Objects and Classes

Yangtao Ge

June 14, 2019

1 Introduction to Object-Oriented Programming

What is Object-Oriented Programming:
Programming with several **objects**, each object has a specific functionality which exposed to its users, but a hidden implementation

Two Ways of thinking:

- Traditional: algorithms → data structures
  Note: fine for small problems but *cannot* handle large problems.
- Morden: data structures → algorithms
  Note: More efficient to **store** data first then **m**anipulate them

1.1 Classes

Class $\xrightarrow{Construct}$ Instance $\xleftarrow{Use}$ program

Encapsulation is the key of OOP:

- **Definition**: It is combining data and behavior in one package and hiding the implementation detail from the users of the object
- **How**: methods *never* directly access instance field in a class than its own i.e. "Black Box behaviour"

1.2 Objects

Three characteristics:

- behaviour: what can it do + what can be done to it
- state: how does the object react when use its method
- identity: how is the object distinguish from others

1.3 Identifying Classes

A Common begin of OOP design: Identify the classes and Add methods to sperate classes

Rule of Naming:

- Class Name: Nouns → What it is
- Method Names: Verbs → What can it do

1.4 Relationships between classes

Common Relations are:

dependence "uses-a" Express a relationship one class manipulates another class

aggregation "has-a" Express a relationship specifying the whole and its parts

inheritance "is-a" Express a relationship between a more special and a more general class

UML(Unified Modeling Language) notations aree used to expressed the relationship by diagram
Ref: p.131 Core Java, COMP0004 Note

2 Using Predefined Classes

2.1 Objects and Object Variables

A constructor is a **special method** whose purpose is to construct and initialize objects

Key facts between Object Variables and Objects:

- a variable called "deadline" with type "Date" is not a object but a variable
- object variables need to be initialized
- object variables doesen't contains an object, but it only *refers* to an object
- Explicitly, an object variable to **null** to indicate that it currently refers to no object

Two ways of INIT:

- *deadline = new Date();* refers to newly constructed object
- *deadline = birthdate;* refers to an existing object

2.2 The 'LocalDate' Class of the Java Library

Ref: pp. 135-137 Core Java

2.3 Mututator and Accessor Methods

Definitions:

- Mutator method: method which will change its own original value and return
- Accessor method: method which will **not** modify its original value

3 Defining Your Own Classes

3.1 Employee class

Basic Structures of A Non-Main Class:

- fields
- constructors i.e. could more than one constructor be found
- methods

Source file(*.java*) $\xrightarrow{compile}$ Compiled file(*.class*)

Example of *Employee class* is as follows:

```
import java.time.*;

public class employee{
    //instance fields
    private String name;
    private double salary;
    private LocalDate hireDay;

    //constructor
    public employee(String name, double salary, LocalDate hirDate){
        this.name = name;
        this.salary = salary;
        this.hireDay = hireDay;
    }

    //methods
    public String getName(){
        return name;
    }
    public void raiseSalary(double byPercent){
        double raise= salary * byPercent / 100;
        salary += raise;
    }
}
```

3.2 Use of Multiple Source Files

Two ways of execute source Files:

- "javac Employee*.java": all source files matching the wildcard will be comiled into class files
- "javac EmployeeTest.java": Find all classes mentiened in 'EmployeeTest' Class, Then compiles it

3.3 Dissecting the Employee Class

public and private:

- public: any methods in any class can call the method tagged with 'public'
- private: only the methods that can access these instance fields or methods are in the *Employee* class itself

3.4 First Steps with Constructors

Some Features of Constructors:

- has the **name** as the class
- can only be called in **conjunction** with *new* operator
  i.e. james.Employee("James Bond") is **WORNG**
- can take zero, one, or more parameters
- has **no** return values

3.5 Declaring Local Variable with 'Var'

var keyword can replace with their type.(Valid from Java 10) and it can only be used with *local* variable inside methods.
e.g.

```
Employee harry = new Employee("Harry Porter", 50000, 1989, 10, 1)
```

is the valid as:

```
var harry = new Employee("Harry Porter", 50000, 1989, 10, 1)
```

3.6 Working with null Reference

When using *null* reference three cases could possible:

- *NullpointerException*: end of execution
  two advantages:
  - has the description of the problem
  - finds the location of the problem
- "permissive": turn a null argument to non-null
  e.g.

  ```
  name = Objects.requireNoneNullElse(n, "unknown")
  ```

- "tough love": reject a null argument
  e.g.

  ```
  name = Objects.requireNoneNull(n, "Error with Null")
  ```

3.7 Implicit and Explicit Parameters

Definition of these two parameters:

- Implicit: the para appears *before* the method name
- Explicit: the para *in the paranthseses*

For example

```
number007.raiseSalary(5)
```

Here, *number007* is Implicit para, *5* is Explicit para.

