OOP

• In this section we will learn what is Object Oriented Programming.

• The result of OOP is reduction in complexity, making design and implementation more manageable and more re-usable.

• OOP reduces complexity through:
  – De-composition
  – Abstraction
  – Hierarchy

• OOP allows design to be easily ported to implementation
Object Model

- Designed to reduce complexity in programs using
  - Abstraction: separate behavior from implementation; focus on essential details of some object, relative to viewer’s perspective.
  - Encapsulation: Provides contractual interface. Separates interface from implementation.
  - Modularity: Allow partitioning of tasks. A system which has been decomposed into a set of cohesive and loosely coupled modules.
Object Model (contd.)

- Hierarchy: Ranking of or ordering of abstractions (is a). May have single inheritance or multiple inheritance.

- Typing: Enforce identity to behavior. Objects of different types may not be interchanged or interchanged in a very restricted way.

- Concurrency: Multiple threads. Many concurrent tasks.

- Persistence: An object’s existence transcends time or space. Not standard in Java Objects
Objects and Classes

• An object is a:
  – Tangible or visible thing
  – Something apprehended intellectually
  – Something towards which thought or action is directed

• An object has a
  – **State**: static properties and the current value of these properties
  – **Behavior**: how an object acts and reacts to state changes and message passing
  – **Identity**: That which distinguishes from all other objects.

• The lifetime of an object is from the time it is created until that space it takes is reclaimed.

• Relationships between objects may be links or aggregation (has).
Objects and Classes (contd.)

• A *class* represents an abstraction of an object.
• The structure and behavior of similar and related objects are defined in a class.
• An object is not a class.
• An object is an instantiation of a class.
• On instantiation, an object obtains an identity and a state.
• An object always knows which class it belongs to.
Structured Programming vs OOP

- Structured Programming used Flow Charts.
- Traditionally concentrate on ‘how’.
- OOP concentrates on ‘what’.
- OOP is the natural way we treat the world, top down approaches.
- Hard to convert to OOP, once converted always converted.
Object Oriented Modeling

- ‘is a’ and ‘has’ relationships
- LINE, SHAPE and RECTANGLE are Classes
- myRect is an object of Class Rectangle and has specific attributes such as length and height
Java/C# is Object-Oriented

- A Java/C# program is a system of interacting **objects**
- Each object in a Java/C# program has a data type, or **class**
- Java/C# program design is class design:
  - generalizing objects into classes
  - defining relationships between classes
- Benefits of object-oriented programming include:
  - products that more accurately meet requirements
  - programs that are highly modular, and therefore easy to debug, enhance, modify
  - code that can be designed for reuse
main() and Classes

• Only one class in a whole package (directory) can have a
  public static void main(…) method (function).

• The class having the main() method is the one that should be invoked by the java/executable command.

• main() specifies the starting point of execution.
Classes, Objects and Variables

• A Java/C# class definition defines the interface and implementation of all objects of that class:
  - **data members** describe the internal state of each object
  - **function members** (methods):
    - describe the functional interface of the object
    - implement the behavior of the object
• To create an object of a class is to **instantiate** the class, i.e. create an **instance** of the class
• A Java/C# class-type variable **refers to** an instance
Instantiating a Class

• To instantiate a class and create an object, use the `new` operator

• For example, if we have a class named Box, we can create an instance of Box (object) as follows:

```java
Box     mybox = new Box(2.0);
```

```
↑          ↑          ↑          ↑
Class name | Object name | keyword | Constructor
           |            |         |
initialization parameter
```
Instantiating a Class (contd.)

• The *new* operator:
  – allocates space for the new object.
  – calls a class constructor.
  – returns a reference to the object.

• Note that the class and it’s constructor have the same name.

• The constructor with the initialization parameter gives details on what state the new object should have on instantiation.

• The object name merely points to the new object and is not the object itself.
Member Access

• Use the “.” operator to access members of an object

• Example: suppose that:
  – myCircle is a variable of class Circle
  – class Circle has a data member named radius of type Double

then the following statements are legal:

```java
myCircle.radius = new Double(42.6);
Double rad = new Double(myCircle.radius.doubleValue());
```
Invoking a Method

• A call to an object’s method is seen as a message to that object, requesting that it perform some action as if the object was a self contained entity

• A message has three components:
  – a target object
  – a method name
  – a parameter list
Invoking a Method (contd.)

- example: suppose the object myCircle contains the following methods:
  - `getArea`, which returns a double
  - `moveTo`, which takes two double arguments

Then we can have calls such as:

```java
Double area = new Double (myCircle.getArea());
myCircle.moveTo(0.0);
```
Class Definitions

• Example of a Java/C# class definition:

```java
class Box {
    // data members
    Double height, width;
    // Constructors
    public Box(Double h, Double w) {
        height = new Double(h.doubleValue());
        width = new Double(w.doubleValue());
    }
    public Box(Double s) {
        height = width = new Double(s.doubleValue());
    }
    public Double getHeight() {
        return height;
    }
    public Double getWidth() {
        return width;
    }
    public Double getArea() {
        return new Double(height.doubleValue() * width.doubleValue());
    }
}
```
Class Definitions (contd.)

