Building Java Programs

Lecture 1: Java Review

reading: Ch. 1-9

Copyright 2008 by Pearson Education

A Java program (1.2)

```
public class name {
    public static void main(String[] args) {
        statement;
        statement;
        ...
        statement;
    }
}
```

- Every executable Java program consists of a **class**, that contains a **method** named `main`, that contains the **statements** (commands) to be executed.

Static methods (1.4)

- **static method**: A named group of statements.
  - denotes the **structure** of a program
  - eliminates **redundancy** by code reuse

- **procedural decomposition**: dividing a problem into methods

- Writing a static method is like adding a new command to Java.

Control flow

- When a method is called, the program's execution...
  - "jumps" into that method, executing its statements, then
  - "jumps" back to the point where the method was called.

```
public class MethodsExample {
    public static void message1() {
        System.out.println("This is message1.");
    }
    public static void message2() {
        System.out.println("This is message2.");
        message1();
        System.out.println("Done with message2.");
    }
    public static void message3() {
        System.out.println("This is message3.");
    }
    public static void message4() {
        System.out.println("This is message4.");
        message2();
        System.out.println("Done with message4.");
    }
    public static void main(String[] args) {
        message1();
        message2();
        message3();
        message4();
    }
}
```
Java's primitive types (2.1)

- **primitive types**: 8 simple types for numbers, text, etc.
  - Java also has **object types**, which we'll talk about later

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>integers</td>
<td>42, -3, 0, 926394</td>
</tr>
<tr>
<td>double</td>
<td>real numbers</td>
<td>3.1, -0.25, 9.4e3</td>
</tr>
</tbody>
</table>
| char   | single text characters | 'a', 'X', '?' , '
' |
| boolean| logical values    | true, false            |

- Why does Java distinguish integers vs. real numbers?

Value semantics (primitives)

- **value semantics**: Behavior where values are copied when assigned to each other or passed as parameters.
  - When one primitive variable is assigned to another, its value is copied.
  - Modifying the value of one variable does not affect others.

```
int x = 5;
int y = x;  // x = 5, y = 5
y = 17;    // x = 5, y = 17
x = 8;     // x = 8, y = 17
```

Reference semantics (objects)

- **reference semantics**: Behavior where variables actually store the address of an object in memory.
  - When one reference variable is assigned to another, the object is not copied; both variables refer to the same object.
  - Modifying the value of one variable will affect others.

```
int[] a1 = {4, 5, 2, 12, 14, 14, 9};
int[] a2 = a1;  // refer to same array as a1
a2[0] = 7;
System.out.println(a1[0]);  // 7
```

Null

- **null**: A reference that does not refer to any object.
  - Fields of an object that refer to objects are initialized to null.
  - The elements of an array of objects are initialized to null.

```
String[] words = new String[5];
DrawingPanel[] windows = new DrawingPanel[3];
```
Null pointer exception

- **dereference**: To access data or methods of an object with the dot notation, such as `.length()`.
  - It is illegal to dereference `null` (causes an exception).
  - `null` is not any object, so it has no methods or data.

```java
String[] words = new String[5];
System.out.println("word is: " + words[0]);
words[0] = words[0].toUpperCase();
```

Output:
```
word is: null
Exception in thread "main"
java.lang.NullPointerException
at Example.main(Example.java:8)
```

Classes and objects (8.1)

- **class**: A program entity that represents either:
  1. A program / module, or
  2. A template for a new type of objects.

  - The `Scanner` class is a template for creating `Scanner` objects.

- **object**: An entity that combines state and behavior.
  - **object-oriented programming (OOP)**: Programs that perform their behavior as interactions between objects.

Fields (8.2)

- **field**: A variable inside an object that is part of its state.
  - Each object has its own copy of each field.
  - **encapsulation**: Declaring fields `private` to hide their data.

  Declaration syntax:
  ```java
  private type name;
  ```

  Example:
  ```java
  public class Student {
      private String name; // each object now has 
      private double gpa; // a name and gpa field
  }
  ```

Instance methods

- **instance method**: One that exists inside each object of a class and defines behavior of that object.

  ```java
  public type name(parameters) {
      statements;
  }
  ```

  ```java
  Example:
  public void shout() {
      System.out.println("HELLO THERE!");
  }
  ```
A Point class

```
public class Point {
    private int x;
    private int y;
    // Draws this Point object.
    public void draw(Graphics g) {
        g.fillOval(x, y, 3, 3);
        g.drawString("(" + x + ", " + y + ")", x, y);
    }
}
```

- Each Point object contains data fields named `x` and `y`.
- Each Point object contains a method named `draw` that draws that point at its current `x/y` position.

