Quick Quiz July 14, 2022

Test ID: 216186783

Question #1 of 50

Question ID: 1327955

You are tasked with developing a new Java 11 application that will run on a Linux operating system. The application will require several different features and will be specifically written using Java 11 components.

Which Java 11 SDK features can the application leverage? (Choose two.)

A) `java.util.Date.getDate()`

B) Transport Layer Security (TLS) 1.3

C) Validate Command-Line Flag Arguments

D) Flight Recorder

E) The Java Linker

Explanation

Transport Layer Security (TLS) 1.3 and Flight Recorder are new features of Java version 11 that the application can leverage. (TLS) 1.3 is the latest TLS protocol used within the industry and has many performance enhancements related to previous TLS protocols, such as TLS 1.1 and TLS 1.2. This new protocol has ChaCha20 cipher suites and edDSA signature algorithms. The Flight Recorder is a tool used to gather Java events that occur between a JVM and the underlying operating system. These events are then updated within a single file and can then be sent to Java support engineers.

The method `java.util.Date.getDate()` was a new feature introduced in Java version 8. This method was replaced in Java version 9 with the `Calendar.get(Calendar.DAY_OF_MONTH)` method.

The Java Linker was a new tool introduced in Java version 9 that gave you the ability to create a smaller subset image generated at run time for application optimization. This way you did not need to have the full version of JDK configured for your application.

Validate Command-Line Flag Arguments is a new feature added in Java version 9. This feature is used to confirm that the command line arguments used in the JVM command line interface is compared and validated against a range of valid values.

Objective:
Java Platform Module System

Sub-Objective:
Deploy and execute modular applications, including automatic modules

References:

openjdk.java.net > Open JDK > JEP 328: Flight Recorder

Question #2 of 50

Question ID: 1327801

Given:

```
public class Insider<E> {
  private List<E> mainData;
  // other methods, constructors and fields omitted

  class Accessor implements Supplier<E> {
    private int index = 0;
    @Override public E get() {
      return mainData.get(index++);
    }
  }
}
```

Which two statements are true?

A) To allow access to `mainData` in the `Accessor` class, the `mainData` field must be marked `final`.

B) If a factory that returns instances of `Accessor` is provided in the `Insider` class, that factory would be an instance method.

C) The `Accessor` class may correctly be marked `static`.

D) The `Accessor` class may correctly be marked `private`.

Explanation

The following statements are true:

- The class `Accessor` may correctly be marked `private`.
- If a factory that returns instances of `Accessor` is provided in the Insider class, that factory would be expected to be an instance method.

The class `Accessor` may be private because its definition is not needed externally. Instead, it is sufficient to refer to the instance of `Accessor` simply as `Supplier<E>`.

Nested classes may be static or non-static. Non-static nested classes are also referred to as inner classes. However, static nested classes do not have a direct reference to an instance of the enclosing class, just as instance methods have an implicit this reference but static ones do not. Because the class `Accessor` attempts to access the `mainData` instance field in the `get()` method, it must either have an explicit reference to an object of that type, or it must not be static. Similarly, any factory method would not normally be static, as this would require an explicit

reference to an instance of the outer class. An instance factory method already has such a reference and would therefore be a more natural choice.

While inner classes can be static, static inner classes cannot refer to type variables (such as `<E>`) because those variables are defined on a per-instance basis. Furthermore, if it were static, it would not be able to directly access the variable `mainData`.

There is no requirement to mark instance variables as `final` simply to permit inner classes to access them. There is a requirement that method local variables that are accessed from inner classes or lambda expressions are *effectively final*. Because the lifetime of such objects is potentially greater than the lifetime of the method local variable, requiring the variable to be unchanging means that a copy can safely be taken and used instead.

Objective:
Java Object-Oriented Approach

Sub-Objective:
Declare and instantiate Java objects including nested class objects, and explain objects' lifecycles (including creation, dereferencing by reassignment, and garbage collection)

References:

The Java Tutorials > Learning the Java Language > Classes and Objects > Nested Classes

The Java Tutorials > Learning the Java Language > Generics (Updated) > Cannot Declare Static Fields Whose Types are Type Parameters

---

Question #3 of 50

Question ID: 1327773

```
public class JavaSETest {
  public static void main(String[] args) {
    List<Integer> weights = new ArrayList<>();
    weights.add(0);
    weights.add(5);
    weights.add(10);
    weights.add(15);
    weights.add(20);
    weights.add(25);
    weights.remove(5);
    System.out.println("Weights are "+ weights);
  }
}
```

What is the output of this code?

A) `Weights are [0, 5, 10, 15, 20]`

B) `Weights are [0, 10, 15, 20, 25]`

C) `Weights are [0, 0, 10, 1, 20]`

D) `Weights are null`

Explanation

The code outputs the following:

```
Weights are [0, 5, 10, 15, 20]
```

When the `remove()` method is invoked, it removes the element of the array that was at the index of 5, namely 25. This does ***not*** remove the number 5 from the list. To explicitly remove the number 5, you would have to use `remove(new Integer(5))`. This is an example where Java does not perform autoboxing when the `remove()` function is invoked.

The other options are incorrect as only the element at index number 5 is removed from the list.

Objective:
Working with Java Data Types

Sub-Objective:
Use primitives and wrapper classes, including, operators, parentheses, type promotion and casting

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Numbers and Strings

---

Question #4 of 50

Question ID: 1327895

Given:

```
class Certification {
   private Double getScore() {
     return 90.2;
   }
}
class JavaI extends Certification {
   public Integer getScore() throws Exception {
     return 65;
   }
}
```

Which statement is true about the code?

A) The `getScore` method in `JavaI` overloads the `getScore` method in `Certification`.

B) The `getScore` method in `JavaI` has no relationship with the `getScore` method in `Certification`.

C) The `getScore` method in `JavaI` hides the `getScore` method in `Certification`.

D) The `getScore` method in `JavaI` overrides the `getScore` method in `Certification`.

Explanation

The `getScore` method in `JavaI` has no relationship with the `getScore` method in `Certification`. Because the `getScore` method is declared with the `private` modifier in `Certification`, a method with the same name in the subclass `JavaI` cannot overload, override, or hide the superclass method `getScore`. This code will compile because there are no return type or exception requirements for the `getScore` method in `JavaI`.

The `getScore` method in `JavaI` does not override the `getScore` method in `Certification`. Private methods cannot be overridden in subclasses.

The `getScore` method in `JavaI` does not overload the `getScore` method in `Certification`. Private methods cannot be overloaded in subclasses.

The `getScore` method in `JavaI` does not hide the `getScore` method in `Certification`. Private methods cannot be hidden in subclasses.

Objective:
Java Object-Oriented Approach

Sub-Objective:
Utilize polymorphism and casting to call methods, differentiate object type versus reference type

References:

Oracle Technology Network > Java SE > Java Language Specification > Classes > Requirements in Overriding and Hiding

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Interfaces and Inheritance > Polymorphism

---

Question #5 of 50

Question ID: 1328049

Given:

```
public class RuntimeExceptionTests {
   public static char performOperation(String str) {
     return str.charAt(0);
   }
   public static void main (String[] args) {
     performOperation(null);
   }
}
```

Which exception is thrown by running the given code?

A) `StringIndexOutOfBoundsException`

B) `IndexOutOfBoundsException`

C) `ArrayIndexOutOfBoundsException`

D) `NullPointerException`

Explanation

`NullPointerException` is the exception thrown by running the given code. This exception is thrown whenever an object is required, such as when accessing instance members. In this code, the `charAt` method is invoked on a non-existing object, so a `NullPointerException` is thrown.

The exceptions `IndexOutOfBoundsException`, `ArrayIndexOutOfBoundsException`, or `StringIndexOutOfBoundsException` will not be thrown by running the given code. This is because the object is not available to attempt executing the `charAt` method. If the `charAt` method is invoked with an invalid index for a string, then a `StringIndexOutOfBoundsException` will be thrown.

Objective:
Exception Handling

Sub-Objective:
Handle exceptions using try/catch/finally clauses, try-with-resource, and multi-catch statements

References:

Oracle Documentation > Java SE 11 API > Class NullPointerException

---

Question #6 of 50

Question ID: 1327907

You define the following interface to represent shippable goods:

```
interface Addressable {
  String getStreet();
```

```
  String getCity();
}
```

You want to add the following method to the `Addressable` interface:

```
String getAddressLabel() {
  return getStreet() + "\n" + getCity();
}
```

Unfortunately, the interface is implemented by many classes in the project. It is not feasible to modify every class definition and have existing classes fail to compile. What is the best solution to this design problem?

A) Add a static method to the interface

B) Add a default method to the interface

C) Convert the interface to an abstract class and add the method to the class

D) Create a helper, or utility, class and add the method to that class

Explanation

The default method mechanism is intended specifically to support interface evolution as described in this scenario. It allows you to add new instance methods to an interface along with a default implementation. That implementation can refer to an instance of the interface through the implied variable this, just as an instance method defined in a class can. If any particular implementation of the interface has reason to do so, the implementation can be replaced by overriding. Because of this, this option is the correct answer.

Converting the interface to an abstract class might work, but this approach has two distinct problems. First, every class that implements the interface must be changed to say extends Addressable, and a specific goal was to avoid extensive code rewriting. Second, if any of the classes that implement Addressable already inherit from anything other than Object, then the solution will fail, since Java permits only one direct parent class.

Creating a utility class will work, but this approach has its own set of issues. It avoids making changes to other parts of code that do not yet need to use the new feature. Prior to Java 8, this would probably have been a good solution. Two disadvantages, though, are that the behavior must be in a separate class, and that such a method must be static. Being in a different class, it lacks cohesion with the other behaviors related to *addressableness*. It will not be so easy to know where to look. Being static will mean that the code does not look like object-oriented code. Instead of calling `myPlace.getAddressLabel()`, the code would probably resemble `AddressableUtils.getAddressLabel(myPlace)`. More importantly, static methods do not support overriding, so the behavior becomes very hard to modify for any given Addressable. In short, this approach makes the behavior brittle.

Creating a static method in the interface is technically feasible, but it has the same issue described for static behavior: it is brittle and poorly designed.

