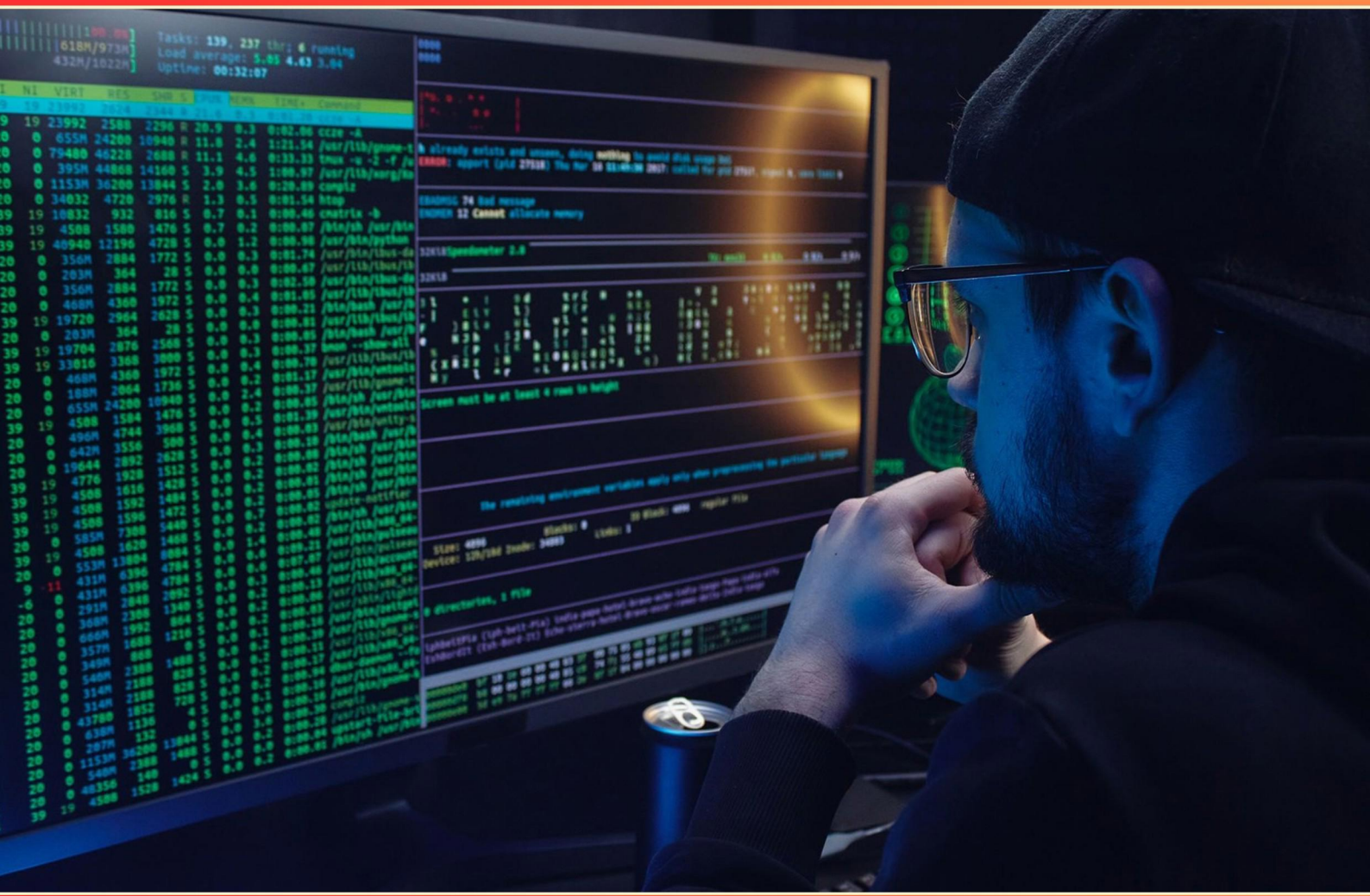
OBJECT ORIENTED PROGRAMMING (OOP) PRACTICE TEST (SAMPLE)

STUDY GUIDE



SAMPLE STUDY GUIDE WITH LIMITED QUESTIONS

VISIT US ONLINE FOR THE FULL VERSION STUDY GUIDE

https://objectorientedprogramming.examzify.com

Copyright © 2026 by Examzify - A Kaluba Technologies Inc. product.

