Quick Quiz September 9, 2022

Test ID: 223107425

Question #1 of 50

Question ID: 1328196

Given a file with the following contents:

```
#GUILabels_fr.properties file
stringOkButton = OK
stringCancelButton = Annuler
stringIgnoreChangesMessage = Ignorer les modifications?
```

Which code fragment will retrieve the `ResourceBundle` associated with these property values?

✗ **A)** `ResourceBundle.getBundle("GUILabels", "FR");`

✓ **B)** `ResourceBundle.getBundle("GUILabels", Locale.FRENCH);`

✗ **C)** **`ResourceBundle.getBundle("GUILabels_fr.properties");`**

✗ **D)** `ResourceBundle.getBundle("GUILabels_fr.properties", "FR");`

Explanation

To retrieve the `ResourceBundle` associated with these property values, you should use the following code fragment:

```
ResourceBundle.getBundle("GUILabels", Locale.FRENCH);
```

A properties file is a text file in the `CLASSPATH` that contains name/value pairs for a specified resource bundle. The properties filename matches the name of the resource bundle, includes the language identifier, and ends with the .properties extension. To load a `ResourceBundle` object from a properties file, invoke the `getBundle` method, specifying the resource bundle name and the locale associated with the language or region. If a derived `ResourceBundle` class is not found, then the `PropertyResourceBundle` object will be loaded using the values in the properties file.

The code fragments that specify the full filename are incorrect. Only the resource bundle name should be specified in the `getBundle` method.

The code fragments that specify the string *FR* are incorrect because the `getBundle` method requires a `Locale` object, not a string representing a language or region.

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 #2 of 50

Question ID: 1328141

Given:

```
Stream.of(new Student("Fred"), new Student("Jim"))
  .reduce(/*expression here*/)
```

Which statement(s) are true of the expression that must be provided at the point `/*expression here*/`? (Choose all that apply.)

- ✓ **A)** The behavior must be associative.
- ✓ **B)** **The expression must implement `BinaryOperator<Student>`.**
- ✗ **C)** **The behavior can safely modify the `Student` objects.**
- ✗ **D)** The behavior is expected to mutate data.

Explanation

The expression must implement `BinaryOperator<Student>` and its behavior must be associated. The `reduce` operation requires that the provided argument implements `BinaryOperator<T>`, where *T* is the data type of the stream. The behavior provided in the implementation of the `BinaryOperator<T>` must adhere to certain constraints. One of these is that the behavior must be associative. This means that for an operation called *op(a,b)* given the values *A*,*B*,*C*, then *op(A, op(B,C))* must be equal to *op(B, op(A,C))*. In other words, the order in which the operation is applied must not affect the correctness of the result.

The behavior is not expected to mutate data. Accessing mutable state should be avoided because it forces undesirable behavior, either non-deterministic memory behavior, or non-scalable thread synchronization.

The behavior should not modify the Student objects. Side effects, such as modifying data, whether in the stream or elsewhere, should be avoided.

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 #3 of 50

Question ID: 1327906

You define the following interface to handle photographic items:

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

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

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

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

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

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

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

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

Explanation

You should declare the `makePhotographs` method as follows:

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

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

You should not use the declaration `List<Photograph> makePhotographs(Scene ... scenes)`. Any method declared in an interface that is neither static nor default will be treated as an abstract method and must not have a

body. This method declaration will cause the code to fail compilation because it attempts to declare an abstract-by-default method, and yet tries to provide a method body.

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

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

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

Objective:
Java Object-Oriented Approach

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

References:

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

---

Question #4 of 50

Question ID: 1327787

Which statement is true about the object life cycle?

- ✓ **A)** The instantiation must use the `new` keyword.
- ✗ **B)** **The initialization must specify the object variable name.**
- ✗ **C)** The programmer must explicitly destroy objects.
- ✗ **D)** The declaration must call the class constructor.

Explanation

The instantiation must use the `new` keyword. The `new` keyword creates the object:

```
Car sedan = new Car();
```

The declaration does not call the constructor. The declaration associates a variable name with its object type.

The initialization does not specify the object variable name. The initialization calls the constructor so that instance fields are set to their default values.

The programmer does not explicitly destroy objects. The garbage collector periodically releases memory of unreferenced objects automatically.

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 > Creating Objects

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

---

Question #5 of 50

Question ID: 1327791

Given the following:

```
public class Java11 {
   static int modify (int[] i) {
     i[0] += 10;
     return i[0] + 10;
   }
   public static void main(String[] args) {
     int[] i = {10};
     //insert code here
   }
}
```

Which statement(s) should be inserted in the code to output `35`?

✗ **A)** `modify(i); System.out.println(i[0]);`

✓ **B)** **`modify(i); System.out.println(i[0] + 15);`**

✗ **C)** `System.out.println(modify(i));`

✗ **D)** `System.out.println(modify(i)+ 15);`

Explanation

The statements `modify(i); System.out.println(i[0] + 15);` should be inserted in the code to output `35`. Initially, the value for the first element of the `i` array is set to 10. Because an array is an object, the object reference

is passed as an argument to the `modify` method. Any modifications to the array will persist after the method returns. In the `modify` method, the first element is incremented by 10 so that it is now 20. In the `println` method, 15 is added to the first element so that the output is 20 + 15, or `35`.

The statement `System.out.println(modify(i));` should not be inserted in the code to output `35`. The output is `30` because the first element is incremented by 10 and then the return value is the first element value (20) plus 10.

The statement `System.out.println(modify(i)+ 15);` should not be inserted in the code to output `35`. The output is `45` because the first element is incremented by 10, then the return value is the first element value (20) plus 10 and finally 15 is added to the return value.

The statements `modify(i); System.out.println(i[0]);` should not be inserted in the code to output `35`. The output is `20` because the first element of the `i` array is set to 10 in the `main` method and then incremented by 10, so that its value is now 20.

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 > Language Basics > Classes and Objects > Passing Information to a Method or a Constructor

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

---

Question #6 of 50

Question ID: 1327909

Consider the following code:

```
public interface shape {
    default void square(String note) {
        shapeShow(note, "THIS IS A SQUARE");
    }
    default void circle(String note) {
        shapeShow(note, "THIS IS A CIRCLE");
    }
    default void hexagon(String note) {
        shapeShow(note, "THIS IS A HEXAGON");
    }
```

```
    default void oval(String note) {
        shapeShow(note, "THIS IS AN OVAL");
    }
    private abstract void shapeShow(String note, String shapeName) {
        // The implementation will contain the actual code
    }
}
```

Which line of code will cause a compiler error?