Objective:
Java Object-Oriented Approach

Sub-Objective:

Create and use interfaces, identify functional interfaces, and utilize private, static, and default methods

References:

The Java Tutorials > Learning the Java Language > Interfaces and Inheritance > Default Methods

---

Question #7 of 50

Question ID: 1328074

Which of the following options is the correct functional method of the `UnaryOperator` interface?

A) `main()`

B) `apply()`

C) `get()`

D) `test()`

Explanation

The correct functional method of the `UnaryOperator` interface is the `apply()` method.

The `test`, `get` and `main` methods are not part of the `UnaryOperator` interface and so are incorrect.

The key difference between a `UnaryOperator` and a `BinaryOperator` is that the

`UnaryOperator` accepts a single argument and `BinaryOperator` works on two arguments.

Here is an example of the usage of the `UnaryOperator` to convert a `String` from uppercase to lowercase:

```
import java.util.function.UnaryOperator;

public class UnaryOpDemo {

  public static void main(String[] args) {
    UnaryOperator<String> s = (a)-> a.toLowerCase();
    System.out.println(s.apply("PLEASE SPEAK SOFTLY"));
  }
}
```

This code outputs:

```
please speak softly
```

Objective:

Working with Streams and Lambda expressions

Sub-Objective:
Implement functional interfaces using lambda expressions, including interfaces from the java.util.function package

References:

Java Platform Standard Edition 11 > API > java.util.function > UnaryOperator

---

Question #8 of 50

Question ID: 1328147

Given the following code fragment:

```
Executor ex = Executors.newScheduledThreadPool(5);
```

How many threads will be created when passing tasks to the `Executor` object?

A) A new thread will be created for each fifth task.

B) Five threads will be created, each thread supporting a single task.

C) A single thread will be created for up to five tasks.

D) A new thread will be created for each task, with five threads available initially.

E) A new thread will be created for each task, up to five threads simultaneously.

Explanation

Because the `newScheduledThreadPool` factory method is invoked with the argument 5, five threads will be created, each thread supporting a single task. Threads not being used will be idle. The number of threads created for an `Executor` object is determined by the method used to obtain it. This thread pool supports tasks that run after a delay or are executed periodically.

A new thread will not be created for each fifth task because no such `Executor` object supports this default pattern.

A single thread will not be created for up to five tasks because the `newScheduledThreadPool` method specifies 5 as its argument. The `newSingleThreadExecutor` method creates a thread pool that executes a single task on a single thread at time.

A new thread will not be created for each task, up to five threads simultaneously because the `newScheduledThreadPool` method will include idle threads if less than five tasks are run. The `newFixedThreadPool` method creates a thread pool that executes up to a specific number of simultaneous threads.

A new thread will not be created for each task, with five threads available initially because the `newScheduledThreadPool` method will include five threads, whether tasks are running or idle.

Objective:
Concurrency

Sub-Objective:

Create worker threads using Runnable and Callable, and manage concurrency using an ExecutorService and java.util.concurrent API

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Essential Classes > Concurrency > Thread Pools

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Essential Classes > Concurrency > Executor Interfaces

---

Question #9 of 50

Question ID: 1327825

Given:

```
public class PrinterClass {
   private String arg;
   public PrinterClass() { System.out.println("no args"); }
   public PrinterClass(String arg) {
     this();
     this.arg = arg;
     System.out.println("arg");
   }
   public void print() {
     System.out.println(arg);
   }
   public static void main(String[] args) {
     new PrinterClass("val").print();
   }
}
```

What is the result?

A)
```
arg
```

B)
```
no args
arg
val
```

C)
```
arg
val
```

D)
```
val
```

Explanation

The result is the following output:

```
no args
arg
val
```

The code in the main method invokes the `PrinterClass` constructor with the `String` parameter, from which the parameterless `PrinterClass` constructor and the `println` method are invoked. The parameterless constructor prints `no args`, then the overloaded constructor `prints arg`. Finally, because the overloaded constructor sets the `arg` field, the `print` method prints `val`.

The result will not exclude `no args` because the `PrinterClass` constructor with a `String` parameter invokes the other overloaded parameterless `PrinterClass` constructor. The parameterless `PrinterClass` constructor prints `no args`.

The result will not omit the output `val` because the `arg` field is set. This value is printed because the `PrinterClass` constructor with the `String` parameter is invoked.

Objective:
Java Object-Oriented Approach

Sub-Objective:
Initialize objects and their members using instance and static initializer statements and constructors

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Language Basics > Classes and Objects > Using the this Keyword

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Language Basics > Classes and Objects > Providing Constructors for Your Classes

---

Question #10 of 50

Question ID: 1327802

Given the following class:

```
1.   public class Machine {
2.     float OSVersion;
3.     public static void main (String[] args) {
4.       //Insert code here
5.     }
6.   }
```

Which code fragment correctly assigns a value to the `OSVersion` field at line 4?

A)
```
Machine.OSVersion = 10.1f;
```

B)
```
Machine myMachine = new Machine();
Machine.OSVersion = 10;
```

C)
```
Machine myMachine = new Machine();
myMachine.OSVersion = 10;
```

D)
```
OSVersion = 10.1f;
```

Explanation

The following code fragment correctly assigns a value to the `OSVersion` field at line 4:

```
Machine myMachine = new Machine();
myMachine.OSVersion = 10;
```

This code fragment instantiates a `Machine` object because the `OSVersion` field is an instance member. Then, it specifies the instance named `myMachine` to set the `OSVersion` field. To access instance members in a class, you must first instantiate the class and access the field using dot notation by referencing the instance.

The code fragments that do not first instantiate the `Machine` class do not correctly assign a value to the `OSVersion` field. To access the `OSVersion` field as an instance field, the `Machine` class must be first instantiated.

The code fragment that specifies the class name `Machine` rather than the instance name does not correctly assign a value to the `OSVersion` field. To access the `OSVersion` field as an instance field, the instance must be referenced using dot notation.

Objective:
Java Object-Oriented Approach

Sub-Objective:
Define and use fields and methods, including instance, static and overloaded methods

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Classes and Objects > Using Objects

---

Question #11 of 50

Question ID: 1327758

Given the following:

```
public class Java11 {
   public static void main (String[] args) {
     int x = 1;
     int y = x;
```

```
    var z = y;
    z = 10;
    System.out.format("x,y,z: %d,%d,%d", x, y, z);
   }
}
```

What is the result when this program is executed?

A) `x,y,z: 10,10,10`

B) `x,y,z: 1,1,10`

C) `x,y,z: 1,10,10`

D) `x,y,z: 1,1,1`

Explanation

The following output is the result when the program is executed:

`x,y,z: 1,1,10`

Variables of primitive types hold only values, not references. When `y` is assigned to `x`, the value of `x` is copied into `y`. When `z` is assigned to `y`, the value of `y` is copied into `z`. Thus, changing the value of `z` will not modify the values stored in `x` or `y`.

The key difference between primitive variables and reference variables are:

- Reference variables are used to store addresses of other variables. Primitive variables store actual values. Reference variables can only store a reference to a variable of the same class or a sub-class. These are also referred to in programming as *pointers*.
- Reference types can be assigned `null` but primitive types cannot.
- Reference types support method invocation and fields because they reference an object, which may contain methods and fields.
- The naming convention for primitive types is camel-cased, while Java classes are Pascal-cased.

The result will not be output with `x` and/or `y` set to `10` because modifications to the value of `z` will not affect the values of `x` and/or `y`.

The result will not be output where `z` is not set to `1`, because `z` is explicitly assigned the value `10`.

Objective:
Working with Java Data Types

Sub-Objective:
Use primitives and wrapper classes, including, operators, parentheses, type promotion and casting

References:

Oracle Technology Network > Java SE > Java Language Specification > Chapter 4. Types, Values, and Variables > 4.12. Variables

Primitive vs Reference Data Types

Question #12 of 50

Question ID: 1327839

Given:

```
public class Container {
   ArrayList<Integer> compartments;
   private int totalItems;
   public Container(int numCompartments) {
     compartments = new ArrayList<>(numCompartments);
   }
   //Other code omitted
}
```

This class is intended to calculate the total number of items within each compartment. Which statement is true about encapsulation in the `Container` class?

A) This class is poorly encapsulated. The `totalItems` field should be calculated in the constructor and in any methods that modify the `compartments` field.

B) This class is properly encapsulated, but the access modifier on the `compartments` field should be changed to `public`.

C) This class is properly encapsulated, but the access modifier on the constructor should be changed to `private`.

D) This class is poorly encapsulated. The `compartments` field should be set in any methods that modify the `totalItems` field.

Explanation

This class is poorly encapsulated. The `totalItems` field should be calculated in the constructor and in any methods that modify the `compartments` field. Performing the `totalItems` calculation within the class only when a dependent field changes will ensure the field reflects the current object state and does not require users of the class to perform the calculation manually. This both protects the `totalItems` field and increases overall usability.

The redesigned `Container` class could resemble the following code:

```
public class Container {
   private ArrayList<Integer> compartments;
   private int totalItems;
```

```
  private void calculateTotalItems() {
    this.totalItems = 0; //reset
    for (int i = 0; i < compartments.size(); i++)
    this.totalItems += compartments.get(i);
  }

  public Container(int numCompartments) {
    this (numCompartments, 0);
  }

  public Container (int numCompartments, int itemsPerCompartment) {
    this.compartments = new ArrayList<>(numCompartments);
    for (int i = 0; i < numCompartments; i++)
      compartments.add(itemsPerCompartment);
    calculateTotalItems();
  }

  public int getTotalItems() {
    return totalItems;
  }

  public int getCompartmentTotal(int i) {
    return compartments.get(i);
  }

  public void addCompartment(int numItems) {
    compartments.add(numItems);
    calculateTotalItems();
  }

  public void addToCompartment(int compartment, int numItems) {
    compartments.set(compartment, compartments.get(compartment) + numItems);
    calculateTotalItems();
  }
}
```

This class is not properly encapsulated. If the access modifier on the constructor were changed to `private`, then users of the class would be unable to instantiate it. You should only choose this solution if instantiation must be controlled internally and is not required by all classes for encapsulation.

If the access modifier on the `compartments` field is changed to `public`, then this field would become more visible and reduce overall encapsulation.

The `compartments` field should not be set in any methods that modify the `totalItems` field. Because the `totalItems` field is calculated from the `compartments` field, the `totalItems` field should be read-only and not directly modifiable.

