Object Oriented Programming

- Encapsulation and Data Hiding
- Classes and Instances
- Constructors
- Objects and References
- Standard Java Classes
- The String Class

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
Abstraction

- Why the fuss about objects, classes, and methods?
  - As a driver, you don’t need to know how the engine works to be able to drive a car
    - You understand the abstract concept of a car and the necessary interfaces (accelerator, steering, etc.)
    - You can drive many different models of cars – you don’t have to learn how to drive all over again for each car model
  - As an engineer at Ford, you don’t have to understand how every single component of every car works
    - You only need to know the component you work on and how it interacts with other components
    - You can design a standard component that can be used in different places in a car, and in different models

An abstraction hides the right details at the right time

- For example, we don’t have to know how the println method works in order to call it
  - println is an example of procedural abstraction
- A human being can manage only a few pieces of information at one time
  - But if we group information into chunks (such as objects) we can deal with many complicated pieces at once
- In Object Oriented design and programming, the emphasis of abstraction is on the data
  - An object is abstract in that we don’t have to think about its internal details in order to use it
  - Classes and objects help us write complex software
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

Examples of Objects

- The players in a game
  - State of each player includes position, strength, team, weapons
  - Weapons could be objects themselves with ammunition, range, accuracy, power as their state
- Desktop on a computer populated with items
  - Each item could be an object with various properties
  - Behavior could include how item repaints and resizes itself
- Files on the disk could be objects
- A music player, each song, albums

- Fundamental idea of OO is that data that belongs together should be encapsulated and "know" how to take care of itself
Using Classes

- We use classes by creating objects of the class type and calling methods for that object.
- Objects are created with the keyword `new` and the class name, followed by parentheses.
  - Using `new` means that an object is created by using a special constructor method of the class.
  - The constructor performs any initialization necessary for the object to work correctly.
  - The constructor may allow parameter values to be passed to affect the way initialization is done.
- Once an object is created, methods may be called.
  - Syntax is the object variable, a dot, the method name and any parameters.
- We have seen all of this with the Scanner class.

Java Strings

- We have been using the Java `String` type in programs.
  - We initialize Strings with literals in double quotes, e.g.,
    ```java
    String msg = "Hello, world";
    ```
  - We can perform some operations, e.g.,
    ```java
    msg = msg + "!";
    ```
  - Numbers can covert to Strings
    ```java
    msg = msg + " guest number " + n;
    ```
- Strings are objects, not primitive types.
  - Strings have methods, e.g., `msg.length()`.
  - Strings can be created with new, e.g.,
    ```java
    String msg = new String("Hello, world");
    ```
  - Initialization from literal is just shorthand for new.
  - Various constructors for Strings to create Strings from literals, other Strings, arrays of characters, etc.
  - Void constructor for String creates empty string, i.e., ""
## String Methods

- Return character at given index (offset from beginning)
  - `String s = "Go Ducks";`
  - `s.charAt(1)` is 'o'
  - `s.charAt(4)` is 'u'
  - `s.length()` is 8

- Form string by pasting two together
  - `s.concat("!!!")` is "Go Ducks!!!"
  - This is a new String, `s` is unchanged (like `s+"!!!"`)

- Extract a substring
  - `s.substring(3,8)` is "Ducks" (another new String)

- Produce capitalization (also can do lower case)
  - `s.toUpperCase()` is "GO DUCKS!!!" (another new String)

- Many more methods [String API page](#)
  - But none of them change the object

## Libraries and Packages

- Java has many useful classes coded in libraries that are part of the Java environment
  - Classes are organized in packages - groups of classes that go together
  - Library classes can be used more easily by importing them
  - E.g., be able to create Scanner objects for getting input:
    ```java
    import java.util.Scanner;
    ```
  - E.g., be able to create NumberFormat objects for formatting numbers:
    ```java
    import java.text.NumberFormat;
    ```
  - The full path name for a class could be used and no import statement would be needed
    - But this is more cumbersome:
      ```java
      java.util.Scanner scan = new java.util.Scanner(System.in);
      ```
  - May also use wildcards to get all classes in a package:
    ```java
    import java.util.*;
    ```
Date and Calendar classes

- System independent date/time operations
- Date objects are created for a specific date/time
  - Time measured in milliseconds since "epoch"
- `import java.util.*;`
- Default Date constructor uses current time
- Calendar objects permit breakdown in days, hours, etc.
  - Calendars set from Dates
  - Fields of calendars can be manipulated
  - New date can be extracted from Calendar

