Lectures 15-16
Object-Oriented Design

1. **Immutable Objects & Class**
   - **String class is immutable; any object of type String is immutable**
   - **Immutable classes must adhere to the following requirements:**
     - **All data fields must be private**
     - **There must be no mutator methods in the class definition**
     - **There must be no accessor methods that returns a reference to a mutable data field**

Example of an accessor method that returns a reference to a mutable data field, i.e., the dateCreated field is not immutable because of the existence of the accessor method public java.util.Date getDateCreated( );

```
public class Student
{
   private int id;
   private String name;
   private java.util.Date dateCreated;

  public Student(int ssn, String newName)
  {
    id = ssn;
    name = newName;
    dateCreated = new java.util.Date( );
  }

  public int getId( )
  {
    return id;
  }

  public String getName( )
  {
    return name;
  }

  public java.util.Date getDateCreated( )
  {
    return dateCreated;
  }
}
```

```
public class Test
{
   public static void main(String [] args)
   {
      Student student = new Student(123, "John");
      java.util.Date dateCreated = student.getDateCreated();
      dateCreated.setTime(200000);

         /* dateCreated field has been changed,
            hence class Student is not immutable*/
   }
}
```

2. **Scope of Variables**
    - **<u>Local Variables</u> are declared & used inside of a method;**
    - **the <u>Scope of a Local Variable</u> extends from its declaration to the end of the method**
    - **the same variable name may be <u>declared many times in nonnested blocks</u> within a method definition**

    - **<u>Instance & Static Variables</u> are the variables, i.e., data fields, for the class; these class variables are declared inside the class definition**
    - **the <u>Scope of Class Variables</u> extend over the entire class regardless of where, in the class definition, they appear**
    - **one exception: if data field B is initialized by data field A, data field A must be declared prior to the declaration of data field B**
    - **a given variable name <u>can only be declared once as a class variable</u> in a given class definition**

    - **if a <u>Local Variable has same name as a Class Variable</u>, the Local Variable takes precedence & the Class Variable with the same name is hidden, i.e., it is currently unavailable**

```
class Foo
{
  int x = 0;
  int y = 0;

  Foo( ) { }

  void p( )
  {
    int x = 1;
    System.out.println("x = " + x);
    System.out.println("y = " + y);
  }
}
```

Foo   f = new Foo( ); // f is an instance of Foo

x is declared as a class variable & also as a local variable; inside the method p( ) the class variable is hidden & the local variable x is used

y is declared outside of the method p( ) but is is available throughout the class, hence it is available inside the method p()

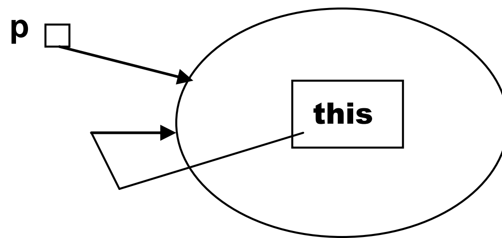
hence
f.p() produces x = 1 & y = 0

Good Practice Principle

Avoid using Class Names as Local Variable Names

3. The ***this*** Reference

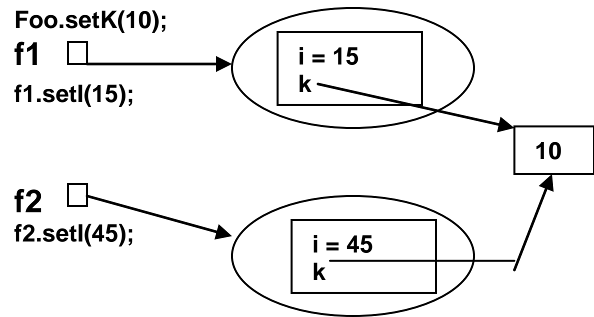
- ***this*** is the name of a reference that refers to the calling object itself




```
public class Foo
{
  int  i = 5;
  static double k = 0;

  void setI(int i)  this.i = i;

  static void setK(double k) Foo.k = k;
}
```



<u>this.i = i</u>
"assign the value of the parameter i to the data field of the calling object"
"this" refers to the object that invokes the instance method