Objective:

Java Object-Oriented Approach

Sub-Objective:

Understand variable scopes, apply encapsulation and make objects immutable

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Language Basics > Classes and Objects > Controlling Access to Members of a Class

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Language Basics > Object-Oriented Programming Concepts > What Is an Object?

Question #13 of 50

Question ID: 1327960

Consider the following code:

```
package applic;

import java.util.ServiceLoader;
import java.util.ServiceLoader.Provider;
import package1.SpeakerInterface;

public class Client {
    public static void main(String[] args) {
        ServiceLoader.load(SpeakerInterface.class)
          .stream()
          .filter((Provider p) -> p.type().getSimpleName().startsWith("Speaker"))
          //Insert code here
          .findFirst()
          .ifPresent(t -> t.speak());
    }
}
```

What code will you insert to ensure that the correct service provider is instantiated when it is required?

A) `.map(Provider::get)`

B) `.requires modserv`

C) `.exports Provider p`

D) `.provides package1.SpeakerInterface`

Explanation

You would use the `get` method, as indicated in the following code fragment:

```
.map(Provider::get)
```

The other options are incorrect because none of them uses the `Provider` class and the `get` method to instantiate a service provider.

Service providers are usually classes, but they can also be interfaces or abstract classes, in which case they need to have static provider methods. The code for service providers can be created within a module and then placed in the module path of the application. A service provider's code can also be placed inside a JAR file in the application's class path. This kind of implementation helps ensure encapsulation by hiding all implementations of the service provider. An application can instantiate service providers by iterating the service loader or by using `Provider` objects within the stream of the service loader.

Any instance of a `ServiceLoader.Provider` interface represents a typical service provider where:

- the `type()` method will return a `Class` object of the implementation of the service
- the `get()` method will create an instance of the service provider

Using the `stream()` method will ensure that each element of the stream of type `ServiceLoader.Provider`.

The stream is then filtered based on the `type()` method ,and the `get()` method can be called to instantiate the required provider. This ensures that a provider is instantiated only when required and not when iterations are carried through several providers.

Objective:
Java Platform Module System

Sub-Objective:
Declare, use, and expose modules, including the use of services

References:

Oracle Technology Network > Java SE > Java SE Documentation > Java Platform Standard Edition 11 > API Specification > java.base > java.util > java.lang > Class ServiceLoader<S>

Oracle Technology Network > Java SE 9 and JDK 9 > ServiceLoader

---

Question #14 of 50

Question ID: 1328118

Given the code fragment:

```
Stream<String> ss = Stream.of("A", "B", "c", "D", "e");
System.out.println(ss.noneMatch(s->s.length()>2));
```

What is the result?

A) `Optional[true]`

B) No output

C) `true`

D) `false`

E) `ABcDe`

Explanation

The output is `true`. The method `noneMatch` takes a predicate and returns a `boolean`. In this case, every item that arrives at the `noneMatch` method will cause the predicate to return false, because there is only one character in each item. Because no item matches the predicate, the final value of `noneMatch` is true.

The output is not `ABcDe` because the output of `noneMatch` is a `boolean`, not a concatenation of the items in the stream.

The output is not `Optional[true]`, because the return of the `noneMatch` is a primitive `boolean`, not an `Optional<Boolean>`. This is the output of `find`*`XXX`* operations, not `XXX`*`Match`* operations.

The output is not `false`, because none of the stream items tested by the predicate returns true. Because there are no matches `noneMatch` will return true.

Output is the result, because the `noneMatch` method always returns a `boolean` value.

Objective:
Working with Streams and Lambda expressions

Sub-Objective:
Use Java Streams to filter, transform and process data

References:

Java Platform Standard Edition 11 > API > java.util.stream > Stream

Java Platform Standard Edition 11 > API > java.util > Optional<T>

Question #15 of 50

Question ID: 1327806

Given the following code fragment:

```
public class StandardMethods {
   public static double getSurfaceArea(double width, double height, double length) {
     return 2 * (width * length + height * length + height * width);
   }
}
```

Assuming the given code is in the same package, which code lines will compile? (Choose all that apply.)

A) `StandardMethods.getSurfaceArea(8.5);`

B) `double surfaceArea =StandardMethods.getSurfaceArea(8.5,11);`

C) `double surfaceArea =StandardMethods.getSurfaceArea(8.5);`

D) `StandardMethods.getSurfaceArea(8.5,11, 1.5);`

E)
```
double surfaceArea =StandardMethods.getSurfaceArea(8.5,11,
1.5);
```

F) `StandardMethods.getSurfaceArea(8.5,11);`

Explanation

The following code lines will compile:

```
StandardMethods.getSurfaceArea(8.5,11, 1.5);
double surfaceArea =StandardMethods.getSurfaceArea(8.5,11, 1.5);
```

Both code lines invoke the `getSurfaceArea` method with arguments that match the data types of the required parameters. The first code line accepts the return value as the required data type, while the second code line ignores the return value. Either invocation is valid in Java.

The code lines that omit one or two arguments will not compile. The `getSurfaceArea` method lists three parameters, so that all three arguments must be specified. By default, named parameters indicate required arguments for valid invocation.

Objective:
Java Object-Oriented Approach

Sub-Objective:
Define and use fields and methods, including instance, static and overloaded methods

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Language Basics > Classes and Objects > Passing Information to a Method or a Constructor

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Language Basics > Classes and Objects > Defining Methods

---

Question #16 of 50

Question ID: 1328188

What are the four ways to create a Locale object?

A) Use the `Locale.forLanguageTag` factory method

B) Use `Region` constructors

C) Use `Locale` constructors

D) Use the `Locale.Builder` class

E) Use constants in the `java.lang` package

F) Use constants in the `Locale` class

Explanation

The four ways to create a `Locale` object are as follows:

- Use the `Locale.Builder` class
- Use `Locale` constructors
- Use constants in the `Locale` class
- Use the `Locale.forLanguageTag` factory method

The `Locale.Builder` class and `Locale.forLanguageTag` factory method are new since Java 8.

Region constructors cannot create a `Locale` object because the `Region` class is obsolete. Also, it was used for Swing component layout, not localization.

Constants cannot be used in the `java.lang` package, because the `java.lang` package does not include constants nor any classes for localization. The contents of the `java.lang` package are fundamental to Java applications and do not need to be imported.

Objective:
Localization

Sub-Objective:
Implement Localization using Locale, resource bundles, and Java APIs to parse and format messages, dates, and numbers

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Internationalization > Setting the Locale > Creating a Locale

---

Question #17 of 50

Question ID: 1328150

Which two methods of the `ExecutorService` class support a single `Runnable` argument?

A) `call`

B) `run`

C) `submit`

D) `execute`

E) `start`

Explanation

Both the `submit` and `execute` methods support a single `Runnable` argument. When executing a task using the `ExecutorService` class, you can invoke either the `execute` or `submit` methods. The `execute` method supports `Runnable` objects that do not return a value, while the `submit` method supports both `Runnable` and `Callable` objects. The `submit` method returns a `Future` object representing the pending results of the task.

The `call` method is not provided by the `ExecutorService` class. The `call` method of the `Callable` interface returns a result or throws an exception.

The `run` method is not provided by the `ExecutorService` class. The `run` method of the `Runnable` interface performs a task but does not return a result or contain an exception in its declaration.

The `start` method is not provided by the `ExecutorService` class. The `start` method of the `Thread` class executes a thread as a child of the current thread.

Objective:
Concurrency

Sub-Objective:
Create worker threads using Runnable and Callable, and manage concurrency using an ExecutorService and java.util.concurrent API

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Essential Classes > Concurrency > Executor Interfaces

Oracle Technology Network > Java SE > Java SE Documentation > Java Platform Standard Edition 11 API Specification > java.util.concurrent > Interface Callable<V>

Question #18 of 50

Question ID: 1328097

Given the code fragment:

```
/* insert here */
  .forEach(System.out::println);
```

Which code added at `/* insert here */` would result in the output 1, 2, 3, and 4?

A) `IntStream.range(1, 5)`

B) `IntStream.iterate(1, 4)`

C) `IntStream.range(1, 4)`

D) `IntStream.rangeClosed(1, 5)`

E) `IntStream.of(1, 4)`

Explanation

The code `IntStream.range(1, 5)` will result in the output 1, 2, 3, and 4. The `IntStream.range` method takes two `int` arguments. The first argument specifies the initial value of the stream. Each subsequent stream element is then calculated as the previous value increased by one for as long as the value remains less than the second argument.

`IntStream.range(1, 4)` will result in a stream containing the numbers 1, 2, and 3.

`IntStream.of(1, 4)` creates a stream with the elements 1 and 4. The method `IntStream.of` takes a variable length argument list and creates a stream of int values containing the same elements that were provided in the array.

`IntStream.iterate(1, 4)` is not a function, so it will fail to compile. The iterate method takes two arguments. The first is an initial value, which becomes the first item in the stream. The second argument is a function that is called repeatedly to produce the next stream value based on the previous one.

`IntStream.rangeClosed(1, 5)` will output the numbers from 1 to 5 inclusive. The `rangeClosed` method is like the `range` method except that it includes the value of the second argument in the generated stream.

Objective:
Working with Streams and Lambda expressions

Sub-Objective:
Use Java Streams to filter, transform and process data

References:

Java Platform Standard Edition 11 > API > java.util.stream > IntStream

---

Question #19 of 50

Question ID: 1328122

Given:

```
System.out.println(
  IntStream.iterate(0, (n)->n+1)
  .limit(5)
  // line n1
  );
```

Which code fragment should be inserted at line `n1` to generate the output `5`?

A) `.sum(System.out::println)`

B) `:.count()`

C) `.max()`

D) `.reduce(0, (a,b)->a+b, v->System.out.println(v));`

Explanation

You would insert the `count` operation. This method counts the number of items in the stream. Given the `limit(5)` operation, this will result in a value of 5 for the entire stream process. Because the stream pipeline is the argument to the `System.out.println` call, the code will print out `5`.

You should not choose `.reduce(0, (a,b)->a+b, v->System.out.println(v));` because it has an invalid third argument and will not compile. The third argument should be `BinaryOperator`, not `Consumer`.

You should not choose `.sum(System.out::println)` because this method does not accept any arguments and the result would not be `5`. Given the numeric values 0,1,2,3,4 occurring five times in a stream, the sum of all the items in the stream would be `10`.

