Quick Quiz September 8, 2022

Test ID: 222998348

Question #1 of 50

Question ID: 1327836

Given:

```
public class CardHand {
   public Card[] cards;
   public int handValue;

   public CardHand(Card... cards) {
     this.cards = cards;
     for (Card c: cards) {
       handValue += c.getVal();
     }
   }
   private int getVal() {return handValue;}
}
```

Which two statements are true about encapsulation in the `CardHand` class?

- ✗ **A)** The class should be declared with the modifier `private`.
- ✗ **B)** **If the `handValue` field is invariant, then the `cards` field should be declared with the modifier `private`.**
- ✗ **C)** If the `handValue` field is invariant, then the `cards` and `handValue` fields and constructor should be declared with the modifier `private`.
- ✓ **D)** **The `getVal` method should be declared with the modifier `public`.**
- ✓ **E)** **If the `handValue` field is invariant, then the `cards` and `handValue` fields should be declared with the modifier `private`.**

Explanation

The following two statements are true about encapsulation in the `CardHand` class:

- If the `handValue` field is invariant, then the `cards` and `handValue` fields should be declared with the modifier `private`.
- The `getVal` method should be declared with the modifier `public`.

Invariant fields are those fields whose values are integral to object state and should be constrained within an acceptable range. The invariant field, and all fields on which it depends, must be protected from direct manipulation.

In this scenario, the value for the `handValue` field is dependent on the `cards` field, so both fields should be declared as `private` to prevent direct access from outside the class. Accessors and mutator methods provide indirect access to variants, so that the class is functional. The `getVal` method is an accessor that should be accessible outside of the class. Although the modifier does not need to be `public`, the `private` modifier is too restrictive to be functional.

If the `handValue` field is invariant, then both the `handValue` field and the `cards` field should be declared with the modifier `private`. The invariant field and all fields on which it depends must be protected from direct manipulation. Because the value for the `handValue` field is dependent on the cards field, the `handValue` field should be declared as `private` as well.

If the `handValue` field is invariant, then the constructor should not be declared with the modifier `private`. Declaring the constructor with the `private` modifier is only required to prevent class instantiation, not protect invariant fields.

The class should not be declared with the modifier `private`. This action would not affect class encapsulation but access to the class itself. Also, the modifiers `private` or `protected` are not valid for a class unless nested within another class.

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 #2 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, 0, 10, 1, 20]`

✗ **C)** `Weights are [0, 10, 15, 20, 25]`

✗ **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 #3 of 50

Question ID: 1328150

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

✗ **A)** `call`

✗ **B)** `run`

✗ **C)** `start`

✓ **D)** `submit`

✓ **E)** `execute`

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 #4 of 50

Question ID: 1328012

Which operator or method should determine whether two `String` variables have the same value?

✓ **A)** `equals`

✗ **B)** `contentEquals`

✗ **C)** `===`

✗ **D)** `==`

Explanation

The `equals` method determines whether two `String` variables have the same value. The `equals` method is overridden to determine equality between `String` objects based on their character sequence. The `equals` method checks the values *inside* the `String` variables rather than their object references.

The == operator does not determine whether two `String` variables have the same value. The == operator determines whether two `String` variables reference the same object.

The === operator does not determine whether two `String` variables have the same value. This operator is available in JavaScript, but not provided by Java.

The `contentEquals` method does not determine whether two `String` variables have the same value. This method determines whether a `String` variable and `StringBuilder` variable contain the same value.

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 > Numbers and Strings > Strings

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Numbers and Strings > Comparing Strings and Portions of Strings

---

Question #5 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)**
```
JavaIICandidate c = new JavaICandidate();
System.out.println(c.title);
```

✗ **B)**
```
JavaIICandidate c = new JavaICandidate();
System.out.println(c.getStudyHours());
```

✗ **C)**
```
Candidate c = new JavaIICandidate();
System.out.println(c.title);
```

✓ **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 #6 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 {
   public String getCitizenRequirement() { return "General
Citizen"; }
}
```

✓ **B)**
```
interface Electable {
   String getCitizenRequirement();
}
```

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

✗ **D)**
```
abstract class 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 #7 of 50

Question ID: 1328090

Which statement(s) are true of the Stream API? (Choose all that apply.)

- ✗ **A)** Stream processing always makes the best use of all available CPU cores.
- ✗ **B)** Stream processing must perform all their operations on all the elements in the stream.
- ✗ **C)** Streams can be created only from data in classes in the core API, such as `ArrayList` and other implementations of `Collection`.
- ✓ **D) Streams may contain an unbounded number of elements.**
- ✓ **E) Streams support internal iteration, avoiding the need to explicitly code loops.**

Explanation

Streams support internal iteration, avoiding the need for coding loops explicitly. Streams may contain an unbounded number of elements. Streams provide a processing model that is sometimes described as a pipeline, comparable to a production line in a factory. A source of items has raw material available, such as a `List<Student>`, and at each step in the processing, items may be transformed or dropped. At the end of the production line, items are processed and/or batched up into a single unit (package) for shipping.

Because items are processed as they progress down the production line, they effectively iterate, but without the programmer having to code loops directly. This process is commonly called internal iteration.

Streams can contain an unbounded number of elements. This situation is exemplified by the `Stream.generate(Supplier s)` method, which creates an unbounded stream of elements by repeatedly calling the get method of the supplier. Of course, collection of a stream does not complete until the stream does. Therefore, it is important that something should limit the stream prior to collection.

Streams do not need to perform all their operations on all of the elements. As noted, not all items have to be passed down the line; some can be dropped instead. This is often performed by the `Stream.filter` operation. In addition, a stream's source can use code to create each item, rather than having all the items stored somewhere. In this situation, it is possible to have an infinite number of elements in the stream. However, the stream processing "pulls" items from the downstream end of the pipeline, rather than pushing them, making it possible to process only some of the items in the stream.

Streams can be created from many types of data outside of the core API. Streams can be created in a variety of ways, such as by using the factory methods `generate()` or `iterate()` in the `Stream` class, or other methods in the `StreamSupport` class.

Streams do not always make the best use of all available CPU cores. While one of the key purposes of the Stream API is to support concurrent processing in a way that allows considerable automatic optimization, the parallel operation mode is not the default. Even when it is selected, it is still possible to create code that operates sub-optimally on a multi-core hardware platform.

Objective:
Working with Streams and Lambda expressions

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

References:

The Java Tutorials > Collections > Lesson: Aggregate Operations

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

The Java Tutorials > Collections > Lesson: Aggregate Operations > LazinessThe Java Tutorials > Collections > Lesson: Aggregate Operations > Executing Streams in Parallel

---

Question #8 of 50

Question ID: 1327990

You need to create a Java application that reads in data from a meteorological report and extract each line of the report and save it as an array. You write the following code:

```
01  Path filePath = Paths.get(nameOfFile);
02  List<String> liner = new ArrayList<>();
```

```
03  Charset encoding = StandardCharsets.UTF_8;
04  Object get = new Object(filePath, encoding);
05  while (get.hasNextLine())
06   liner.add(get.nextLine());
07  get.close();
08  return liner;
```

However, this code does not work as expected. Which of the following changes will you need to make to make this code compile and run correctly?

✗ **A)** Replace `Object` with `File` at line `04`

✗ **B)** Replace `Path` with `File` at line `01`

✓ **C) Replace `Object` with `Scanner` at line `04`**

✗ **D)** Replace `List` with `Collection` at line `02`

Explanation

You should use the `Scanner` class to replace `Object`:

```
Path filePath = Paths.get(nameOfFile);
List<String> liner = new ArrayList<>();
Scanner get = new Scanner(filePath, encoding);
while (get.hasNextLine())
  liner.add(get.nextLine());
get.close();
return liner;
```

You need to use the `Scanner` class to return each line of a file into a collection. The other classes will not be able to allow this operation.

You should not replace `Object` with `File` at line `04` because you need a class that can *read* and extract data from a path. The `File` class will not do this. The `File` class only allows access to a `File` and its metadata.

You should not replace `Path` with `File` at line `01` because you need the `Path` class to store the full filepath to the specific file.

You should not replace `List` with `Collection` at line `02` because you need to use the `List` class to store the `String` data that will be extracted from the file that is being accessed.

Objective:
Java File I/O

Sub-Objective:
Handle file system objects using java.nio.file API

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Essential Classes > Basic I/O > File I/O (Featuring NIO.2)

Question #9 of 50

Question ID: 1328055

Which two method declarations will compile if their implementation code throws an `UnsupportedOperationException`?

