PassQueen
a.k.a dumpsqueen.com

GIAC Secure Software Programmer – Java

GIAC GSSP-Java

Version Demo

Total Demo Questions: 30

Total Premium Questions: 308

Buy Premium PDF

https://passqueen.com

support@passqueen.com

Topic Break Down

Topic	No. of Questions
Topic 1, Volume A	105
Topic 2, Volume B	104
Topic 3, Volume C	79
Total	308

QUESTION NO: 1

Mark works as a Programmer for InfoTech Inc. He develops a program that defines a class named Inventory that has an instance variable named NumOfItems. Which of the following properties will be applied by NumOfItems?

Each correct answer represents a complete solution. Choose all that apply.

A. It will not be visible in Static methods if it passed as a parameter.

B. It will be visible in Static methods if it passed as a parameter.

C. It will be available for all the instance methods of the class.

D. It becomes available for garbage collection if it is no longer in scope.

ANSWER: B C

Explanation:

An instance variable is a field declared in a class but outside its methods, constructors, and initializer blocks. Therefore, *It will be available for all the instance methods of the class.* Instance methods operate on a particular `Inventory` object and can access that object's `NumOfItems` field directly, subject to Java access-control rules. Each separate `Inventory` object has its own instance-field value.

It will be visible in Static methods if it passed as a parameter. is also correct. A static method has no implicit instance (`this`), so it cannot directly refer to an instance field by name alone. However, it can receive either the field's value as a parameter or an `Inventory` reference as a parameter and use that input during execution. For example, a static method receiving an `int` parameter can read the supplied value of `NumOfItems`; alternatively, a method receiving an `Inventory` parameter can access `inventory.NumOfItems` when permitted by the field's access modifier.

The Java Language Specification describes fields as variables declared in classes and explains that instance variables belong to class instances. See JLS §8.3.1.1, Static Fields and JLS §15.12.1, Determine Type to Search.

QUESTION NO: 3

Which of the following options directs the Java compiler to search files in the current directory?

A. dir

B. ; (i.e. a semi colon)

C. -d

D. current

E. . (i.e. a dot)

ANSWER: E

Explanation:

The correct answer is **. (i.e. a dot)**. In Java command-line usage, a single dot denotes the current working directory. When used as an entry in the class path, such as `javac -classpath . MyClass.java`, it instructs the compiler to use the directory from which the command is being run when locating referenced classes and packages. This is especially important when compilation depends on classes that have already been compiled into the current directory or its package-subdirectory structure.

The class path is a list of locations in which Java tools search for user-defined classes. A dot is the conventional relative-path notation for the current directory on supported operating systems. It may appear alone or alongside additional class-path entries; the separator between entries is platform-specific, but the dot itself remains the notation for the current directory. Oracle's javac documentation describes the compiler's class-path option and class-search behavior. The Java tutorial also documents class path and path conventions, including use of the current directory.

QUESTION NO: 4

Which of the following are valid thread state transitions?

Each correct answer represents a complete solution. Choose all that apply.

A. From running to ready.

B. From ready to waiting.

C. From running to waiting.

D. From waiting to ready.

E. From ready to running.

F. From waiting to running.

ANSWER: A C D E

Explanation:

The valid conceptual lifecycle transitions are **From running to ready.**, **From running to waiting.**, **From waiting to ready.**, and **From ready to running.**. A runnable thread that is currently executing can return to the ready pool when the scheduler preempts it or when it voluntarily yields execution. The scheduler can then select a ready thread and assign it the processor, producing the ready-to-running transition.

A running thread enters a waiting condition when it performs an operation that requires it to pause, such as waiting for monitor coordination, sleeping for a period, or waiting for relevant execution conditions. Once the waiting condition is resolved—for example, through notification, timeout, or completion of the awaited condition—the thread becomes eligible for scheduling again, which is represented as waiting to ready. It must then be selected by the scheduler before it executes.

Java's `Thread.State` API expresses these concepts using states such as `RUNNABLE`, `WAITING`, `TIMED_WAITING`, and `BLOCKED`; the classic ready/running model is a scheduling abstraction rather than a direct one-to-one enumeration in that API. See the Java Thread.State documentation and the Object.wait() documentation.

QUESTION NO: 5

Mark works as a Software Developer for ZenTech Inc. He writes the following code.

```
1. public class Ques0352 {
2. public static void main(String[] args) {
3. String s1="JavaObject";
4. String s1="ObjectJava";
5. String s2="ObjectJava";
6. if(s2.equals(s1))
7. { System.out.println("Equal"); }
8. else
9. { System.out.println("Unequal"); }
10. }
11. }
```

Which of the following will happen to the code when he attempts to compile and run it?

