Quick Quiz September 12, 2022

Test ID: 223362231

Question #1 of 50

Question ID: 1327856

Given:

```
public abstract class Voter {
   private String partyAffiliation;
   public Voter(String partyAffiliation) {
     this.partyAffiliation = partyAffiliation;
   }
   public String getPartyAffiliation() { return partyAffiliation; }
   public abstract void vote();
}
```

Which two statements are true about `Voter`?

- ✗ **A)** Concrete subclasses of `Voter` must use a default constructor.
- ✓ **B) The `Voter` class cannot be instantiated.**
- ✓ **C) Concrete subclasses of `Voter` must implement the `vote` method.**
- ✗ **D)** Subclasses of `Voter` cannot override its methods.
- ✗ **E)** The `Voter` class cannot be extended.
- ✗ **F)** Subclasses of `Voter` must implement the `getPartyAffiliation` method.

Explanation

The following two statements are true about the `Voter` class:

- This class cannot be instantiated.
- Concrete subclasses of `Voter` must implement the `vote` method.

Because `Voter` is declared as abstract, the class cannot be instantiated. However, abstract classes can include constructors for subclasses to invoke. Because the `vote` method is declared as abstract, a subclass must implement this method or declare itself abstract.

The `Voter` class can be extended because abstract classes require subclasses to provide implementation. A class declared with the keyword `final` cannot be extended.

Subclasses of `Voter` can override its methods. In addition, any abstract methods must be overridden in concrete subclasses.

Concrete subclasses of `Voter` should not use a default constructor. A default constructor will invoke a parameterless constructor in `Voter`. Because `Voter` declares a constructor with a parameter, subclasses must contain a constructor that explicitly invokes this superclass constructor.

Concrete subclasses do not need to implement the `getPartyAffiliation` method. Because this method is not declared as abstract, this method can be overridden, but overriding it is completely optional.

Objective:
Java Object-Oriented Approach

Sub-Objective:
Create and use subclasses and superclasses, including abstract classes

References:

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

---

Question #2 of 50

Question ID: 1327774

Which statement is true about string manipulation?

- ✗ **A)** **Manipulation methods in the `StringBuilder` class create and return new `String` objects.**
- ✗ **B)** `String` objects are stored as variable-length arrays of characters.
- ✗ **C)** Manipulation methods in the `String` class accept `StringBuilder` objects.
- ✓ **D)** `String` objects cannot be modified once they are created.

Explanation

`String` objects are immutable and cannot be modified once they are created. Manipulation methods do not modify the actual `String` object themselves, but instead create and return new `String` objects.

`String` objects are not stored as variable-length arrays of characters. `String` objects are fixed-length arrays of characters, because their lengths cannot be changed.

Manipulation methods in the `StringBuilder` class do not create and return new `String` objects. Manipulation methods in the `StringBuilder` class allow for direct modification with string literals.

Manipulation methods in the `String` class do not accept `StringBuilder` objects. `String` objects are accepted by manipulation methods in the `StringBuilder` class.

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 > Strings

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Numbers and Strings > The StringBuilder Class

---

Question #3 of 50

Question ID: 1328013

Given the following code fragment:

```
public class JavaSETest {
   public static void main(String[] args) {
     int index = 0;
     int number = (index<5)? 10 : "No";
     System.out.println(number);
   }
}
```

What is the output?

✓ **A) Compiler error**

✗ **B)** `0`

✗ **C)** `No`

✗ **D)** `10`

Explanation

There will be a compiler error because a `String` cannot be implicitly converted to an `int` inside of the ternary conditional expression. This line causes the following error:

```
int number = (index<5)? 10 : "No";
```

The other options are incorrect because the code does not compile at all.

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 > Equality, Relational and Conditional Operators

---

Question #4 of 50

Question ID: 1327978

Consider the following class:

```
public class Employee {
  private static final long SerialVersionUID = 1234L;
  private String emp_name;
  private int emp_age;
  private String emp_gender;
  Employee() {};
  Employee(String emp-name, int emp-age, String emp-gender) {
    this.emp_name = emp-name;
    this.emp_age = emp-age;
    this.emp_gender = emp-gender;
  }
  @Override
  public String toString() {
    return "Employee Name:" + emp_name + "\nAge: " + emp_age + "\nGender: " + emp_gender;
  }
}
```

Which interface needs to be implemented by this class to successfully convert its objects to a byte stream?

- ✓ **A)** **`Serializable`**
- ✗ **B)** `Closeable`
- ✗ **C)** `Iterable`
- ✗ **D)** `Comparable`

Explanation

You should implement the `Serializable` interface. The class requires the following code in its definition:

`public class Employee` **`implements Serializable`**

Serialization is a process of converting objects to a stream of bytes that can be persisted. Objects converted to a byte stream can then be written to a file. The Java library provides the Serialization API for this process. A Java object is serializable if its class or superclass implements the `java.io.Serializable` or `java.io.Externalizable` interfaces. The `Serializable` interface is a marker interface whose purpose is simply to *mark* the objects of classes that implement it so that these objects have certain capabilities.

Each serializable class requires a `SerialVersionUID`, which is a version number needed by the Serialization runtime environment for verifying the compatibility of the sender and receiver of a serialized object during the process of deserialization.

The `Iterable` interface allows objects implementing it to be targets for `foreach` statements. This interface does not help implement serialization.

The `Comparable` The interface, when implemented, allows objects to be compared with other objects. This interface does not help implement serialization.

The `Closeable` interface allows users to close the source or destination of data. This interface does not help implement serialization.

Objective:
Java File I/O

Sub-Objective:
Implement serialization and deserialization techniques on Java objects

References:

Oracle Technology Network > Java SE > Java SE Documentation > Java Platform Standard Edition 11 > API Specification > java.base > java.io > Interface Serializable

Oracle Technology Network > Java SE > Java SE Documentation > Java Platform Standard Edition 11 > API Specification > java.base > Package: java.io

Oracle Technology Network > Software Downloads > Documentation > Java Object Serialization

Oracle Java Documentation > Learning the Java Language > Interfaces and Inheritance > Interfaces

---

Question #5 of 50

Question ID: 1327935

Given the following code fragment:

```
public class TestArrayList {
   public static void main (String[] args) {
     ArrayList<String> names = new ArrayList<>(
    Arrays.asList("Amy","Anne","Brian","George","Ruth","Todd"));
     names.add("Jason");
     System.out.println(names[6]);
   }
}
```

What is the result?

- ✗ **A)** Code throws a runtime exception.
- ✗ **B)** `Todd`
- ✗ **C)** `Jason`
- ✗ **D)** `Ruth`
- ✓ **E)** **Code compilation fails.**

Explanation

The code fragment will fail compilation because the syntax of `names[6]` is invalid. This is the syntax for accessing the seventh element in an array, not an `ArrayList` object. If the correct syntax `names.get(6)` were used, the result would be `Jason`.

The code fragment will not throw a runtime exception. Unlike arrays, `ArrayList` objects are resizable to accommodate the removal or addition of new elements. An `IndexOutOfBoundsException` will not be thrown because the `ArrayList` object resizes to seven elements after adding the new element.

The code fragment will not result in the output `Ruth`, `Todd`, or `Jason`. The syntax error prevents the code from being compiled and run.

Objective:
Working with Arrays and Collections

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

References:

Oracle Documentation > Java SE 11 API > Class ArrayList<E>

Question #6 of 50

Question ID: 1328159

Given the code fragment:

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

Which two changes will make read/write operations on `empList` thread-safe?

- ✗ **A)** The `ArrayList` class should be replaced with the `ConcurrentHashMap` class.
- ✗ **B)** The code fragment should invoke the `CyclicBarrier.await` method on the current thread.
- ✓ **C)** **The `ArrayList` class should be replaced with the `CopyOnWriteArrayList` class.**

✗ **D)** The code fragment should be declared in a synchronized block.

✗ **E)** The code fragment should invoke the `Thread.sleep` method.

✓ **F)** **Read/write operations should be declared in a synchronized block.**

✗ **G)** The code fragment should invoke the `Thread.interrupt` method on the current thread.

Explanation

To ensure that read/write operations on `empList` are thread-safe, you should make one of the following two changes:

- Read/write operations should be declared in a synchronized block.
- The `ArrayList` class should be replaced with the `CopyOnWriteArrayList` class.

To prevent memory consistency errors, a resource must ensure that write operations are visible to all threads when they occur, so that subsequent operations are consistent known as a *happens-before* relationship. You can either use the thread-safe collection for the `ArrayList` class that provides automatic synchronization, namely `CopyOnWriteArrayList`, or perform manual locking using a synchronized block.

The code fragment should not be declared in a synchronized block. Instantiation within a synchronized block will not ensure that read/write operations are thread-safe.

The `ArrayList` class should not be replaced with the `ConcurrentMap` class. The thread-safe version of `ArrayList` is the `CopyOnWriteArrayList` collection, while the interface `ConcurrentMap` is the thread-safe subinterface of `Map`.

The code fragment should not invoke the `Thread.sleep` method. This method will not make read/write operations thread-safe. The `Thread.sleep` method will only pause execution of the current thread.

The code fragment should not invoke the `Thread.interrupt` method on the current thread. This method will not make read/write operations thread-safe. The `Thread.interrupt` method will only terminate or redirect execution a thread.

The code fragment should not invoke the `CyclicBarrier.await` method on the current thread. This is an instance method requiring an initialized `CyclicBarrier` object. A `CyclicBarrier` determines a size of specific number of waiting threads, which once exceeded, triggers an optional predefined action. The `await` method will pause the thread until all other threads in the barrier have been paused.

