Register | Login | Logout | Contact Us

Java-Success.com

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

search here … Go

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 › ♦ 30+ FAQ Java Object Oriented Programming (i.e. OOP) interview Q&As

♦ 30+ FAQ Java Object Oriented Programming (i.e. OOP) interview Q&As

Posted on August 11, 2014 by Arulkumaran Kumaraswamipillai — 2 Comments ↓

2 Like Share Tweet 2 G+1 15 Share

No Java Object Oriented Programming (i.e. OOP) & Java Functional Programming (FP) interview questions and answers success, then say OOPS!!!! to your interview success.

Java OOPs Interview Q&A and study Links:

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 (
  - Java Overview (4)
  - Data types (6)
  - constructors-methc
  - Reserved Key Wor
  - Classes (3)
  - Objects (8)
  - OOP (10)
    - ♥ Design princip
    - ♦ 30+ FAQ Java

Explain abstraction, encapsulation, Inheritance, and polymorphism with the given code? | Why favor composition over inheritance? | Top 5 OOPs tips for Java developers | Top 6 tips to go about writing loosely coupled Java applications | How to create a well designed Java application? | Design principles interview questions & answers for Java developers | Understanding Open/Closed Principle | Designing your classes & interfaces coding problems

Q1. Can you describe "**method overloading**" versus "**method overriding**"? Does it happen at compile time or runtime?

A1. **Method overloading**: Overloading deals with multiple methods in the same class with the same name but different method signatures. Both the below methods have the same method names but different method signatures, which mean the methods are overloaded.

```

public class {
      public static void evaluate(String param1);
      public static void evaluate(int param1); //m
}

```

This happens at compile-time. This is also called **compile-time polymorphism** because the compiler must decide how to select which method to run based on the data types of the arguments. If the compiler were to compile the statement:

```

evaluate("My Test Argument passed to param1");

```

it could see that the argument was a string literal, and generate byte code that called method #1.

Overloading lets you define the same operation in different ways for different data within the same class.

Method overriding: Overriding deals with two methods, one in the parent class and the other one in the child class and

As a Java Architect

Java architecture & design concepts

has the same name and signatures. Both the below methods have the same method names and the signatures but the method in the subclass “B” overrides the method in the superclass “A”.

```java

public class A {
   public int compute(int input) {
        return 3 * input;//method #3
   }
}

```

```java

public class B extends A {
   @Override
   public int compute(int input) {
        return 4 * input; //method #4
   }
}

```

This happens at runtime. This is also called **runtime polymorphism** because the compiler does not and cannot know which method to call. Instead, the JVM must make the determination while the code is running.

The method compute(..) in subclass “B” overrides the method compute(..) in super class “A”. If the compiler has to compile the following method,

```java

public int evaluate(A reference, int arg2)  {
     int result = reference.compute(arg2);
}

```

The compiler would not know whether the input argument ‘reference’ is of type “A” or type “B”. This must be determined during runtime whether to call method #3 or method #4 depending on what type of object (i.e. instance of Class A or instance of Class B) is assigned to input variable “reference”.

```java

A obj1 = new B( );
```

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

```
A obj2 = new A( );
evaluate(obj1);     // method #4 is invoked as st
evaluate(obj2);     // method #3 is invoked as st

```

Overriding lets you define the same operation in different ways for different object types. This is known as polymorphism. Which method is invoked depends on the type of object stored “A” or “B”, and NOT on the reference type which is “A” for both.

Q2. What is the difference among Overriding, Hiding, and Overloading in Java? How does overriding give polymorphism?

A2. **Overriding**, **Hiding**, **Overloading** and **Obscuring** are important core Java concepts and you will be quizzed on job interviews or written tests.

Q. What is overriding?

An instance method overrides all accessible instance methods with the same signature in super classes. If overriding were not possible, you can’t have the OO concept known as polymorphism. Let’s explain this with a simple example.

Q. Can you tell what will be the output of the following code snippet?

```

public class Base {
 public void f() {
  System.out.println("Base");
 }
}

```

```

public class DerivedOne extends Base{

 @Override //ensures signature is the same at com
 public void f() {
  System.out.println("Derived 1"); // overrides B
 }
}

```

Preparing for Java written & coding tests

How good are your...to go places?

```

public class DerivedTwo extends Base{

 @Override //ensures signature is the same at com
 public void f() {
  System.out.println("Derived 2"); //overrides Ba
 }
}

```

Test **polymorphism**:

```

public class PolymorphismTest {

 public static void main(String[] args) {

  Base b = new DerivedOne();
  b.f();//1

  b = new DerivedTwo();
  b.f();//2

  b = new Base();
  b.f();//3
 }
}

```

A. The output will be

Derived 1
Derived 2
Base

What is happening here?

