DUMPSARENA

Java SE 8 Programmer II

Oracle 1z0-809

Version Demo

Total Demo Questions: 28

Total Premium Questions: 289

Buy Premium PDF

https://dumpsarena.co

sales@dumpsarena.co

sales@dumpsarena.co
dumpsarena.co

Topic Break Down

Topic	No. of Questions
Topic 1, Java Class Design	38
Topic 2, Advanced Java Class Design	5
Topic 3, Generics and Collections	23
Topic 4, Lambda Built-In Functional Interfaces	37
Topic 5, Java Stream API	56
Topic 6, Exceptions and Assertions	23
Topic 7, Use Java SE 8 Date/Time API	13
Topic 8, Java IO Fundamentals	15
Topic 9, Java File I/O (NIO.2)	24
Topic 10, Java Concurrency	23
Topic 11, JDBC	27
Topic 12, Mix Questions	5
Total	289

QUESTION NO: 1

Given:

```
interface P { public void method1(); }
interface Q extends P {  public void method1(); }
interface R extends P { public void method2();}
interface S { public default void method() { } }
interface T { public void method1();  public void method2(); }
interface U { public void method1();  public abstract void method2(); }
```

Which two interfaces can you use to create lambda expressions? (Choose two.)

A. T

B. R

C. P

D. S

E. Q

F. U

ANSWER: C D

Explanation:

`P` and `S` can be used as target types for lambda expressions because each is a functional interface: after considering its inherited members, it has exactly one abstract method contract. A lambda expression supplies the implementation of that single abstract operation, and Java determines the lambda's parameter types, return requirements, and permitted thrown exceptions from that operation's function type.

The functional-interface test is based on abstract methods rather than merely counting all declarations. Default and static interface methods do not add abstract-method requirements. Also, methods that match public instance methods of `Object`, such as `equals`, do not count toward the single abstract method. Where inherited abstract declarations have the same signature or represent one overriding method contract, they likewise correspond to one functional operation. Thus, the declarations and inherited method structure of `P` and `S` leave one eligible abstract method for a lambda to implement.

The `@FunctionalInterface` annotation can be used to request compiler validation, but it is not required for an interface to be a valid lambda target. See the Java Language Specification sections on functional interfaces and lambda expressions.

QUESTION NO: 2

Given the code fragment:

9. Connection conn = DriveManager.getConnection(dbURL, userName, passWord);

10. String query = “SELECT id FROM Employee”;

11. try (Statement stmt = conn.createStatement()) {

12. ResultSet rs = stmt.executeQuery(query);

13. stmt.executeQuery(“SELECT id FROM Customer”);

14. while (rs.next()) {

15. //process the results

16. System.out.println(“Employee ID: “+ rs.getInt(“id”));

17. }

18. } catch (Exception e) {

19. System.out.println (“Error”);

20. }

Assume that:

The required database driver is configured in the classpath.

The appropriate database is accessible with the dbURL, userName, and passWord exists.

The Employee and Customer tables are available and each table has id column with a few records and the SQL queries are valid.

What is the result of compiling and executing this code fragment?

A. The program prints employee IDs.

B. The program prints customer IDs.

C. The program prints Error.

D. compilation fails on line 13.

E. Compilation fails on line 9 because `DriveManager` cannot be resolved.

ANSWER: E

Explanation:

Compilation fails on line 9 because `DriveManager` is not the JDBC API class that supplies `getConnection`. The standard JDBC class is `java.sql.DriverManager`, with the spelling `DriverManager`. Assuming no separately declared class named `DriveManager` is available in scope, the compiler cannot resolve the identifier and reports a cannot-find-symbol error before the program can execute. Configuring a database driver on the class path does not create or rename Java API classes; it only makes the JDBC driver implementation available for use by the correctly named JDBC facilities. Oracle’s Java SE 8 API specifies `DriverManager` as the class responsible for managing JDBC drivers and obtaining database connections through its `getConnection` methods. Therefore, the fragment must be corrected to use `DriverManager.getConnection(...)` before any database activity, exception handling, or result processing can occur. See the Java SE 8 DriverManager API and the Java Language Specification rules for names and scope.

QUESTION NO: 3

Given:

Message.properties:

msg = Welcome!

Message_fr_FR.properties:

msg = Bienvenue!

Given the code fragment:

// line n1

Locale.setDefault(locale);

ResourceBundle bundle = ResourceBundle.getBundle(" Message ");

System.out.print(bundle.getString(" msg "));

Which two code fragments, when inserted at line n1 independently, enable to print Bienvenue!?

A. Locale locale = new Locale(" fr-FR ");

B. Locale locale = Locale.FRANCE;

C. Locale locale = new Locale (" fr " , " FR ");

D. Locale locale = new Locale (" FRANCE " , " FRENCH ");

E. Locale locale = Locale.forLanguageTag(" fr ");

ANSWER: B C

Explanation:

`Locale locale = Locale.FRANCE;` is correct because `Locale.FRANCE` is the predefined locale constant for French as used in France. Its language and country values are `fr` and `FR`, respectively. After it is installed as the default locale, resource-bundle lookup for the `Message` base name constructs the locale-specific candidate bundle name `Message_fr_FR.properties`. That properties file supplies the `msg` key with the value `Bienvenue!`, which is then returned by `getString` and printed.

