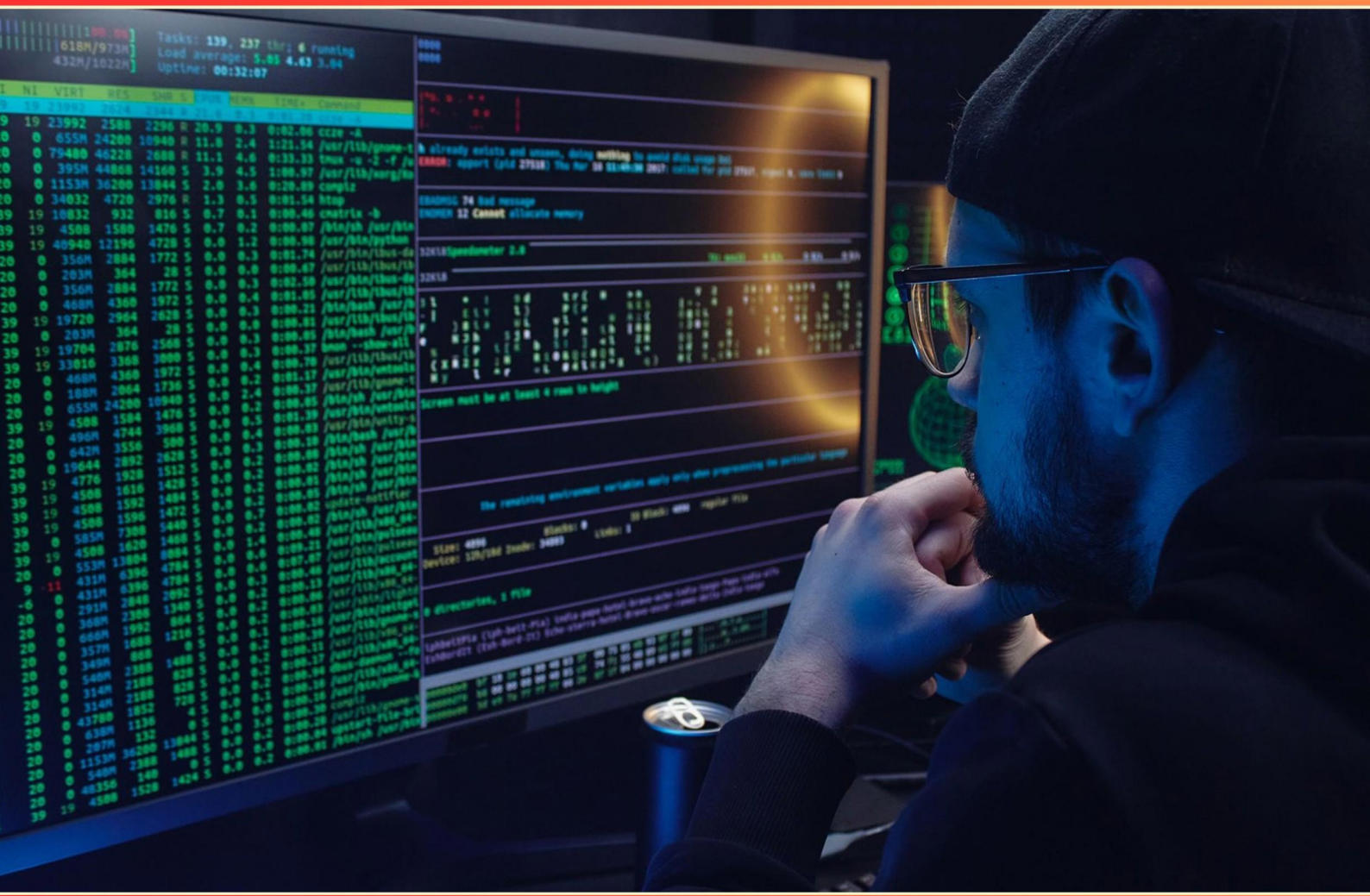
OBJECT ORIENTED PROGRAMMING (OOP) PRACTICE TEST (SAMPLE)

STUDY GUIDE



SAMPLE STUDY GUIDE WITH LIMITED QUESTIONS

VISIT US ONLINE FOR THE FULL VERSION STUDY GUIDE

https://objectorientedprogramming.examzify.com

Table of Contents

Introduction

Preparing for a certification exam can feel overwhelming, but with the right tools, it becomes an opportunity to build confidence, sharpen your skills, and move one step closer to your goals. At Examzify, we believe that effective exam preparation isn't just about memorization, it's about understanding the material, identifying knowledge gaps, and building the test-taking strategies that lead to success.