ALL RIGHTS RESERVED.

No part of this book may be reproduced or transferred in any form or by any means, graphic, electronic, or mechanical, including photocopying, recording, web distribution, taping, or by any information storage retrieval system, without the written permission of the author.

Notice: Examzify makes every reasonable effort to obtain accurate, complete, and timely information about this product from reliable sources.

SAMPLE

Table of Contents

Copyright 1

Table of Contents 2

Introduction 3

How to Use This Guide 4

Questions 5

Answers 8

Explanations 10

Next Steps 15

SAMPLE

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.

SAMPLE

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!

SAMPLE

Questions

SAMPLE

1. What are getters and setters?

A. Directly exposing private fields

B. Public methods providing indirect read/write access to private fields

C. Static accessors

D. Constructors for initialization

2. Which block always executes after try-catch?

A. Catch

B. Else

C. Cleanup

D. Finally

3. Why are unit tests and test doubles important in object-oriented development?

A. They validate behavior in isolation, enable fast feedback, and mocks isolate dependencies for reliable tests.

B. They are optional in OO development.

C. They replace production tests entirely.

D. They only increase code complexity.

4. Which tool technique helps verify acyclic dependencies in a codebase?

A. Code formatting check.

B. Unit tests that execute all code paths.

C. Static analysis tool that detects cycles.

D. Performance profiling.

5. Car has a Driver — what type of OOP relationship is this?

A. Aggregation — both objects survive independently

B. Composition — dependent lifecycles

C. Inheritance — Driver extends Car

D. Association — loose relationship

SAMPLE

6. Which statement best describes dependency injection?

A. Dependencies are resolved at compile time and fixed.

B. Dependencies are provided from outside the object, reducing coupling and enabling mocks in tests.

C. Dependencies are created inside the class with new; this improves encapsulation.

D. The pattern only applies to singleton objects.

7. Which option represents a form of data hiding related to encapsulation?

A. Direct access to private fields is allowed

B. Data hiding

C. Exposing global variables

D. Polymorphism

8. How does the Decorator pattern differ from inheritance?

A. Decorator replaces the component with a subclass.

B. Decorator changes the interface to a new one.

C. Decorator ensures a single instance.

D. Decorator adds responsibilities by wrapping objects at runtime; inheritance adds behavior at compile time; decorators can stack.

9. What does immutability mean in object design, and what are its benefits?

A. Mutable objects cannot be changed after creation.

B. Immutable objects cannot change after creation; benefits include simplicity, thread-safety, and easier reasoning.

C. Immutability means objects cannot have methods.

D. Immutability prevents any state in the program.

10. What does the 'I' in SOLID stand for?

A. Interface Design Principle

B. Inversion of Dependencies Principle

C. Interface Segregation Principle

D. Independent Modules Principle

SAMPLE

Answers

SAMPLE

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

SAMPLE

Explanations

SAMPLE

1. What are getters and setters?

A. Directly exposing private fields

B. Public methods providing indirect read/write access to private fields

C. Static accessors

D. Constructors for initialization

Getters and setters are public methods that provide indirect read/write access to private fields. They keep the internal data hidden, so code outside the class can observe or change values only through controlled channels. A getter returns the current value of a private field, while a setter updates it, often with validation or other logic to maintain invariants. This encapsulation protects the object's state and allows the class to enforce rules when data changes. Exposing private fields directly would bypass these safeguards, and while static accessors or constructors have their own roles, they don't serve the ongoing, instance-level access pattern that getters and setters provide.

