Basic Object-Oriented Programming in Java

Topics in This Section

- Similarities and differences between Java and C++
- Object-oriented nomenclature and conventions
- Instance variables (data members, fields)
- Methods (member functions)
- Constructors
- Person class with four variations

"Object-oriented programming is an exceptionally bad idea which could only have originated in California."   -- Edsger Dijkstra, 1972 Turing Award winner.

Tutorial Progression

- **Progression of topics**
  - This lecture
    - Instance variables
    - Methods
    - Constructors
  - Next lecture
    - Overloading
    - Private instance variables and accessor methods
      - From this point onward, examples are consistent with real-world coding guidelines
    - JavaDoc documentation
    - Inheritance

Object-Oriented Terminology

- **“Class” means a category of things**
  - A class name can be used in Java as the type of a field or local variable or as the return type of a function (method)
    - There are also fancy uses with generic types such as List<String>. This is covered later.
- **“Object” means a particular item that belongs to a class**
  - Also called an “instance”
- **Example**

```
String s1 = "Hello";
```

  - Here, String is the class, and the variable s1 and the value "Hello" are objects (or “instances of the String class”)

Comparisons to Similar Languages

- **C++**
  - Similar on the surface
    - User-defined classes can be used like built-in types.
    - Basic syntax
  - Very different under the hood
    - See next slide
- **C#**
  - Very similar throughout. Different libraries, but core languages are very close
  - Details:
    - http://www.harding.edu/fmccown/java_csharp_comparison.html
    - http://en.wikipedia.org/wiki/Comparison_of_C_Sharp_and_Java

Comparisons to Similar Languages

- **Differences from C++**
  - Methods (member functions) are the only function type
  - Object is the topmost ancestor for all classes
  - All methods use the run-time, not compile-time, types (i.e. all Java methods are like C++ virtual functions)
  - The types of all objects are known at run-time
  - All objects are allocated on the heap (so, always safe to return objects from methods).
    - No difference between “s is a String” and “s is pointer to String”
  - Single inheritance only
    - Java 8 has multiple inheritance (as we will see), but via interfaces instead of by normal classes, so is a bit of a nonstandard variation of multiple inheritance

Instance Variables

- **Definition**
  - Data that is stored inside an object. "Instance variables" can also be called "data members" or "fields".
- **Syntax**

```
public class MyClass {
  public SomeType field1, field2;
}
```

- **Note**
  - In any class that also has methods, it is almost always better to declare instance variables private instead of public. But, we need more tools before we can do this.
    - We will show how and why in the next tutorial section.

Motivation

- **Persistence**
  - Instance variables let an object have values that persist over time

```
Person p = new Person();
p.firstName = "Jane";
doSomethingElse();
checkValueOf(p.firstName); // Still "Jane"
```

- **Object-oriented programming features**
  - It is often said that in OOP, objects have three characteristics:
    - State
    - Behavior
    - Identity
  - The instance variables provide the state

Ship Example 1: Instance Variables (ship1/Ship.java)

```
package ship1;

public class Ship {
  public double x, y, speed, direction;
  public String name;
}
```

Ship Tester (ship1/ShipTest.java)

```
package ship1;

public class ShipTest {
  public static void main(String[] args) {
    Ship s1 = new Ship();
    s1.x = 0.0;
    s1.y = 0.0;
    s1.speed = 1.0;
    s1.direction = 0.0;   // East
    s1.name = "Ship1";
    Ship s2 = new Ship();
    s2.x = 0.0;
    s2.y = 0.0;
    s2.speed = 2.0;
    s2.direction = 135.0; // Northwest
    s2.name = "Ship2";
```

Ship Tester (Continued)

```
    ...
    s1.x = s1.x + s1.speed
           * Math.cos(s1.direction * Math.PI / 180.0);
    s1.y = s1.y + s1.speed
           * Math.sin(s1.direction * Math.PI / 180.0);
    s2.x = s2.x + s2.speed
           * Math.cos(s2.direction * Math.PI / 180.0);
    s2.y = s2.y + s2.speed
           * Math.sin(s2.direction * Math.PI / 180.0);
    System.out.println(s1.name + " is at ("
                       + s1.x + "," + s1.y + ").");
    System.out.println(s2.name + " is at ("
                       + s2.x + "," + s2.y + ").");
  }
}
```

Move the ships one step based on their direction and speed.

