Explain abstraction, encapsulation, Inheritance, and polymorphism with the given code?

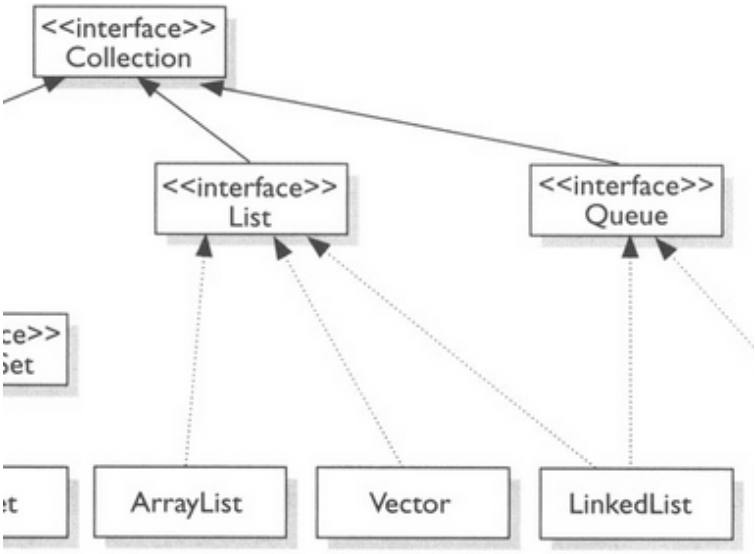
Posted on March 7, 2015 by Arulkumaran Kumaraswamipillai

Given code:

```
1 List<String> list = new ArrayList<>();
2 list.add("Java");
3 list.add("JEE");
```

A. Firstly, let's take **abstraction** and **encapsulation** as the difference is subtle. Abstraction is often not possible without encapsulation because if a class exposes its internal state, it can't change its inner workings. Encapsulation hides the underlying state. One of the ways to achieve abstraction is by

sub classing. The interface "**List**" is an abstraction for a sequence of items indexed by their position. The concrete examples of a list are "ArrayList<E>", "LinkedList<E>", CopyOnWriteArrayList<E>, etc.

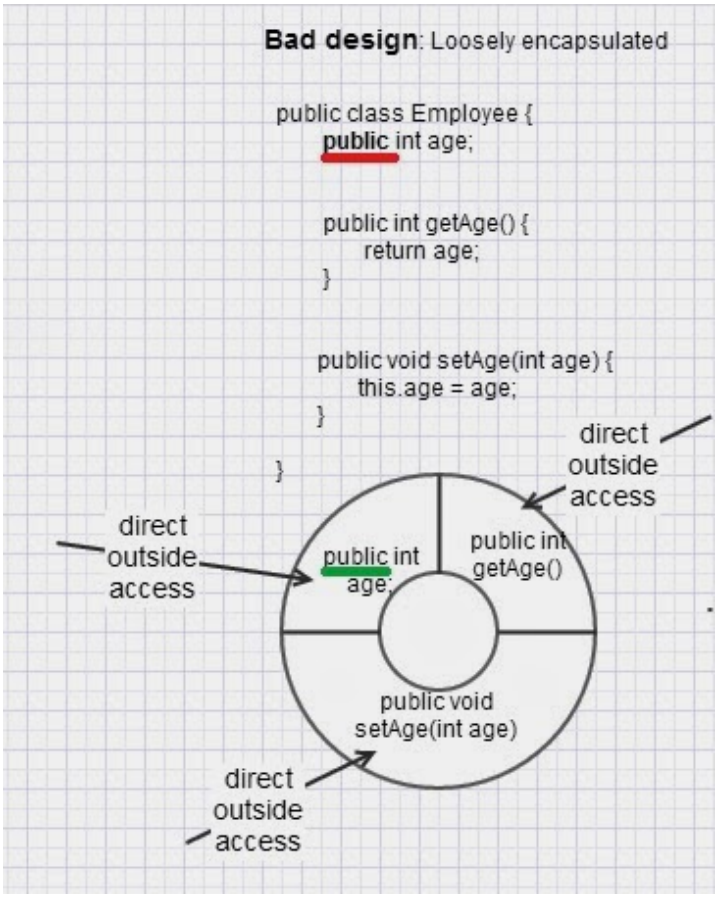


Both abstraction and encapsulation solve same **problem of complexity** in different dimensions. Encapsulation exposes only the required details of an object to the caller by forbidding access to certain members,

