Object Oriented Programming

- Encapsulation and Data Hiding
- Classes and Instances
- Constructors
- Objects and References
- Inheritance

OO Design Principles

- **Abstraction**: capture concepts in design
  - Promotes reuse and versatility of software
- **Encapsulation**: classes define data that belongs together
  - Presents coherent view of an object
- **Data Hiding**: users of an object should not need to know the implementation details of the class
  - Users should see just the methods that the class author chooses to make visible
- Java provides language constructs to allow code to adhere to these principles
  - **Class** definition
  - Visibility modifiers

Objects

- An **object** models a real world entity or abstraction
  - An object has a **state**, which is modeled in the **data**
  - An object has **behavior**, which is modeled in **methods**
  - For example, a gift card is an entity that can be modeled in software
    - The state could consist of the card owner’s name, the current balance, and the account number
    - The behavior could consist of methods to add value to the card, make purchases, and check the balance
    - The class definition for a card would encapsulate the data and methods
    - Then we could create a single card object rather than having separate, loosely connected variables for the name, balance, and number

Classes and Objects

First TunesCard object
- **Data**: name is John Doe
  - cardNo is 1234
  - balance is $25.00

Second TunesCard object
- **Data**: name is Mary Smith
  - cardNo is 9876
  - balance is $32.81

Third TunesCard object
- **Data**: name is Lisa White
  - cardNo is 9999
  - balance is $35.12
public class TunesCard {
    private int cardNo;
    private double balance;
    private String owner;

    // Add given amount - adjust balance and return
    public double addValue(double amount) {
        balance = balance + amount;
        return balance;
    }

    // Returns current balance in the account
    public double getBalance() {
        return balance;
    }
}

public class TunesCardTest {
    public static void main(String[] args) {
        TunesCard hers = new TunesCard("Mary D");
        TunesCard his = new TunesCard("John S");
        . . .
    }
}

COMP 110 Fall 2010
Week 7 - Classes

Notes
- A "driver" program is often used to test a class
- If a class provides a toString method, then the object may be passed as an argument to System.out.println

Recall that a variable's scope is where it can be used
- The scope of a local variable is the block in which it is declared
- The scope of a public class method is everywhere
- But an object of the class type is needed to qualify the name
- The scope of a private class variable or method is the code of the class
- Within a class, no qualification of a name is necessary for public or private identifiers
  - The name implicitly refers to the current instance data
  - But the identifier can be qualified with this to make explicit
  - this is the "name" of the current object instance
Scope and this

- Suppose that in the TunesCard constructor we wanted to name the parameter `owner` instead of `who`

```
public TunesCard(String owner) {
    owner = owner;  // Oops! Both uses of owner refer to the parameter, so class data does not get changed
    balance = 0;
}
```

- Use keyword `this` to qualify the data member so it is distinguished from the parameter

```
public TunesCard(String owner) {
    this.owner = owner; // Now left side refers to class data, and right side to parameter
    balance = 0;
}
```

Static vs. Non-static

- Math class has **static** methods
  - No need for a Math object (can't actually create one anyway)
  - Static methods do not (can not) use instance data
- Non-static (**instance**) methods only make sense in context of a particular object
  - Can only be called with an explicit object (or implicitly from another instance method – same object)
  - Instance methods use the instance data
- What about static data?
  - Not associated with an object
  - Exists independently of the creation of any objects
  - This is **per-class** data, not per-object data
  - Useful for things like counters for the class, or even a count of the number of instances of objects of the class

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 Salared, but still Employees
  - Some are Managers, but still Salared and still Employees
  - Some Managers are company Officers (but still Mangers, and still Salared, 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. Salared
    - 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
Inheritance Example

- Recall TunesCard example
  - A TunesCard had a name, account number, balance
- Suppose we want a special version of a TunesCard that earns bonus points
  - New class will be called ClubCard
  - It does everything TunesCard does and more
  - We should not have to rewrite or copy TunesCard code

```
public class ClubCard extends TuneCard {
    // Additional methods, data for club members
}
```

Java Syntax for Inheritance

- Derived class definition specifies that it extends the base class
  ```java
  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 TunesCard
- Where ever a TunesCard 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

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

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

