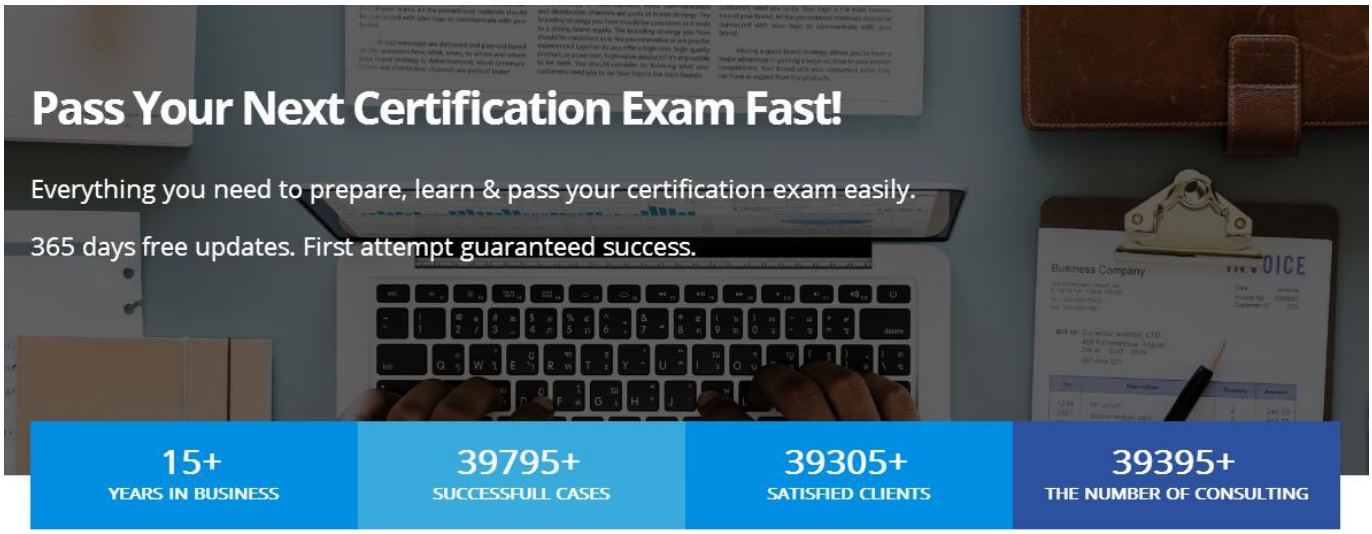
PracticeVCE



TRY BEFORE YOU BUY

Download a free sample of any of our exam questions and answers

- 24/7 customer support, Secure shopping site
- Free One year updates to match real exam scenarios
- If you failed your exam after buying our products we will refund the full amount back to you.

365 Days Free Updates

Free update is available within 365 days after your purchase. After 365 days, you will get 50% discounts for updating.

Security & Privacy

We respect customer privacy. We use McAfee's security service to provide you with utmost security for your personal information & peace of mind.

Instant Download

After Payment, our system will send you the products you purchase in mailbox in a minute after payment. If not received within 2 hours, please contact us.

Money Back Guarantee

Full refund if you fail the corresponding exam in 60 days after purchasing. And Free get any another product.

http://www.practicevce.com

Professional Study Tool and Reliable Exam Practice Material

Exam : **PDI**

Title : Platform Developer I (PDI)

Vendor : Salesforce

Version : DEMO

NO.1 Which two statements are true about using the @testSetup annotation in an Apex test class?
Choose 2 answers
A. Records created in the test setup method cannot be updated in individual test methods.
B. In a test setup method, test data is inserted once and made available for all test methods In the test class.
C. A method defined with the @testSetup annotation executes once for each test method in the test class and counts towards system limits.
D. The @testSetup annotation is not supported when the @isTest(SeeAllData=True) annotation is used .
Answer: B D
Explanation:
B:@testSetupinserts data once and shares it across all test methods in the class, saving system resources.
D:@isTest(SeeAllData=true)is incompatible with@testSetupdue to their differing purposes for managing test data.
Reference:@testSetup Annotation
Incorrect Options:
A:Records can be updated in individual test methods.
C:@testSetupdoes not execute repeatedly; it runs once per class.