`Locale locale = new Locale ( " fr " , " FR " );` produces the same language-country locale explicitly. The two-argument `Locale` constructor accepts a language code followed by a country code; therefore, this instance represents the `fr_FR` locale. ResourceBundle's standard lookup convention incorporates those locale components into the candidate resource name, allowing it to resolve `Message_fr_FR.properties` before falling back to the base `Message.properties` file. Since this locale-specific bundle is found and contains `msg`, the printed result is `Bienvenue!`.

Oracle documents the predefined locale constants and locale construction in the Locale API, and documents locale-based resource-bundle candidate selection in the ResourceBundle API.

QUESTION NO: 4

Given:

```
Path source = Paths.get("/reports/monthly.txt");
Path target = Paths.get("/archive/monthly.txt");
```

Which two statements are true about the following invocation? (Choose two.)

```
Files.move(source, target, StandardCopyOption.REPLACE_EXISTING);
```

A. An existing target file can be replaced.

B. The source file is copied but remains at its original path.

C. The invocation can throw IOException.

D. The target path must identify an existing directory.

E. The operation always performs an atomic move.

ANSWER: A C

Explanation:

If the target path already exists, `StandardCopyOption.REPLACE_EXISTING` permits the move operation to replace it, subject to implementation-specific restrictions such as an inability to delete a non-empty directory. Without this option, an existing target normally causes `FileAlreadyExistsException`.

The move operation can throw `IOException`. File-system operations can fail for many checked-I/O reasons, including access restrictions, unavailable storage, or problems with the source or target path. Therefore, the enclosing method must handle or declare `IOException`. See the Files.move API documentation and StandardCopyOption API documentation.

QUESTION NO: 5

Given:

```
Path p = Paths.get("/home/./docs/../notes/todo.txt");
```

Which two statements are true? (Choose two.)

A. `p.normalize()` represents `/home/notes/todo.txt`.

B. `p.getFileName()` returns `notes`.

C. Creating p verifies that the `/home/notes` directory exists.

D. The root component of p is `/`.

E. `p.normalize()` changes the file system by removing the docs directory.

ANSWER: A D

Explanation:

Calling `normalize()` removes redundant name elements where possible. The current-directory element `.` is removed, and the `docs/..` pair is eliminated. Therefore, `p.normalize()` represents `/home/notes/todo.txt`.

Creating and manipulating a `Path` object is generally a syntactic operation. The call to `Paths.get()` parses the supplied path string but does not verify that the represented file or directories exist. File-system access occurs only when an operation that requires it, such as `Files.exists()` or `Files.newInputStream()`, is performed. See the Path.normalize and Paths.get API documentation.

QUESTION NO: 6

Given the code fragment:

Stream<>> iStr= Stream.of (

Arrays.asList (“1”, “John”),

Arrays.asList (“2”, null)0;

Stream< nInSt = iStr.flatMapToInt ((x) -> x.stream ()); nInSt.forEach (System.out :: print);

What is the result?

A. 1John2null

B. 12

C. A NullPointerException is thrown at run time.

D. A compilation error occurs.

ANSWER: D

Explanation:

A compilation error occurs. The stream source contains lists of `String` values, so invoking `x.stream()` in the lambda produces a `Stream<String>`. However, `flatMapToInt` is specifically the primitive-stream flattening operation. Its mapper must return an `IntStream`, not an object stream such as `Stream<String>`. Consequently, the lambda expression is not compatible with the functional-interface type required by `flatMapToInt`, and the expression cannot be compiled. Additionally, a successful `flatMapToInt` invocation produces an `IntStream`, so its result would need an `IntStream` target variable rather than a `Stream<String>` variable. To flatten the nested string lists, the appropriate operation would be `flatMap(x -> x.stream())`; to obtain primitive integers, the strings would first need to be converted and the mapper would need to return an `IntStream`. Oracle defines the required `flatMapToInt` mapper return type in the Java SE 8 Stream API. The resulting primitive stream type is documented by the IntStream API.

QUESTION NO: 7

The data.doc, data.txt and data.xml files are accessible and contain text. Given the code fragment:

Stream paths = Stream.of (Paths. get(“data.doc”),

Paths. get(“data.txt”), Paths. get(“data.xml”));

paths.filter(s-> s.toString().contains(“data”)).forEach(s -> { try {

Files.readAllLines(s)

.stream()

.forEach(System.out::println); //line n1

} catch (IOException e) {

System.out.println(“Exception”);

}

});

What is the result?

A. The program prints the content of data.txt file.

B. The program prints:
Exception
<>
<>

C. A compilation error occurs at line n1.

D. The program prints the content of the three files.

ANSWER: D

Explanation:

The program prints the content of the three files. The stream is created with three `Path` objects: one for `data.doc`, one for `data.txt`, and one for `data.xml`. For each path, `Path.toString()` supplies the path’s string representation, and every supplied filename contains the text `data`. Consequently, all three paths pass the `filter` operation and are delivered to the terminal `forEach` operation in encounter order for this sequential stream.