- Date API page  GregorianCalendar API page

BigInteger

- Built-in Java primitive types `int` or `long` have limits
  - Cannot be used for numbers larger than $2^{32}$ or $2^{64}$
  - Good enough for lots of things, but not everything
- BigInteger objects created from string representation of number
  - Constructor is given String argument
- BigInteger has methods for all arithmetic operations
  - Function syntax – not as nice as built in operators
- But no limit on size of values
  - Eventually machine could run out of resources
  - But calculations will be correct up to that point
- `import java.math.BigInteger;`
- Also BigDecimal for precision values
- BigInteger API page
BigInteger Example

- Contest problem:
  - Given a number, find the smallest multiple of that number which consists of just ‘1’ digits
  - E.g., 111 is a multiple of 3
  - The original number must not be divisible by 2 or 5 (or there won’t be a multiple consisting just of 1’s)

- Strategy:
  - Use BigIntegers
  - Start with 1, then 11, then 111, etc.

Designing Classes

- A **class** definition is a blueprint for creating objects
  - It describes the state and behavior that each object of the class must have
  - A class is really a **type** definition, much like int or double, but more complex
- A TunesCard class could specify that all music purchase card objects have:
  - A current balance, kept as a double
  - An account number, kept as an int
  - An owner's name, kept as a String
  - Methods addValue, debit, and getBalance
- Once we have a class definition, we can create objects of that class type
  - All such objects will have the same form, but their data values (state) will be specific to the object
  - Objects of the class type are called **instances**
  - Creating objects is called **instantiation**
Classes and Objects

Class name: TunesCard
Data Fields: owner is _____
cardNo is _____
balance is _____
Methods: addValue, debit, getBalance

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 Lee White
cardNo is 9999
balance is $35.12

Coding Classes in Java

- Java is an Object Oriented language
- A program in Java consists of class definitions
- A class definition contains
  - Data definitions
  - Method definitions
- One "main" class in the program contains a definition of the static main method
  - Execution of the program begins with main
  - Typically, main instantiates one or more objects and calls their methods (which may create more objects and call their methods . . .)
  - The Java standard library defines hundreds of classes that can be used
Java Class Syntax

```java
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;
    }
}
```

Notes about Class code

- A class definition should be in a file named for the class and should be `public`
  - E.g., class TunesCard is defined in TunesCard.java
  - May have other class definitions in the file, but only one can be public (the one the file is named after)
- Data definitions are like variable declarations and are typically `private`
  - Also called class variables or instance variables
- Method definitions may use the class variables
  - Can be used as if they were local variables
- More about public and private later
Object Types

- Java has a handful of built in primitive types
  - Numerical types, Boolean, single character
- Objects are more interesting types
  - A class defines a new object type
  - Objects have methods to perform operations, obtain data values, etc.
  - Objects can be very simple or very complicated
- Thousands of object types are defined in the Java library
  - Graphical objects: Buttons, Windows, ScrollBars, etc.
  - Computation objects: BigInteger, Date, etc.
- **String** is an object type, not a primitive
  - Method to get number of chars: `message.length()` is 12
  - Method to get a particular char: `message.charAt(1)` is 'e'
  - The String type allows a set of characters to be treated as one entity
  - String is a special object type in Java

Initialization

- How does an object of the class type come into existence?
  - Objects are created with the `new` operator
  - For example
    ```java
    TunesCard myCard = new TunesCard();
    ```
  - This creates a variable named `myCard` whose type is a TunesCard object – it is an instance of the TunesCard class
- What are the initial data values in an object?
  - Same as for local variables: zeroes
- Seems like we should be able to control the initial values
  - We can, with special methods called **constructors**
Coding Constructors

- Constructors are special methods defined in a class to perform initialization
  - A constructor is called automatically when object is created with `new`
  - The constructor is never called explicitly
  - The name of the constructor method is the name of the class
  - The constructor has no return type (not void, just no type at all)
- A constructor typically sets data values for the class variables
  - Values may be determined from parameters passed to the constructor
  - The parameter values are provided with the `new` operator
  - Overloading allows multiple constructors for different ways to initialize
  - The parameters to `new` must match a defined constructor
- If no constructors are defined, then Java defines a "void" constructor that takes no arguments and has no statements

Constructor Example

- A constructor for the TunesCard class

```java
public class TunesCard {
    ...
    // Constructor requires name on the card
    public TunesCard(String who) {
        owner = who;
        balance = 0;
    }
}
```

- Creating an account object with the constructor

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

- Recall that static methods for a class are called by using the **class name** and a dot
  - E.g., `Math.random()`
- Instance methods are called by using the **object variable name** and a dot
  - E.g., `herCard.addValue(100);`
  - Thus the code of the method deposit refers to the data of the object herCard
  - If an instance method of a class is called from another method of the same class, then the instance variable is not necessary – the same object is used
- Instance methods can only be called in the context of a particular object