1. Method overriding (i.e. polymorphic behavior) is occurring at run time to determine stored object type.
2. Even though the variable "b" is of type Base, which f( ) method gets called depends on what type of object is stored in that variable.

//1 the stored object is of type DerivedOne, hence prints "Derived 1".
//2 the stored object is of type DerivedTwo, hence prints "Derived 2".
//3 the stored object is of type Base, hence prints "Base"

This is polymorphism. Poly in greek means many and Morph means change. So polymorphism is the ability (in programming) to present the same interface for differing underlying forms (data types).

Q. What is hiding or shadowing?

The polymorphism and overriding are applicable only to non-static methods (i.e. instance methods). If you try to override a static method, it is known as hiding. For example,

```java

public class Base {

 int a = 5;

 public static void f() {
  System.out.println("Base");
 }
}

```

```java

public class DerivedOne extends Base{

 int a = 7; //hides new Base().a

 public static void f() {
  System.out.println("Derived 1"); // hides Base
 }
}

```

Test **hiding**:

```java

public class PolymorphismTest {

 public static void main(String[] args) {

  Base b = new DerivedOne();
  b.f(); // prints "Base"

  b = new Base();
  b.f();  //prints "Base"
 }
}

```

Why prints “Base” in both cases? Because, it only looks at what type the variable “b” is? in both cases of type Base. It does not care what type of object is stored. Same behavior for variables, and this is known as “Shadowing”. So, the overriding static methods and instance variables are “**shadowed**“, whereas the overriding instance methods gives polymorphic behavior.

```

public class PolymorphismTest {

 public static void main(String[] args) {

  Base b = new DerivedOne();
  System.out.println(b.a);        //prints 5

  b = new DerivedTwo();
  System.out.println(b.a); //prints 5
 }

}

```

prints 5. So, hiding and shadowing are Bad Practices, and should be avoided.

Q. What is method overloading? Unlike, overriding and hiding that happen when you have an inheritance hierarchy like Parent, Child classes, Overloading happens within the same class.

Methods in a class overload one another if they have the same name and different signatures. Unlike overriding, overloading happens at compile time. It knows at compile time based on the method arguments you pass.

```

public class Base {

 public  void f() {
  System.out.println("Base");
 }

 public  void f(String a) {
  System.out.println("Base"); //overloaded
 }

 public  void f(int b) {
  System.out.println("Base"); // overloaded
  }
}

```

Finally,

Q. What is obscuring? A variable obscures a type with the same name if both are in scope. Say, your instance and local variables have the same name.

```

public class Base {

 int a = 5;

 public  void f() {
  int a = 7; //obscuring instance variable a
  System.out.println(a);
 }
}

```

Test obscuring:

```

public class PolymorphismTest {

 public static void main(String[] args) {

  Base b = new DerivedOne();
  b.f(); //prints 7 -- local varible obscures in
 }

}

```

So, **overriding** and **overloading** are good, but avoid method hiding (i.e. having same name static methods in parent/child classes), variable shadowing (i.e. having similar variable names in parent/child classes), and variable obscuring (i.e. having same variable name for instance, static, and local variables).

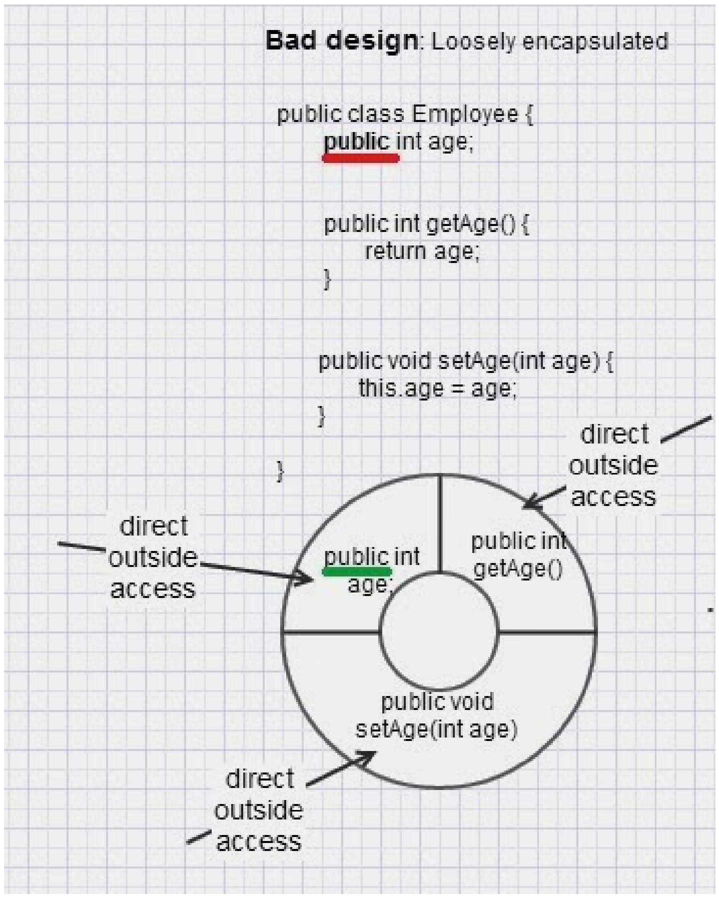
Q3. What do you achieve through good class and interface design?
A3.

