# L2,3: Java Overview I

## 1. OBJECT-ORIENTED, EVENT-DRIVEN PROGRAMMING

Think about a computer with lots of windows on the screen. Its response to your input depends on the window you are in. Each window is controlled by a program, usually an application. These all run separately, and have different properties, but are all tied together. This is the model we will program in this course. Many of our programs will be graphical by nature. Though most will have one or two windows, they will have multiple objects. These objects are essentially independent programs. You, the programmer, determine how they act and interact.

Java supports this programming model. That is, when programming in Java, we think in terms of *objects* and *events*: each object has its own data and manages its own state; objects interact with each other through methods. For this reason, Java is said to be an *object-oriented, event-driven* language: it models the world consisting of objects which respond to events.

For example, let's think of a world as consisting of colleges, students, courses, and professors. Here is how it could look like:

—**Colleges**
- —Some properties: location, name, list of courses, students, professors,
- —Things they do: give degrees
- Instances:
- —Bowdoin: inherits properties from colleges; specifies some information that was generic
- —Colby: also inherits information; the specifics are different

—**Students**
- —Some properties: name, major, transcript, age, ID of some type
- —Things they do: take courses, register, attend classes, do homework, pass tests
- Instances: ...

—**Professors**
- —Some properties: discipline,specialty, office hours,
- —Things they do:
  - —giveLecture(course, date)
  - —giveTest(course, exam#)
  - —grade(assignment, material)
- Instances: ...

—**Courses**
- —Some properties: pre-reqs, name, location, professors, students
- Instances: ...

You may wonder, why is object-oriented programming "nice"? Think of the world around you. There are objects everywhere. It is intuitive. You can use a TV without knowing what's inside — you just need to read its users manual :). You do the same in Java.

### Advantages of object-oriented

(1) Abstraction and encapsulation: Thinking in terms of objects and events allows programmers to separate the details that are important to the user of an object from the details that are not important to the user of the object. For e.g., you don't need to know how an array is implemented, you just need to know how to use it. As a programmer, just like a user, you'll separate the behaviour of a class from its implementation. You'll get used to using classes knowing only their *interface*, without knowing the specifics of how the class is implemented. This is refered to as *data abstraction and encapsulation.* It is not unique to Java, but rather a feature of good programming in general.
One consequence is that programming often means figuring out what has been done before and reading/understanding the interfaces of the classes that are available in Java libraries. You will spend a fair amount of time using other classes, abstracting away from their implementation details.

(2) modularity: objects are a natural way to split up a complex world.

(3) code re-use: use a class in several projects.

(4) debugging easier: find where the error is, and debug that class.

## 2. CLASSES AND OBJECTS

To create objects in Java you first have to define a class. A class is an encapsulation of data and methods: For example:

```
class Bicycle {

       private int speed;
       private int gear;

       public  Bicycle() {
           speed = 0;
           gear = 0;
       }
       public void changeGear(int newValue) {
            gear = newValue;
       }
       public void speedUp(int increment) {
            speed = speed + increment;
       }
       public void applyBrakes(int decrement) {
            speed = speed - decrement;
       }
       public void printStates() {
            System.out.println("speed:"+speed+" gear:"+gear);
        }
}
```

—data: public, private or protected

—methods:
  —accessors (getters)
  —mutators (setters)
  —both

—constructor(s)
  —method with same name as the class called to create a new instance of the class

A class is a recipe for creating objects. Objects are instances of classes. An object is created by calling the constructor of a class— this is called *instantianting* a class.

```
          // Create two different Bicycle objects
          Bicycle bike1 = new Bicycle();
          Bicycle bike2 = new Bicycle();

          // Invoke methods on those objects
          bike1.speedUp(10);
          bike1.changeGear(2);
          bike1.printStates();

          bike2.speedUp(10);
          bike2.changeGear(2);
          bike2.speedUp(10);
          bike2.changeGear(3);
          bike2.printStates();
```

## 3. ABSTRACTION

We'll look over some more examples. The goal is to practice abstraction; that is, we'll try to understand the meaning of a class, and use it, without actually knowing its implementation details.