```
TunesCard card = new ClubCard();
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() {
           //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() {
            // overridden method
            ...
        }
    }
}
```

Inheritance Example

- Complete example
  - TunesCard.java  ClubCard.java  TunesCardTest2.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

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

```java
TunesCardTest3.java
```

Interfaces

- 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
Interfaces

- 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

Objects, Memory and References

- The built-in types are called `primitive` data types
  - `int`, `long`, `float`, `double`, `char`, `byte`, `boolean`
  - When we declare a variable of this type, appropriate space is reserved and initialized
  - E.g., for a local variable of type `double`, 8 bytes of the stack is used
- `Object` types are handled differently
  - When a variable of any object type is declared, space is only reserved for a `reference` to the object
  - Think of a `reference` as a “handle” - a memory address (4 bytes)
  - The reference is initially set to an invalid address (the `null` pointer)
  - Space for the object itself is only reserved when the `new` operation is performed
  - For Strings, initialization to a literal is shorthand for a `new` operation
  - The object will be in a separate memory pool called the `heap`

Objects and References

- If an object variable has not been associated with a valid object, then we cannot access methods or data through the variable

  ```java
  StringBuffer x, y;
  x.append("World");
  y.append("Hello");
  ```

  `null` causes null pointer exception since there is no object

  ```java
  y = new StringBuffer("Hello");
  ```

- Remember – all objects must be created
  - Primitive types are automatically "created" by the declaration

Objects and Assignment

- When we assign one `primitive` type variable to another, the `value is copied` from the one variable to the other

  ```java
  int a = 13;
  int b = 7;
  ```

  after `b = a`;

  ```java
  a 13
  b 7
  ```

- What about when one `object` variable is assigned to another?
  - The `reference` to one object is reassigned
  - This means there are now two variables referring to the `same` object
  - There is only one object now

  ```java
  Card a = new Card("John");
  Card b = new Card("Mary");
  ```

  after `b = a`;

  ```java
  a John / 1234 / 0.0
  b Mary / 1235 / 0.0
  ```
Object References

- Assignment of objects makes copies of the reference, not the object itself
  - The assigned variable becomes an alias for the same object
- Done implicitly when objects are passed to methods
  - The method gets a reference to the same object
- If the object is mutable, the object's state may be changed during the method call
  - Thus a method may produce side effects on the objects passed
- What happens to the object no longer being referenced after an assignment between object variables?
  - The memory used by the object is reclaimed at some point by the JVM automatic garbage collection
  - Otherwise memory would eventually run out

Strings as Objects

- What's wrong with this?
  ```java
  String msg;
  System.out.println(msg);
  String msg = "hello";
  System.out.println(msg.length());
  ```
  Null Pointer Exception since no String Object!
- A String object might look like
  ```java
  String msg = "hello";
  ```
  length of string | characters of string
  `h` | `e` | `l` | `l` | `o`

- When a String is created, memory is allocated for data
  - The characters
  - The length

Comparing Strings

- What does the following code do?
  ```java
  String a, t;
  if (s == t) . . .
  ```

- Scenario 1: `s = "hello";
  - `t = s;
  ```java
  s
  t
  a = t is true
  ```

- Scenario 2: `s = "hello";
  - `t = "world";
  ```java
  s
  t
  a = t is false
  ```

- Scenario 3: `s = "hello";
  - `t = s.substring(0, 4) + "o";
  ```java
  s
  t
  a = t is false
  ```

- The operator `==` asks if the variables refer to the same object, but that is not what we really want to know
  - We want to know if two strings match
  - I.e., are they equivalent - same letters in the same order

Comparing Strings

- String provides method `equals` to determine if Strings are the "same"
  - In OD fashion, Strings "know" when two Strings are equivalent
  - The method is given one argument - another String
  - This String compares itself to the other String
  - if (s.equals(t)) . . .
  - In Scenarios 1 and 3, `s.equals(t)` is true, but false in Scenario 2

- String has `compareTo` to determine how two Strings are ordered
  - Returns 0 if the Strings are equal (as above)
  - Returns distance between first differing characters (as positive or negative)
  - This produces lexicographic ordering of Strings

- Most objects implement an `equals` method (default is `==`)
  - Comparable objects implement a `compareTo` method

StringTest1.java