CIS2571 – Intro to Java

Chapter10 → Thinking in Objects

Information consolidated from a variety of textbook and online resources

Topic Objectives

- Immutable Objects and Classes
- Variable Scope
- this Reference
- Class Abstraction and Encapsulation
- Object-Oriented Thinking
- Object Composition
- Class Design Guidelines
- Wrapper Classes
  - Conversion Between Primitive Types and Wrapper Objects
- BigInteger and BigDecimal Class

Object Oriented Programming (OOP)

- Just like the benefits of modularizing code with methods, OOP with classes helps to:
  - Reduce redundant code
  - Enable code reuse
  - Improve program quality and consistency
- Object whose contents cannot be changed
  - Immutable object from immutable class
    - Data fields must be private
    - Cannot provide mutator methods for data fields
    - Cannot provide accessor methods that return reference to mutable data field object
      - Reference can be changed

→Examples

Object Oriented Programming (OOP)

```
public class Student {
  private int id;
  private String name;
  private java.util.Date dateCreated;
  public Student() {
  }
  public Student(int ssn, String newName) {
    id = ssn;
    name = newName;
    dateCreated = new java.util.Date();
  }
  public int getId() {
    return id;
  }
  public String getName() {
    return name;
  }
  public java.util.Date getDateCreated() {
    return dateCreated;
  }
}
```

private data members

public accessor methods

Object Oriented Programming (OOP)

```
public class Test {
  public static void main(String[] args) {
    Student student = new Student(111223333, "John");
    java.util.Date dateCreated = student.getDateCreated();
    dateCreated.setTime(2000000); // Now dateCreated field is changed!
  }
}
```

Variable Scope

- Instance and static class variables (or data fields)
  - Can be declared anywhere inside class definition
  - Have scope of entire class, regardless of declaration placement
    - Exception for data field initialization based upon reference to another data field
- Local variables
  - Declared (and initialized) before they are used in a method
  - Scope continues until end of block containing variable
  - If same name as class variable
    - Local variable takes precedence
    - Class variable is hidden *(but still can be accessed….)*

→Examples

Variable Scope

```
public class Circle {
  public double findArea() {
    return radius * radius * Math.PI;
  }
  private double radius = 1;
} // variable radius and method findArea() can
  // be declared in any order
```

```
public class Foo {
  private int i;
  private int j = i + 1;
} // i has to be declared before j because j’s
  // initial value is dependent upon i
```

Variable Scope

```
public class Foo {
  private int x = 0; // instance variable
  private int y = 0;
  public Foo() {
  }
  public void p() {
    int x = 1; // local variable hides instance
    System.out.println("x = " + x);
    System.out.println("y = " + y);
  } // output of x = 1 and y = 0
}
```

this Reference

- this keyword is name of reference that refers to calling object
  - Can use to refer to object's instance members

```
public class Circle {
  private double radius;
  . . .
  public double getArea() {
    return radius * radius * Math.PI;
  }
```

equivalent

```
public class Circle {
  private double radius;
  . . .
  public double getArea() {
    return this.radius * this.radius * Math.PI;
  }
```

this Reference

- this keyword is name of reference that refers to calling object
  - Used to references class’s hidden data fields
    - Remember, hidden static fields can be accessed using class name

```
public class F {
   int i = 5; // instance variable
   static double k = 0; // static variable
   void setI(int i) {
     this.i = i; // instance and local variable
   }
   static void setK(double k){
     F.k = k; // static and local variable
   }
}

Suppose f1 and f2 are objects of type F:
   f1.setI(10) will set this.i to 10 where this refers to f1
   f2.setI(45) will set this.i to 45 where this refers to f2
F.setK(33) will set static variable k to 33
```

After f1.setK(100) executes, what is the value of f2.k? of F.k?

this Reference

- this keyword name of reference that refers to calling object
  - Invoke an overloaded constructor

```
public class Circle {
  private double radius;

  public Circle(double radius) {
    this.radius = radius;
  }

  public Circle() {
    this(1.0);
  }

  public double getArea() {
    return this.radius * this.radius * Math.PI;
  }
}
```

this must be explicitly used to reference the data field radius of the object being constructed

this is used to invoke another constructor

Every instance variable belongs to an instance represented by this, which is normally omitted