For each selected path, `Files.readAllLines(s)` reads all lines from the accessible text file and returns a `List<String>`. Calling `stream()` on that list creates a stream of its individual lines. The method reference `System.out::println` then prints each line, so the complete contents of each file are output, with the files processed in

the order `data.doc`, `data.txt`, and `data.xml`. The surrounding `try/catch` correctly handles the checked `IOException` declared by `readAllLines`. Oracle documents `readAllLines` at Files.readAllLines and stream traversal at Stream.forEach.

QUESTION NO: 8

Given:

class UserException extends Exception { } class AgeOutOfLimitException extends UserException { }

and the code fragment:

class App {

public void doRegister(String name, int age) throws UserException, AgeOutOfLimitException { if (name.length () <= 60) { throw new UserException ();

} else if (age > 60) {

throw new AgeOutOfLimitException ();

} else {

System.out.println(“User is registered.”);

} }

public static void main(String[ ] args) throws UserException {

App t = new App ();

t.doRegister(“Mathew”, 60);

} }

What is the result?

A. User is registered.

B. An AgeOutOfLimitException is thrown.

C. A UserException is thrown.

D. A compilation error occurs in the main method.

ANSWER: C

Explanation:

`"Mathew"` has a length of 6, which satisfies the first condition, `name.length() <= 60`. Therefore, execution enters that branch immediately and executes `throw new UserException()`. The result is that a `UserException` is thrown. Because `UserException` extends `Exception`, it is a checked exception. The `main` method is permitted to invoke `doRegister` without a local `try/catch` because its own `throws UserException` clause declares that this exception may be propagated to the JVM. The declared exception relationship is also appropriate: `AgeOutOfLimitException` is a subclass of `UserException`, so a declaration of `UserException` covers that subtype when exceptions are propagated. Java evaluates an `if/else if` chain in order; once the first condition is true, the later age test is not reached. Oracle’s Java Language Specification describes abrupt completion caused by a `throw` statement and the compile-time rules for checked exceptions in JLS Chapter 11: Exceptions. The inheritance behavior used here follows the Java platform’s Exception API documentation.

QUESTION NO: 9

Which two are elements of a singleton class? (Choose two.)

A. a transient reference to point to the single instance

B. a public method to instantiate the single instance

C. a public static method to return a copy of the singleton reference

D. a private constructor to the class

E. a public reference to point to the single instance

ANSWER: C D

Explanation:

A singleton class restricts construction so that clients cannot freely create additional objects. Therefore, **a private constructor to the class** is a core element: access control prevents code outside the class from invoking `new` for that type. The class itself controls the one permitted construction path and retains the sole instance.

Clients need a controlled way to obtain that already-managed object, which is supplied by **a public static method to return a copy of the singleton reference**. Here, "copy of the reference" means the method returns the reference value to the same singleton object; it does not create or clone another object. A typical accessor is `public static MySingleton getInstance()`, which returns the static field holding the singleton. This approach preserves a single object identity while making that object accessible to callers. The Java Language Specification defines constructors and their access control, including private constructors, in its constructor declaration rules: JLS 8.8.9: Default Constructors. Java's object-creation documentation also describes construction through `new`, the operation a private constructor prevents outside callers from performing: Oracle Java Tutorial: Creating Objects.

QUESTION NO: 10

Given the code fragment:

```
public static void main(String[] args) {
    Stream.of("Java", "Unix", "Linux")
    .filter(s -> s.contains("n"))
    .peek(s -> System.out.println("PEEK: " + s))
    // line n1
}
```

Which two code fragments, when inserted at line n1 independently, result in the output PEEK: Unix?

A. .anyMatch ();

B. .allMatch ();

C. .findAny ();

D. .noneMatch ();

E. .findFirst ();

ANSWER: C E

Explanation:

`.findAny ();` and `.findFirst ();` are terminal operations, so either one triggers evaluation of the otherwise lazy stream pipeline. As the pipeline begins processing its eligible first element, the `peek` operation executes its supplied action and prints `PEEK: Unix`. The result returned by either terminal operation need not be consumed for traversal to occur; invoking the terminal operation itself is what starts processing.

`peek` is an intermediate operation intended primarily for observing elements as they flow through a pipeline. Intermediate operations do not process elements until a terminal operation is invoked. Both selected operations are short-circuiting terminal operations: `findFirst` obtains an element according to the stream encounter order, while `findAny` may return an available element and allows evaluation to stop once a result is found. With the given sequential pipeline, evaluation reaches `Unix`, causing the stated diagnostic output before the terminal result is produced. Oracle documents the lazy nature of stream pipelines and the role of terminal operations in initiating traversal in the java.util.stream package summary; the specific behavior of `findFirst` and `findAny` is defined in the Stream API.

QUESTION NO: 11

Given the definition of the Vehicle class:

```
class Vehicle {

String name;

void setName (String name) {

this.name = name;

}

String getName() {

return name;

}

}
```

Which action encapsulates the Vehicle class?