### Example 1: BankAccount class

```
public class BankAccount {

    /**
     * @pre account is a string identifying the bank account
     * balance is the starting balance
     * @post constructs a bank account with desired balance
     */
    public BankAccount(String acc, double bal);
    /**
     * Compare two Bank Accounts (non-standard equals form)
     * @pre other is a valid bank account
     * @post returns true if this bank account is the same as other
     */
    public boolean equals(Object other);


    /**
     * Account accessor.
     * @post returns the bank account number of this account
     */
    public String getAccount();


    /**
     * Balance accessor.
     * @post returns the balance of this bank account
     */
    public double getBalance();


    /**
     * Deposit mutator.
     * @post deposit money in the bank account
     */
    public void deposit(double amount);


    /**
     * Withdrawal mutator.
     * @pre there are sufficient funds in the account
     * @post withdraw money from the bank account
     */
    public void withdraw(double amount);
```

The set of methods of the class, with their return type and the types of the parameters is called the *interface* of a class.

Suppose somebody gives us this class, but hides the implementation details. We'd like to use this class to answer the following question: Is it better to invest $100 over 10 years at 5%, or to invest $100 over 20 years at 2.5%? How would you do it?

```
BankAccount jon = new BankAccount("Jon",100.00);
BankAccount tom = new BankAccount("Tom",100.00);
```

```
for (int years = 0; years < 10; years++) {
  jon.deposit(jon.getBalance() * 0.05);
}

for (int years = 0; years < 20; years++) {
  tom.deposit(tom.getBalance() * 0.025);
}
System.out.println("Jon invests  $100 over 10 years at 5%.");
System.out.println("After 10 years " + jon.getAccount() +
" has $" + jon.getBalance());

System.out.println("Tom invests $100 over 20 years at 2.5%.");
System.out.println("After 20 years " + tom.getAccount() +
" has $" + tom.getBalance());
}
```

## Example 2: WordList class

Let's practice abstraction some more. Suppose we want to build a game of Hangman. The computer selects random words and we try to guess them. For this purpose we'll implement a class `WordList` that maintains and manipulates a list of words. To have an idea of how we'd design this class, let's think of how we'd use it in a game of Hangman —it could look something like this.

```
public void hangman() {

   WordList list;  //declaration
   String targetWord;

   list = new WordList(10); //construction
   list.add("disambiguate");
   list.add("input");
   list.add("bookkeeper");
   while (!list.isEmpty()) {
        targetWord = list.selectAny(); //select a random word
        //...play the game using target word
        list.remove(targetWord); //update the list
   }
}
```

Q: What can you infer about class `WordList`?

We can infer its interface:

```
public class WordList {

   //construct a word list capable to hold "size" words
   public WordList(int size);

   //add a word to the list
   public void add(String word);

   //return a random word from the list
   public String selectAny();

   //remove the word from the list
   public void remove(String word);

   //is the list empty?
   public boolean isEmpty();
}
```

## 4. JAVA REVIEW

```
//import java.io.*
//import java.util.*

public class  HelloWorld {

  //data
  //methods

  public static void main(String[] args) {
     System.out.println("Hello world!");
  }
}
```

—Java packages: `java.lang` (automatically imported; contains e.g. `System` and `String`). `java.util` contains for e.g. random nb generator.

—when the application is run, `main` is executed.

—if the name of your class is Xxx, then it should occur in a file called Xxx.Java

—comments: // or /* .... */

—public

### Classes

A class is really just a complex variable. A class (1) stores things (2) has operators. Major differences: (1) constructors; (2) create an object using `new`.

Typically the data stored is more than just a single variable.

The operators are (called) *methods.*

## Methods

Every method has an associated class; in other words, methods are defined only within classes (not standalone).

Withing the class, a method is invoked by calling its name.

Between classes you must use ".": `<classname>.methodname(<arguments>)`.

Methods are usually public (unless they are private :)).

## Basic types

—`boolean`: { true, false}

—`char`: any character

—`int`: $\{-2^{31}, 2^{31}-1\}$

—`long`: $\{-2^{63}, 2^{63}-1\}$

—`float`:$\{-10^{38}.......10^{38}\}$ or so

—`double`: even bigger

## Declaring variables

`<type> <variable-name>;`

—`<type>` is either a basic (primitive) type or a non-primitive type (class).

—Variable names should make sense.

—All class variables must be commented.

—Declaring CONSTANTS:

```
static final int MONDAY = 1;
```

—To refer to a character use single quotes: `char a = 'a';`. Double quotes are used for Strings: `String s = ''s''`.

—`int` and `long`: both hold integers, but longs can hold bigger integers.

—float and doubles both hold reals, but doubles are bigger.

## Operators (in order of precedence)

(1) `++`
(2) `--`
(3) `!`
(4) `*, /, %`
(5) `+, -`
(6) `<, <=, >=.  >`
(7) `==, !=`
(8) `&&, ||`
(9) `=`

Note: $a/b$ returns an integer (truncated value of a/b) if both a,b are integers.