A. It will compile successfully and Unequal will be displayed as output.

B. It will give a compile-time error at line 3.

C. It will compile successfully and Equal will be displayed as output.

D. It will give a compile-time error at line 4.

ANSWER: D

Explanation:

It will give a compile-time error at line 4. Both declarations of `s1` occur in the same method block, which means they have the same local-variable scope. Java requires each local variable declared within a scope to have a unique name; a second declaration of `String s1` in that same scope is prohibited. Consequently, the compiler reports that `s1` has already been defined in `main`, identifying the second declaration on line 4 as the location of the error. Compilation must complete successfully before the Java Virtual Machine can run the program, so the `equals` call and either output statement are never reached. To make the program valid, the second variable would need a different identifier, or the code would need to assign a new value to the already declared variable using `s1 = "ObjectJava";` rather than declaring it again. The Java Language Specification defines the scope of a local variable declaration and the restriction that a local variable cannot be redeclared within an overlapping scope. See JLS §6.3, Scope of a Declaration and JLS §14.4, Local Variable Declaration Statements.

QUESTION NO: 6

Mark writes a class Practice.java. This class needs to access the com.bar.Test class that is stored in the Test.jar file in the directory /practice. How would you compile your code?

A. javac -classpath /practice/Test.jar Practice.java

B. javac -classpath /practice/ Practice.java

C. javac -classpath /practice/Test.jar/com/bar Practice.java

D. javac -classpath /practice Practice.java

ANSWER: A

Explanation:

The command `javac -classpath /practice/Test.jar Practice.java` is correct because the compiler's class path must name the JAR file that contains the compiled `com.bar.Test` class. A JAR is a class-path container: when it is listed on the class path, `javac` searches within it for class files using the package's directory structure. Thus, an import or reference to `com.bar.Test` is resolved from the JAR entry `com/bar/Test.class`. The source file `Practice.java` is compiled from the current directory unless a path is provided for it, while `-classpath` (or its short form, `-cp`) supplies the locations used to find referenced classes and packages during compilation. The Java compiler documentation specifies that class-path entries may be directories or archive files such as JAR files, and Oracle's JAR documentation describes JAR files as supporting class loading through the class path. See the javac command documentation and the JAR command documentation.

QUESTION NO: 7

Which of the following will be returned by the expression "string" instanceof String?

A. 0

B. 1.0

C. null

D. false

E. 0.0

F. true

ANSWER: F

Explanation:

The expression `"string" instanceof String` evaluates to `true`. In Java, a string literal such as `"string"` is a reference to an object whose type is `java.lang.String`. The `instanceof` operator performs a runtime type test: it returns a boolean value indicating whether its left-hand operand is an instance of the reference type named on its right-hand side, including instances of subclasses where applicable. Since the literal is a `String` object and the tested type is exactly `String`, the runtime test succeeds and produces `true`.

This behavior is part of Java's type system and is useful when code receives an object through a broader reference type, such as `Object`, and needs to determine whether it can safely be handled as a particular class. Although the expression uses a literal directly, the same type-test rule applies to String values held in variables. The Java Language Specification defines the `instanceof` operator and specifies that its result is a boolean value. See Java Language Specification, §15.20.2 and Java String API documentation.

QUESTION NO: 8

Which directory in JAR files is used to store package and extension configuration data?

A. META-INF

B. GAMMA-INF

C. ZIP-INF

D. TAR-INF

ANSWER: A

Explanation:

`META-INF` is the reserved directory in a Java Archive file for metadata and configuration resources that describe the archive, its packages, and related extensions. Most notably, it commonly contains `MANIFEST.MF`, whose main attributes can identify the application entry point, class-path dependencies, implementation and specification details, package sealing, and other archive-level settings. Package-specific manifest sections can provide metadata that applies to individual Java packages. The same directory is also the conventional location for signing-related files, including signature files and signature blocks, as well as service-provider configuration files under `META-INF/services`. This reserved layout allows the Java runtime, class loaders, deployment tooling, and libraries to locate archive metadata consistently without treating it as ordinary application content. The JAR specification explicitly defines the `META-INF` directory and the special interpretation of files it contains, including the manifest and signature material. Java's JAR documentation likewise describes manifest attributes used for package and extension-related configuration. See the JAR File Specification and the Java jar tool documentation.

QUESTION NO: 9

Which of the following, in JDBC 2.0, are the sub interfaces of the Statement interface? Each correct answer represents a complete solution. Choose two.

A. ParsedStatement

B. CompiledStatement

C. PreparedStatement

D. CallableStatement

ANSWER: C D

Explanation:

`PreparedStatement` and `CallableStatement` are the JDBC interfaces that form the specialized statement hierarchy rooted at `Statement`. A `PreparedStatement` extends `Statement` and represents a precompiled SQL statement. Its SQL can include parameter markers, allowing applications to bind values through typed setter methods before executing the statement. This supports repeated execution of the same SQL structure with different values and is a core JDBC mechanism for parameterized database operations.