✓ **A)** `private abstract void shapeShow(String note, String shapeName) {`

✗ **B)** `default void oval(String note) {`

✗ **C)** `public interface shape {`

✗ **D)** `shapeShow(note, "THIS IS AN OVAL");`

Explanation

Using the `private` and `abstract` keywords together will cause a compiler error because `private` and `abstract` both have separate uses in Java interfaces. When a method inside an interface is deemed `private`, it indicates that the method is accessible only *inside* the interface and cannot be accessed from this interface by other interfaces or classes. The purpose of this is to expose only the method implementations that are intended to be exposed. However, when a method is deemed `abstract`, then it needs to be inherited and overridden by subclasses.

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

Using `default void` with a method is a valid return type in Java and will not cause a compiler error. Default methods allow interfaces to have methods that have implementations, but do not affect the implementations done by the classes that have inherited the interface.

Using `public interface` will not result in a compiler error as it is a valid format for creating a Java interface. Top-level interfaces are expected to have `public` access by default. Marking an interface as `final`, `protected` or `private` will result in a compilation error.

The `shapeShow()` method does not violate any compilation requirements and will not cause an error because it is marked as `private` and `abstract`, which will compile successfully. Additionally, method declarations inside interfaces with a body represented by a semicolon are `public` and `abstract` implicitly.

Objective:
Java Object-Oriented Approach

Sub-Objective:

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

References:

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

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

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

---

Question #7 of 50

Question ID: 1327801

Given:

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

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

Which two statements are true?

- ✓ **A)** If a factory that returns instances of `Accessor` is provided in the `Insider` class, that factory would be an instance method.
- ✓ **B)** **The `Accessor` class may correctly be marked `private`.**
- ✗ **C)** To allow access to `mainData` in the `Accessor` class, the `mainData` field must be marked `final`.
- ✗ **D)** **The `Accessor` class may correctly be marked `static`.**

Explanation

The following statements are true:

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

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

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

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

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

Objective:
Java Object-Oriented Approach

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

References:

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

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

---

Question #8 of 50

Question ID: 1327821

Given:

```
public class SuperEmptyClass {
   public SuperEmptyClass (Object obj) { /*Implementation omitted*/ }
}

public class EmptyClass extends SuperEmptyClass { }
```

Which constructor is provided to `EmptyClass` by the compiler?

✗ **A)** `public EmptyClass(Object obj) {}`

✓ **B)** `public EmptyClass() { super();}`

✗ **C)** **`public EmptyClass(Object obj) { super(obj);}`**

✗ **D)** `public EmptyClass() {}`

Explanation

The following constructor is provided to the `EmptyClass` by the compiler:

```
public EmptyClass() {super();}
```

This constructor is the default constructor. 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. In this scenario, the code will fail compilation because `SuperEmptyClass` does not provide a parameterless constructor.

The following constructor is not provided to EmptyClass by the compiler:

```
public EmptyClass() { }
```

The default constructor is not empty. The default constructor invokes the parameterless constructor of the superclass.

The constructors that specify an `Object` parameter are not provided to `EmptyClass`. The default constructor specifies no parameters.

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 #9 of 50

Question ID: 1327889

Given:

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

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

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

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

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

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

Explanation

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

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

Objective:
Java Object-Oriented Approach

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

References:

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

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

---

Question #10 of 50

Question ID: 1328118

Given the code fragment:

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

What is the result?

- ✗ **A)** `false`
- ✗ **B)** No output
- ✗ **C)** `ABcDe`
- ✓ **D)** **`true`**
- ✗ **E)** `Optional[true]`

Explanation

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

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

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

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

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

Objective:
Working with Streams and Lambda expressions

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

References:

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

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

Question #11 of 50

Question ID: 1328180

You have written a banking application in Java which will allow users from all over the world to enter their bank account information. The following code is an excerpt:

```
public List<BankAccount> FindAccountsByUserId(String userId) throws SQLException {
  String query = "select " + "user_id,acc_num,branch_id,total_balance "
   + "from Accounts where user_id = '" + userId + "'";
  Connection con = dataSource.getConnection();
  ResultSet res = con.createStatement().executeQuery(query);
}
```

Which of the following should you use in this code to safeguard it from injection attacks?

- ✗ **A)** ad-hoc SQL query
- ✓ **B)** **`prepareStatement()`**
- ✗ **C)** `javax.script`
- ✗ **D)** `Double.isInfinite()`

Explanation

You should use the `prepareStatement()` method. This lets you place user-entered values into an SQL query before it is executed. As it appears in the example, the code uses an SQL query that contains a user's ID without any validation. If the user ID contained a value from an untrusted source, it could change the intended behavior of the application. The following adjusted code is safe from such injection attacks:

```
Connection con = dataSource.getConnection();
PreparedStatement ps = con.prepareStatement(sql);
```

```
ps.setString(1, userId);
ResultSet res = ps.executeQuery(sql));
```

Using the `Double.isInfinite()` method is an incorrect option. This method helps avoid exceptional floating point values. which can be used in a buffer overflow attack.

Ad-hoc queries implement a dynamic SQL query and should be avoided in favor of parameterized queries.

`javax.script` is an incorrect option because untrusted code can be executed by it.

Injection and inclusion attacks involve an unintentional change of control by applications that utilize text formatting and data interpretation for this purpose. Here are some of the ways these attacks can happen:

- Incorrect formatting outputs - These attacks involve the use of special characters in input strings or partially removing them. A means to avoid this attack is to perform data validation on the input string before parsing it. Additionally, you should use a legitimate tested library for formatting or generating XML and not an untrusted source.
- Dynamic SQL - For sake of security, dynamic SQL should be avoided altogether. These are vulnerable to command injection. This is usually by inserting a quote symbol (`'`) right after an SQL statement. You should use `java.sql.PreparedStatement` instead of using `java.sql.Statement`. The following code illustrates the use of secured SQL statements:

```
String query = "SELECT * FROM Emp WHERE empId = ?";
PreparedStatement prepstmt = con.prepareStatement(query);
prepstmt.setString(1, empId);
ResultSet res = prepStmt.executeQuery();
```

- HTML and XML generation - Data that could be invalid needs to be correctly sanitized before being included in the outputs for XML and HTML. This can help avoid Cross Site Scripting (XSS) and XML injection. This holds especially true when using Java Server Pages (JSP). One way to sanitize data is to encode it. The most feasible option is to allow safe characters and filter and encode only characters that are known to be troublesome. It’s preferred to use a trusted library to perform the task of sanitization and encoding. For example, the Apache Commons Lang library provides functions that help with escaping content. This allows user input to be rendered as pure text instead of HTML content.
- Invalid command line data - Placing invalid data on the command line can cause processes to interpret arguments as either several arguments or invalid options. For this reason, it is recommended to encode arguments or pass it to the process through a trusted channel or temporary files.
- XML inclusion - These include XML Eternal Entity (XXE) attacks that involve attackers placing local files into the data of the XML file which may then be illegally accessed. The best way to safeguard against these types of attacks is to lower the privilege and use the most restricted configuration for the XML parser.
- BMP files - BMP files can contain references to International Color Consortium (ICC) files, and the attacker may cause a forced read of these files. It’s best to avoid BMP files or restrict their privileges.
- HTML display in Swing - Various Swing components can interpret any text with an `<html>` header as actual HTML, which can include malicious code. One means to avoid this is to set the “`html.disable`” property in the client to `Boolean.TRUE`.

