Quick Quiz July 14, 2022

Test ID: 216186532

Question #1 of 50

Question ID: 1328066

Given a method declaration:

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

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

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

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

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

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

E) `s->s`

Explanation

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

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

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

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

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

Objective:
Working with Streams and Lambda expressions

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

References:

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

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

Question #2 of 50

Question ID: 1328006

Consider the following code:

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

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

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

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

A) `@Deprecated`

B) `@SupressWarnings`

C) `@Override`

D) `@Documented`

Explanation

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

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

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

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

Java has seven predefined annotation types:

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

Objective:
Annotations

Sub-Objective:
Create, apply, and process annotations

References:

Oracle Technology Network > Java SE Documentation > Annotations

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

---

Question #3 of 50

Question ID: 1327919

Given:

```
public enum Architecture {
  ARM, x86, x86_64, RISC, MIPS, SPARC, UNIVAC;
  public static void main(String[] args) {
    for(Architecture a: Architecture.toArray()) {
      System.out.print(a.ordinal() + " ");
    }
  }
}
```

What is the result?

A) An exception is thrown at runtime.

B) `1 2 3 4 5 6 7`

C) `ARM x86 x86_64 RISC MIPS SPARC UNIVAC`

D) `0 1 2 3 4 5 6`

E) Compilation fails.

Explanation

Compilation fails. The `Enum` class does not provide the method `toArray`, which is available in the `Collection` class. The `Enum` class instead provides the values method that can be used to iterate through enumeration constants.

The result is not the output `0 1 2 3 4 5 6` because compilation fails. This would be the output if the `toArray` method were replaced with the `values` method. The ordinal method returns the zero-based index of an enumeration constant.

The result is not the output `1 2 3 4 5 6 7` because compilation fails. Also, the ordinal method returns a zero-based index.

The result is not the output `ARM x86 x86_64 RISC MIPS SPARC UNIVAC` because compilation fails. Also, this is not the result because the ordinal method is invoked. This would be the output if the `name` or `toString` methods were invoked.

The result is not an exception at runtime because the code fails to compile.

Objective:
Java Object-Oriented Approach

Sub-Objective:
Create and use enumerations

References:

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

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

Question #4 of 50

Question ID: 1327986

Given the following code fragment:

```
Path fPath = Paths.get("C:\\Documents\\JavaProjects\\Java11\\protected.enc");
try {
  UserPrincipal jhester = fPath.getFileSystem().getUserPrincipalLookupService().
  lookupPrincipalByName("jhester");
  BasicFileAttributeView fView = Files.getFileAttributeView(fPath,
BasicFileAttributeView.class);
  fView.setOwner(jhester);
} catch (Exception ex) {
  System.err.println(ex.toString());
}
```

Assuming the file `protected.enc` exists in the specified path and the program has adequate permission to perform its operations, what is the result of compiling and executing this code?

A) The user jhester has read, write, and execute permissions on the `protected.enc` file.

B) The user jhester has read and write permissions on the `protected.enc` file.

C) The user jhester controls permission assignment on the `protected.enc` file.

D) A runtime exception is thrown.

E) Compilation fails.

Explanation

Compilation fails, because the `BasicFileAttributeView` interface does not support the `owner` attribute. The `FileOwnerAttributeView` interface and its subinterfaces support reading and setting the owner of a file, but `BasicFileAttributeView` does not provide this functionality. The subinterfaces `PosixFileAttributeView` and `AclFileAttributeView` also support additional security attributes required by Unix and Windows platforms.

The result will not affect permissions for the user jhester on the `protected.enc` file because compilation fails with the `setOwner` invocation. If the `FileOwnerAttributeView` interface or one of its subinterfaces were used, then the result would allow the user jhester to control permission assignment on the `protected.enc` file.

The code fragment will not throw a runtime exception because exceptions are handled by the catch block.

Objective:

Java File I/O

Sub-Objective:

Handle file system objects using java.nio.file API

References:

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

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Essential Classes > Basic I/O > Managing Metadata (File and File Store Attributes)

Question #5 of 50

Question ID: 1327839

Given:

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

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

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

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

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

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

Explanation

This class is poorly encapsulated. The `totalItems` field should be calculated in the constructor and in any methods that modify the `compartments` field. Performing the `totalItems` calculation within the class only when a dependent

field changes will ensure the field reflects the current object state and does not require users of the class to perform the calculation manually. This both protects the `totalItems` field and increases overall usability.

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

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

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

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

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

   public int getTotalItems() {
     return totalItems;
   }

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

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

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

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

controlled internally and is not required by all classes for encapsulation.

