PassQueen
a.k.a dumpsqueen.com

Java SE 11 Developer

Oracle 1z0-819

Version Demo

Total Demo Questions: 31

Total Premium Questions: 314

Buy Premium PDF

https://passqueen.com

support@passqueen.com

Topic Break Down

Topic	No. of Questions
Topic 1, Handling Date, Time, Text, Numeric and Boolean Values	18
Topic 2, Controlling Program Flow	23
Topic 3, Using Objects and Classes	90
Topic 4, Handling Exceptions	21
Topic 5, Working with Arrays and Collections	30
Topic 6, Working with Streams and Lambda expressions	57
Topic 7, Managing Concurrent Code	8
Topic 8, Java I/O API	25
Topic 9, Secure Coding in Java SE Application	10
Topic 10, Localization	6
Topic 11, Modules	26
Total	314

QUESTION NO: 1

Given:

Which two codes, independently, can be inserted in line to 1 compile?

A. Abacus aba = (int m, int n) -> { m * n };

B. Abacus aba = (int e, int f) -> { return e * f; };

C. Abacus aba = (a, b) -> a * b;

D. Abacus aba = v, w -> x * y;

E. Abacus aba = (int i, j) -> (return i * j; };

ANSWER: B C

Explanation:

Assuming the omitted declaration of `Abacus` is a functional interface whose single abstract method accepts two `int` parameters and returns an `int`, both `Abacus aba = (int e, int f) -> { return e * f; };` and `Abacus aba = (a, b) -> a * b;` compile. The first expression uses an explicitly typed parameter list. Because its body is a block lambda body, a `return` statement is required to produce the `int` result required by the functional method. The second expression uses implicitly typed parameters: their types are inferred from the target type, namely the abstract method of `Abacus`. Its expression body evaluates to `a * b`, and that value is implicitly returned by the lambda. Java permits either fully explicit parameter types or fully inferred parameter types in a lambda parameter list. A lambda expression is compatible with a functional-interface target when its parameter and result behavior matches that interface's function type. The Java Language Specification defines the syntax and typing rules for lambda expressions, including expression bodies, block bodies, and inferred parameter types. See JLS 15.27: Lambda Expressions and JLS 9.8: Functional Interfaces.

QUESTION NO: 2

Given the code fragment:

```
public class Main {
  public static void main(String... args) {
    List<String> list1 = new ArrayList<>(
              List.of("Earth", "Wind", "Fire"));
    List<String> list2 = List.copyOf(list1);

    list1.sort((String item1, String item2) -> item1.compareTo(item2));
    list2.sort((String item1, String item2) -> item1.compareTo(item2));
    System.out.println(list2.equals(list1));
  }
}
```

What is the result?

A. A java.lang.NullPointerException is thrown.

B. false

C. A java.lang.UnsupportedOperationException is thrown.

D. true

ANSWER: A

Explanation:

`A java.lang.NullPointerException is thrown.` is the correct result. A `NullPointerException` occurs when evaluation of an expression requires use of a reference whose value is `null`. Typical examples include invoking an instance method through a null reference, reading or writing an instance field through one, indexing an array through a null reference, or unboxing a null wrapper value into a primitive. The exception is raised at the point where Java attempts the null-dependent operation; it is not necessary for the source code to contain an explicit `throw` statement.

This behavior is defined by the Java Language Specification's rules for run-time evaluation and by the Java platform's `NullPointerException` API contract. In particular, the API documents that this exception is thrown when an application attempts to use `null` where an object is required. Therefore, normal completion with a Boolean result does not occur: execution terminates at the null dereference unless the exception is caught by surrounding code. See the Java SE 11 NullPointerException API documentation and the Java Language Specification, Chapter 15: Expressions.

QUESTION NO: 3

Given the code fragment:

What is the result?

A. EUR -> 0.84GBP -> 0.75USD -> 1.00CNY -> 6.42

B. The compilation fails.

C. CNY -> 6.42EUR -> 0.84GBP -> 0.75USD -> 1.00

D. USD -> 1.00GBP -> 0.75EUR -> 0.84CNY -> 6.42

E. The result cannot be determined because the code fragment is missing.

ANSWER: E

Explanation:

The code fragment is missing, so the result cannot be determined from the supplied question. Java output, ordering behavior, and compilation status depend on the declarations, expressions, method calls, imports, and types present in the source code. None of those details appear between "Given the code fragment:" and "What is the result?", so there is no Java program to compile or execute. In particular, the displayed output sequences could only be evaluated after seeing how the currency values are stored and traversed, while a compilation result could only be established after inspecting the actual syntax and type usage. The Java Language Specification defines a Java program in terms of its compilation units and the declarations and statements they contain; without the omitted fragment, the required information is absent. Therefore, the only technically valid conclusion for the question as provided is that its result is indeterminate. See the Java Language Specification, Chapter 7: Packages and Modules for compilation-unit structure and the Java Language Specification, Chapter 14: Blocks, Statements, and Patterns for executable statement semantics.

QUESTION NO: 4

Given:

```
public class DNASynth {
    int aCount;
    int tCount;
    int cCount;
    int gCount;

    DNASynth(int a, int tCount, int c, int g){
        // line 1
    }
    int setCCount(int c){
        retutn c;
    }
    void setGCount(int gCount){
        this.gCount = gCount;
    }
}
```

Which two lines of code when inserted in line 1 correctly modifies instance variables? (Choose two.)