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 > Java Platform Standard Edition 11 API Specification > java.util.concurrent > Class CopyOnWriteArrayList<E>

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 > Concurrent Collections

---

Question #7 of 50

Question ID: 1327842

Given:

```
public class VarScope {
   static int var;
   public static void main (String[] args) {
     int var = 9;
     printVar();
   }
   public static void printVar() {
     int var = 10;
     System.out.print("var = " + var++);
   }
}
```

What is the result?

- ✗ **A)** `var = 0`
- ✗ **B)** `var = 9`
- ✗ **C)** `var = 11`
- ✓ **D)** **`var = 10`**

Explanation

The result will be the output `var = 10`. The output is based on the local variable named `var` in the `printVar` method. The variable `var` in this scope is set to `10`, printed, and then incremented to `11`.

The result will not be the output `var = 0` because the local variable in the `printVar` method will be printed. This result would be the output if the class variable `var` were printed, because class and instance variables are automatically initialized to their default values.

The result will not be the output `var = 9` because the local variable in the `printVar` method will be printed. This result would be the output if the local variable `var` in the `main` method were printed.

The result will not be the output `var = 11` because the local variable in the `printVar` method will be printed before it is incremented. This result would be the output if the prefix increment operator were specified, not the postfix increment operator.

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 #8 of 50

Question ID: 1327757

Which three declaration statements correctly initialize their variables? (Choose three.)

- ✓ **A)** **`float f2 = 10.01E2f;`**
- ✗ **B)** `var v;`
- ✓ **C)** **`boolean b1 = (6 < 4);`**
- ✗ **D)** `boolean b2 = 1;`
- ✗ **E)** `int i2 = -6,000;`
- ✗ **F)** `float f1 = 10.01;`
- ✓ **G)** **`var v = true;`**
- ✗ **H)** `int i1 = 40.4;`

Explanation

The following three declaration statements correctly initialize their variables:

```
boolean b1 = (6 < 4);
```

```
float f2 = 10.01E2f;
```

```
var v = true;
```

The `boolean` variable `b1` is assigned a conditional expression that evaluates to `false` . A `boolean` variable can be only two possible values: `true` and `false` . The `float` variable `f2` is assigned a floating-point number with no loss of precision. By default, floating-point literals are treated as double values. The floating-point literal assigned to `f2` uses the `f` postfix, so that the literal is treated as a float value. The variable `v` is declared using the `var` keyword, meaning that the data type is inferred by the value assigned to it. Thus, `v` will be allocated as a `boolean` value.

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.

The declaration statement `boolean b2 = 1;` does not correctly initialize its variable. A `boolean` variable can be only two possible values: `true` and `false`.

The declaration statements `int i1 = 40.4;` and `int i2 = -6,000;` do not correctly initialize their variables. An `int` variable can be assigned only integer values between `-2,147,483,648` and `2,147,483,647` (inclusive). A valid literal cannot include a decimal point or comma separator. In Java 7 and above, you can use the underscore in numeric literals like a comma separator. For example, `int i2 = -6_000`; is a valid declaration statement.

The declaration statement `float f1 = 10.01`; does not correctly initialize its variable. Because floating-point literals are treated as `double` values, a loss of precision warning will prevent the code from compiling. You can either cast the value as `double` or use the `f` postfix in the literal to ensure compilation.

The declaration statement `var v;` will not compile, because it does not specify an initialization value from which to infer a data type. When using the `var` keyword, you must ensure the data type can be inferred by providing an initial value.

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 #9 of 50

Question ID: 1327913

Given the following code, in a single source file:

```
@FunctionalInterface
interface Funct1 {
}

@FunctionalInterface
interface Funct2 {
  static void doStuff(){}
}

@FunctionalInterface
interface Funct3 {
  static void doStuff(){}
  void doOtherStuff();
}

@FunctionalInterface
interface Funct4 {
  default void doStuff(){}
  abstract void doOtherStuff();
  void doMoreStuff();
}

@FunctionalInterface
interface Funct5 extends Funct2 {
  void doStuff();
}
```

Which interfaces will compile successfully? (Choose all that apply.)

- ✗ **A)** Funct1
- ✗ **B)** Funct2
- ✓ **C)** **Funct3**
- ✓ **D)** Funct5
- ✗ **E)** Funct4

Explanation

The interfaces `Funct3` and `Funct5` will compile successfully. The annotation `@FunctionalInterface` causes the compiler to verify that an interface defines exactly one abstract method. `Funct3` declares a single abstract method. It also declares a single static method, which is irrelevant to the issue of being a functional interface. `Funct5` extends `Funct2`, but its static methods in interfaces are not inherited, and therefore do not conflict with methods in extending interfaces. Because `Funct5` also declares a single abstract method, it compiles successfully.

`Funct1` will not compile because it declares no abstract methods. `Funct1` fails to fulfill the requirements of the annotation `@FunctionalInterface`.

`Funct2` will not compile because it does not declare an abstract method. `Funct2` declares a single static method, but because it does not declare an abstract method, it fails to compile.

`Funct4` will not compile because it does not declare a static abstract method. `Funct4` declares two abstract methods, but the methods `doOtherStuff` and `doMoreStuff` are not static, so it fails to compile.

Objective:
Java Object-Oriented Approach

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

References:

Java Platform Standard Edition 11 > API > java.lang > Annotation Type FunctionalInterface

---

Question #10 of 50

Question ID: 1327950

Given the code fragment:

```
ArrayList<Student> students = new ArrayList<>();
students.add(new Student(3, "George" ));
students.add(new Student(4, "Robin" ));
students.add(new Student(1, "Aima" ));
students.add(new Student(3, "Robin" ));
Collections.sort(students);
System.out.print(students);
```

You want to produce this output:

```
[1: Aima, 3: George, 3: Robin, 4: Robin]
```

Which of the following implementations of the `Student` class will generate this output?

✗ **A)**
```
class Student implements Comparator<Student> {
  private String name; private int id;
  public Student(String name, int id) {
    this.name = name; this.id = id;
  }
  public int compare(Student s1, Student s2) {
    return s1.name.equals(s2.name) ?
      Integer.valueOf(s1.id).compareTo(s2.id) :
s1.name.compareTo(s2.name);
  }
  public String toString() { return id + ": " + name; }
}
```

✓ **B)**
```
class Student implements Comparable <Student> {
  private int id; private String name;
  public Student(int id, String name) {
    this.id = id; this.name = name;
  }
  public int compareTo(Student s) {
    return name.equals(s.name) ?
      Integer.valueOf(id).compareTo(s.id) :
name.compareTo(s.name);
  }
  public String toString() { return id + ": " + name; }
}
```

✗ **C)**
```
class Student {
  private String name; private int id;
  public Student(String name, int id) {
    this.name = name; this.id = id;
  }
  public String toString() { return id + ": " + name; }
}
```

✗ **D)**
```
class Student {
  private int id; private String name;
  public Student(int id, String name) {
    this.id = id; this.name = name;
  }
  public String toString() { return id + ": " + name; }
}
```

Explanation

The following implementations of the `Student` class will generate the required output:

```
class Student implements Comparable <Student> {
  private int id; private String name;
  public Student(int id, String name) {
    this.id = id; this.name = name;
  }
  public int compareTo(Student s) {
    return name.equals(s.name) ?
      Integer.valueOf(id).compareTo(s.id) : name.compareTo(s.name);
  }
  public String toString() { return id + ": " + name; }
}
```

The `Comparable` interface is implemented by elements that can be sorted within a collection. The `compareTo` method returns an integer indicating whether an element is equal to, less than, or greater than another element. In this implementation, elements are first sorted alphabetically by name. If two elements have the same name, as in `Student(3, "Robin" )` and `Student(4, "Robin" )`, then they are sorted by integer value. The default sort is ascending order (natural order).

The classes that do not implement the `Comparable` interface will not generate the required output. Elements in a collection must implement `Comparable` to use the `sort` method. Also, the order of fields in a class will not affect how elements are sorted.

The class that implements the `Comparator` interface will not generate the required output. A `Comparator` implementation is required when sorting elements in an order that is not the default natural order. An overloaded version of the `sort` method accepts both a collection of `Comparable` objects and a `Comparator` object.

Objective:
Working with Arrays and Collections

Sub-Objective:
Sort collections and arrays using Comparator and Comparable interfaces

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Collections > Interfaces > Object Ordering

---

Question #11 of 50

Question ID: 1328031

Given the following output:

```
8
5
2
-1
```

Which code fragment generates this output?

✗ **A)**
```
int x = 0, y = 10;
do {
   x += 2;
   System.out.println(x - y);
   y --;
} while ( x - y > 0 );
```

✗ **B)**
```
int x = 10, y = 0;
do {
   x += 2;
   System.out.println(x - y);
   y --;
} while ( x - y > 0 );
```

✓ **C)**
```
int x = 0, y = 10;
do {
   x += 2;
   System.out.println(y - x);
   y --;
} while ( y - x > 0 );
```

✗ **D)**
```
int x = 10, y = 0;
do {
   x += 2;
   System.out.println(y - x);
   y --;
} while ( y - x > 0 );
```

Explanation

The following code fragment will generate the required output:

```
int x = 0, y = 10;
do {
   x += 2;
   System.out.println(y - x);
   y --;
} while ( y - x > 0 );
```

This code fragment initializes x to 0 and y to 10. In the `do-while` block, x is incremented by 2 and y is decremented by 1. The difference between y and x is printed and repeated until the difference is 0 or less. The variable x is incremented by 2 before the difference is printed, while y is decremented afterwards. The following table tracks variable values for each iteration:

Iteration	x value	Difference	y value
1	2	8	9
2	4	5	8
3	6	2	7
4	8	-1	6

The code fragments that initialize x to 10 and y to 0 will not generate the required output. If the difference in the expression for the print statement and the `while` statement is `y - x`, then the output will be `-12`. Otherwise, the result will be an endless loop because the expression `y - x` will always be positive.