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

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

Objective:
Java Object-Oriented Approach

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

References:

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

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

Question #6 of 50

Question ID: 1327927

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

A) `List<? super Buffer> bbList = new ArrayList<>();`

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

C) `List<Buffer> bbList = new ArrayList<ByteBuffer>();`

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

Explanation

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

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

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

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

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

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

Objective:
Working with Arrays and Collections

Sub-Objective:
Use generics, including wildcards

References:

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

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

---

Question #7 of 50

Question ID: 1328142

Given the code fragment:

```
System.out.println(Stream.of("Fred", "Jim", "Sheila")
  .parallel()
  .collect(String::new, String::concat, String::concat)); // line n1
```

What is the result?

A) The three names, *Fred*, *Jim*, *Sheila*, are concatenated without spaces, but not necessarily in the original order

B) `Fred Jim Sheila`

C) A compilation error at line `n1`

D) A single blank line

E) `FredJimSheila`

Explanation

The result is a single blank line. The collect operation works with mutable intermediate values. Each sub-stream created to support multiple threading is given an object created by the first argument of the collect invocation. The `collect` behavior then proceeds by attempting to modify that value repeatedly with each new value from the stream.

It is inappropriate to use `String` as the intermediate value because it is immutable. The effect of the code is to create an empty `String` (or one for each thread that is allocated behind the scenes). This new empty string, call *bucket*. Then, for each new value (*value*) from the stream, the collection operation will perform in effect `bucket.concat(value)`. Unfortunately, the new value is abandoned, and the object referred to by *bucket* is unchanged.

When all the threads have completed, the separate bucket values from each thread are empty, and are concatenated the in the same flawed manner. The result is that the `collect` operation creates an empty string, and the only reason we see any output at all is the use of the `println` statement.

The other results are incorrect because the collector creates an empty string.

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 > Reduction

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

---

Question #8 of 50

Question ID: 1327903

Given:

```
public interface ShiftCipher {
   String encrypt(String plaintext ,int shift);
   String decrypt(String ciphertext, int shift);
   default int getRandomShift(int max) {
     return (new Random()).nextInt(max) + 1;
   }
}
```

Which two types use `ShiftCipher` correctly?

A)
```
public interface AutoShiftCipher implements ShiftCipher {
      public byte[] encrypt(byte[] plaintext);
      public byte[] decrypt(byte[] ciphertext);
   }
```

B)
```
public abstract class CaesarCipher implements ShiftCipher {
   public String encryptAndDecrypt(String txt)
{/*implementation omitted*/}
   public int getRandomShift(int max) {/*implementation
omitted*/}
}
```

C)
```
public interface AutoShiftCipher extends ShiftCipher {
   public byte[] encrypt(byte[] plaintext);
   public byte[] decrypt(byte[] ciphertext);
}
```

D)
```
public class CaesarCipher implements ShiftCipher {
   public byte[] encrypt(byte[] plaintext, int shift)
{/*implementation omitted*/}
   public byte[] decrypt(byte[] ciphertext, int shift) {/*
implementation omitted */}
}
```

E)
```
public class CaesarCipher extends ShiftCipher {
   public String encrypt(String plaintext, int shift)
{/*implementation omitted*/}
   public String decrypt(String ciphertext, int shift) {/*
implementation omitted */}
}
```

Explanation

The following types use `ShiftCipher` correctly:

```
public interface AutoShiftCipher extends ShiftCipher {
   public byte[] encrypt(byte[] plaintext);
   public byte[] decrypt(byte[] ciphertext);
}

public abstract class CaesarCipher implements ShiftCipher {
   public String encryptAndDecrypt(String txt) {/*implementation omitted*/}
}
```

The first type is an interface that extends `ShiftCipher`, while the second type is an abstract class that implements `ShiftCipher`. `AutoShiftCipher` is another interface that adds overloaded methods to `ShiftCipher`. Interfaces can provide additional method declarations or constants to an existing interface by extending it. These methods can overload and override methods from the original interface. The reason that `CaesarCipher` is declared abstract is because this class does not implement the required `encrypt` and `decrypt` methods. Using an interface contract, a class must either provide implementation for all non-default methods declared in an interface or declare itself as abstract so that its subclasses can provide implementation. Because the `getRandomShift` method is declared with

the `default` keyword, implementing classes can inherit the default implementation or override it. Default methods do not require explicit declaration or implementation in implementing classes.

The interface that uses the `implements` keyword does not use `ShiftCipher` correctly. An interface cannot contain non-default implementation, and so it cannot use the `implements` keyword. An interface can extend another interface using the `extends` keyword.

The class that uses the `extends` keyword does not use `ShiftCipher` correctly. Because `ShiftCipher` is an interface, the keyword `implements` is required for a class to use `ShiftCipher`. The `extends` keyword is used between interfaces and between classes, not between an interface and a class.