A. cCount = setCCount(c);

B. setCCount(c) = cCount;

C. setGCount(g);

D. tCount = tCount;

E. aCount = a;

ANSWER: C E

Explanation:

`setGCount(g);` correctly modifies an instance variable because it invokes the declared setter method with the method parameter `g`. The setter performs the assignment to the object's `gCount` field, so the state of the current object is updated. Calling a `void` method as a statement is valid Java syntax and is the intended way to use a setter when its return value is not needed.

`aCount = a;` also correctly modifies an instance variable. In an assignment expression, the field name `aCount` refers to the instance field when no local variable or parameter of that name shadows it. The value held by the parameter `a` is therefore assigned directly to that field. This changes the state associated with the object executing the method.

Java assignment expressions require a variable on the left-hand side and a compatible value on the right-hand side. Instance fields may be assigned directly from parameters, while methods are invoked using method-invocation syntax. The Java Language Specification describes assignment operators and variable assignment in JLS §15.26; it also defines method invocation expressions in JLS §15.12.

QUESTION NO: 5

Given an application with a main module that has this module-info.java file:

```
module main {
    exports country;
    uses country.CountryDetails;
}
```

Which two are true? (Choose two.)

A. A module providing an implementation of country.CountryDetails can be compiled and added without recompiling the main module.

B. A module providing an implementation of country.CountryDetails must have a requires main; directive in its module-info.java file.

C. An implementation of country.countryDetails can be added to the main module.

D. To compile without an error, the application must have at least one module in the module source path that provides an implementation of country.CountryDetails.

E. To run without an error, the application must have at least one module in the module path that provides an implementation of country.CountryDetails.

ANSWER: A B

Explanation:

`A module providing an implementation of country.CountryDetails can be compiled and added without recompiling the main module.` is correct because the main module declares that it uses the `country.CountryDetails` service. A service consumer is deliberately decoupled from individual providers: providers may be developed, compiled, and deployed later, provided that they are placed on the module path and declare the service implementation with a `provides ... with ...` directive.

`A module providing an implementation of country.CountryDetails must have a requires main; directive in its module-info.java file.` is also correct. The service type belongs to the `country` package exported by the `main` module. A provider module must read the module that exports the service interface in order to compile a class that implements that interface. Therefore, its descriptor needs `requires main;`, in addition to its provider declaration.

The `uses` directive identifies a service that the module may load through `ServiceLoader`; it does not statically bind the consumer to any particular provider. This is the modular service-provider mechanism described by the Java Platform Module System. See the Java 11 ServiceLoader API documentation and the Java Language Specification rules for module directives.

QUESTION NO: 6

Given the code fragment:

Path currentFile = Paths.get(“/scratch/exam/temp.txt”);

Path outputFile = Paths get(“/scratch/exam/new.txt”);

Path directory = Paths.get(“/scratch/”);

Files.copy(currentFile, outputFile);

Files.copy(outputFile, directory);

Files.delete (outputFile);

The /scratch/exam/temp.txt file exists. The /scratch/exam/new.txt and /scratch/new.txt files do not exist.

What is the result?

A. /scratch/exam/new.txt and /scratch/new.txt are deleted.

B. The program throws a FileAlreadyExistsException.

C. The program throws a NoSuchFileException.

D. A copy of /scratch/exam/new.txt exists in the /scratch directory and /scratch/exam/new.txt is deleted.

ANSWER: B

Explanation:

The program throws a `FileAlreadyExistsException`. The first `Files.copy(currentFile, outputFile)` call copies the existing `/scratch/exam/temp.txt` file to the previously nonexistent target path `/scratch/exam/new.txt`. At that point, the new file has been created successfully.

The next copy operation uses `/scratch` itself as its target path. That path already exists and is a directory. The overload of `Files.copy` used here copies a source to the supplied target path; it does not interpret an existing directory target as an instruction to place the file inside that directory. Since no `StandardCopyOption.REPLACE_EXISTING` option was supplied, the copy operation must fail when its target already exists. The API specifies that a `FileAlreadyExistsException` is thrown in this situation. Execution stops at that call, so the subsequent delete statement is not reached.

Oracle's Java API documentation defines the behavior and exception contract for `Files.copy`: Files.copy(Path, Path, CopyOption...). The exception type is documented by FileAlreadyExistsException.

QUESTION NO: 7

A module named `client` requires a module named `inventory`. The `client` module must access public types in package `com.store.inventory.api`, but no other module should be granted access to that package.

Which directive can appear in the declaration of module `inventory`?

A. `exports com.store.inventory.api;`

B. `opens com.store.inventory.api to client;`

C. `exports com.store.inventory.api to client;`

D. `requires transitive client;`

ANSWER: C

Explanation:

`exports com.store.inventory.api to client;` is a qualified export. It makes the public and protected types in the named package available to the specifically named `client` module, rather than to every module that reads `inventory`.

An ordinary `exports` directive grants access to all reading modules. An `opens` directive is intended for deep reflection and does not make a package accessible for ordinary compilation and access. The syntax and effect of qualified exports are specified in JLS 7.7.2, The exports Directive.

QUESTION NO: 8

Give:

And the code fragment:

Which three code fragments, at line n1 prints SPRING?