`CallableStatement` is the further specialization used to invoke database stored procedures. It extends `PreparedStatement`, and is therefore also a subinterface of `Statement` through that inheritance chain. In addition to inherited parameter binding behavior, it provides facilities relevant to procedure calls, including registration and retrieval of OUT parameters. The Java API documentation explicitly declares the relationships: `PreparedStatement` extends `Statement`, while `CallableStatement` extends `PreparedStatement`. See the PreparedStatement API documentation and the CallableStatement API documentation.

QUESTION NO: 10

Mark works as a Programmer for InfoTech Inc. He develops a code snippet for a class named servletClassA that extends the HttpServlet class. Which of the following HttpServlet class methods are not required to be overridden by the servletClassA?

Each correct answer represents a complete solution. Choose all that apply.

A. doDelete()

B. doPost()

C. doGet()

D. doOptions()

E. service()

ANSWER: A B C D E

Explanation:

All listed methods are not required to be overridden merely because a class extends `HttpServlet`. The servlet API supplies concrete inherited implementations for `doDelete()`, `doPost()`, `doGet()`, `doOptions()`, and `service()`. Consequently, a subclass can compile without declaring replacements for any particular method. In normal application design, a servlet overrides the request-handling method or methods needed for the HTTP verbs it intends to support, commonly `doGet()` and/or `doPost()`. That is an application behavior decision, not an inheritance requirement imposed by the Java type system or the servlet API. The inherited `service()` implementation performs HTTP request dispatching to the appropriate `doXxx` handler, while the inherited verb handlers provide the API-defined default behavior when they are not customized. `doOptions()` likewise has an inherited implementation that determines allowed methods. Thus every method listed satisfies the wording “not required to be overridden.” The Jakarta Servlet API documents these concrete methods on HttpServlet; the corresponding legacy Java EE API documentation is available from Oracle’s HttpServlet reference.

QUESTION NO: 11

Which of the following methods are used in the verification of a signature?

Each correct answer represents a complete solution. Choose all that apply.

A. initVerify

B. update

C. verify

D. initSign

ANSWER: A B C

Explanation:

Signature verification with Java's `java.security.Signature` API follows a defined lifecycle. `initVerify` prepares the `Signature` object for verification by initializing it with the public key that corresponds to the private key used to generate the signature. This initialization establishes that the object will perform verification rather than signing. The application then supplies the exact signed data through one or more calls to `update`. Supplying data incrementally is supported, and the bytes provided must match the data for which the signature was originally generated. Finally, `verify` receives the signature bytes and performs the cryptographic verification operation, returning a Boolean result that indicates whether the signature is valid for the initialized public key and accumulated data. These methods therefore represent the complete verification workflow: initialize for verification, provide the signed content, and verify the supplied signature. This usage is specified in the Java `Signature` API documentation, including its state transitions and method contracts. See the Java Signature API documentation and the initVerify method specification.

QUESTION NO: 12

You work as a Software Developer for UcTech Inc. You build an online book shop, so that users can purchase books using their credit cards. You want to ensure that only the administrator can access the credit card information sent by users. Which security mechanism will you use to accomplish the task?

A. Confidentiality

B. Authorization

C. Authentication

D. Data integrity

ANSWER: B

Explanation:

Authorization is the appropriate security mechanism because it determines whether an already identified user has permission to perform a particular action on a particular resource. In this case, the application must enforce an access-control policy that permits access to stored or transmitted credit-card information only when the authenticated user has the administrator role or an explicitly granted administrative privilege. The policy must be enforced server-side at every endpoint, service method, and data-access path that can retrieve, display, export, or modify payment information. A practical implementation uses role- or attribute-based access controls, denies access by default, and checks the user's permissions on every request rather than relying on hidden interface elements. Authentication is an important prerequisite because the system must establish who the requester is, but authorization is what makes the decision that the requester may access card data. OWASP recommends validating permissions for every request and applying least privilege in authorization design; see the OWASP Authorization Cheat Sheet. NIST similarly defines access control as limiting system access to authorized users and processes in NIST's Access Control glossary entry.

QUESTION NO: 13

Identify whether the given statement is true or false.

"There is no method to create a new thread other than extending the Thread class."

A. True

B. False

ANSWER: B

Explanation:

The correct answer is *False*. Java supports creating work that runs concurrently without requiring a class to extend `Thread`. A traditional alternative is to implement the `Runnable` interface, supply its `run()` implementation to a `Thread` object, and invoke `start()`. This separates the task being performed from the mechanism that executes it and avoids consuming Java's single class-inheritance relationship merely to define concurrent behavior.

In modern Java code, tasks are commonly submitted to an `ExecutorService`, which manages a pool of worker threads. This approach lets an application control resource use, queue tasks, obtain results through `Future`, and shut down execution cleanly. Java also offers `Callable` for tasks that return a value or throw checked exceptions. Therefore, extending `Thread` is one possible mechanism, not the only method for executing code in a separate thread. Oracle's documentation describes `Runnable` as a task intended to be executed by a thread and documents executor services as mechanisms for executing submitted tasks: Runnable API and ExecutorService API.