The class that implements the `ShiftCipher` by overloading its methods does not use `ShiftCipher` correctly. Using an interface contract, a class must provide implementation for all non-default methods declared in an interface. Overloading methods is allowed, but only within the class after the implementation is provided for the non-default methods declared in the interface.

Objective:
Java Object-Oriented Approach

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

References:

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

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

---

Question #9 of 50

Question ID: 1328033

Given the following:

```
public class Java11_Looping {
   public static void main(String[] args) {
     char[] charArray = { 'e', 's', 'p', 'r', 'e', 's', 's', 'o', '8', '9', '0'};
     int i = 48; //Start range for digits
     do {
       for(char c : charArray)
         if ((char) i == c)
           System.out.println(c + " found!");
     } while (i++ < 57); //End range for digits
```

```
    }
}
```

How many times is `found!` printed?

A) Twice

B) Thrice

C) Once

D) None

Explanation

`found!` is printed thrice as follows in the output:

```
0 found!
8 found!
9 found!
```

The outer `do-while` loop will iterate 10 times, once for each digit character, while the inner enhanced `for` loop will iterate 11 times, once for each character in `charArray` for a total of 110 loops. The output `found!` is printed only when there is a character match. In this case, `charArray` contains only three digit characters, so `found!` is printed three times.

`found!` is not printed only once, twice or not at all. The outer loop will iterate through all digit characters, while the inner loop will check each character in `charArray`. There are three digit matches in `charArray`, so `found!` is printed three times.

Objective:
Controlling Program Flow

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

References:

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

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

---

Question #10 of 50

Question ID: 1327914

What statement is true of a functional interface?

A) Must contain one abstract method that is not inherited from a parent interface

B) Must be defined as implementing `FunctionalInterface`

C) Must be annotated with `@FunctionalInterface`

D) Must be defined as extending `FunctionalInterface`

E) Must contain a single abstract method or inherit it from a parent interface

Explanation

A functional interface is one that defines exactly one abstract method, though it is not important where that method originates. Therefore, it must contain a single abstract method or inherit it from a parent interface.

There is no parent interface or base class, so it cannot be defined as extending or implementing `FunctionalInterface`. Also, interfaces extend other interfaces, but they cannot implement each other.

The annotation `@FunctionalInterface` is not involved in the definition of a functional interface. This annotation declares your intent to define a functional interface to the compiler. By using the annotation, if you accidentally define more than one abstract method, or zero abstract methods, the compiler can inform you of the error right away, rather than discovering the error later.

Functional interfaces must contain one abstract method, which can be inherited from a parent interface. Provided that the interface has exactly one abstract method, it does not matter to the functional-interface aspect whether that method was inherited or not. Therefore, it is incorrect to state that a functional interface must contain one abstract method that was not inherited from a parent interface.

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 6: Use Standard Functional Interfaces with Lambda Expressions

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

---

Question #11 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) `:.count()`

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

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

D) `.max()`

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

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

D)
```
void printToConsole(StringBuffer buffer) {
   System.out.println(buffer.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 #13 of 50

Question ID: 1327761

Given the following:

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

   public static void main (String[] args) {
     RefType x = new RefType(1);
     modifyVar(x);
     x.val = x.val + 5;
     System.out.println("x: " + x.val);
   }

   public static void modifyVar(RefType var) {
     var.val = 10;
   }
}
```

What is the result when this program is executed?

A) `x: 1`

B) `x: 15`

C) `x: 6`

D) `x: 10`

Explanation

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

```
x: 15
```

Variables of reference types hold references to object instances. When `var` is assigned to `x` in the `modifyVar` method, `var` now references the same object referenced by `x`. Thus, changing the `val` field for `var` will affect `x`,

because all both variables reference the same object. The `val` field is initialized to `1`, then reassigned to `10` and incremented by `5` after returning from the `modifyVar` method. The final value for the `val` field is `15`.

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 the `val` field of `x` set to `1` or to `6`, because the `val` value of `x` is modified by the `modifyVar` method. Initially, `var` is set to `1` and then set to `10` in the `modifyVar` method.

The result will not be the output with the `val` field of `x` set to `10`, because after returning from the `modifyVar` method, the `val` field is incremented by `5`. The final value of `val` field is `15`, not `10`.

Objective:
Working with Java Data Types

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

References:

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

Primitive vs Reference Data Types

Question #14 of 50

Question ID: 1328029

Given the following code fragment:

```
int i1 = 2;
for (int i2 = 8; i1 < i2; i2 -= 2 ) {
   System.out.print(++i1 + " ");
}
```

What is the output?