NO.2 The orderHelper class is a utility class that contains business logic for processing orders.
Consider the following code snippet:
Public class without sharing orderHelper {// code implementation
}
A developer needs to create a constant named delivery_multiplier with a value of 4.15. The value of the constant should not change at any time in the code.
How should the developer declare the delivery multiplier constant to meet the business objectives?
A. static decimal DELIVERY_MULTIPLIER = 4.15;
B. constant decimal DELIVERY_MULTIPLIER = 4.15;
C. static final decimal DELIVERY_MULTIPLIER = 4.15;
D. decimal DELIVERY_MULTIPLIER = 4.15;
Answer: C
Explanation:
Declaring the constant asstaticensures it is shared across all instances of the class.
Declaring it asfinalensures its value cannot be modified after initialization.
Static and Final Variables in Apex

NO.3 Universal Containers wants to assess the advantages of declarative development versus programmatic customization for specific use cases in its Salesforce implementation.
What are two characteristics of declarative development over programmatic customization?
Choose 2 answers
A. Declarative development does not require Apex test classes.
B. Declarative development has higher design limits and query limits,
C. Declarative development can be done using the Setup menu.

D. Declarative code logic does not require maintenance or review.

Answer: A C

Explanation:

A: Declarative development does not require Apex test classes since it leverages built-in Salesforce features such as flows, process builders, and validation rules.

C: Declarative development can be done using the Setup menu, which includes point-and-click tools like page layouts, workflows, and Lightning App Builder.

Why not other options?

B: Declarative development has strict limits (e.g., process builders and flows have governor limits that are lower than programmatic approaches like Apex).

D: Declarative logic still requires maintenance, especially as business needs evolve.

Declarative vs Programmatic Development

NO.4 A developer must troubleshoot to pinpoint the causes of performance issues when a custom page loads in their org.

Which tool should the developer use to troubleshoot query performance?

A. Setup Menu

B. Visual Studio Code IDE

C. AppExchange

D. Developer Console

Answer: D

Explanation:

Why Developer Console?

TheDeveloper Consoleprovides tools to troubleshoot query performance, including the Query Plan tool.

The Query Plan tool evaluates SOQL query performance and provides insight into query costs and optimization.

Why Not Other Options?

A: Setup Menu: Used for administrative tasks, not performance troubleshooting.

B: Visual Studio Code IDE: Useful for development, but does not offer query performance tools.

C: AppExchange: Provides apps but does not directly help with query troubleshooting.

References:Query Plan Tool: https://developer.salesforce.com/docs/atlas.en-us.220.0.salesforce_console.meta/salesforce_console/salesforce_console_query_plan.htm

NO.5 A developer at AW Computing is tasked to create the supporting test class for a programmatic customization that leverages records stored within the custom object, Pricing Structure c. AW Computing has a complex pricing structure for each item on the store, spanning more than 500 records.

Which two approaches can the developer use to ensure Pricing _Structure__c records are available when the test class is executed?

Choose 2 answers

A. Use a Test Date Factory class.

B. Use the @TsTest (SeeAllData=true) annotation.

C. Use the Test. leadteat{) method.

D. Use without sharing on the class declaration.

Answer: A B
Explanation:
Option A: Using aTest Data Factoryclass is a recommended approach to create test records programmatically.
This ensures that the test is independent of existing data and can be executed in isolation.
Option B: The@IsTest(SeeAllData=true)annotation allows the test to access existing records in the database, includingPricing_Structure__c. However, this is less recommended as it creates a dependency on existing org data.
Not Suitable:
Option C:Test.loadDatais used to load records from a static resource (CSV) but doesn't suit the requirement directly.
Option D: Addingwithout sharingto a class impacts sharing rules but does not create or ensure the existence of test data.
Testing Best Practices

NO.6 An org has an existing flow that edits an Opportunity with an Update Records element. A developer must update the flow to also create a Contact and store the created Contact's ID on the Opportunity.
Which update must the developer make in the flow?
A. Add a new Update Records element.
B. Add a new Roll Back Records element.
C. Add a new Create Records element.
D. Add a new Get Records element.
Answer: C
Explanation:
Why Create Records Element?
TheCreate Recordselement adds the functionality to create a Contact record.
The Contact's ID can then be stored on the Opportunity using a variable or field update.
Why Not Other Options?
A:Update Recordsis for updating existing records, not creating new ones.
B:Roll Back Recordsis used for rolling back transactions, not for creating records.
D:Get Recordsretrieves records but does not create them.
References:Flow Builder:https://help.salesforce.com/s/articleView?id=sf.flow_build.htm

NO.7 For which three items can a trace flag be configured?
Choose 3 answers
A. Apex Class
B. Flow
C. User
D. Visualforce
E. Apex Trigger
Answer: A C E
Explanation:
Option A (Apex Class): Trace flags can be configured for specific Apex classes to debug issues during their execution.