- ✓ **A)** `void trySomething (OperationType op) throws UnsupportedOperationException`
- ✗ **B)** `void trySomething (OperationType op) throw ReadOnlyBufferException`
- ✗ **C)** `void trySomething (OperationType op) throw BadStringOperationException`
- ✓ **D)** `void trySomething (OperationType op)`

Explanation

The following two method declarations will compile if their implementation code throws an `UnsupportedOperationException`:

```
void trySomething (OperationType op)
```

```
void trySomething (OperationType op) throws UnsupportedOperationException
```

Because `UnsupportedOperationException` is a subclass of `RuntimeException`, handling this exception is not required. The compiler does not check `RuntimeException` and its subclasses.

The method declarations that use the `throw` keyword will not compile because the `throws` keyword must be specified in the method declaration to specify exceptions. The `throw` keyword is used within a method to explicitly throw an exception. Also, the exceptions `ReadOnlyBufferException` and `BadStringOperationException` are not superclasses of `UnsupportedOperationException`.

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 > Specifying the Exceptions Thrown by a Method

Question #10 of 50

Question ID: 1327776

Given:

```
String str1 = "salt";

String str2 = "sAlT";
```

Which two code fragments will output `str1 and str2 are equal`?

✓ **A)**
```
if (str1.equalsIgnoreCase(str2) )
        System.out.println("str1 and str2 are equal");
```

✗ **B)**
```
if (str1.equals(str2) )
        System.out.println("str1 and str2 are equal");
```

✗ **C)**
```
if (str1 == str2.toLowerCase() )
        System.out.println("str1 and str2 are equal");
```

✓ **D)**
```
if (str1.equals(str2.toLowerCase()) )
        System.out.println("str1 and str2 are equal");
```

✗ **E)**
```
if (str1 == str2 )
        System.out.println("str1 and str2 are equal");
```

Explanation

The following two code fragments will output `str1 and str2 are equal`:

```
if (str1.equals(str2.toLowerCase()) )
   System.out.println("str1 and str2 are equal");

if (str1.equalsIgnoreCase(str2) )
   System.out.println("str1 and str2 are equal");
```

The `equals` method is overridden to determine equality between `String` objects based on their character sequence. The `equals` method is a case-sensitive comparison. Because `str1` and `str2` differ by letter-case, you need to either retrieve a lower-case version of `str2` using the `toLowerCase` method or use the `equalsIgnoreCase` method rather than the `equals` method.

The code fragments that use the == operator will not output `str1 and str2 are equal` because this operator compares object references, not values. Unlike primitives, object comparison using the == operator determines whether the same object is being referenced. The `equals` method determines whether value(s) in different objects are equivalent.

The code fragment that uses the `equals` method without the `toLowerCase` method will not output `str1` and `str2` are `equal` because the `equals` method is a case-sensitive comparison. Because `str1` and `str2` differ by letter-case, you need to either retrieve a lower-case version of `str2` using the `toLowerCase` method or use the `equalsIgnoreCase` method rather than the `equals` method.

Objective:
Working with Java Data Types

Sub-Objective:
Handle text using String and StringBuilder classes

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Numbers and Strings > Comparing Strings and Portions of Strings

Java API Documentation > Java SE 11 & JDK 11 > Class String

Question #11 of 50

Question ID: 1328155

Which two interfaces execute both `Callable` and `Runnable` objects as worker threads?

- ✓ **A)** **`ExecutorService`**
- ✗ **B)** `Executor`
- ✓ **C)** **`ScheduledExecutorService`**
- ✗ **D)** `Future`

Explanation

The `ExecutorService` and `ScheduledExecutorService` interface can execute both `Callable` and `Runnable` objects. This is because the `ExecutorService` interface provides the `submit` method that accepts both types of objects. The `ScheduledExecutorService` interface is a subinterface of the `ExecutorService` interface. The `ScheduledExecutorService` interface supports execution of tasks after a fixed delay or a periodic execution.

The `Executor` interface can only execute `Runnable` objects as worker threads because its `execute` method can only accept `Runnable` objects.

The `Future` interface cannot execute `Callable` or `Runnable` objects as worker threads. The `Future` interface represents the result of a worker thread and can poll or cancel a task.

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 ExecutorService

---

Question #12 of 50

Question ID: 1327807

Given the following code fragment:

```
public class StandardMethods {
   public static void printPerimeter(double... sides) {
     double result = 0;
     for (double side: sides) {
       result += side;
     }
     System.out.println("Perimeter is " + result);
   }
}
```

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

✗ **A)** `double perimeter = StandardMethods.printPerimeter();`

✓ **B)** **`StandardMethods.printPerimeter();`**

✗ **C)** `double perimeter = StandardMethods.printPerimeter(7.5, 9.8, 11);`

✓ **D)** **`StandardMethods.printPerimeter(7.5, 9.8, 11);`**

Explanation

The following code lines will compile:

```
StandardMethods.printPerimeter();
StandardMethods.printPerimeter(7.5, 9.8, 11);
```

Both code lines invoke the `printPerimeter` method without expecting a return value because the method declares `void`. Because varargs (`...`) is used for the `printPerimeter` method, an arbitrary number of arguments can be specified during invocation. The first code line invokes `printPerimeter` with no arguments, while the second code line invokes `printPerimeter` with three arguments.

The code lines that attempt to retrieve a return value will not compile. The `getSurfaceArea` method declares `void`, so no return value is expected. The compiler will indicate that the variable `perimeter` of the `double` type is not compatible with the `void` type.

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 #13 of 50

Question ID: 1327889

Given:

```
public interface Shape {
   public long getArea();
   public int getPerimeter();
}

public class Rectangle implements Shape {
   //implementation omitted
   public int getWidthLength() {return width;}
   public int getHeightLength() {return height;}
   public long getArea() {return width * height;}
   public int getPerimeter() {return 2 * (width + height);}
   public double getAngle() {return angle1;}
}

public class Square extends Rectangle {
   //implementation omitted
   public int getSideLength() {return side;}
}

public class Rhombus extends Square {
   //implementation omitted
   public double getAngle1() {return angle1;}
   public double getAngle2() {return angle2;}
   public static void main(String[] main) {
     Shape sh = new Rhombus(5,65, 115);
     Square sq = (Square) sh;
```

```
    }
}
```

Which methods are available when using the `sq` variable? (Choose all that apply.)

- ✗ **A)** `getAngle2`
- ✓ **B)** **`getHeightLength`**
- ✗ **C)** `getAngle1`
- ✓ **D)** **`getSideLength`**
- ✓ **E)** **`getWidthLength`**
- ✓ **F)** **`getAngle`**
- ✓ **G)** **`getArea`**
- ✓ **H)** **`getPerimeter`**

Explanation

The `getArea`, `getAngle`, `getPerimeter`, `getSideLength`, `getWidthLength`, and `getHeightLength` methods are available using the `sq` variable. Because the reference type is the `Square` class, only the methods declared in or inherited by `Square` are available to the `sq` variable. Although the object type is `Rhombus`, the reference type determines member availability.

The `getAngle1` and `getAngle2` methods are not available using the `sq` variable. These methods would be available if the reference type was `Rhombus` because these methods are declared in the `Rhombus` class.

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

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

---

Question #14 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 properly encapsulated, but the access modifier on the `compartments` field should be changed to `public`.

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

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

✓ **D) This class is poorly encapsulated. The `totalItems` field should be calculated in the constructor and in any methods that modify the `compartments` 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 #15 of 50

Question ID: 1327922

Given the following contents of a source file:

```
class ThreadRunner <T extends Runnable> {
  private T runnable;
  public ThreadRunner(Class<T> c) {
    try {
      this.runnable = __________;
    } catch(Exception ex) {}
  }
  public void start() {
    Thread th = new Thread(this.runnable);
    th.start();
  }
}
class RunnableObject implements Runnable {
  public void run() {System.out.println("Running!");}
  public static void main(String[] args) {
    new ThreadRunner<>(RunnableObject.class).start();
  }
}
```

Which code segment should be inserted to compile the source file?

✗ **A)** `new Runnable<T>()`

✗ **B)** `new Class<T>()`

✓ **C)** `c.newInstance()`

✗ **D)** `new T()`

✗ **E)** **`new c()`**

Explanation

The code segment `c.newInstance()` should be inserted to compile the source file. Generic parameters cannot be directly instantiated because the compiler must replace all parameters with existing types that match either the required bounds or `Object`, if the type parameter is unbounded. This is known as *type erasure*. To overcome this limitation, a `Class` argument must be accepted and the `newInstance` method invoked to delegate instantiation.