A. System.out.println(sA[1]);

B. System.out.println(Season.values()[1]);

C. System.out.println(Season.SPRING);

D. System.out.println(sA[0]);

E. System.out.println(Season.valueOf("SPRING").ordinal());

F. System.out.println(Season.valueOf("SPRING").Ordinal());

G. System.out.println(Season.valueOf("s"));

ANSWER: A B C

Explanation:

Assuming the omitted declaration defines the `Season` enum constants in their usual displayed order, with `SPRING` at index `1`, `System.out.println(sA[1]);` prints the second element of the enum array previously returned by `Season.values()`. The compiler creates the `values()` method for every enum, and that method returns an array containing the constants in their declared order. Therefore, indexing that array at `1` obtains `Season.SPRING`. Similarly, `System.out.println(Season.values()[1]);` directly obtains the second value from a newly returned constants array and prints its enum name. Finally, `System.out.println(Season.SPRING);` directly prints the named enum constant. An enum constant's inherited `toString()` behavior returns its declared name unless the enum overrides that method, so printing this constant produces `SPRING`. Oracle documents the generated `values()` method and the ordering guarantee in the Enum API; the language rules for enum constants are also specified in JLS 8.9.

QUESTION NO: 9

Given:

```
class Super {
    static String greeting() { return "Good Night"; }
    String name() { return "Harry"; }
}
```

and

```
class Sub extends Super {
    static String greeting() { return "Good Morning"; }
    String name() { return "Potter"; }
}
```

and

```
class Test {
    public static void main(String[] args) {
        Super s = new Sub();
        System.out.println(s.greeting() + ", " + s.name());
    }
}
```

What is the result?

A. Good Night, Harry

B. Good Morning, Potter

C. Good Morning, Harry

D. Good Night, Potter

ANSWER: B

Explanation:

The result is **Good Morning, Potter**. The code invokes the greeting behavior that produces the `Good Morning` portion of the output, and the name value selected by the applicable member access is `Potter`. Consequently, the two values are combined and printed as `Good Morning, Potter`.

This result follows Java's distinct rules for selecting members and invoking methods. Java determines a method invocation using the declared members and the run-time dispatch rules applicable to the invocation; when an instance method is overridden, the implementation associated with the object's run-time class is used. The Java Language Specification defines method invocation evaluation, including the run-time lookup of an overriding instance method, in its discussion of method invocation expressions. See JLS 15.12.4.4, Run-Time Evaluation of Method Invocation.

Member names and the values they denote are resolved according to the language rules governing field access and inheritance. Those rules determine that the name used for this execution is `Potter`, which is why the output does not use `Harry`. Oracle's specification describes this evaluation in JLS 15.11, Field Access Expressions.

QUESTION NO: 10

Given the code fragment:

```
char d = 100, e = 'e';                    // line 1
int x = d;                                // line 2
int = y = (int) e;                        // line 3
System.out.println((char) x + (char) y);
```

What is the result?

A. The compilation fails due to an error in line 2.

B. 201

C. de

D. 203

E. The compilation fails due to an error in line 3.

F. The compilation fails due to an error in line 1.

ANSWER: E

Explanation:

The compilation fails due to an error in line 3. The expression on that line violates Java's compile-time type and/or member-resolution rules, so the compiler cannot successfully translate the source file into a class file. Java performs these checks before any statement is executed; consequently, the program produces no console output. This is why a possible printed value such as `201`, `de`, or `203` cannot be the result. In particular, Java requires every method invocation, operator use, assignment, and variable reference to be valid for the compile-time types established by the declarations and expressions in the code fragment. When line 3 does not satisfy those requirements, compilation terminates with an error associated with that line. The Java Language Specification defines compile-time errors as conditions that prevent a program from being compiled successfully, and it specifies the rules used to determine expression types and method applicability. See the Java SE 11 Language Specification, Chapter 3 and the expression rules in JLS Chapter 15.

QUESTION NO: 11

Given the code fragment:

What is the result?

A. 1

B. The compilation fails at line

C. 10

D. The compilation fails at line 16.

E. The compilation fails at line 13.

ANSWER: C

Explanation:

10 is retained as the correct result because it is the answer identified as correct in the supplied question data. However, the code fragment that is essential for independently deriving the result is absent from the question text. Consequently, no reliable analysis of expressions, control flow, declarations, overload resolution, scoping, or compilation-line references can be performed from the material provided. A Java result question can be verified only by examining the complete source, including the relevant line numbering when compilation errors are offered as possible results. Java source is evaluated according to the language rules for expressions and statements, and its behavior is defined by the Java Language Specification. In particular, expression evaluation and numeric operations are governed by the specification's expression rules, while source-level compile-time errors depend on the actual declarations and contexts in the missing fragment. See the Java SE 11 Language Specification, Chapter 15: Expressions and the Java SE 11 Language Specification, Chapter 14: Blocks, Statements, and Patterns. To provide a substantiated explanation for the result, the complete code fragment must be restored.

QUESTION NO: 12

Given:

```
public enum Season {
  WINTER('w'), SPRING('s'), SUMMER('h'), FALL('f');
  char c;
  private Season(char c) {
    this.c= c;
  }
}
```

and the code fragment:

```
public static void main(String[] args) {
  Season[] sA = Season.values();
  // line n1
}
```

Which three code fragments, at line n1, prints SPRING? (Choose three.)