1. Loosely coupled classes, objects, and components enable your application to easily grow and adapt to changes without being rigid or fragile.

2. Less complex and reusable code that increases maintainability, extendability and testability.

Q4. What are the 4 main concepts of OOP?
A4. Encapsulation, polymorphism, and inheritance are the 3 main concepts or pillars of an object oriented programming. Abstraction is another important concept that can be applied to both object oriented and non object oriented programming. [Remember: “**a pie**” abstraction, polymorphism, inheritance, and encapsulation.]

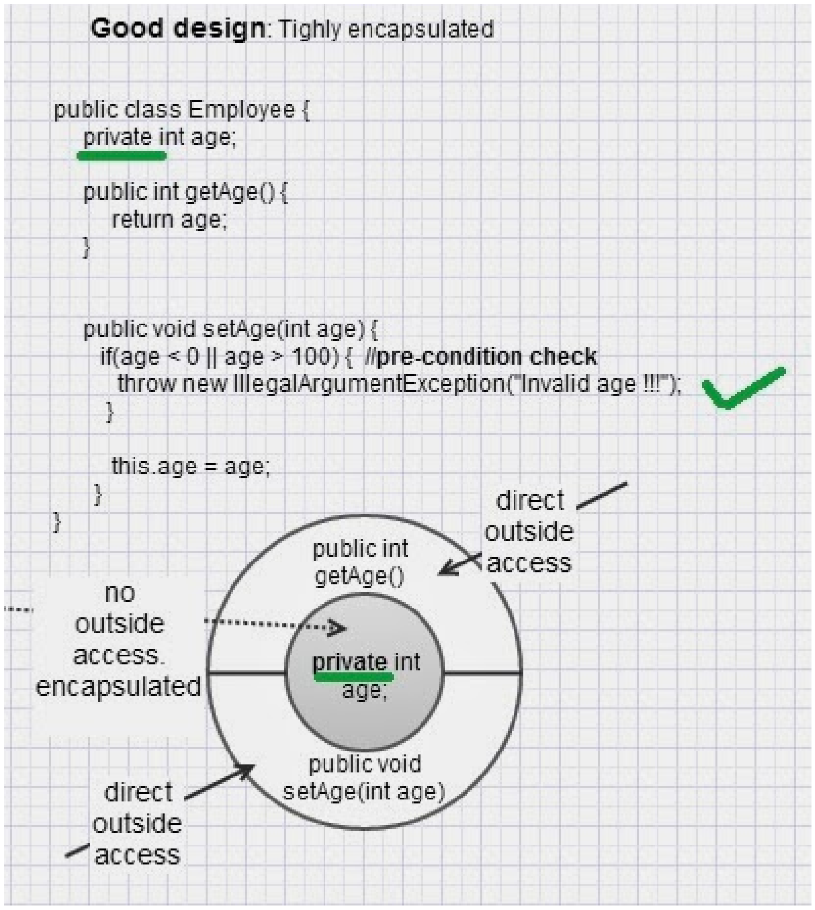
Q5. What problem(s) does abstraction and encapsulation solve?
A5. Both abstraction and encapsulation solve same problem of complexity in different dimensions. Encapsulation exposes only the required details of an object to the caller by forbidding access to certain members,



Bad design: Loosely encapsulated

```
public class Employee {
      public int age;

      public int getAge() {
            return age;
      }

      public void setAge(int age) {
            this.age = age;
      }

}
```