Option C (User): Trace flags can be set for specific users to debug issues occurring in their transactions.
Option E (Apex Trigger): Trace flags can be set for specific triggers to debug execution.
Not Suitable:
Option B (Flow): Trace flags cannot be configured for Flows directly.
Option D (Visualforce): Trace flags are not used to debug Visualforce pages directly.
Debug Logs and Trace Flags

NO.8 Which statement describes the execution order when triggers are associated to the same object and event?

A. Triggers are executed in the order they are modified.

B. Trigger execution order cannot be guaranteed.

C. Triggers are executed alphabetically by trigger name.

D. Triggers are executed in the order they are created.

Answer: B

Explanation:
When multiple triggers are associated with the same object and event, Salesforce does not guarantee the order of execution. This is why it is a best practice to consolidate all logic into a single trigger per object and control execution order using helper classes.
Reference:Apex Triggers Best Practices

NO.9 A lead developer creates a virtual class called "OrderRequest". Consider the following code snippet:
apex
Copy
public class CustomerOrder {
// code implementation
}
How can a developer use the OrderRequest class within the CustomerOrder class?

A. extends (class="OrderRequest")public class CustomerOrder

B. public class CustomerOrder implements Order

C. public class CustomerOrder extends OrderRequest

D. @Implements (class="OrderRequest")public class CustomerOrder

Answer: C

Explanation:
Comprehensive and Detailed Explanation From Exact Extract:
To determine how a developer can use the OrderRequest class within the CustomerOrder class, we need to evaluate the options based on Apex syntax for inheritance, focusing on the fact that OrderRequest is a virtual class. Let's analyze the problem and each option systematically, referencing Salesforce's official Apex Developer Guide.
Understanding Virtual Classes and Inheritance in Apex:
* Virtual Class: In Apex, a virtual class allows other classes to extend it and inherit its methods and properties. It can also define methods that can be overridden by subclasses. The Apex Developer Guide states: "A virtual class can be extended by another class, allowing the subclass to inherit its methods and properties, and override virtual methods if needed" (Salesforce Apex Developer Guide,

Classes).
* Inheritance: Apex supports single inheritance using the extends keyword, where a subclass inherits from a single parent class. The Apex Developer Guide notes: "Use the extends keyword to create a subclass that inherits from a parent class" (Salesforce Apex Developer Guide, Inheritance).
* Virtual vs. Interface:
* A virtual class can contain method implementations and is extended using extends.
* An interface defines method signatures without implementations and is implemented using implements. The question specifies OrderRequest as a virtual class, not an interface.
* Requirement: The CustomerOrder class needs to use OrderRequest, which likely means inheriting its functionality, as OrderRequest is a virtual class.
Evaluating the Options:
* A. extends (class="OrderRequest")public class CustomerOrder
* Syntax: This option attempts to use extends followed by (class="OrderRequest") and then public class CustomerOrder.
* Analysis: The syntax is incorrect. In Apex, the extends keyword is followed directly by the name of the class to extend, with no parentheses or class= notation. The correct syntax would be public class CustomerOrder extends OrderRequest. The Apex Developer Guide specifies: "The extends keyword is followed by the name of the parent class, without additional attributes or parentheses" (Salesforce Apex Developer Guide, Inheritance). The (class="OrderRequest") syntax resembles annotations or markup, but it's not valid in Apex for inheritance.
* Conclusion: Incorrect due to invalid syntax.
* B. public class CustomerOrder implements Order
* Syntax: Uses the implements keyword to implement an interface named Order.
* Analysis: The implements keyword is used to implement an interface, not to extend a class. The question states that OrderRequest is a virtual class, not an interface, so implements is inappropriate. Additionally, the option references Order, not OrderRequest, which appears to be a mismatch with the class name provided in the question. The Apex Developer Guide clarifies:
"The implements keyword is used for interfaces, while extends is used for classes, including virtual classes" (Salesforce Apex Developer Guide, Interfaces). Even if Order were intended to be OrderRequest, implements would still be incorrect for a virtual class.
* Conclusion: Incorrect, as implements is for interfaces, not virtual classes, and Order does not match OrderRequest.
* C. public class CustomerOrder extends OrderRequest
* Syntax: Declares CustomerOrder as a subclass of OrderRequest using the extends keyword.
* Analysis: This is the correct syntax for inheritance in Apex. Since OrderRequest is a virtual class, CustomerOrder can extend it to inherit its methods and properties. The extends keyword allows CustomerOrder to use (inherit) the functionality of OrderRequest. The Apex Developer Guide confirms: "A class can extend a virtual class using the extends keyword, inheriting its methods and properties" (Salesforce Apex Developer Guide, Inheritance). This matches the question's intent of "using" OrderRequest within CustomerOrder, which typically means inheritance when dealing with a virtual class.
* Conclusion: Correct, as it uses the proper extends keyword to inherit from the virtual OrderRequest class.
* D. @Implements (class="OrderRequest")public class CustomerOrder
* Syntax: Uses an @Implements annotation with (class="OrderRequest") before the class declaration.
* Analysis: Apex does not have an @Implements annotation. The implements keyword (without