2. Which block always executes after try-catch?

A. Catch

B. Else

C. Cleanup

D. Finally

The key idea is guaranteed cleanup after a block of code that might raise an error. The finally block is designed to run regardless of whether an exception occurred or not, making it the reliable place for cleanup actions like releasing resources, closing files, or restoring state. It executes after the try block finishes and, if an exception was caught, after the corresponding catch block as well. Catch only runs when an error happens, so it doesn't always execute. Else runs only when no exception occurred, so it isn't guaranteed either. That universal execution is what makes the finally block the correct choice.

3. Why are unit tests and test doubles important in object-oriented development?

A. They validate behavior in isolation, enable fast feedback, and mocks isolate dependencies for reliable tests.

B. They are optional in OO development.

C. They replace production tests entirely.

D. They only increase code complexity.

In object-oriented development, unit tests focus on verifying the behavior of a single unit (often a class or method) in isolation from the rest of the system. Test doubles—such as mocks, stubs, fakes, and spies—let you replace real collaborators with controllable, predictable substitutes. This combination is powerful because it lets you: - Check that a unit behaves correctly on its own, without interference from other parts of the system. - See fast, deterministic feedback since tests don't depend on real, possibly slow or flaky components. - Verify not only the results but also how a unit interacts with its dependencies, such as whether it calls a method with the right arguments or in the right order. These practices are especially valuable in OO design, where objects collaborate to achieve behavior. By isolating a unit and controlling its collaborators, you can pinpoint issues, encourage cleaner interfaces, and support safe refactoring. They aren't optional in OO development, they don't replace production tests, and they don't simply add unnecessary complexity. Used well, unit tests with test doubles improve reliability and design quality.

SAMPLE

4. Which tool technique helps verify acyclic dependencies in a codebase?

A. Code formatting check.

B. Unit tests that execute all code paths.

C. Static analysis tool that detects cycles.

D. Performance profiling.

Verifying that dependencies form a cycle-free structure is best done with a static analysis approach that inspects the codebase and builds the dependency graph. A static analysis tool can examine how modules or components depend on each other and detect cycles in that graph without running the program. When it finds a cycle, it flags it so you can refactor to keep dependencies acyclic, which helps reduce coupling and improve maintainability. Code formatting checks focus on style and syntax, not how parts depend on one another. Unit tests verify that code behaves correctly across paths, but they don't specifically ensure there are no circular dependencies unless those tests happen to exercise that import path, which is incidental. Performance profiling measures runtime efficiency, not the structure of dependencies. So, the most direct and reliable technique for confirming acyclic dependencies is static analysis that detects cycles.

5. Car has a Driver — what type of OOP relationship is this?

A. Aggregation — both objects survive independently

B. Composition — dependent lifecycles

C. Inheritance — Driver extends Car

D. Association — loose relationship

This question tests how lifecycles and ownership define how objects relate. The Car having a Driver is a has-a relationship, but the Driver's lifecycle isn't tied to the Car's. A Driver exists independently and can continue to exist even if a particular Car is removed, and the Driver could drive different cars over time. That weaker, lifecycle-independent connection is aggregation: the Car holds a reference to the Driver without owning its lifetime. In contrast, composition would imply the Driver's lifecycle is bound to the Car—creating or destroying the Car would also create or destroy the Driver, which isn't implied here. Inheritance would suggest Car is a kind of Driver, which isn't accurate. A plain association describes a link between objects without emphasizing lifecycles, but aggregation is the more precise way to express a has-a relationship where both objects survive independently.

6. Which statement best describes dependency injection?

A. Dependencies are resolved at compile time and fixed.

B. Dependencies are provided from outside the object, reducing coupling and enabling mocks in tests.

C. Dependencies are created inside the class with new; this improves encapsulation.

D. The pattern only applies to singleton objects.

Dependency injection is a pattern where an object's collaborators are supplied from outside, rather than created by the object itself. This reduces coupling because the class depends on abstractions (interfaces) rather than concrete implementations, so the same class can work with different dependencies without changing its code. It also makes testing easier, since you can inject mock or fake dependencies during tests to verify behavior without relying on real services. This approach best describes why it's useful: dependencies provided from outside let you swap implementations, configuration, or test doubles without touching the dependent class. In contrast, having dependencies resolved at compile time and fixed means the class hard-codes specific implementations, which tightens coupling and harms flexibility and testability. Creating dependencies inside the class with new further increases coupling because the class decides exactly which collaborators to instantiate, making it harder to substitute them in tests or in different environments. And dependency injection is a broad pattern that applies to regular objects, not only to singletons.