QUESTION NO: 14

Which of the following statements about data integrity of a container are true? Each correct answer represents a complete solution. Choose two.

A. It ensures that an eavesdropper cannot read an HTTP message being sent from a client to a container.

B. Data integrity ensures that information has not been modified by a third party while it is in transit.

C. It ensures that a hacker cannot alter the contents of an HTTP message while it is in transit from a container to a client.

D. Data integrity ensures that information is made available to users who are authorized to access it.

ANSWER: B C

Explanation:

Data integrity is the security property that ensures data remains accurate, complete, and unaltered except through authorized actions. For HTTP traffic involving a Java container, integrity protection detects or prevents unauthorized modification of a message while it travels across the network. Therefore, "Data integrity ensures that information has not been modified by a third party while it is in transit." correctly describes integrity protection. "It ensures that a hacker cannot alter the contents of an HTTP message while it is in transit from a container to a client." is also correct because it applies that principle specifically to a response sent by the container. In practice, HTTPS uses TLS protections, including message-authentication mechanisms, to provide integrity for HTTP data in transit; when an integrity check fails, the received data must not be accepted as authentic and unchanged. Integrity is distinct from confidentiality and authorization: its focus is whether the message content was modified without authorization. The Java security guidance identifies integrity as protection against unauthorized modification, and TLS is the standard transport-layer mechanism used by web applications to protect HTTP exchanges. See Oracle's Java Security Overview and the IETF's TLS 1.3 specification.

QUESTION NO: 15

Which of the following methods are overridden by the FileInputStream class? Each correct answer represents a complete solution. Choose all that apply.

A. void reset()

B. void write(int b)

C. void flush()

D. long skip(long numBytes)

E. void close()

ANSWER: D E

Explanation:

`long skip(long numBytes)` is correct because `FileInputStream` supplies its own implementation of the `InputStream.skip(long)` contract. For a file-backed stream, skipping can be performed by moving the underlying file position forward rather than necessarily reading and discarding every intervening byte. The method returns the number of bytes actually skipped, which may be less than requested when the stream reaches the end of the file. The parameter name does not affect overriding; the matching method signature is defined by the method name and parameter types.

`void close()` is also correct because `FileInputStream` overrides `InputStream.close()` to release the resources associated with its underlying file descriptor. Closing a file stream is essential for prompt release of operating-system file handles and related resources. Java's API documentation identifies `close()` and `skip(long)` among the methods declared by `FileInputStream`. See the FileInputStream API documentation and the inherited-method contract in the InputStream API documentation.

QUESTION NO: 16

Which of the following exceptions will be thrown by the validate method if the result type does not match the Source type, or if the specified source is neither SAXSource nor DOM Source?

A. SAXException

B. NullPointerException

C. IllegalArgumentException

D. IOException

ANSWER: C

Explanation:

`IllegalArgumentException` is the exception specified for this validation API contract when the supplied `Result` is incompatible with the supplied `Source`, or when the source implementation is not one of the supported source types for this method. The `Validator.validate(Source, Result)` method processes XML represented by the source and may write an augmented result, so the source and result representations must be compatible. For example, a SAX-oriented source requires the corresponding SAX-oriented result handling; the API cannot silently convert arbitrary source and result representations on behalf of the caller. The JDK API documentation explicitly lists `IllegalArgumentException` for an unsupported source type and for a result type that does not match the source type. This is a caller-usage error detected from the method arguments, rather than a validation failure in the XML content itself. Developers should select supported `Source` and `Result` implementations and ensure they form a valid pair before invoking validation. See the official Validator.validate API documentation and the Java XML Validation package documentation.

QUESTION NO: 17

Which of the following statements about a native modifier in Java are true? Each correct answer represents a complete solution. Choose two.

A. It can be applied to methods and variables.

B. A method with a native modifier must end with a semicolon.

C. It can be applied only to methods.

D. It can be applied only to variables.

E. A separate Java class must be written to provide implementation for a native method.

ANSWER: B C

Explanation:

A method with a native modifier must end with a semicolon because a native method is declared in Java but its implementation is supplied outside Java, typically in platform-specific code accessed through the Java Native Interface (JNI). Since the Java declaration does not include a Java method body, it uses the declaration form that terminates with `;`, such as `public native int process(byte[] input);`. The runtime links the declaration to a compatible native implementation when the relevant native library is loaded.

It can be applied only to methods because `native` is a Java method modifier. The Java Language Specification defines native method declarations and requires that a native method declaration have no block body. Native code can interact with Java fields and objects through JNI APIs, but Java fields themselves are not declared `native`. This distinction is important when reviewing declarations: `native` identifies a method whose executable implementation is provided by non-Java code, rather than identifying data stored externally. See the Java Language Specification, Method Modifiers and Oracle's JNI specification introduction.