The previous slide seemed good: grouping variables together. But the code on this slide violates the primary goal of OOP: to avoid repeating identical or nearly-identical code. So, although instance variables are good, they are not enough: we need methods also.

Instance Variables: Results

- **Compiling and running in Eclipse (common)**
  - Save Ship.java and ShipTest.java
  - R-click inside ShipTest.java, Run As → Java Application

- **Compiling and running manually (rare)**

```
> javac ship1\ShipTest.java
> java ship1.ShipTest
```

- **Output:**

```
Ship1 is at (1,0).
Ship2 is at (-1.41421,1.41421).
```

Example 1: Major Points

- Java naming conventions
- Format of class definitions
- Creating classes with “new”
- Accessing fields with “variableName.fieldName”

Java Naming Conventions

- **Start classes with uppercase letters**
  - Constructors (discussed later in this section) must exactly match class name, so they also start with uppercase letters

```
public class MyClass {
    ...
}
```

Java Naming Conventions

- **Start other things with lowercase letters**
  - Instance variables, local variables, methods, parameters to methods

```
public class MyClass {
  public String firstName, lastName;

  public String fullName() {
    String name = firstName + " " + lastName;
    return(name);
  }
}
```

Objects and References

- **Once a class is defined, you can declare variables (object reference) of that type**

```
Ship s1, s2;
Point start;
Color blue;
```

- **Object references are initially `null`**
  - The `null` value is a distinct type in Java and is not equal to zero
  - A primitive data type (e.g., int) cannot be cast to an object (e.g., String), but there are some conversion wrappers

- **The `new` operator is required to explicitly create the object that is referenced**

```
ClassName variableName = new ClassName();
```

Accessing Instance Variables

- **Use a dot between the variable name and the field**

```
variableName.fieldName
```

- **Example**
  - For example, Java has a built-in class called `Point` that has `x` and `y` fields

```
Point p = new Point(2, 3); // Build a Point object
int xSquared = p.x * p.x;  // xSquared is 4
int xPlusY = p.x + p.y;    // xPlusY is 5
p.x = 7;
xSquared = p.x * p.x;      // Now xSquared is 49
```

- **Exceptions**
  - Can access fields of current object without varName
    - See upcoming method examples
  - It is conventional to make all instance variables private
    - In which case outside code can't access them directly. We will show later how to hook them to outside with methods.

Methods

- **Definition**
  – Functions that are defined inside a class. "Methods" can also be called "member functions".
- **Syntax**

```
public class MyClass {
  public ReturnType myMethod(...) { ... }
}
```

- **Note**
  – This example uses public methods because we have not yet explained about private. Once you learn about private, your strategy is this:
    - If you want code that uses your class to access the method, make it public.
    - If your method is called only by other methods in the same class, make it private.
    - Make it private unless you have a specific reason to do otherwise.

Motivation

- **Behavior**
  - Methods let an object calculate values or do operations, usually based on its current state (instance variables).

```
public class Person {
  public String firstName, lastName;
  ...
  public String getFullName() {
    return(firstName + " " + lastName);
  }
}
```

- **Object-oriented programming features**
  - It is often said that objects have three characteristics: state, behavior, and identity
  - The methods provide the behavior

Ship Example 2: Methods (ship2/Ship.java)

In next lecture, we will show that the instance variables (x, y, etc.) should be private. But we need to first explain how to hook them to the outside world if private. So, just keep in the back of your mind the fact that we are making the fields public for now, but would not do so in real life.

```
package ship2;

public class Ship {
  public double x=0.0, y=0.0, speed=1.0, direction=0.0;
  public String name = "UnnamedShip";

  private double degreesToRadians(double degrees) {
    return(degrees * Math.PI / 180.0);
  }

  public void move() {
    double angle = degreesToRadians(direction);
    x = x + speed * Math.cos(angle);
    y = y + speed * Math.sin(angle);
  }

  public void printLocation() {
    System.out.println(name + " is at (" + x + "," + y + ").");
  }
}
```