A) `6 8`

B) `8 6`

C) `4 3`

D) `3 4`

Explanation

The output is `3 4`. In the `for` block, the `i1` variable is incrementing by 1 with each iteration, while the variable `i2` is decrementing by 2 with each iteration. Before the first iteration occurs, the value for `i1` is 2, and `i2` is initialized to 8. In the first iteration, the value for `i1` becomes 3 and `i2` becomes 6 before the next iteration. In the second iteration, `i1` is incremented to 4 and `i2` is decremented to 4 before the next iteration. This is the last iteration because the conditional expression `i1 < i2` is `false` when both `i1` and `i2` are equal.

The output is not `4 3` because `i1` is incrementing, not decrementing, its value in each iteration.

The output is not `8 6` or `6 8` because these are the values for `i2`, not `i1`. `i2` is decrementing in value, so the result would be the output `8 6` if `i2` were printed instead of `i1`.

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 #15 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: 6`

B) `x: 10`

C) `x: 16`

D) `x: 1`

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 #16 of 50

Question ID: 1327800

Given:

```
public class Turkey {
  public class Duck {}
}
```

and:

```
public static void main(String [] args) {
  Object o = /* add code here */;
}
```

Which expression should you add to create an instance of the class `Duck`?

A) `new Turkey().new Duck()`

B) `Turkey.new Duck()`

C) `new Duck()`

D) `new Turkey.Duck()`

Explanation

You should add the expression `new Turkey().new Duck()` to create an instance of the class `Duck`.

Instance inner classes, such as `Duck` in this example, are members of an object of the enclosing type, such as `Turkey` in this example. As a result, they can only be instantiated in a situation that provides the enclosing object. This is normally done by invoking the `new InnerClass()` operation on an instance of the enclosing class. The enclosing instance can be created directly, as in this example, or it can be an existing reference, such as `myTurkeyObj.new Duck()`.

In the context of a static nested class, it is possible to create an instance of the nested class by using the form `new OuterClass.InnerClass()`. However, this approach only works for static nested classes, and not for instance types.

The syntax `Turkey.new Duck()` would work if `Turkey` were a variable of the enclosing type. However, no such variable has been declared in the `main` method at this point. Further, such a variable would be contrary to the standard style guide because it has an initial capital letter and because having a class by that name would risk enormous confusion.

The syntax `new Duck()` would work in a member method defined in the `Turkey` class, but it cannot be used without an implicit this in the scope referring to an instance of `Turkey`. Therefore, it cannot work at this point in the source code.

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

---

Question #17 of 50

Question ID: 1327956

Which of the following is true when declaring a Java module?

A) The Java module must contain at least one class member.

B) The Java module name does not need to be unique in a directory.

C) The Java module can use implied packages without explicit declaration.

D) The Java module must be declared directly under the root directory.

Explanation

When you declare a Java module, it is required that the module be declared under the root directory. If you had packages defined under a directory path such as `javaprogam/finance/usingmymodules`, then your root directory must be the same name as your module. All packages and coding files should be placed under the root directory. The root directly in this example can also be added to a JAR file or to the `classpath` parameter. When declaring your module name, you also should not use _ (underscores) in your naming conventions, but instead and reserve it for packages, classes, variables, and methods.

It is required that your Java module be a unique name and not duplicated within the directory.

Your Java module must explicitly declare the packages that are required for your Java code. This is a key benefit of using modular JDK.

If you are planning on using a specific module, but do not predefine a package that needs to be exported in the module, then you can create an empty module declaration as follows:

```
module javaprogam/finance/usingmymodules {

}
```

Objective:
Java Platform Module System

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

References:

tutorials.jenkov.com > Java Modules > Java Module Basics

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

---

Question #18 of 50

Question ID: 1328011

Given the following:

```
public class Java11 {
   public static void main (String[] args) {
     String s1 = "salty";
     String s2 = new String("salty");
     String s3 = s2;
     if (s1.equals(s2) && s2.equals(s3))
       System.out.println("We are equal!");
     if (s1 == s2 && s2 == s3)
       System.out.println("We are really equal!");
   }
}
```

What is the result when this program is executed?

A)
```
We are equal!
We are really equal!
```

B)
```
We are really equal!
```

C)
```
We are equal!
```

D) Nothing prints.

Explanation

The result is the output `We are equal!`. This is because all three variables have the same character sequence, but do not reference the same `String` object. The `equals` method is overridden to determine equality between `String` objects based on their character sequence. When comparing objects, the == operator determines whether the same object is being referenced. In this case, the variables `s2` and `s3` reference the same `String` object, but the variables `s1` and `s2` do not.

The result will not include the output `We are really equal!`. The == operator determines whether two `String` variables reference the same object. Although the variables `s2` and `s3` do reference the same object, the variables `s1` and `s2` do not.

The result is not nothing prints. All three variables have the same character sequence, so the conditional expression in the first if statement will evaluate to `true` and print `We are equal!`.