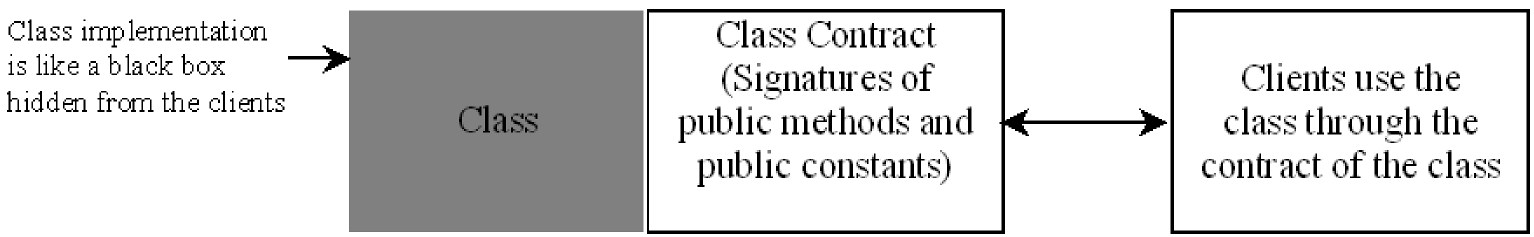
Class Abstraction and Encapsulation

- **Class abstraction** → separation of class implementation from class use
  - Class **creator** provides description of class and lets user know how class can be used
  - Class **user** (aka client) does not need to know how class is implemented, only how class can be used
- **Class encapsulation** → details of implementation are encapsulated and hidden from user

OOP Fundamentals

Class Abstraction and Encapsulation

- **Class contract** → description of methods/fields accessible from outside as well as how these members interface
- Class should provide a **variety of ways** for customization through constructors, properties, and methods



See 10.1 TestLoanClass.java
See 10.2 Loan.java

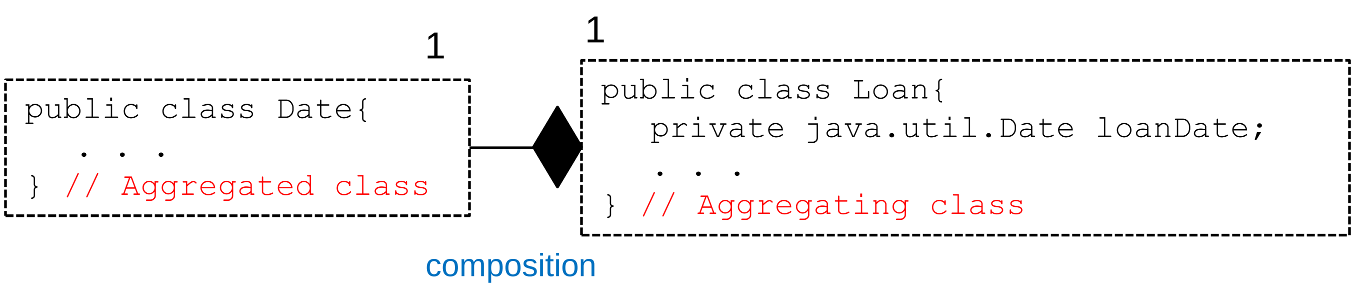
Object-Oriented Thinking

- Primitive data types, arrays, selection, looping, and methods provide the foundation for object oriented programming
- Software Design Methodologies
  - Procedural paradigm focuses on **designing methods**
  - Object-oriented paradigm couples **data and methods together into objects**
- Classes provide additional flexibility and modularity for building reusable software
- Java programs are a **collection of cooperating objects**