♦ Why favor com
08: ♦ Write code
Explain abstracti
How to create a
Top 5 OOPs tips
Top 6 tips to go a
Understanding C
What are good r
GC (2)
Generics (5)
FP (8)
IO (7)
Multithreading (12)
Algorithms (5)
Annotations (2)
Collection and Data
Differences Betwee
Event Driven Progr
Exceptions (2)
Java 7 (2)
Java 8 (24)
JVM (6)
Reactive Programm
Swing & AWT (2)
JEE Interview Q&A (3
Pressed for time? Jav
SQL, XML, UML, JSC
Hadoop & BigData Int
Java Architecture Inte
Scala Interview Q&As
Spring, Hibernate, & M
Testing & Profiling/Sa
Other Interview Q&A f
▶ Free Java Interviev

As a Java Architect

Java architecture & design concepts



Bad: No encapsulation

interview Q&As with diagrams | What should be a typical Java EE architecture?

Senior Java developers must have a good handle on

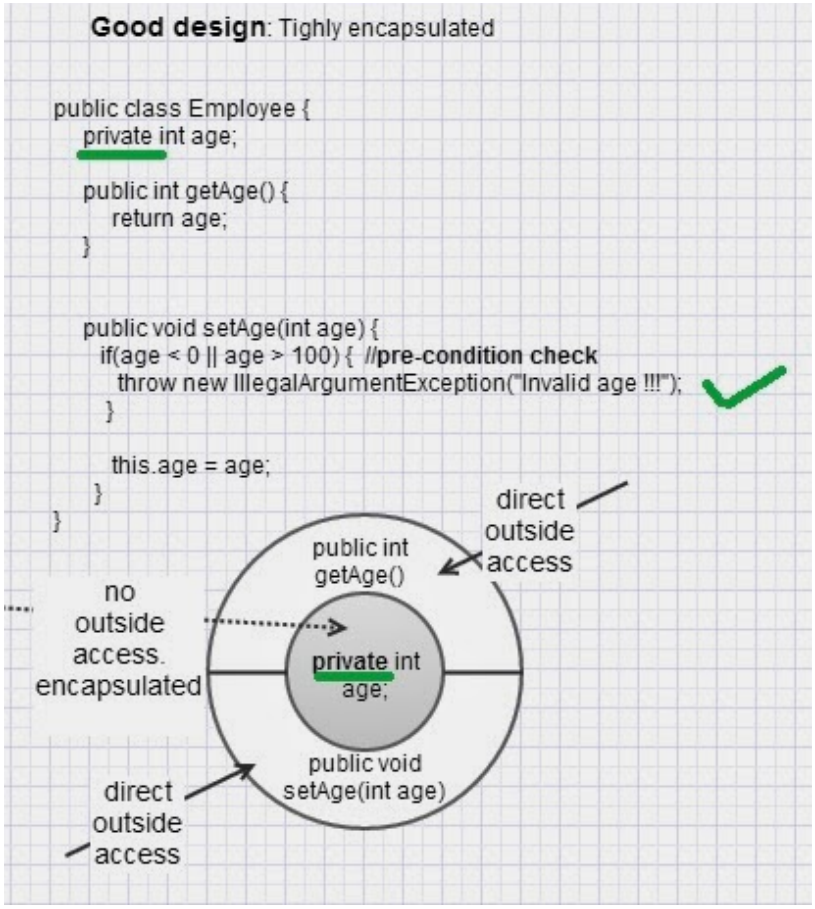
open all | close all

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

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 Tuto



Good: Encapsulated

Q. Why is it good to access state via public methods like getAge(), setAge(), etc as opposed to directly accessing the state “age”?
A.