Objective:
Controlling Program Flow

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

References:

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

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

---

Question #19 of 50

Question ID: 1328205

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

Which class should you use to track meteorological events?

A) `TemporalUnit`

B) `Instant`

C) `Period`

D) `Duration`

Explanation

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

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

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

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

Objective:

Localization

Sub-Objective:

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

References:

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

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

---

Question #20 of 50

Question ID: 1327911

Consider the following code:

```
interface TestInterface {
    static void methodA() {
        printThis();
        System.out.println("This is method1");
    }
    static void methodB() {
        printThis();
        System.out.println("This is method2");
    }
    private abstract void printThis() {
        System.out.println("Method begins");
        System.out.println("Method works!");
    }
    default void testmethods() {
        methodA();
        methodB();
    }
}
public class TestClass implements TestInterface{
    public static void main(String args[]) {
        TestClass tc = new TestClass();
        tc.testmethods();
    }
}
```

What would be the output?

A) `Method works!`

B) None. The code does not compile.

C) `This is method2`

D) `This is method1`

Explanation

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

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

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

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

Objective:
Java Object-Oriented Approach

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

References:

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

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

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

---

Question #21 of 50

Question ID: 1328199

Given the following code:

```
01  public static void main(String args[]){
02    LocalDate day = LocalDate.of(2020, Month.DECEMBER, 25);
03    day = day.plusHours(24);
```

```
04    System.out.println(day);
05  }
```

The code does not compile. Which line is causing the compilation error?

A) `01`

B) `04`

C) `03`

D) `02`

Explanation

The code does not compile because of line `03`:

```
day = day.plusHours(24);
```

The error occurs because the `LocalDate` object contains a date (25 December 2020), but it does not contain a time value. Therefore, adding time (in this case hours) to a `LocalDate` object will result in a compiler error.

You can add or remove time from a `LocalDate` object using valid date units, such as *`plus/minus`*`Days()`, *`plus/minus`*`Weeks()`, *`plus/minus`*`Months()`, and so on. You can add time or remove time from a `LocalTime` object using valid temporal units, such as hours, minutes, or nanoseconds. You will get a compiler error if you attempt to add time to any incompatible object:

```
LocalDateTime lunchTime = lunchTime.of(1, 00);
lunchTime = lunchTime.plusDays(7); // generates a compiler error
```

The other options are incorrect because those options are valid pieces of code that will not generate compiler errors.

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 > The Java Tutorials > Java Date Time Formatter

---

Question #22 of 50

Question ID: 1327948

Given the following:

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

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

A) `matrix[[1][1][1]] = 42;`

B) `matrix[1][1][1] = 42;`

C) `matrix[1,1,1] = 42;`

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

E) `matrix[0,0,0] = 42;`

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

Explanation

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

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

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

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

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

Objective:
Working with Arrays and Collections

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

References:

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

---

Question #23 of 50

Question ID: 1327778

Given:

```
StringBuilder sb = new StringBuilder("Test");
```

What is the initial capacity of the `StringBuilder` object?

A) 20

B) 4

C) 12

D) 16

Explanation

The initial capacity of the `StringBuilder` object is 20. The capacity of a `StringBuilder` object is 16 if not specified explicitly in the constructor as an `int` argument.

For example, `new StringBuilder(128)` would initialize the `StringBuilder` object at 128 characters. If a `String` object is specified in the constructor, then the initial capacity is 16 plus the length of the `String` object. In this case, the literal string `Test` is 4 characters in length, so the capacity is 16 + 4 or 20.

The initial capacity of the `StringBuilder` object is not 4 or 12. The initial capacity defaults to 16 unless the capacity is specified explicitly in the constructor.

The initial capacity of the `StringBuilder` object is not 16. The initial capacity defaults to 16 if no constructor arguments are specified. If a `String` object is specified, then the initial capacity is 16 plus the length of the `String` object.

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 > The StringBuilder Class

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

---

Question #24 of 50

Question ID: 1328103

Given this class:

```
public class Tests {
  public static int countLower(String s) {
    int count = 0;
    for (char c : s.toCharArray()) {
      if (Character.isLowerCase(c)) {
```

```
        count++;
      }
     }
  return count;
  }
}
```

Which of the following two code statements will compile?

A) `String s = "abc"; Supplier<Integer> cL = (s)->Tests::countLower;`

B) `BiFunction<String, Integer> cH = Tests::countLower;`

C) `Function<String, Integer> cN = Tests::countLower;`

D) `String s = "abcdEFGH"; Supplier<Integer> s1 = s::countLower;`

E) `Supplier<Integer> cR = Tests::countLower("abc");`

F) `ToIntFunction<String> cP = Tests::countLower;`

Explanation

`Function<String, Integer> cN = Tests::countLower;` and `ToIntFunction<String> cP = Tests::countLower;` will compile.