<u>Foo.k = k</u>
"assign the value of the parameter k to the static data field k of the class which is shared by all instances of the class

```
public class Circle
{
  private double radius;

  public Circle(double radius) { this.radius = radius; }

  public Circle( ) { this(1.0); }

  public double getArea( )
  {
    return this.radius * this.radius * Math.PI;
  }
}
```

must be explicitly used to reference

used to invoke another constructor

not required; system infers that the reference **this** is the desired meaning

<u>Good Practice Principle</u>

1. A constructor with no or fewer arguments can invoke a constructor with more arguments by using this(…); this simplifies coding and improves readability

2. Java requires that all constructors appear before any other statements

4. Class Abstraction & Encapsulation

- **Class Abstraction**
  - **Separation of <u>Class Implementation</u> from the <u>Use of the Class</u>**

- **Class Contract – Provides the User with**
  - **Collection of Fields and Methods that are accessible from outside the Class, i.e., signatures of public methods & public constants**
  - **Description of how these members are expected to behave, i.e., the Creator of the Class describes it and lets the user know how it is to be used**

- **Class Encapsulation**
  - **Details of the Implementation are encapsulated and hidden from the user**
  - **Black Box**

Examples:

- **Math class   Math.random( );**
- **Loan Program   see Liang pages 308-310**
  <u>study this program in detail</u>

Hint: Write a program that uses the desired classes before the classes are implemented; this provides a set of methods and data items that need to be incorporated into the implementation. In summary,

- **<u>Developing a Class</u> & <u>Using a Class</u> are two separate tasks**
- **Attempting to use a class helps determine what features, data items and methods the class should provide**
- **It is easier to implement a class after you have determined how it is to be used in practice**

5. Object-Oriented MindSet

- **In real life, <u>objects are associated with both attributes and activities</u>, thus when dealing with programs which are designed to mirror real life,**
- **<u>Couple relevant Data & Methods together in Objects</u>, making the program a <u>Collection of Cooperating Objects</u>**

- **Such an approach provides for an easy way to produce <u>Reusable Software</u>**

6. Course Class

The UML public contract provides the basis to design programs using the Student Class; any implementation must conform to the expectations delineated in the published agreement and is immaterial to its usage

Course
-courseName: String -students: String [ ] -numberOfStudents: int
+Course(courseName: String) +getCourseName( ): String +addStudent(student: String): void +dropStudent(student: String): void +getStudents( ): String [ ] +getNumberOfStudents( ): int

Study the program listings 9.5 & 9.6 in detail; listing 9.5 shows a typical use of the Course class to construct multiple instances, i.e., objects, and manipulate their data. Listing 9.6 show one of multiple possible implementations of the Course class; exactly how the data is stored is not important to the program in listing 9.5, in fact, the internal data structures and the internals of the methods could be changed without having any effect on the program in listing 9.5

When you invoke Math.random( ) you are not concerned how the system produces the random number as long as it is sufficiently random for your stated purposes. If the internal algorithm is changed, it does not affect your programs as long as the public interface, i.e., the public contract, i.e., the UML specifications, are not changed.

7. Stack Class

```
public class StackOfIntegers
{
  private int [ ] elements;
  private int size;
  public static final int DEFAULT_CAPACITY = 16;
---------------------------------------------------
  public StackOfIntegers( )
  {
    this(DEFAULT_CAPACITY
  }
---------------------------------------------------
  public StackOfIntegers( int capacity)
  {
    Elements = new int[capacity];
  }
---------------------------------------------------
  public int push(int value)
  {
    if (size >= elements.length)
    {
    int [ ] temp = new int [elements.length * 2];
    System.arraycopy( elements, 0, temp, 0, elements.length);
    elements = temp;
    }
    return elements[size++] = value
  }
---------------------------------------------------
  public int pop( )
  {
    return elements[--size];
  }
---------------------------------------------------
  public int peek( )
  {
    return elements[size – 1];
  }
---------------------------------------------------
  public boolean empty( )
  {
    return size == 0;
  }
---------------------------------------------------
  public int getSize( )
  {
    return size;
  }
---------------------------------------------------
}
```

StackOfIntegers
-elements: int [ ] -size: int
+StackOfIntegers( ) +StackOfIntegers(capacity: int) +empty( ): boolean +peek( ): boolean +push(value: int): int +pop( ): int +getSize( ): int