This guide was designed to help you do exactly that.

Whether you're preparing for a licensing exam, professional certification, or entry-level qualification, this book offers structured practice to reinforce key concepts. You'll find a wide range of multiple-choice questions, each followed by clear explanations to help you understand not just the right answer, but why it's correct.

The content in this guide is based on real-world exam objectives and aligned with the types of questions and topics commonly found on official tests. It's ideal for learners who want to:

- Practice answering questions under realistic conditions,
- Improve accuracy and speed,
- Review explanations to strengthen weak areas, and
- Approach the exam with greater confidence.

We recommend using this book not as a stand-alone study tool, but alongside other resources like flashcards, textbooks, or hands-on training. For best results, we recommend working through each question, reflecting on the explanation provided, and revisiting the topics that challenge you most.

Remember: successful test preparation isn't about getting every question right the first time, it's about learning from your mistakes and improving over time. Stay focused, trust the process, and know that every page you turn brings you closer to success.

Let's begin.

How to Use This Guide

This guide is designed to help you study more effectively and approach your exam with confidence. Whether you're reviewing for the first time or doing a final refresh, here's how to get the most out of your Examzify study guide:

1. Start with a Diagnostic Review

Skim through the questions to get a sense of what you know and what you need to focus on. Your goal is to identify knowledge gaps early.

2. Study in Short, Focused Sessions

Break your study time into manageable blocks (e.g. 30 – 45 minutes). Review a handful of questions, reflect on the explanations.

3. Learn from the Explanations

After answering a question, always read the explanation, even if you got it right. It reinforces key points, corrects misunderstandings, and teaches subtle distinctions between similar answers.

4. Track Your Progress

Use bookmarks or notes (if reading digitally) to mark difficult questions. Revisit these regularly and track improvements over time.

5. Simulate the Real Exam

Once you're comfortable, try taking a full set of questions without pausing. Set a timer and simulate test-day conditions to build confidence and time management skills.

6. Repeat and Review

Don't just study once, repetition builds retention. Re-attempt questions after a few days and revisit explanations to reinforce learning. Pair this guide with other Examzify tools like flashcards, and digital practice tests to strengthen your preparation across formats.

There's no single right way to study, but consistent, thoughtful effort always wins. Use this guide flexibly, adapt the tips above to fit your pace and learning style. You've got this!