A. Make the Vehicle class public.

B. Make the name variable public.

C. Make the setName method public.

D. Make the name variable private.

E. Make the setName method private.

F. Make the getName method private.

ANSWER: D

Explanation:

Make the name variable private. Encapsulation is the object-oriented principle of hiding an object's internal state and requiring access to that state through a controlled interface. In this class, `name` is currently declared with package-private access, so other classes in the same package can read or replace its value directly. Declaring it `private` prevents direct access outside the `Vehicle` class. The class can then provide controlled access through its `setName` and `getName` methods. This design lets the class preserve ownership of its state and makes it possible to add validation, normalization, logging, or other rules later in `setName` without requiring callers to change how they interact with a `Vehicle`. For example, a future implementation could reject a null or blank name before assigning the field. Java's access-control mechanism supports this design: a private member is accessible only within its declaring top-level class. Oracle's Java tutorial describes encapsulation as bundling data and methods and commonly implementing it by keeping fields private while exposing suitable methods. See Oracle Java Tutorial: Encapsulation and Java Language Specification: Determining Accessibility.

QUESTION NO: 12

You want to create a singleton class by using the Singleton design pattern.

Which two statements enforce the singleton nature of the design? (Choose two.)

A. Make the class static.

B. Make the constructor private.

C. Override equals() and hashCode() methods of the java.lang.Object class.

D. Use a static reference to point to the single instance.

E. Implement the Serializable interface.

ANSWER: B D

Explanation:

Make the constructor private. is required because it prevents code outside the class from invoking `new` and creating additional instances. Constructor access control is the fundamental mechanism that restricts instance creation to code managed by the singleton class itself. The Java language supports private constructors specifically through its member-access rules, so the class can control how, or whether, its objects are constructed.

Use a static reference to point to the single instance. supplies the one shared object and a class-level access path to it. A static field belongs to the class rather than to any particular object, making it appropriate for retaining the sole instance. The singleton can initialize this field eagerly, or initialize it lazily in a static accessor while ensuring that concurrent callers cannot create more than one instance. Together, private construction and a class-owned static instance reference ensure controlled creation and shared access, which are the essential structural properties of the Singleton pattern.

Oracle's Java tutorials describe static fields as class variables shared by all instances and document private access as restricted to the declaring class: Class Variables and Controlling Access to Members of a Class.

QUESTION NO: 13

Given:

```
class Counter extends Thread {
     int i = 10;
     public synchronized void display(Counter obj) {
          try {
               Thread.sleep(5);
               obj.increment(this);
               System.out.printIn(i);
          } catch (InterruptedException ex) {   }
     }
     public synchronized void increment (Counter obj) {
          i++;
     }
}
public class Test {
     public static void main(String[] args) {
          final Counter obj1 = new Counter();
          final Counter obj2 = new Counter();
          new Thread(new Runnable() {
               public void run() {obj1.display(obj2);
               }
          }).start();
          new Thread(new Runnable() {
               public void run() { obj2.display(obj1); }
          }).start();
     }
}
```

From what threading problem does the program suffer?

A. race condition

B. deadlock

C. starvation

D. livelock

ANSWER: B

Explanation:

deadlock is correct. A deadlock occurs when two or more threads are permanently blocked because each thread holds a lock needed by another thread while waiting to acquire a different lock. Consequently, none of the involved threads can proceed to the point where it releases the lock it already owns. This is a circular waiting condition: one thread has acquired the first monitor and waits for the second, while another thread has acquired the second monitor and waits for the first. Since Java intrinsic locks used by `synchronized` are mutually exclusive, the threads remain blocked unless some external intervention terminates the program or otherwise breaks the cycle.

The Java platform documentation describes deadlock as a situation in which threads are each waiting for a lock held by the other, so neither can make progress. It also notes that deadlocks can be difficult to diagnose in larger applications, where lock acquisition paths may be spread across multiple methods. A practical prevention technique is to define and consistently follow one global order for acquiring related locks. See Oracle's Deadlock tutorial and the Thread.State API documentation for the blocked-state context.

QUESTION NO: 14

Which two statements are true about the Fork/Join Framework? (Choose two.)

A. The RecursiveTask subclass is used when a task does not need to return a result.

B. The Fork/Join framework can help you take advantage of multicore hardware.

C. The Fork/Join framework implements a work-stealing algorithm.

D. The Fork/Join solution when run on multicore hardware always performs faster than standard sequential solution.

ANSWER: B C

Explanation:

The Fork/Join framework can help an application take advantage of multicore hardware because a `ForkJoinPool` maintains multiple worker threads that can execute independent portions of a divide-and-conquer computation concurrently. A task can split a sufficiently large problem into smaller subtasks, fork those subtasks for asynchronous execution, and then join their results. This model is particularly useful when the work can be recursively decomposed and the subtasks have enough computational cost to justify the scheduling overhead.

The Fork/Join framework implements a work-stealing algorithm. Each worker thread generally processes tasks from its own deque, but an idle worker can steal tasks from another worker's queue. Work stealing helps keep available processors busy when task sizes are uneven or when one worker completes its local work earlier than others. The Java API describes `ForkJoinPool` as using work stealing to efficiently support many tasks that produce subtasks, and it is designed to use available parallelism effectively. See the Java SE 8 ForkJoinPool API documentation and Oracle's Fork/Join tutorial.