Completing the example

- [TunesCard.java](#)
- [TunesCardTest.java](#)

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`
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 = new StringBuffer("Hello");
y.append("World");
```

- 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

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

- After

```
 after b = a;
```

```
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

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

- After

```
b = a;
```

```
a John / 1234 / 0.0
b Mary / 1235 / 0.0
```

```
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
Visibility Modifiers

- Class information is designed as **external** and **internal**.
- External information is the **interface**
  - The information we must know to use the object
  - Generally, the interface just consists of methods
  - These methods must be declared as **public** to be able to be used by code outside of the class definition
- Internal information is the **implementation**
  - We do not need to know implementation details to use the object (e.g., variable names, types, internal methods)
  - Data and methods which are internal are declared as **private**
  - Generally, all data should be private
- Goal is to keep public interface as small as possible

---

Visibility Modifiers

<table>
<thead>
<tr>
<th>public</th>
<th>private</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Variables</strong></td>
<td><strong>Methods</strong></td>
</tr>
<tr>
<td>Violate encapsulation</td>
<td>Enforce encapsulation</td>
</tr>
<tr>
<td>Support other methods in the class</td>
<td>Support other methods in the class</td>
</tr>
</tbody>
</table>

- Public methods/variables are accessible from any code in **any** class
  - The scope is everywhere (with the proper object/class qualifier)
- Private methods/variables are only accessible from the code **inside** the class itself
  - The scope is limited to the class code
Visibility Modifiers

- Proper use of public and private expresses design intent in code
  - The Java compiler can then enforce this design
  - Private data can not be accessed outside of the class code – it will produce a compiler error
- General rule of thumb: objects should take care of themselves
  - All data should be private, and changeable from the outside only by public methods
  - One exception is constants – since they are not changeable, it is safe to have public constants
- public and private support encapsulation and data hiding
  - In a good class design, the interface will be as large as it needs to be, but no larger
  - Encapsulation means the object is like a black box – as users, we only know the interface, not the inner workings

Scope and this

- 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`

```java
public TunesCard(String owner) {
    owner = owner;
    balance = 0;
}
```

**Oops!** Both uses of owner refer to the parameter, so class data does not get changed.

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

```java
public TunesCard(String owner) {
    this.owner = owner;
    balance = 0;
}
```

Now left side refers to class data, and right side to parameter.

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

TunesCard.java

Week 4  CIS 210  Winter 2010  33

Week 4  CIS 210  Winter 2010  34
Strings as Objects

- What’s wrong with this?
  ```java
  String msg;
  System.out.println(msg);
  System.out.println(msg.length());
  ```
  Null Pointer Exception
  since no String Object!

- A String object might look like
  ```java
  String msg = "hello";
  ```
  ![String diagram]

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

Comparing Strings

- What does the following code do?
  ```java
  String s, t; . . .
  if (s == t) . . .
  ```
  ```java
  Scenario 1  Scenario 2  Scenario 3
  s = "hello";
  t = s;
  s = "hello";
  t = "world";
  s = s.substring(0,4)+"o";
  ```
  ![String diagram]

- 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 OO fashion, Strings "know" when two Strings are equivalent
  - The method is given one argument - another String
  - `this` String compares itself to the other String
    ```java
    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

The toString Method

- Every object in Java has a method named `toString` with the signature:
  ```java
  public String toString()
  ```
- If an object does not define the method, the default behavior is to return a string with the object's location in memory
- A class can define the method to provide a meaningful string representation of the objects of that type
- This method will be used whenever an object is the argument for `println`
  - It may also be called explicitly
  - Also used when object is "added" to a String
Mutable vs. Immutable Objects

- Account objects have methods that change the state of the object
  - Deposit and withdraw methods change the balance field
- Do any methods of String change the String object?
  - `length()`? No  `concat()`? No  `toLowerCase()`? No  `equals()`? No  `substring()`? No
- None of the String methods change the state of the String object
  - String objects are immutable
  - Once created, the String object can never change
  - Other immutable classes: BigInteger
- Mutable objects may be changed by methods
  - Some methods are getters and don't change data
  - Other methods are setters or otherwise change
  - Mutable classes: Scanner, Date, StringBuffer

A Note on String Efficiency

- Java has special treatment for the String class
  - Use of symbolic '+' operator (and +=) for Strings
  - Shorthand for String object creation from literal without new
- Efficient treatment of constant literal Strings
  - Pool of constant strings
  - All uses of same literal share single instance
  - E.g., if "Go Ducks" appears in five places in the program, there will still be only one instance in memory
  - This is called an interned String
  - Possible since String objects are immutable

StringTest2.java