core

WEB

programming

Basic Object-Oriented Programming in Java

Agenda

- Similarities and differences between Java and C++
- Object-oriented nomenclature and conventions
- Instance variables (fields)
- Methods (member functions)
- Constructors

Object-Oriented Programming in Java

- **Similarities with C++**
  - User-defined classes can be used the same way as built-in types.
  - Basic syntax
- **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 (always safe to return objects from methods)
  - Single inheritance only

Object-Oriented Nomenclature

- **“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)
- **“Object” means a particular item that belongs to a class**
  - Also called an “instance”
- **For example, consider the following line:**

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

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

Example 1: Instance Variables (“Fields” or “Data Members”)

```
class Ship1 {
  public double x, y, speed, direction;
  public String name;
}

public class Test1 {
  public static void main(String[] args) {
    Ship1 s1 = new Ship1();
    s1.x = 0.0;
    s1.y = 0.0;
    s1.speed = 1.0;
    s1.direction = 0.0;   // East
    s1.name = "Ship1";
    Ship1 s2 = new Ship1();
    s2.x = 0.0;
    s2.y = 0.0;
    s2.speed = 2.0;
    s2.direction = 135.0; // Northwest
    s2.name = "Ship2";
    ...
```

Instance Variables: Example (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 + ").");
  }
}
```

Instance Variables: Results

- **Compiling and Running:**

```
javac Test1.java
java Test1
```

Output:

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

Example 1: Major Points

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

Java Naming Conventions

- Leading uppercase letter in class name

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

- Leading lowercase letter in field, local variable, and method (function) names
  - `myField`, `myVar`, `myMethod`

First Look at Java Classes

- The general form of a simple class is

```
modifier class Classname {

  modifier data-type field1;
  modifier data-type field2;
  ...
  modifier data-type fieldN;

  modifier Return-Type methodName1(parameters) {
    //statements
  }

  ...

  modifier Return-Type methodName2(parameters) {
    //statements
  }
}
```

Objects and References

- **Once a class is defined, you can easily declare a variable (object reference) of the class**

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

- **Object references are initially `null`**
  - The `null` value is a distinct type in Java and should not be considered equal to zero
  - A primitive data type cannot be cast to an object (use wrapper classes)
- **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 name, as follows:**

```
variableName.fieldName
```

- **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
```

- **One major exception applies to the “access fields through varName.fieldName” rule**
  - Methods can access fields of current object without varName
  - This will be explained when methods (functions) are discussed

Example 2: Methods

```
class Ship2 {
  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 + ").");
  }
}
```

Methods (Continued)

```
public class Test2 {
  public static void main(String[] args) {
    Ship2 s1 = new Ship2();
    s1.name = "Ship1";
    Ship2 s2 = new Ship2();
    s2.direction = 135.0; // Northwest
    s2.speed = 2.0;
    s2.name = "Ship2";
    s1.move();
    s2.move();
    s1.printLocation();
    s2.printLocation();
  }
}
```

- **Compiling and Running:**

```
javac Test2.java
java Test2
```

- **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(something of ReturnType);
}
```

- **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

- **Here are two examples:**
  - The first squares an integer
  - The second returns the faster of two **`Ship`** objects, assuming that a class called **`Ship`** has been defined that has a field named **`speed`**

```
// 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);
  }
}
```

Exception to the "Field Access with Dots" Rule

- **You normally access a field through**

  ```
  variableName.fieldName
  ```

  but an exception is when a method of a class wants to access fields of that same class
  - In that case, omit the variable name and the dot
  - For example, a move method within the Ship class might do:

    ```
    public void move() {
      x = x + speed * Math.cos(direction);
      ...
    }
    ```

    - Here, **`x`**, **`speed`**, and **`direction`** are all fields within the class that the **`move`** method belongs to, so **`move`** can refer to the fields directly
  - As we'll see later, you still can use the **`variableName.fieldName`** approach, and Java invents a variable called **`this`** that can be used for that purpose

Calling Methods

- **The term “method” means “function associated with an object” (I.e., “member function”)**
  - The usual way that you call a method is by doing the following:

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

- **For example, the built-in `String` class has a method called `toUpperCase` that returns an uppercase variation of a `String`**
  - This 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"
```