QUESTION NO: 15

Given:

1. abstract class Shape {

2. Shape ( ) { System.out.println ("Shape"); }

3. protected void area ( ) { System.out.println ("Shape"); } 4. } 5.

6. class Square extends Shape {

7. int side;

8. Square int side {

9. /* insert code here */

10. this.side = side;

11. }

12. public void area ( ) { System.out.println ("Square"); }

13. }

14. class Rectangle extends Square {

15. int len, br;

16. Rectangle (int x, int y) {

17. /* insert code here */

18. len = x, br = y;

19. }

20. void area ( ) { System.out.println (“Rectangle”); }

21. }

Which two modifications enable the code to compile? (Choose two.)

A. At line 1, remove abstract

B. At line 9, insert super ( );

C. At line 12, remove public

D. At line 17, insert super (x);

E. At line 17, insert super (); super.side = x;

F. At line 20, use public void area ( ) {

ANSWER: D F

Explanation:

The constructor in `Rectangle` must explicitly invoke `Square`’s constructor with an integer argument, so `super(x);` is required. A constructor’s first statement is either an explicit constructor invocation or an implicit call to the no-argument superclass constructor. Here, `Square` declares a constructor that receives the side length, so construction of a `Rectangle` must supply the required value before assigning its own `len` and `br` fields. Passing `x` correctly initializes the inherited `side` field through the `Square` constructor. The Java Language Specification describes this superclass-constructor invocation rule in JLS 8.8.7.1.

The `area()` method declared by `Square` is `public`. Therefore, the overriding implementation in `Rectangle` must also be `public`; using `public void area()` preserves the required access level while providing the rectangle-specific implementation. Java permits an overriding method to have the same or broader access, and this declaration has the same public accessibility as the inherited method. The overriding requirements are specified in JLS 8.4.8.3.

QUESTION NO: 16

Given the code fragment:

Path p1 = Paths.get(“/Pics/MyPic.jpeg”);

System.out.println (p1.getNameCount() +

“:” + p1.getName(1) +

“:” + p1.getFileName());

Assume that the Pics directory does NOT exist.

What is the result?

A. An exception is thrown at run time.

B. 2:MyPic.jpeg:MyPic.jpeg

C. 1:Pics:/Pics/ MyPic.jpeg

D. 2:Pics: MyPic.jpeg

ANSWER: B

Explanation:

`2:MyPic.jpeg:MyPic.jpeg` is printed. Creating a `Path` with `Paths.get` parses the supplied path string; it does not access the file system or require the referenced directory or file to exist. Therefore, the absence of the `Pics` directory has no effect on these operations. For the absolute path `/Pics/MyPic.jpeg`, the root component is `/` and is not included in the path's name elements. The two name elements are `Pics`, at index zero, and `MyPic.jpeg`, at index one. Consequently, `getNameCount()` returns `2`, while `getName(1)` returns `MyPic.jpeg`. The `getFileName()` method returns the last name element of the path, which is also `MyPic.jpeg`. The two literal colon strings are concatenated directly with these values, so no spaces occur in the output. Oracle's `Path` documentation specifies that the root is not counted as a name element and defines both name-indexing and file-name behavior: Java SE 8 Path API. Path creation behavior is described in the Java SE 8 Paths API.

QUESTION NO: 17

Given the code fragment:

Path path1 = Paths.get("/app/./sys/");

Path res1 = path1.resolve("log");

Path path2 = Paths.get("/server/exe/");

Path res1 = path2.resolve("/readme/");

System.out.println(res1);

System.out.println(res2);

What is the result?

A. /app/sys/log/readme/server/exe

B. /app/log/sys/server/exe/readme

C. /app/./sys/log/readme

D. /app/./sys/log/server/exe/readme

E. /app/./sys/log
/readme

ANSWER: E

Explanation:

Assuming the repeated declaration of `res1` is the intended declaration of `res2`, the output is `/app/./sys/log` followed by `/readme`. Calling `resolve` with the relative path `log` appends that name to the existing path. A `Path` is not automatically normalized by `resolve`, so the existing `.` name remains in the resulting path. Therefore, resolving `log` against `/app/./sys/` produces `/app/./sys/log`.

The second call uses `/readme/`, which is an absolute path because it begins at the root. The `Path.resolve` contract specifies that when the argument is absolute, the method returns that argument rather than combining it with the original path. Thus, resolving `/readme/` against `/server/exe/` produces the absolute path `/readme`; the provider's usual string representation omits the redundant trailing separator. This behavior follows the Java NIO `Path` resolution rules documented by Oracle: Path.resolve(Path) and Path normalization.

QUESTION NO: 18

Given:

class Worker extends Thread { CyclicBarrier cb;

public Worker(CyclicBarrier cb) { this.cb = cb; }

public void run () { try { cb.await();

System.out.println(“Worker…”);

} catch (Exception ex) { }

}

}

class Master implements Runnable { //line n1 public void run () { System.out.println(“Master…”);

} }

and the code fragment:

Master master = new Master();

//line n2

Worker worker = new Worker(cb); worker.start();

You have been asked to ensure that the run methods of both the Worker and Master classes are executed. Which modification meets the requirement?