- Untrusted code - Untrusted code can be concealed anywhere and various APIs and components can run such code. Some of these APIs and components are `javax.script`, LiveConnect interfaces with JavaScript, the XSLT interpreter, and long-term persistence of JavaBeans, among others.
- Floating point values - You need to introduce certain sanitization code to avoid injection of exceptional floating point values like NaN (Not a number) or infinite values. To sanitize code you can use methods like `isNan` or `isInfinite` with the `Double` and `Float` classes.

Objective:
Secure Coding in Java SE Application

Sub-Objective:
Develop code that mitigates security threats such as denial of service, code injection, input validation and ensure data integrity

References:

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

Question #12 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)** `2`

✗ **B)** Compilation fails.

✗ **C)** `1`

✓ **D)** `MID`

✗ **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 #13 of 50

Question ID: 1328167

Which is an advantage of `Lock` objects over implicit locks in synchronized blocks?

- ✓ **A) `Lock` objects can stop from acquiring a lock if it is unavailable or a timeout elapses.**
- ✗ **B)** `Lock` objects support the wait/notify mechanism.
- ✗ **C)** `Lock` objects are simpler to implement than synchronized blocks.
- ✗ **D)** Only one thread can own a `Lock` object at the same time.

Explanation

Unlike implicit locks in synchronized blocks, `Lock` objects can stop from acquiring a lock if it is unavailable or a timeout elapses. The `tryLock` method stops from acquiring a lock if it is unavailable or a timeout elapses, while the `lockInterruptibly` method stops from acquiring a lock if another thread sends an interruption.

`Lock` objects are not simpler to implement than synchronized blocks. synchronized blocks use implicit locking, while `Lock` objects require explicit code.

For both `Lock` objects and synchronized blocks, only one thread can own a `Lock` object at a time.

Both `Lock` objects and synchronized blocks support the wait/notify mechanism. The wait/notify mechanism is provided by associated `Condition` objects, so that threads can be suspended until a notification signal is received.

Objective:
Concurrency

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

References:

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

---

Question #14 of 50

Question ID: 1328206

You need to create a yearly automated maintenance operation on your system for the next 5 years. You write the following code:

```
01  public class YearlyMaintenance {
02    public static void main(String[] args) {
03      LocalDate begin = LocalDate.of(2020, Month.AUG, 28);
04      LocalDate conclude=LocalDate.of(2025, Month.AUG, 28);
05      //Insert code here
06      doMaintenance(begin, conclude, timePeriod);
07    }
08  }
```

Which code statement should you add at line `05`?

✗ **A)** `Date timePeriod = Date.getYear(1);`

✗ **B)** `Instant timePeriod = Instant.plus(5, ChronoUnit.YEARS);`

✓ **C)** **`Period timePeriod = Period.ofYears(1);`**

✗ **D)** `Duration timePeriod = Duration.ofYears(5);`

Explanation

You should use the `Period` class at line `05` as follows:

```
Period timePeriod = Period.ofYears(1);
```

The following option is incorrect:

```
Date timePeriod = Date.getYear(1);
```

This is because the `getYear()` method is deprecated and does not take an argument.

The following option is incorrect:

```
Instant timePeriod = Instant.plus(5, ChronoUnit.YEARS);
```

This is because `timePeriod` will now store an `Instant`, which does not represent a period of time. It will instead represent an instant five years from now.

The following option is incorrect because the `Duration` class does not have the `ofYears` method:

```
Duration timePeriod = Duration.ofYears(5);
```

The `Duration` class measures time in days, not years, using methods like `ofDays`, `ofHours`, `ofMinutes`, `ofSeconds`, `ofMillis`, and so on.

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 #15 of 50

Question ID: 1328027

Given the code fragment:

```
int i = 0;
for (;;) {
   i++;
}
System.out.println(i);
```

What is the result?

✗ **A)** `2147483647`

✗ **B)** `-2147483648`

✓ **C) Code compilation fails.**

✗ **D)** Code throws a runtime exception.

✗ **E)** `0`

Explanation

The result is the code compilation fails. If no expressions are specified in the `for` statement, then an infinite loop is created. The compilation error occurs on the line after the `for` block because any line after the block is unreachable.

The code does not throw a runtime exception because the code does not compile.

The result will not be the output `0`, `2147483647`, or `-2147483648`. The code will not compile because of the unreachable statement. The range for `int` is from -2,147,483,648 to 2,147,483,647. Once the upper bound is reached, the sign bit will turn negative and continue incrementing.

Objective:
Controlling Program Flow

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

References:

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

---

Question #16 of 50

Question ID: 1328117

You are the programmer for your company's Human Resources department. They need to check if any of their employees is named Josh. You create the following code logic:

```
01  boolean found = empNames.stream()
02  // INSERT CODE
03  System.out.println(found);
```

Which statement should you use in line `02` to make this code function correctly?

✗ **A)** `.allMatch((s) -> s.equals("Josh"));`

✓ **B)** **`.anyMatch((s) -> s.equals("Josh"));`**

✗ **C)** `.noneMatch((s) -> s. equals("Josh"));`

✗ **D)** `.findAny((s) -> s.equals("Josh"));`

Explanation

You should to use the `anyMatch` method as shown below:

```
boolean found = empNames.stream()
  .anyMatch((s) ->  s.equals("Josh"));
System.out.println(found);
```

The `anyMatch` method returns a `boolean` value of true if any elements of the stream match the predicate, and `false` otherwise.

The `allMatch` method is incorrect because `allMatch` returns true when all elements of a stream match the predicate. No results would be returned in this case because not every employee is named Josh.

The `noneMatch` method is incorrect because `noneMatch` returns true when none of the elements match the predicate.

The `findAny` method is incorrect because it returns an `Optional` object, not a `boolean` as required in this scenario.

Objective:
Working with Streams and Lambda expressions

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

References:

Oracle Technology Network > Java SE > Java SE Documentation > Java Platform Standard Edition 11 API Specification > java.util.stream

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

---

Question #17 of 50

Question ID: 1328147

Given the following code fragment:

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

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

- ✗ **A)** A new thread will be created for each task, with five threads available initially.
- ✗ **B)** A new thread will be created for each fifth task.
- ✓ **C) Five threads will be created, each thread supporting a single task.**
- ✗ **D)** A new thread will be created for each task, up to five threads simultaneously.
- ✗ **E)** A single thread will be created for up to five tasks.