The basic format of a method reference is broadly `<scope>::<method-name>`. To create a method reference from a static method, the `<scope>` part should be the class name. The method that is identified in the `<method-name>` part must then take the same parameter types and count as the interface method being implemented, and the return type of the method must match the return type of the interface method. Thus, `Tests::countLower` is a valid method reference.

To be valid, the method reference must also properly match the interface that it is being used to implement. `Function<String, Integer>` defines a method that takes a `String` argument and returns an `Integer` argument (possibly after autoboxing). This is compatible with the `countLower` method, so the declaration `Function<String, Integer> cN = Tests::countLower;` compiles.

`ToIntFunction<String>` defines a method that takes a `String` and returns an `int`. This matches the `countLower` method exactly, so the declaration `ToIntFunction<String> cP = Tests::countLower;` also compiles.

The method references of `Tests::countLower("abc")` and `(s)->Tests::countLower` are invalid, so statements with those method references would not compile.

The statement `String s = "abc"; Supplier<Integer> cL = (s)->Tests::countLower;` will not compile. Using an object reference, such as the `String` variable `s`, as the scope is possible only when the method is an instance method of the scope object. In this case, code attempts to use a static method in the `Tests` class, but in the context of a `String`, this reference is not possible.

A `Supplier` does not take any arguments, and a `BiFunction` takes two, so neither of the statements that references those function interfaces will compile.

Objective:
Working with Streams and Lambda expressions

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

References:

The Java Tutorials > Learning the Java Language > Classes and Objects > Method References

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

Question #25 of 50

Question ID: 1327759

Given the following:

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

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

What is the result when this program is executed?

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

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

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

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

Explanation

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

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

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

The key difference between primitive variables and reference variables are:

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

The result will not be output with the `val` field of x, z and/or y set to `1`, because modifications to the val field of z will affect the values of x and/or y. Since the `val` field of z is set to `10`, the `val` field of x and y will also be set to `10`.

Objective:
Working with Java Data Types

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

References:

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

Primitive vs Reference Data Types

Question #26 of 50

Question ID: 1328172

Which statement is true about livelocked threads?

A) Thread A and Thread B are said to be livelocked if they are too busy responding to each other to complete their work.

B) Thread A is said to livelocked if it is blocked from a shared resource while Thread B has access.

C) Thread A and Thread B are said to be livelocked if they are blocked forever while waiting for access to the same resource.

D) Thread A is said to be livelocked if it is frequently unable to access a resource shared with Thread B.

Explanation

Thread A and Thread B are said to be livelocked if they are too busy responding to each other to complete their work. In this scenario, Thread A and Thread B are not blocked, but the access mechanism itself is consuming their normal execution.

Thread A and Thread B are not said to be livelocked if they are blocked forever while waiting for access to the same resource. This situation describes deadlocking.

Thread A is not said to be livelocked if it is frequently unable to access a resource shared with Thread B. This situation describes thread starvation.

Thread A is not said to livelocked if it is blocked from a shared resource while Thread B has access. This situation describes thread synchronization.

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 > Liveness

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

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Essential Classes > Concurrency > Starvation and Livelock

---

Question #27 of 50

Question ID: 1327960

Consider the following code:

```
package applic;

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

public class Client {
    public static void main(String[] args) {
        ServiceLoader.load(SpeakerInterface.class)
          .stream()
          .filter((Provider p) -> p.type().getSimpleName().startsWith("Speaker"))
```

```
        //Insert code here
        .findFirst()
        .ifPresent(t -> t.speak());
    }
}
```

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

A) `.provides package1.SpeakerInterface`

B) `.requires modserv`

C) `.exports Provider p`

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

Explanation

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

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

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

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

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

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

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

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

Objective:
Java Platform Module System

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

References:

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

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

---

Question #28 of 50

Question ID: 1328116

Consider the following code:

```
List<String> empNames = new ArrayList<>();
memberNames.add("Josh");
memberNames.add("Troy");
memberNames.add("Ann");
memberNames.add("Aima");
memberNames.add("Robin");
memberNames.add("George");

boolean found = empNames.stream()
  .noneMatch((s) -> s.startsWith("A"));
System.out.println(found);
```

What would be the output?

A) `Ann`

B) `true`

C) `Aima`

D) `false`

Explanation

The output will be `false`. The `noneMatch` method will find two employees whose letters begin with *A*, Ann and Aima, which will return `false`. The `noneMatch` method returns a `boolean` value based on matching none of the elements of the stream to the predicate passed to it. It returns `true` if no elements of the stream successfully match the predicate, and `false` if even a single element does.