A. At line n2, insert CyclicBarrier cb = new CyclicBarrier(2, master);

B. At line n2, insert CyclicBarrier cb = new CyclicBarrier(1);

C. At line n2, insert CyclicBarrier cb = new CyclicBarrier(1, master);

D. At line n2, insert CyclicBarrier cb = new CyclicBarrier(master);

ANSWER: C

Explanation:

`At line n2, insert CyclicBarrier cb = new CyclicBarrier(1, master);` is correct because it creates a barrier requiring exactly one participating thread to call `await()`. The `Worker` thread is started and enters its `run()` method, where it calls `cb.await()`. As that call supplies the sole required arrival, the barrier trips immediately. When a `CyclicBarrier` is constructed with both a party count and a barrier action, its barrier action runs once the last required party arrives and before the waiting threads are released. Here, `master` is supplied as that action and implements `Runnable`, so its `run()` method prints `Master…`. Once the barrier action completes, the worker is released from `await()` and continues to print `Worker…`. Therefore, this construction ensures execution of both required `run()` methods, with the master action occurring as part of tripping the barrier. The barrier action is executed by the thread that reaches the barrier last; with one party, that is the worker thread. See the Java SE 8 CyclicBarrier API documentation and its Runnable interface documentation.

QUESTION NO: 19

Given the code fragments:

interface CourseFilter extends Predicate { public default boolean test (String str) { return str.contains (“Java”);

} } and

List strs = Arrays.asList(“Java”, “Java EE”, “Embedded Java”);

Predicate cf1 = s - > s.length() > 3; Predicate cf2 = new CourseFilter() { //line n1 public boolean test (String s) { return s.startsWith (“Java”);

} };

long c = strs.stream()

.filter(cf1)

.filter(cf2 //line n2 .count();

System.out.println(c);

What is the result?

A. 2

B. 3

C. A compilation error occurs at line n1.

D. A compilation error occurs at line n2.

ANSWER: A

Explanation:

The result is **2**. `CourseFilter` is a functional interface because it inherits the single abstract `test` method from `Predicate`; its supplied `default` implementation is also legal. The anonymous implementation assigned to `cf2` provides its own implementation of `test(String)`, which overrides that default behavior. Therefore, its effective test is whether a string starts with `"Java"`.

The first stream filter uses `cf1`, which retains every listed string because `"Java"`, `"Java EE"`, and `"Embedded Java"` each have a length greater than three. The second filter invokes the anonymous predicate's overridden `test` method. It retains `"Java"` and `"Java EE"`, both of which start with `"Java"`; the remaining string does not. Consequently, the stream has two elements when `count()` is evaluated.

The raw declaration of `cf2` can lead to an unchecked warning, but it does not prevent this code from compiling. Oracle documents the `Predicate` contract at Predicate API and the stream filtering operation at Stream.filter API.

QUESTION NO: 20

Which two code blocks correctly initialize a Locale variable? (Choose two.)

A. Locale loc1 = "UK";

B. Locale loc2 = Locale.getInstance("ru");

C. Locale loc3 = Locale.getLocaleFactory("RU");

D. Locale loc4 = Locale.UK;

E. Locale loc5 = new Locale ("ru", "RU");

ANSWER: D E

Explanation:

`Locale loc4 = Locale.UK;` correctly initializes the variable by assigning Oracle Java's predefined `Locale` constant for the United Kingdom. `Locale.UK` is a public static constant in `java.util.Locale`, representing the language and country combination for English and the United Kingdom. It can therefore be assigned directly to a variable whose declared type is `Locale`.

`Locale loc5 = new Locale("ru", "RU");` also correctly initializes a `Locale` variable. The two-argument `Locale` constructor accepts a language code followed by a country code and creates a locale using those values. Here, `ru` identifies Russian as the language and `RU` identifies the Russian Federation as the country. Java locale identifiers conventionally use lowercase language codes and uppercase country codes, matching the values used in this initialization. The constructor returns a newly created object whose type is `Locale`, so the assignment is type-correct. Oracle documents both the

predefined locale constants and the `Locale(String language, String country)` constructor in the Java SE 8 Locale API. See also Oracle's locale creation tutorial for examples of constructing locales from language and country codes.

QUESTION NO: 21

Given:

interface Doable {

public void doSomething (String s); }

Which two class definitions compile? (Choose two.)

A. public abstract class Task implements Doable { public void doSomethingElse(String s) { } }

B. public abstract class Work implements Doable { public abstract void doSomething(String s) { } public void doYourThing(Boolean b) { } }

C. public class Job implements Doable { public void doSomething(Integer i) { } }

D. public class Action implements Doable { public void doSomething(Integer i) { } public String doThis(Integer j) { } }

E. public class Do implements Doable { public void doSomething(Integer i) { } public void doSomething(String s) { } public void doThat (String s) { } }

ANSWER: A E

Explanation:

`public abstract class Task implements Doable { public void doSomethingElse(String s) { } }` compiles because an abstract class is permitted to implement an interface without supplying implementations for every inherited abstract interface method. Since `Task` is declared `abstract`, it may leave `doSomething(String)` unimplemented. Its separately declared `doSomethingElse(String)` method is simply an additional concrete instance method and has no effect on that rule.