See 3.5 ComputeAndInterpretBMI.java
See 10.3 UseBMIClass.java
See 10.4 BMI.java

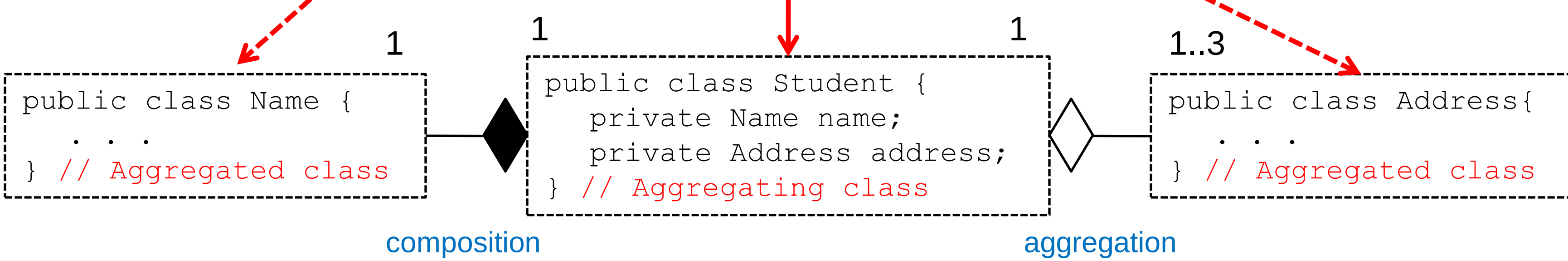
Object Composition

- Object can **contain** another object
  - Relationship between the two is referred to as composition
  - Composition special type of aggregation relationship
    - Object is **exclusively owned** by an aggregating object
  - Example:
    - Loan class contains a Date class field → loanDate



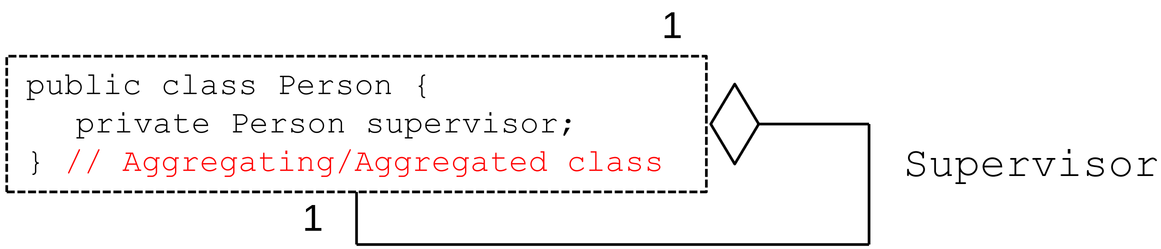
Object Composition

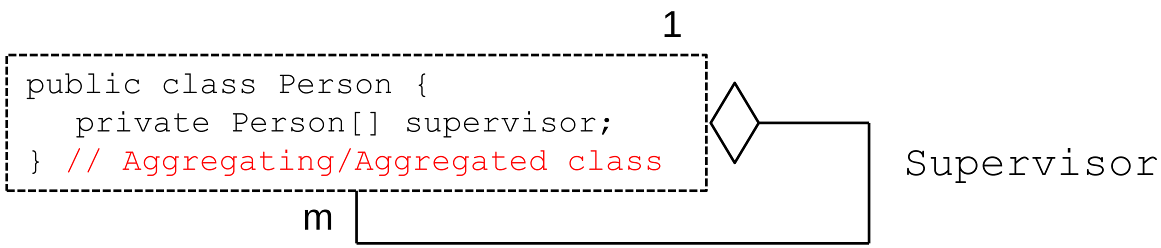
- Aggregation (**has-a**) relationship represents an ownership relationship between two objects
  - **Owner** → aggregating object/class
  - **Subject** → aggregated object/class
- Multiplicity specifies how many objects of class are involved in relationship
  - m..n → between m and n, inclusive
  - * → unlimited