The output will not be `true`, because at least one employee's name begins with the letter *A*.

The output will not be `Aima` or `Ann` because the `noneMatch` method returns a `boolean` value.

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 #29 of 50

Question ID: 1327827

Given:

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

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

What is the result?

A)
```
Super
Sub 3
```

B)
```
Sub 1
```

C)
```
Sub 3
```

D)
```
Super
Sub 1
```

Explanation

The result is the following output:

```
Super
Sub 3
```

The code in the main method invokes the `OutputSubClass` constructor with the two `int` parameters, which first invokes the parameterless `OutputSuperClass` constructor. The parameterless `OutputSuperClass` constructor prints `Super`, then the `OutputSubClass` constructor prints `Sub 3`. A subclass constructor automatically invokes the parameterless constructor of the superclass.

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

The result will omit the output `Sub 1`. This overloaded constructor is not invoked based on the two `int` arguments, nor does the third constructor explicitly invoke the first constructor.

Objective:
Java Object-Oriented Approach

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

References:

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

---

Question #30 of 50

Question ID: 1328132

Consider the following code fragments of a Java stream pipeline:

```
01  public class Student {
02    Integer studentID;
03    String firstName;
04    String lastName;
05    String location;
06    LocalDate regDate;
      // implementation omitted
07  }
08  public class StudentFinder {
09    public findAtlantaStudents (List<Student> college) {
10      return
11       .filter( student -> student.isLocated("Atlanta")
12       college.stream()
```

```
13       .collect(Collectors.toList())
14      ;
15    }
16  }
```

You need to verify if the code will execute correctly. What should be the correct line order of code fragments?

A) 12, 11, 13

B) 13, 11, 12

C) 11, 13, 12

D) 11, 12, 13

Explanation

You should place the lines in the order of12, 11, 13, resulting in the following code:

```
12  college.stream()
11  .filter( student -> student.isLocated("Atlanta")
13  .collect(Collectors.toList()
```

Pipeline methods need to follow the order of operations that moves from filtering to sorting, mapping, and finally to collecting. This code creates a `Stream` using the `college.stream()` method, filters the stream by student location, and collects the results to a list.

The Java Collections API provides methods that allow you to process a stream of Java objects from a collection. The main operations in a Stream pipeline and the order in which they occur are:

`filter`
`sorted`
`map`
`collect`

Before you save the results of operations on a stream to a collection, you must first create a stream. A stream in Java is a sequence of information. A stream can consist of anything from bytes and primitive data to objects.

Objective:
Working with Streams and Lambda expressions

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

References:

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

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

Question #31 of 50

Question ID: 1328035

Given the code fragment:

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

How many times is `Loop!` printed?

A) 9

B) 10

C) An infinite number

D) 11

Explanation

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

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

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

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

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

Objective:
Controlling Program Flow

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

References:

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

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

---

Question #32 of 50

Question ID: 1328094

You create a Java program to get the phone numbers of all employees who have more than 10 years of work experience. You first create a stream from a collection of `User` objects.

Which of these code fragments correctly uses a Stream interface and pipeline for the `User` collection?

A)
```
List<User> experienced = users.stream()
    .filter(e -> e.getExperience() > 10)
    .forEach(System.out::println)
    .map(e -> e.getPhone())
    .collect(Collectors.toSet());
```

B)
```
List<User> experienced = users.stream()
    .filter(e -> e.getExperience() > 10)
    .map(e -> e.getPhone())
    .forEach(System.out::println)
    .collect(Collectors.toSet());
```

C)
```
List<User> experienced = users.stream()
    .forEach(System.out::println)
    .filter(e -> e.getExperience() > 10)
    .map(e -> e.getPhone())
    .collect(Collectors.toSet());
```

D)
```
List<User> experienced = users.stream()
    .filter(e -> e.getExperience() > 10)
    .map(e -> e.getPhone())
    .collect(Collectors.toSet())
    .forEach(System.out::println);
```

Explanation

The following code fragment correctly uses a `Stream` interface and pipeline:

```
List<User> experienced = users.stream()
  .filter(e -> e.getExperience() > 10)
  .map(e -> e.getPhone())
  .collect(Collectors.toSet())
  .forEach(System.out::println);
```

The other options are incorrect because `forEach` is a terminal method that is part of the `Iterable` interface. This means that it needs to be at the *end* of a Java pipeline, and not in the middle or front of it.

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 #33 of 50

Question ID: 1327805

Given the following class:

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

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

What is the result?

A) A compile error is produced.

B) `Jessica Martinez (1) is preferred.`

C) `Jessica Martinez (1) is not preferred.`

D) A runtime error is produced.