`public class Do implements Doable { public void doSomething(Integer i) { } public void doSomething(String s) { } public void doThat (String s) { } }` compiles because it provides the required public implementation whose signature is exactly `void doSomething(String)`. The `doSomething(Integer)` declaration is a valid overload, rather than an implementation of the interface method, because overload resolution distinguishes methods by their parameter types. A class may freely declare this overload and the unrelated `doThat(String)` method in addition to meeting its interface contract. The Java Language Specification permits abstract classes to defer implementation of inherited abstract methods and defines method signatures using the method name and parameter types. See the JLS rules for abstract classes and the JLS definition of method signatures.

QUESTION NO: 22

Given the code fragments:

class TechName { String techName;

TechName (String techName) { this.techName=techName;

} } and

List tech = Arrays.asList (

new TechName("Java-"), new TechName("Oracle DB-"), new TechName("J2EE-")

);

Stream stre = tech.stream(); //line n1

Which should be inserted at line n1 to print Java-Oracle DB-J2EE-?

A. stre.forEach(System.out::print);

B. stre.map(a-> a.techName).forEach(System.out::print);

C. stre.map(a-> a).forEachOrdered(System.out::print);

D. stre.forEachOrdered(System.out::print);

ANSWER: B

Explanation:

`stre.map(a-> a.techName).forEach(System.out::print);` is correct because the stream contains `TechName` objects, while the required output consists of the values held in each object's `techName` instance field. The `map` intermediate operation applies the lambda expression to every stream element, transforming each `TechName` object into its associated `String`: `"Java-"`, `"Oracle DB-"`, and `"J2EE-"`. It therefore creates a `Stream<String>` suitable for direct output.

The terminal `forEach` operation then invokes `System.out.print` once for each resulting string. Because the source is a sequential stream created with `tech.stream()`, its encounter order follows the order of the list produced by `Arrays.asList`. The strings are printed consecutively without any added separator or line terminator, producing exactly `Java-Oracle DB-J2EE-`. Oracle's Stream API specifies that `map` transforms stream elements and that `forEach` consumes them as a terminal operation. See the Stream.map documentation and the Stream.forEach documentation.

QUESTION NO: 23

Given:

```
Map<String, Integer> scores = new HashMap<>();
scores.put("Ann", 10);
```

Which two statements are true? (Choose two.)

A. `scores.computeIfAbsent("Ann", k -> 20)` replaces Ann's value with 20.

B. `scores.computeIfAbsent("Bob", k -> null)` adds the mapping Bob=null.

C. `scores.computeIfAbsent("Bob", k -> null)` does not add a mapping for Bob.

D. `scores.merge("Bob", 5, Integer::sum)` adds the mapping Bob=5.

E. `scores.merge("Ann", 5, Integer::sum)` replaces Ann's value with 5.

ANSWER: C D

Explanation:

The `computeIfAbsent()` method invokes its mapping function only when the specified key is not already associated with a value, or is mapped to `null`. If the mapping function returns `null`, no mapping is recorded for that key. Thus, `scores.computeIfAbsent("Bob", k -> null)` does not add a mapping for `Bob`.

The `merge()` method associates its supplied value with a key when that key is absent or currently mapped to `null`. Therefore, `scores.merge("Bob", 5, Integer::sum)` adds the mapping `Bob=5`; the remapping function is not invoked because no previous non-null value exists. See the Map.computeIfAbsent and Map.merge API documentation.

QUESTION NO: 24

Which two statements are true about an `ExecutorService`? (Choose two.)

A. `submit(Callable)` returns a Future.

B. `shutdown()` prevents execution of tasks submitted before it is called.

C. `shutdown()` rejects tasks submitted after it is called.

D. `execute(Runnable)` returns a Future.

E. An ExecutorService can be restarted after shutdown().

ANSWER: A C

Explanation:

The `submit(Callable<T>)` method returns a `Future<T>`. The Future represents the pending completion of the submitted task and can be used to retrieve the value returned by the task's `call()` method.

Calling `shutdown()` initiates an orderly shutdown. Tasks that were previously submitted are allowed to execute, but the executor rejects new task submissions. It does not forcibly stop currently executing tasks; `shutdownNow()` is the more aggressive shutdown request. See the Java SE 8 ExecutorService API and Future API.

QUESTION NO: 25

Given:

```
Connection con = DriverManager.getConnection(url, user, password);
PreparedStatement ps = con.prepareStatement(
        "SELECT NAME FROM EMPLOYEE WHERE ID = ?");
```

Which two statements are true? (Choose two.)

A. `ps.setInt(1, 101)` can be used to bind a value to the parameter marker before execution.

B. `ps.executeQuery()` returns a ResultSet.

C. `ps.executeUpdate()` returns a ResultSet for this query.

D. The parameter marker must be written as `'?'` in the SQL statement.

E. A PreparedStatement cannot execute a SELECT statement.

ANSWER: A B

Explanation:

A `PreparedStatement` represents a precompiled SQL statement that can contain parameter markers. The application supplies a value for the parameter marker by calling a suitable setter method, such as `setInt(1, 101)`, before executing the statement. Parameter indexes begin with 1.

