01 Objects and Classes

Instructor: Ke Wei（柯韋）

➽ A319 ✆ Ext. 6452 ✉ wke@ipm.edu.mo

http://brouwer.ipm.edu.mo/COMP212/19/

Bachelor of Science in Computing, School of Applied Sciences, Macao Polytechnic Institute

August 22 (September 3), 2019

Text Books and References

- Y. D. Liang (2014).
  Introduction to Java Programming – Comprehensive, 10th Edition.
  Prentice Hall.

- J. Bloch (2008).
  Effective Java, 2nd Edition.
  Addison-Wesley.

- P. Deitel and H. Deitel (2014).
  Java SE8 for Programmers, 3rd Edition.
  Prentice Hall.

- B. Eckel (2006).
  Thinking in Java, 4th Edition.
  Prentice Hall.

Outline

1. Text Books and References
2. Course Overview
3. Object-Oriented Programming
4. Objects and Classes
5. Static Variables and Methods
6. Summary of Variables
7. Data Field Encapsulation
8. Practice

Course Overview

This course covers the principles of object-oriented programming using Java language. Fundamental programming skills and methods related to object-oriented approaches are discussed. Topics include:

- objects and classes,
- encapsulation,
- abstract classes and interfaces,
- generics and collections,
- exception handling,
- threads and concurrency,
- functional programming.

Object-Oriented Programming (OOP)

- Data and related operations are placed in a single entity, called an *object*.
  - Common variables are shared among certain operations.
  - The variables are global only to these operations, but not to others.
  - The shared variables can be packaged into a common context for the related operations, as an object.
- Using objects improves software reusability and makes programs easier to develop and easier to maintain (flexibility, modularity, clarity).
- Programming in Java involves thinking in terms of objects.
- A Java program can be viewed as a collection of cooperating objects.

Objects and Classes

- An *object* represents an entity that can be distinctly identified.
- An object has a set of data *fields* (also known as *attributes*) of its own.
- An object has a set of *methods*, representing the operations on it. Invoking a method on an object means that you ask the object to perform a task.
- Objects of the same type are defined using a common *class*.
- A class is a *template* or blueprint that defines what an object's data fields and methods will be.
- An object is an *instance* of a class. There can be *many* instances of a class.

A Class Definition

```java
class Circle {
    private double x, y;   // center coordinate
    private double radius;
    public double getArea() { return Math.PI*radius*radius; }
    public boolean contains(double x, double y) {
        double dx = x - this.x, dy = y - this.y;
        return dx*dx + dy*dy <= radius*radius;
    }
}
```

- *x*, *y* and *radius* are data fields, defined as variables.
- Each instance of the class has its own copy of the variables, so they are called *instance variables*.
- *getArea* and *contains* are methods, defined for *all* instances of the class.
- When a method is called, it operates only on a particular instance: *theInstance*.*getArea*().

Constructors

- A *constructor* is a special kind of method designed to initialize the data fields of objects.
- A constructor operates on a newly created instance, called by "new".

```
new Circle()  // creates a new instance and returns the its reference
```

- The constructor has exactly the *same* name as the defining class. There can be multiple constructors, each with a different parameter list.
- One constructor can call other constructors of the class using "this" *before* any other statements.

```
class Circle {
    public Circle(double x, double y, double radius) { ... }
    public Circle(double radius) { this(0.0, 0.0, radius); }
    ...
}
```

The `this` Instance

- Within a method, "`this`" refers to the instance which the method is operating on.
- Within a method, if a name is not declared as a local variable or a parameter, it is prefixed with "`this.`" by default.
- A local variable or a parameter *hides* the field with the same name, to access the field, "`this.`" must be specified explicitly.
- When invoking *myCircle*.*contains*(1.0,1.0), `this`.*x* in *contains* refers to *myCircle*.*x*. When invoking `new` *Circle*(0.0,0.0,3.0), `this`.*radius* in the constructor refers to the field *radius* of the newly created instance.