Explanation

Because the `newScheduledThreadPool` factory method is invoked with the argument 5, five threads will be created, each thread supporting a single task. Threads not being used will be idle. The number of threads created

for an `Executor` object is determined by the method used to obtain it. This thread pool supports tasks that run after a delay or are executed periodically.

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

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

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

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

Objective:
Concurrency

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

References:

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

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

---

Question #18 of 50

Question ID: 1327951

Which one of the following method signatures is required to sort a collection of `String` objects without using natural order?

✗ **A)** `public int compareTo(String s)`

✓ **B)** `public int compare(String s1, String s2)`

✗ **C)** `public boolean compareTo(String s)`

✗ **D)** **`public boolean compare(String s1, String s2)`**

Explanation

The method signature `public int compare(String s1, String s2)` is required to sort a collection of `String` objects without using natural order. The `compare` method returns 0 if the arguments are equal, a negative integer if

the first argument is less than the second argument, and a positive integer if the first argument is greater than the second argument.

This method is provided by the `Comparator` interface. A `Comparator` implementation is required when sorting elements in an order other than the default natural order.

The method signatures that return a `boolean` value are not valid signatures for either the `compareTo` or `compare` methods. These methods return an `int` value, indicating the relative positioning of objects to each other.

The `compareTo` method is not required to sort a collection without using natural order. The `compareTo` method is provided by the `Comparable` interface for elements to be sorted in their natural order. The `Comparable` interface is implemented by elements that can be sorted within a collection. The `compareTo` method returns negative, zero or a positive integer if the object being compared to the one that calls the method is less than equal or greater.

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 #19 of 50

Question ID: 1328050

Given:

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

Which exception is thrown by running the given code?

✗ **A)** `NullPointerException`

✗ **B)** `ArrayIndexOutOfBoundsException`

✓ **C)** `StringIndexOutOfBoundsException`

✗ **D)** `IndexOutOfBoundsException`

Explanation

`StringIndexOutOfBoundsException` is the exception thrown by running the given code. This is because the `charAt` method is invoked with an invalid index. A string with no characters (`""`) is an empty string, so that the effective size of the underlying `char` array is 0.

`NullPointerException` is not thrown by running the given code. Although the code uses an empty string, a `String` object has been created by specifying the literal `""`. `NullPointerException` is thrown whenever an object is required, such as when accessing instance members.

The exceptions `IndexOutOfBoundsException` nor `ArrayIndexOutOfBoundsException` will be thrown by running the given code. `IndexOutOfBoundsException` is more general than the thrown exception, while `ArrayIndexOutOfBoundsException` is thrown only when using an invalid index with an array, not a string.

Objective:
Exception Handling

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

References:

Oracle Documentation > Java SE 11 API > Class StringIndexOutOfBoundsException

---

Question #20 of 50

Question ID: 1328120

Given the code fragment:

```
System.out.println(
  Stream.iterate(0, (n)->n+1) // line n1
  .limit(2)
  .max() // line n2
);
```

What is the result?

- ✗ **A)** `1`
- ✓ **B)** Compilation fails at line `n2`.
- ✗ **C)** Compilation fails at line `n1`.
- ✗ **D)** `Optional[1]`
- ✗ **E)** `0`

Explanation

Compilation fails at line `n2`. The stream in the code fragment is an object stream, not a primitive stream. Unlike primitive streams, the `max()` method for an object stream requires a `Comparator` of the appropriate type. Because no `Comparator` is provided, compilation fails at line `n2`.

Compilation does not fail at line `n1`. Even though the stream is an object stream, the autoboxing system successfully (although inefficiently) converts the `int` result of the iteration `Function` into an `Integer` object, and then the `Integer` argument of the iteration `Function` to an `int`.

There is no output because the code fails to compile. If there was a valid `Comparator` provided to the `max()` method, then the output would be an `Optional`.

Objective:
Working with Streams and Lambda expressions

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

References:

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

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

Question #21 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) The `Voter` class cannot be instantiated.**
- ✗ **B)** Subclasses of `Voter` cannot override its methods.
- ✗ **C)** Concrete subclasses of `Voter` must use a default constructor.
- ✗ **D)** Subclasses of `Voter` must implement the `getPartyAffiliation` method.
- ✗ **E)** The `Voter` class cannot be extended.

✓ **F) Concrete subclasses of `Voter` must implement the `vote` 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 #22 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)** Compilation and execution complete normally, but no output is generated.

✗ **B)**
```
Fred
Jim
Sheila
```

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

✓ **D)**
```
Fred
Jim
```

✗ **E)**
```
Fred
```

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 #23 of 50

Question ID: 1328083

Consider following code:

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

public class NumberWorks {
  public static void main(String[] args) {
    IntPredicate iP = (a)-> a == 1980;
```

```
    System.out.println(iP.test(0));
  }
}
```

What is the output?

✗ **A)** `true`

✗ **B)** `0`

✗ **C)** `1980`

✓ **D)** **`false`**

Explanation

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

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

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

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

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

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

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

```
package java.util.function;

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

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

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

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

Objective:
Working with Streams and Lambda expressions

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

References:

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

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

---

Question #24 of 50

Question ID: 1328000

Which type of object should you use to execute stored procedures?

- ✓ **A)** `CallableStatement`
- ✗ **B)** `FilteredRowSet`
- ✗ **C)** `ResultSet`
- ✗ **D)** `Statement`
- ✗ **E)** `PreparedStatement`
- ✗ **F)** `CachedRowSet`

Explanation

You should use the `CallableStatement` interface to execute stored procedures. The correct option is to use the following code:

```
String storedProc = "{call addAgent(?, ?, ?)}";
CallableStatement callSt = con.prepareCall(sqlQuery);
callSt.setString(1, "James");
callSt.setString(2, "Bond");
callSt.setLong(3, 007);
callSt.addBatch();
```

```
callSt.setString(1, "Alec");
callSt.setString(2, "Trevelyan");
callSt.setLong(3, 006);
callSt.addBatch();
int[] res = callSt.executeBatch();
```

`CallableStatement` allows you to store three types of parameters for stored procedures:

1. *in* parameters are sent as input values to stored procedures.
2. *out* parameters are used to store the output and result parameters returned from the stored procedure.
3. *in-out* parameters support bi-directional input and output with the stored procedure.

Note that only in-out and out parameters must be registered before calling the stored procedure using the `CallableStatement` interface.