The implicit parameter

- **implicit parameter**: The object on which an instance method is called.
  - During the call `p1.draw(g);` the object referred to by `p1` is the implicit parameter.
  - During the call `p2.draw(g);` the object referred to by `p2` is the implicit parameter.

  The instance method can refer to that object’s fields.
  - We say that it executes in the context of a particular object.
  - `draw` can refer to the `x` and `y` of the object it was called on.

Kinds of methods

- **Instance methods** take advantage of an object’s state.
  - Some methods allow clients to access/modify its state.

- **accessor**: A method that lets clients examine object state.
  - Example: A `distanceFromOrigin` method that tells how far a Point is away from (0, 0).
  - Accessors often have a non-`void` return type.

- **mutator**: A method that modifies an object’s state.
  - Example: A `translate` method that shifts the position of a Point by a given amount.

Constructors (8.4)

- **constructor**: Initializes the state of new objects.

```
public type(parameters) {
    statements;
}
```

  Example:
  ```
  public Point(int initialX, int initialY) {
      x = initialX;
      y = initialY;
  }
  ```

  - runs when the client uses the `new` keyword
  - does not specify a return type; implicitly returns a new object
  - If a class has no constructor, Java gives it a default constructor with no parameters that sets all fields to 0.
toString method (8.6)

- tells Java how to convert an object into a String

```java
public String toString() {
    code that returns a suitable String;
}
```

- Example:

```java
public String toString() {
    return "(" + x + ", " + y + ")";
}
```

- called when an object is printed/concatenated to a String:

```java
Point p1 = new Point(7, 2);
System.out.println("p1: "+ p1);
```

- Every class has a toString, even if it isn’t in your code.
- Default is class’s name and a hex number: Point@9e8c34

this keyword (8.7)

- this: A reference to the implicit parameter.
  - implicit parameter: object on which a method is called

- Syntax for using this:
  - To refer to a field:
    ```java
    this.field
    ```
  - To call a method:
    ```java
    this.method(parameters)
    ```
  - To call a constructor from another constructor:
    ```java
    this(parameters)
    ```

Static methods

- static method: Part of a class, not part of an object.
  - shared by all objects of that class
  - good for code related to a class but not to each object’s state
  - does not understand the implicit parameter, this;
    therefore, cannot access an object’s fields directly
  - if public, can be called from inside or outside the class

- Declaration syntax:

```java
public static type name(parameters) {
    statements;
}
```

Inheritance (9.1)

- inheritance: A way to form new classes based on existing classes, taking on their attributes/behavior.
  - a way to group related classes
  - a way to share code between two or more classes

- One class can extend another, absorbing its data/behavior.
  - superclass: The parent class that is being extended.
  - subclass: The child class that extends the superclass and inherits its behavior.
    - Subclass gets a copy of every field and method from superclass
### Inheritance syntax (9.1)

```java
public class name extends superclass {
    // Example:
    public class Secretary extends Employee {
    ... }

    // By extending Employee, each Secretary object now:
    - receives a getHours, getSalary, getVacationDays, and getVacationForm method automatically
    - can be treated as an Employee by client code (seen later)
}
```

### Overriding methods (9.1)

- **override**: To write a new version of a method in a subclass that replaces the superclass’s version.
  - No special syntax required to override a superclass method. Just write a new version of it in the subclass.

```java
public class Secretary extends Employee {
    // overrides getVacationForm in Employee class
    public String getVacationForm() {
        return "pink";
    }
    ... }
```

### super keyword (9.3)

- Subclasses can call overridden methods with `super.method(parameters)`

```java
public class LegalSecretary extends Secretary {
    public double getSalary() {
        double baseSalary = super.getSalary();
        return baseSalary + 5000.0;
    }
    ... }
```

### Polymorphism

- **polymorphism**: Ability for the same code to be used with different types of objects and behave differently with each.
  - Example: `System.out.println` can print any type of object. Each one displays in its own way on the console.

```java
Employee ed = new LegalSecretary();
System.out.println(ed.getSalary()); // 55000.0
System.out.println(ed.getVacationForm()); // pink
```