Questions

1. Fan-in is defined as which of the following?

A. Incoming dependencies from outside to inside

B. Outgoing dependencies from inside to outside

C. Total number of interfaces

D. Number of public methods

2. Template Method pattern with a concrete scenario.

A. Subclasses define the algorithm steps; the base class provides the skeleton.

B. Subclasses override all behavior in the base class.

C. It relies on creating new objects to implement steps.

D. Base class defines the algorithm skeleton; subclasses override specific steps.

3. What should you do with 'fat' interfaces?

A. Split them into several small, specific interfaces

B. Merge them into a single large interface

C. Remove interfaces entirely and use concrete classes

D. Expose them through adapters

4. Generics allow a single class or method to operate on different types, avoiding what?

A. Runtime type checks

B. Memory leaks

C. Rigid, hard-coded type definitions — one class/method works for many types

D. Null checks

5. Encapsulation is defined as what?

A. Inheriting properties from a base class

B. Wrapping data and methods together as a single unit + data hiding

C. Exposing all fields publicly

D. Allowing multiple unrelated interfaces

6. What does the Singleton pattern guarantee?

A. No instance sharing across threads

B. Multiple identical instances allowed with no central access

C. Exactly one instance of the class in the JVM + one global access point

D. It uses private constructors but allows multiple instances

7. Which SOLID principle encourages splitting large interfaces into smaller, specific ones?

A. Single Responsibility Principle

B. Open/Closed Principle

C. Dependency Inversion Principle

D. Interface Segregation Principle

8. If two modules A and B depend on each other, what does that indicate in ADP terms?

A. There is no cycle; this is fine.

B. They must be merged into a single module.

C. This is only a problem if both modules are concrete.

D. A cycle exists between A and B.

9. Arrow token in a lambda expression?

A. ->

B. =>

C. ->=

D. <-

10. Is String a primitive type in Java?

A. No — String is a reference type

B. Yes

C. It depends on the context

D. String is both primitive and reference

Answers

1. A
2. D
3. D
4. C
5. B
6. C
7. D
8. D
9. A
10. A

Explanations

1. Fan-in is defined as which of the following?

A. Incoming dependencies from outside to inside

B. Outgoing dependencies from inside to outside

C. Total number of interfaces

D. Number of public methods

Fan-in describes how many external components rely on a given module; it counts incoming dependencies from outside to inside. This matches the idea that many other parts of the system depend on this module, so its change impact spans those dependents. If a module is used by many others, it has high fan-in, which can make changes riskier because breaking it could affect many callers. In contrast, fan-out refers to how many external components a module itself depends on (outgoing dependencies), not how many depend on it. The total number of interfaces measures how many entry points the module exposes, and the number of public methods reflects API size, not dependency direction. For example, if a module is used by three other modules, its fan-in is three; if it calls two other modules, that's its fan-out.

2. Template Method pattern with a concrete scenario.

A. Subclasses define the algorithm steps; the base class provides the skeleton.

B. Subclasses override all behavior in the base class.

C. It relies on creating new objects to implement steps.

D. Base class defines the algorithm skeleton; subclasses override specific steps.

The Template Method pattern centers on defining the overall algorithm in a base class as a fixed sequence (the template), while letting subclasses supply the concrete steps. The base class provides the skeleton of the algorithm—a template method that calls several steps in a specific order—and some of those steps are abstract (or have default behavior) so subclasses can customize them. This arrangement lets the common workflow stay consistent across different implementations, while the varying parts are encapsulated in the subclass overrides. That's why the best description is: the base class defines the algorithm skeleton and subclasses override specific steps. It preserves the overall structure while enabling customization where needed. The idea isn't to let subclasses dictate the entire sequence, nor to override every behavior, nor to rely on object creation to implement the steps—that would point to different patterns like Strategy, Factory, or Builder, not Template Method.

3. What should you do with 'fat' interfaces?

A. Split them into several small, specific interfaces

B. Merge them into a single large interface

C. Remove interfaces entirely and use concrete classes

D. Expose them through adapters

A fat interface packs many methods into one contract, which makes every client depend on more functionality than it actually needs. Exposing that interface through adapters is a practical way to give each client a lean, tailored surface while keeping the heavy interface intact behind the scenes. The adapter translates the smaller, client-focused calls into the appropriate calls on the fat interface, so clients don't have to learn or depend on all those methods. This approach preserves backward compatibility with existing code, localizes changes to the adapter, and makes it easier to evolve the underlying API over time. In contexts where you can't or don't want to refactor the interface itself, using adapters provides a clean separation between what a client uses and how the system implements the fat interface. Splitting into smaller interfaces is another valid strategy in many cases, but adapters specifically address the need to present a usable surface without forcing a broad redesign.

4. Generics allow a single class or method to operate on different types, avoiding what?

A. Runtime type checks

B. Memory leaks

C. Rigid, hard-coded type definitions — one class/method works for many types

D. Null checks

Generics let you write a single class or method that can work with many different types without tying the code to one specific type. The main win is avoiding rigid, hard-coded type definitions so one class or method can operate over many types. Without generics, you'd end up duplicating code for each type or resorting to using a general Object type with casts, which undermines type safety and makes maintenance harder. With generics, you declare a type parameter and apply it to different types, preserving type safety while keeping the implementation reusable. Some runtime checks or null handling may still occur in certain situations, but the essential benefit is the flexibility to support many types with a single, type-safe implementation.

5. Encapsulation is defined as what?

A. Inheriting properties from a base class

B. Wrapping data and methods together as a single unit + data hiding

C. Exposing all fields publicly

D. Allowing multiple unrelated interfaces

Encapsulation means bundling the object's data with the methods that operate on that data into a single unit, and restricting direct access to the internal state. In practice, you hide the internal fields (make them private) and expose a controlled interface (methods) to interact with the object. This protects invariants, reduces coupling, and lets you change the implementation later without breaking external code. That's why the description of wrapping data and methods together as a single unit plus data hiding is the best match. The other ideas describe different concepts: the first is inheritance, exposing all fields publicly breaks encapsulation, and interfaces alone don't define encapsulation.

