SE 350 - Object-Oriented Software Development

Java Object-Oriented Programming Basics

Instructor: Stefan Mitsch

Learning Objectives

- Understand classes and objects
- Understand attributes and methods
- Understand constructors

Object-Oriented Programming

- Type of data + Functions = Object: group the variables and functions by context
- Procedural vs. Object-Oriented Programming
  - Building blocks: functions vs. objects
  - Variables and functions vs. object contains both
  - Functions operate on arguments vs. objects operate on own data and expose selectively
- Information hiding
- Code organization: model complex structures
- Reuse through composition and inheritance

Concepts Overview

- Classes and objects
- Inheritance
- Encapsulation
- Access modifiers
- Getters and setters
- Generics
- Abstraction, interfaces, mixins
- Polymorphism
- Constructors, destructors
- Association, aggregation, composition
- Static classes, static methods
- Delegates
- Methods, attributes

Classes and Objects

- Java is a statically-typed, class-based, object-oriented language
- Real-world objects have `state` (attributes) and `behavior` (methods)
- Class is a **blueprint** of objects
- Instantiation turns blueprint into an object

Class

- Group of objects with common properties
- User-defined datatypes
- Specifies methods and attributes
- Code reusability, maintenance

Object

- Instance of a class
- Every object has 3 characteristics
  - Identity (memory address)
  - State (attributes)
  - Behavior (methods)

Classes and Objects in Java

Class

- **Logical** entity
- How to create a class?

```java
public class Cat {
  private int weight;
  public Cat(int weight) {
    this.weight = weight;
  }
  public String sound() {
    return "Meow";
  }
}
```

Object

- **Physical** entity
- How to create an object?

```java
Cat c = new Cat(3);
```

  - Declaration: `Cat c;`
  - Instantiation: `new Cat(...)`
  - Initialization: `new Cat(3)`

Attributes and Methods

Attributes

- **State** of an object
- Also called properties, fields, or instance variables
- Each object has own copy in memory

Methods

- **Behavior** of an object
- Return or update attributes
- Method parameters: pass data to object
- Method overloading: reuse method name with different parameters

Constructors and Destructors

- Constructor
  - Initializes a new object
  - Can be overloaded
  - No return type
  - Default constructor: non-parameterized
  - Parameterized constructor: takes arguments
- Destructor
  - Clean up object resources
  - Not in Java (automated garbage collection)

Example

```
class Date {
  private int day;
  private int month;
  private int year;
  public Date() {
    // default constructor
    day = 0;
    month = 0;
    year = 0;
  }
  public Date(int d, int m, int y) {
    // parameterized constructor
    day = d;
    month = m;
    year = y;
  }
}
```

Java OOP Cheat Sheet

```
public class Charge {
  // Instance variables
  private final double rx, ry, q;
  // Constructor with arguments x0, y0, q0
  public Charge(double x0, double y0, double q0) { rx = x0; ry = y0; q = q0; }
  // Instance methods with arguments x, y
  public double potentialAt(double x, double y) {
    double k = 8.99e09; // local variable
    double dx = x - rx; // difference between argument x and instance variable rx
    double dy = y - ry;
    return k * Math.sqrt(dx*dx + dy*dy); // calls static method in class Math
  }
  public String toString() { return q + " at " + "(" + rx + ", " + ry + ")"; }
  // Test client
  public static void main(String[] args) {
    double x = Double.parseDouble(args[0]);
    double y = Double.parseDouble(args[1]);
    Charge c = new Charge(0.5, 0.6, 21.0); // create and initialize object
    System.out.println(c.potentialAt(x, y)); // call method
  }
}
```

Example 1: Initialization

```
public class Example1 {
  // a public field with optional initialization
  public int i = 1;
  // example usage
  public static void main(String[] args) {
    System.out.println("Example 1");
    Example1 oa = new Example1();
    Example1 ob = new Example1();
    System.out.println("oa.i = " + oa.i);
    System.out.println("ob.i = " + ob.i);
    oa.i = 2;
    System.out.println("oa.i = " + oa.i);
    System.out.println("ob.i = " + ob.i);
  }
}
```

- When is the field `i` being initialized?
- What is the value of `i` if we omit initialization?
- Where is the constructor `Example1()`?
- Constructors do not have a return type. Is it `void`?
- Is it bad practice to have public fields?

Example 2: Compiles or Not?

```
public class Example2 {
  int i;
  // a parameterized constructor
  public Example2(int i) { this.i = i; }

  public static void main(String[] args) {
    Example2 oa = new Example2();
  }
}
```

- Does it compile? Why or why not?
- If it does not compile: how to fix the code?
- What is the meaning of `this`?

Example 3: Constructor Overloading

```
public class Example3 {
  int i;
  public Example3()      { this.i = 5; }
  public Example3(int i) { this.i = i; }

  public static void main(String[] args) {
    Example3 oa = new Example3();
    Example3 ob = new Example3(1);
  }
}
```

- Constructor overloading: how does Java distinguish between constructors?
- When can `this` be omitted?
- What is the difference between a local variable and an instance variable?

Example 4: Variable Arguments

```
public class Example4 {
  public int sum(int... values) {
    int total = 0;
    for (int i : values) total = total + i;
    return total;
  }

  public static void main(String[] args) {
    Example4 oa = new Example4();
    System.out.println(oa.sum(1, 2));
    System.out.println(oa.sum(1, 2, 3, 4));
  }
}
```

- Variable argument lists with `...`
- What is bad practice about the example?

Summary

- Objects hide internal details, can be reused, passed as arguments
- Classes are logical entities, groups of objects with common properties
- Objects are class instances
- Constructors instantiate and initialize objects
- Scope and lifetime of variables are important: local vs. instance variables
- `this` to explicitly refer to the current object

Exercises

- Implement a Fibonacci class that memoizes results instead of recomputing every time
- Implement a Nim game
  - Class that represents a board in the Nim game
    - The board holds a number of pieces, set in constructor, can be read
    - Up to half of the pieces can be removed from the board, must remove at least 1 piece
  - Player who removes the last piece loses the game