This approach does require exception handling at runtime, because the generic type may not have an accessible parameterless constructor, or it may represent a type that does not support instantiation.

The code segment `new T()` should not be inserted to compile the source file. This code segment will cause a compilation error because generic parameters cannot be directly instantiated.

The code segment `new c()` should not be inserted to compile the source file. This code segment will cause a compilation error because a type named `c` is neither declared in the source file nor available through the Java API.

The code segment `new Class<T>()` should not be inserted to compile the source file. This code segment will cause a compilation error because the `Class` constructor is not accessible.

The code segment `new Runnable<T>()` should not be inserted to compile the source file. This code segment will cause a compilation error because the `Runnable` interface does not accept a generic parameter, and interfaces cannot be instantiated.

Objective:
Working with Arrays and Collections

Sub-Objective:
Use generics, including wildcards

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Generics (Updated) > Type Erasure

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Generics (Updated) > Generic Types

Oracle Technology Network > Java SE > Java Language Specification > Classes > Class Declarations > 8.1.2. Generic Classes and Type Parameters

---

Question #16 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)** `thread`
- ✗ **B)** `collect()`
- ✓ **C)** `filter()`
- ✗ **D)** `reduce`
- ✓ **E)** `forEach()`
- ✗ **F)** `new`

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 #17 of 50

Question ID: 1328019

Given the following:

```
switch(cardVal) {
   case 4: case 5: case 6:
   case 7: case 8:
     System.out.println("Hit");
     break;
   case 9: case 10: case 11:
     System.out.println("Double");
     break;
   case 15: case 16:
     System.out.println("Surrender");
```

```
    break;
  default:
    System.out.println("Stand");
}
```

Which two values for the variable `cardVal` will output `Stand`?

- ✗ **A)** 6
- ✗ **B)** 10
- ✓ **C) 14**
- ✓ **D) 18**
- ✗ **E)** 16

Explanation

The values 14 and 18 for the variable `cardVal` will output `Stand`. Any value for `cardVal` not specified in case labels will reach the default label and print the output `Stand`. Only values in the ranges 4-11 and 15-16 have associated case labels.

The value 6 will not output `Stand`. The output will be `Hit` because its value matches the first set of case labels in the range 4-8.

The value 10 will not output `Stand`. The output will be `Double` because its value matches the second set of case labels in the range 9-11.

The value 16 will not output `Stand`. The output will be `Surrender` because its value matches the third set of case labels in the range 15-16.

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 switch Statement

---

Question #18 of 50

Question ID: 1327792

Given the following:

```
public class MyBasicClass {
  //Insert code here
```

```
}
```

Which three lines of code can be included in the class?

- ✓ **A)** `static final int VAL=1000;`
- ✓ **B)** `void BasicMethod() {}`
- ✗ **C)** `package basicPackage;`
- ✓ **D)** `enum ClassType {basic, advanced}`
- ✗ **E)** `import java.text.*;`
- ✗ **F)** `module basicModule {}`

Explanation

The three lines of code can be included in the class as follows:

```
public class MyBasicClass {
  enum ClassType {basic, advanced}
  void BasicMethod() {}
  static final int VAL=1000;
}
```

A class body can include static and non-static fields, methods, constructors, and nested enumerations and classes.

The code line `package basicPackage;` cannot be included in the class because a `package` statement must be the first executable line in the source file, not inside a class body.

The code line `import java.text.*;` cannot be included in the class because `import` statements are not allowed in class bodies.

The code line `module basicModule {}` cannot be included in the class source file, but must be included in a separate `module-info.java` file.

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:

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

Oracle.com > Java 9 | Excerpt > Understanding Java 9 Modules

Question #19 of 50

Question ID: 1328164

Given:

```
public class MultithreadClass {
  public static synchronized void concurrentMethod(byte[] data) {
    //implementation omitted
  }
}
```

Which method provides the same synchronization as `concurrentMethod`?

✗ **A)**
```
public void anotherMethod1(byte[] data) {
  synchronized(data) {
    //implementation omitted
  }
}
```

✗ **B)**
```
public void anotherMethod3(byte[] data) {
  synchronized(this) {
    //implementation omitted
  }
}
```

✗ **C)**
```
public void anotherMethod2(byte[] data) {
  synchronized(void) {
    //implementation omitted
  }
}
```

✓ **D)**
```
public void anotherMethod4(byte[] data) {
  synchronized(MultithreadClass.class) {
    //implementation omitted
  }
}
```

Explanation

The following method provides the same synchronization as `concurrentMethod`:

```
public void anotherMethod4(byte[] data) {
  synchronized(MultithreadClass.class) {
    //implementation omitted
  }
}
```

Unlike a synchronized method, the object with the intrinsic lock must be specified explicitly in synchronized blocks. Static methods use the `Class` object associated with the current class.

`anotherMethod1` and `anotherMethod2` do not provide the same synchronization as `concurrentMethod`. Rather than using the current object or class, these methods use a parameter or return value for the intrinsic lock. Although the parameter is valid, the result will not provide effective synchronization. The return type `void` is not valid for the synchronized statement and the code will fail compilation.

`anotherMethod3` does not provide the same synchronization as `concurrentMethod`. Instance methods, not static methods, use the current object associated with a class with the `this` keyword.

Objective:
Concurrency

Sub-Objective:
Develop thread-safe code, using different locking mechanisms and java.util.concurrent API

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Essential Classes > Concurrency > Intrinsic Locks and Synchronization

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Essential Classes > Concurrency > Atomic Variables

---

Question #20 of 50

Question ID: 1327996

Given the code fragment:

```
01  try {
02    String dbURL = "jdbc:derby://localhost:1527/sample;user=test;password=p@$$w0rd";
03    String query = "SELECT * FROM CUSTOMER WHERE EMAIL='www.centralcomp.com'";
04    Connection cn = DriverManager.getConnection(dbURL);
05    Statement stmt = cn.createStatement();
06    ResultSet rs = stmt.executeQuery(query);
07    //Insert code here
08    System.out.println("Customer ID: " + rs.getInt("CUSTOMER_ID"));
09  } catch (Exception ex) {
10    throw ex;
11  }
```

The result should be the `CUSTOMER_ID` value of the first row. Which code fragment, when inserted at line `07`, enables the code to execute without throwing an exception?

✗ **A)** `rs.getRow();`

✓ **B)** **`rs.next();`**

✗ **C)** `rs.refreshRow();`

✗ **D)** `rs.last();`

✗ **E)** `rs.first();`

✗ **F)** `rs.previous();`

Explanation

The `rs.next();` code fragment should be inserted at line `07` to enable the code to execute without throwing an exception. By default, the cursor of a result set points before the first row. The `next` method moves the cursor to the first and any subsequent rows.

The `first`, `last` and `previous` methods will not enable the code to execute without throwing an exception. A `SQLException` is thrown because only scrollable `ResultSet` objects support arbitrary backward and forward movements.

The `getRow` method will not enable the code to execute without throwing an exception, because this method does not affect the cursor position. The `getRow` method retrieves the current row number.

The `refreshRow` method will not enable the code to execute without throwing an exception, because this method does not affect the cursor position. The `refreshRow` method retrieves the most recent values at the current cursor.

Objective:
Database Applications with JDBC

Sub-Objective:
Connect to and perform database SQL operations, process query results using JDBC API

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > JDBC(TM) Database Access > JDBC Basics > Retrieving and Modifying Values from Result Sets

Oracle Technology Network > Java SE > Java SE Documentation > Java Platform Standard Edition 11 API Specification > java.sql > Interface ResultSet

---

Question #21 of 50

Question ID: 1328111

Given the `Student` class,

```
public class Student {
  private String name;
  private List<String> courses;
```

```
  public Student(String name, String ... courses) {
    this.name = name;
    this.courses = Arrays.asList(courses);
  }
  public String getName() { return name; }
  public List<String> getCourses() { return courses; }
}
```

And the code fragment:

```
Stream.of(
  new Student("Fred", "Math", "Physics", "Economics"),
  new Student("Sheila", "Astronomy", "Physics", "Math")
)
  // line n1
.distinct()
.forEach(System.out::println);
```

Your goal is to create a single list of course names, with duplicates removed for students in a class. Which code should you insert at line `n1` to output the course name list as required?