You should not choose `.max()` because the result would be `Optional[4]`. The `max()` method returns the top value in the stream as an `Optional`, in case the stream is empty.

Objective:
Working with Streams and Lambda expressions

Sub-Objective:
Use Java Streams to filter, transform and process data

References:

Java Platform Standard Edition 11 > API > java.util.stream > Stream

---

Question #20 of 50

Question ID: 1328066

Given a method declaration:

```
public void doStuff(Function<String, String> f)
```

Which of the following are valid arguments to invoke `doStuff`? (Choose all that apply.)

A) `s->s`

B) `String s->"Message is: " + s`

C) `s->s.length()`

D) `(final s) -> s + "."`

E) `()->f.apply("")`

Explanation

`s->s` is the only valid argument to invoke `doStuff`. This is a well-formed lambda expression that defines behavior that takes and returns a `String`; therefore, this construction can implement `Function<String, String>`.

The zero-argument lambda `()->f.apply("")` is not correct. The `Function<String, String>` defines a method that takes a single argument of type `String` and returns a `String`.

`s->s.length()` is not correct because the return type must also be `String`, not an `int` type.

`String s->"Message is: " + s` is not correct. If a single argument lambda is being created, you can omit the parentheses around that argument. However, if the type is being explicitly specified, or if modifiers will be used, then you are not permitted to omit the parentheses.

`(final s) -> s + "."` is not correct because the `final` keyword cannot be used in this way without also including the type.

Objective:
Working with Streams and Lambda expressions

Sub-Objective:
Implement functional interfaces using lambda expressions, including interfaces from the java.util.function package

References:

The Java Tutorials > Learning the Java Language > Classes and Objects > Syntax of Lambda Expressions

Java Language Specification > Java SE 11 Edition > Lambda Expressions > Lambda Parameters

---

Question #21 of 50

Question ID: 1328146

Given the code statement:

```
int [] r = IntStream.range(1, 100)
  .collect(()->new int[1], (a,b)->a[0]+=b, /* insert here */);
```

Which code fragment should be inserted at `/* insert here */` to complete the collection?

A) `(a,b)->a[0]+=b[0]`

B) `(c,d)->d[0]+=c`

C) `(c,d)->c[0]+=d`

D) `(a,b)->a[0]+=b`

Explanation

You should insert the code fragment `(a,b)->a[0]+=b[0]`. The `collect` method that takes three arguments is defined in terms of two types: the type of the data in the stream as *T*, and the type of the collected result, referred to as *R*.

This `collect` method requires three arguments. The first is a *Supplier<R>* used to create empty buckets for collecting intermediate results. The second argument is an *ObjIntConsumer<R>*, which is used as the accumulator. This adds individual stream items into a bucket. Notice that in this example, *R* is array of one `int`.

The third argument is used to merge intermediate results in buckets (that is in the type *R*). The argument type is `BiConsumer<R,R>`, so the lambda provided must take the contents of one array and add it into the other array. The only plausible code fragment is the one that extracts the value from one array and adds it into the other.

The other code fragments treat one of the lambda arguments as an `int` and are incorrect because both arguments must be array of `int`.

Objective:
Working with Streams and Lambda expressions

Sub-Objective:
Perform decomposition and reduction, including grouping and partitioning on sequential and parallel streams

References:

Java Platform Standard Edition 11 > API > java.util.stream > IntStream > collect

---

Question #22 of 50

Question ID: 1328133

You need to perform stream operations on a collection of `User` objects. Consider the code below:

```
01  List<Integer> userIDs =
02  // INSERT CODE
03  .filter(u -> u.getDep() == User.IT)
04  .sorted(comparing(User::getValue).reversed())
05  .map(User::getUserID)
06  .collect(toList());
```

Which code snippet should you insert at line `02` to make the stream processing as efficient as possible?

A) `users.parallelStream()`

B) `users.stream().stream()`

C) `users.stream().users.stream()`

D) `users.users.stream()`

Explanation

You should insert `users.parallelStream()` at line `02` to make the stream processing as efficient as possible:.

```
01  List<Integer> userIDs =
02  users.parallelStream()
03  .filter(u -> u.getDep() == User.IT)
04  .sorted(comparing(User::getValue).reversed())
05  .map(User::getUserID)
06  .collect(toList());
```

The other options are syntactically incorrect and will not create parallel streams. Only the `parallelStream` method creates a parallel stream in Java. When you call the appropriate method for creating a parallel stream, the Java streams API decompose queries internally automatically to make use of multiple cores on a computer. This is what results in a more efficient processing by the system.

The option `users.stream().stream()` is incorrect as it tries to create a stream from a stream.

The option `users.stream().users.stream()` is incorrect as the syntax for streams does not allow using a collection after a `Stream` method.

The option `users.users.stream()` is incorrect because you cannot join two instances of the same collection using the dot operator. This is syntactically incorrect.

Objective:
Working with Streams and Lambda expressions

Sub-Objective:
Perform decomposition and reduction, including grouping and partitioning on sequential and parallel streams

References:

Oracle Technology Network > Java SE > Articles > Processing Data with Java 8 Streams

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Essential Classes > Basic I/O > Object Streams

---

Question #23 of 50

Question ID: 1327938

You need to create an immutable list of employee types for a company. Which code segment(s) will meet these requirements? (Choose all that apply.)

A)
```
Map<Integer, String> employee =
Map.ofEntries("trainee","executive","manager","director","ceo");
```

B)
```
List<String> employee =
Map.asList("trainee","executive","manager","director","ceo");
```

C)
```
List<String> employee =
List.of("trainee","executive","manager","director","ceo");
```

D)
```
Map<String> employee = Map.of("trainee", "executive",
"manager", “director”, “ceo”);
```

Explanation

You should use the `List.of()` factory method. This creates an immutable list from the arguments entered. The following code would create the `List` from the entered elements:

```
List<String> employee = List.of("trainee","executive","manager","director","ceo");
```

An indefinite number of elements can be entered for inclusion in the immutable list using the `List.of()` method.

Starting with Java 9, Java includes a collection library that provides static factory methods for the interfaces `List`, `Set`, and `Map`. You can use these methods for creating immutable instances of collections. An immutable collection is a collection that cannot be modified after it is created. This includes the references made by the collections, their order, as well as the number of elements. Attempting to modify an immutable collection results in a `java.lang,UnsupportedOperationException` being thrown.

A key difference between `List` and `Set` is that a `List` allows duplicate entries but a `Set` does not. A `Map` has key value pairs, but key values are never duplicates. Additionally, a `List` is an ordered collection while a `Set` is not.

Using the `Map.asList()` method is incorrect because you need a `List` method and not a `Map` method to create the list. You would use the requisite `Map` method when you need to create key-value pairs for data.

Using the `Map.of()` factory method is an incorrect option. You use the `Map.of()` factory method for creating mappings, for example between numbers and names.

Using the `Map.ofEntries()` factory method is an incorrect option. You would use this method when you need to create a map using instances of `Map.Entry`.

Objective:
Working with Arrays and Collections

Sub-Objective:
Use a Java array and List, Set, Map and Deque collections, including convenience methods

References:

Oracle Technology Network > Java > Oracle JDK9 Documentation > Java Platform, Standard Edition Core Libraries > Creating Immutable Lists, Sets, and Maps

---

Question #24 of 50

Question ID: 1327875

Given:

```
public class SuperString {
   public String toString() {
     return "Super String";
   }
   public Object toString(String str) {
     return "Super " + str;
   }
}

class SubString extends SuperString {
   public String toString() {
     return "Sub String";
   }
   public String toString(String str) {
     return "Sub " + str;
   }
}
```

Which two statements will generate the output `Super String`?

A) `System.out.println(((Object) new SubString()).toString());`

B) `System.out.println(((Object) new SuperString()).toString("String"));`

C) `System.out.println(((Object) new SuperString()).toString());`

D) `System.out.println(((Object) new SubString()).toString("String"));`

E) `System.out.println(new SubString().toString());`

F) `System.out.println(new SuperString().toString("String"));`

Explanation

The following two statements will generate the output `Super String`:

`System.out.println(new SuperString().toString("String"));`

`System.out.println(((Object) new SuperString()).toString());`

The first statement instantiates the `SuperString` class and then invokes the second overloaded version of the `toString` method. By feeding the literal `String` as the argument for the parameter, the output is `Super String`. The second statement instantiates the `SuperString` class, casts the reference as an `Object` type, and then invokes the first version of the overloaded `toString` method. Because the actual object is a `SuperString` object, the implementation in the `SuperString` class is executed.

The statements that instantiate the `SubString` class and invoke the parameterless version of `toString` will not generate the output `Super String`. This is because no versions of the `toString` method in `SubString` class return the text `Super, only Sub`.

The statements that cast the `SuperString` or `SubString` object to an `Object` type and then invoke the `toString` method with a `String` argument will not generate the output `Super String`. The statements will fail to compile because the `Object` class does not provide a `toString` method with a `String` parameter.

Objective:
Java Object-Oriented Approach

Sub-Objective:
Utilize polymorphism and casting to call methods, differentiate object type versus reference type

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Interfaces and Inheritance > Overriding and Hiding Methods

---

Question #25 of 50

Question ID: 1328110

Given the contents of the UTF-8 encoded file `quick.txt`:

```
The quick brown fox
jumped
over the lazy
dog.
```

And the code fragment:

```
Files.lines(Paths.get("quick.txt"))
  .flatMap(s->s.split("\\W+"))  // line n1
  .forEachOrdered(System.out::println);
```

Assuming that the `String.split` method returns an array of strings, and that the file `quick.txt` is readable and in the current directory, what is the result?

A) Compilation fails at line `n1`.

B)
```
The
quick
brown
fox
jumped
over
the
lazy
dog
```

C)
```
The quick brown fox jumped over the lazy dog.
```

D)
```
The quick brown fox
jumped
over the lazy
dog.
```

Explanation

Compilation fails at line `n1`.

The `flatMap` method requires an argument that is a `Function`. The `Function` must accept the stream item type as an argument and return a `Stream` of some type (which may be the same type as, or a different type from, the input type.) In this case, since the argument to the `flatMap` method is a lambda that takes the input string and converts it to an array of string, the lambda has the wrong return type for the `flatMap` method, and the code fails to compile.

Given that the code does not compile, no output is possible.

