

Java-Success.com

Industrial strength Java/JEE Career Companion for those who want to go places

[Home](#)
[Java FAQs](#)
[600+ Java Q&As](#)
[Career](#)
[Tutorials](#)
[Member](#)
[Why?](#)
[Can u Debug?](#)
[Java 8 ready?](#)
[Top X](#)
[Productivity Tools](#)
[Judging Experience?](#)

[Home](#) › [member-paid](#) › Top 5 OOPs tips for Java developers

Top 5 OOPs tips for Java developers

Posted on November 28, 2014 by Arulkumaran Kumaraswamipillai — No

[Comments](#) ↓

Tip #1: Tightly encapsulate your classes. A class generally contains data as well as methods, and is responsible for the integrity of its own data. The standard way to protect the data is to make it private, so that no other class can get direct access to it, and then write a couple of public methods to get the data and set the data.

In the example below,

```
1 package com.oo;
2
3 import java.math.BigDecimal;
4
5 import org.apache.commons.lang.StringUtils;
6
7 public class Employee {
8
9     private String name;
10    private int age;
```

[9 tips to earn more](#) | [What can u do to go places?](#) | **945+** members. [LinkedIn Group](#). [Reviews](#)

600+ Full Stack Java/JEE Interview Q&As ♥Free ♦FAQs

[open all](#) | [close all](#)

☒ [Ice Breaker Interview](#)

☒ [Core Java Interview C](#)

☒ [Java Overview \(4\)](#)

☒ [Data types \(6\)](#)

☒ [constructors-methc](#)

☒ [Reserved Key Wor](#)

☒ [Classes \(3\)](#)

☒ [Objects \(8\)](#)

☒ [OOP \(10\)](#)

☒ [♥ Design princip](#)

☒ [♦ 30+ FAQ Java](#)

```

11 private BigDecimal salary;
12
13 public String getName() {
14     return name;
15 }
16 public void setName(String name) {
17     if(StringUtils.isEmpty(name)){
18         throw new IllegalArgumentException("Invalid name")
19     }
20     this.name = name;
21 }
22 public int getAge() {
23     return age;
24 }
25 public void setAge(int age) {
26     if(age < 0 || age > 100) {
27         throw new IllegalArgumentException("Invalid age")
28     }
29     this.age = age;
30 }
31 public BigDecimal getSalary() {
32     return salary;
33 }
34 public void setSalary(BigDecimal salary) {
35     if(salary.compareTo(BigDecimal.ZERO) < 0){
36         throw new IllegalArgumentException("Invalid salary")
37     }
38     this.salary = salary;
39 }
40
41 }
42
43
44

```

Data: name, age, and salary are made private. So, you can't directly set these values from outside like

```

1 Employee employee1 = new Employee();
2 employee1.age = -20; //Illegal. Compile Error
3

```

The data is encapsulated, and only access is possible via the public methods. The public methods fail-fast by validating the input data. It won't allow any negative ages by throwing an "IllegalArgumentException" at run time. So, your data is protected via compile-time and run time checks. The other fields like name and salary are encapsulated as well from illegal use.

```

1 Employee employee1 = new Employee();
2 employee1.setAge(20); // okay
3

```

- ◆ Why favor composition
- 08: ◆ Write code that explains itself
- Explain abstraction
- How to create a design
- Top 5 OOPs tips
- Top 6 tips to go from Junior to Senior
- Understanding Class
- What are good naming conventions
- GC (2)
- Generics (5)
- FP (8)
- IO (7)
- Multithreading (12)
- Algorithms (5)
- Annotations (2)
- Collection and Data Structures
- Differences Between Java and C++
- Event Driven Programming
- Exceptions (2)
- Java 7 (2)
- Java 8 (24)
- JVM (6)
- Reactive Programming
- Swing & AWT (2)
- JEE Interview Q&A (3)
- Pressed for time? Java Interview Questions
- SQL, XML, UML, JSC
- Hadoop & BigData Interview Questions
- Java Architecture Interview Questions
- Scala Interview Q&As
- Spring, Hibernate, & JSP
- Testing & Profiling/Static Analysis
- Other Interview Q&A for Java
- Free Java Interview Questions

As a Java Architect

[Java architecture & design concepts](#)

Tip #2: Hide non-essential details through **abstraction**. A good OO design should hide non-essential details through abstraction. **Encapsulation** is about hiding the implementation details whereas, **abstraction** is about providing a generalization.