SAMPLE

7. Which option represents a form of data hiding related to encapsulation?

A. Direct access to private fields is allowed

B. Data hiding

C. Exposing global variables

D. Polymorphism

Data hiding is a form of encapsulation where you restrict direct access to an object's internal data and expose a safe interface for interaction. By making fields private and providing public methods (getters/setters or other operations), you protect the internal state from external misuse and allow the implementation to change without breaking external code. This is exactly what encapsulation aims to achieve: control over how data is accessed and modified. Directly allowing access to private fields violates data hiding and defeats encapsulation, so it isn't the right choice. Exposing global variables or other forms of external state breaks encapsulation as well, and polymorphism deals with how objects of different types can be treated uniformly rather than with hiding data.

8. How does the Decorator pattern differ from inheritance?

A. Decorator replaces the component with a subclass.

B. Decorator changes the interface to a new one.

C. Decorator ensures a single instance.

D. Decorator adds responsibilities by wrapping objects at runtime; inheritance adds behavior at compile time; decorators can stack.

The key idea is extending behavior by wrapping objects at runtime rather than through static inheritance. A decorator holds a reference to a component and implements the same interface, so it can add its own behavior before or after delegating to the wrapped component. This setup allows you to stack multiple decorators, combining features in flexible ways without changing existing classes. Inheritance, by contrast, adds behavior at compile time through subclassing, which fixes which features are present and often requires creating many subclasses to cover all combinations. Decorators preserve the original interface, so clients treat decorated and undecorated objects uniformly, and they enable dynamic composition of features.

9. What does immutability mean in object design, and what are its benefits?

A. Mutable objects cannot be changed after creation.

B. Immutable objects cannot change after creation; benefits include simplicity, thread-safety, and easier reasoning.

C. Immutability means objects cannot have methods.

D. Immutability prevents any state in the program.

Immutability means an object's state is fixed at creation and cannot be changed afterward. If you need a different state, you create a new object with the new values. This doesn't stop an object from having methods or from existing in a program with other mutable state; it just means that particular object won't mutate its own fields. The benefits are clear: it makes reasoning about code easier because once you have a reference to an immutable object, you know its state won't change unexpectedly. It also helps with thread-safety: multiple threads can safely access the same instance without worrying about concurrent mutations, often reducing or eliminating the need for synchronization. In practice, this leads to simpler, more predictable code and can enable optimizations like caching or memoization since the data stays the same. For example, a Point class with coordinates x and y can be immutable by storing these values once and not providing any setters. To represent a moved point, you create a new Point with the new coordinates instead of altering the original one. The other statements miss the essence: immutability isn't about forbidding methods, nor about preventing all state in the program, and mutable objects can still exist alongside immutable ones.

SAMPLE

10. What does the 'I' in SOLID stand for?

A. Interface Design Principle

B. Inversion of Dependencies Principle

C. Interface Segregation Principle

D. Independent Modules Principle

The I in SOLID stands for Interface Segregation Principle. It says you should design small, client-specific interfaces so that nothing forces a client to depend on methods it doesn't use. By splitting fat interfaces into focused ones, you keep implementations lean and decoupled from unnecessary features, making changes safer and evolution easier. A practical example is having separate interfaces for printing and scanning instead of one large interface that includes both capabilities; classes that only print won't be tied to scanning methods. The other options don't match the standard terminology here and describe different ideas, so only Interface Segregation captures the need to keep interfaces narrow and task-specific.

SAMPLE

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!

SAMPLE