Objective:
Working with Streams and Lambda expressions

Sub-Objective:
Use Java Streams to filter, transform and process data

References:

Java Platform Standard Edition 11 > API > java.util.stream > Stream

---

Question #26 of 50

Question ID: 1327873

Given the classes:

```
class Candidate {
   String title = "Uncertified";
   public int getStudyHours() {return 0;}
}
```

```
class JavaICandidate extends Candidate {
   String title = "Oracle Certified Associate";
   public int getStudyHours() { return 20 * 3; }
}

class JavaIICandidate extends JavaICandidate {
   String title = "Oracle Certified Professional";
   public int getStudyHours() { return 40 * 3; }
}
```

Which one of the following code fragments demonstrates polymorphism?

A)
```
Candidate c = new JavaIICandidate();
System.out.println(c.title);
```

B)
```
JavaIICandidate c = new JavaICandidate();
System.out.println(c.title);
```

C)
```
JavaIICandidate c = new JavaICandidate();
System.out.println(c.getStudyHours());
```

D)
```
Candidate c = new JavaIICandidate();
System.out.println(c.getStudyHours());
```

Explanation

The following code fragment demonstrates polymorphism:

```
Candidate c = new JavaIICandidate();
System.out.println(c.getStudyHours());
```

When an overridden method is invoked on an object, the version of the method is determined by the instantiated class, even if the reference type of the variable is of a different type from the class. This is known as polymorphism. In this code, the reference type of variable `c` is `Candidate`, but the instantiated class is `JavaIICandidate`. Thus this code will output `120`, not 0.

The code fragments that reference the `title` field do not demonstrate polymorphism. Because title is declared as a field in both `JavaICandidate` and `JavaIICandidate`, the field in `Candidate` is effectively hidden for these classes. Polymorphism does not apply in this scenario. This means that the reference type determines which field is accessed. If the reference type is `Candidate`, then the value of `title` will be `Uncertified`, regardless of the instantiated class.

The code fragments that instantiate `JavaICandidate` and assign the object to a variable of type `JavaIICandidate` do not demonstrate polymorphism. Supertypes cannot be assigned to subtype variables, so these code fragments will fail to compile.

Objective:
Java Object-Oriented Approach

Sub-Objective:

Utilize polymorphism and casting to call methods, differentiate object type versus reference type

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Interfaces and Inheritance > Polymorphism

---

Question #27 of 50

Question ID: 1327931

Given the code fragment:

```
List<String> grades = new ArrayList<>();
grades.addAll(Arrays.asList(new String[]{"C", "A", "B", "A", "D"}));
Collections.rotate(grades,2);
```

Which two statements will output `A`?

A) `System.out.print(grades.get(0));`

B) `System.out.print(grades.remove(4));`

C) `System.out.print(grades.remove(2));`

D) `System.out.print(grades.remove(3));`

E) `System.out.print(grades.get(2));`

F) `System.out.print(grades.get(1));`

Explanation

The statements `System.out.print(grades.get(0));` and `System.out.print(grades.remove(3));` will output `A`. In the sample code, the `rotate` method shifts elements to the right by 2 positions. The element A is located at index positions 0 and 3. Both the `get` and `remove` methods return the element at the specified position, but the `remove` method also deletes the element and shifts all subsequent elements to the left.

The statement `System.out.print(grades.get(1));` will output `D`, not `A`.

The statements `System.out.print(grades.get(2));` and `System.out.print(grades.remove(2));` will output `C`, not `A`.

The statement `System.out.print(grades.remove(4));` will output `B`, not `A`.

Objective:

Working with Arrays and Collections

Sub-Objective:

Use a Java array and List, Set, Map and Deque collections, including convenience methods

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Collections > Interfaces > The List Interface

---

Question #28 of 50

Question ID: 1328127

Given the code fragment:

```
Stream.of("5","7","3","2","4","1","6")
  // line n1
  .forEach(System.out::print);
```

Which code should be inserted at line `n1` to generate the output `1234567`?

A) `.sort((a,b)->a.compareTo(b))`

B) `.sorted((a,b)->a.compareTo(b))`

C) `.sorted(String::compare)`

D) `.sort()`

Explanation

You should insert the code `.sorted((a,b)->a.compareTo(b))` at line `n1`. The `Stream` class has two `sorted()` methods. One version takes a `Comparator` as a parameter and uses that to sort the items coming down the stream. The other version takes no arguments and depends on the items in the stream implementing `Comparable`. Because the `String` class implements `Comparable`, the code `sorted()` would also have worked.

The code `.sorted(String::compare)` is incorrect because the instance comparison method that implements the `Comparable` interface is called `compareTo`, not `compare`.

The code segments that use the `sort()` method are incorrect because no such method is provided by the `Stream` class.

Objective:
Working with Streams and Lambda expressions

Sub-Objective:
Use Java Streams to filter, transform and process data

References:

Java Platform Standard Edition 11 > API > java.util.stream > Stream

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Essential Classes > Basic I/O > Object Streams

Question #29 of 50

Question ID: 1328041

Given:

```
public void copy(Path srcFile, Path destFile) {
   try {
     byte[] readBytes = Files.readAllBytes(srcFile);
     Files.write(destFile, readBytes);
   } catch (_____________ e ) {
     System.err.println(e.toString());
   }
}
```

Which insertion will allow the code to compile?

A) `FileSystemNotFoundException`

B) `IOError`

C) `Error`

D) `FileNotFoundException`

E) `IOException`

Explanation

To allow the code to compile, the insertion should specify the class `IOException`. Although it was not given as an alternative, the insertion could also specify its superclass `Exception`. The `readAllBytes` and write methods specify that they throw `IOException`, so any invocations of those methods must either catch `IOException` or specify that the parent method throws that exception. This is known as a checked exception. Checked exceptions include any exceptions, except for `RuntimeException` and its subclasses.

The code will not compile if the insertion specifies `Error` or `IOError`. The `readAllBytes` and `write` methods require handling `IOException` or its superclass `Exception`. Errors are abnormal conditions external to the application and often are unrecoverable. The compiler does not check catching and specifying `Error` and its subclasses.

The code will not compile if the insertion specifies `FileNotFoundException`. The `readAllBytes` and `write` methods require handling `IOException` or its superclass `Exception`. Subclasses of the `IOException` can be specified in the `catch` clause, but the more general `IOException` must be also caught, or the code will not compile.

The code will not compile if the insertion specifies `FileSystemNotFoundException`. The `readAllBytes` and `write` methods require handling `IOException` or its superclass `Exception`. Runtime exceptions are abnormal conditions internal to the application and often are unrecoverable. The compiler does not check catching and specifying `RuntimeException` and its subclasses.

Objective:
Exception Handling

Sub-Objective:
Handle exceptions using try/catch/finally clauses, try-with-resource, and multi-catch statements

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Essential Classes > Exceptions > The Catch or Specify Requirement

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Essential Classes > Exceptions > The catch Blocks

---

Question #30 of 50

Question ID: 1328093

You need to select and print the names of all employees from a collection of `User` objects that are based out of Atlanta. Which method(s) of the `Stream` interface will you use for this?

A) `collect()`

B) `new`

C) `reduce`

D) `filter()`

E) `forEach()`

F) `thread`

Explanation

You will use the `filter`, and `forEach` methods to select the names of all employees from a collection of `User` objects.

```
List<User> basedinLA = users.stream() .filter(e -> e.getLocation() == "Atlanta") .map(e -> e.getName().forEach(System.out::println);
```

The `collect` method is an incorrect option in this case. You use the `Stream.collect` method to modify a single value based on the elements of the stream.

The `reduce` method is an incorrect option in this case. You use the `Stream.reduce` method to make one value out of all the elements of the stream.

The `Thread` class in Java allows you to create multiple threads of execution in a Java program, and so is an incorrect option here.

The `new` keyword is used to create a new instance of a class, and so is an incorrect option here.

Objective:

Working with Streams and Lambda expressions

Sub-Objective:

Use Java Streams to filter, transform and process data

References:

Java Platform Standard Edition 11 > API > java.util.stream > Stream

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Essential Classes > Basic I/O > Object Streams

---

Question #31 of 50

Question ID: 1327898

Given the following classes:

```
class CongressionalCandidate implements Electable {
   public String getCitizenRequirement() { return "Naturalized Citizen"; }
}

class PresidentialCandidate implements Electable {
   public String getCitizenRequirement() { return "Birthright Citizen"; }
}
```

Which code fragment correctly defines Electable?

A)
```
interface Electable {
   String getCitizenRequirement();
}
```

B)
```
abstract class Electable {
   public String getCitizenRequirement() { return "General
Citizen"; }
}
```

C)
```
abstract class Electable {
   abstract String getCitizenRequirement();
}
```

D)
```
interface Electable {
   public String getCitizenRequirement() { return "General
Citizen"; }
}
```

Explanation

The following code fragment correctly defines `Electable`:

```
interface Electable {
   String getCitizenRequirement();
}
```

Because `CongressionalCandidate` and `PresidentialCandidate` both use the `implements` keyword, `Electable` must be declared as an interface. `Electable` should contain the method `getCitizenRequirement` overridden in `CongressionalCandidate` and `PresidentialCandidate`.

By default, all methods declared in an interface are implicitly abstract and public, so implementing classes must use the `public` keyword as well. Since Java 9, interfaces can also support private methods for code reuse without providing visibility to implementing classes.

The code fragments that declare abstract classes do not correctly define `Electable`. The `implements` keyword requires an interface, not an abstract class, for implementing classes. The `extends` keyword is required for an abstract class because implementing classes must use inheritance.

The code fragment that declares an interface with a method body does not correctly define `Electable`. Methods in an interface cannot contain a body and are implicitly abstract and public.

Objective:
Java Object-Oriented Approach

Sub-Objective:
Create and use interfaces, identify functional interfaces, and utilize private, static, and default methods

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Interfaces and Inheritance > Defining an Interface

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Implementing an Interface

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Interfaces and Inheritance > Overriding and Hiding Methods

---

Question #32 of 50

Question ID: 1327818

Given:

```
public class Circle {
   public double getCircumference(double radius ) {
     return java.lang.Math.PI * 2 * radius;
   }
```

```
   public static double getArea(double radius) {
     return java.lang.Math.PI * radius * radius;
   }
}
```

Which two code fragments will fail compilation?