There are also Math operators:

—`Math.sqrt(x)`

—`Math.abs(x)`

—`Math.pow(x,y)`
—`Math.sin(x)` and other trigonometric functions
—`Math.round(double x)`

## Type casting

```
double pi = 3.14;
int i;
i = (int) pi;
```

Sometimes Java will do casts automatically. Better style is to do the cast explicitly yourself:
Even better style is to avoid casts like this one altogether, if you can.
Casting is a powerful feature of Java and we'll come back to it later.

## Arrays

```
int arr1[];
arr1 = new int [10];

int arr2[] = {1, 2, 3, 4};
int arr3 = new int [100];
int arr4[] = arr3;
```

—Don't forget: arrays are 0-based.
—Discuss the implications of the last statement.
—You can ask for an array length: `arr1.length`. Note the lack of parentheses.
—Exercise: initializing an array; copying an array; printing an array.
—Passing an array as parameter:

```
public double largest(double [] elt) {

}

double arr1[];
...
blah.largest(arr1);
```

Note: you are passing the array, NOT a copy of it. If you modify the array inside the method...the changes stay.
—Exercise: a method that finds the largerst element in an array.

## Conditionals

```
if (condition) {
    <statements>;
};

if (condition) {
    <statements>;
} else {
    <statements>;
}

switch (expression) {
  case <constant>: <statement>; break;
  default: <statement>; break;
}
```

Example: Craps (see website for demo)

```
if ( newGame ) {                      // start a new game
     if ( roll == 7 || roll == 11 ) {  // 7 or 11 wins on first throw
           status.setText( "You win!" );
      }
      else if ( roll == 2 || roll == 3 || roll == 12 ) {  // 2, 3, or 12 loses
           status.setText( "You lose!" );
      }
      else {                      // Set roll to be the point to be made
           status.setText( "Try for your point!" );
           point = roll;
           newGame = false;         // no longer a new game
      }
}
else {     // continue trying to make the point
      if ( roll == 7 ) {          // 7 loses when trying for point
           status.setText( "You lose!" );
           newGame = true;         // set to start new game
       }
       else if ( roll == point ) { // making the point wins!
           status.setText( "You win!" );
           newGame = true;
       }
       else {                      // keep trying
           status.setText( "Keep trying for " + point + " ..." );
       }
}
```

—Note$_1$: You don't need to do:

```
if (boxGrabbed == true)
```

You can simply say

```
if (boxGrabbed)
```

—Note$_2$: Instead of

```
if (box.contains(point)) {
   boxGrabbed = true;
} else {
   boxGrabbed = false;
}
```

you can simply say

```
boxGrabbed = box.contains(point);
```

### Loops

```
for (<initialize>; <test>; <iteration>) {
    <statements>;
}

while (condition) {
    <statements>;
}
```

Example 1: what prints?

```
for (int i=0; i<3; i++)
   for (int j = 0; j<3; j++)
        System.out.println(i + " " + j);
```

Example 2: what prints?

```
for (int j=0; j<3; j++)
   for (int i = 0; i<3; i++, j++)
        System.out.println(j + " " + i);
```

Example 3: what prints?

```
int j=4;
for (int i=0; i<j; i++,j=j+4)
   for (; j>2; j=j/2)
        System.out.println(j + " " + i);
```

Example 4: what prints?

```
for (int i=0; i<10; i++,System.out.println(i))
     {}
```

### Class String

Defined in Java package `java.lang` which is included by default in any of your programs.

```
String s = "dog";
s = "hot " + s;
System.out.println("the length of " + s + " is " + s.length());

String t;
t = new String("this is an example");

String u;
//before u is initialized, u is null
//calling any method on a null string is illegal and will result in a run-time error
```

Operators (methods):

—`s.length()`
—`s.charAt(i)`
—`s.compareTo(String t)`
—`s.equals(String t)`
—`s.indexOf(String t)`
—`s.substring(start, end)`

You can also put `ints` and `doubles` into `String` expressions:

```
System.out.println("The x coordinate is " + x + "; the y coordinate is " + y);
```

## 4.1 Programming style

Naming conventions: classes start with Caps; variables start with small, constants ALL CAPS, methods start with small.

### Reading

—BlueJ tutorial
—Java tutorials
—Bailey Appendix B, chapter 1, 2
if you need more background: Eventful chapter 1-8