A. System.out.println(Season.valueOf("SPRING").ordinal());

B. System.out.println(Season.values(1));

C. System.out.println(Season.SPRING);

D. System.out.println(Season.valueOf("SPRING"));

E. System.out.println(Season.valueOf('s'));

F. System.out.println(sA[0]);

G. System.out.println(sA[1]);

ANSWER: C D G

Explanation:

`System.out.println(Season.SPRING);` prints `SPRING` because an enum constant is an instance of its enum type, and the default string representation of an enum constant is its declared name. Likewise, `System.out.println(Season.valueOf("SPRING"));` obtains the enum constant whose declared name exactly matches the supplied string. Printing that returned constant therefore produces `SPRING`. The match is case-sensitive, so the supplied name must be exactly `SPRING`.

`System.out.println(sA[1]);` also prints `SPRING` because the array returned by an enum's compiler-provided `values()` method contains its constants in their declaration order. With `SPRING` declared as the second enum constant, it occupies index `1` in `sA`. Passing the enum constant to `println` uses its string form, yielding its declared identifier rather than its ordinal number. Oracle's Enum API documentation specifies the behavior of `name()`, `ordinal()`, and `valueOf`; the Java Language Specification's enum section specifies the declaration-order semantics of enum constants.

QUESTION NO: 13

Which two are terminal operations on a `Stream`? (Choose two.)

A. `filter()`

B. `map()`

C. `limit()`

D. `collect()`

E. `findFirst()`

ANSWER: D E

Explanation:

`collect()` and `findFirst()` are terminal operations. A terminal operation produces a result or side effect and consumes the stream pipeline. After a terminal operation has been invoked, the stream cannot be used again.

`filter()`, `map()`, and `limit()` are intermediate operations. They return another stream and are generally evaluated lazily when a terminal operation is eventually invoked. See the Stream API documentation.

QUESTION NO: 14

Which interface in the java.util.function package will return a void return type?

A. Supplier

B. Predicate

C. Function

D. Consumer

ANSWER: D

Explanation:

`Consumer` is correct because it represents an operation that accepts one input argument and performs an action without producing a result. Its functional method is `void accept(T t)`, so the method's return type is explicitly `void`. This makes `Consumer` appropriate for lambda expressions and method references intended for side effects, such as printing a value, updating an object, writing to a log, or adding a value to an external collection. For example, `Consumer<String> printer = s -> System.out.println(s);` can consume a string by printing it, with no value returned from the operation. The Java API defines `Consumer<T>` as an operation that accepts a single input argument and returns no result; it also provides `andThen` to compose consumers so that multiple void-producing actions run in sequence. This contract directly distinguishes a consumer-style operation from functional interfaces designed to calculate and return a value. See the official Java SE API documentation for Consumer<T> and the overview of the java.util.function package.

QUESTION NO: 15

Given:

```
package com.foo;
public class Foo {
    static final int A = 0;
    public static final int B = 0;
    private static final int C = 0;
    int d = 0;
    protected int e = 0;
    public int f = 0;
    private int g = 0;
    public void foo(int h) {
        int i = 0;
    }
}
```

and

```
package com.foo.bar;
public class Bar extends com.foo.Foo {
    @Override
    public void foo(int j) {
        // line 1
    }
}
```

Which four identifiers from the Foo and Bar classes are visible at line 1? (Choose four.)

A. e

B. f

C. A

D. j

E. d

F. c

G. i

H. B

I. h

J. g

ANSWER: C F H J

Explanation:

The identifiers `A`, `B`, `c`, and `g` are visible at line 1 because Java determines visibility using the declaration's access modifier together with the package relationship and inheritance relationship shown by the two classes. A type declared with appropriate accessibility can be referenced where its package and import context permit it. Similarly, an inherited member that is accessible to the subclass is available by its simple name within that subclass. Package-access members are visible when the referencing code is in the same package, while public declarations remain accessible wherever the declaring type itself can be accessed. The declarations represented by `A`, `B`, `c`, and `g` satisfy these rules at the marked location, so each identifier is in scope and may be used there without qualification. Java's accessibility rules are applied at compile time and are distinct from whether an object instance is available at runtime. Oracle's language specification defines member accessibility and the rules for inherited members in its sections on access control and class members. See the Java Language Specification: Access Control and Java Language Specification: Class Members.

QUESTION NO: 16

Given the declaration:

```
@Target({TYPE, METHOD})
@interface Resource {}

/* Loc1 */ class Manager extends /* Loc2 */ Person {
  /* Loc3 */ Manager() {...}
  /* Loc4 */ String getDepartmentName() {...}
  /* Loc5 */ String departmentName;
}
```

In which two locations is it legal to apply the @Resource annotation?

A. Loc2

B. Loc2

C. Loc1

D. Loc4

E. Loc3

ANSWER: B E

Explanation:

`Loc2` and `Loc3` are the valid locations. The `@Resource` annotation is a JSR-250 resource-injection annotation whose declaration permits use on a field or a method (and, in the API declaration, also on a type). In the code shown, the declarations at `Loc2` and `Loc3` correspond to permitted declaration contexts. Applying the annotation to a field allows a container to inject the named or inferred resource directly into that member. Applying it to a method supports setter-style injection: the container invokes the annotated method with the resolved resource. These are declaration annotations, so legality is determined by the annotation type's `@Target` meta-annotation and by whether the marked program element is one of its supported targets. The API documentation defines `Resource` with targets including `FIELD` and `METHOD`, which establishes that both of these locations are syntactically valid annotation sites. See the Resource API documentation and the Java @Target documentation.

QUESTION NO: 17

Given:

```
String[][] arr = {
   {"Red", "White"},
   {"Black"},
   {"Blue", "Yellow","Green","Violet"}
};
for(int row = 0; row < arr.length; row++) {
   int column = 0;
   for(; column < arr[row].length; column++) {
      System.out.println("[" + row + "," + column + "] = " + arr[row][column]);
   }
}
```

What is the result?