Object Composition

- Aggregation may exist between objects of the same class





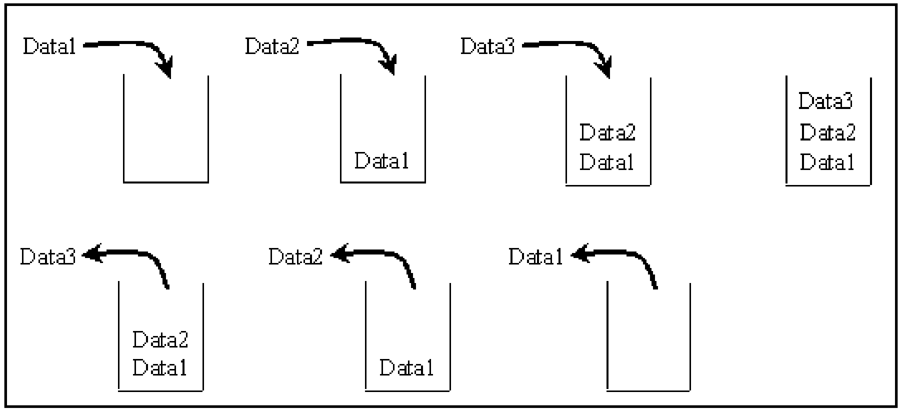
Object Composition Example

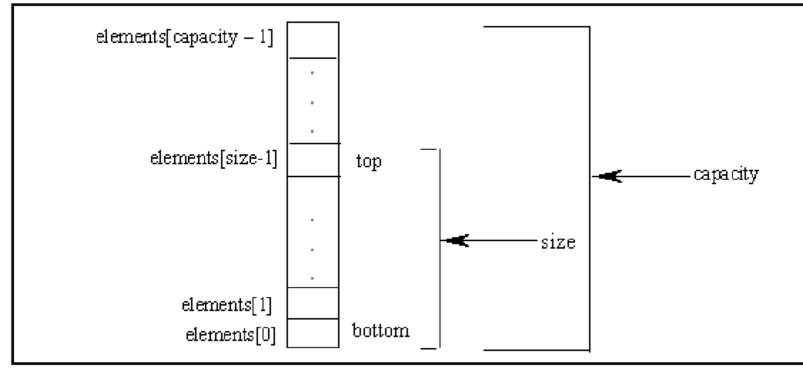
Course	
-courseName: String	The name of the course.
-students: String[]	An array to store the students for the course.
-numberOfStudents: int	The number of students (default: 0).
+Course(courseName: String)	Creates a course with the specified name.
+getCourseName(): String	Returns the course name.
+addStudent(student: String): void	Adds a new student to the course.
+dropStudent(student: String): void	Drops a student from the course.
+getStudents(): String[]	Returns the students in the course.
+getNumberOfStudents(): int	Returns the number of students in the course.

See 10.5 TestCourse.java
See 10.6 Course.java

Stack Class Example

Stack → data structure that holds data in LIFO (last in first out) fashion





StackOfIntegers
-elements : int [ ]
-size : int
<u>+DEFAULT_CAPACITY : int</u>
+StackOfIntegers()
+StackOfIntegers(capacity : int)
+empty() : boolean
+peek() : int
+push(value : int) : void
+pop() : int
+getSize() : int

An array to store integers in the stack.
The number of integers in the stack.
The static default capacity of a stack.

Constructs an empty stack with a default capacity of 16.
Constructs an empty stack with a specified capacity.
Returns true if the stack is empty.
Returns the integer at the top of the stack without removing it from the stack.
Stores an integer into the top of the stack.
Removes the integer at the top of the stack and returns it.
Returns the number of elements in the stack.

See 10.7 TestStackOfIntegers.java
See 10.8 StackOfIntegers.java

Class Design Guidelines

- **Cohesion**
  - describe a single entity, and all the class operations should logically fit together to support a coherent purpose
- **Consistency**
  - Follow standard Java programming style and naming conventions
  - Choose informative names for classes, data fields, and method
- **Encapsulation**
  - Use private modifiers for data fields to prevent direct access by clients
  - Provide public **accessor/mutator** methods as needed

Class Design Guidelines

- **Clarity**
  - Provide clear contract that is easy to explain and understand
    - properties and methods work in an order independent manner
- **Completeness**
  - Design for use by many different clients
- **Instance vs. Static**
  - Use **static** variable/method when shared by all instances
    - Instance variable/method **cannot** be accessed from static method
  - Use **instance** variable/method when dependent upon specific class instance
    - Constructors are **always** instance methods
    - Static variable/method **can be** accessed from instance method
      - **Should be** referenced from static method for consistency