You should not use `Statement` interface or its subinterface `PreparedStatement` because they do not support stored procedures.

You should not use `ResultSet` or its subinterfaces `CachedRowSet` and `FilteredRowSet` because these interfaces are used for iterating through returned rows with read operations. `CachedRowSet` and `FilteredRowSet` are examples of disconnected interfaces, supporting read operations after the connection has been closed.

Objective:
Database Applications with JDBC

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

References:

Oracle Technology Network > MySQL Connector > Using JDBC CallableStatements to Execute Stored Procedures

---

Question #25 of 50

Question ID: 1327959

Consider the following code for a service client application:

```
package applic;

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

public class ClientApp {
    public static void main(String[] args) {
        ServiceLoader<SpeakerInterface> services =
ServiceLoader.load(SpeakerInterface.class);
        services.findFirst().ifPresent(t -> t.speak());
```

```
    }
}

module modClient {
    requires modServ;
//Insert code here
}
```

Which code fragment will you insert into the code above to successfully load instances of service providers for the `package1.SpeakerInterface` service interface?

✗ **A)** `import package1.SpeakerInterface;`

✗ **B)** `package package1;`

✗ **C)** `provides package1.SpeakerInterface;`

✓ **D)** `uses package1.SpeakerInterface;`

Explanation

You will use the following code fragment to successfully load instances of service providers for the `package1.SpeakerInterface` service interface:

```
uses package1.SpeakerInterface;
```

To successfully load instances of service providers for the `package1.SpeakerInterface` service interface, the module declaration needs to include the keyword `uses`.

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.

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.

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 #26 of 50

Question ID: 1327940

Consider the following code:

```
import java.util.Map;
public class CompSci {
    public static void main(String[] args) {
    Map<Integer,String> compscimap = Map.of(101,"C Language",102,"Mathematics",103,"Computer Architecture");
    for(Map.Entry<Integer, String> m : compscimap.entrySet()){
            System.out.println(m.getKey()+" "+m.getValue());
    }
    compscimap.clear();
    }
}
```

What would be the output?

✓ **A)** An `UnsupportedOperationException` is thrown.

✗ **B)** The code fails compilation.

✗ **C)**
```
101 C Language
102 Mathematics
103 Computer Architecture
```

✗ **D)**
```
101 C Language
```

Explanation

An `UnsupportedOperationException` is thrown. The following output will be displayed:

```
Exception in thread “main” java.lang.UnsupportedOperationException
at java.util.Collections$UnmodifiableMap.clear(Unknown Source)
at main(mapped.java)
```

This is because it is an illegal operation to attempt to modify an immutable list, map, or set. In this code example, the method `compscimap.clear()` attempts to clear the contents of the map. Any object in Java is called “immutable” if its state is not allowed to be changed after it is constructed. An immutable instance of a collection will always have the same data in it while a reference to it exists. A map created using `Map.of()` is an immutable map. The `Map.of()` and `Map.ofEntries()` are both static factory methods, both of which create immutable maps.

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

To create immutable maps, you use the `Map.of()` and `Map.ofEntries()` static factory methods. The maps made using these methods do not allow null keys or values and reject duplicate keys on creation. The map is serializable if all key value pairs are serializable as well.

The two code outputs are both incorrect because the contents of the map would never be printed to standard output. An `UnsupportedOperationException` exception will be generated. These outputs would be displayed if the following line of code (which causes the exception) is omitted:

```
compscimap.clear();
```

Objective:
Working with Arrays and Collections

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

References:

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

---

Question #27 of 50

Question ID: 1328122

Given:

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

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

✗ **A)** `.max()`

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

✗ **C)** `.sum(System.out::println)`

✓ **D)** `:.count()`

Explanation

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

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

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

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

Objective:
Working with Streams and Lambda expressions

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

References:

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

---

Question #28 of 50

Question ID: 1327760

Given the following:

```
public class Java11 {
   public static void main (String[] args) {
     int x = 1;
     modifyVar(x + 5);
     System.out.println("x: " + x);
   }
   public static void modifyVar(int var) {
     var = 10;
   }
}
```

What is the result when this program is executed?

✓ **A)** `x: 1`

✗ **B)** `x: 16`

✗ **C)** `x: 10`

✗ **D)** `x: 6`

Explanation

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

```
x: 1
```

Variables of primitive types hold only values, not references. When `var` is assigned to `x` because of invoking the `modifyVar` method, the value of `x` is copied into `var` . The actual argument is `x + 5`, so that `var` is initially set to `6`. Within the `modifyVar` method, the `var` variable is set to `10`, overwriting its previous value. Because `var` only received a copy of `x` in the expression `x + 5`, any modifications to `var` are discarded after returning from the `modifyVar` method. Thus, the value of `x` remains `1`.

The key differences 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 results will not be the output with `x` set to `6` or `10`, because only a copy of the `x` value is stored in `var` for the `modifyVar` method. Initially, `var` is set to `6` and then set to `10`.

The result will not be the output with `x` set to `16`, because only a copy of the `x` value is stored in `var` and `var` is never set to `16`. The final value of `var` is `10`, not `16`.

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 #29 of 50

Question ID: 1327907

You define the following interface to represent shippable goods:

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

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

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

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

- ✓ **A)** Add a default method to the interface
- ✗ **B)** Add a static method to the interface
- ✗ **C)** Create a helper, or utility, class and add the method to that class
- ✗ **D)** Convert the interface to an abstract class and add the method to the class

Explanation

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

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

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

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

Objective:
Java Object-Oriented Approach

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

References:

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

---

Question #30 of 50

Question ID: 1327795

Given the following:

```
public class MyBasicClass {
   static int field;  //line 1
   int getField() {}  //line 2
   static void Main(String[] args) {  //line 3
     System.out.println(field);  //line 4
   }
}
```

Which line causes a compilation error?

- ✓ **A)** line 2
- ✗ **B)** line 1
- ✗ **C)** line 4
- ✗ **D)** line 3

Explanation

Line 2 causes a compilation error. This is because the `getField` method body is missing a `return` statement. In a concrete class, all methods and constructors must have bodies and return any declared values. If `getField` method does not return a value, then it needs to have `void` as its return type instead of `int`.

Line 1 does not cause a compilation error because `static` fields are allowed in classes. Classes can contain both `static` and instance fields.

Line 3 will not cause a compilation error because the `main` method is not required for every class. The `main` method is the entry point for an application and is only required in the starting class.

Line 4 will not cause a compilation error because variable fields can be referenced in the `Main` method. `static` fields can be accessed from a static context.

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 > Declaring Member Variables

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

---

Question #31 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)** Reference to the variable is copied to the lambda statement.

✗ **B)** Value of the variable can be changed within the lambda statement.