A) `double a = new Circle().getArea(5.5);`

B) `new Circle().getArea(5.5);`

C) `double c = Circle.getCircumference(10.5);`

D) `new Circle().getCircumference(10.5);`

E) `Circle.getCircumference(10.5);`

F) `Circle.getArea(5.5);`

Explanation

The following code fragments will fail compilation:

```
Circle.getCircumference();
double c = Circle.getCircumference(10.5);
```

These code fragments fail compilation because the `getCircumference` method is an instance method and cannot be invoked as a `static` class method. The code must instantiate the `Circle` class before accessing the `getCircumference` method.

The other code fragments will compile because the `getCircumference` method is accessible from an instance context and the `getArea` method is accessible from either a `static` or instance context.

Objective:
Java Object-Oriented Approach

Sub-Objective:
Define and use fields and methods, including instance, static and overloaded methods

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Language Basics > Classes and Objects > Understanding Instance and Class Members

---

Question #33 of 50

Question ID: 1328006

Consider the following code:

```
class Jedi {
 public void Speak() {
  System.out.println("May the force be with you");
 }
 public static void main(String args[]) {
  Jedi j1 = new Skywalker();
  j1.Speak();
 }
}

// Insert Annotation Here
public @interface SkywalkerChronicles {
 // Code omitted
}

@SkywalkerChronicles
class Skywalker extends Jedi {
 public void Speak() {
  System.out.println("May the 4th");
 }
}
```

You need to ensure that the `@SkywalkerChronicles` annotation is included by JavaDoc. What will you use for this?

A) `@SupressWarnings`

B) `@Documented`

C) `@Override`

D) `@Deprecated`

Explanation

You should use the `@Documented` annotation. This annotation is used to indicate that all elements that use the annotation are documented by JavaDoc.

`@Deprecated` is incorrect because it does not indicate annotation inclusion to JavaDoc. It is a marker annotation indicating that the associated declaration is has now been replaced with a newer one.

`@Override` is incorrect because it does not indicate annotation inclusion to JavaDoc. It is a marker annotation only to be used with methods that override methods from the parent class. It helps ensure methods are overridden and not just overloaded.

`@SupressWarnings` is incorrect because it does not indicate annotation inclusion to JavaDoc. It is an annotation that specifies warnings in string form that the compiler must ignore.

Java has seven predefined annotation types:

- `@Retention` – This indicates how long an annotation is retained. It has three values: SOURCE, CLASS, and RUNTIME.
- `@Documented` – This indicates to tools like Javadoc to include annotations in the generated documentation, including the type information for the annotation.
- `@Target` – This is meant to be an annotation to another annotation type. It takes a single argument that specifies the type of declaration the annotation is for. This argument is from the enumeration `ElementType`:
  - ANNOTATION_TYPE – This is used for another annotation
  - CONSTRUCTOR – For constructors
  - FIELD – For fields
  - METHOD – For methods
  - LOCAL VARIABLE – For local variables
  - PARAMETER – For parameters
  - PACKAGE – For packages
  - TYPE – This can include classes, interfaces or enumerations
- `@Inherited` – This can only be used on annotation declarations. It makes an annotation for a superclass become inherited by a subclass. It is used only for annotations on class declarations.
- `@Deprecated` – This is a marker annotation indicating that the associated declaration is has now been replaced with a newer one.
- `@Override` – This is a marker annotation only to be used with methods that override methods from the parent class. It helps ensure methods are overridden and not just overloaded.
- `@SupressWarnings` – This specifies warnings in string form that the compiler must ignore.

Objective:
Annotations

Sub-Objective:
Create, apply, and process annotations

References:

Oracle Technology Network > Java SE Documentation > Annotations

Oracle Technology Network > Java SE Documentation > Annotations > Predefined Annotation Types

---

Question #34 of 50

Question ID: 1328144

Given:

```
int [] res = IntStream.of(1,2,3,4,5,6,7,8,9,10)
  .parallel()
  .collect( ()->new int[2],  // line n1
  (a,b)->{if (b%2==0) a[1]+=b; else a[0]+=b;}, // line n2
```

```
  (a,b)->{a[0]+=b[0];a[1]+=b[1];});
System.out.println("Odd sum = " + res[0] + " even sum = " + res[1]);
```

What is the result?

A) Non-deterministic output

B) Compilation fails at line `n2`

C) Compilation fails at line `n1`

D) `Odd sum = 25 even sum = 30`

E) `Odd sum = 55 even sum = 55`

Explanation

The result is the following output:

```
Odd sum = 25 even sum = 30
```

This collect method takes three arguments. The first is a `Supplier<R>` where *R* is the result type of the collection operation. In this case, the supplier creates an array of two `int` values, and that is consistent with the result type for the operation.

The second argument for this collector is a `BiConsumer<R,T>` where *T* is the stream type. Since the stream contains integers, these might be `int` primitives or `Integer` objects. The behavior of the block lambda is to test if the second argument (the item from the stream) is odd or even, and to add that stream value to one or the other element of the array depending on whether the stream value is odd or even. The arguments are `int[2]` and `int`/`Integer`, and are used correctly for those types.

The return type of the `BiConsumer` is `void`, and the block of the lambda does not return anything. The resulting behavior is a collection of the sums for odd and even numbers seen in the two elements of the array of int, and to output the odd sum, *25*, and the even sum, *30*.

The output will not have the same value of 55 for both odd and even numbers because the lambda expression in the second argument checks for whether items are odd or even and third argument increments these values independently.

The output is deterministic. The collect operation differs from a reduce operation in that each thread that is created in a parallel stream situation is given its own mutable storage for collecting intermediate results. Because each thread is given its own mutable storage, the resulting output is predictable and expected.

The code at line `n1` is correct. It defines a supplier of `int[2]`, which is appropriate as the first argument to the collector and compatible with the rest of the collector arguments.

The code at line `n2` is also correct. It defines a lambda that is compatible with `BiConsumer<int[2], int>`.

Objective:
Working with Streams and Lambda expressions

Sub-Objective:
Perform decomposition and reduction, including grouping and partitioning on sequential and parallel streams

References:

The Java Tutorials > Collections > Aggregate Operations > Parallelism

Java Platform Standard Edition 11 > API > java.util.stream > Stream

---

Question #35 of 50

Question ID: 1327952

You are designing a program that requires multiple types of authentication modules. You also need a graphical GUI to manage and monitor Java applications that are running.

Which JDK modules will allow you to meet this requirement? (Choose two.)

A) `java.base`

B) `jdk.security.auth`

C) `jdk.jconsole`

D) `jdk.jdi`

E) `java.compiler`

Explanation

The JDK modules you should select are `jdk.security.auth` and `jdk.jconsole`. The module `jdk.security.auth` provides four packages, which include `Principal`, `CallbackHandler`, `Configuration`, and `LoginModule`. These packages have several classes that allow you to create and manage security authentication within your Java program. The module `jdk.jconsole` has a key package named `com.sun.tools.jconsole`, with the `JConsolePlugin` class for a custom console connection to an application.

The `java.base` represents the base APIs within the Java SE platform.

The `java.compiler` is part of the Java SE platform. This module provides the Java compiler interface and is considered a language model.

The `jdk.jdi` is a JDK module, but this module is used as a Java debug interface that allows you to control the execution of Java programs on virtual machines. This module also provides a command line debugger called `jdb`.

Objective:
Java Platform Module System

Sub-Objective:
Deploy and execute modular applications, including automatic modules

References:

docs.oracle.com > Java® Platform, Standard Edition & Java Development Kit Version 11 API Specification

Question #36 of 50

Question ID: 1328018

Which statement is true about `if` statements nested in `if` and `else` statements?

A) The outer `else` statement is evaluated only if the inner `if` statement(s) are evaluated.

B) The inner `if` statement(s) are evaluated only if the outer `if` statement is evaluated.

C) The inner `if` statement(s) are evaluated only if the outer `if` statement is `true`.

D) The outer `else` statement is evaluated only if the inner `if` statement(s) are `true`.

Explanation

The inner `if` statement(s) are evaluated only if the outer `if` statement is `true`. If the `if` statement is `false`, then the inner `if` statement(s) are not evaluated.

The inner `if` statement(s) are not evaluated only if the outer `if` statement is evaluated. The inner `if` statement(s) are not evaluated if the outer `if` statement evaluates to `false`.

The outer `else` statement is not evaluated only if the inner `if` statement(s) are evaluated or evaluated as `true`. Whether inner `if` statement(s) are evaluated or evaluated as `true` does not affect whether the outer `else` statement is evaluated. The `else` statement is reached when its associated `if` statement(s) are `false`.

Objective:
Controlling Program Flow

Sub-Objective:
Create and use loops, if/else, and switch statements

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Language Basics > The if-then Statement

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Language Basics > Equality, Relational and Conditional Operators

Question #37 of 50

Question ID: 1327921

Given:

```
public enum WaterTemperature {
  BOILING(100), FREEZING(0);
  private double temp;
  private WaterTemperature(double temp) {this.temp = temp;}
  public String getCelsius() {return temp + "\u00B0 C";}
  public String getFarenheit() {return (temp * 9 / 5 + 32) + "\\u00B0 F";}
  public String getKelvin() {return (temp + 273.15) + " K";}
}
```

Which code fragment will generate the output `100.0 C`?

A) `System.out.println(WaterTemperature.BOILING.values());`

B) `System.out.println(WaterTemperature.BOILING.name());`

C) `System.out.println(WaterTemperature.BOILING.getCelsius());`

D) `System.out.println(WaterTemperature.BOILING);`

Explanation

The following code fragment will generate the output `100.0 C`:

`System.out.println(WaterTemperature.BOILING.getCelsius());`

To access an instance member on an enumeration, you can use an enumeration variable or invoke the instance member from the enumeration constant directly.

The code fragments `System.out.println(WaterTemperature.BOILING);` and `System.out.println(WaterTemperature.BOILING.name());` will not generate the output `100.0 C`. They both will output the name of the enumeration constant: `BOILING`.

The code fragment `System.out.println(WaterTemperature.BOILING.values())`; will not generate the output `100.0 C`. The `values` method returns an array of all enumeration constants. The output will be the name and hash code of the array.