6. What does the Singleton pattern guarantee?

A. No instance sharing across threads

B. Multiple identical instances allowed with no central access

C. Exactly one instance of the class in the JVM + one global access point

D. It uses private constructors but allows multiple instances

A Singleton guarantees there is exactly one instance of a class and a single global way to access that instance. The pattern achieves this by letting the class control its own creation—typically with a private constructor—and exposing a public static method (often called getInstance) that always returns the same instance. Because there is only one object in the runtime, all parts of the program share that same instance through that global access point, ensuring consistent state and coordinated access. This aligns with the idea of exactly one instance in the JVM (per class loader context) plus a single global entry point to obtain it. Other options misrepresent the pattern—for example, they imply multiple instances or no central access—so they don't fit the intended guarantee.

7. Which SOLID principle encourages splitting large interfaces into smaller, specific ones?

A. Single Responsibility Principle

B. Open/Closed Principle

C. Dependency Inversion Principle

D. Interface Segregation Principle

The main concept here is Interface Segregation Principle. This principle says that you shouldn't force clients to depend on methods they don't use, so it's better to have several small, focused interfaces than one large, bloated one. In practice, you break broad capabilities into separate interfaces like Printable, Scannable, and Faxiable, so a class only implements what it actually supports and clients depend on the specific subset they need. That keeps code decoupled and easier to modify or extend. The other SOLID principles address different concerns: Single Responsibility focuses on a class having one reason to change; Open/Closed on extending behavior without modifying existing code; Dependency Inversion on depending on abstractions rather than concrete implementations. So the choice that matches splitting large interfaces into smaller, specific ones is the Interface Segregation Principle.

8. If two modules A and B depend on each other, what does that indicate in ADP terms?

A. There is no cycle; this is fine.

B. They must be merged into a single module.

C. This is only a problem if both modules are concrete.

D. A cycle exists between A and B.

In ADP terms, dependencies are modeled as a directed graph and they should form an acyclic graph. If two modules depend on each other, you've created a cycle: you can go from A to B and back to A. That cycle violates the principle and signals tight coupling that makes maintenance, testing, and evolution harder. So the correct interpretation is that a cycle exists between A and B. To fix it, you'd break the cycle by introducing an abstraction or reassigning responsibilities so dependencies flow in one direction (for example, both modules depend on a common interface or new shared module). The idea that there's no cycle, or that you must merge them, or that the problem only matters if both are concrete, aren't accurate because the cycle is the core issue here.

9. Arrow token in a lambda expression?

A. ->

B. =>

C. ->=

D. <-

*In lambda expressions in C++, the arrow token "->" introduces the trailing return type. It sits after the parameter list and before the function body, giving an explicit return type for the lambda, for example: auto f = [] (int x) -> double { return x * 0.5; }. If you omit the return type, the compiler tries to deduce it from the return statements inside the lambda. You'd specify a trailing return type when the return type can't be reliably deduced or when you need to enforce a specific type, such as returning a type different from what the body would otherwise imply or when using complex expressions or templates. Note that in other languages, lambdas may use a different arrow token like "=>" (as in JavaScript or C#), so this particular syntax is specific to how C++ expresses lambda return types.*

10. Is String a primitive type in Java?

A. No — String is a reference type

B. Yes

C. It depends on the context

D. String is both primitive and reference

In Java, types fall into two broad categories: primitive types and reference types. Primitive types store raw values directly (like 42, true, 3.14). String, however, is a class in java.lang, so variables of type String hold references to String objects. The actual characters are stored inside those objects on the heap, and Strings are immutable. This distinction matters in practice: since String is a reference type, you typically compare content with equals() rather than using the equality operator (which checks if two references point to the same object). So String is not a primitive type; it's a reference type.

Next Steps

Congratulations on reaching the final section of this guide. You've taken a meaningful step toward passing your certification exam and advancing your career.

As you continue preparing, remember that consistent practice, review, and self-reflection are key to success. Make time to revisit difficult topics, simulate exam conditions, and track your progress along the way.

If you need help, have suggestions, or want to share feedback, we'd love to hear from you. Reach out to our team at hello@examzify.com.

Or visit your dedicated course page for more study tools and resources:

https://objectorientedprogramming.examzify.com

We wish you the very best on your exam journey. You've got this!