✓ **C)** The variable must be effectively final.

✗ **D)** The variable must be declared using the final keyword.

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 #32 of 50

Question ID: 1328078

Given the lambda expression:

```
(v)->System.out.println("The value is " + v)
```

Which interface(s) could it implement? (Choose all that apply.)

- ✗ **A)** `Supplier`
- ✗ **B)** `UnaryOperator`
- ✗ **C)** `Predicate`
- ✓ **D)** `Consumer`
- ✗ **E)** `Function`

Explanation

This lambda expression could implement the `Consumer` interface. The `Consumer` interface defines the `accept` method, which takes a single reference type parameter and returns `void`. This matches the given lambda, which takes a single parameter (`v`) and returns `void`, because that is the return type of the `println` method.

All of the other interfaces return a value. `Function` and `Supplier` return arbitrary object types, `Predicate` returns `boolean`, and `UnaryOperator` returns the same type as its input argument.

Objective:
Working with Streams and Lambda expressions

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

References:

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

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

---

Question #33 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_fr.properties`
- ✓ **C)** `MessageTextBundle.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 #34 of 50

Question ID: 1328170

Given the following code:

```
01 public class ThreadingMain extends Thread {
02  private static int counter;
03
04  public static void main(String[] args) {
05    Thread th1 = new ThreadingMain();
06    Thread th2 = new ThreadingMain();
07    Thread th3 = new ThreadingMain();
08    Thread th4 = new ThreadingMain();
09    Thread th5 = new ThreadingMain();
10    th1.start();th2.start();th3.start();
11    th4.start();th5.start();
12  }
13  @Override
14  public void run() {
15    String threadName = Thread.currentThread().getName();
16    while (++counter < 10) {
17      try {
18        Thread.sleep(500);
19      } catch (Exception ex) {
20        System.err.println(ex.toString());
21      }
22    }
23    System.out.println(threadName + ": " + counter);
```

```
24  }
25 }
```

Which statement is true about compiling and executing this code?

- ✗ **A)** The final value of the `counter` variable will be 9.
- ✓ **B)** The current value of the `counter` variable may not be visible to all threads.
- ✗ **C)** The current value of the `counter` variable will be visible to all threads.
- ✗ **D)** The final value of the `counter` variable will be 10.
- ✗ **E)** The final value of the `counter` variable will be 11.

Explanation

Because the `counter` variable is shared across five threads without any locking mechanism or `volatile` declaration, the current value of the counter variable may not be visible to all threads. This means that some threads may not have access to the current value when performing the increment operation and conditional evaluation on line `16`. One solution is to resolve the visibility issue by to apply the `volatile` keyword to line `02`. However, increment and decrement operations are not atomic. A better solution that will also ensure read and write operations are atomic would be to use a simple locking mechanism with a synchronized accessor and mutator. The following code demonstrates this technique:

```
public synchronized int getCurrentCounter() { return counter; }
public synchronized int incrementCounter() { return counter++; }
```

Then, lines `16` and `23` would be modified to use these two methods, rather than accessing the variables directly. The `synchronized` keyword only allows access to the specified method or code block by a single thread at a time.

The statements that predict the final value of the `counter` variable are incorrect. Because multiple threads are performing read and write operations dependent on non-volatile variable, the final value cannot be predicted for every execution cycle. If every thread can access the `counter` variable, then the most likely final value will be 14. This value is most likely because if all threads retrieve the counter value, then only one thread will end based on the condition, while all other threads will increment at least once.

The current value of the `counter` variable will not necessarily be visible to all threads. Without using the `volatile` keyword being used in the counter declaration, visibility to multiple threads is not guaranteed.

Objective:
Concurrency

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

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Essential Classes > Concurrency > Synchronized Methods

Question #35 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)** `import java.text.*;`

✓ **B)** `static final int VAL=1000;`

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

✗ **D)** `module basicModule {}`

✗ **E)** `package basicPackage;`

✓ **F)** `void BasicMethod() {}`

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 #36 of 50

Question ID: 1327923

Given:

```
class ObjectFileWriter<_________________> {
  public void write(T obj, String filename) {
    try (FileOutputStream fos = new FileOutputStream(filename);
      ObjectOutputStream oos = new ObjectOutputStream(fos);){
        oos.writeObject(obj);
        oos.flush();
    } catch (IOException ex) {}
  }
}
```

Which code segment should be inserted to compile `ObjectFileWriter`?

✗ **A)** `? super Serializable`

✗ **B)** `T implements Serializable`

✗ **C)** `Serializable`

✓ **D)** `T extends Serializable`

Explanation

The code segment `T extends Serializable` should be inserted to compile `ObjectFileWriter`. This code segment bounds the generic parameter `T`, so that only those types that implement the `Serializable` interface can be specified. The keyword `extends` applies to an interface or a superclass when bounding generic parameters.

The code segment `Serializable` should not be inserted to compile `ObjectFileWriter`. Although this code segment does specify a type, it does not define the generic parameter `T` used in the write method.

The code segment `? super Serializable` should not be inserted to compile `ObjectFileWriter`. The wildcard character `?` is only used for variables and return types, not for generic methods and classes. The `super` keyword in this context specifies a lower bound, so that any super type of `Serializable` is allowed.

The code segment `T implements Serializable` should not be inserted to compile `ObjectFileWriter`. The `implements` keyword is not valid in this context. The keyword `extends` applies to an interface or a superclass when bounding generic parameters.

Objective:

Working with Arrays and Collections

Sub-Objective:

Use generics, including wildcards

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Bounded Type Parameters

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

---

Question #37 of 50

Question ID: 1327813

Given the following code line:

```
printToConsole("Cup of Java with a shot of espresso");
```

Which overloaded method is invoked?

✗ **A)**
```
void printToConsole(Integer intObj) {
    System.out.println("My number is " + intObj.toString());
}
```

✗ **B)**
```
void printToConsole(StringBuffer buffer) {
    System.out.println(buffer.toString());
}
```

✗ **C)**
```
void printToConsole() {
    System.out.println("Hello, world!");
}
```

✓ **D)**
```
void printToConsole(Object obj) {
    System.out.println("My object is " + obj.toString());
}
```

Explanation

The following overloaded method is invoked:

```
void printToConsole(Object obj) {
   System.out.println("My object is " + obj.toString());
}
```

This method is invoked because there is no overloaded method that accepts a `String` argument, and any argument other than a `StringBuffer` or `Integer` will invoke this method. All objects inherit from the `Object` class, so this overloaded method is an effective catch-all for an argument whose data type is not explicitly declared in another overloaded method. When a method is invoked on an overloaded method, only the list of parameter data types determines which method is executed.