✗ **A)** `.map(s->s.getCourses().stream())`

✗ **B)** `.for(String c : s.getCourses())`

✗ **C)** `.flatMap(s->s.getCourses())`

✗ **D)** `.map(s->s.getCourses())`

✓ **E)** `.flatMap(s->s.getCourses().stream())`

Explanation

You should insert `.flatMap(s->s.getCourses().stream())` to correctly convert each individual student into a stream of strings representing course names. The `Stream` method `flatMap` takes an expression that is a `Function<T, Stream<R>>` where `T` is the element type of the input stream and `R` is the element type of the output stream. Notice that the lambda must create a stream of the output type, not any other kind of grouping.

The code fragment `.flatMap(s->s.getCourses())` will not compile because the return type of the lambda is not a `Stream`. The option is therefore incorrect.

The syntax of `for(String c : s.getCourses())` in isolation might be a potential candidate for iterating the courses of a particular student `c`. However, because `for` is a keyword, it is not a valid method name. This code would not compile.

Both options that use `map` are incorrect because the number of items in the stream must change to satisfy the scenario requirements. The `Stream` method `map` always creates a one-to-one relationship between the number of items in the stream that it processes and the stream that it produces. In this scenario, the input will be a stream of two student objects, but the output will be a stream of six string objects.

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 #22 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)** `Sub 3`

✗ **B)** `Sub 2`

✗ **C)**
```
Super
    Sub 1
```

✓ **D)**
```
Super
Sub 2
```

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

---

Question #23 of 50

Question ID: 1328009

Consider the following code:

```
package lordsource;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Documented;

@Documented
@Retention(RetentionPolicy.RUNTIME)
@interface AnnoLord {
 String Maker() default "Lord";
```

```
 String LastDate();
}

public class Driver {
 // Insert Annotation Here
 void method1() {
  System.out.println("This is method number 1");
 }
 public static void main(String args[]) {
  System.out.println("Welcome to the machine");
 }
}
```

You now need to insert the custom annotation indicated the programmer and end date of the code. Which of these code fragments will you insert?

- ✗ **A)** `@Target({ElementType.METHOD})`
- ✗ **B)** `@AnnoLord(RetentionPolicy.SOURCE)`
- ✓ **C)** `@AnnoLord(Developer="Lord", LastDate="31-12-2019")`
- ✗ **D)** `@AnnoLord("Lord", "31-12-2019")`

Explanation

You should use the following code fragment:

```
@AnnoLord(Developer="Lord", LastDate="31-12-2019")
void method1() {
 System.out.println("This is method number 1");
}
```

The following code option is syntactically incorrect:

`@AnnoLord("Lord", "31-12-2019")`

The following code option is incorrect because it should be used with the `@Retention` annotation type and not the custom annotation `@AnnoLord`:

`@AnnoLord(RetentionPolicy.SOURCE)`

The following code option is incorrect because the `@Target` annotation does not create custom annotation, which is what is needed in this scenario.

You can have custom annotations for annotating elements of your program like methods, variables and constructors. These annotations are applied before the program element is declared. The syntax for user-defined annotations is as given below:

```
[Access-Specifier] @interface<NameofAnnotation> {
  Data-Type <Name of Method>() [default-value];
}
```

The `NameofAnnotation` specifies what your annotation is called. The `default-value` field is optional. The specified `Data-Type` of the method needs to be either enum, primitive, String, class or an array.

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 #24 of 50

Question ID: 1327941

Consider the following code:

```
import java.util.Set;
public class MusicalInstruments {
    public static void main(String[] args) {
        Set<String> instrset = new HashSet<>(Arrays.asList
("Guitar","Drums","Piano","Saxophone","Bass"));
        instrset.add(String “Bagpipes”);
        for(String l:instrset) {
            System.out.println(l);
        }
    }
}
```

What is the output?

✓ **A)**
```
Guitar
Drums
Piano
Saxophone
Bass
Bagpipes
```

✗ **B)** `UnsupportedOperationException`

✗ **C)** The code fails compilation.

✗ **D)** `ArithmeticException`

Explanation

The following output is displayed:

```
Guitar
Drums
Piano
Saxophone
Bass
Bagpipes
```

`UnsupportedOperationException` is an incorrect option. This exception is displayed when there is an attempt to modify an immutable list, map, or set. In this code example, if the set was created using the `Set.of()` factory method, the method `instrset.add(String "Turntable");` attempts to add an element to the immutable set resulting in an exception being thrown. This exception is *not* thrown because a mutable set was created using the `new HashSet` operation.

Java 9 and above 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.

You can use the `Set.of()` static factory method for creating an immutable set with no null elements. If the elements in the set are serializable then the set becomes serializable as well. It also has no duplicate elements.

`ArithmeticException` is an incorrect option. This exception is thrown when an illegal arithmetic operation is performed, for instance division by zero. None of the operations in the code fragment perform an illegal arithmetic operation.

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 #25 of 50

Question ID: 1327948

Given the following:

```
int[][][] matrix = new int[5][5][5];
```

Which line of code sets the first element of the matrix array?

✗ **A)** `matrix[1,1,1] = 42;`

✗ **B)** `matrix[[0][0][0]] = 42;`

✗ **C)** `matrix[1][1][1] = 42;`

✓ **D)** `matrix[0][0][0] = 42;`

✗ **E)** `matrix[[1][1][1]] = 42;`

✗ **F)** `matrix[0,0,0] = 42;`

Explanation

The following line of code sets the first element of the matrix array:

```
matrix[0][0][0] = 42;
```

When accessing an element in a multidimensional array, each dimension index should be specified in separate square bracket pairs. Indexes in arrays are zero-based, so that the first element is always located at position 0. The first element of a multidimensional array is accessed by specifying 0 for each of its dimension indices.

The lines of code that specify position 1 for each dimension index will not set the first element. Indexes in arrays are zero-based, so that the first element is always located at position 0. Specifying 1 for each dimension index will access the second element within the second inner array of the second outer array.

The lines of code that use commas and additional square brackets will not set the first element. These lines of code contain syntax errors and will fail to compile.

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 > Learning the Java Language > Language Basics > Arrays

---

Question #26 of 50

Question ID: 1328079

Which statement(s) are true of the `java.util.function.Function` interface? (Choose all that apply.)

✗ **A)** Its single method may throw checked exceptions.

✗ **B)** Its single method is named `accept`.

✓ **C)** Its single method takes a reference type parameter.

✗ **D)** It contains no methods.

✓ **E)** Its single method is named `apply`.

Explanation

The `java.util.function.Function` interface has a single method named `accept` that takes a reference type parameter. The `Function` interface is:

```
public interface Function<T,R> {
  R apply(T t);
}
```

The `apply` method takes a single parameter, which is generic and therefore a reference type.

Its single method does not declare any exceptions, so it cannot throw any checked exceptions.

It does contain a method, and that method is named `apply`, not `accept`.

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 > Function

---

Question #27 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: 1,1,10`

✗ **B)** `x,y,z: 10,10,10`

✗ **C)** `x,y,z: 1,1,1`

✗ **D)** `x,y,z: 1,10,10`

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 #28 of 50

Question ID: 1328065

Which lambda expressions correctly implement `BiFunction<String, Number, String>`? (Choose all that apply.)

✗ **A)** `s, n -> String.format(s, n)`

✗ **B)** `(s, n) -> n`

✗ **C)** `(final s, n) -> String.format(s, n)`

✓ **D)** `(final String s, Number n) -> String.format(s, n)`

✗ **E)** `(String s, n) -> String.format(s, n)`

✓ **F)** `(s, n) -> s + n`

Explanation

The lambda expressions `(final String s, Number n) -> String.format(s, n)` and `(s, n) -> s + n` correctly implement `BiFunction<String, Number, String>`.

The `BiFunction` interface defines a method, called `apply`, that takes two direct arguments. The first two type arguments define the two arguments to the method, which in this case requires that that arguments be `String` and `Number`, in that order. A third type argument defines the return type of the method. Therefore, the lambda used to satisfy this invocation must take `String` and a `Number` as arguments and return a `String`. Both correct answers define syntactically correct lambda expressions that are compatible with these types.

The option `(s, n) -> n` returns a `Number`, not a `String`.

`s, n -> String.format(s, n)` is not a correct lambda expression implementation. If a single argument lambda is being created, you can omit the parentheses around that argument if the type is inferred and no modifiers are used. However, this omission is never permitted when multiple arguments are being passed.

`(String s, n) -> String.format(s, n)` is not a correct lambda expression implementation. Lambda expression argument types can often be omitted and inferred from the surrounding code. However, if a lambda includes any argument types, then all the argument types must be specified.