• The setX() and the getX() are common methods to retrieve and modify internal variables of an object. These help to protect internal variables from incorrect use.
Class Definitions (Contd.)

• The header tells us that:
  ```java
  public class Box
  ```
  any other class can use this class
  this is a class type
  this class is named Box

• The class body enclosed within “{...}” defines the members of the class:
  – variables, or data members
  – methods, or function members
Class Definitions (Contd.)

• Members can be declared public, or private since they are not by default.

• A private member can only be accessed by another member of that class.

• A public member can be accessed directly from outside the object.

• The principle of *encapsulation* tells us that data members should be private and that at least some of the function members should be public.

• The public methods define a class’ interface.
Constructors

• A constructor is a special method whose job it is to initialize a newly instantiated object.

• Constructors are defined as follows:
  – method name identical to class name.
  – no return type, not even void.
  – arbitrary argument list.

• Constructors are normally placed as the first methods following the class name.

• When no constructor exists, a default one is created which does nothing and takes no argument.
Constructors (contd.)

```java
public class Box {
    Double height, width;
    public Box() {
        height = new Double(5.0);
        width = new Double(3.0);
    }
}

public class Display {
    Box box = new Box();
    ... }
```
Constructors (contd.)

- Overloading is the process where multiple functions may exist with the same name but different argument list.

- Constructors may be overloaded. For example, let us give our Box class two constructors:

```java
public class Box {
    Double height, width;
    public Box (Double hw) {
        height = width = hw;
    }
    public Box (Double h, Double w) {
        height = h;
        width = w;
    }
}
```
Constructors (contd.)

• A new operation always calls a constructor to fully initialize a new object.

• The choice of constructor to call depends on normal overload resolution rules (argument list matching) based on the initialization parameters.

• Example:

```java
Box b = new Box(1.0, 2.0);
/* calls: Box(Double h,
            Double w)
 */
```
Declaring instance Variables

- Variables defined inside a class are *instance variables* by default:
  - in the class definition for `Box`:
    ```java
    Double height, width;
    ```
    defines two instance variables.
  - every instance of type `Box` receives its own `height` and `width` variables.
Declaring instance Variables (contd.)

• Instance variables are always initialized, either explicitly or by default.

• Example of explicit initialization:
  
  `Double height = new Double(1.0);`

• Default initialization value for objects is null.
Declaring instance Variables (contd.)

• Instance variables are *already initialized* by the time the constructor executes:
  - the constructor need only perform additional initialization

• If you define no constructors for a class, the compiler provides a default one that:
  - takes no arguments
  - does nothing

example default constructor for class Box:

```java
Box ()
{
}
```
Declaring Instance Methods

- Methods defined inside a class are *instance methods* by default.
- An instance method must be applied to an instance of the class of which it is a member.
- For example, the class `Box` defines the methods:

  ```java
  public Double getHeight()
  {...}
  public Double getWidth()
  {...}
  public Double getArea()
  {...}
  
  These methods can be applied to any instance of the class `Box`.
Declaring Instance Methods (contd.)

- Each method is fully within the class definition.
- The body of the method is enclosed within “{…}”
- Methods can refer to instance variables of the target object directly:

```java
public Double getArea ()
{
    return new Double(
        height.doubleValue() *
        width.doubleValue());
}
```
The **this** Reserved Word

- Java/C# provides a reserved word called **this** which gives the object a name by which it can refer to itself.
- **this** is a reference to the object in question.
- The **this** reserved word is an implied object name for all members within the object itself.
- Useful if parameter names are identical to data member names and if the return value is the object itself.
- If an object has a data member **width**, then the following statements are equivalent when used from within the object’s function members:

```java
width = w;
this.width = w;
```
The `this` Reserved Word in Java (contd.)

- `this` can also be used within a constructor to invoke other constructors within the same object

```java
public Box(Double s) {
    // invoke 2 parameter constructor
    this(s, s);
}
```

- `this` is a Java/ class-type variable with reference semantics.

- Note constructor call (using `this`) can only be in first line of a constructor and never in a method.

- This construct is not available in C# yet no functionality is really lost.
Class Definitions and Source Files

• Each public class should be defined in its own source file.
• The name of each public class must match the name of its source file.
• For example:
  – the public class Box should be defined in a source file named Box.java/Box.cs
  – there should be no other public classes defined in Box.java
• These restrictions do not apply to non-public classes.
• A unique name must exist for each public class in the same package (directory).
Exercise

• Design a bicycle using the Object Oriented Model of design. Pinpoint the minimum amount of classes required and where each class has common elements with other classes. Identify clearly the ‘is a’ and the ‘has’ relationships.
Exercise (contd.)

• Construct three classes one for a Square, one for a Circle and one for a Rectangle with private data members and public setX and getX methods.

• Create another class called Tester which has a main method in it.

• Inside Tester create an instance of a Square, Circle and Rectangle Object.

• Access the setX and getX methods through Tester.

• Provide multiple overloaded constructors for Square, Circle and Rectangle and use them when instantiating the objects.