Calling Methods (Continued)

- **There are two exceptions to requiring a variable name for a method call**
  - Calling a method defined inside the current class definition
  - Functions (methods) that are declared "`static`"
- **Calling a method that is defined inside 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

Static Methods

- **Static functions typically do not need to access any fields within their class and are almost like global functions in other languages**
- **You can call a static method through the class name**

```
ClassName.functionName(arguments);
```

  - For example, 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

- **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)

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 about

Declaring Variables in Methods

- When you declare a local variable inside of a method, the normal declaration syntax looks like:

```
Type varName = value;
```

- The value part can be:
  - A constant,
  - Another variable,
  - A function (method) call,
  - A “constructor” invocation (a special type of function prefaced by **`new`** that builds an object),
  - Some special syntax that builds an object without explicitly calling a constructor (e.g., strings)

Declaring Variables in Methods: Examples

```
int x = 3;
int y = x;

// Special syntax for building a String object
String s1 = "Hello";

// Building an object the normal way
String s2 = new String("Goodbye");

String s3 = s2;
String s4 = s3.toUpperCase(); // Result: s4 is "GOODBYE"

// Assume you defined a findFastestShip method that
// returns a Ship
Ship ship1 = new Ship();
Ship ship2 = ship1;
Ship ship3 = findFastestShip();
```

Example 3: Constructors

```
class Ship3 {
  public double x, y, speed, direction;
  public String name;

  public Ship3(double x, double y,
               double speed, double direction,
               String name) {
    this.x = x; // "this" differentiates instance vars
    this.y = y; //  from local vars.
    this.speed = speed;
    this.direction = direction;
    this.name = name;
  }

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

Constructors (Continued)

```
  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 + ").");
  }
}

public class Test3 {
  public static void main(String[] args) {
    Ship3 s1 = new Ship3(0.0, 0.0, 1.0,   0.0, "Ship1");
    Ship3 s2 = new Ship3(0.0, 0.0, 2.0, 135.0, "Ship2");
    s1.move();
    s2.move();
    s1.printLocation();
    s2.printLocation();
  }
}
```

Constructor Example: Results

- **Compiling and Running:**

```
javac Test3.java
java Test3
```

- **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!)

Constructors

- **Constructors are special functions called when a class is created with `new`**
  - Constructors are especially useful for supplying values of fields
  - Constructors are declared through:

```
public ClassName(args) {
  ...
}
```

  - Notice that the constructor name must exactly match the class name
  - Constructors have no return type (not even `void`), unlike a regular method
  - Java automatically provides a zero-argument constructor if and only if the class doesn't define it's own constructor
    - That's why you could say

```
Ship1 s1 = new Ship1();
```

      in the first example, even though a constructor was never defined

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:**

1. To pass a reference to the current object as a parameter to other methods

```
someMethod(this);
```

2. To resolve name conflicts
    - Using **`this`** permits the use of instance variables in methods that have local variables with the same name

– Note that it is only necessary to say **`this.fieldName`** when you have a local variable and a class field with the same name; otherwise just use **`fieldName`** with no **`this`**

Destructors

This Page Intentionally Left Blank

Summary

- **Class names should start with upper case; method names with lower case**
- **Methods must define a return type or `void` if no result is returned**
- **Access fields via objectName.fieldName**
- **Access methods via objectName.methodName(args)**
- **If a method accepts no arguments, the arg-list in the method declaration is empty instead of `void` as in C**
- **Static methods do not require an instance of the class; they can be accessed through the class name**
- **The `this` reference refers to the *current* object**
- **Class constructors do not declare a return type**
- **Java performs its own memory management and requires no destructors**

core WEB programming

Questions?