@) is used for interfaces, and annotations in Apex (e.g., @AuraEnabled, @TestVisible) do not use this format for inheritance or interface implementation. The Apex Developer Guide does not define @Implements as a valid annotation (Salesforce Apex Developer Guide, Annotations). The correct way to extend a virtual class is with extends, as in option C.
* Conclusion: Incorrect due to invalid syntax (@Implements is not a valid Apex annotation).
Why Option C is Correct:
Option C is correct because:
* It uses the extends keyword to properly declare CustomerOrder as a subclass of OrderRequest, which is a virtual class.
* This allows CustomerOrder to inherit and use the methods and properties defined in OrderRequest, fulfilling the requirement to "use" the OrderRequest class within CustomerOrder.
* The syntax aligns with Apex best practices for inheritance as outlined in the Salesforce Apex Developer Guide.
Example for Clarity:
Here's how the correct implementation (option C) would look:
apex
Copy
public virtual class OrderRequest {
public String getOrderDetails() {
return 'Order Details';
}
}
public class CustomerOrder extends OrderRequest {
public String getCustomerOrderInfo() {
// Inherit and use OrderRequest's method
String orderDetails = getOrderDetails();
return 'Customer Order: ' + orderDetails;
}
}
* CustomerOrder extends OrderRequest, inheriting getOrderDetails().
* The developer can use OrderRequest's functionality (e.g., call its methods) within CustomerOrder.
Handling Typos:
* The question's code snippet and options are mostly clear, but option B references Order instead of OrderRequest, which appears to be a typo. The analysis assumes the intended class is OrderRequest, as stated in the question stem.
* Option D has a typo in the class name: Customerorder should be CustomerOrder (capital O). This does not affect the analysis, as the syntax error (@Implements) is the primary issue.
References:
Salesforce Apex Developer Guide:
"Classes" section: Defines virtual classes and their ability to be extended.
"Inheritance" section: Explains the extends keyword for class inheritance.
"Interfaces" section: Clarifies the implements keyword for interfaces, distinguishing it from extends.
"Annotations" section: Confirms that @Implements is not a valid Apex annotation.(Available at: https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/) Platform Developer I Study Guide:
Section on "Developer Fundamentals": Covers Apex object-oriented programming concepts,

including inheritance and virtual classes.(Available at:
https://trailhead.salesforce.com/en/content/learn/modules
/platform-developer-i-certification-study-guide)

NO.10 Universal Containers has a support process that allows users to request support from its engineering team using a custom object, Engineering Support c.
Users should be able to associate multiple Engineering Support __c records to a single Opportunity record.
Additionally, aggregate information
about the Engineering Support _c records should be shown on the Opportunity record.
Which relationship field should be implemented to support these requirements?

A. Lookup field from Opportunity to Engineering _support__c

B. Master-detail field from Engineering Support__c to Opportunity

C. Master-detail field from Opportunity to Engineering Support__c

D. Lookup field from Engineering Support __c to Opportunity

Answer: B

Explanation:
A master-detail relationship fromEngineering_Support__ctoOpportunityallows multiple childEngineering_Support__crecords to be associated with a single parentOpportunity. It also enables roll-up summary fields to aggregate data from the child records onto the parentOpportunity.
Reference:Master-Detail Relationships
Incorrect Options:
A & D:Lookup relationships do not support roll-up summary fields.
C:Master-detail cannot go in the opposite direction (Opportunity to Engineering_Support__c) for this use case.