Ship Tester (ship2/ShipTest.java)

```
package ship2;

public class ShipTest {
  public static void main(String[] args) {
    Ship s1 = new Ship();
    s1.name = "Ship1";
    Ship s2 = new Ship();
    s2.direction = 135.0; // Northwest
    s2.speed = 2.0;
    s2.name = "Ship2";
    s1.move();
    s2.move();
    s1.printLocation();
    s2.printLocation();
  }
}
```

Methods: Results

- **Compiling and running in Eclipse (common)**
  - Save Ship.java and ShipTest.java
  - R-click inside ShipTest.java, Run As → Java Application

- **Compiling and running manually (rare)**

```
> javac ship2\ShipTest.java
> java ship2.ShipTest
```

- **Output:**

```
Ship1 is at (1,0).
Ship2 is at (-1.41421,1.41421).
```

Example 2: Major Points

- **Format of method definitions**
- **Methods that access local fields**
- **Calling methods**
- **Static methods**
- **Default values for fields**
- **public/private distinction**

Defining Methods (Functions Inside Classes)

- **Basic method declaration:**

```
public ReturnType methodName(Type1 arg1, Type2 arg2, ...) {
   ...
   return(somethingOfReturnType);
}
```

- **Exception to this format: if you declare the return type as `void`**
  – This special syntax that means "this method isn't going to return a value – it is just going to do some side effect like printing on the screen"
  – In such a case you do not need (in fact, are not permitted), a **`return`** statement that includes a value to be returned

Examples of Defining Methods

```
// Example function call:
//   int val = square(7);

public int square(int x) {
  return(x*x);
}

// Example function call:
//   Ship faster = fasterShip(someShip, someOtherShip);

public Ship fasterShip(Ship ship1, Ship ship2) {
  if (ship1.speed > ship2.speed) {
    return(ship1);
  } else {
    return(ship2);
  }
}
```

Calling Methods

- **Terminology**
  - "Method" means "function associated with an object" (I.e., "member function")

- **Calling methods**

```
variableName.methodName(argumentsToMethod);
```

- **Example**
  - The toUpperCase method doesn't take any arguments, so you just put empty parentheses after the function (method) name.

```
String s1 = "Hello";
String s2 = s1.toUpperCase(); // s2 is now "HELLO"
```

Accessing External and Internal Methods

- **Accessing methods in other classes**
  - Get an object that refers to instance of other class

    ```
    Ship s = new Ship();
    ```
  - Call method on that object

    ```
    s.move();
    ```
- **Accessing instance vars in same class**
  - Call method directly (no variable name and dot in front)

    ```
    move();
    double d = degreesToRadians();
    ```
    - For local methods, you can use a variable name if you want, and Java automatically defines one called “this” for that purpose. See constructors section.
- **Accessing static methods**
  - Use ClassName.methodName(args)

    ```
    double d = Math.cos(Math.PI/2);
    ```

Calling Methods (Continued)

- **Calling a method of the current class**
  - You don't need the variable name and the dot
  - For example, a **`Ship`** class might define a method called **`degreeesToRadians`**, then, within another function in the same class definition, do this:

```
double angle = degreesToRadians(direction);
```

    - No variable name and dot is required in front of **`degreesToRadians`** since it is defined in the same class as the method that is calling it

- **Calling static methods**
  - Use ClassName.methodName(args)

```
double randomNumber = Math.random();
```

Method Visibility

- **public/private distinction**
  - A declaration of private means that "outside" methods can't call it – only methods within the same class can
    - Thus, for example, the **`main`** method of the **`Test2`** class could not have done

      ```
      double x = s1.degreesToRadians(2.2);
      ```

      - Attempting to do so would have resulted in an error at compile time
  - Only say public for methods that you *want to guarantee your class will make available to users*
  - You are free to change or eliminate private methods without telling users of your class
- **private instance variables**
  - In next lecture, we will see that you *always* make instance vars private and use methods to access them