3.8 Benefits of Encapsulation

Basic principle of Encapsulation:

- A private data field
- Accessor (getter)
- Mutator (setter)

Two Benefits:

- can change internal implementation without affecting any code other than the method of the class
- can perform error checking which can protect from any unexpected input

3.9 Class-Based Access Privileges

Method could be valid for *accessing the private data* of **all objects of its class**
e.g. equal method:

```
Class Employee{
    ...
    public boolean euqals(Employee other){
        return name.equals(other.name)
    }
}
```

N.B. this method call name of the current object and and name of 'other' which is another private field

3.10 Private Methods

Usually are used in 'help functions' to prevent accidentally call

3.11 Final Instance Field

Some Features of 'Final': (needs more reading)

- field value shoud be set after **the end of every constructor**
- the field may not be modified again
- usually used for *primitive type fields* or *immutable classes*
  i.e. *immutable class* means none of its method ever mutate its object

4 Static Fields and Methods

4.1 Static Fields

Definition: If you define a field *static*, then thereis only one such field per **class** (Not a instance)

Usually, it performs as a *counter*, which indicates how many instances have been created.
e.g.

```
Class Employee{
    private static int nextId = 1;  //a static field
    private int id;

    public void setId(){
        id = nextId;
        nextId++;
    }
}
```

Here, id belongs to each instance, but nextId belongs to the '**class Employee**'

4.2 Static Constants

Common to use 'static' in constants:

- **pi**: 'public static final double PI = 3.1415926535...'
- **out**: 'public static final PrintStream out = ...' i.e. this we commenly used 'System.out.println()'

Noticed that it is bad to have public fields but it is good to have public constants (usually tagged with '*static final*')

4.3 Static Methods

Definition: methods don't have a *this* parameter(with no Implicit parameter) i.e.

- static methods of *Employee class* cannot access the *id* instance field
- static methods of *Employee class* can access a static field.

```
int n = Employee.getNextId();
```

Two situations to use Static method: when a method

- doesn't need to access the object state(i.e. get info from Explicit paras) e.g. 'Math.Pow'
- only needs to *access* static field of the class
  e.g. 'Employee.getNextId'

4.4 Factory Methods

Ref: pp.159-160 more reading needed

Why using Factory methods rather than constructors:

- can't give diffrent two names to constructors
- can't vary the **type** of the constructed object

4.5 The main Method

main method is a static method.

Every class can have a main method for unit test:

- if testing Employee class in *isolation* – 'java Employee'
- if it is a part of large program – 'java Application'

5 Method Parameters

Definition

- **call by value**: gets the *value* that the caller provides
- **call by reference**: gets the *location* of the variable that the caller provides

****Java Only use **call by value** (ref: pp.166-167 – swap method)*

Two cases in java of method parameters:

- **parameter variables**(Primitive types): methods *cannot* modify the contents of any parameters variables passed to it.
  e.g.

```
public static void tripleValue(double){
    x = x * 3
}
double percent = 10
tripleValue(percent)
```

  what happens:

  1. x is init with a a copy of the value 'percent' (x = 10)
  2. x is tripled(x = 30), percent remains (percent = 10)
  3. method ends, x is no longer used

- **Object variables**(Object references): it is easily *possible* to implement Methods that change the state of an object parameter.
  e.g.

```
public static void tripleSalary(Employee x){
    x.raiseSalary(200)
}
harry = new Employee(...)
tripleSalary(harry)
```

  1. x is init with *a copy of* value of harry (object refernece)
  2. 'raiseSalary' method is applied to the object refernece(the object *harry* referred to get raised salary)
  3. methods ends, x is no longer used **BUT** *harry* continues to refer to the employee object

6 Object Construction

This section mentiends mechanisms for writing constructors

6.1 Overloading

Definition: Overloading occurs if several methods have the **same** name but **different** parameters i.e. constructor is a special method

the compile-time error occurs when:

- there is *no match* at all
- there is not one that is *better* than all others (?)

6.2 Default Field Initialization

The Default init:

- number: *0*

- boolean: *false*
- object ref: *null*

6.3 The Constructor with No Arguments

Two cases:

- No constructor at all: get a *free* no argument constructor
- User defined constructor:
  - have several Constructor: one with no argument
  - can't use no argument mechanism

6.4 Explicit Field Initialization

Two Ways of field init:

- Assign a value:

```
Class Employee{
    private String name = '''';
}
```

- Method call:

```
Class Employee{
    private static int nextId;
    private int id = assignId();
    ...
    private static int assignId(){
        int r = nextId;
        nextId++;
        return r;
    }
}
```

6.5 Parameter Names

Three types of parameter names:

- single letter names
- prefix with 'a' / 'an'
- using *this*
  e.g.

```
public Employee(String name, double salary){
    this.name = name;
    this.salary = salary;
}
```

6.6 Calling Another Constructor

Using *this* to refer another constructor in the current class.
e.g.

```
public Employee(double s){
    // calls Employee(String, double)
    this("Employee #" nextId, s);
    nextId++;
}
```

6.7 Initialization Blocks

3 Types of initialization:

- setting value in a constructor
- assigning a value in the declaration
- initialization blocks

Process of running a constructor:

1. The first line of constructor call the another constructor – execute the second constructor
2. Otherwise,
   (a) All data fields are init to *default*
   (b) All field initilizers and init blocks are executed
3. Body is executed

6.8 Object Destruction and the *finalize* Method

Java is automatic garbage collection – **NO** destructors
Ref: p.180 and Chapter 7