NO.11 A developer has the following requirements:
* Calculate the total amount on an Order.
* Calculate the line amount for each Line Item based on quantity selected and price.
* Move Line Items to a different Order if a Line Item is not in stock.
Which relationship implementation supports these requirements on its own?

A. Line Item has a re-parentable master-detail field to Order.

B. Line Item has a re-parentable lookup field to Order.

C. Order has a re-parentable lookup field to Line Item.

D. Order has are-parentable master-detail field to Line Item.

Answer: A

Explanation:
Why Re-Parentable Master-Detail Field?
A master-detail relationship allows Line Items to roll up totals (e.g., calculate total amount) to the Order.
Setting the relationship asre-parentablesupports moving Line Items between different Orders.
Why Not Other Options?
B). Lookup field: Does not support roll-up summary fields for calculations.
C and D: Relationships in the opposite direction (Order to Line Item) do not support the required logic.

References:Master-Detail Relationships:https://developer.salesforce.com/docs/atlas.en-us.object_reference.
meta/object_reference/relationships_master_detail.htm
Below is the formatted response for the provided question, adhering to the specified format and requirements.
The question falls under the Salesforce Platform and Declarative Features topic, as it involves implementing a SOAP Web Service in Apex, which is a key concept in the Salesforce Platform Developer I certification. The answer is based on official Salesforce Platform Developer I documentation, with a comprehensive explanation and references to relevant Salesforce documentation.

NO.12 A developer must implement a CheckPaymentProcessor class that provides check processing payment capabilities that adhere to what is defined for payments in the PaymentProcessor interface.
apex
Copy
public interface PaymentProcessor {
void pay(Decimal amount);
}
Which is the correct implementation to use the PaymentProcessor interface class?
A. apex
Copy
public class CheckPaymentProcessor implements PaymentProcessor {
public void pay(Decimal amount) {}
}
B. apex
Copy
public class CheckPaymentProcessor implements PaymentProcessor {
public void pay(Decimal amount);
}
C. apex
Copy
public class CheckPaymentProcessor extends PaymentProcessor {
public void pay(Decimal amount);
}
D. apex
Copy
public class CheckPaymentProcessor extends PaymentProcessor {
public void pay(Decimal amount) {}
}
Answer: A
Explanation:
Comprehensive and Detailed Explanation From Exact Extract:
To determine the correct implementation of the CheckPaymentProcessor class that adheres to the PaymentProcessor interface, we need to evaluate each option based on Apex syntax for interfaces, class implementation, and method overriding. Let's analyze the problem and each option systematically, referencing Salesforce's official Apex Developer Guide.

Understanding Interfaces in Apex:
* Interface Definition: An interface in Apex defines a contract of methods that must be implemented by any class that implements the interface. The PaymentProcessor interface declares a single method: void pay(Decimal amount);. The Apex Developer Guide states: "An interface is a collection of method signatures that a class can implement, requiring the class to provide concrete implementations for each method" (Salesforce Apex Developer Guide, Interfaces).
* Implementing an Interface: A class implements an interface using the implements keyword, and it must provide a concrete implementation (method body) for each method defined in the interface. The Apex Developer Guide notes: "A class that implements an interface must implement all the methods declared in the interface, with matching signatures" (Salesforce Apex Developer Guide, Interfaces).
* Key Points:
* Interfaces are implemented using implements, not extends.
* The implemented methods must match the interface's method signature (name, return type, parameters) and provide a method body.
* Interfaces cannot be extended using extends because they are not classes; extends is used for class inheritance.
Requirement Analysis:
* Interface: PaymentProcessor defines one method: void pay(Decimal amount);.
* Class: CheckPaymentProcessor must implement this interface to provide check processing payment capabilities, meaning it must define the pay method with a matching signature and a concrete implementation.
* Correct Implementation: The class must use implements PaymentProcessor and provide a concrete pay method with a body.
Evaluating the Options:
* A.
apex
Copy
public class CheckPaymentProcessor implements PaymentProcessor {
public void pay(Decimal amount) {}
}
* Syntax: Uses implements PaymentProcessor, which is correct for implementing an interface in Apex.
* Method Implementation: Defines public void pay(Decimal amount) {}, which matches the interface's method signature (void pay(Decimal amount)) and provides a concrete implementation (empty body, but still valid). The Apex Developer Guide confirms: "The implementing class must provide a method body for each interface method, even if it's empty" (Salesforce Apex Developer Guide, Interfaces).
* Access Modifier: The pay method is public, which is required because interface methods are implicitly public, and the implementing method must have the same or less restrictive visibility.
* Conclusion: Correct, as it properly implements the PaymentProcessor interface with a concrete pay method.
* B.
apex
Copy
public class CheckPaymentProcessor implements PaymentProcessor {