Static Methods

- **Also called "class methods" (vs. "instance methods")**
  - Static functions do not access any non-static methods or fields within their class and are almost like global functions in other languages
- **Call a static method through the class name**
  - ClassName.functionName(arguments);
- **Example: Math.cos**
  - The Math class has a static method called cos that expects a double precision number as an argument. So, you can call Math.cos(3.5) without ever having any object (instance) of the Math class

```
double cosine = Math.cos(someAngle);
```

- **Note on the main method**
  - Since the system calls main without first creating an object, static methods are the only type of methods that main can call *directly* (i.e. without building an object and calling the method of that object)

Constructors

- **Definition**
  – Code that gets executed when “new” is called
- **Syntax**
  – “Method” that exactly matches the class name and has no return type (not even void).

```
public class MyClass {
  public MyClass(...) { ... }
}
```

Motivation

- **Shorter code**
  - Lets you build an instance of the class, and assign values to instance variables, all in one line
    - Vs. one line to build instance, then several additional lines to assign instance variables
- **Consistency**
  - Lets you enforce that all instances have certain properties
    - For example, a Ship might not be legal without a name, but with instance variables, there is no way to force the programmer to assign a name
- **Side effects**
  - Constructors let you run extra code when class is instantiated. You can draw the Ship on the GUI, add the Ship to the fleet, keep a count of all Ships, etc.

Example: No User-Defined Constructor

- **Person**

```
public class Person1 {
  public String firstName, lastName;
}
```

- **PersonTest**

```
public class Person1Test {
  public static void main(String[] args) {
    Person1 p = new Person1();
    p.firstName = "Larry";
    p.lastName = "Ellison";
    // doSomethingWith(p);
  }
}
```

It took <u>three</u> lines of code to make a properly constructed person. It <u>would</u> be possible for a programmer to build a person and forget to assign a first or last name.

Example: User-Defined Constructor

- **Person**

```
public class Person2 {
  public String firstName, lastName;

  public Person2(String initialFirstName, String initialLastName) {
    firstName = initialFirstName;
    lastName = initialLastName;
  }
}
```

Constructor. This one takes two strings as arguments.

- **PersonTest**

```
public class Person2Test {
  public static void main(String[] args) {
    Person2 p = new Person2("Larry", "Page");
    // doSomethingWith(p);
  }
}
```

It took one line of code to make a properly constructed person. It would not be possible for a programmer to build a person and forget to assign a first or last name.

Ship Example 3: Constructors (ship3/Ship.java)

```
public class Ship {
  public double x, y, speed, direction;
  public String name;

  public Ship(double x, double y,
              double speed, double direction,
              String name) {
    this.x = x; // "this" differentiates instance
    this.y = y; //  vars from local vars.
    this.speed = speed;
    this.direction = direction;
    this.name = name;
  }

  ... // Same methods as last example
}
```

Ship Tester (ship3/ShipTest.java)

```
package ship3;

public class ShipTest {
  public static void main(String[] args) {
    Ship s1 = new Ship(0.0, 0.0, 1.0,   0.0, "Ship1");
    Ship s2 = new Ship(0.0, 0.0, 2.0, 135.0, "Ship2");
    s1.move();
    s2.move();
    s1.printLocation();
    s2.printLocation();
  }
}
```

Constructors: Results

- **Compiling and running in Eclipse (common)**
  - Save Ship.java and ShipTest.java
  - R-click inside ShipTest.java, Run As → Java Application

- **Compiling and running manually (rare)**

```
> javac ship3\ShipTest.java
> java ship3.ShipTest
```

- **Output:**

```
Ship1 is at (1,0).
Ship2 is at (-1.41421,1.41421).
```

Example 3: Major Points

- **Format of constructor definitions**
- **The “this” reference**
- **Destructors (not!)**