QUESTION NO: 18

You work as a Software Developer for NewTech Inc. You write a bean class called

EmployeeBean. The class contains two methods, EmpSal() and EmpAttendance(). Both these methods can be accessed by the ADMIN role. The EmpSal() method can be accessed only by the HR role, while the EmpAttendance () method can be accessed only by the DBA role. You want the EmpAttendance() method to be accessed by the HR role also. However, no other roles in the class except ADMIN, DBA, and HR should be able to access the EmpAttendance() method.

Which of the following steps will you take to accomplish the task?

Each correct answer represents a complete solution. Choose all that apply.

A. Use the element of the deployment descriptor and declare the as DBA.

B. Use the @RunAs("ADMIN") annotation to allow the HR the privileges as ADMIN.

C. Use the @RunAs("DBA") annotation to allow the HR the privileges as DBA.

D. Use the element of the deployment descriptor to declare the role names as ADMIN, DBA, and HR and the method-name as EmpAttendance.

E. Use the @PermitAll annotation with the EmpAttendance() method to allow the HR to access the EmpAttendance() method.

F. Use the @RolesAllowed({"ADMIN", "DBA", "HR"}) annotation on the EmpAttendance() method.

ANSWER: D F

Explanation:

Container-managed authorization for an enterprise bean method is configured by associating that specific method with every role that is permitted to invoke it. The deployment descriptor solution is to define a method-permission for `EmpAttendance` that includes both the `DBA` and `HR` roles. Method permissions are additive: listing both authorized roles means that a caller in either role may invoke the method, while callers in roles not listed for that method are not granted access through this permission. Existing access for `ADMIN` must also remain configured if the requirement is that administrative callers continue to access the method.

The annotation equivalent is `@RolesAllowed({"ADMIN", "DBA", "HR"})` on `EmpAttendance()`. This explicitly expresses the complete allowed caller set at the method level and is appropriate when annotation-based security metadata is used. `@RolesAllowed` is the standard declarative mechanism for restricting invocation to named roles; it is evaluated by the container before the business method is called. The Jakarta Annotations API documents this annotation at Jakarta RolesAllowed API. Enterprise Beans declarative security and method-permission behavior are specified in the Jakarta Enterprise Beans specification.

QUESTION NO: 19

Which of the following statements about the form-based authentication are true? Each correct answer represents a complete solution. Choose two.

A. It provides a weaker security check than the HTTP Digest and HTTPS Client authentications.

B. It requires a hidden field that supplies the login-constraint used by the application.

C. It requires that the action of the login form must be j_security_check.

D. It transmits username and password over the network in the form of Base64 encoding.

ANSWER: A C

Explanation:

Form-based authentication delegates credential collection to an application-supplied HTML login page. Because the credentials are submitted as ordinary form parameters, it does not itself provide cryptographic protection for the username and password. A secure deployment must protect the login page and its submission with TLS/HTTPS. In contrast, HTTP Digest authentication uses a challenge-response protocol intended to avoid sending the reusable password directly, and HTTPS client authentication relies on client certificates and the TLS protocol. Thus, form-based authentication is considered a weaker authentication mechanism when it is not combined with secure transport.

For servlet-container form authentication, the login form must submit to the container-defined endpoint `j_security_check`. The form normally uses the POST method and supplies the special fields `j_username` and `j_password`; the container intercepts that request, validates the credentials through its configured security realm, and establishes the authenticated identity. This endpoint is part of the conventional Servlet form-login contract, rather than an application business-action URL. See the Oracle Java EE form-based authentication documentation and the Jakarta Servlet security documentation.

QUESTION NO: 20

Which of the following classes of java.util.logging prints a brief summary of the LogRecord in a human readable format?

A. XMLFormatter

B. SimpleFormatter

C. StreamHandler

D. MemoryHandler

ANSWER: B

Explanation:

`SimpleFormatter` is the standard `java.util.logging` formatter intended to render a `LogRecord` as concise, human-readable text. A formatter converts the structured data held by a log record—such as its timestamp, logger name, severity level, source information, message, and any associated exception—into the character representation written by a handler. The Java API specifically describes `SimpleFormatter` as formatting records into a brief human-readable form, which directly matches the wording in the question.

The precise appearance of this text is configurable. The `java.util.logging.SimpleFormatter.format` system property can define the output pattern, allowing applications and operational environments to tailor details such as date formatting, level display, and exception-stack-trace output without changing application logging calls. This makes `SimpleFormatter` appropriate for conventional console or text-file logs that developers and operators read directly. See the official Java API documentation for SimpleFormatter and the Formatter base class.

QUESTION NO: 21

Which of the following code fragments will compile without error?

Each correct answer represents a complete solution. Choose all that apply.