public void pay(Decimal amount);
}
* Syntax: Uses implements PaymentProcessor, which is correct.
* Method Implementation: Declares public void pay(Decimal amount); with a semicolon instead of a method body (curly braces {}). In Apex, a method declaration without a body (ending in a semicolon) is valid in an interface or abstract class, but in a concrete class implementing an interface, it must provide a method body. The Apex Developer Guide states: "A class implementing an interface must provide a concrete implementation for each method, or the class must be declared abstract" (Salesforce Apex Developer Guide, Interfaces). Since CheckPaymentProcessor is not abstract, this results in a compilation error: "Method does not override any method from its superclass or interfaces."
* Conclusion: Incorrect, as it fails to provide a concrete implementation of the pay method, causing a compilation error.
* C.
apex
Copy
public class CheckPaymentProcessor extends PaymentProcessor {
public void pay(Decimal amount);
}
* Syntax: Uses extends PaymentProcessor, which is incorrect. In Apex, extends is used for class inheritance, not for implementing interfaces. The PaymentProcessor is an interface, not a class, so it cannot be extended using extends. The Apex Developer Guide clarifies: "Use implements to implement an interface; extends is used for class inheritance" (Salesforce Apex Developer Guide, Interfaces). This results in a compilation error: "Interfaces cannot be extended using extends."
* Method Implementation: Even if the syntax were correct, the pay method lacks a body (ends with a semicolon), causing the same issue as option B.
* Conclusion: Incorrect due to the invalid use of extends for an interface and the lack of a method body.
* D.
apex
Copy
public class CheckPaymentProcessor extends PaymentProcessor {
public void pay(Decimal amount) {}
}
* Syntax: Uses extends PaymentProcessor, which, as noted in option C, is incorrect for an interface. The correct keyword is implements, not extends, leading to a compilation error.
* Method Implementation: Provides a concrete pay method, which is correct in form, but the class declaration error makes the entire implementation invalid.
* Conclusion: Incorrect due to the invalid use of extends for an interface, despite the correct method implementation.
Why Option A is Correct:
Option A is correct because:
* It uses implements PaymentProcessor to properly declare that CheckPaymentProcessor implements the PaymentProcessor interface.
* It provides a concrete implementation of the pay method (public void pay(Decimal amount) {}), matching the interface's method signature and fulfilling the contract.

* The public access modifier aligns with the implicit public visibility of interface methods.
* This implementation adheres to Apex best practices for interfaces as outlined in the Salesforce Apex Developer Guide.
Example for Clarity:
Here's the complete correct implementation (option A) with a sample method body:
apex
Copy
public interface PaymentProcessor {
void pay(Decimal amount);
}
public class CheckPaymentProcessor implements PaymentProcessor {
public void pay(Decimal amount) {
// Example implementation for check processing
System.debug('Processing check payment of amount: ' + amount);
// Add logic to process a check payment
}
}
* The CheckPaymentProcessor class implements the PaymentProcessor interface and provides a concrete pay method, satisfying the interface's contract.
* This class can now be used wherever a PaymentProcessor is required, such as in a payment processing system.
Handling Typos:
* The question's code snippet and options are syntactically correct, with no typos to address.
* The interface and class names are consistent throughout the question.
References:
Salesforce Apex Developer Guide:
"Interfaces" section: Explains how to define and implement interfaces using the implements keyword, and the requirement for concrete method implementations.
"Classes" section: Clarifies the difference between extends (for class inheritance) and implements (for interfaces).(Available at: https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/) Platform Developer I Study Guide:
Section on "Developer Fundamentals": Covers Apex object-oriented programming concepts, including interfaces and their implementation.(Available at: https://trailhead.salesforce.com/en/content/learn/modules/platform-developer-i-certification-study-guide)

NO.13 How should a developer write unit tests for a private method in an Apex class?