For example, say you want to capture employment type like part-time, full-time, casual, semi-casual, and so on, it is a bad practice to define them as classes as shown below.

```
1 package com.oo;
2
3 public class PartTimeEmployee extends Employee {
4
5 }
6
```

```
1 package com.oo;
2
3 public class FullTimeEmployee extends Employee {
4
5 }
6
7
```

You will end up creating new classes for each new employment type, making your code more rigid and tightly coupled. The better approach is to **abstract** out the **employmentType** as a field. This way, instead of creating new classes for every employment type, you will be just creating new objects at run time with different **employmentType**.

```
1 public class Employee {
2
3     enum EmploymentType {PART_TIME, FULL_TIME, CASUAL}
4
5     private String name;
6     private int age;
7     private BigDecimal salary;
8     private EmploymentType employmentType;
9
10    // ... getters and setters
11 }
12
13
```

[interview Q&As with diagrams](#) | [What should be a typical Java EE architecture?](#)

Senior Java developers must have a good handle on

open all | close all

- ☐ Best Practice (6)
- ☐ Coding (26)
- ☐ Concurrency (6)
- ☐ Design Concepts (7)
- ☐ Design Patterns (11)
- ☐ Exception Handling (3)
- ☐ Java Debugging (21)
- ☐ Judging Experience (1)
- ☐ Low Latency (7)
- ☐ Memory Management (1)
- ☐ Performance (13)
- ☐ QoS (8)
- ☐ Scalability (4)
- ☐ SDLC (6)
- ☐ Security (13)
- ☐ Transaction Management (1)

80+ step by step Java Tutorials

open all | close all

- ☐ Setting up Tutorial (6)
- ☐ Tutorial - Diagnosis (2)
- ☐ Akka Tutorial (9)
- ☐ Core Java Tutorials (2)
- ☐ Hadoop & Spark Tutorials (1)

Tip #3: Loosely couple your classes. There are basically three relationships between classes

1. **is a** relationship. Also known as an **inheritance** and in UML terms **generalization**
2. **has a** relationship. Also known as a **composition** and in UML terms **association**.
3. **uses a** relationship. Also known as a delegation in UML terms.

Coupling is about the flow of data between modules. Which of the following definitions is loosely coupled?

Definition 1:

```






1 package com.oo;
2
3 import java.math.BigDecimal;
4
5 public interface SalaryProcessor {
6     BigDecimal processSalary(Employee employee)
7 }
8
9
```

Definition 2:

```











1 package com.oo;
2
3 import java.math.BigDecimal;
4
5 public interface SalaryProcessor {
6     BigDecimal processSalary(String name, Big
7 }
8
```

The **Definition 1** is more loosely coupled. If in the future the method **processSalary(...)** requires **employmentType** to process the salary, adding a new parameter to the method **processSalary(...)** can break all the classes that depends on it. But if you were to pass the **Employee** object as an argument, you don't have to change the **processSalary(...)** definition, but just change the implementation to use the **employmentType**.

-  [JEE Tutorials \(19\)](#)
-  [Scala Tutorials \(1\)](#)
-  [Spring & Hibernate T](#)
-  [Tools Tutorials \(19\)](#)
-  [Other Tutorials \(45\)](#)



Preparing for Java written & coding tests

[open all](#) | [close all](#)

-  [◆ Complete the given](#)
-  [Can you write code? I](#)
-  [Converting from A to I](#)
-  [Designing your classe](#)
-  [Java Data Structures](#)
-  [Passing the unit tests](#)
-  [What is wrong with th](#)
-  [Writing Code Home A](#)
-  [Written Test Core Jav](#)
-  [Written Test JEE \(1\)](#)

How good are your...to go places?