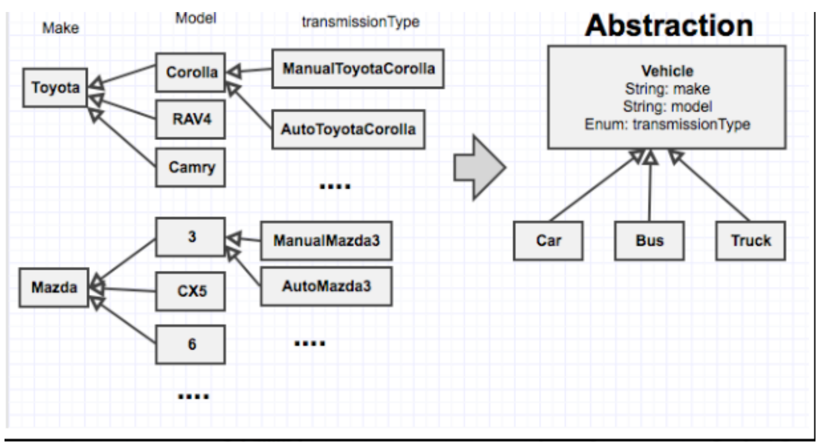
Bad: No encapsulation


Good design: Tighly encapsulated
direct outside access
public int getAge()
no outside access. encapsulated
private int age;
public void setAge(int age)
direct outside access

```
public class Employee {
    private int age;

    public int getAge() {
        return age;
    }


    public void setAge(int age) {
      if(age < 0 || age > 100) {  //pre-condition check
        throw new IllegalArgumentException("Invalid age !!!");
      }

        this.age = age;
    }
}
```

Good: Encapsulated

whereas an 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*, *ToyotaCamry*, *ToyotaCorolla*, 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.



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.

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.

Bad non abstracted example:

```java

package com.oo;

public class PartTimeEmployee extends Employee {

}

```

```java

package com.oo;

public class FullTimeEmployee extends Employee {

}


```

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

```

public class Employee {

 enum EmploymentType {PART_TIME, FULL_TIME, CASU

 private String name;
 private int age;
 private BigDecimal salary;
 private EmploymentType employmentType;

    // ... getters and setters for external acce

}


```

The above class is both properly abstracted and encapsulated.

Q6. What problem(s) does inheritance & composition solve?
A6. **Reusability**. How can logic be easily used in two places? In object-oriented language, there are four primary ways to accomplish this:

- **Copy and Paste** – Bad as it is hard to maintain. Any changes original logic need to be applied across all the pasted locations.
- **Inheritance** – Ok. Takes place at compile-time. So, can be fragile.
- **Composition** – Good, and favored over inheritance. Happens at runtime.
- **Mixins** – Good, but not supported in Java and can be abused and consequently increase complexity. The mixins are kind of composable abstract classes. They are used in a multi-inheritance context to add services to a class. Java 8 has a naive emulation of mixins with virtual extension or default methods.

There is a post dedicated to why favor composition over inheritance in the links shown below.

Important: In Java job interviews, you may be asked to define a simple OO application. You will also be probed "if you will be defining it as a class or attribute".

Java OOPs Interview Questions & Study Links:

Explain abstraction, encapsulation, Inheritance, and polymorphism with the given code? | Why favor composition over inheritance? | Top 5 OOPs tips for Java developers | Top 6 tips to go about writing loosely coupled Java applications | How to create a well designed Java application? | Design principles interview questions & answers for Java developers | Understanding Open/Closed Principle | Designing your classes & interfaces coding problems

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.

‹ ♥♦ 6 Java Modifiers every interviewer seems to like

♦ Why favor composition over inheritance? a must know interview question for Java developers ›

Posted in member-paid, OOP, OOP & FP Essentials

Tags: Core Java FAQs, Java/JEE FAQs, Novice FAQs

2 comments on "♦ 30+ FAQ Java Object Oriented Programming (i.e. OOP) interview Q&As"

Anshul Katta says:

November 29, 2015 at 1:05 am

the most asked questions are –

SOLID principle ,
diff between composition and aggregation,
what are non functional requirements in project
DI and IOC
why composition favored over inheritance

Reply

Arulkumaran Kumaraswamipillai says:

November 29, 2015 at 8:17 am

Very correct for OOP interview Q&As. All of the above questions are answered in detail. The non functional requirememts are included in the 16 tech key areas like security, transaction management, performance & scalability to mee the SLAs, QoS that includes monitoring, disaster recovery plans, auditing & archiving, etc.

With the advent of Java 8, FP interview questions will become popular as well.

Every interviewer has his/her favourite questions. Some love to grill on architecture/design & your handle on the 16 tech key areas with open-ended questions whilst others favor datastructures & coding. Web services & messaging interview Q&A are popular as well. Agile process & unit testing/TDD/BDD are hot topics at job interviews.

Reply

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