Static Variables and Methods

- If you want all the instances of a class to share data, use *static variables*.
- All instances of the same class are affected if one instance changes the value of a static variable.
- A class can also have *static methods*, a static method can be invoked without an instance of the class, e.g. `Item.resetNumOfItems()`. A static method does not have the “`this`” reference.

```java
class Item {
    private static int numOfItems = 0;
    public Item() { numOfItems++; }
    public static int getNumOfItems() { return numOfItems; }
    public static void resetNumOfItems() { numOfItems = 0; }
}
```

Summary of Variables

- Variables are introduced by *typings*: $\boxed{T\ \ x}$.
- Instance variables: the typings are in class definitions.

```
class C { T x; }
```

- Static variables: the typings are in class definitions and decorated by `static`.

```
class C { static T x; }
```

- Local variables: the typings are in statement blocks.

```
void m() { T x; ... while ( e ) { S y; ... } ... }
```

- Parameters: the typings are in method parameter lists.

```
void m(T x, S y) { ... }
```

Data Field Encapsulation

- Data should only be operated by related operations, not arbitrarily.
- To prevent direct modifications of fields, the fields should be declared `private`. This is known as data field *encapsulation*.
- Encapsulation prevents data from being tampered.
- Encapsulation makes data easy to maintain.
- Data fields are get and set via public methods, in an abstract way. Such abstract data fields are often called *properties*.

```
    public PropertyType getProperty() { ... }
    public boolean isBooleanProperty() { ... }
    public void setProperty(PropertyType propertyValue) { ... }
```

Practice: Define and Test the *Loan* Class

Tasks:

1. Declare a class.
2. Import a class from the library.
3. Define data fields.
4. Define *getters* and *setters* for the data fields.
5. Define multiple constructors.
6. Define methods.
7. Define the *main* method in a test class.
8. Create instances.
9. Set and get properties.
10. Invoke methods.
11. Try out visibilities.

The *Loan* Class

```
import java.util.Date;

public class Loan {
    private double annualRate;
    private int years;
    private double amount;
    private Date startDate;
    public Loan() { this(7.5, 30, 100000.0); }
    public Loan(double annualRate, int years, double amount) {
       this.annualRate = annualRate;
       this.years = years;
       this.amount = amount;
       startDate = new Date();
    }                                                                           // ...
```

The *Loan* Class (2)

```
    public double getAnnualRate() { return annualRate; }
    public void setAnnualRate(double annualRate) {
        this.annualRate = annualRate;
    }
    public int getYears() { return years; }
    public void setYears(int years) {
        this.years = years;
    }
    public double getAmount() { return amount; }
    public void setAmount(double amount) {
        this.amount = amount;
    }
    public Date getStartDate() { return startDate; }                      // ...
```

The *Loan* Class (3)

```
    public double getMonthlyPayment() {
        double monthlyRate = annualRate / 1200;
        return amount * monthlyRate /
            (1 - Math.pow(1/(1+monthlyRate), years*12));
    }
    public double getTotalPayment() {
        return getMonthlyPayment() * years * 12;
    }
} // class Loan
```

$$a = \frac{x}{(1+r)} + \frac{x}{(1+r)^2} + \cdots + \frac{x}{(1+r)^n} = x\frac{1}{(1+r)}\frac{\frac{1}{(1+r)^n}-1}{\frac{1}{(1+r)}-1} = x\frac{1-\frac{1}{(1+r)^n}}{r}.$$

The *TestLoan* Class

```
public class TestLoan {
    public static void main(String... args) {
        Loan dfLoan = new Loan();
        System.out.println(dfLoan.getAnnualRate());
        System.out.println(dfLoan.getYears());

        Loan spLoan = new Loan(4.0, 25, 500000.0);
        System.out.println(spLoan.getMonthlyPayment());
        spLoan.setYears(15);
        System.out.println(spLoan.getMonthlyPayment());
    }
}
```

Hoemwork

1. Change data field *years* to public and assign to it directly:

   spLoan.*years* = 15;

2. Change *years* back to private, write the error message you get as a comment line in the source file.
3. Change back to *spLoan*.*setYears*(35)
4. Add a method

   getYearsByMonthlyPayment(double *maxMonthlyPayment*)

   to take a maximum monthly payment and set the number of years of the loan.
5. Invoke the new method on *spLoan* with the maximum monthly payment of 2000.
6. Write the result of *getYears*() and *getMonthlyPayment*() on *spLoan* as a comment line in the source file.
7. Zip your source files into Loan.zip for future upload.