Assessments

Chapter 1, Getting Started with Kotlin

Question 1

What's the difference between `var` and `val` in Kotlin?

Answer

The `val` keyword declares an immutable value that cannot be modified once assigned. The `var` keyword declares a mutable variable that can be assigned multiple times.

Question 2

How do you extend a class in Kotlin?

Answer

To extend a class, you can specify its name and constructor after a semi-colon. If it's a regular class, it must be declared `open` for your code to be able to extend it.

Question 3

How do you add functionality to a `final` class?

Answer

To add functionality to a class that we cannot inherit from, we can use an extension function. The extension function will have access only to the class itself and to its public fields and functions.

Chapter 2, Working with Creational Patterns

Question 1

Name two uses for the **`object`** keyword we learned about in this chapter.

Answer

The **`object`** keyword is used to declare a singleton if it's used in a global scope or as a collection of static methods if it's used in a conjunction with the **`companion`** keyword inside a class.

Question 2

What is the **`apply()`** function used for?

Answer

The **`apply()`** function is used when we want to change the state of an object and then return it immediately.

Question 3

Provide one example of a static factory method that we discussed in this chapter.

Answer

The JVM **`valueOf()`** method on the **`Long`** objects is a static factory method.

Chapter 3, Understanding Structural Patterns

Question 1

What differences are there between the implementations of the Decorator and Proxy design patterns?

Answer

The Decorator and Proxy design patterns could be implemented in the same manner. The only difference is in their intent – the Decorator design pattern adds functionality to an object, while the Proxy design pattern may change an object's functionality.

Question 2

What is the main goal of the Flyweight design pattern?

Answer

The goal of the Flyweight design pattern is to conserve memory by reusing the same immutable state across multiple lightweight objects.

Question 3

What is the difference between the Facade and Adapter design patterns?

Answer

The Facade design pattern creates a new interface to simplify working with complex code, while the Adapter design pattern allows one interface to substitute another interface.

Chapter 4, Getting Familiar with Behavioral Patterns

Question 1

What's the difference between Mediator and Observer design patterns?

Answer

Both serve a similar purpose. Mediator introduces tight coupling between components that may serve different purposes, while Observer operates on similar components that are loosely coupled.

Question 2

What is a **Domain-Specific Language** (**DSL**)?

Answer

A DSL is a language that focuses on solving problems in a specific domain. This is different from a general-purpose language, such as Kotlin, that can be applied to different domains. Kotlin encourages developers to create DSLs for their needs.

Question 3

What are the benefits of using a sealed class or interface?

Answer

Since all types of a sealed class are known at compile time, Kotlin compiler can verify that the **`when`** statement covers all cases or, in other words, is exhaustive.

Chapter 5, Introducing Functional Programming

Question 1

What are higher order functions?

Answer

A higher order function is any function that either receives another function as input or returns a function as output.

Question 2

What is the **`tailrec`** keyword in Kotlin?

Answer

The purpose of the **`tailrec`** keyword is to allow the Kotlin compiler to optimize tail recursion and avoid stack overflow.

Question 3

What are pure functions?

Answer

Pure functions are functions that don't have any side effects, such as I/O.

Chapter 6, Threads and Coroutines

Question 1

What are the different ways to start a coroutine in Kotlin?

Answer

A coroutine in Kotlin could be started with either the **`launch()`** or **`async()`** functions. The difference is that **`async()`** also returns a value, while **`launch()`** doesn't.

Question 2

With structured concurrency, if one of the coroutines fails, all the siblings will be canceled as well. How can we prevent that behavior?

Answer

We can prevent canceling siblings by using **`supervisorScope`** instead of **`coroutineScope`**.

Question 3

What is the purpose of the **`yield()`** function?

Answer

The **`yield()`** function returns a value and suspends the coroutine until it has been resumed.

Chapter 7, Controlling the Data Flow

Question 1

What is the difference between higher order functions on collections and on concurrent data structures?

Answer

Higher order functions on collections will process the entire collection, creating a copy of it, before proceeding to the next step. Higher order functions on concurrent data structures are reactive, processing one element after the other.

Question 2

What is the difference between cold and hot streams of data?

Answer

A cold stream repeats itself for each new consumer, while the hot stream will only send the available data to the new consumer from the time of subscription.

Question 3

When should a conflated channel/flow be used?

Answer

A conflated flow can be used in situations when the consumer is slower than the producer and some of the messages could be dropped, leaving only the most recent message for consumption.

Chapter 8, Designing for Concurrency

Question 1

What does it mean when we say that the **`select`** expression in Kotlin is biased?

Answer

A biased **`select`** expression means that in case of a *draw* between two channels, the first channel listed in the **`select`** expression will always be

picked.

Question 2

When should you use a mutex instead of a channel?

Answer

Mutexes are used to protect a resource shared between multiple coroutines. Channels are used to pass data between coroutines.

Question 3

Which of the concurrent design patterns could help you implement **MapReduce** or a **divide and conquer** algorithm efficiently?

Answer

For divide and conquer algorithms, the fan-out design pattern could be used to split the data and a fan-in design pattern could be used to combine the results.

Chapter 9, Idioms and Anti-Patterns

Question 1

What is the alternative to Java's `try`-with-resources in Kotlin?

Answer

In Kotlin, the `use()` function works on the `Closeable` interface to make sure that resources are released after use.

Question 2

What are the different options for handling nulls in Kotlin?

Answer

There are multiple options to handle nulls: the Elvis operator, smart casts, and the `let` and `run` scope functions can help with that.

Question 3

Which problem can be solved by reified generics?

Answer

On JVM, types are erased at runtime. By inlining the generic function body into the call site, it allows preservation of the actual types used by the compiler.

Chapter 10, Concurrent Microservices with Ktor

Question 1

How are the Ktor applications structured and what are their benefits?

Answer

Ktor applications are divided into modules, each module being an extension function on the **`Application`** object. Modularizing our application allows us to test different aspects of it separately.

Question 2

What are plugins in Ktor and what are they used for?

Answer

Plugins are a way Ktor addresses cross-cutting concerns. They are used for serializing and deserializing requests and responses, and setting headers, and even routing itself is a plugin.

Question 3

What is the main problem that the **`Exposed`** library solves?

Answer

The **`Exposed`** library provides a higher-level API for working with databases.

Chapter 11, Reactive Microservices with Vert.x

Question 1

What's a verticle in Vert.x?

Answer

A verticle is a lightweight actor that allows us to separate our business logic into small reactive units.

Question 2

What's the goal of the Event Bus in Vert.x?

Answer

The Event Bus allows verticles to communicate with each other indirectly by sending and consuming messages.

Question 3

Why shouldn't we block the event loop?

Answer

The event loop uses a limited number of threads to process many requests concurrently. If even one of the threads is blocked, it reduces the performance of a Vert.x app.