Objective:
Java Object-Oriented Approach

Sub-Objective:
Create and use enumerations

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Classes and Objects

Oracle Technology Network > Java SE > Java SE Documentation > Java Platform Standard Edition 8 API Specification > java.lang > Class Enum

Question #38 of 50

Question ID: 1328083

Consider following code:

```
import java.util.function.*;

public class NumberWorks {
  public static void main(String[] args) {
    IntPredicate iP = (a)-> a == 1980;
    System.out.println(iP.test(0));
  }
}
```

What is the output?

A) `true`

B) `0`

C) `false`

D) `1980`

Explanation

The output is false because the `IntPredicate` functional interface implements the lambda expression `(a)-> a == 1980`. This evaluates to `true` if the number passed to it via the `test` method is equal to `1980`. The code in the scenario passes it the value `0`, which is less than `1980`.

The other options are incorrect because the lambda expression neither evaluates to `true` nor returns another value. It returns `false` because the primitive passed to it does not satisfy its condition of equality.

Generic parameters, like `T` in functional interfaces like `Predicate`, refer to reference types:

```
interface Predicate<T> {
  boolean check(T t);
}
```

These generic parameters do not work with primitives like `int`, `float`, or `double`. The process of converting primitive types like int to reference types like `Integer` is called boxing. The reverse process is called *autoboxing*.

The process of boxing and unboxing has a large performance overhead. To counter this, Java provides primitive versions of functional interfaces to allow you to use primitive types as inputs and outputs for the functional interfaces. Using primitive versions of functional interfaces improves performance by avoiding autoboxing operations.

For example, `IntPredicate` provides a functional interface for integers:

```
package java.util.function;

@FunctionalInterface
public interface IntPredicate {
public boolean test(int i);
/* code that implements this */
}

IntPredicate intpred = i -> i == 1986;
intpred.test(1986);
```

You can similarly use a functional interface for a primitive type by prefixing the primitives name with the functional interface you want to use. Here are some examples:

- `IntConsumer` - This accepts one int argument and returns void.
- `IntToDoubleFunction` - This takes an int as argument and returns a double-valued result.
- `BooleanSupplier` - This is a `Supplier` functional interface that provides boolean-valued results.
- `ObjIntConsumer<T>` - This takes an Object and an int as arguments and returns no result.

Objective:
Working with Streams and Lambda expressions

Sub-Objective:
Implement functional interfaces using lambda expressions, including interfaces from the java.util.function package

References:

Oracle Technology Network > Java SE > Java SE Documentation > Java Platform Standard Edition 11 API Specification >Java.util.function > Interface IntPredicate

Java Platform Standard Edition 11 > API > Package java.util.function

---

Question #39 of 50

Question ID: 1327819

Which code fragment correctly declares a method as a class member?

A) `final void methodD() {}`

B) `public void methodB() {}`

C) `static void methodC() {}`

D) `void methodA() {}`

Explanation

The following code fragment declares a method as a class member:

```
static void methodC() {}
```

The keyword `static` indicates a class member. This is the only keyword required for declaring a method as a class member.

The code fragments that omit the `static` keyword do not declare the method as a class member. The `public` keyword increases a method's visibility, while the `final` keyword prevents overriding in subclasses.

Objective:
Java Object-Oriented Approach

Sub-Objective:
Define and use fields and methods, including instance, static and overloaded methods

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Language Basics > Classes and Objects > Understanding Instance and Class Members

---

Question #40 of 50

Question ID: 1327963

Consider the following code output:

```
D:\lord-java>jar --create --file service-client.jar -C service-client .

D:\lord-java>jar --describe-module --file=service-client.jar

modClient jar:file:///D:/lord-java/service-client.jar/!module-info.class
requires java.base mandated
requires modServ
uses package1.SpeakerInterface
contains app
```

Based on this output, which module does the client module depend on?

A) `package1`

B) `modServ`

C) `modProv`

D) `SpeakerInterface`

Explanation

The client module depends on `modServ`. The provider module also depends on `java.base`.

The other options are incorrect because the code output specifies mandatory modules by the indicating via the keyword `requires`. Neither `package1`, `SpeakerInterface` nor `modProv` have *requires* preceding them in the code output. The client module `modClient` does NOT depend on `modProv`, which is the provider module. Additionally, `modClient` is unaware of `modProv` until the time of compilation.

When service providers are deployed as modules, they need to specified using the `provides` keyword within the declaration of the module. Using the `provides` directive helps specify the service as well as the service provider. This directive helps to find the service provider if another module that has a `uses` directive for the same service gets a service loader for that service.

Service providers that are created inside modules cannot control when they are instantiated. However, if the service provider declares a `provider` method, then the service loader will invoke an instance of the service provider via that method. If a `provider` method is not declared in the service provider, there is a direct instantiation of the service provider by the service provider's constructor. In this case the service provider needs to be assignable to the class or interface of the service. A service provider that exists as an automatic module inside an application's module path needs a provider constructor.

Objective:
Java Platform Module System

Sub-Objective:
Declare, use, and expose modules, including the use of services

References:

Oracle Technology Network > Java SE > Java SE Documentation > Java Platform Standard Edition 11 > API Specification > java.base > java.util > java.lang > Class ServiceLoader<S>

Oracle Technology Network > Java SE 9 and JDK 9 > ServiceLoader

---

Question #41 of 50

Question ID: 1327911

Consider the following code:

```
interface TestInterface {
    static void methodA() {
        printThis();
        System.out.println("This is method1");
    }
    static void methodB() {
        printThis();
        System.out.println("This is method2");
    }
    private abstract void printThis() {
```

```
        System.out.println("Method begins");
        System.out.println("Method works!");
    }
    default void testmethods() {
        methodA();
        methodB();
    }
}
public class TestClass implements TestInterface{
    public static void main(String args[]) {
        TestClass tc = new TestClass();
        tc.testmethods();
    }
}
```

What would be the output?

A) `This is method2`

B) `This is method1`

C) None. The code does not compile.

D) `Method works!`

Explanation

The code does not compile and produces no output. You need to use the `private` and `static` access modifiers with the `printThis()` method. This allows you to share common code inside of `static` methods using a non-static `private` method.

A `private` method inside a Java interface allows you to avoid redundant code by creating a *single* implementation of a method inside the interface itself. This was not possible before Java 9. You can create a `private` method inside an interface using the `private` access modifier.

You cannot add the keyword `abstract` because the `abstract` and `private` keywords cannot be used together in a Java interface. This is because `private` and `abstract` both have separate uses in Java. When a method is deemed `private`, it indicates that any subclass cannot inherit it or override it. However, when a method is deemed `abstract`, it needs to be inherited and overridden by subclasses.

The other options are incorrect because no output is displayed due to compilation failure.

Objective:
Java Object-Oriented Approach

Sub-Objective:
Create and use interfaces, identify functional interfaces, and utilize private, static, and default methods

References:

Oracle > Java Documentation > The Java Tutorials > Interfaces and Inheritance > Defining an Interface

Oracle > Java Documentation > The Java Tutorials > Interfaces and Inheritance > Default Methods

Chapter 3. Java Interfaces > 3.1. Create and use methods in interfaces

---

Question #42 of 50

Question ID: 1328036

Given the following:

```
public class Java11_Looping {
   public static void main(String[] args) {
    for (int x = 0; x < 6; x-=2) {
      System.out.println("Outer x: " + x);
      System.out.print("Inner x:");
      while (x++ < 7)
        System.out.print(" " + x);
      System.out.println("\nOuter x: " + x);
    }
   }
}
```

What is the output?

A)
```
Outer x: 0
Inner x: 1 2 3 4 5 6 7
Outer x: 8
```

B)
```
Outer x: 0
Inner x: 1 2 3 4 5 6 7
```

C)
```
Outer x: 0
Inner x: 1 2 3 4 5 6
```

D)
```
Outer x: 0
Inner x: 1 2 3 4 5 6
Outer x: 7
```

Explanation

The output is the following:

```
Outer x: 0
Inner x: 1 2 3 4 5 6 7
Outer x: 8
```

The outer `for` block initializes `x` to 0 and decrements its value by 2 after each iteration. The inner `while` block will execute if `x` is less than 7 and increment `x` by 1 after each iteration. If `x` is 6 or greater, then execution will exit the outer `for` block. The outer block initializes `x` but does not decrement its value in the first iteration. The inner `while` block increments `x` until its value is 7 and then increments it to 8 before exiting. Once the inner block exits, the outer `for` block prints the current value of `x`, decrements its value by 2, and exits because `x` is now 6.

The output will not omit the value 7. The expression increments `x` by 1 after its evaluation in the `while` statement, so that it is incremented once more in the final iteration of the `while` block.

The output will not omit `Outer x: 8`. The expression increments `x` by 1 after its evaluation in the `while` statement, so that it is incremented once more when the inner `while` block exits. Thus, the final `Outer`: output will not be omitted, nor will the final value of `x` be 7.

Objective:
Controlling Program Flow

Sub-Objective:
Create and use loops, if/else, and switch statements

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Language Basics > The for statement

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Language Basics > The while and do-while Statements

---

Question #43 of 50

Question ID: 1327840

Given:

```
public class Pedometer {
   private double stride;
   private double[] measurements;
}
```

Which code fragment is a method that meets good encapsulation principles?

A)
```
public void getStride(double stride) {
      stride = this.stride;
   }
```

B)
```
public void getStride(double stride) {
      this.stride = stride;
   }
```

C)
```
public double[] getMeasurements() {
    return this.measurements;
}
```

D)
```
public double[] getMeasurements() {
    return measurements.clone();
}
```

Explanation

The following code fragment is a method that meets good encapsulation principles:

```
public double[] getMeasurements() {
   return measurements.clone();
}
```

In this code fragment, the accessor method uses the `clone` method to return a copy of the `measurements` field, rather than a reference to the field itself. Accessor methods should return copies of field objects. A copy of a field will prevent other classes from modifying fields directly and will require going through mutator methods.

The two versions of the accessor method `getStride` do not meet good encapsulation principles. Neither method returns the value of the `stride` field as expected, but instead modifies the `stride` field like a mutator method. Also, the statement `stride = this.stride;` overwrites the `stride` parameter, rather than assigning the `stride` parameter to the `stride` field as expected.

The accessor method `getMeasurements` that returns a direct reference to the `measurements` field object does not meet good encapsulation principles. Accessor methods should not return direct references to field objects. Direct access will allow other classes to modify fields directly without going through mutator methods.