A. Use the SeeAllData annotation.

B. Add a test method in the Apex class.

C. Mark the Apex class as global.

D. Use the @TestVisible annotation.

Answer: D

Explanation:
The@TestVisibleannotation allows private methods or variables to be accessed in test classes. This ensures that the private method can be adequately tested without changing its access modifier.
Reference:Testing Apex Code

Below is the formatted response for the provided question, adhering to the specified format and requirements.
The question falls under the Salesforce Platform and Declarative Features topic, as it involves securing SOQL queries against injection vulnerabilities in the context of Visualforce, which is a key concept in the Salesforce Platform Developer I certification. The answer is based on official Salesforce Platform Developer I documentation, with a comprehensive explanation and references to relevant Salesforce documentation. Since the question asks for two correct answers, the response will identify both and explain why they are safe, as well as why the others are not.

NO.14 A developer created a trigger on the Account object. While testing the trigger, the developer sees the error message 'Maximum trigger depth exceeded'.
What could be the possible causes?
A. The developer does not have the correct user permission.
B. The trigger is too long and should be refactored into a helper class.
C. The trigger does not have sufficient code coverage.
D. The trigger is getting executed multiple times.
Answer: D
Explanation:
Maximum Trigger Depth Exceeded:
This error occurs when a trigger causes recursive calls, leading to an infinite loop of execution.
Solution:
Use static variables in helper classes to prevent recursive trigger execution.
Example:

```
public class TriggerHelper {
public static Boolean isTriggerExecuted = false;
}
trigger AccountTrigger on Account (after update) {
if (!TriggerHelper.isTriggerExecuted) {
TriggerHelper.isTriggerExecuted = true;
// Trigger logic here
}
}
```

Why Not Other Options?
A: Permissions do not affect trigger recursion.
B: Length of the trigger is unrelated to recursion.
C: Code coverage does not influence runtime errors like recursion.
References:Trigger Best Practices:https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode
/apex_triggers_best_practices.htm

NO.15 A developer wants to send an outbound message when a record meets a specific criteria.
Which two features satisfy this use case?
A. Flow Builder can be used to check the record criteria and send an outbound message.
B. Approval Process can be used to check the record criteria and send an outbound message without Apex code.

C. Entitlement Process can be used to check the record criteria and send an outbound message without Apex code.
D. Next Best Action can be used to check the record criteria and send an outbound message.
Answer: A B
Explanation:
Flow Builder:
Flow Builder can evaluate record criteria using a Decision element and then send an outbound message using the "Action" element.
Allows automation without Apex.
Approval Process:
Approval Processes can evaluate criteria and trigger outbound messages as one of the associated actions.
Why Not Other Options?
C: Entitlement Process: Focuses on managing entitlements, not triggering outbound messages.
D: Next Best Action: Used for recommendations and guided actions, not for triggering outbound messages.
References:Flow Builder
Documentation:https://help.salesforce.com/s/articleView?id=flow_overview.htm Approval Process
Documentation:https://help.salesforce.com/s/articleView?id=approval_defining.htm

NO.16 Managers at Universal Containers want to ensure that only decommissioned containers are able to be deleted in the system. To meet the business requirement a Salesforce developer adds "Decommissioned" as a picklist value for the Status _c custom field within the Container _c object.
Which two approaches could a developer use to enforce only Container records with a status of "Decommissioned" can be deleted?
Choose 2 answers
A. Before record-triggered flaw
B. Apex trigger
C. After record-triggered flow
D. Validation rule
Answer: B D
Explanation:
Apex Trigger (Option B):
An Apex trigger can be written on theContainer__cobject to prevent deletion unless theStatus__cfield value is
"Decommissioned".
This allows custom logic to be executed during the deletion process.
Apex Triggers Documentation
Validation Rule (Option D):
A validation rule is typically used to enforce conditions during record creation or update. However, it cannot directly restrict deletions.
To ensure deletion restriction, an Apex trigger is more effective.
Not Suitable:
Option A(Before record-triggered flow): Flows triggered "before delete" are not available.
Option C(After record-triggered flow): These occur after a record is deleted, so they cannot prevent

the deletion process.