The overloaded method with no parameters is not invoked. The invocation specifies a `String` argument, so this overloaded method will not be executed. This overloaded method would be executed if the invocation contained no argument.

The overloaded method with a `StringBuffer` parameter is not invoked. Although the invocation specifies a `String` argument, `String` arguments are not automatically converted into `StringBuffer` objects because these classes are not related with inheritance. This overloaded method would be executed if the invocation contained a `StringBuffer` argument.

The overloaded method with an `Integer` parameter is not invoked. `String` arguments are not converted into `Integer` objects because these classes are not related with inheritance. This overloaded method would be executed if the invocation contains an `Integer` argument.

Objective:
Java Object-Oriented Approach

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

References:

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

---

Question #38 of 50

Question ID: 1327770

Given the following code:

```
public class WrapperTest {
  public static void main(String[] args) {
    System.out.println(Integer.valueOf("777.77"));
  }
}
```

What would be the output of this code fragment?

✗ **A)** `77`

✗ **B)** `777.77`

✓ **C)** `NumberFormatException`

✗ **D)** `777`

Explanation

The code throws a `NumberFormatException`. The error occurs because the `Integer` wrapper is passed a value that is incorrect for its type, in this case a value containing dots.

All the other options are incorrect because the code does not execute normally due to a `NumberFormatException` error.

Objective:
Working with Java Data Types

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

References:

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

---

Question #39 of 50

Question ID: 1327957

Once you have a module declared and the dependencies identified, you need to compile your Java module and send the generated class files to a specific folder.

Which command would you use to do this?

✗ **A)** `javac -s`

✗ **B)** `javac -g`

✓ **C)** `javac -d`

✗ **D)** `java`

Explanation

The command you would use to compute your Java module and send the class file to a specific folder is `javac -d`. The `-d` option stands for the directory path. Here you articulate where your class files are to be generated and sent. The `-s` option is used to specify where the source files are to be placed. The `-g` option is used to enable detailed debugging during the compile process.

The command `javac -s` is used to determine the source file location and is not used to specify the class directory location.

The command `javac -g` is used to add detailed debugging to your output and is not used for sending the class file to a specific directory.

The command `java` is not used to compile your Java source code but is used to execute the Java program.

Objective:
Java Platform Module System

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

References:

docs.oracle.com > Module jdk.compiler > javac

fedoraproject.org > JDK on Fedora > JDK components

Question #40 of 50

Question ID: 1327912

Which two types are valid functional interfaces?

✗ **A)**
```
public interface Useful<E> {
  E getStuff();
  void putStuff(E e);
}
```

✓ **B)**
```
public interface Useful { void doStuff(); }
```

✗ **C)**
```
@FunctionalInterface
  public interface Useful { }
```

✗ **D)**
```
@FunctionalInterface
  public interface { default void doStuff(){} }
```

✓ **E)**
```
interface Useful {
  void doStuff();
  default void doOtherStuff() {}
}
```

Explanation

The following two types are valid functional interfaces:

```
public interface Useful { void doStuff(); }
```

```
interface Useful {
  void doStuff();
```

```
  default void doOtherStuff() {}
}
```

A functional interface is one in which exactly one abstract method is declared. Default methods or static methods do not affect whether an interface is functional.

The annotation `@FunctionalInterface` does not make an interface functional. This annotation declares the programmer's intent to create a functional interface, allowing the compiler to issue an error if the requirements are not met.

The interface with no declared methods fails because it does not provide an abstract method. Remember that the use of the annotation is irrelevant.

The interface with a single default method also fails because it does not declare one abstract method.

The interface that declares the two methods, `E getStuff()` and `void putStuff(E e)`, fails because they are both abstract methods. Therefore, the interface declares one too many abstract methods.

Objective:
Java Object-Oriented Approach

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

References:

The Java Tutorials > Learning the Java Language > Classes and Objects > Approach 5: Specify Search Criteria Code with a Lambda Expression

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

---

Question #41 of 50

Question ID: 1327961

Consider the following module definition

```
module modProv {            // Line 1
    requires modServ;         // Line 2
    provides package1.SpeakerInterface
  with package2.SpeakerImplementation;  //Line 3
    exports package2;          // Line 4
};
```

Which line of code is not an appropriate way of defining the module?

- ✗ **A)** Line 3
- ✗ **B)** Line 1

✓ **C)** Line 4

✗ **D)** Line 2

Explanation

Line 4 is inappropriate because a service provider that is developed as a module must *not* export the implementation of the service. There exists no support for modules that specify other service providers in separate modules via a `provides` directive.

Line 1 is appropriate because it is a valid usage of the `module` keyword for definition of a module.

Line 2 is appropriate because it is a valid usage of the `requires` keyword needed for specifying a necessary module.

Line 3 is appropriate because it contains necessary uses of the `provides` and `with` keywords, both of which are necessary for finding and loading service providers.

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

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

Objective:
Java Platform Module System

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

References:

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

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

---

Question #42 of 50

Question ID: 1328127

Given the code fragment:

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

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

✗ **A)** `.sorted(String::compare)`

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

✗ **C)** `.sort((a,b)->a.compareTo(b))`

✗ **D)** `.sort()`

Explanation

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

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

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

Objective:
Working with Streams and Lambda expressions

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

References:

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

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

---

Question #43 of 50

Question ID: 1327775

Given:

```
String str = "Goodbye, Dog";
```

Which code fragment will output `Good, Dog`?

✗ **A)**
```
str.replace("bye","");
System.out.println(str);
```

✗ **B)**
```
var strNew = str.delete(4,8);
System.out.println(strNew);
```

✗ **C)**
```
str.delete(4,8);
System.out.println(str);
```

✓ **D)**
```
var strNew = str.replace("bye","");
System.out.println(strNew);
```

Explanation

The following code fragment will output `Good, Dog`:

```
var strNew = str.replace("bye","");
System.out.println(strNew);
```

The `replace` method will attempt to find every instance of the first string and replace them with the second string. As with all manipulation methods in the `String` class, the original `String` object is not modified, so a new `String` object is created and returned.

The code fragments that use the `delete` method will fail compilation because the `String` class does not provide this method. The `StringBuilder` class provides the `delete` method to remove characters based on the specified start and end positions.

The following code fragment will not result in the output `Good, Dog`:

```
str.replace("bye","");
System.out.println(str);
```

The output is `Goodbye, Dog`, which is the original string literal stored in `str`. This is because `String` objects are immutable. Manipulation methods do not modify the actual `String` object itself, but create and return new `String` objects.

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 > Manipulating Characters in a String

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

Question #44 of 50

Question ID: 1328019

Given the following:

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

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

✗ **A)** 10

✓ **B)** 18

✓ **C)** 14

✗ **D)** 16

✗ **E)** 6

Explanation

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

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

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

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

Objective:
Controlling Program Flow

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

References:

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

---

Question #45 of 50

Question ID: 1327794

Which statement is true about declaring members in a concrete class?