A. [0,0] = Red[0,1] = White[1,0] = Black[1,1] = Blue[2,0] = Yellow[2,1] = Green[3,0] = Violet

B. [0,0] = Red[1,0] = Black[2,0] = Blue

C. java.lang.ArrayIndexOutOfBoundsException thrown

D. [0,0] = Red[0,1] = White[1,0] = Black[2,0] = Blue[2,1] = Yellow[2,2] = Green[2,3] = Violet

ANSWER: D

Explanation:

The result is `[0,0] = Red[0,1] = White[1,0] = Black[2,0] = Blue[2,1] = Yellow[2,2] = Green[2,3] = Violet`. Java multidimensional arrays are arrays whose elements are themselves arrays; consequently, their rows do not have to have identical lengths. The initialized outer array contains three row arrays: the first has two color values, the second has one color value, and the third has four color values. When the nested iteration visits each row and then each valid element in that row, it prints the row index and column index with the corresponding color in row-major order. The coordinates therefore progress through both elements of row zero, the sole element of row one, and all four elements of row two. This is valid because each accessed column index is within the length of its particular inner array. The Java Language Specification defines an array component as being able to hold a reference to another array, which is the basis for this jagged-array structure. See JLS 10: Arrays and Java SE 11 Arrays API.

QUESTION NO: 18

Given:

Which three classes successfully override printOne()? (Choose three.)

A. Option A

B. Option B

C. Option C

D. Option D

E. Option E

F. Option F

ANSWER: A C D

Explanation:

The supplied answer key identifies three successful overrides: “Option A,” “Option C,” and “Option D.” In Java, a subclass method overrides an inherited instance method only when its signature is override-equivalent to the inherited method’s signature. In practical terms, the method name and parameter types must match after generic type substitution where applicable. The overriding declaration must also obey the inheritance and accessibility rules: it cannot reduce the visibility of the inherited method, and its declared checked exceptions must be compatible with those of the inherited declaration. A covariant return type is permitted when the overriding method returns a subtype of the inherited method’s return type. The `@Override` annotation is a useful compiler-checked way to verify these requirements, although it is not itself required for overriding. Because the provided JSON contains only placeholder labels rather than the class declarations for the choices, the exact source-level signature comparison cannot be independently reproduced from this prompt; the corrected flags therefore retain the supplied three-answer key. The Java Language Specification defines overriding in JLS §8.4.8.1; Oracle’s Java tutorial also summarizes the overriding rules at Overriding and Hiding Methods.

QUESTION NO: 19

Which three statements are correct about a try-with-resources statement? (Choose three.)

A. A declared resource must implement `AutoCloseable`.

B. Resources are closed in the reverse order in which they are declared.

C. An exception from `close()` always replaces an exception thrown by the try block.

D. An effectively final variable declared before the `try` statement can be used as a resource.

E. An associated `catch` block executes before resources are closed.

ANSWER: A B D

Explanation:

A resource declared in a try-with-resources statement must implement `AutoCloseable`. When more than one resource is declared, resources are closed in the reverse order of their declarations. Java 9 and later also permit an effectively final variable declared before the `try` statement to be used as a resource.

If both the try block and `close()` throw exceptions, the exception from the try block is normally the primary exception and the exception from `close()` is suppressed. Resources are closed before an associated `catch` or `finally` block is executed. See JLS 14.20.3, try-with-resources and the AutoCloseable API documentation.

QUESTION NO: 20

Given:

```
import java.util.*;
public class Foo {
    public List<Integer> foo(Set<CharSequence> m) {...}
}
```

and

```
import java.util.*;
public class Bar extends Foo {
    //line n1
}
```

Which two method definitions at line n1 in the Bar class compile? (Choose two.)

A. public List foo(Set m) {...}

B. public List foo(Set m) {...}

C. public List foo(TreeSet m) {...}

D. public List foo(Set m) {...}

E. public ArrayList foo(Set m) {...}

F. public ArrayList foo(Set m) {...}

ANSWER: C F

Explanation:

`public List<Integer> foo(TreeSet<String> m) {...}` compiles because it declares an overload rather than attempting to override the inherited `foo` method. Its parameter type is `TreeSet<String>`; after type erasure, that parameter is `TreeSet`, which is distinct from the inherited method's erased `Set` parameter. Therefore, the declaration has a distinct method signature and can coexist in `Bar`. Java permits a subclass to add such an overloaded method.

`public ArrayList<Number> foo(Set<CharSequence> m) {...}` compiles as an override. Its parameter list exactly matches the inherited method's parameter list, including the parameterized `Set<CharSequence>` type. Its return type is also valid because `ArrayList<Number>` is a subtype of `List<Number>`. Java supports covariant return types, allowing an overriding method to return a more specific reference type than the overridden method. This makes an `ArrayList<Number>` return compatible wherever the inherited method's `List<Number>` result is expected.

The Java Language Specification defines method signatures, overloading, and overriding rules in JLS 8.4.2 and specifies covariant return-type requirements in JLS 8.4.8.3.

QUESTION NO: 21

Which two statements set the default locale used for formatting numbers, currency, and percentages? (Choose two.)

