Interfaces allow polymorphic treatment of collections
Graphical User Interfaces

A GUI in Java is created with at least three kinds of objects:
- Components
- Events
- Listeners

Components are frames, panels, buttons, text boxes, menus, etc.
- We won't discuss these much, but may see them used in examples

We want to understand the event-listener model
- It is a good example of the use of Java interfaces
- It illustrates issues of software architecture and access control

Events and Listeners

An event is some activity or occurrence to which we may want to respond
- Mouse clicks, mouse movement, key presses, timer going off, button presses, menu selections

Many components in the Java API generate events
- A graphical button object generates an event when the button is pushed
- A panel tracks mouse movements and clicks in it

We would like our program to be able to handle events
- We want to perform some action when the event occurs
- We need to have code that "listens" for events to occur, and responds appropriately
Events and Listeners

A component object that generates an event

A corresponding listener object is designed to respond to the event

When the event occurs, the component calls the appropriate method of the listener, passing an object that describes the event.

- Java API defines interfaces for listeners of events
  - The interface specifies the header of the method that must be implemented to handle the event

Listener Interfaces

- A listener object is created by writing a class that implements a particular Java listener interface
- The API has several interfaces that correspond to particular event categories
  - MouseListener interface contains methods that correspond to mouse events
  - The implementation of these methods specifies what we want to do when the mouse event occurs
- After the listener object is created, it must be added to the component that might generate the event
  - This sets up a formal relationship between the generator of the event and the listener
  - Because the listener implements an interface, the event generator knows how to call it
Listener Example

- **ActionListener** is the *interface* for handling an action event
  - It defines only one method, called `actionPerformed`
  - This method is called when the event occurs
- In this example, **PushButtonListener** implements the **ActionListener** interface
  - It is the listener for the event generated by the button
  - Implemented as an *inner class* (i.e., it is defined within another class)
  - Inner classes should only be used when there is a close relationship among the classes and the inner class is not needed in another context
- When the button is pushed, the **JButton** object invokes the **actionPerformed** method, passing it an **ActionEvent**
  - The listener method may or may not make use of the event object
- Another inner class implements **MouseListener** and **MouseMotionListener** interfaces to handle mouse events

*Sweep.java*