The code fragment that initializes x to 0 and y to 0 and uses the expression `y - x` for the print statement and `while` statement will not generate the required output. The output will be `-8`.

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 while and do-while Statements

---

Question #12 of 50

Question ID: 1327970

Given:

```
import java.io.*;

class CharacterName implements Serializable {
  String given, sur;
}
class GameCharacter {
  CharacterName name = new CharacterName();
  int level, experience;
}
class PlayerCharacter extends GameCharacter implements Serializable {
  Date created = new Date();
  transient String player;
  static int numPlayers = 1;
}

public class ObjectSerializer {
  public static void main(String[] args) {
    PlayerCharacter pc = new PlayerCharacter();
    PlayerCharacter.numPlayers = 2;
    pc.name.given="Tristan"; pc.name.sur="Bolt";
    pc.level = 1; pc.experience = 1000;
    pc.player="Joshua";
    try(ObjectOutputStream strObj = new ObjectOutputStream(
     new FileOutputStream("object.txt")) ) {
      strObj.writeObject(pc);
    }catch (Exception ex) {
      System.err.print(ex);
    }
  }
}
```

Which field(s) of pc are stored in `object.txt` after the program executes?

- ✗ **A)** **`name`**
- ✓ **B)** **`created`**
- ✗ **C)** `level`
- ✗ **D)** **`numPlayers`**
- ✗ **E)** `player`
- ✗ **F)** `experience`

Explanation

Only the created field of `pc` is stored in `object.txt` after the program executes. A class that can be written to and read from a stream uses the marker interface `Serializable`. By default, all instance, non-transient fields are serializable, regardless of the access modifier. If a field is declared with the `transient` keyword, then this field will be omitted during serialization. Any inherited fields are serializable only if the superclass is also declared with the interface `Serializable`. Because `GameCharacter` does not implement `Serializable`, only fields declared in `PlayerCharacter` are serializable. The `created` field is the only one that does not include the keyword `static` or `transient`.

The `name` field is not stored in `object.txt` after the program executes. Although the class `CharacterName` is serializable, the `GameCharacter` class in which it is declared is not serializable. If `name` were declared in `PlayerCharacter`, then its fields `given` and `sur` would be serialized as an object field in `PlayerCharacter`.

The `level` and `experience` fields are not stored in `object.txt` after the program executes. Although `PlayerCharacter` inherits these fields from `GameCharacter`, `GameCharacter` does not implement `Serializable`. Thus, the fields in `GameCharacter` are not serializable.

The `player` field is not stored in `object.txt` after the program executes because player is declared with the `transient` keyword. By default, transient fields are not serializable.

The `numPlayers` field is not stored in `object.txt` after the program executes because `numPlayers` is declared with the `static` keyword. By default, static fields are not serializable. Static fields are associated with the class, not its instances. As long as the class is loaded, its fields are stored in memory.

Objective:
Java File I/O

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

References:

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

Oracle Technology Network > Java SE > Java Language Specification > System Architecture > Chapter 1 > Defining Serializable Fields for a Class

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

---

Question #13 of 50

Question ID: 1328053

Given the following method:

```
public static String readData (Path filePath)
  throws IOException, IllegalArgumentException {
   //implementation omitted
}
```

Which two code fragments will compile?

✓ **A)**
```
public static void main(String[] args)
     throws IOException{
      readData(Paths.get("test.txt"));
    }
```

✗ **B)**
```
public static void main(String[] args) {
   readData(Paths.get("test.txt"));
}
```

✓ **C)**
```
public static void main(String[] args) {
   try {
     readData(Paths.get("test"));
   } catch (IOException ex) {
     System.err.println(ex);
   }
}
```

✗ **D)**
```
public static void main(String[] args)
  throws IllegalArgumentException{
   readData(Paths.get("test.txt"));
}
```

✗ **E)**
```
public static void main(String[] args) {
   try {
     readData(Paths.get("test"));
   } catch (IllegalArgumentException ex) {
     System.err.println(ex);
   }
}
```

Explanation

The following two fragments will compile:

```
public static void main(String[] args)
  throws IOException{
   readData(Paths.get("test.txt"));
}
```

```
public static void main(String[] args) {
   try {
     readData(Paths.get("test"));
   } catch (IOException ex) {
     System.err.println(ex);
   }
}
```

The `readData` method specifies it throws `IOException`, so any invocations of those methods must either catch `IOException` or specify that the parent method will throw that exception. This is known as a checked exception. Because `IllegalArgumentException` is a subclass of `RuntimeException`, handling this exception is not required. Checked exceptions include any exceptions, except for `Error`, `RuntimeException`, and their subclasses.

The code fragments that omit specifying or catching `IOException` will not compile. The `readData` method specifies it throws `IOException`, so any invocations of those methods must catch `IOException` or specify that the parent method throw that exception.

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 > Putting it All Together

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

---

Question #14 of 50

Question ID: 1327805

Given the following class:

```
public class Java11 {
   static Customer cust;
   public static void main (String[] args) {
     cust.id = 1;
     cust.name = "Jessica Martinez";
     cust.display();
   }
}

class Customer {
   public int id;
   public String name;
   public boolean preferred;
   public void display() {
     String pOutput = (preferred)? "preferred" : "not preferred";
     System.out.format("%s (%d) is %.", name, id, pOutput);
   }
   public boolean isPreferred() {
     return preferred;
   }
}
```

What is the result?

- ✗ **A)** **`Jessica Martinez (1) is not preferred.`**
- ✗ **B)** A compile error is produced.
- ✗ **C)** `Jessica Martinez (1) is preferred.`
- ✓ **D)** A runtime error is produced.

Explanation

The result is a runtime error because the `cust` variable is `null` and throws a `NullPointerException` when attempting to access `Customer` instance members. By default, `static` and instance members are provided default values automatically. The default value for a reference type is `null`.

The result is not `Jessica Martinez (1) is preferred.` because a runtime error occurs before the `display` method is invoked. If the `cust` variable referenced an instantiated `Customer` object, then the preferred variable would default to the value `false`. This output would be the result if the `preferred` field were set explicitly to `true`.

The result is not `Jessica Martinez (1) is not preferred.` because a runtime error occurs before the `display` method is invoked. If the `cust` variable referenced an instantiated `Customer` object, then this output would be the result.

The result is not a compile error because the code has no syntax issues.

Objective:
Java Object-Oriented Approach

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

References:

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

---

Question #15 of 50

Question ID: 1327988

Which code statement would you use to write a collection named `listLines` to a file located at `filePath`?

- ✓ **A)** `Files.write(filePath, listLines, Charset.forName("US-ASCII"));`
- ✗ **B)** **`BufferedWriter.write(filePath, listLines, Charset.forName("US-ASCII"));`**
- ✗ **C)** `UTF-8.write(filePath, listLines, Charset.forName("US-ASCII"));`
- ✗ **D)** `Path.write(filePath, listLines, Charset.forName("US-ASCII"));`

Explanation

You should use the following code:

```
Files.write(filePath, listLines, Charset.forName("US-ASCII"));
```

The options using `Path`, `BufferedWriter`, and the ASCII character set are all incorrect because only the `Files` class provides the methods to write to a file at a path.

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 #16 of 50

Question ID: 1327998

Given the code fragment:

```
01  try (
02  Connection cn = DriverManager.getConnection(dbURL);
03  ) {
04    //Insert code here
05    Statement stUpdateCost = cn.createStatement();
06    Statement stUpdatePrice = cn.createStatement();
07    stUpdateCost.executeUpdate(update1);
08    stUpdatePrice.executeUpdate(update2);
09    cn.commit();
10  } catch (Exception ex) {
11    System.err.println(ex.getMessage());
12  }
```

Which code statement, when inserted at line `04`, will ensure that `stUpdateCost` and `stUpdatePrice` are executed in a single transaction?

✓ **A)** `cn.setAutoCommit(false);`

✗ **B)** `cn.setAutoCommit(true);`

✗ **C)** **`cn.rollback();`**

✗ **D)** `cn.setSavepoint();`

✗ **E)** `cn.releaseSavepoint();`

Explanation

To ensure that `stUpdateCost` and `stUpdatePrice` are executed in a single transaction, you should insert the following code statement at line `04`:

```
cn.setAutoCommit(false);
```

By default, SQL statements in a single connection are committed automatically after being executed. This is known as *auto-commit* mode. To control a transaction explicitly using the commit method, you should first disable auto-commit mode.

The code statement `cn.setAutoCommit(true);` is incorrect, because it will explicitly enable auto-commit mode. In the default auto-commit mode, SQL statements in a single connection are committed automatically after being executed.

The code statement `cn.setSavepoint();` is incorrect, because it will not execute the SQL statements in a single transaction. It will delineate a save point for rollback.

The code statement `cn.releaseSavepoint();` is incorrect, because it will not execute the SQL statements in a single transaction. It will remove a save point for rollback.