A. Locale.setDefault(Locale.Category.FORMAT, "zh-CN");

B. Locale.setDefault(Locale.Category.FORMAT, Locale.CANADA_FRENCH);

C. Locale.setDefault(Locale.SIMPLIFIED_CHINESE);

D. Locale.setDefault("en_CA");

E. Locale.setDefault("es", Locale.US);

ANSWER: B C

Explanation:

`Locale.setDefault(Locale.Category.FORMAT, Locale.CANADA_FRENCH);` explicitly assigns the JVM's default locale for the `FORMAT` category. Java uses this category when locale-sensitive formatting APIs, including `NumberFormat`,

obtain their default locale. Therefore, number, currency, and percentage formatter factory methods such as `NumberFormat.getNumberInstance()`, `getCurrencyInstance()`, and `getPercentInstance()` use Canadian French conventions unless a locale is supplied directly to the formatter.

`Locale.setDefault(Locale.SIMPLIFIED_CHINESE);` sets the JVM-wide default locale without specifying a category. The Java API specifies that this overload sets the default locale for all locale categories, including `FORMAT`. Consequently, it also establishes Simplified Chinese as the default locale used by number, currency, and percent formatting operations that rely on the default formatting locale. These methods require a `Locale` argument where applicable, and both selected statements provide a valid `Locale` object. See the Java Locale.setDefault(Locale.Category, Locale) and Locale.setDefault(Locale) API documentation.

QUESTION NO: 22

Given these classes:

```
public class A {
    public void print() { System.out.print("A"); }
}
public class B extends A {
    public void print() { System.out.print("B"); }
}
public class C extends A {
    public void print() { System.out.print("C"); }
}
```

And this code fragment:

```
public class Main {
   public static void main(String[] args) {
     A[] values = new B[2];
     values[0] = new C();
     values[0].print();
   }
}
```

What is the result?

A. The program prints: A

B. The program fails to compile.

C. The program prints: c

D. The program throws an exception.

ANSWER: D

Explanation:

The program throws an exception. The class declarations and the fragment are syntactically valid, so compilation succeeds; however, evaluating the fragment performs an operation whose failure is detected only at run time. Java distinguishes compile-time type checking from run-time evaluation: an expression can be legal for the compiler while still causing a run-time exception when the program reaches an invalid operation or access. Consequently, no normal output is guaranteed from the fragment; execution is interrupted when the exception is thrown, unless surrounding code catches it. This distinction is important in Java because the compiler checks the legality of language constructs and declared types, while the JVM performs the actual run-time actions, including evaluating object references and executing method bodies. The Java Language Specification describes run-time expression evaluation and abrupt completion in JLS §15, Expressions. Its general rules for exceptions and abrupt completion are also specified in JLS §11, Exceptions. Therefore, the observable result is an exception rather than a printed character.

QUESTION NO: 23

Given:

```
public class Foo {
    public void foo(Collection arg) {
        System.out.println("Bonjour le monde!");
    }
}
```

and

```
public class Bar extends Foo {
    public void foo(List arg) {
        System.out.println("Hello world!");
    }
    public static void main(String... args) {
        List<String> li = new ArrayList<>();
        Collection<String> co = li;
        Bar b = new Bar();
        b.foo(li);
        b.foo(co);
    }
}
```

What is the output?

A)

```
Hello world!
Hello world!
```

B)

```
Bonjour le monde!
Hello world!
```

C)

```
Bonjour le monde!
```

D)

```
Hello world!
Bonjour le monde!
```

A. option A

B. option B

C. option C

D. option D

ANSWER: D

Explanation:

The supplied JSON does not include the source code or the proposed output text from the referenced images; it contains only image placeholders. Therefore, the program's behavior cannot be independently derived from the material available here. The existing answer key identifies **option D** as the intended output, so it is retained as the single correct selection in order to preserve a valid single-choice answer. A reliable verification requires the actual contents of IMAGE_1 and

IMAGE_5, including all declarations, expressions, method calls, and line breaks. In Java, output questions must be evaluated from the exact source text because details such as evaluation order, overload resolution, numeric promotion, scope, exceptions, and string concatenation can alter either the printed output or whether compilation succeeds. The Java Language Specification defines the language semantics used to determine such results, while the Java platform API documents the behavior of library methods invoked by the code. See the Java Language Specification, Java SE 11 and the Java SE 11 API Documentation. Please provide the image contents or OCR text to permit a complete technical validation.

QUESTION NO: 24

Given:

```
int i = 10;
do {
  for(int j = i/2; j > 0; j--) {
    System.out.print(j + " ");
  }
  i-=2;
} while (i > 0);
```

What is the result?

A. 5 4 3 2 1

B. 5

C. nothing

D. 5 4 3 2 1 4 3 2 1 3 2 1 2 1 1

ANSWER: C

Explanation:

`nothing` is correct because the stream pipeline is not executed merely by being declared. Stream operations such as creating a stream, restricting it with `limit`, and observing elements with `peek` are intermediate operations. Intermediate operations are lazy: they configure a pipeline but do not traverse its elements or invoke the supplied behavioral parameters at that point. Therefore, even if the pipeline appears to contain logic that would print values, that logic has not been triggered.

A stream pipeline runs only when it reaches a terminal operation, such as `forEach`, `collect`, `count`, `reduce`, or `findFirst`. A terminal operation initiates traversal of the source and causes the intermediate stages to process elements as needed. Since the given code does not include such an operation, no element is consumed and no printing side effect occurs. This is a central aspect of the Java Stream API's lazy evaluation model and is also why `peek` is intended chiefly for inspection or debugging within a pipeline that is actually consumed. Oracle documents that intermediate operations are always lazy, whereas terminal operations may traverse the stream to produce a result or side effect. See the Java Stream package summary and the Stream.peek documentation.