[open all](#) | [close all](#)

-  [Career Making Know-](#)
-  [Job Hunting & Resum](#)

Even if the *processSalary(..)* method needed a new parameter like say *leaveLoading*, you don't have to change the signature of the *processSalary()* method, and just add the new field to the *Employee* class, and include that in the implementation. The interfaces provide the contract, and you need to make minimal changes. The implementations can change as they don't break the contract.

```
1 public class Employee {
2
3     enum EmploymentType {PART_TIME, FULL_TIME, CASU
4
5     private String name;
6     private int age;
7     private BigDecimal salary;
8     private EmploymentType employmentType;
9     private BigDecimal leaveLoading;
10
11     //.....getters and setters
12
13 }
14
15
```

With the advent of IoC frameworks like Spring, you can increase loose coupling through dependency injection. Use the [Dependency Inversion principle for loose coupling](#).

Being loosely coupled goes hand in hand with the idea of **Separation of Concerns (SoC)**, which is the process of breaking a program into distinct features in order to reduce overlapping functionalities.

For example, the following *Employee* class tries to do too many things like being a model class, data access logic, and validation logic.

```
1 public class Employee {
2
3     enum EmploymentType {PART_TIME, FULL_TIME, CASU
4
5     private String name;
6     private int age;
7     private BigDecimal salary;
8     private EmploymentType employmentType;
9     private BigDecimal leaveLoading;
10
11
```

```
12 public List<employee> loadEmployees() {
13     //sql logic to load employees
14 }
15
16 public boolean validateEmployeeSalary() {
17     //validation logic
18 }
19
20     //getters and setters
21 }
22
23
```

The data access logic can be extracted out to a separate class.

```
1 public interface EmployeeDao {
2     abstract List<employee> loadEmployees() ;
3 }
4 </employee>
```

```
1 public class EmployeeDaoImpl implements Employee
2
3     @Override
4     public List<employee> loadEmployees() {
5         // TODO .....
6         return null;
7     }
8
9 }
10
```

Similarly, a *Validator* interface can be used for validation.

Tip #4: Favor composition over inheritance to get code reuse.

- With composition, you will have full control of your implementations. i.e., you can expose only the methods you intend to expose.
- With composition, it's easy to change behavior on the fly with Dependency Injection / Setters. Inheritance is more rigid as most languages do not allow you to derive from more than one type.

This does not mean that you can't use inheritance. Apply "Liskov Substitution & Interface Segregation Principles" to ensure that it makes sense to use inheritance.

In a more simpler terms, when you use inheritance to reuse code from the super class, rather than to override methods and define another polymorphic behavior, it's often an indication that you should use composition instead of inheritance. A typical example is that

In geometry a "Square **is a** rectangle". But in programming, a square is not a rectangle [as explained in detail here](#).

A **template-method** design pattern is a good example for using inheritance, which is often used in frameworks. The Gang of Four (GoF) design patterns favor composition over inheritance.

Tip #5: Use polymorphism.

So, use

```
1 List<Employee> employees = new ArrayList<Employee>  
2
```

instead of

```
1 ArrayList<Employee> employees = new ArrayList<Empl  
2
```

As a List can take different forms like ArrayList, AttributeList, CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, and Vector as they all implents the List interface. So, you can easily swith to CopyOnWriteArrayList if you want thread-safety, etc.

Finally.....be aware of the OO design principles and judiciously apply them where appropriate.

- **DRY (Don't repeat yourself):** Don't write duplicate code, instead use abstraction to abstract common things in one place.