A. boolean a = false; if(a)
System.out.println(a);

B. int a = 10; if(a != 10)
System.out.println(a);

C. int a = 0; if(a)
System.out.println(a);

D. boolean a = true; if(!a);

ANSWER: A B D

Explanation:

The fragments `boolean a = false; if(a) System.out.println(a);`, `int a = 10; if(a != 10) System.out.println(a);`, and `boolean a = true; if(!a);` compile because each `if` statement has an expression of type `boolean`. A boolean variable may be used directly as the condition, so `if(a)` is valid when `a` is declared as `boolean`, regardless of whether its runtime value is `true` or `false`. Likewise, the relational expression `a != 10` evaluates to a boolean value and is therefore a valid condition for an `if` statement.

The expression `!a` is also boolean when `a` is boolean. The semicolon immediately following `if(!a)` is legal Java syntax: it serves as the empty statement controlled by the `if`. Although an empty controlled statement is usually unintentional and should generally be avoided for readability and maintainability, it does not produce a compilation error. The Java Language Specification defines the required grammar for `if` statements and specifies that the condition expression must be compatible with `boolean`. See JLS §14.9: The if Statement and JLS §15.21: Equality Operators.

QUESTION NO: 22

Identify whether the given statement is true or false.

"When a Java program starts up, one thread begins running immediately."

A. True

B. False

ANSWER: A

Explanation:

True is correct. For a conventional Java application, the Java Virtual Machine starts execution by creating an initial thread and invoking the application's `main` method on that thread. This is commonly called the main thread. It provides the initial execution path for the program: class initialization needed to begin execution occurs, the specified main class is initialized, and its `public static void main(String[] args)` method is invoked. Consequently, at startup there is at least one thread executing application code.

The main thread can subsequently create additional threads, for example by calling `Thread.start()` or by using an executor service. The JVM itself may also use supporting threads for runtime activities, but those details do not alter the statement's core point: program execution begins with an initial thread. The Java Language Specification explicitly describes startup as the JVM creating an initial thread and invoking the main method. See the Java Language Specification, section 12.1.3. The Java API also identifies the main thread as the thread that starts execution of an application; see the Thread API documentation.

QUESTION NO: 23

Which of the following is the appropriate deployment descriptor elements entry for the code given below? @RunAs("admin")

@Stateless public class StudentBean implements Student { //more code ...

}

A. <enterprise-beans>
...
<session>
.
<ejb-name>Student</ejb-name>
...
<security-identity>
<run-as>
<method-permission>admin</method-permission>
</run-as> </security-identity>
...
</session>
..
</enterprise-beans>

B. <enterprise-beans>
...
<session>
.
<ejb-name>Student</ejb-name>
...
<security-identity>
<run-as>
<role-name>admin</role-name>
</run-as> </security-identity>
...
</session>
..
</enterprise-beans>

C. <enterprise-beans>
...
<session>
.
<ejb-name>Student</ejb-name>
...
<security-identity>
<run-as>
< security-role-ref>admin</ security-role-ref>
</run-as> </security-identity>
...
</session>
..
</enterprise-beans>
D. <enterprise-beans>
...
<session>
.
<ejb-name>Student</ejb-name>
...
<security-identity>
<run-as>admin</run-as>
</security-identity>
...
</session>
..
</enterprise-beans>

A. Option A

B. StudentBeanadmin

C. Option C

D. Option D

ANSWER: B

Explanation:

The appropriate deployment-descriptor entry is `<run-as><role-name>admin</role-name></run-as>` nested within the `<session>` element for `StudentBean`. The `@RunAs("admin")` annotation declares the run-as identity for the enterprise bean: when the container invokes another component or accesses protected resources while executing the bean, it performs that work using the security identity associated with the `admin` role. In an EJB deployment descriptor, the bean is identified with `<ejb-name>StudentBean</ejb-name>`, and its run-as role is expressed through the nested `<run-as>` and `<role-name>` elements. This descriptor form represents the same run-as configuration declared by the annotation and can be used as deployment metadata for the stateless session bean. The role name is the literal application role, `admin`; it is not a principal name or a Java class name. Jakarta Enterprise Beans defines `run-as` as a session-bean deployment descriptor element and specifies that its role name identifies the security role used for run-as behavior. See the Jakarta Enterprise Beans specification and the RunAs API documentation.

QUESTION NO: 24

Which of the following web-resource element descriptions will be used if you want to restrict all URL ' s in the application and perform authentication for the http delete method?