`(final s, n) -> String.format(s, n)` is not a correct lambda expression implementation. While lambda expression arguments may carry modifiers, such as `final`, this may only be done when the types are specified.

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

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

Question #29 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)** `@Deprecated`
- ✗ **C)** `@Override`
- ✓ **D)** `@Documented`

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 #30 of 50

Question ID: 1327906

You define the following interface to handle photographic items:

```
public interface Photographer {
  Photograph makePhotograph(Scene s);
  /* insert here */ {
    List<Photograph> result = new ArrayList<>();
    for (Scene s : scenes) {
      result.add(makePhotograph(s));
    }
    return result;
  }
}
```

You need to define the `makePhotographs` method to support making multiple photographs with any `Photographer` object. This behavior should be modifiable by specific implementations.

Which code should be inserted at the point marked `/* insert here */` to declare the `makePhotographs` method?

✗ **A)** `static List<Photograph> makePhotographs(Scene ... scenes)`

✓ **B)** `default List<Photograph> makePhotographs(Scene ... scenes)`

✗ **C)** `List<Photograph> makePhotographs(Scene ... scenes)`

✗ **D)** `public List<Photograph> makePhotographs(Scene ... scenes)`

✗ **E)** `List<Photograph> makePhotographs(Scene ... scenes);`

Explanation

You should declare the `makePhotographs` method as follows:

`default List<Photograph> makePhotographs(Scene ... scenes)`

A default method creates an instance method with an implementation. The implementation can be overridden in specializing types, which meets the scenario requirements. Interfaces can define method implementations only in two conditions: either the method is static, or the method is default. Otherwise, abstract methods may be declared, but implementation must be deferred to a specialized type.

You should not use the declaration `List<Photograph> makePhotographs(Scene ... scenes)`. Any method declared in an interface that is neither static nor default will be treated as an abstract method and must not have a body. This method declaration will cause the code to fail compilation because it attempts to declare an abstract-by-default method, and yet tries to provide a method body.

`List<Photograph> makePhotographs(Scene ... scenes);` will cause a compilation failure because the method body block immediately follows the semicolon (`;`) that terminates what is essentially a valid abstract method declaration.

`public List<Photograph> makePhotographs(Scene ... scenes)` is incorrect because it will be treated as an abstract method, and cannot have a body. Interface methods are public whether or not they are declared as such. This method declaration is effectively identical to `List<Photograph> makePhotographs(Scene ... scenes)`.

A static method declaration would not compile because in a static, there is no current context reference this. Without that context, the call to `makePhotographs` inside the statement `result.add(makePhotograph(s))`; cannot be invoked. This method declaration would cause the code to fail compilation.

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 #31 of 50

Question ID: 1328165

In a multi-threaded application, a single `Queue` object must be read by multiple threads and modified by another thread. Multiple threads should be able to read concurrently or allow a write operation by a single thread. Which strategy should you use?

- ✗ **A)** Replace the `Queue` object with a `BlockingQueue` object.
- ✗ **B)** Use the `volatile` keyword when declaring the Queue variable.
- ✓ **C)** Implement a custom lock with the `ReadWriteLock` interface.
- ✗ **D)** Use a synchronized block for read and write operations.

Explanation

To meet the scenario requirements, you should implement a custom lock with the `ReadWriteLock` interface. Using the `ReadWriteLock` interface, there are two associated locks, one for read operations and another for write operations. The read lock is not exclusive and can be held by multiple threads that perform read operations, if no write operations are performed. The write lock is exclusive, so that only a single thread can perform write operations.

You should not use a synchronized block for read and write operations because this strategy will limit read and write operations to a single thread at a time.

You should not use the `volatile` keyword when declaring the `Queue` variable because this strategy will only affect the `Queue` object itself, not read/write operations on its contents. The `volatile` keyword ensures that a shared variable value is visible to multiple threads. You should use the `volatile` keyword on a shared variable when write operations do not depend on its value and locking is not required during access.

You should not replace the `Queue` object with a `BlockingQueue` object because this strategy will limit all read and write operations. The `BlockingQueue` prevents memory consistency errors, so that write operations are visible to all threads when they occur. This is equivalent to using a single mutex lock, not a lock pair.

Objective:

Concurrency

Sub-Objective:

Develop thread-safe code, using different locking mechanisms and java.util.concurrent API

References:

Oracle Technology Network > Java SE > Java SE Documentation > Java Platform Standard Edition 11 API Specification > java.util.concurrent.locks > Interface ReadWriteLock

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Essential Classes > Concurrency > Lock Objects

---

Question #32 of 50

Question ID: 1328183

You have written a class to store employee records including bank account numbers.

```
public abstract class Employee {
  protected Employee() {
    //Insert code here
  }
  private Employee(Void ignored) {
    // Implementation omitted
  }
  private static Void securityCheck() {
    SecurityManager secured = System.getSecurityManager();
    if (secured != null) {
      secured.checkCreateClassLoader();
    }
    return null;
  }
}
```

What line of code will you insert to disallow malicious subclassing of this class?

✗ **A)** `PreparedStatement ps = con.prepareStatement(sql);`

✗ **B)** `Runtime run = Runtime.getRuntime();`

✗ **C)** `System.out.println("Check this!");`

✓ **D)** `this(securityManagerCheck());`

Explanation

The following line of code needs to be inserted:

```
this(securityManagerCheck());
```

The other options are incorrect because they do not invoke the `SecurityManager` check, which is required for securing constructors.

Secure construction of confidential classes by ensuring a check by `SecurityManager` each time a class can be instantiated. This check should be done at the beginning of each `public` or `protected` constructor in the class. Checks must also be enforced before any `public static` factory methods and `readObject` or `readObjectNoData` methods of serializable classes.

To secure sensitive objects during construction, you need to hide constructors by using `static` factory methods instead of using `public` constructors. Also, inheritance should be implemented using delegation instead of using inheritance. You also need to avoid cloning and using implicit constructors.

Objective:
Secure Coding in Java SE Application

Sub-Objective:
Secure resource access including filesystems, manage policies and execute privileged code

References:

Oracle Technology Network > Java > Secure Coding Guidelines for Java SE

Question #33 of 50

Question ID: 1327881

Given:

```
public abstract class Writer {
   public void write() {System.out.println("Writing...");}
}

public class Author extends Writer {
   public void write() {System.out.println("Writing book");}
}

public class Programmer extends Writer {
   public void write() {System.out.println("Writing code");}
   public static void main(String[] args) {
     Writer w = new Programmer();
     w.write();
   }
}
```

What is the result?

- ✗ **A)** Compilation fails.
- ✓ **B)** `Writing code`
- ✗ **C)** An exception is thrown at run time.
- ✗ **D)** `Writing book`
- ✗ **E)** `Writing...`

Explanation

The result is the output `Writing code`. This is because the object type determines which implementation of the `write` method is executed. Because the object type is `Programmer`, the `write` method found in `Programmer` will output `Writing code`.

The result is not the output `Writing ...` because the reference type does not determine which implementation of the `write` method is executed. The reference type is `Writer`, not the object type.

The result is not the output `Writing book`. This is because `Author` is not the object type.

Compilation does not fail, nor is an exception thrown at run time. The code is syntactically and semantically correct. A reference type can be a supertype of the object type.

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 > Classes and Objects

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

---

Question #34 of 50

Question ID: 1327777

Which statement is true about string equality?

- ✗ **A)** The `compareTo` method returns `1` for equality.
- ✓ **B)** The `compareTo` method returns `0` for equality.
- ✗ **C)** The `equals` method compares object references.

✗ **D)** The == operator compares character sequences.

Explanation

The `compareTo` method returns `0` for equality. The `compareTo` method returns a positive integer if the specified `String` object comes before the `String` instance, and returns a negative integer if the specified `String` object comes after the `String` instance. The `compareTo` method evaluates each character by its underlying Unicode value in sequence.

The `equals` method does not compare object references. The `equals` method compares character sequences.

The == operator does not compare character sequences. The == operator compares object references.

The `compareTo` method does not return `1` for equality. The `compareTo` method returns `0` for equality and returns a positive integer if the specified `String` object comes before the `String` instance.