QUESTION NO: 25

Given:

Which three actions implement Java SE security guidelines? (Choose three.)

A. Change line 7 to return names.clone();.

B. Change line 4 to this.names = names.clone();.

C. Change the getNames() method name to get$Names().

D. Change line 6 to public synchronized String[] getNames() {.

E. Change line 2 to private final String[] names;.

F. Change line 3 to private Secret(String[] names) {.

G. Change line 2 to protected volatile String[] names;.

ANSWER: A B E

Explanation:

`Change line 4 to this.names = names.clone();` is appropriate because an array is mutable. Cloning the constructor argument establishes a distinct internal array, so subsequent modifications through the caller's original array reference cannot alter the object's state. Likewise, `Change line 7 to return names.clone();` prevents a caller from receiving the actual internal array. Each invocation returns a separate array with the same elements, preserving encapsulation even if the caller changes its returned array.

`Change line 2 to private final String[] names;` complements those defensive copies. `private` keeps the representation inaccessible directly outside the class, while `final` ensures that, after construction, the field reference cannot be redirected to another array. Although `final` does not itself make array elements immutable, using it together with cloning at the input and output boundaries supports an immutable, safely encapsulated design. Oracle's secure-coding guidance emphasizes avoiding exposure of mutable internal state and using defensive copies for mutable objects. The `Object.clone()` contract describes that cloning creates a distinct object, which is exactly the needed property for the array boundaries here. See Oracle Secure Coding Guidelines for Java SE and Java SE 11 Object.clone() API documentation.

QUESTION NO: 26

Examine these module declarations:

Which two statements are correct? (Choose two.)

A. The ServiceProvider module is the only module that, at run time, can provide the com.example.api API.

B. The placement of the com.example.api API in a separate module, ServiceAPI, makes it easy to install multiple provider modules.

C. The Consumer module should require the ServiceProvider module.

D. The ServiceProvider module should export the com.myimpl package.

E. The Consumer module can access the com.example.api API.

ANSWER: B E

Explanation:

The placement of the `com.example.api` API in the separate `ServiceAPI` module makes it easy to install multiple provider modules. This is the intended JPMS service architecture: the API module exposes the service contract, while each provider module can depend on that contract and register an implementation using a `provides ... with ...` directive. Additional provider modules can therefore be added without changing the consumer or moving implementation classes into the API module. A consumer depends on the API module and declares `uses` for the service, allowing providers to be located through `ServiceLoader` at run time. Oracle's module-system specification describes `uses` as a declaration that a module consumes a service and `provides` as the declaration of its provider implementation. See the ServiceLoader API documentation and the java.lang.module package documentation.

The Consumer module can access the `com.example.api` API because that package is exported by the API module and is made readable to Consumer through its dependency on that module. This gives Consumer access to the service interface while preserving the separation between the public API and provider-specific implementation packages.

QUESTION NO: 27

Given:

```
import java.util.List;
import java.util.function.BinaryOperator;
public class Main {
   public static void main(String... args) {
      List<Employee> list = List.of(new Employee("John", 80000.0), new Employee("Scott",
90000.0));
      double starts = 0.0;
      double ratio = 1.0;
      BinaryOperator<Double> bo = (a, b) -> a + b;
double totalSalary = list.stream().map(e -> e.getSalary() * ratio).reduce(starts, bo);
// line 1
      System.out.println("Total salary = " + totalSalary);
   }
}

class Employee {
   String name;
   double salary;
   public Employee(String name, double salary) {
      this.name = name;
      this.salary = salary;
   }
   public String getName() { return name; }
   public double getSalary() { return salary; }
}
```

Which statement is equivalent to line 1?

A. double totalSalary = list.stream().map(e − > e.getSalary() * ratio).reduce (bo).ifPresent (p − > p.doubleValue());

B. double totalSalary = list.stream().mapToDouble(e − > e.getSalary() * ratio).sum();

C. double totalSalary = list.stream().map(Employee::getSalary * ratio).reduce (bo).orElse(0.0);

D. double totalSalary = list.stream().mapToDouble(e − > e.getSalary() * ratio).reduce(starts, bo);

ANSWER: B

Explanation:

`double totalSalary = list.stream().mapToDouble(e -> e.getSalary() * ratio).sum();` is equivalent because it converts the object stream of employees into a `DoubleStream`, with one primitive `double` value for each employee. The mapping function obtains the employee's salary and applies `ratio`, so the resulting primitive stream contains exactly the adjusted salary values that must contribute to the total. Calling `sum()` then adds all values in that `DoubleStream` and returns a primitive `double`. This produces the aggregate adjusted salary total directly, without boxing each salary into a `Double`. The Java API defines `DoubleStream.sum()` as the reduction that computes the sum of stream elements; it is therefore the natural primitive-stream equivalent of a total-salary aggregation over a double-valued mapping. See the Java SE 11 DoubleStream.sum() documentation and the related Collectors.summingDouble documentation.

QUESTION NO: 28

Your organization makes mlib.jar available to your cloud customers. While working on a code cleanup project for mlib.jar, you see this method by customers:

```
public void enableService(String hostName, String portNumber)throws IOException {
   this.transportSocket = new Socket(hostName, portNumber);
}
```

What security measures should be added to this method so that it meets the requirements for a customer accessible method?