Format of Constructors

- **Syntax**

```
public class MyClass {
  public MyClass(…) { … }
}
```

- **When used**

```
MyClass m = new MyClass();
```

The “this” Variable

- **The this variable**
  - The this object reference can be used inside any non-static method to refer to the current object
- **The common uses of the “this” reference are:**
  - To pass pointer to the current object to another method

```
someMethod(this);
```

  - To resolve name conflicts

```
public class Blah {
  private int x;
  public Blah(int x) { this.x = x; }
}
```

    - It is only necessary to say this.fieldName when you have a local variable and a field with the same name; otherwise just use fieldName with no “this”

Destructors

This Page Intentionally Left Blank

Example: Person Class

- **Goal**
  - Make a class to represent a person’s first and last name
- **Approach: 4 iterations**
  - Person with instance variables only
    - And test case
  - Add a getFullName method
    - And test case
  - Add a constructor
    - And test case
  - Change constructor to use “this” variable
    - And test case
    - Also have test case make a Person[]

Iteration 1: Instance Variables

Person.java

```
public class Person {
  public String firstName, lastName;
}
```

PersonTest.java

```
public class PersonTest {
  public static void main(String[] args) {
    Person p = new Person();
    p.firstName = "Larry";
    p.lastName = "Ellison";
    System.out.println("Person's first name: " +
                                  p.firstName);
    System.out.println("Person's last name: " +
                                  p.lastName);
  }
}
```

Iteration 2: Methods

Person.java

```
public class Person {
  public String firstName, lastName;

  public String getFullName() {
    return(firstName + " " + lastName);
  }
}
```

PersonTest.java

```
public class PersonTest {
  public static void main(String[] args) {
    Person p = new Person();
    p.firstName = "Bill";
    p.lastName = "Gates";
    System.out.println("Person's full name: " +
                                   p.getFullName());
  }
}
```

Iteration 3: Constructors

Person.java

```
public class Person {
  public String firstName, lastName;

  public Person(String initialFirstName,
                         String initialLastName) {
    firstName = initialFirstName;
    lastName = initialLastName;
  }

  public String getFullName() {
    return(firstName + " " + lastName);
  }
}
```

PersonTest.java

```
public class PersonTest {
  public static void main(String[] args) {
    Person p = new Person("Larry", "Page");
    System.out.println("Person's full name: " +
                                   p.getFullName());
  }
}
```

Iteration 4: Constructors with the “this” Variable (and Arrays)

Person.java

```
public class Person {
  public String firstName, lastName;

  public Person(String firstName,
                String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  public String getFullName() {
    return(firstName + " " + lastName);
  }
}
```

PersonTest.java

```
public class PersonTest {
  public static void main(String[] args) {
    Person[] people = new Person[20];
    for(int i=0; i<people.length; i++) {
      people[i] =
        new Person(NameUtils.randomFirstName(),
                   NameUtils.randomLastName());
    }
    for(Person person: people) {
      System.out.println("Person's full name: " +
                         person.getFullName());
    }
  }
}
```

Helper Class for Iteration 4

```
public class NameUtils {
  public static String randomFirstName() {
    int num = (int)(Math.random()*1000);
    return("John" + num);
  }

  public static String randomLastName() {
    int num = (int)(Math.random()*1000);
    return("Smith" + num);
  }
}
```

To Do: Later Iterations

- **Use accessor methods**
  - Make instance variables private, then use getFirstName, setFirstName, getLastName, and setLastName
- **Document code with JavaDoc**
  - Add JavaDoc-style comments so that the online API for Person class will be useful
- **Use inheritance**
  - Make a class (Employee) based on the Person class. Don’t repeat the code from the Person class.

- **Next lecture**
  - Covers all of these ideas, then shows updated code

Summary

- **Conventions**
  - Class names start with upper case. Names for methods, variables, and packages start with lower case
  - Indent nested blocks consistently
- **Example class**

```
public class Circle {
  public double radius; // We'll make this private next lecture
  public Circle(double radius) { this.radius = radius; }
  public double getArea() { return(Math.PI*radius*radius); }
}
```

- **Example usage**

```
Circle c1 = new Circle(10.0);
double area = c1.getArea();
```