Because the SQL text is a `SELECT` statement, `executeQuery()` is the appropriate execution method. It returns a `ResultSet` that provides access to the rows produced by the query. The parameter marker itself must not be enclosed in quotation marks; the JDBC driver handles the SQL representation of the value supplied through the setter method. See the PreparedStatement and ResultSet API documentation.

QUESTION NO: 26

Given the code fragment:

List < String > codes = Arrays.asList (“DOC”, “MPEG”, “JPEG”);

codes.forEach (c - > System.out.print(c + “ “));

String fmt = codes.stream()

.filter (s- > s.contains (“PEG”))

.reduce((s, t) - > s + t).get();

System.out.println(“\n” + fmt);

What is the result?

A. DOC MPEG JPEG
MPEGJPEG

B. DOC MPEG MPEGJPEGMPEGMPEGJPEG

C. MPEGJPEGMPEGJPEG

D. The order of the output is unpredictable.

ANSWER: A

Explanation:

`DOC MPEG JPEG`
`MPEGJPEG` is correct. The list returned by `Arrays.asList` has the encounter order `DOC`, `MPEG`, and `JPEG`. Calling `forEach` on that list invokes the supplied lambda once for each element in that order. Each invocation uses `System.out.print`, rather than `println`, so the first statement writes `DOC MPEG JPEG` on one line, including a trailing space after the final value.

The stream pipeline then retains the strings containing `PEG`: `MPEG` and `JPEG`. The two-argument `reduce` operation combines those elements with the accumulator expression `s + t`, producing the single string `MPEGJPEG`. Because the filtered stream is nonempty, `reduce` returns an `Optional` containing that string, and `get()` obtains it. Finally, the argument to `println` starts with `\n`, which first moves output to the next line; `println` then prints `MPEGJPEG` and terminates that line. Oracle documents the reduction behavior in the Stream.reduce API and ordered traversal in the Iterable.forEach API.

QUESTION NO: 27

Given the code fragment:

```
// Login time:2015-01-12T21:58:18.817Z
Instant loginTime = Instant.now();
Thread.sleep(1000);

// Logout time:2015-01-12T21:58:19.880Z
Instant logoutTime = Instant.now();

loginTime = loginTime.truncatedTo(ChronoUnit.MINUTES);   // line n1
logoutTime = logoutTime.truncatedTo(ChronoUnit.MINUTES);

if (logoutTime.isAfter(loginTime))
    System.out.println("Logged out at:"+logoutTime);
else
    System.out.println("Can't logout");
```

What is the result?

A. A compilation error occurs at line n1.

B. Logged out at: 2015-01-12T21:58:19.880Z

C. Can't logout

D. Logged out at: 2015-01-12T21:58:00Z

ANSWER: D

Explanation:

`Logged out at: 2015-01-12T21:58:00Z` is the result because the code truncates the represented instant to minute precision. An `Instant` is a point on the UTC timeline, and truncating it with `ChronoUnit.MINUTES` removes all units smaller than a minute. Therefore, the seconds and fractional seconds in `2015-01-12T21:58:19.880Z` are discarded, leaving `2015-01-12T21:58:00Z`. Truncation does not round to the nearest minute; it retains the current minute and sets the smaller time fields to zero. The trailing `Z` denotes the UTC offset, also called Zulu time, and remains part of the textual representation because the value is still an `Instant`. The Java 8 `Instant.truncatedTo` API specifies that the returned instant is truncated to the specified temporal unit, provided that unit divides evenly into a standard day. Minutes satisfy that requirement. The default `Instant` string representation uses an ISO-8601 UTC form, which produces the displayed value. See the Instant.truncatedTo API documentation and the Instant.toString API documentation.

QUESTION NO: 28

Given the code fragment:

```
List<String> nums = Arrays.asList("EE", "SE");
String ans = nums
    .parallelStream()
    .reduce("Java ", (a, b) -> a.concat(b));
System.out.print(ans);
```

What is the result?

A. Java EEJava EESE

B. Java EESE

C. The program prints either:Java EEJava SEorJava SEJava EE

D. Java EEJava SE

ANSWER: C

Explanation:

The program prints either:Java EEJava SEorJava SEJava EE because the two output operations are performed by separate tasks that may execute concurrently. Submitting tasks to an executor service does not establish a guaranteed execution order between those tasks. The executor makes them eligible to run, but the JVM and operating-system scheduler determine when each worker thread actually receives CPU time. Consequently, the task that writes `Java EE` may complete before the task that writes `Java SE`, or the reverse may happen.

Although the order is nondeterministic, each output string is emitted as a complete unit by the print operation in this context, so the meaningful possible results are the two complete-string orderings shown. The code's completion or shutdown handling ensures that the submitted work is allowed to finish; it does not impose an ordering dependency between the tasks. Oracle's concurrency tutorial explains that independently executing threads are scheduled asynchronously and that their relative execution order cannot generally be assumed. See Oracle's Processes and Threads tutorial and the Java SE 8 ExecutorService API documentation.