A. < security-constraint >< web-resource-collection >< web-resource-name > AccountServlet < /web-resource-name >< url-pattern > /* < /url-pattern >< method-name > GET < /method-name >< method-name > DELETE < /method-name >< /web-resource-collection >< auth-constraint >< role-name > Manager < /role-name >< /auth-constraint >< /security-constraint >

B. < security-constraint >< web-resource-collection >< url-pattern > * < /url-pattern >< method-name > GET < /method-name >< method-name > DELETE < /method-name >< /web-resource-collection >< auth-constraint >< role-name > Manager < /role-name >< /auth-constraint >< /security-constraint >

C. < security-constraint >< web-resource-collection >< web-resource-name > AccountServlet < /web-resource-name >< url-pattern > /* < /url-pattern >< http-method > GET < /http-method >< http-method > DELETE < /http-method >< /web-resource-collection >< auth-constraint >< role-name > Manager < /role-name >< /auth-constraint >< /security-constraint >

D. < security-constraint >< web-resource-collection >< web-resource-name > AccountServlet < /web-resource-name >< url-pattern > * < /url-pattern >< http-method > GET < /http-method >< http-method > DELETE < /http-method >< /web-resource-collection >< auth-constraint >< role-name > Manager < /role-name >< /auth-constraint >< /security-constraint >

ANSWER: C

Explanation:

The security-constraint containing `<url-pattern>/*</url-pattern>` and `<http-method>DELETE</http-method>` is correct because a URL pattern of `/*` maps the constraint to all application paths. Within a `web-resource-collection`, the Servlet deployment-descriptor element used to select HTTP request methods is `http-method`. Listing `DELETE` therefore makes the authorization constraint apply to DELETE requests for every URL matched by the collection. The included `auth-constraint` identifies the `Manager` role as the authorized role, so the container will require the configured authentication mechanism before it can determine whether the caller is in that role and permit the protected request. The `web-resource-name` is a descriptive label for the resource collection and does not alter the URL or method matching behavior. The same collection also explicitly protects GET requests; this does not prevent it from correctly protecting DELETE requests as required. The Jakarta Servlet deployment descriptor schema defines `http-method` as the request-method selector in a web-resource collection and permits URL patterns such as `/*`. See the Jakarta Servlet security-constraints specification and the Jakarta EE web-app schema.

QUESTION NO: 25

Which of the following statements correctly describe the features of the singleton pattern? Each correct answer represents a complete solution. Choose all that apply.

A. Singletons are used to control object creation by limiting the number to one but allowing the flexibility to create more objects if the situation changes.

B. Singletons can only be stateless, providing utility functions that need no more information than their parameters.

C. A singleton class may disappear if no object holds a reference to the Singleton object, and it will be reloaded later when the singleton is needed again.

D. The behavior of a singleton can be obtained by static fields and methods such as java.lang.Math.sin(double).

ANSWER: A D

Explanation:

Singletons are used to control object creation by limiting the number to one but allowing the flexibility to create more objects if the situation changes. This reflects the pattern's central purpose: centralizing control over instance creation. A conventional singleton enforces one instance, but the creation logic is kept behind a controlled access point. Consequently, an implementation can be evolved into a controlled multiton or instance pool if operational requirements later require a bounded or otherwise managed number of instances. The important design feature is that callers do not directly construct unmanaged instances.

The behavior of a singleton can be obtained by static fields and methods such as java.lang.Math.sin(double). Static members are associated with the class rather than with an object, so they provide globally accessible behavior without requiring callers to obtain an instance. This can satisfy the same practical need when no polymorphism, interface-based substitution, instance lifecycle control, or dependency injection is needed. For example, the Java platform exposes `Math.sin(double)` as a static utility method. The singleton pattern remains useful when an object is preferable because it can implement interfaces, encapsulate state, and participate in object-oriented dependency management. See Refactoring.Guru's Singleton pattern reference and the Java Math API documentation.

QUESTION NO: 26 - (SIMULATION)

SIMULATION

Fill in the_____blank with the required interface name to complete the statement below.

An object of the interface is provided by the container to invoke the next filter in a chain of filters.

ANSWER: See the explanation for the answer

Explanation:

The required interface name is `FilterChain`.

In the Servlet API, the container supplies a `FilterChain` object to a filter's `doFilter()` method. The filter invokes the next filter (or ultimately the target servlet/resource) with:

```
chain.doFilter(request, response);
```

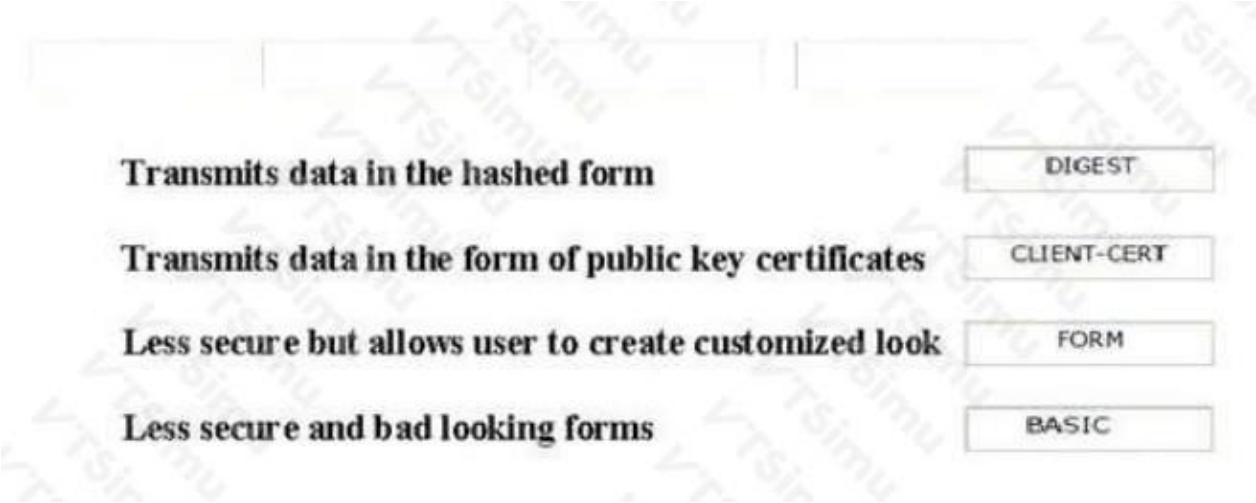
QUESTION NO: 27 - (DRAG DROP)

