SE 350 - Object-Oriented Software Development

Java Object-Oriented Programming Basics

Instructor: Stefan Mitsch

Learning Objectives

- Understand difference between static members and instance members
- Understand access modifiers
- Understand nested classes

Static Methods and Variables

Class Member `static`

- Common to all instances of a class
- Useful to define procedures
- Useful to define (class-)*global* variables
- No access to instance members

Instance Member

- Each instance has own member

```
public class Rectangle {
  // static method (procedure)
  public static double area(double width, double length) {
    return width*length;
  }
}
```

Access Modifiers

- Specify visibility of members
  - `private` : accessible only in the same class (*objects of the same class can access each others private members!*)
  - package-private (default): accessible in the same package
  - `protected` : accessible in the same package and in sub-classes
  - `public` : accessible everywhere
- Specify visibility of classes
  - package-private (default): accessible in the same package
  - `public` : accessible everywhere

Example: Access Modifiers

```
public class AccessModifiers {
  public int i = 1;
  public void printI() { System.out.println(i); }
  private int j = 2;
  private void printJ() { System.out.println(j); }

  public static void main(String[] args) {
    AccessModifiers oa = new AccessModifiers();
    System.out.println("oa.i = " + oa.i);
    oa.printI();
    System.out.println("oa.j = " + oa.j);
    oa.printJ();
  }
}
```

- Why can `main` (left) access `j`?
- Why can `main` (below) not access `j`?

```
public class Outside {
  public static void main(String[] args) {
    AccessModifiers oa = new AccessModifiers();
    System.out.println("oa.i = " + oa.i);
    oa.printI();
    //System.out.println("oa.j = " + oa.j);
    //oa.printJ();
  }
}
```

Encapsulation

- How to access `private` members?
  - Through public methods
  - Variables: getters and setters
- Benefits
  - Controlled access
  - Increased data correctness
  - Read-only, write-only

```
public class GetterSetter {
  private int i;
  public int getI() { return i; }
  public void setI(int i) {
    // check contract before changing value
    this.i = i;
  }
}

public class Outside {
  public static void main(String[] args) {
    GetterSetter oa = new GetterSetter();
    oa.setI(5);
    System.out.println("oa.getI() = " + oa.getI());
  }
}
```

Initialization Blocks

- Initialization alternative to constructors
  - `static` : initialize class variables
  - `non-static` : initialize instance variables
- Executed before constructors
- Executed in the order of appearance in the class
- Useful for accessing resources during initialization (file, database, ...)

```
public class Initializer {
  private static int s;
  int a, b, c;
  static {
    System.out.println("Initializing s");
    s = 0;
  }
  {
    System.out.println("Initializing a");
    a = 1;
  }
  {
    System.out.println("Initializing b");
    b = 2;
  }
  public Initializer() {
    System.out.println("Constructor sets c");
    c = 3;
  }

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

Nested Classes

- Class declared in the body of other class
- Inner class has access to outer class
- Improved encapsulation
- Logical grouping
- Accessible inner classes can be instantiated also from outside

```
public class Outer {
  private int i = 1;

  // nested class
  private class Inner {
    private int j = 2;
    public void print() {
      System.out.println("i = " + ", j = " + j);
    }
  }

  public void print() {
    Inner ob = new Inner();
    ob.print();
  }

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

Object Cloning

- Alternative to construction from scratch
- Copy constructor, cloning, serialization and deserialization
- Java does not support default copy constructors
  - Implement own copy constructor (recommended)
  - Implement `Cloneable` (not recommended, Josh Bloch)

```java
public class Copyable {
  private int i;
  public Copyable(int i) { this.i = i; }

  // explicit copy constructor
  public Copyable(Copyable other) {
    this.i = other.i;
  }

  public void print() {
    System.out.println("i = " + i);
  }

  public static void main(String[] args) {
    Copyable oa = new Copyable(5);
    Copyable ob = new Copyable(oa);
    ob.print();
  }
}
```

Summary

- Static members do not require instances: global data and functions
- Instance members are local to an object: local data and methods
- Use access modifiers to distinguish public interface from private implementation
- Nested classes can help organize class implementation
    - Useful for internal reusable code
    - Objects of nested class have access to outer instance variables

Exercises

- Add access modifiers

```
class SmallInts {
  int[] ints;

  Int(int n) { ... }
  Int(int... is) { ... }

  boolean isValid(int[] is) { Arrays.stream(is).allMatch(i -> i <= 20); }

  setInts(int... is) {
    if (!isValid(is)) throw new IllegalArgumentException("...");
    ints = is;
  }

  getInts() { return ints; }
}
```