4. Java OOP

3. Encapsulation

Class Access Modifiers

- If a class has no modifier (the default, also known as *package-private*), it is visible only within its own package
- Modifier *public* means that class is visible to all classes everywhere

Methods Access Modifiers

- *public* - visible to all classes everywhere
- *no modifier* (*package-private*) - visible only within its own package
- *protected* - accessed within its own package and by a subclass of its class in another package
- *private* - can only be accessed in its own class

Fields Access

- Avoid public fields except for constants
- Public fields tend to link you to a particular implementation and limit your flexibility in changing your code
- Use special methods to get and/or set class field value

Static Fields and Methods

- **static** keyword is used to create fields and methods that belong to the class
- static fields and methods are referenced by the class name itself

Static Fields

- every instance of the class shares a static field
- any object can change the value of a static field
- static field can be manipulated without creating an instance of the class
- static field can be used to determine a number of created objects for example

Static Field Example

```
public class Employee{
    private int id;
    private static int nextId = 1;
     public Employee(){
      id = nextId;
      nextId++;
    }
    .    .    .    .    .    .
}
```

Static Methods

- Instance methods can access instance and static variables/methods directly.
- Class methods can access class variables and class methods directly.
- Class methods ***cannot*** access instance variables or instance methods directly—they must use an object reference.
- Also, class methods ***cannot*** use the this keyword as there is no instance for this to refer to.

Static Method Examples

- You can add to the Employee class below the following static method:

```
public static int getNextId(){
    return nextId;
}
```

- Methods of Math class are static:

```
Math.sqrt(x)
Math.round(y)
```

Static Methods Invocation

- Use the following construction for static method call:

  ClassName.method(paremeterList);

- Examples:

```
int n = Employee.getNextId();
double x = 2.0;
double y = Math.sqrt(x);
```

Constants

- The *static* modifier, in combination with the *final* modifier, is also used to define constants
- Constants defined in this way cannot be reassigned
- The names of constant values are spelled in uppercase letters

Constants Example

- Static variables are quite rare
- Static constants are more common
- The Math class defines a static constant:

```
public class Math {
   . . .
   public static final double PI = 3.14159265358979323846;
   . . .
}
```

- You can access this constant as Math.PI

Private Constructor

- Private constructors prevent a class from being explicitly instantiated by callers
- Private constructor can be useful if:
  - classes containing only static utility methods
  - classes containing only constants
  - type safe enumerations

Initializing Fields

- You can often provide an initial value for a field in its declaration
- If initialization requires some logic, simple assignment is inadequate
- Instance variables can be initialized in constructors
- How to provide the same capability for static fields?

Static Initialization Blocks

- A *static initialization block* is a normal block of code enclosed in braces, { }, and preceded by the static keyword:

```
static {
// whatever code is needed for initialization goes here
}
```

- A class can have any number of static initialization blocks
- They can appear anywhere in the class body

Manuals

- http://docs.oracle.com/javase/tutorial/java/javaOO/index.html

Exercise 4.3.1: SimpleDepo Class

- Create a class for simple deposit, that calculates interest for paying on maturity date as follows:

interest = sum * (interestRate / 100.0) * (days / 365 or 366)

Step by Step Solution

1. Check problem definition. If it is clear go to step 2
2. Create class
3. Describe class fields
4. Create constructors and accessors
5. Create method signatures
6. Create unit tests
7. Create method bodies

Test Cases

Start Date	Day Long	Sum	Interest Rate	Interest
08.09.2012	20	1000	15	8.20
08.09.2012	180	1000	15	73.84
08.09.2014	20	1000	15	8.22
12.09.2014	180	1000	15	73.97

Exercise: SimpleDepo Class

- See 431DepoSimple project for full text

JUnit Testing

- JUnit is a simple framework to write repeatable tests
- We'll create unit tests for SimpleDepo class using Junit with the following steps:
  - Create new 431aSimpleDepoTest project
  - Copy DepoSimple class to this project
  - Create JUnit test case
  - Create test methods
  - Run tests

Create JUnit Test Case

1. Open the New wizard (**File > New > JUnit Test Case**).
2. Select **New Junit 4 test** and enter "*TestAll"* as the name of your test class
3. Click **Finish** to create the test class
4. Click Ok in a warning message window asking you to add the junit library to the build path

Create Test Methods (1 of 2)

```
@Test
public void test1() {
DepoSimple depo = new DepoSimple();
depo.setStartDate(new GregorianCalendar(2012,
   Calendar.SEPTEMBER, 8).getTime());
depo.setDayLong(20);
depo.setSum(1000.00);
depo.setInterestRate(15.0);
double interest = 0.0;
```

Create Test Methods (2 of 2)

```
    try{
     interest = depo.getInterest();
    }
    catch(Exception ex){
     fail("Error: " + ex.getMessage());
    }
    assertEquals(8.20, interest, 0.005);
}
```

Run Tests I

- To run TestAll hit the run button in the toolbar
- You can inspect the test results in the *JUnit* view
- You can rerun a test by clicking the **Rerun** button in the view's tool bar

Run Tests II

- Run all tests inside a project or package:
  Select a project or package run all the included tests with **Run as > JUnit Test**
- Run a single test method:
  Select a test method in the Outline or Package Explorer and choose **Run as > JUnit Test**

JUnit Manual

- http://junit.sourceforge.net/doc/cookbook/cookbook.htm

Exercise 4.3.2.

- Create BarrierDepo class to calculate interest accordingly to the following:
  - If sum <= 50000.0 then

    interest = sum * (interestRate / 100.0) * (days / 365 or 366)
  - If 50000.0 < sum < 100000.0 interestRate is increased by 1%
  - If sum > 100000.0 interestRate is increased by 2%
- Use JUnit for tests

Test Cases

Start Date	Day Long	Sum	Interest Rate	Interest
08.09.2012	20	1000	15	8.20
08.09.2012	30	60000	15	786.89
08.02.2014	30	60000	15	789.04
12.05.2014	180	100001	15	8383.65

Exercise 4.3.2.

- See 432BarrierDepo project for the full text

Home Exercise 4.3.3: DepoMonthCapitalize Class

- Modify SimpleDepo class to calculate interest with monthly capitalization (calculated interest every month is added to the deposit sum)

Test Cases

Start Date	Day Long	Sum	Interest Rate	Interest
08.09.2013	20	1000	15	8.22
08.09.2013	30	1000	15	12.36
12.05.2014	180	1000	15	76.32