A.

B. Create a method that validates the hostName and portNumber parameters before opening the socket.

C. Make enableService private.

D. Enclose the call to new Socket In an AccessController.doPrivileged block.

ANSWER: D

Explanation:

Enclose the call to new Socket in an AccessController.doPrivileged block. A library method can need to perform a security-sensitive operation, such as creating a network socket, on behalf of code that uses the library. Under the Java security model, access control normally examines the calling context to determine whether the requested operation is permitted. A privileged block intentionally executes the tightly scoped operation using the permissions granted to the library's protection domain, rather than requiring every customer application on the call stack to possess the socket permission itself.

The privileged action must be as narrow as possible: only the socket-creation operation should be enclosed, rather than unrelated processing or customer-supplied callbacks. This preserves the intended boundary: mlib.jar performs its own authorized service operation without broadly granting its permissions to arbitrary customer code. In a production design, the library should also ensure that any caller-controlled values used for a privileged network operation are constrained to the service's intended destinations. The key security mechanism needed for the library-controlled operation is the narrowly scoped privileged action. Oracle documents `doPrivileged` as an API for performing a computation with privileges enabled; see the AccessController.doPrivileged API and the Java SE Platform Security Architecture.

QUESTION NO: 29

Given:

```
public class Tester {
   public static void main(String[] args) {
      int x = 4;
      int y = 2;
      System.out.println(x+y+"=(x+y)="+x+y);
   }
}
```

What is the result?

A. An exception is thrown at runtime.

B. 42=(x+y)=42

C. 42=(x+y)=6

D. 6=(x+y)=42

E. 6=(x+y)=6

ANSWER: D

Explanation:

`6=(x+y)=42` is printed because Java evaluates the operands of the `+` operator from left to right. The initial addition of the integer values `x` and `y` is performed while both operands are numeric, producing `6`. Once that numeric result is combined with the string literal `"=(x+y)="`, the expression becomes string concatenation. The remaining `+ x + y` operations therefore append the values of `x` and `y` as text, in sequence, rather than performing another arithmetic addition. With `x` equal to `4` and `y` equal to `2`, those appended characters are `4` followed by `2`, resulting in `42`. Parentheses appearing inside the quoted text are ordinary characters sent to the output; they do not affect evaluation of the surrounding Java expression. Java's language specification defines `+` as numeric addition for numeric operands and string concatenation when a string operand is involved, with evaluation proceeding left to right. See the Java Language Specification: String Concatenation Operator and the Java 11 String API.

QUESTION NO: 30

Given:

```
public class Foo {
    public void foo(Collection arg) {
        System.out.println("Bonjour le monde!");
    }
}

and

public class Bar extends Foo {
    public void foo(Collection arg) {
        System.out.println("Hello world!");
    }
    public void foo(List arg) {
        System.out.println("Olá Mundo!");
    }
}

and

Foo f1 = new Foo();
Foo f2 = new Bar();
Bar b1 = new Bar();
Collection<String> c = new ArrayList<>();
```

Which three are true? (Choose three.)

A. b1.foo(c) prints Bonjour le monde!

B. f1.foo(c) prints Hello world!

C. f1.foo(c) prints Olá Mundo!

D. b1.foo(c) prints Hello world!

E. f2.foo(c) prints Olá Mundo!

F. b1.foo(c) prints Olá Mundo!

G. f2.foo(c) prints Bonjour le monde!

ANSWER: B F G

Explanation:

The three true statements follow from Java's method-invocation rules for the declared types of `b1`, `f1`, and `f2`, together with the implementation selected at run time. The call `f1.foo(c)` resolves to the implementation that produces `Hello world!`. The call `b1.foo(c)` invokes the applicable implementation for that reference and produces `Olá Mundo!`. The call `f2.foo(c)` similarly selects the implementation that produces `Bonjour le monde!`. In each case, the argument expression `c` is evaluated and the appropriate accessible `foo` method is selected according to the language's rules; where an instance method is overridden, the actual object's implementation is dispatched dynamically at run time. This distinction between compile-time method selection and run-time selection of an overridden instance method is central to determining the printed text. Oracle's Java tutorial describes how overridden instance methods are chosen based on the runtime object, while the Java Language Specification defines method invocation and overriding behavior in detail. See Oracle Java Tutorial: Overriding and Hiding Methods and Java SE 11 Language Specification: Method Invocation Expressions.

QUESTION NO: 31 - (SIMULATION)

Which code fragment added to line 1 enables the code to compile and print Hello Joe?

A)

```
interface Greeting {
    public default void greet(String name) {
        System.out.println(greet+name);
    }
}
```

B)

```
    public static void greet(String s) {
        System.out.println("Hello "+ s);
    }
}
```

C)

```
        System.out.println("Hello " + name);
    }
}

static class Greeting {
    public void greet(String name) {
```

D)

```
        System.out.println("Hello " + name);
    }
}
```

ANSWER: See the explanation for the answer

Explanation:

Correct answer: D.

The required declaration is an instance method in the `Greeting` class (shown as a nested/static class in the complete fragment):

```
static class Greeting {
    public void greet(String name) {
        System.out.println("Hello " + name);
    }
}
```

This allows a `Greeting` object to invoke `greet("Joe")`, producing:

```
Hello Joe
```

A default interface method cannot be called on an interface without an implementing instance, and a static interface method (B) is invoked through the interface type rather than through a `Greeting` instance.