1) The methods like setAge(int age) can perform precondition checks when setting the age by validating and throwing exception when age is -ve or very high positive like 120+.
2) Encapsulation promotes abstraction by only exposing public methods and the private methods are for internal implementation use. If a class exposes its internal state, it can’t change its inner workings in different implementations.

Abstraction not only hides the implementation details, but also provides a basis for your application to grow and change over a period of time. For example, if you abstract out the make and model of a vehicle as class attributes as opposed to as individual classes like Toyota, Camry, Corolla,

JEE Tutorials (19)
Scala Tutorials (1)
Spring & HIbernate Tu
Tools Tutorials (19)
Other Tutorials (45)

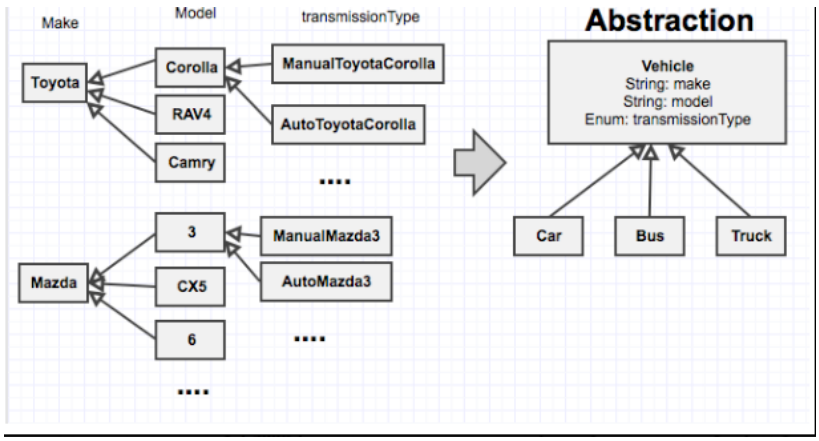
Preparing for Java written & coding tests

open all | close all
♦ Complete the given
Can you write code? (
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

ManualToyotaCorolla, AutoToyotaCorolla etc, you can easily incorporate new types of cars at runtime by creating a new car object with the relevant make and model as arguments as opposed to having to declare a new set of classes. So, in the below diagram, RHS is abstracted. So, abstraction is a more generic concept compared to encapsulation.



Java Abstraction

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. The “Vehicle” class on the RHS in the diagram captures the “make and model” as attributes to represent vehicles in an abstract manner. You don’t have to create new different classes for different vehicle make and model.

The ArrayList, LinkedList, etc are different implementations of the interface **List**. So, these implementations inherit the behavior of a **List**. Logic wise these implementations have the similar behavior like adding, removing, iterating, etc, but the implementation wise LinkedList and ArrayList are two different implementations of the List interface. LinkedList implements it with a doubly-linked list. ArrayList implements it with a dynamically resizing array. CopyOnWritrArrayList provides the functionality in a thread-safe manner without throwing the ConcurrentModificationException.

ANALOGY on interfaces Vs implementations: If you take a car, even though there are so many different makes and models of cars with **different implementations** under the bonnet like 4 cylinder engine, 6 cylinder engine, turbo charged engine, anti-lock braking vs normal breaking, and the list goes on. As a driver of a car the **interfaces** are very similar with a steering wheel, brake pedal, accelerator pedal, etc. So, if you know how to drive a car, you can drive different makes and models of a car.