Objective:
Working with Java Data Types

Sub-Objective:
Handle text using String and StringBuilder classes

References:

Java API Documentation > Java SE 11 & JDK 11 > Class String

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Numbers and Strings > Comparing Strings and Portions of Strings

---

Question #35 of 50

Question ID: 1327944

Given the following:

```
1.  public class Java11{
2.   public static void main (String[] args) {
3.    char[] src =
4.      { 'j', 'e', 's', 'p', 'r', 'e', 's',
5.      's', 'o', 'a', 'v', 'a', '7' };
6.    char[] dest = new char[8];
7.    //Insert code here
8.    System.out.println(new String(dest));
9.   }
10.  }
```

Which line of code, when inserted independently at line 7, would output `espresso`?

✓ **A)** `System.arraycopy(src, 1, dest, 0, 8);`

✗ **B)** `System.arraycopy(src, 2, dest, 0, 8);`

✗ **C)** `System.arraycopy(src, 8, dest, 0, 2);`

✗ **D)** `System.arraycopy(src, 8, dest, 0, 1);`

Explanation

The line of code `System.arraycopy(src, 1, dest, 0, 8);` , when inserted independently at line 7, would output `espresso`. The first argument specifies the array from which elements will be copied, while the third argument specifies the array into which elements are pasted. The second argument references the index position in the source array, while the fourth argument references the index position in the destination array. The fifth and final argument indicates how many elements will be copied in the operation.

The line of code `System.arraycopy(src, 8, dest, 0, 1);` would not output `espresso`. This invocation reverses the second and fifth arguments, so that the copy operation begins at the ninth element in the source array and only one character is copied. The output of this line of code would be `o`.

The line of code `System.arraycopy(src, 8, dest, 0, 2);` would not output `espresso`. This invocation begins at the ninth element in the source array and only two characters are copied. The output of this line of code would be `oa`.

The line of code `System.arraycopy(src, 2, dest, 0, 8);` would not output `espresso`. This invocation begins at the third element in the source array. The output of this line of code would be `spressoa`.

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 > Learning the Java Language > Language Basics > Arrays

---

Question #36 of 50

Question ID: 1328046

Given the following code:

```
int num1=20;
int num2=0;
int solution = num1/num2;
System.out.println(solution);
```

What will be the output?

✗ **A)** `0`

✗ **B)** `200`

✗ **C)** `20`

✓ **D)** `ArithmeticException`

Explanation

The system will throw an `ArithmeticException` because of the divide by zero operation in the following line:

```
int solution = num1/num2;
```

The other options are incorrect because a divide by zero returns an undefined value. It cannot return a numerical value.

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 ArithmeticException

---

Question #37 of 50

Question ID: 1327890

Given the following code:

```
public class DataEnvelope {
   private StringBuilder contents;
   public DataEnvelope() {contents = new StringBuilder("_"); }
   public void Stuff(byte i) { contents.append("b" + i); }
   public void Stuff(long i) { contents.append("l" + i); }
   public void Stuff(float i) { contents.append("f" + i); }
   public void Stuff(Double i) { contents.append("D" + i); }
   public void Stuff(Long i) { contents.append("L" + i); }
   public void Stuff(Object i) { contents.append("O" + i); }
   public void end() {contents.append("_");}
   public void open() {System.out.println(contents);}
}
```

Which two code fragments will generate the output `_1213b4_`?

✗ **A)**
```
DataEnvelope env = new DataEnvelope();
env.Stuff((long)2);
env.Stuff(3.0);
env.Stuff(4);
env.end();
env.open();
```

✓ **B)**
```
DataEnvelope env = new DataEnvelope();
env.Stuff(2);
env.Stuff(3);
env.Stuff((byte) 4);
env.end();
env.open();
```

✗ **C)**
```
DataEnvelope env = new DataEnvelope();
env.Stuff(2.0);
env.Stuff(3d);
env.Stuff(4);
env.end();
env.open();
```

✓ **D)**
```
DataEnvelope env = new DataEnvelope();
env.Stuff(2L);
env.Stuff(3);
env.Stuff((byte)4.0);
env.end();
env.open();
```

✗ **E)**
```
DataEnvelope env = new DataEnvelope();
env.Stuff(2);
env.Stuff(3);
env.Stuff(4);
env.end();
env.open();
```

Explanation

The following two code fragments will generate the output `_1213b4_` (line numbers are used for reference only):

```
01   DataEnvelope env = new DataEnvelope();
02   env.Stuff(2L);
03   env.Stuff(3);
04   env.Stuff((byte)4.0);
05   env.end();
06   env.open();
```

```
01   DataEnvelope env = new DataEnvelope();
02   env.Stuff(2);
03   env.Stuff(3);
04   env.Stuff((byte) 4);
05   env.end();
06   env.open();
```

In the first code fragment, line 02 matches the second overloaded `Stuff` method by explicit type `long`. Line 03 of the first code fragment and line 02 and 03 of the second code fragment match the second overloaded `Stuff` method by widening primitive conversion from the default `int` to `long`. Line 04 in both code fragments matches the first overloaded `Stuff` method by explicit casting to the type `byte`.

The following code fragment will not generate the output `_l2l3b4_` (line numbers are used for reference only):

```
01   DataEnvelope env = new DataEnvelope();
02   env.Stuff(2);
03   env.Stuff(3);
04   env.Stuff(4);
05   env.end();
06   env.open();
```

Lines 02-04 match the second overloaded `Stuff` method by widening primitive conversion from the default `int` to `long`. This code fragment will generate the output `_l2l3l4_`.

The following code fragment will not generate the output `_l2l3b4_` (line numbers are used for reference only):

```
01   DataEnvelope env = new DataEnvelope();
02   env.Stuff(2.0);
03   env.Stuff(3d);
04   env.Stuff(4);
05   env.end();
06   env.open();
```

Lines 02-03 match the fourth overloaded `Stuff` method by boxing conversion from `double` to the wrapper class `Double`. Line 04 matches the second overloaded `Stuff` method by widening primitive conversion from the default `int` to `long`. This code fragment will generate the output `_D2.0D3.0l4_`.

The following code fragment will not generate the output `_l2l3b4_` (line numbers are used for reference only):

```
01   DataEnvelope env = new DataEnvelope();
02   env.Stuff((long)2);
03   env.Stuff(3.0);
04   env.Stuff(4);
05   env.end();
06   env.open();
```

Line 02 matches the second overloaded `Stuff` method by casting to type long. Line 03 matches the fourth overloaded `Stuff` method by boxing conversion from `double` to the wrapper class `Double`. Line 04 matches the second overloaded `Stuff` method by widening primitive conversion from the default `int` to `long`. This code fragment will generate the output `_l2D3.0l4_`.

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 > Classes and Objects > Defining Methods

Oracle Technology Network > Java SE > Java Language Specification > Classes > Conversions and Promotions > Method Invocation Conversion

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Object-Oriented Programming Concepts > What is Inheritance

---

Question #38 of 50

Question ID: 1328121

You are creating coding logic for a collection of users of a particular brand of laptops in the United States:

```
List<String> laptopUsers = new ArrayList<>();
```

You have implemented a method that identifies whether a user is located in a particular city, called `isLocated()`.

Next, you write the lambda expression to use with the method:

```
user -> user.isLocated("Atlanta")
```

You need to pass the lambda expression as a parameter. Which of the following stream methods should you use? (Choose all that apply.)

- ✗ **A)** `count()`
- ✓ **B)** `filter()`
- ✓ **C)** `map()`
- ✗ **D)** `collect()`

Explanation

You could pass the lambda expression to both the `filter` and `map` methods, as both accept lambda expressions as parameters.

You could not pass a lambda expression to either the `count` or `collect` methods. Both are terminal operations that do not accept lambda expressions as arguments.

The main operations in a `Stream` pipeline and the order in which they occur are:

```
filter
sorted
map
collect
```

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 #39 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)** Compilation fails.
- ✗ **B)** `Luke, I am your father.`
- ✗ **C)** `It's not true!`
- ✗ **D)** Runtime error occurs.

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 #40 of 50

Question ID: 1327982

Given the code fragment:

```
Path path1 = Paths.get("Documents\\JavaProjects\\Java11");
Path path2 = Paths.get("Java11\\NIO\\src");
//Manipulate
String strResult = path1.resolve(path2).toString();
```

Assuming these paths are on the C: drive of a Windows system, which value is stored in the `strResult` variable?

✓ **A)** `Documents\JavaProjects\Java11\Java11\NIO\src`

✗ **B)** `C:\Documents\NetBeansProjects\JavaNIO\Java11`

✗ **C)** `C:\Documents\NetBeansProjects\JavaNIO\Java11\NIO\src`

✗ **D)** `..\..\..\Java11\NIO\src`

Explanation

The value stored in the `strResult` variable is the following:

`Documents\JavaProjects\Java11\Java11\NIO\src`

The `resolve` method appends a partial path to an existing `Path` object. In this case, `path2` is appended to the end of `path1`.