- ✗ **A)** Only `static` fields and methods are supported.
- ✗ **B)** Only instance fields and methods are supported.
- ✗ **C)** All fields must be initialized explicitly.
- ✓ **D)** All methods and constructors must contain bodies.

Explanation

All methods and constructors must contain bodies in a concrete class. Only interfaces and abstract classes support methods without bodies. Constructors cannot be declared without bodies, even in abstract classes.

All fields do not need to be initialized explicitly. By default, `static` and instance fields are automatically initialized to their default values.

Both instance and `static` fields and methods are supported by concrete classes.

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 > Declaring Member Variables

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

---

Question #46 of 50

Question ID: 1327810

Which two characteristics distinguish overloaded methods?

- ✓ **A)** Parameter data types
- ✓ **B)** Parameter number
- ✗ **C)** Return type
- ✗ **D)** Access modifiers
- ✗ **E)** Method name

Explanation

Parameter number and parameter data types distinguish overloaded methods. A method signature consists of the method name and its ordered list of parameter data types. To create an overloaded method, you should use the same method name with a different list of parameter data types. When a method is invoked on an overloaded method, it is only the list of parameter data types that determines which method is executed. No other method aspects, including visibility, return type, and other modifiers, affect the method signature.

Access modifiers do not distinguish overloaded methods. Access and other modifiers are not included in the method signature.

Method names do not distinguish overloaded methods. The method name is the same across overloaded methods.

Return type does not distinguish overloaded methods. Return type is not included in the method signature.

Objective:
Java Object-Oriented Approach

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

References:

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

---

Question #47 of 50

Question ID: 1328109

Given the following code:

```
List<String> metalStyles = new ArrayList<String>(Arrays.asList("heavy", "speed", "power",
"black"));

System.out.println(metalStyles.stream().peek(c->System.out.println(c)).count());
```

What would be the output of this code?

✓ **A)**
```
heavy
speed
power
black
4
```

✗ **B)**
```
black
true
```

✗ **C)**
```
heavy
true
```

✗ **D)**
```
speed
true
```

✗ **E)**
```
power
true
```

Explanation

The correct output is:

```
heavy
speed
power
black
4
```

The other options are incorrect because the `peek` method will work on results it receives through the lambda expression. Since only the lambda expression specifies a `println` command, it prints each item on a separate line, including the total number of items with the `count` method.

The `peek()` method is used to view the top-most element of a stack without having to remove it from the stack. A queue and a stack are both collections in Java that hold elements that can then be processed. A stack is a data structure that follows the Last-in-First-Out principle, where the last element that is inserted into the stack is the one that is removed first before another element can be accessed.

You can use the `peek` method for a stream. When used on a stream, the `peek` method performs the action as specified by a lambda expression on *each* stream element.

Objective:
Working with Streams and Lambda expressions

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

References:

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

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

Question #48 of 50

Question ID: 1327968

Which code fragment prompts for a password using secure entry?

✗ **A)**
```
Console cmd = System.console();
String pwd = cmd.readLine("Please enter password: ");
```

✓ **B)**
```
Console cmd = System.console();
char[] pwd = cmd.readPassword("Please enter password: ");
```

✗ **C)**
```
Console cmd = Console.getInstance();
String pwd = cmd.readLine("Please enter password: ");
```

✗ **D)**
```
Console cmd = Console.getInstance();
char[] pwd = cmd.readPassword("Please enter password: ");
```

✗ **E)**
```
Console cmd = new Console();
String pwd = cmd.readLine("Please enter password: ");
```

✗ **F)**
```
Console cmd = new Console();
char[] pwd = cmd.readPassword("Please enter password: ");
```

Explanation

The following code fragment prompts for a password using secure entry:

```
Console cmd = System.console();
char[] pwd = cmd.readPassword("Please enter password: ");
```

This code retrieves a `Console` instance from the `System` class and then invokes the `readPassword` method with a text prompt. The `readPassword` method ensures secure entry by suppressing the echo of characters and returning a `char` array that can be overwritten.

The code fragments that invoke the `readLine` method do not prompt using secure entry. The `readLine` method echoes characters as they are typed and returns a `String` object.

The code fragments that invoke the `getInstance` method on `Console` will fail compilation. The `Console` class does not provide this method. To get a `Console` instance, use the `console` method on the `System` class.

The code fragments that directly instantiate the `Console` class will fail compilation. The `Console` class does not provide a public constructor.

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 #49 of 50

Question ID: 1327963

Consider the following code output:

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

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

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

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

- ✗ **A)** `modProv`
- ✗ **B)** `SpeakerInterface`
- ✓ **C)** `modServ`
- ✗ **D)** `package1`

Explanation

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

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

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

Service providers that are created inside modules cannot control when they are instantiated. However, if the service provider declares a `provider` method, then the service loader will invoke an instance of the service provider via

that method. If a `provider` method is not declared in the service provider, there is a direct instantiation of the service provider by the service provider's constructor. In this case the service provider needs to be assignable to the class or interface of the service. A service provider that exists as an automatic module inside an application's module path needs a provider constructor.

Objective:
Java Platform Module System

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

References:

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

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

---

Question #50 of 50

Question ID: 1327953

Which of the following are principles of how JDK implements a modular structure? (Choose two.)

- ✓ **A)** Additional modules not governed by JCP start with the prefix `jdk.`.
- ✗ **B)** Standard modules must grant implied readability only standard modules.
- ✗ **C)** Standard modules will only depend on standard modules with no dependencies.
- ✗ **D)** Standard modules can only contain standard API packages.
- ✓ **E)** Standard modules managed by JCP start with the name `java.` .

Explanation

The modular structure of how JDK designs these principles are based off standard modules who are managed by Java Community Program (JCP) and can be identified with the prefix `.java`. The other design principle is that the additional modules outside of the JCP start with `jdk.`.

The main goal of a modular JDK design was to make java implementations easier to maintain, improve security, and improve application performance and to give developers better tools for a more user-friendly experience.

Standard modules can only contain standard API packages. Actually, these standard modules can have API packages that are both standard and non-standard API packages.

Standard modules will only depend on standard modules with no dependencies. Standard modules can have dependencies that exists on more than one standard module and could have dependencies on non-standard dependencies as well.

You would not have standard modules that only grant implied readability only standard modules. If you have a non-standard module, you cannot grant implied readability from a standard module.

Objective:

Java Platform Module System

Sub-Objective:

Deploy and execute modular applications, including automatic modules

References:

openjdk.java.net > JEP 200: The Modular JDK > Design principles