The "add" method shown above is defined in the "**List**" interface. The ArrayList, LinkedList, etc provide their own implementation. In other words override the method. So, if you write something like

```
1 List<String> list = new ArrayList<>();
2 list.add("Java");
3 list.add("JEE");
```

The ArrayList class's implementation is executed. If you write something like

```
1 List<String> list = new LinkedList<>();
2 list.add("Java");
3 list.add("JEE");
```

The LinkList class's implementation is executed. The determination of which implementation's "**add**" method to be executed takes place at the runtime. Overriding lets you define the same operation in different ways for different object types. This is polymorphism. Which method is invoked depends on the type of object stored e.g. "ArrayList" or "LinkedList", and not on the reference type which is a "List" for both. If **overriding** were not possible, you can't have this OO concept known as **polymorphism**.

Q. Why is it a best practice to code to an interface?
A. In the example below, the "**readList**" method can be used with any implementations of a List. LinkedList, ArrayList, CopyOnWriteList, etc.

```
1  package test;
2
3  import java.util.ArrayList;
4  import java.util.LinkedList;
5  import java.util.List;
6
7  public class GoodExample {
8
9      public static void main(String[] args) {
10
11         GoodExample ex = new GoodExample();
12
13         //Working with an ArrayList implementati
14         List<String> list = new ArrayList<>();
15         list.add("Java");
16         list.add("JEE");
17         ex.readList(list); //prints Java, JEE
18
19         //Working with a LinkedList implementati
20         list = new LinkedList<>();
21         list.add("Java");
22         list.add("JEE");
23         ex.readList(list); //prints Java, JEE
24
25     }
26
27     //coded to interface to take any implementat
28     //loosely coupled
29     public void readList(List<String> list) {
30         for (String e : list) {
31             System.out.println(e);
32         }
33     }
34
35 }
```

The above example is based on the design practice of **coding to an interface**, which promotes loose coupling. You can easily switch implementations from a LinkedList to an ArrayList and vice versa. The following code tightly couples "**readList**" to a specific implementation, and requires specific implementations for each type like ArrayList, LinkedList, etc.

```
1  package test;
2
3  import java.util.ArrayList;
4  import java.util.LinkedList;
5
6  public class BadExample {
7
8      public static void main(String[] args) {
9
10         BadExample ex = new BadExample();
11
12         // Bad: Coding to an implementation
13         ArrayList<String> list = new ArrayList<>
14         list.add("Java");
15         list.add("JEE");
16         ex.readList(list); // prints Java, JEE
```

```
17
18          // Bad: Coding to an implementation
19          LinkedList<String> list2 = new LinkedLis
20          list2.add("Java");
21          list2.add("JEE");
22          ex.readList(list2); // prints Java, JEE
23
24      }
25
26      // Can only work with an ArrayList.
27      // tightly coupled to an ArrayList.
28      // To work with other types, different overl
29      public void readList(ArrayList<String> list)
30          for (String e : list) {
31              System.out.println(e);
32          }
33      }
34
35      // Can only work with a LinkedList
36      // tightly coupled to a LinkedList
37      // To work with other types, different overl
38      public void readList(LinkedList<String> list
39          for (String e : list) {
40              System.out.println(e);
41          }
42      }
43
44     //...more for other List types.
45
46 }
47
```

The Java OO concepts are explained in 5 Java OO interview questions and answers | Top 5 OO tips for Java developers | Why favor composition over inheritance? | Top 6 tips to go about writing loosely coupled Java applications. This will give you more confidence to tackle Java interview questions on OOP.

It is also worth revising on the design principles. Design principles interview questions & answers for Java developers | Understanding Open/Closed Principle (OCP) from the SOLID OO principles with a Java example

OOP interview questions are in the category of "**must know**". So, it pays to understand and apply OO concepts. A more thorough answer can go a long way, and compensate for any other minuses like not having hands-on experience with a "flavor of the month" framework/tool. It can also make you standout from your competition with similar or more skills/experience.

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 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 in 2005, and sold 35,000+ copies. Books are outdated and replaced with this subscription based site.

‹ Q6 – Q11 Swing interview questions and answers

08: ♦ Write code to add, subtract, multiply, and divide given numbers? ›

Posted in member-paid, OOP

Select Category

© Disclaimer

 ↑