Objective:
Java Object-Oriented Approach

Sub-Objective:
Understand variable scopes, apply encapsulation and make objects immutable

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Language Basics > Classes and Objects > Controlling Access to Members of a Class

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Language Basics > Object-Oriented Programming Concepts > What Is an Object?

Question #44 of 50

Question ID: 1327809

Which code fragment is a valid method declaration for an arbitrary number of values?

A)
```
public Result performOperation (Object varargs, OperationType
type) {
   //implementation omitted
}
```

B)
```
public Result performOperation (OperationType type, Object...
args) {
   //implementation omitted
}
```

C)
```
public Result performOperation (Object... args, OperationType
type) {
   /implementation omitted
}
```

D)
```
public Result performOperation (OperationType type, Object
varargs) {
   //implementation omitted
}
```

Explanation

The following code fragment is a valid method declaration for an arbitrary number of values:

```
public Result performOperation (OperationType type, Object... args) {
   //implementation omitted
}
```

Varargs allows an arbitrary number of arguments of the same data type. Varargs are declared using the ellipse (`...`) and must be declared as the last parameter in a method.

The code fragments that specify the name varargs do not indicate an arbitrary number of values. Varargs are declared using the ellipse (`...`), not using a specific name for the parameter.

The code fragments that place varargs as the first parameter of the method are not valid. Varargs must be declared as the last parameter in a method.

Objective:
Java Object-Oriented Approach

Sub-Objective:
Define and use fields and methods, including instance, static and overloaded methods

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Language Basics > Classes and Objects > Passing Information to a Method or a Constructor

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Language Basics > Classes and Objects > Defining Methods

Question #45 of 50

Question ID: 1327803

Given the following class:

```
1.   public class Machine {
2.     static String manufacturer;
3.     public static void main (String[] args) {
4.       //Insert code here
5.     }
6.   }
```

Which three code fragments correctly assign a value to the manufacturer field at line 4?

A) `super.manufacturer = "Oracle";`

B) `manufacturer = "Oracle";`

C) `this.manufacturer = "Oracle";`

D)
```
Machine myMachine = new Machine();
myMachine.manufacturer = "Oracle";
```

E) `Machine.manufacturer = "Oracle";`

Explanation

The following code fragments correctly assign a value to the manufacturer field at line 4:

`manufacturer = "Oracle";`

`Machine.manufacturer = "Oracle";`

```
Machine myMachine = new Machine();
myMachine.manufacturer = "Oracle";
```

All three code fragments access the `manufacturer` field as a `static` member. `static` members do not require class instantiation for access.

The first code fragment accesses the `manufacturer` field directly within the same class because the main method is also a `static` member. The second code fragment is the preferred approach to accessing `static` members outside the class. This code fragment specifies the class name `Machine` and `manufacturer` field using dot notation.

The third code fragment is confusing because it accesses the `static` member as if it were an instance member. Although this is syntactically correct, this approach is not recommended. This code fragment will compile because both `static` and instance members are available to objects.

The code fragment that specifies the keyword `this` does not correctly assign a value to the `manufacturer` field. The `this` keyword references the current instance, which is unavailable in a static context.

The code fragment that specifies the keyword `super` does not correctly assign a value to the `manufacturer` field. The `super` keyword references the parent class associated with the current instance, which is unavailable in a static context.

Objective:
Java Object-Oriented Approach

Sub-Objective:
Define and use fields and methods, including instance, static and overloaded methods

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Classes and Objects > Understanding Instance and Class Members

---

Question #46 of 50

Question ID: 1328001

Consider the following code:

```
public class Vader {
  public void speak() {
    System.out.println("Luke, I am your father.");
  }
}

public class Luke extends Vader {
  @Override
  public void speak(int x) {
    System.out.println("It’s not true!");
  }
}

public class Jedi {
  public static void main(String args[]) {
    Luke skyW = new Luke();
    skyW.speak();
  }
}
```

What would be the result of compiling the code above?

A) `It's not true!`

B) Compilation fails.

C) Runtime error occurs.

D) `Luke, I am your father.`

Explanation

A compiler error would be generated because a method annotated as `@Override` does not override the method in the parent class correctly. An error of the following kind would be displayed:

`Method does not override or implement a method from a supertype.`

To ensure correct compilation, the parameter `int x` needs to be removed from the argument list in the overridden `speak()` method.

The other options are incorrect because a compiler error would occur, and so none of the messages in the other options would be displayed.

Annotations in Java provide metadata for the code and can be used to keep instructions for the compiler. They can also be used to set instructions for tools that process source code. Annotations start with the `@` symbol and attach metadata to parts of the program like variables, classes, methods, and constructors, among others.

Objective:
Annotations

Sub-Objective:
Create, apply, and process annotations

References:

Oracle Technology Network > Java SE Documentation > Annotations

---

Question #47 of 50

Question ID: 1328192

You have successfully created a Java application that is ready to be launched in Japan. Which file name could you use for localization data?

A) `JapaneseBundle`

B) `Bundle.jp`

C) `Bundle_jp.properties`

D) `Bundleproperties`

Explanation

`Bundle_jp.properties` is a validly named properties file.

The options `Bundle.jp`, `bundleproperties`, and `JapaneseBundle` are all incorrect because neither of them have a `.properties` file name extension, which is mandatory for a Java properties file.

Objective:
Localization

Sub-Objective:
Implement Localization using Locale, resource bundles, and Java APIs to parse and format messages, dates, and numbers

References:

Oracle Technology Network > Java SE > Java Tutorials > Internationalization

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Essential Classes > The Platform Environment > Properties

---

Question #48 of 50

Question ID: 1327976

Which statement is true about the try-with-resources block?

A) All exceptions are thrown from the try-with-resources block.

B) Only one resource can be declared in a try-with-resources statement.

C) A separate finally block must close resources declared in a try-with-resources statement.

D) A resource can be declared and assigned only in a try-with-resources statement.

Explanation

In a try-with-resources block, a resource can be declared and assigned only in a try-with-resources statement. If an auto-closeable resource is reassigned within a try-with-resources block, compilation will fail. The following example will fail compilation because the resources `reader` and `writer` are declared outside of the try-with-resources statement:

```
StringReader reader;
StringWriter writer;
try (reader = new StringReader(strInput);
  writer = new StringWriter();
){
  //Perform IO operations
}
```

The following example will fail compilation because the resources reader and writer are assigned in the block after the try-with-resources statement:

```
try (StringReader reader;
  StringWriter writer;
){
  reader = new StringReader(strInput);
  writer = new StringWriter();
  //Perform IO operations
} catch (IOException ex) {
  System.err.print(ex.getMessage());
}
```

Also, resources cannot be reassigned in a block using a try-with-resources statement.

All exceptions are not thrown from the try-with-resources block. Exceptions thrown by a try-with-resources statement are suppressed when an exception is thrown from the try block.

One or more resources can be declared in a try-with-resources statement.

A separate finally block is not required when using a try-with-resources statement. Resources declared in a try-with-resources statement are automatically closed without a finally block. Declared resources must implement the `AutoCloseable` interface or the derived interface `Closeable`.

Objective:
Java File I/O

Sub-Objective:
Read and write console and file data using I/O Streams

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Essential Classes > Exceptions > The try-with-resources Statement

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Essential Classes > Basic I/O > Reading, Writing, and Creating Files

---

Question #49 of 50

Question ID: 1328064

Given the custom exception:

```
class OutsideStringException extends StringIndexOutOfBoundsException {}
class CannotFindPatternInStringException extends OutsideStringException {}
```

Given the method:

```
boolean searchString(String pattern, String str) throws OutsideStringException {
  //implementation omitted
}
```

Which handling action is required when invoking the `searchString` method for the code to compile?

A) Catch `StringIndexOutOfBoundsException`.

B) Specify `CannotFindPatternInStringException`.

C) No handling action is required.

D) Specify `StringIndexOutOfBoundsException`.

E) Catch `CannotFindPatternInStringException`.

Explanation

No handling action is required for the code to compile because `OutsideStringException` is a subclass of `StringIndexOutOfBoundsException`, which is a subclass of `RuntimeException`. Checked exceptions are only `Throwable` and its subclasses, excluding `RuntimeException` and its subclasses.

Caching or specifying `StringIndexOutOfBoundsException` or `CannotFindPatternInStringException` is not required for the code to compile. Runtime exceptions are abnormal conditions internal to the application and often are unrecoverable. The compiler does not check catching and specifying `RuntimeException` and its subclasses.

Objective:
Exception Handling

Sub-Objective:
Create and use custom exceptions

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Essential Classes > Exceptions > The Catch or Specify Requirement

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Essential Classes > Exceptions > Specifying the Exceptions Thrown by a Method

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Essential Classes > Exceptions > Creating Exception Classes

---

Question #50 of 50

Question ID: 1327826

Given:

```
public class OutputSuperClass {
   public OutputSuperClass() {
     System.out.println("Super");
   }
}

public class OutputSubClass extends OutputSuperClass {
   public OutputSubClass () {
     System.out.println("Sub 1");
   }
   public OutputSubClass (int x) {
     System.out.println("Sub 2");
   }
   public OutputSubClass (int x, int y) {
     System.out.println("Sub 3");
   }
   public static void main(String[] args) {
     new OutputSubClass(1);
   }
}
```

What is the result?

A) `Super`
`Sub 1`

B) `Super`
`Sub 2`

C) `Sub 2`

D) `Sub 3`

Explanation

The result is the following output:

```
Super
Sub 2
```

The code in the `main` method invokes the `OutputSubClass` constructor with the single `int` parameter, which first invokes the parameterless `OutputSuperClass` constructor. The parameterless `OutputSuperClass` constructor prints `Super`, then the `OutputSubClass` constructor prints `Sub 2`. A subclass constructor automatically invokes the parameterless constructor of the superclass.

The result will not omit the output `Super`. A subclass constructor automatically invokes the parameterless constructor of the superclass.

The result does not include the output `Sub 1` or `Sub 2`. Neither overloaded constructor is invoked based on the single `int` argument, nor does the second constructor explicitly invoke either constructor.

Objective:

Java Object-Oriented Approach

Sub-Objective:

Initialize objects and their members using instance and static initializer statements and constructors

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Language Basics > Classes and Objects > Providing Constructors for Your Classes