NO.17 Where are two locations a developer can look to find information about the status of batch or future methods?
Choose 2 answers
A. Developer Console
B. Apex Jobs
C. Paused Flow Interviews component
D. Apex Flex Queue
Answer: A B
Explanation:
Option A (Developer Console):The Developer Console provides a detailed view of the execution logs and the status of Apex batch or future methods.
Option B (Apex Jobs):The Apex Jobs page in Salesforce Setup shows the status of queued, in-progress, or completed batch and scheduled jobs.
Reference:Monitoring Apex Jobs
Incorrect Options:
C (Paused Flow Interviews):Only used for paused flows, not batch or future jobs.
D (Apex Flex Queue):Used for managing batch job execution order, not monitoring job status.

NO.18 Universal Containers implemented a private sharing model for the Account object. A custom Account search tool was developed with Apex to help sales representatives find accounts that match multiple criteria they specify. Since its release, users of the tool report they can see Accounts they do not own.
What should the developer use to enforce sharing permissions for the currently logged in user while using the custom search tool?
A. Use the with sharing keyword on the class declaration.
B. Use the without sharing keyword on the class declaration.
C. Use the userInfo Apex class to filter all SOQL queries to returned records owned by the logged-in user.
D. Use the schema describe calls to determine if the logged-in user has access to the Account object.
Answer: A
Explanation:
Thewith sharingkeyword ensures that the Apex class respects the sharing rules of the current user, including private sharing models. This means the logged-in user will only see records they own or have been shared with them.
Reference:Apex Class Sharing Rules
Incorrect Options:
B:without sharingignores sharing rules, exposing data the user does not have access to.
C:Filtering withUserInfois not a scalable or maintainable solution.
D:Schema describe calls do not enforce sharing permissions but only check object-level access.

NO.19 A developer is alerted to an issue with a custom Apex trigger that is causing records to be duplicated.
What is the most appropriate debugging approach to troubleshoot the issue?

A. Review the Historical Event logs to identify the source of the issue.
B. Add system.debug statements to the code to track the execution flow and identify the issue.
C. Use the Apex Interactive Debugger to step through the code and identify the issue.
D. Disable the trigger in production and test to see if the issue still occurs.
Answer: C
Explanation:
The Apex Interactive Debugger allows developers to step through code execution in real time, which is the most effective way to identify the root cause of the duplication issue.
Reference:Apex Debugging Tools
Incorrect Options:
A:Historical event logs are not specific enough for debugging triggers.
B:system.debugcan help but is less efficient than an interactive debugger.
D:Disabling the trigger disrupts the system and prevents further testing.

NO.20 (Full question statement)
Universal Containers recently transitioned fromClassic to Lightning Experience. One of its business processes requirescertain values from the Opportunity objectto be sent via anHTTP REST calloutto itsexternal order management systemwhen the user presses a custom button on the Opportunity detail page.
Example fields:
Name
Amount
Account
Which two methods should the developer implement to fulfill the business requirement?
Choose 2 answers.
A. Create a customVisualforce quick actionthat performs the HTTP REST callout and use it on the Opportunity detail page.
B. Create anafter update triggeron the Opportunity object that calls a helper method using @future (callout=true) to perform the HTTP REST callout.
C. Create aLightning component quick actionthat performs the HTTP REST callout and use it on the Opportunity detail page.
D. Create aRemote Actionon the Opportunity object that executes an Apex immediate action to perform the HTTP REST callout whenever the Opportunity is updated.
Answer: A C
Explanation:
Comprehensive and Detailed Explanation From Exact Extract:
A (Correct):AVisualforce quick actioncan include Apex logic for a callout, and Visualforce is still valid in Lightning Experience.
C (Correct):ALightning Web Component (or Aura component)exposed as aquick actioncan perform callouts via Apex when the user clicks a button. This supports a more modern UI experience.
Incorrect options:
B:@future(callout=true) works but cannot be triggeredsynchronously by a button press. It's for background
/automated processes.
D:Remote Actionis legacy functionality used with Visualforce/Aura and doesn't suit modern Lightning

and button-driven synchronous actions.
Reference:Salesforce Developer Guide - Callouts from ApexSalesforce Lightning Developer Guide - Quick Actions This is part ofUser Interface (25%)andProcess Automation and Logic (30%), involvingintegration techniques and callouts.