The value `..\..\..\Java11\NIO\src` is not stored in the `strResult` variable. This would be the value if the `relativize` method were used instead of the `resolve` method. The `relativize` method determines the relative path between two `Path` objects. In this case, the `strResult` variable would return the relative path for reaching `path2`, starting at the location of `path1`.

The value `C:\Documents\NetBeansProjects\JavaNIO\Java11` is not stored in the `strResult` variable. This would be the value if `path1.toAbsolutePath().toString();` were executed. The `toAbsolutePath` method will convert a relative `Path` object into an absolute `Path` object.

The value `C:\Documents\NetBeansProjects\JavaNIO\Java11\NIO\src` is not stored in the `strResult` variable. This would be the value if `path2.toAbsolutePath().toString();` was executed. The `toAbsolutePath` method will convert a relative `Path` object into an absolute `Path` object.

Objective:
Java File I/O

Sub-Objective:
Handle file system objects using java.nio.file API

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Essential Classes > Basic I/O > Path Operations

Oracle Technology Network > Java SE > Java SE Documentation > Java Platform Standard Edition 11 API Specification > java.nio.file > Interface Path

---

Question #41 of 50

Question ID: 1327756

Given the following:

```
1.   public class Java11 {
2.      public static void main (String[] args) {
3.        //Code goes here
4.        System.out.println("Value: " + b);
5.     }
6.    }
```

Which declaration line, when inserted at line 3, enables the code to compile and run?

✗ **A)** `boolean b = 0;`

✗ **B)** `boolean b = null;`

✗ **C)** `byte b = 1101;`

✓ **D)** `byte b = 0b1101;`

Explanation

The following declaration line, when inserted at line 3, enables the code to compile and run:

```
byte b = 0b1101;
```

Local variables must be initialized explicitly before they are accessed, or compilation will fail. This statement declares the variable `b` as the primitive type `byte` and initializes it to the binary value `1101` using the prefix `0b`. The decimal value is `13`, which is in the valid range for a byte.

Java provides eight primitive data types, each with a default value:

- `byte`: 8-bit data type ranging from `-128` to `127` with a default value of `0`.
- `short`: 26-bit data type ranging from `-32,768` to `32,767` with a default value of `0`.
- `int`: 32-bit data type ranging from `-2,147,483,648` to `2,147,483,647` with a default value of `0`.
- `long` 64-bit data type ranging from `-9,223,372,036,854,775,808` to `9,223,372,036,854,775,807` with a default value of `0L`.
- `float`: 32-bit floating point data type ranging from `3.40282347e38` to `1.40239846e-45` with default value of `0.0f`.
- `double`: 64-bit floating point data type ranging from `1.7976931348623157e308` to `4.9406564584124654e-324` with a default value of `0.0d`.
- `boolean`: Has only two possible values of `true` and `false`, with a default value of `false`.
- `char`: 16-bit Unicode character with default value of `"u0000"`.

In Java SE 8 and above, you can use the `int` data type to represent an unsigned 32-bit integer with a range of `0` to `2e32-1`. Similarly, you use an unsigned version of `long` to represent an unsigned 64-bit integer with a range of `0` to `2e64-1`.

When declaring and initializing variables in Java, you should delineate each declaration with a semi colon (`;`). In Java SE 10 and above, you also use the `var` keyword to infer the data type of the variable based on its

initialization. For example, line 3 could have been written as follows:

```
var b = 0b1101;
```

The declaration lines `boolean b = 0 ;` and `boolean b = null;` will not enable the code to compile and run. A `boolean` variable can be only two possible values: `true` and `false` . A `boolean` variable could be assigned to conditional expression, such as `(6 > 2)` or `(2 != 6)`.

The declaration line byte `b = 1101`; will not enable the code to compile and run. A `byte` value must be within the decimal range of `-128` to `127` (inclusive). By default, decimal literals are treated as `int` data types and must be explicitly cast to `byte` to accept the loss of precision.

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 > Language Basics > Primitive Data Types

OpenJDK > Java Local Variable Type Inference: Frequently Asked Questions

---

Question #42 of 50

Question ID: 1328189

Which of the following option(s) best describes the purpose of localizing a Java application? (Choose all that apply.)

- ✗ **A)** Improving file management
- ✓ **B)** Separating program logic from content
- ✗ **C)** Making the application more efficient
- ✗ **D)** Matching the Java API to local Java API
- ✓ **E)** Making the application region independent

Explanation

You most likely need to localize a Java application to make the application region independent and to separate program logic from content.

When localizing an application, you need to ensure that the program matches the language and other location specific requirements of a place or country. This process called *localization* is done by adding locale-specific data to the Java application. This way the same Java application can be used in different regions in the world, making it region-independent. By applying localization you can modify the content of a Java application without having to change the program's logic.

Localization may or may not improve a program's efficiency, depending on the program's actual code logic.

File management is something that the program logic and Java API will handle. Localizing an application will thus not affect file management.

Matching the Java API to local Java API is incorrect because Java API names do not need to be localized. They are not seen by the end user and are internal to the Java application.

You adapt your Java application to different languages and regions by adding localization data to the executable file. The term localization describes the process of adapting an internationalized application to a specific region or language. Localization is abbreviated as *L10n*.

Similarly, the term *internationalized* is used to describe an application that can adapt to various languages and regions of the world without needing recompilation. Internationalization is abbreviated as *i18n*.

When internationalizing your application, you need to cover both the applications input and output:

- The input will include form data, service calls, and files from users.
- The output will include all displayed information that includes language text, currencies, character and string comparisons and dates.

Also, you may need to factor in local taxes and currencies when creating an application that may need to compute monetary data.

To internationalize your application, you first need to get certain key data from the user, which includes their country, language, location, and time zone.

To perform internationalization on a Java program, you can use Java's built in classes:

- The `Locale` class, which represents the language or country and covers formatting.
- A `ResourceBundle` that contains localized information corresponding to a certain Locale.

Other classes used for internationalization include `SimpleDateFormat`, `DateFormat`, `DecimalFormat`, and `NumberFormat`.

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

---

Question #43 of 50

Question ID: 1327843

Given:

```
public class VarScope {
   int var;
   public static void main (String[] args) {
     int var = 10;
     VarScope scope = new VarScope();
     scope.var = var + 2;
     scope.adjustVar(scope.var + 2);
     System.out.println("var = " + var);
   }
   private void adjustVar(int var) {
     var += 2;
   }
}
```

What is the result?

✗ **A)** `var = 14`

✗ **B)** `var = 16`

✗ **C)** `var = 12`

✓ **D)** `var = 10`

Explanation

The result will be the output `var = 10`. The output is based on the local variable named `var` in the `main` method. The variable `var` in this scope is set to `10` and not modified until it is printed.

The result will not be the output `var = 12` because the local variable in the `main` method will be printed. This result would be the output if the instance variable `var` were printed, because the class variable is set to the local variable incremented by two.

The result will not be the output `var = 14` because the local variable in the `main` method will be printed. This result would be the output if the local variable `var` in the `adjustVar` method were printed before it is incremented by two.

The result will not be the output `var = 16` because the local variable in the `main` method will be printed. This result would be the output if the local variable `var` in the `adjustVar` method were printed after it is incremented by two.

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 > Variables

Question #44 of 50

Question ID: 1328005

Consider the following code:

```
public class AnnotationsTest {
 // Insert annotation here
 public void play() {
  System.out.println("Time to play");
 }
 public static void main(String args[])  {
  AnnotationsTest at = new AnnotationsTest();
 at.play();
 }
}
```

The `play()` method is now obsolete and replaced by a newer implementation. Which annotation will you insert for it?

✗ **A)** `@SupressWarnings`

✗ **B)** `@Inherited`

✗ **C)** `@Documented`

✓ **D)** `@Deprecated`

Explanation

You should use the `@Deprecated` annotation because the `play` method is no longer in use. `@Deprecated` is a marker annotation indicating to the compiler that the associated declaration is has now been replaced with a newer one.

`@Documented` is an incorrect option. It does not mark obsolete code but indicates Javadoc to include annotations.

`@Inherited` is an incorrect option. It does not mark obsolete code but inheritance by a subclass. It can be used on class declarations.