Wrapper Classes

- For performance reasons, primitive data types are not used as objects
- Java includes classes to "wrap", or convert, a primitive data type into an object when the need for an object exists (generic programming)
  - Byte
  - Double
  - Float
  - Integer
  - Long
  - Short
- Inherited from Number
  - byteValue()
  - doubleValue()
  - floatValue()
  - intValue()
  - longValue()
  - shortValue()
  - byteValue()

Wrapper Classes

- Wrapper classes have
  - MIN_VALUE and MAX_VALUE
  - Overloaded parsing methods based upon base 10 or another specified radix

java.lang.Integer
-value: int +MAX_VALUE: int +MIN_VALUE: int
+Integer(value: int) +Integer(s: String) +byteValue(): byte +shortValue(): short +intValue(): int +longVlaue(): long +floatValue(): float +doubleValue():double +compareTo(o: Integer): int +toString(): String +valueOf(s: String): Integer +valueOf(s: String, radix: int): Integer +parseInt(s: String): int +parseInt(s: String, radix: int): int

java.lang.Double
-value: double +MAX_VALUE: double +MIN_VALUE: double
+Double(value: double) +Double(s: String) +byteValue(): byte +shortValue(): short +intValue(): int +longVlaue(): long +floatValue(): float +doubleValue():double +compareTo(o: Double): int +toString(): String +valueOf(s: String): Double +valueOf(s: String, radix: int): Double +parseDouble(s: String): double +parseDouble(s: String, radix: int): double

→Examples

Wrapper Classes Examples

- Wrapper classes
  - Can construct object from primitive data type or from string
    - Have immutable instances (no no-arg constructor)
  - Include static range constants
  - Static method **valueOf(String s)** to create new object represented by specified string
  - Static **parse** methods to convert from String to primitive value

```
Double dObj1 = new Double(5.0);
Double dObj2 = new Double("5.0");

System.out.println("The maximum integer is " + Integer.MAX_VALUE);
System.out.println("The minimum positive float is " + Float.MIN_VALUE);

Double doubleObject = Double.valueOf("12.4");
Integer integerObject = Integer.valueOf("12");

System.out.println(Integer.parseInt("13")); // prints 13
System.out.println(Integer.parseInt("D", 16)); // prints 13
```

See WrapperTest.java

Conversion between Primitive Types and Wrapper Class Types

- Java allows primitive type and wrapper classes to be converted automatically
  - boxing/autoboxing → converting a primitive value to a wrapper object
    - Compiler will automatically 'box' a primitive value that appears in context requiring object

equivalent

```
Integer intObject = new Integer(2);
```

⟺

```
Integer intObject = 2;
```

```
Integer[] intObjArr = {new Integer(1),
new Integer(2), new Integer(3)};
```

⟺

```
Integer[] intObjArr =
{1, 2, 3};
```

↑ autoboxing

Conversion between Primitive Types and Wrapper Class Types

- Java allows primitive type and wrapper classes to be converted automatically
  - unboxing/autounboxing → converting a wrapper object to a primitive value
    - Compiler will automatically 'unbox' an object that appears in context requiring a primitive value

```
Integer[] intObjArr = {new Integer(1), new Integer(2), new Integer(3)};
```

equivalent

```
System.out.println(intObjArr[0]
+ intObjArr[1] + intObjArr[2]);
```

⟺

```
System.out.println(1 + 2 + 3);
```

↑ autounboxing

BigInteger and BigDecimal Clases

- BigInteger and BigDecimal classes are used to compute with very large integers or high precision floating-point values
  - Immutable classes
  - BigInteger can represent integer of any size
  - No limit to precision of BigDecimal object

```
BigInteger a = new BigInteger("9223372036854775807");
BigInteger b = new BigInteger("2");
BigInteger c = a.multiply(b); // 9223372036854775807 * 2
System.out.println(c);

BigDecimal a = new BigDecimal(1.0);
BigDecimal b = new BigDecimal(3);
BigDecimal c = a.divide(b, 20, BigDecimal.ROUND_UP);
System.out.println(c); // 0.33333333333333333334
```

avoid ArithmeticException error by specifying scale and rounding

See 10.11 LargeFactorial.java