- **Open Closed Principle(OCP):** The Open Close Principle states that the design and writing of the code should be done in a way that new functionality should be added with minimum changes in the existing code. The design should be done in a way to allow the adding of new functionality as new classes, keeping as much as possible existing code unchanged.
- **Single Responsibility Principle (SRP):** If you put more than one functionality in one Class in Java it introduce coupling between two functionality. This is also known as the SoC (Separation of Concerns) they was discussed earlier with ***EmployeeDao*** example.
- **Dependency Inversion Principle (DIP):** High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.
- **Interface Segregation Principle (ISP):** a client should not implement an interface if it doesn't use that. So, don't have fat interface with 10+ methods. Segregate them into separate interfaces.
- **Liskov Substitution Principle (LSP):** This is related to Single Responsibility Principle and Dependency Inversion Principle. Subtypes must be substitutable for super type. When you use inheritance to reuse code from the super class, rather than to override methods and define another polymorphic behavior, it's often an indication that you should use composition instead of inheritance as it breaks the LSP.

Popular Posts

♦ [11 Spring boot interview questions & answers](#)

861 views

♦ [Q11-Q23: Top 50+ Core on Java OOP Interview Questions & Answers](#)

829 views

18 Java scenarios based interview Questions and Answers

448 views

001A: ♦ 7+ Java integration styles & patterns interview questions & answers

407 views

♦ 7 Java debugging interview questions & answers

311 views

♦ 10 ERD (Entity-Relationship Diagrams) Interview Questions and Answers

303 views

01b: ♦ 13 Spring basics Q8 – Q13 interview questions & answers

294 views

01: ♦ 15 Ice breaker questions asked 90% of the time in Java job interviews with hints

288 views

♦ Q24-Q36: Top 50+ Core on Java classes, interfaces and generics interview questions & answers

263 views

8 Git Source control system interview questions & answers

215 views

Bio

Latest Posts



Arulkumaran Kumaraswamipillai

Mechanical Eng to freelance Java developer in 3 yrs. Contracting since 2003, and attended 150+ Java job interviews, and often got 4 - 7 job offers to choose from. It pays to prepare. So, published Java interview Q&A books via [Amazon.com](https://www.amazon.com) in 2005, and sold 35,000+ copies. Books are outdated and replaced with this subscription based site.



About [Arulkumaran Kumaraswamipillai](#)



Mechanical Eng to freelance Java developer in 3 yrs. Contracting since 2003, and attended 150+ Java job interviews, and often got 4 - 7 job offers to choose from. It pays to prepare. So, published Java interview Q&A books via [Amazon.com](https://www.amazon.com) in 2005, and sold 35,000+ copies. Books are outdated and replaced with this subscription based site.

◀ sed and awk to write shell scripts for Java developers

04: ♦ Top 10 most common Core Java beginner mistakes ▶

Posted in member-paid, OOP

Tags: TopX

Leave a Reply

Logged in as geethika. [Log out?](#)

Comment

Post Comment

Empowers you to open more doors, and fast-track

Technical Know Hows

☀ [Java generics in no time](#) ☀ [Top 6 tips to transforming your thinking from OOP to FP](#) ☀ [How does a HashMap internally work? What is a hashing function?](#)

☀ [10+ Java String class interview Q&As](#) ☀ [Java auto un/boxing benefits & caveats](#) ☀ [Top 11 slacknesses that can come back and bite you as an experienced Java developer or architect](#)

Non-Technical Know Hows

☀ [6 Aspects that can motivate you to fast-track your career & go places](#) ☀ [Are you reinventing yourself as a Java developer?](#) ☀ [8 tips to safeguard your Java career against offshoring](#) ☀ [My top 5 career mistakes](#)

Prepare to succeed

☀ [Turn readers of your Java CV go from “Blah blah” to “Wow”?](#) ☀ [How to prepare for Java job interviews?](#) ☀ [16 Technical Key Areas](#) ☀ [How to choose from multiple Java job offers?](#)

Select Category ▼

© Disclaimer

The contents in this Java-Success are copy righted. The author has the right to correct or enhance the current content without any prior notice.

These are general advice only, and one needs to take his/her own circumstances into consideration. The author will not be held liable for any damages caused or alleged to be caused either directly or indirectly by these materials and resources. Any trademarked names or labels used in this blog remain the property of their respective trademark owners. No guarantees are made regarding the accuracy or usefulness of content, though I do make an effort to be accurate. Links to external sites do not imply endorsement of the linked-to sites.