`@SupressWarnings` is an incorrect option. It does not mark obsolete code by specifies exact warnings that the compiler needs to 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 #45 of 50

Question ID: 1328119

Given:

```
Stream<String> ss = Stream.of("A", "B", "c", "D", "e");
System.out.println(ss.sequential().findAny());
```

What is the result?

✗ **A)** `A`

✗ **B)** A runtime exception is thrown.

✓ **C)** `Optional[A]`

✗ **D)** `null`

✗ **E)** `ABcDe`

Explanation

The result is `Optional[A]`. The `findAny` method picks an item from the stream. For an ordered stream that is being processed sequentially, this will be the first item to reach the `findAny` method. However, the method does not simply return the item that it finds, because if the stream were empty, `findAny` could return `null`. To avoid the trouble with null pointers, `findAny` returns the result in an `Optional` wrapper instance instead.

The output is not `ABcDe` because `findAny` returns only one item from the stream and wraps the result in an `Optional` wrapper instance.

The output is not `A` because the returned value from `findAny` is wrapped in an `Optional`.

The output is not `null` because an item is found in the stream. Also, the return of `findAny` is always an `Optional`, even if no item is found.

A runtime exception is not thrown because there are no errors in the given code.

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 #46 of 50

Question ID: 1327899

Which code segment correctly defines an interface?

✗ **A)**
```
interface Vehicle2 {
        public final static int wheels = 4;
        protected abstract String move() { return "moving"; }
    }
```

✗ **B)**
```
interface Vehicle1 {
   public final int wheels;
   public abstract void move();
}
```

✓ **C)**
```
interface Vehicle4 {
   int wheels = 4;
   void move()throws
javax.naming.OperationNotSupportedException;
}
```

✗ **D)**
```
interface Vehicle3 {
   abstract static int wheels;
   public final String move(){ return "moving"; }
}
```

Explanation

The following code segment correctly defines an interface:

```
interface Vehicle4 {
   int wheels = 4;
   void move() throws javax.naming.OperationNotSupportedException;
}
```

`Vehicle4` defines an interface with the constant `wheels` and the method `move`. By default, all members of an interface are public, while variables are implicitly static and final, and methods are implicitly abstract. Constants must be set to a value within the interface. Methods declarations can also contain exception declarations.

`Vehicle1` does not correctly define an interface. The constant `wheels` is not set to a value. Also, the `final`, `public`, and `abstract` keywords are not required when declaring variables and methods in an interface.

`Vehicle2` does not correctly define an interface. The `move` method cannot contain a body nor use the `protected` keyword. Also, the `final`, `static`, and `abstract` keywords are not required when declaring variables and methods in an interface.

`Vehicle3` does not correctly define an interface. Variables cannot be declared with the `abstract` keyword, and methods cannot be declared with the `final` keyword or include a method body.

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 #47 of 50

Question ID: 1327831

Given:

```
class Job {
   private String name;
   private String[] reqs;
   public Job(String name, String... reqs) {
     this.name = name;
     this.reqs = reqs;
   }
   public void post() {/*Implementation omitted*/}
   //insert code here
}

class Programmer extends Job {
   private String language;
   public Programmer(String name, String... reqs) {super(name, reqs);}
   public void post(String language) {
     post();
     post(language.toCharArray());
   }
}
```

Which method, when inserted in the code, will compile?

✗ **A)** `public void post(String rawData) {/*Implementation omitted*/}`

✓ **B)** `protected void post(char[] rawData) {/*Implementation omitted*/}`

✗ **C)** `void post(String rawData) {/*Implementation omitted*/}`

✗ **D)** `private void post(char[] rawData) {/*Implementation omitted*/}`

Explanation

The following method, when inserted in the code, will compile:

```
protected void post(char[] rawData) {/*Implementation omitted*/}
```

Because this overloaded method is invoked in the subclass `Programmer` , it must be declared with an access modifier other than `private` . This method is declared with the access modifier `protected`, so that it is available only to subclasses, such as `Programmer`.

The method declared with the `private` keyword will not compile when inserted in the code. This is because the method is unavailable to the `Programmer` class. The access modifier `private` indicates that a class member is only available to that class.

The following methods, when inserted in the code, will not compile:

```
public void post(String rawData) {/*Implementation omitted*/}
void post(String rawData) {/*Implementation omitted*/}
```

Although both methods are available to the `Programmer` class, they do not match the signature required by the `Programmer` class. The statement `post(language.toCharArray());` requires a method that specifies a `char` array parameter.

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

---

Question #48 of 50

Question ID: 1328035

Given the code fragment:

```
for (int x = 0; x < 10; x--) {
   do {
     System.out.println("Loop!");
     System.out.println("x:" + x);
   } while (x++ < 10);
}
```

How many times is `Loop!` printed?

✗ **A)** 9

✗ **B)** 10

✗ **C)** An infinite number

✓ **D)** 11

Explanation

`Loop!` is printed 11 times. The outer `for` block initializes `x` to 0 and decrements its value by 1 after each iteration. The inner `do-while` block will execute once and then increment `x` by 1 after each iteration. If `x` is 10 or greater, then execution will exit both blocks. The following table tracks the variable `x` value for each iteration in each block:

Iteration	x value	Block	`Loop!` Printed
1	0	Outer	No
1	0	Inner	Yes
2	1	Inner	Yes – 2nd time
3	2	Inner	Yes – 3rd time
4	3	Inner	Yes – 4th time
5	4	Inner	Yes – 5th time
6	5	Inner	Yes – 6th time
7	6	Inner	Yes – 7th time
8	7	Inner	Yes – 8th time
9	8	Inner	Yes – 9th time
10	9	Inner	Yes – 10th time
11	10	Inner	Yes – 11th time
11	11	Outer	No

`Loop!` is not printed 9 times because `x` is initialized to 0 and the `do-while` block will execute at least once.

`Loop!` is not printed 10 times because the `do-while` block will execute at least once.

`Loop!` is not printed an infinite number because the inner `do-while` block will increment `x` so that both inner and outer blocks will exit.

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 #49 of 50

Question ID: 1328128

Given:

```
System.out.println(
  Stream.of(new StringBuilder("A"), new StringBuilder("B"))
  .collect(Collectors.joining(",", ">", "<"))); // line n1
```

What is the result?

- ✗ **A)** `AB`
- ✗ **B)** Compilation fails at line `n1`.
- ✗ **C)** `A,B`
- ✓ **D)** `>A,B<`
- ✗ **E)** `<A,B>`

Explanation

The result is the output `>A,B<`. The `Collectors.joining` method creates a `Collector` that joins the three provided `CharSequence` objects to decorate the output. The first argument is a separator between stream items, the second argument is the prefix before the stream items, and the third is the suffix at the end of stream items.

The output `AB` is incorrect because the output omits the prefix, suffix, and delimiter characters.

The output `A,B` is incorrect because the output omits the prefix and suffix characters.

The output `<A,B>` is incorrect because the output reverses the prefix and suffix characters.

There are no compilation errors at line `n1`.

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 > Collectors

Question #50 of 50

Question ID: 1327929

Which two types extend `SortedMap`?

✗ **A)** `ArrayList`

✓ **B)** `NavigableMap`

✓ **C)** `TreeMap`

✗ **D)** `HashMap`

✗ **E)** `Hashtable`

Explanation

The `TreeMap` class and `NavigableMap` interface extend `SortedMap`.

The `Map` interface is intended for name/value pairs with no duplicate keys. `SortedMap` extends this functionality by adding a natural order of elements. The `NavigableMap` interface adds additional navigation methods to `SortedMap`, while `TreeMap` is an implementation class of `NavigableMap`. The `TreeMap` class implements the `Map` interface. It stores key-value pairs in a sorted order. This allows swift retrievals of these values during searches.

For example, if you wanted to store names and employee codes, you could implement it with code similar to this:

```
TreeMap<Integer,String> peopleList = new TreeMap<Integer,String>();

peopleList.put(001,"Anita");
peopleList.put(103,"Rene");
peopleList.put(231,"Jessica");
peopleList.put(543,"Darin");
```

`Hashtable` and `HashMap` do not extend `SortedMap`. `Hashtable` and `HashMap` are classes that implement `Map`, but not `SortedMap`. Unlike `Hashtable`, `HashMap` provides additional operations and permits a null key and null values. `TreeMap` objects are different from `HashMap` objects in that `TreeMap` objects cannot contain null values and maintain items in an ascending order.

The `ArrayList` class in Java provides a resizable array of objects. It is a subclass of the `AbstractList` class and implements the `List` interface.

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 Map Interface

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Collections > Implementations > Map Implementations