DRAG DROP

Drag and drop the appropriate authentication types from the given options to match their properties.

Select and Place:

ANSWER:



Explanation:

The correct relationships describe the standard servlet-container authentication mechanisms used by Java web applications. DIGEST authentication transmits credentials using a digest calculation rather than putting the password itself directly into the request. The client and server use a challenge-response process involving a hash, which is why it matches the property that data is transmitted in hashed form. The Jakarta Servlet specification describes this mechanism as digest authentication and defines the request digest calculation in its HTTP authentication coverage.

CLIENT-CERT authentication is based on an X.509 client certificate presented during the TLS connection process. Such certificates contain public-key material and identity information, so this mechanism naturally matches transmission and authentication using public key certificates. The application server can use the certificate identity to establish the authenticated caller without requiring a traditional form-based username and password submission.

FORM authentication uses an application-provided login page and error page. Because those pages are implemented and styled by the web application, this approach gives developers control over the user experience and permits a customized look. BASIC authentication instead relies on the browser's built-in HTTP authentication prompt. That default browser dialog provides minimal presentation control and is commonly regarded as visually unappealing. Its credentials are only Base64 encoded at the HTTP level, so it should be protected with TLS in production. Oracle's Java EE security guide summarizes BASIC, FORM, DIGEST, and CLIENT-CERT authentication, while the Jakarta Servlet specification defines login configuration and supported authentication methods.

QUESTION NO: 28

Which of the following JDBC interfaces is described in the statement below?

"It provides support for executing SQL statements and stored procedures."

A. Driver

B. ResultSet

C. PreparedStatement

D. Connection

E. CallableStatement

ANSWER: E

Explanation:

`CallableStatement` is the JDBC interface intended for invoking database stored procedures. It extends `PreparedStatement`, so it retains the ability to execute parameterized SQL, while adding stored-procedure-specific capabilities. A program obtains a `CallableStatement` from a JDBC `Connection`, typically using SQL call escape syntax such as `{call procedure_name(?, ?)}`. The application binds values to input parameters with setter methods, registers output parameters using `registerOutParameter`, executes the call, and then retrieves output values through

the appropriate getter methods. This design supports procedures that accept input, return output values, or both, and it also supports function calls through JDBC call syntax. The Java API documentation defines `CallableStatement` as the interface used to execute SQL stored procedures, making it the precise match for a description that explicitly includes stored procedures while still operating within JDBC's SQL execution model. See the official Java CallableStatement API and the JDBC stored procedures tutorial.

QUESTION NO: 29

Which of the following statements about a filter are true?

Each correct answer represents a complete solution. Choose all that apply.

A. Like a servlet, a filter is also declared in the deployment descriptor.

B. The life cycle of a filter is managed by the container.

C. The life cycle of a filter has three methods, namely init(), service(), and destroy().

D. Every filter must implement the Filter interface.

ANSWER: A B D

Explanation:

“Like a servlet, a filter is also declared in the deployment descriptor.” is correct in the traditional Servlet deployment model: filters are configured using `<filter>` and `<filter-mapping>` elements in `web.xml`, which identifies the filter class and the requests, URL patterns, servlet names, or dispatcher types to which it applies. Modern Servlet applications can alternatively use annotation-based configuration, but deployment-descriptor declaration remains a valid and standard mechanism.

“The life cycle of a filter is managed by the container.” is correct because the Servlet container loads and instantiates configured filters, invokes initialization before use, calls the filter for matching requests, and invokes destruction when the filter is taken out of service. This container-managed lifecycle allows a filter to prepare shared resources during initialization and release them during shutdown.

“Every filter must implement the Filter interface.” is correct because the Jakarta Servlet API defines a filter as a component implementing `jakarta.servlet.Filter`. That interface provides the lifecycle contract through `init(FilterConfig)`, `doFilter(ServletRequest, ServletResponse, FilterChain)`, and `destroy()`. See the Jakarta Servlet Filter API documentation and the Jakarta Servlet specification’s Filters section.