The code statement `cn.rollback();` is incorrect, because it will not execute the SQL statements in a single transaction. This method will release any database locks held by the connection and undo any temporary results.

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 > Using Transactions

Question #17 of 50

Question ID: 1328025

Given the following:

```
int x = 0;
```

Which code fragment increments x to 10?

✗ **A)** `while (x < 11 ? 1 : 0) { x++; }`

- ✗ **B)** `while (x < 10 ? 1 : 0) { x++; }`
- ✗ **C)** `while (x < 11) { x++; }`
- ✓ **D)** **`while (x < 10) { x++; }`**

Explanation

The code fragment `while (x < 10) { x++; }` increments x to 10. The expression in the `while` statement will be evaluated 11 times. In the first iteration, the value of x is 0. It is then incremented to 1 using the statement x++. In the final iteration where the `while` expression evaluates to true, x is 9, and the statement x++ increments x to 10.

The code fragment `while (x < 11) { x++; }` will not increment x to 10. The final value of x will be 11 because the expression in the `while` statement will evaluate to `true` when x is 10.

The code fragments `while (x < 10 ? 1 : 0) { x++; }` and `while (x < 11 ? 1 : 0) { x++; }` will not increment x to 10 because they will not compile. These expressions use the conditional operator (`?:`) to return an `int` value, which is not a compatible type for a `while` statement. To be a valid expression in a `while` statement, it must evaluate to a `boolean` value. The conditional operator (`?:`) uses a `boolean` expression but can return a data type other than `boolean` when the expression is `true` or `false`.

Objective:
Controlling Program Flow

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

References:

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

Question #18 of 50

Question ID: 1328150

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

- ✗ **A)** `start`
- ✓ **B)** **`execute`**
- ✓ **C)** **`submit`**
- ✗ **D)** `call`
- ✗ **E)** `run`

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 #19 of 50

Question ID: 1327918

Given:

```
public enum GPA {
  LOW, MID, HIGH, TOP;
  public static void main(String[] args) {
    System.out.println(GPA.MID);
  }
}
```

What is the result?

- ✗ **A)** `1`
- ✓ **B)** `MID`
- ✗ **C)** `2`

✗ **D)** Compilation fails.

✗ **E)** An exception is thrown at runtime.

Explanation

The result of the output is `MID`. The implicit invocation of the `toString` method will output the name of the enumeration constant because `toString` is overridden in the `Enum` class. The return of `toString` is the same as the `name` method.

The result is not the output `1`. This would be the output if the ordinal method were invoked. The ordinal method returns the zero-based index of the enumeration constant.

The result is not the output `2`. This would be the output if the ordinal method were invoked with the constant `GPA.HIGH`, not `GPA.MID`.

The result is not an exception at runtime or failure in compilation. There are no unexpected arguments or invalid syntax in the given code.

Objective:
Java Object-Oriented Approach

Sub-Objective:
Create and use enumerations

References:

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

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

---

Question #20 of 50

Question ID: 1327849

Given:

```
interface Biped {}

class Tail {}

abstract class Animal {}
```

Which class represents an animal that is a biped and has a tail?

✗ **A)**
```
class Kangaroo implements Animal, Biped {
    static Tail t;
}
```

✗ **B)** `class Lemur implements Biped extends Animal, Tail {}`

✓ **C)**
```
class Baboon extends Animal implements Biped {
    Tail t;
}
```

✗ **D)** `class Archosaur extends Biped, Animal, Tail {}`

Explanation

The `Baboon` class represents an animal that is a biped and has a tail as follows:

```
class Baboon extends Animal implements Biped {
   Tail t;
}
```

Generalization, also known as an *is a* relationship, involves subclasses, subinterfaces, or class(es) implementing an interface. `Baboon` subclasses the `Animal` class and then implements the interface `Biped`.

Aggregation, also known as a *has a* relationship, involves a class referencing another instance using a static or instance field. `Baboon` references a `Tail` object using the instance `t` field because the specialized composition *part-whole* relationship is required.

The `Archosaur` class does not represent an animal that is a biped and has a tail. This class represents an animal that is both a biped and a tail in and of itself. This class will not compile because Java does not allow multiple inheritance, and the `Biped` interface cannot be the target of an `extends` clause.

The `Kangaroo` class does not represent an animal that is a biped and has a tail. This class represents an animal that is a biped, but all `Kangaroo` objects will share the same tail because `t` is declared with the `static` keyword. This class will not compile because the `Animal` class cannot be the target of an `implements` clause.

The `Lemur` class does not represent an animal that is a biped and has a tail. This class represents an animal that is both a biped and a tail in and of itself. This class will not compile because Java does not allow the `implements` clause to precede the `extends` clause.

Objective:
Java Object-Oriented Approach

Sub-Objective:
Create and use subclasses and superclasses, including abstract classes

References:

Oracle Technology Network > Developer Tools > JDeveloper > Documentation > White Paper: Getting Started With UML Class Modeling (PDF)

JavaWorld > Core Java > Java Platform APIs > Inheritance versus composition: Which one should you choose? > Page 2 > Comparing composition and inheritance

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

Question #21 of 50

Question ID: 1327759

Given the following:

```
public class Java11 {
   static class RefType {
     int val;
     RefType(int val) {this.val = val;}
   }

   public static void main (String[] args) {
     RefType x = new RefType(1);
     RefType y = x;
     RefType z = y;
     z.val = 10;
     System.out.format("x,y,z: %d,%d,%d", x.val, y.val, z.val);
   }
}
```

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: 10,10,10
```

Variables of reference types hold references to object instances. When y is assigned to x, y now references the same object referenced by x. When z is assigned to y, z now references the same object referenced by y. Thus, changing the `val` field for z will affect x and y, because all three variables reference the same object.

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 the `val` field of `x`, `z` and/or `y` set to `1`, because modifications to the `val` field of `z` will affect the values of `x` and/or `y`. Since the `val` field of `z` is set to `10`, the `val` field of `x` and `y` will also be set to `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 #22 of 50

Question ID: 1328020

Examine the code fragment:

```
public static String getGradeFB (String grade) {
   String comment = null;
   switch (grade) {
     case "A":
     case "B":
       comment = "Excellent job!";
       break;
     case "C":
     case "D":
       comment = "You should try again.";
       break;
     case "F":
       comment = "You should study more.";
     default:
       throw new RuntimeException();
   }
   return comment;
}
```

Which three code statements will throw an exception?

- ✗ **A)** `getGradeFB("B");`
- ✗ **B)** `getGradeFB("A");`
- ✓ **C)** **`getGradeFB(null);`**
- ✓ **D)** **`getGradeFB("d");`**
- ✓ **E)** **`getGradeFB("F");`**
- ✗ **F)** `getGradeFB("C");`

Explanation

The following code statements will throw an exception (line numbers for reference only):

```
1.  getGradeFB("d");
2.  getGradeFB("F");
3.  getGradeFB(null);
```

Code in the default label is executed when there is no match among existing case labels. In this code fragment, the default label throws an exception. Because the `String` comparison in the `switch` statement is case-sensitive, the argument in line 1 will ensure execution reaches the default label. Because there is not a terminating break statement in the case label for the value F, line 2 will fall through the last case label and reach the default label as well.

Line 3 specifies an invalid argument for `String` object used in a `switch` statement. String comparison in `switch` statements is equivalent to the `String.equals` method, so a `null` argument will throw a `NullPointerException`.

The argument in the other three code statements matches case labels that do not throw exceptions.

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

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Language Basics > Summary of Variables

Question #23 of 50

Question ID: 1328192

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

- ✗ **A)** `JapaneseBundle`
- ✗ **B)** `Bundle.jp`
- ✗ **C)** `Bundleproperties`
- ✓ **D)** **`Bundle_jp.properties`**

Explanation

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

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

Objective:
Localization

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

References:

Oracle Technology Network > Java SE > Java Tutorials > Internationalization

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

---

Question #24 of 50

Question ID: 1327783

Which of the following statements is true about the variable `j` referenced in the following lambda statement?

```
for (int j = 0; j < num; j++) {
  new Thread(() -> System.out.println(j)).start();
}
```

- ✓ **A)** **The variable must be effectively final.**
- ✗ **B)** Reference to the variable is copied to the lambda statement.
- ✗ **C)** The variable must be declared using the final keyword.
- ✗ **D)** Value of the variable can be changed within the lambda statement.

Explanation

The variable must be effectively final. A lambda expression's body contains a scope which is the same as a regular nested block of code. Additionally, lambda expressions can access local variables from a scope which are functionally final. This means that these variables must either declared as `final` or are not modified.

You can only reference variables whose values do not change inside a lambda expression. As an example, the following block of code would generate a compile time error:

```
for (int j = 0; j < num; j++) {
  new Thread(() -> System.out.println(j)).start();
}
```

This is because the variable `j` keeps changing and so it cannot be captured by the lambda expression.

The option stating that the variable must be declared using the `final` keyword is incorrect. This is because until a variable is not *effectively* final within the scope of the lambda expression, the code wont compile.

The option stating that the reference to the variable is copied to the lambda statement is incorrect. A variable needs to be effectively final as not reference to the variable is copied.

The option stating that the value of the variable can be changed within the lambda statement is incorrect. The reason for this is that it is illegal to attempt to mutate a captured variable from a lambda expression.

A lambda expression's body contains a scope which is the same as a regular nested block of code. Additionally, lambda expressions can access local variables from a scope which are functionally final. This means that these variables must either declared as `final` or are not modified.

Objective:
Working with Java Data Types

Sub-Objective:
Use local variable type inference, including as lambda parameters

References:

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

Lambda Expressions and Variable Scope

Question #25 of 50

Question ID: 1327954

Which Java 11 module implements a new method called `repeat` that gives you the ability to duplicate values in a `String`?

- ✗ **A)** **`java.compiler`**
- ✓ **B)** `java.base`

✗ **C)** `java.desktop`

✗ **D)** `java.logging`

Explanation

The Java 11 module `java.base` consists of a package called `java.lang`. This package has several methods, and of the many new methods, one of the them, `repeat`, gives you the ability to duplicate values within a `String`.

The return type of this method consists of a `String` and requires a `count` parameter to determine the number of times the string value is duplicated. The method signature is:

```
String repeat(int count)
```

This is an example using the `String.repeat` method:

```
var Mynewstr = "CyberVista";

var MyRepeatObject = Mynewstr.repeat(2);

System.out.println("My String Value is Repeated = " + MyRepeatObject);

System.out.println(" ");

//My String Value is Repeated = CyberVistaCyberVista
```

The `java.compiler` is used for compiling java source code and consists of java compiler APIs. This module consists of several packages like `javax.tools` and `javax.lang.model`.

The `java.desktop` consists of Java APIs that are related to audio, imaging, and printing.

The `java.logging` is a utility module used for Java 2 platforms that allows developers to debug their applications to provide detailed logging options.

Objective:
Java Platform Module System

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

References:

Oracle Documentation > Java SE 11 API > Class String

Question #26 of 50

Question ID: 1328037

Which is true about branching statements in an iterative block?

✗ **A)** An unlabeled `break` statement terminates the outermost block.

- ✗ **B)** A `continue` statement terminates only `for` and `switch` blocks.
- ✗ **C)** **A labeled `break` statement terminates only the innermost block.**
- ✓ **D)** A `return` statement that does not return a value terminates all blocks in the current method.

Explanation

A `return` statement that does not return a value will exit all blocks in the current method. A `return` statement, whether it returns a value or not, exits the current method. By exiting the current method, all iterative blocks are effectively terminated.

An unlabeled `break` statement does not terminate the outermost block. Because no label is specified, the `break` statement will terminate the innermost iterative block.

A labeled `break` statement does not terminate only the innermost block. Because any labeled block can be specified in a `break` statement, an outer block can be terminated.

A `continue` statement does not terminate only `for` and `switch` blocks. A `continue` statement skips the code in the current iteration and continues to the next iteration of the block. A labeled continue statement skips iteration in an outer loop.

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 > Branching Statements

---

Question #27 of 50

Question ID: 1328138

Given the code fragment:

```
IntStream.iterate(0, (n)->n+1)
  .parallel()
  .limit(1000)
  .forEach(System.out::println);
```

Which best describes the result?

- ✗ **A)** The numbers 1 to 1000 in an unpredictable order
- ✗ **B)** The numbers 1 to 1000 in sequential order

✗ **C)** The numbers 0 to 999 in sequential order

✓ **D) The numbers 0 to 999 in an unpredictable order**

Explanation

The result will be the numbers 0 to 999 in an unpredictable order. Below is an excerpt of some possible output:

```
119
120
121
122
123
124
0
1
2
3
4
5
6
7
8
9
10
```

The `iterate` method of a `Stream` defines the first element (0) of the stream and then a `UnaryOperator` that uses that element to determine the next element. In this case, the first number issued by the stream source will be zero, and numbers will be issued that increase by one with each new element. The `limit` method will end the sequence after 1000 items have passed that point in the stream. This stream is parallel, which permits concurrent processing of the elements, but is nevertheless still ordered. Because the stream is parallel, and the iteration is performed using the `forEach` method, the order of processing by the `Consumer` in the `forEach` is not predictable.

The output will not be completely sequential because the stream is parallel. The order of processing by the `Consumer` in the `forEach` is not guaranteed. Thus, the output will include sequential ranges, but the entire output will not be sequential or predictable.

The output will not include 1000 because the stream is ordered. If the stream were unordered, then it would be possible for numbers outside the range 0-999 to be seen.

Objective:
Working with Streams and Lambda expressions

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

References:

The Java Tutorials > Collections > Aggregate Operations > Parallelism

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

---

Question #28 of 50

Question ID: 1327958

Consider the following service interface implementation:

```
package package2;
import package1.SpeakerInterface;

public class SpeakerImplementer implements SpeakerInterface {
    @Override
    public void speak() {
        System.out.println("Speaking from SpeakerImplementer!");
    }
}
```

Which fragment of code will correctly create a service provider module?

✗ **A)**
```
module modPro {
  requires modServ;
  package package2;
}
```

✗ **B)**
```
module modPro {
  requires modServ;
  import package1;
}
```

✗ **C)**
```
module modPro {
  requires modServ;
  import package2.SpeakerImplementer;
}
```

✓ **D)**
```
module modPro {
  requires modServ;
  provides package1.SpeakerInterface with
package2.SpeakerImplementer;
}
```

Explanation

You should use the fragment of code that contains the `provides` and `with` keywords:

```
provides package1.SpeakerInterface with package2.SpeakerImplementer;
```

The other options do not correctly implement the service provider functionality, which requires the use of the `provides` and `with` keywords to the exact service interface to be used.

The `java.util.ServiceLoader` class allows for the use of the Service Provider Interface (SPI), or Service for short, and for implementations of this class, called Service Providers. Java 9 and onwards allows the development of Services and Service Providers as modules. Service modules declare several interfaces whose implementations are provided by provider modules at runtime. Each provider module declares the specific implementations of Service modules that it is providing. Every module that finds and loads Service providers needs a `uses` directive within its declaration.

For any service to be used successfully, each of its service providers is required to be found and then loaded. The `ServiceLoader` class exists for this purpose and performs this function. Any module that is created to find and load service providers requires the `uses` keyword as a directive within its declaration specifying the service interface it uses.

Objective:
Java Platform Module System

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

References:

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

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

---

Question #29 of 50

Question ID: 1328205

You need to create an application that monitors the weather events on Earth.

Which class should you use to track meteorological events?

- ✗ **A)** `TemporalUnit`
- ✓ **B)** `Instant`
- ✗ **C)** `Duration`
- ✗ **D)** **`Period`**

Explanation

You should use the `Instant` class, which represents a single point on the current timeline and is often used for an event timestamp. The `Instant.now()` method creates an object representing single point on the current timeline, based on the system UTC clock.

The `Duration` class is an incorrect option because it represents an amount of time in terms of seconds and nanoseconds. You use the `Duration` class to represent elapsed time between two points of time. The method `Duration.between(inst1,inst2)` creates an object representing the elapsed time between two `Instant` objects `inst1` and `inst2`, in terms of seconds and nanoseconds.

The `TemporalUnit` interface is an incorrect option because it is a Java interface that represents the duration of time between two points of time. A `TemporalUnit` interface allows you to measure time in units. It is implemented by the `ChronoUnit` enumeration.

The `Period` class is an incorrect option because it represents an amount of time in terms of years, months, and days.

Objective:
Localization

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

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Java Date and Time Classes

Oracle Technology Network > Java SE > Java SE Documentation > Java Platform Standard Edition 11 API Specification > java.time > Package java.time

---

Question #30 of 50

Question ID: 1327765

Given the following:

```
int i = 10;
```

Which two expressions evaluate to 3?

- ✓ **A)** `(i + 5) - 6 * (10 / 5)`
- ✗ **B)** `i + (5 - 6) * 10 / 5`
- ✗ **C)** `(i + 5 - 6) * 10 / 5`
- ✗ **D)** `((i + 5 - 6) * 10) / 5`
- ✓ **E)** **`(i + 5) - 6 * 10 / 5`**
- ✗ **F)** `i + (5 - 6 * 10) / 5`

Explanation

The following two expressions evaluate to 3:

```
(i + 5) - 6 * 10 / 5
```

```
(i + 5) - 6 * (10 / 5)
```

In the first expression, `(i + 5) - 6 * 10 / 5` first evaluates `i + 5` as 15 because these operators are inside the parentheses. The next evaluation is `6 * 10` as 60 because multiplicative operators precede additive operators. The next evaluation is `60 / 5` as 12 because multiplicative operators precede additive operators, and this operator is next in the sequence from left to right. The final evaluation is `15 - 12` as 3 because it involves the additive operator.

In the second expression, `(i + 5) - 6 * (10 / 5)` first evaluates `i + 5 as 15` and then evaluates `10 / 5` as 2 because these operators are inside two separate sets of parentheses. The next evaluation is `6 * 2` as 12 because multiplicative operators precede additive operators. The final evaluation is `15 - 12` as 3 because it involves the additive operator.

Knowing operator precedence can help you identify which parts of an expression are evaluated first and which parts will follow. Here is an operator precedence list from highest precedence to lowest precedence:

1. Postfix unary: `num++`, `num--` (value change only occurs *after* overall expression is evaluated)
2. Prefix unary: `++num`, `--num`, `+num`, `-num`, `~ !`
3. Multiply, Divide, Modulus: `* / %`
4. Add, Subtract: `+ -`
5. Shift: `<< >> >>>`
6. Relational: `< > <= >= instanceof`
7. Equality: `== !=`
8. Bitwise AND: `&`
9. Bitwise exclusive OR: `^`
10. Bitwise inclusive OR: `|`
11. Logical AND: `&&`
12. Logical OR: `||`
13. Ternary: `? :`
14. Assignment: `= += -= *= /= %= &= ^= |= <<= >>= >>>=`

Unary operators (`++`, `--`) operate on a variable in the order in which they are placed.

The expression `i + (5 - 6) * 10 / 5` does not evaluate to 3. This expression evaluates to 8. First, `5 - 6` is evaluated as -1 because the operator is inside parentheses. The next evaluation is `-1 * 10` as -10 and then `-10 / 5` as -2 because operators with the same precedence level are evaluated from left to right. The final evaluation is `i + -2`, or `10 - 2` as 8.

The expressions `(i + 5 - 6) * 10 / 5` and `((i + 5 - 6) * 10) / 5` do not evaluate to 3. Both expressions evaluate to 18. In both expressions, `i + 5 - 6` is evaluated as 9. With or without the extra parentheses, the next evaluation is `9 * 10` as 90 because operators with the same precedence level are evaluated left to right. The final evaluation is `90 / 5` as 18.

The expression `i + (5 - 6 * 10) / 5` does not evaluate to 3. This expression evaluates to -1. In the parentheses, `6 * 10` is evaluated as 60 first and then `5 - 60` as -55 because multiplicative operators are evaluated before additive operators. For the same reason, `-55 / 5` is evaluated as -11 and then `i + -11` is evaluated as -1.

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 > Assignment, Arithmetic, and Unary Operators

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

---

Question #31 of 50

Question ID: 1327853

Which statement(s) are true about the `super` keyword? (Choose all that apply.)

- ✗ **A)** The `super` keyword can access all constructors in a superclass or subclass.
- ✗ **B)** **The `super` keyword can access all superclass constructors from a subclass.**
- ✓ **C)** **The `super` keyword can invoke superclass methods in overloaded methods in a subclass.**
- ✗ **D)** The `super` keyword can access all members in a superclass or subclass.
- ✗ **E)** **The `super` keyword can access all members of a superclass from a subclass.**
- ✓ **F)** **The `super` keyword can invoke superclass methods in overridden methods in a subclass.**

Explanation

The following two statements are true about the `super` keyword:

- The `super` keyword can invoke superclass methods in overridden methods in a subclass.
- The `super` keyword can invoke superclass methods in overloaded methods in a subclass.

The `super` keyword can be used to access any visible members, including constructors, in a superclass from one of its subclasses. You use the `super` keyword to invoke the overridden method of a superclass from its subclass:

The `super` keyword cannot access all superclass constructors or all superclass members from a subclass. Only constructors and other members visible to the subclass are available using the `super` keyword. Constructors and other members declared with the `private` keyword remain unavailable.

The `super` keyword cannot access all constructors or members in a subclass. Only accessible constructors and members in a superclass are available to a subclass using the `super` keyword.

Objective:
Java Object-Oriented Approach

Sub-Objective:
Create and use subclasses and superclasses, including abstract classes

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Interfaces and Inheritance > Using the Keyword super

---

Question #32 of 50

Question ID: 1327804

Given the following:

```
public class SmartPhone {
   float screenResolution, width, height;
   public static void main (String[] args) {
     SmartPhone phone;
     phone.height = 112.2f;
     phone.width = 56.8f;
     System.out.format("%.0f dpi - %.1f X %.1f",
     phone.screenResolution, phone.height, phone.width);
   }
}
```

What is the result?

- ✗ **A)** `0 dpi - 112.2 X 56.8`
- ✓ **B)** A compile error is produced.
- ✗ **C)** A runtime error is produced.
- ✗ **D)** `null dpi - 112.2 X 56.8`

Explanation

A compile error is produced because the `phone` variable does not reference an instantiated `SmartPhone` object. Local variables in methods are not provided default values automatically. To access instance members in a class, you must first instantiate the class and access the field using dot notation by referencing the instance.

The result is not `null dpi - 112.2 X 56.8` because the code does not compile. If the `phone` variable did reference an instantiated `SmartPhone` object, then the `screenResolution` field would not be `null`. Instance and `static` fields are provided default values automatically. The default value for the `float` type is 0.0.

The result is not `0 dpi - 112.2 X 56.8` because the code does not compile. If the `phone` variable did reference an instantiated `SmartPhone` object, then this would be the output.

A runtime error is not produced because the code does not compile. If the `phone` variable were set to `null`, then a `NullPointerException` would be thrown.

Objective:
Java Object-Oriented Approach

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

References:

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

---

Question #33 of 50

Question ID: 1327983

Given:

```
String strPath = "C:\\kaplan\\oracle\\java11\\816.java";
Path path = Paths.get(strPath);
```

Which code fragment will retrieve the string *java11*?

- ✗ **A)** `path.subpath(1,2).toString();`
- ✗ **B)** `path.getName(3).toString();`
- ✗ **C)** **`path.getName(4).toString();`**
- ✓ **D)** `path.getName(2).toString();`
- ✗ **E)** `path.subpath(2,4).toString();`
- ✗ **F)** `path.subpath(0,2).toString();`

Explanation

The following code fragment will retrieve the string java11:

```
path.getName(2).toString();
```

The `Path` interface represents a hierarchical structure of folder and/or file elements. The `getName` method returns an element name (folder or file) specified by a zero-based index indicated by path separators, excluding any drive letters. In this case, *java11* is the third element and is located at the index 2.

The code fragment `path.getName(3).toString();` does not retrieve the string *java11* because this specifies the fourth element at index 3. The retrieved string will be *816.java*.

The code fragment `path.getName(4).toString();` does not retrieve the string *java11* because this specifies the fifth element at index 4. Because there is no fifth element in this `Path` object, the application will throw an `IllegalArgumentException`.

The code fragment `path.subpath(0,2).toString();` does not output the string *java11*. The `subpath` method takes a start index (inclusive) and end index (exclusive), and includes the path separator if more than one element is matched. These arguments will match the first and second elements to retrieve the string *cybervista\oracle*.

The code fragment `path.subpath(1,2).toString();` does not output the string *java11*. The `subpath` method takes a start index (inclusive) and end index (exclusive), and includes the path separator if more than one element is matched. These arguments will match only the second element to retrieve the string *oracle*.

The code fragment `path.subpath(2,4).toString();` does not output the string *java11*. The `subpath` method takes a start index (inclusive) and end index (exclusive), and includes the path separator if more than one element is matched. These arguments will match the third and fourth elements to retrieve the string *java11\816.java*.

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 > Java Platform Standard Edition 11 API Specification > java.nio.file > Interface Path

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

---

Question #34 of 50

Question ID: 1328075

Which statements are true concerning the declaration(s) in the `java.util.function.Supplier` interface? (Choose all that apply.)

- ✗ **A)** Declares a method with a `void` return type
- ✗ **B)** Declares a method with a single parameter
- ✗ **C)** Declares a method called `supply`
- ✓ **D) Declares exactly one method**
- ✗ **E)** Declares a method called `produce`

Explanation

The `Supplier` interface is declared with exactly one method named `get`, as follows:

```
public interface Supplier<T> {
  public T get();
}
```

Unlike other functional interfaces, the `Supplier` interface does not declare any default methods. It only declares a single abstract method.

The `Supplier` interface does not define a method named `supply` or `produce`.

The `get` method takes no parameter. It does not take a single parameter.

The `get` method does not have a `void` return type. It returns an object type.

Objective:
Working with Streams and Lambda expressions

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

References:

HYPERLINK "https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/Supplier.html"Java Platform Standard Edition 11 > API > java.util.function > Supplier

Question #35 of 50

Question ID: 1327829

Which statement is true about constructor overloading?

- ✓ **A) A default constructor can be overloaded in a subclass.**
- ✗ **B)** The constructor must use a different name.
- ✗ **C)** The constructor must use the `this` keyword.
- ✗ **D)** A default constructor can be overloaded in the same class.

Explanation

A default constructor can be overloaded in a subclass. If no constructor is defined for a class, then the compiler will automatically provide the default constructor. Because a subclass can define its own constructors without affecting the superclass, a constructor with parameters can be defined that invokes the superclass constructor, implicitly or explicitly.

A default constructor cannot be overloaded in the same class. This is because once a constructor is defined in a class, the compiler will not create the default constructor. Thus, an attempt to overload the default constructor will effectively remove it from the class.

The constructor must not use a different name. In the same class, an overloaded constructor uses the same name. Because subclasses differ in name from their superclass, an overloaded constructor will have a different name.

The constructor does not need to use the `this` keyword. The `this` keyword allows a constructor to reference other constructor methods and/or instance context. Using the `this` keyword is not required in an overloaded constructor.

You can use the `this` keyword in a constructor to call another constructor in the same class. This technique is called explicit constructor invocation.

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 #36 of 50

Question ID: 1328195

Given the following files:

```
MessageTextBundle.properties
MessageTextBundle_fr.properties
MessageTextBundle_fr_CA.properties
MessageTextBundle_de.properties
```

And given the following code fragment:

```
ResourceBundle.getBundle("MessageTextBundle", new Locale("es", "US"));
```

Which properties file will be loaded by the `ResourceBundle`?

- ✗ **A)** `MessageTextBundle_de.properties`
- ✓ **B)** **`MessageTextBundle.properties`**

✗ **C)** `MessageTextBundle_fr.properties`

✗ **D)** `MessageTextBundle_fr_CA.properties`

Explanation

The properties file `MessageTextBundle.properties` will be loaded by the `ResourceBundle`. Because there is not a match for the Spanish language or United States region, the default properties file will be loaded. If either the `MessageTextBundle_es.properties` or `MessageTextBundle_es_US.properties` properties files existed, then one of these files would be loaded instead.

The `MessageTextBundle_fr.properties` properties file will not be loaded. This properties file would be loaded if the locale were set to the French language.

The `MessageTextBundle_fr_CA.properties` properties file will not be loaded. This properties file would be loaded if the locale were set to the French language and Canadian region.

The `MessageTextBundle_de.properties` properties file will not be loaded. This properties file would be loaded if the locale were set to the German language.

Objective:
Localization

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

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Internationalization > Setting the Locale > Isolating Locale-Specific Data > Backing a ResourceBundle with Properties Files

---

Question #37 of 50

Question ID: 1327823

Given:

```
public class CardDeck {
   public CardDeck() {/*Implementation omitted*/}
   public CardDeck (int suits) {/*Implementation omitted*/}
   public CardDeck (int suits, boolean includeJokers) {/*Implementation omitted*/}
}
```

Which constructor is the default constructor?

✗ **A)** `CardDeck()`

✓ **B)** **Not provided.**

✗ **C)** `CardDeck (int)`

✗ **D)** `CardDeck (int, boolean)`

Explanation

The default constructor is not provided. Because the `CardDeck` class contains constructors, no default constructor is provided. If no constructor is defined for a class, then the compiler will automatically provide the default constructor. The default constructor specifies no parameters and invokes the parameterless constructor of the superclass. If any constructor is defined in a class, then the compiler will not provide the default constructor.

The constructor `CardDeck()` is not the default constructor. Although this constructor specifies no parameters, the default constructor is not user-defined, but provided by the compiler.

The constructors `CardDeck (int)` and `CardDeck (int, boolean)` are not default constructors. The default constructor specifies no arguments, contains no body, and is provided automatically by the compiler.

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 #38 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) `void BasicMethod() {}`**

✗ **B)** `import java.text.*;`

✗ **C)** `package basicPackage;`

✗ **D)** `module basicModule {}`

✓ **E) `static final int VAL=1000;`**

✓ **F) `enum ClassType {basic, advanced}`**

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 #39 of 50

Question ID: 1327920

Given:

```
public enum Rank implements Cloneable {
  ACE(1),DEUCE(2),THREE(3),FOUR(4),FIVE(5),SIX(6),SEVEN(7),EIGHT(8),
  NINE(9),TEN(10),JACK(10), QUEEN(10), KING(10);
  public int value;
  private Rank(int value) {
    this.value = value;
  }
08    public static void main (String[] args) {
09      Rank card1 = Rank.ACE;
10      Rank card2 = Rank.KING;
11      System.out.println(card1.compareTo(card2));
12    }
13  }
```

What is the result?

- ✗ **A)** Compilation fails due to an error on line 05.
- ✗ **B)** `12`
- ✗ **C)** Compilation fails due to an error on line 01.
- ✗ **D)** Compilation fails due to an error on line 08.
- ✗ **E) Compilation fails due to an error on line 11.**
- ✓ **F)** `-12`

Explanation

The result is the output `-12`. This is because the `ACE` constant is less than the `KING` constant and is located 12 positions back based upon the order in which they are declared in the `Rank` enumeration. All enumerations implicitly implement the `Comparable` interface.

The result is not the output `12`. This would be the output if the `KING` constant were compared to `ACE`, because it is greater based on the declaration order of the constants in `Rank`.

Compilation does not fail due to an error on line 01. Enumerations can implement interfaces.

Compilation does not fail due to an error on line 05. Enumerations can have private constructors.

Compilation does not fail due to an error on line 08. Enumerations can have public members.

Compilation does not fail due to an error on line 11. Enumerations implicitly implement the `Comparable` interface because the `Enum` class from which they extend implements the `Comparable` and `Serializable` interfaces.

Objective:
Java Object-Oriented Approach

Sub-Objective:
Create and use enumerations

References:

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

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

Question #40 of 50

Question ID: 1328045

Given the following code:

```
public String messWithString(String str)

  throws StringIndexOutOfBoundsException {
    //implementation omitted
}
```

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

- ✗ **A)** Catch `StringIndexOutOfBoundsException`.
- ✗ **B)** Invoke the method within a `try` block without an associated `catch` or `finally` block.
- ✓ **C)** **No handling action is required.**
- ✗ **D)** Specify `StringIndexOutOfBoundsException`.

Explanation

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

Neither caching nor specifying `StringIndexOutOfBoundsException` is required for the code to compile. Runtime exceptions are abnormal conditions internal to the application and often are unrecoverable. The compiler does not check catching and specifying `RuntimeException` and its subclasses.

Invoking the method within a `try` block without an associated `catch` or `finally` block is not required for the code to compile. A `try` block is required to have an associated `catch` or `finally` block. Declaring a `try` block without an associated `catch` or `finally` block is a syntax error and will cause the code to fail compilation.

Objective:
Exception Handling

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

References:

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

Question #41 of 50

Question ID: 1327966

Which statement is true about standard streams?

- ✓ **A)** The Java platform supports three standard streams.
- ✗ **B)** `System.err` is the only stream available for console output.
- ✗ **C)** **All standard streams are character streams.**
- ✗ **D)** The standard output and error streams cannot be diverted from the console.

Explanation

The Java platform supports three standard streams. The standard input stream is available through the built-in `System.in` field, while standard output and error streams are available through the `System.out` and `System.err` fields, respectively.

All standard streams are not character streams. Standard streams are byte streams.

`System.err` is not the only stream available for console output. `System.out` is also available for console output.

The standard output and error streams can be diverted from the console. The standard error stream is often diverted to a file for logging purposes. By default, standard output and error streams display in the console, but both can be diverted to other destinations. The standard input stream can also be redirected to other sources. Once a standard stream is redirected, the console will no longer accept input or display output from the console.

Objective:
Java File I/O

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

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Essential Classes > Basic I/O > I/O from the Command Line

---

Question #42 of 50

Question ID: 1327927

Which code statement correctly instantiates and assigns a generic collection limited to `ByteBuffer` objects and subtypes?

- ✗ **A)** `List<?> bbList = new ArrayList<>();`
- ✗ **B)** `List<Buffer> bbList = new ArrayList<ByteBuffer>();`
- ✓ **C)** **`List<? extends ByteBuffer> bbList = new ArrayList<>();`**
- ✗ **D)** `List<? super Buffer> bbList = new ArrayList<>();`

Explanation

The following code statement correctly instantiates and assigns a generic collection limited to `ByteBuffer` objects and subtypes:

```
List<? extends ByteBuffer> bbList = new ArrayList<>();
```

This code uses the wildcard character (`?`) with the `extends` keyword to specify an *upper bound* for allowable data types. Only those data types that extend or match `ByteBuffer` are allowed in the collection.

The code statement should not specify a wildcard in the variable declaration without a bound or matching type parameter in the instantiation. Because no type is captured, the collection will not support any data type.

The code statement should not use the wildcard character (`?`) with the `super` keyword. This will result is a *lower bound*, where any super class of `Buffer` is allowed including `Object`, but not the `ByteBuffer` type.

The code statement should not specify unmatching type parameters in the declaration and instantiation. The type parameter `ByteBuffer` in the `ArrayList` instance does not match `Buffer` in the `bbList` variable declaration. Type parameters must match if not inferred.

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 > Wildcards

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Generics > Type Inference

Question #43 of 50

Question ID: 1328160

Which two methods of the `ConcurrentMap` interface will add an entry for a key only if that key already exists in the collection?

- ✗ **A)** **`put(K key, V value)`**
- ✗ **B)** `remove(Object key, Object value)`
- ✗ **C)** `remove(Object key)`
- ✗ **D)** `putIfAbsent(K key, V value)`
- ✓ **E)** **`replace(K key, V value)`**

✓ **F)** `replace(K key, V oldValue, V newValue)`

Explanation

The `replace(K key, V value)` and `replace(K key, V oldValue, V newValue)` methods will add an entry for a key only if that key already exists in the collection. The `replace(K key, V value)` method replaces a value only if the key exists, while the `replace(K key, V oldValue, V newValue)` method replaces a value only if the key exists and matches the specified value. `ConcurrentMap` is the thread-safe subinterface of the `Map` interface.

The `put(K key, V value)` method adds a key/value pair without checking to see if the key already exists.

The `putIfAbsent(K key, V value)` method is a thread-safe method that only adds a key/value pair if the key does not already exist.

The `remove(Object key)` method is an overloaded method that removes an entry for a key, but does not check whether that key contains a value.

The `remove(Object key, Object value)` method is an overloaded remove method that removes an entry for a key, but only if that key contains a value.

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 > Java Platform Standard Edition 11 API Specification > java.util.concurrent > Interface ConcurrentMap<K,V>

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Essential Classes > Concurrency > Concurrent Collections

---

Question #44 of 50

Question ID: 1327892

Given:

```
public class Walker {
  protected double elevation;
  protected int distance;
  int move (int data) {
    return distance += data;
  }
  protected static void getWalkerInfo(Walker w) {
    System.out.println(w.distance);
  }
}

public class UphillHiker extends Walker {
  public double elevation;
  public int move (int data) {
    elevation += .5;
    return distance += data;
  }
  public static void getWalkerInfo(Walker w) {
    System.out.println(w.distance + "-" + w.elevation);
  }
}
```

Which code fragments will output `10-0.0`? (Choose all that apply.)

✗ **A)**
```
Walker w = new Walker();
w.move(10);
w.getWalkerInfo(w);
```

✓ **B)**
```
Walker w = new Walker();
w.move(10);
UphillHiker.getWalkerInfo(w);
```

✗ **C)**
```
Walker w = new UphillHiker();
w.move(10);
w.getWalkerInfo(w);
```

✓ **D)**
```
UphillHiker w = new UphillHiker();
w.move(10);
w.getWalkerInfo(w);
```

Explanation

The following code fragments will output `10-0.0`:

```
UphillHiker w = new UphillHiker();
w.move(10);
w.getWalkerInfo(w);
```

```
Walker w = new Walker();
w.move(10);
UphillHiker.getWalkerInfo(w);
```

The first code fragment instantiates the `UphillHiker` class, while the second code fragment instantiates the `Walker` class. Because of polymorphism, the first code fragment invokes the overridden move method in

`UphillHiker`, and the second code fragment invokes the original move method in `Walker`. Unlike the `move` method in `Walker`, the `move` method in `UphillHiker` increments the `elevation` field. The `getWalkerInfo` method is a class member that is invoked on the `UphillHiker` class in both code fragments.

Polymorphism determines which overridden method is used based on the object type. Member hiding does not use polymorphism and instead relies on the reference type. The `elevation` field is hidden by the `UphillHiker` class, so that when the reference type is `Walker`, its value for `elevation` is in the output, not the value of the same field in `UphillHiker`.

The following code fragments will output `10`, not `10-0.0`:

```
Walker w = new UphillHiker();
w.move(10);
w.getWalkerInfo(w);
```

```
Walker w = new Walker();
w.move(10);
w.getWalkerInfo(w);
```

This output is `10` because the `getWalkerInfo` method is hidden. Member hiding does not use polymorphism and instead relies on the reference type. In both code fragments, the reference type is `Walker`.

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 > Hiding Fields

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

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

---

Question #45 of 50

Question ID: 1328087

Given:

```
Arrays.asList("Fred", "Jim", "Sheila")
  .stream()
  .peek(System.out::println) // line n1
  .allMatch(s->s.startsWith("F"));
```

What is the result?

✓ **A)** **`Fred`**
`Jim`

✗ **B)** Compilation and execution complete normally, but no output is generated.

✗ **C)** `Fred`

✗ **D)** Compilation fails at line `n1`.

✗ **E)** `Fred`
`Jim`
`Sheila`

Explanation

The code shown compiles and executes successfully with the following output:

```
Fred
Jim
```

The `peek` method invokes the consumer's behavior with each object that passes downstream. In this example, the method `allMatch` draws items down the stream until either the stream is exhausted or an item is found that does not match. In this example, the item "Jim" does not match, and at that point--after both Fred and Jim have been printed--the `allMatch` method returns the value false and stream processing completes.

The other options are incorrect because they do not describe the behavior of the 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

The Java Tutorials > Collections > Lesson: Aggregate Operations

Question #46 of 50

Question ID: 1327936

Given the following code fragment:

```
public class TestArrayList {
   public static void main (String[] args) {
     ArrayList<String> names = new ArrayList<>(2);
     names.add("Amy");
     names.add("Anne");
     names.add("Jason");
     System.out.println(names.get(3));
   }
}
```

What is the result?

- ✓ **A)** Code throws a runtime exception.
- ✗ **B)** `Anne`
- ✗ **C)** Code compilation fails.
- ✗ **D)** `Jason`

Explanation

The code fragment will throw a runtime exception. An `IndexOutOfBoundsException` will be thrown because the index 3 is beyond the last element of the `ArrayList` object.

The code fragment will not fail compilation. There are no syntax errors in the code fragment.

The code fragment will not result in the output `Anne` or `Jason`. The runtime exception prevents any output.

Objective:
Working with Arrays and Collections

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

References:

Oracle Documentation > Java SE 11 API > Class ArrayList<E>

Question #47 of 50

Question ID: 1327828

Given:

```
public class Pedometer {
   private String units;
   private double stride;
   public Pedometer(String units) {
    this.units = units;
  }
}
```

Which code fragment correctly overloads the constructor?

✗ **A)**
```
public Pedometer (String units, double stride) {
   super("inches");
   super.stride = 25;
}
```

✗ **B)**
```
public static Pedometer init(String units, double stride) {
   Pedometer ped = new Pedometer(units);
   ped.stride = stride;
   return ped;
}
```

✗ **C)**
```
public static Pedometer init() {
   Pedometer ped = new Pedometer("inches");
   ped.stride = 25;
   return ped;
}
```

✓ **D)**
```
public Pedometer (double stride) {
   this("inches");
   this.stride = 25;
}
```

Explanation

The following code fragment correctly overloads the constructor:

```
public Pedometer (double stride) {
   this("inches");
   this.stride = 25;
}
```

An overloaded constructor includes different parameters than other constructor methods. As in this code segment, constructors often use the `this` keyword to reference other constructors and the current instance context.

The code fragment that uses the `super` keyword does not correctly overload the constructor. Although the method declaration does overload the constructor, its implementation will fail compilation. The `super` keyword is used to reference the superclass, not other class constructors and the current instance.

The code fragments that define the `init` method do not correctly overload the constructor. Although these factory methods allow an alternative to object creation with a constructor, they are not constructor methods. An overloaded constructor is a constructor method that includes different parameters than other constructor methods, not a factory method.

Objective:
Java Object-Oriented Approach

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

References:

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

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

---

Question #48 of 50

Question ID: 1328099

Given the code fragment:

```
String [] rainbow = {
  "Red", "Orange", "Yellow", "Green",
  "Blue", "Indigo", "Violet"
};
/* insert here */
  .forEach(System.out::println);
```

Which code, when added at `/* insert here */`, will result in the output `Yellow, Green, and Blue`?

✓ **A) `Arrays.stream(rainbow, 2, 5)`**

✗ **B)** `Arrays.stream(2, 3, rainbow)`

✗ **C)** `Arrays.stream(rainbow, 2, 4)`

✗ **D)** `Arrays.stream(rainbow, 2, 3)`

Explanation

The correct code is `Arrays.stream(rainbow, 2, 5).`

There are several overloaded `Arrays.stream` methods. Generally, they use the contents of an array as the items that will be sent down the stream. The type of the stream is determined by the type of the array.

One overload of `Arrays.stream` takes the array first and uses the second and third arguments to select a sub-range of the array contents. The second argument specifies the first array element to be used, and the third argument defines the first array element ***not*** to use. Given this, if the first argument is the reference to the array of color names, then the second argument necessary to generate the correct values in the stream will be 2, which is the index of “Yellow”. To ensure that the last element to be seen in the stream is "Blue", the third argument must be the index of Indigo, or 5.

`Arrays.stream(rainbow, 2, 4)` will only output `Yellow and Green`, since it will stop before index 4.

`Arrays.stream(rainbow, 2, 3)` will only output the single color `Yellow`. Notice that the second argument is not the count of elements to send to the stream, but the index of the first element of the array ***not*** to include in the stream.

`Arrays.stream(2, 3, rainbow)` will fail to compile. There is no overload of `stream` that takes numbers for the first two arguments and an array as the third.

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 > Arrays

Question #49 of 50

Question ID: 1327798

Given:

```
public class Computer {
  private Computer() {}
  class CPU {
    private void performWork() {System.out.println("CPU working!");}
  }
  public static void main(String[] args) {
    //Insert code here
  }
}
```

Which statement, inserted in the `main` method, enables the code to compile?

- ✗ **A)** `CPU.performWork();`
- ✗ **B)** `(new Computer()).(new CPU()).performWork();`
- ✗ **C)** `Computer.CPU.performWork();`
- ✗ **D)** `new Computer().CPU().performWork();`
- ✗ **E)** `new Computer.CPU().performWork();`

✓ **F)** `new Computer().new CPU().performWork();`

Explanation

The statement `new Computer().new CPU().performWork();`, when inserted in the `main` method, enables the code to compile. This statement first instantiates the outer class `Computer`, then instantiates the inner class `CPU`, and finally invokes the `performWork` method. The `performWork` method is available to the `main` method because private members in an inner class are available not only in the inner class, but also to the outer class. Access modifiers in inner classes only affect access beyond the outer class and do not affect inner class visibility to its outer class.

The statements `CPU.performWork();` or `Computer.CPU.performWork();`, inserted in the `main` method, do not enable the code to compile. This is because the `main` method is a static member and `CPU` is not a static inner class.

The statement `new Computer.CPU().performWork();`, inserted in the `main` method, does not enable the code to compile. This is because `CPU` is not a static inner class.

The statements `new Computer().CPU().performWork(); or (new Computer()).(new CPU()).performWork();`, inserted in the main method, do not enable the code to compile. This is because these statements use invalid syntax.

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 > Nested Classes

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

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

---

Question #50 of 50

Question ID: 1327962

Consider the following code output:

```
jar --describe-module --file=provider.jar
```

```
modProv jar:file:///D:/lord-java/provider.jar/!module-info.class
requires java.base mandated
requires modServ
provides package1.SpeakerInterface with package2.SpeakerImplementation
contains package2
```

Based on this output, what does the provider module depend on?

- ✗ **A)** `provider.jar`
- ✗ **B)** `package1`
- ✓ **C)** `java.base`
- ✗ **D)** `package2`

Explanation

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

The other options are incorrect because the code output specifies mandatory modules via the keyword `requires`. Neither `package1`, `package2` or `provider.jar` have *requires* preceding them in the code output.

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

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

Objective:
Java Platform Module System

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

References:

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

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