Explanation

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

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

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

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

Objective:
Java Object-Oriented Approach

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

References:

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

Question #34 of 50

Question ID: 1327925

Examine the following code fragment:

```
public static void printGrades() {
  Map<Double, List<String>> grades = new HashMap<Double, ArrayList<String>>();
  List<String> students;
  students = Arrays.asList("Sam", "Mary", "Ann");
  grades.put(98.5, students);
  students = Arrays.asList("George", "Aima");
  grades.put(88.2, students);
  students = Arrays.asList("Bob", "Christen", "Robin");
  grades.put(76.8, students);
  for (Map.Entry<Double, List<String>> gradeEntry : grades.entrySet()) {
    System.out.printf("Students with %s : %s\n",
      gradeEntry.getKey(), gradeEntry.getValue().toString());
  }
}
```

What is the result?

A)
```
Students with 88.2 : [George, Aima]
Students with 98.5 : [Sam, Mary, Ann]
Students with 76.8 : [Bob, Christen, Robin]
```

B) Code throws a runtime exception.

C)
```
Students with [George, Aima] : 88.2
Students with [Sam, Mary, Ann] : 98.5
Students with [Bob, Christen, Robin] : 76.8
```

D)
```
Students with :
```

E) Code compilation fails.

Explanation

Code compilation fails because of how the `grades` variable is assigned. The type parameter `List<String>` in the `grades` variable declaration does not match `ArrayList<String>` in the `HashMap` instantiation. Type parameters must match if not inferred.

Either of the following code statements should be used in its place to compile:

```
Map<Double, List<String>> grades = new HashMap<Double, List<String>>();
```

```
Map<Double, List<String>> grades = new HashMap<>();
```

The second code statement relies on type inference by using an empty set of type parameters, known as the *diamond*.

The code will not throw a runtime exception. The code will fail compilation, because of how the `grades` variable is assigned.

The code will not print output. The code will fail compilation, because of how the `grades` variable is assigned. If this statement is corrected, then the following output will be printed:

```
Students with 88.2 : [George, Aima]
Students with 98.5 : [Sam, Mary, Ann]
Students with 76.8 : [Bob, Christen, Robin]
```

Objective:
Working with Arrays and Collections

Sub-Objective:
Use generics, including wildcards

References:

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

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Collections > Lesson: Implementations

---

Question #35 of 50

Question ID: 1327773

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

What is the output of this code?

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

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

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

D) `Weights are null`

Explanation

The code outputs the following:

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

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

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

Objective:
Working with Java Data Types

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

References:

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

---

Question #36 of 50

Question ID: 1327776

Given:

```
String str1 = "salt";
```

```
String str2 = "sAlT";
```

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

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

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

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

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

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

Explanation

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

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

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

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

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

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

Objective:
Working with Java Data Types

Sub-Objective:
Handle text using String and StringBuilder classes

References:

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

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

Question #37 of 50

Question ID: 1328076

Which is the best scenario for implementing the `Supplier` interface in a lambda expression?

A) Preparing a complex formatted message with multiple elements for a log file, when the log message may not be required

B) Returning values supplied by a database, when the database is unavailable

C) Creating values for use in a `Stream`, when each value is derived from its predecessor

D) Supplying values which are computed from a formula and an input value to that formula

Explanation

The best scenario for implementing the `Supplier` interface is preparing a complex formatted message with multiple elements for a log file, when the log message may not be required. Preparing complex formatting for a log message, only to have the entire effort wasted because the logging filter discards the message, is a problem for which a lambda implementing `Supplier` is particularly well-suited. The business of formatting can be deferred until and unless the logging system needs the message, reducing wasted computation while providing clean, easy to read source code. The Java built-in logging API is implemented in this way.

Here is an example that uses a lambda expression based on this scenario:

```
LOG.file(
  ()->String.format("%2s was reported at %1$tb %1$te, %1$tY",
  LocalDateTime.now(),
  ex.getLocalizedMessage())
);
```

The `Supplier` interface is not best suited for creating values for use in a `Stream`, when each value is derived from its predecessor. The `Supplier` interface defines a single method, called `get()`, which takes no arguments and returns a value. Therefore, it is generally not suited to situations where the value being computed depends on some pre-existing value. This scenario better suits the `Function` interface, which defines a method called `apply` that takes a value and returns another. This pattern of use is implemented in the `Stream.iterate` method.

The `Supplier` interface is not best suited for returning values supplied by a database when the database is unavailable. Fetching data from a database might be represented as a `Supplier` if the state necessary to connect to the database and maintain the progress through a result set were embedded in the object that implements the interface. However, a lambda expression rarely implements such complex logic.

The `Supplier` interface is not best suited for supplying values which are computed from a formula and an input value to that formula. Values computed from a formula that has an input value are best represented using `Function` operations because the function directly supports an input and an output.

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 > Supplier

Java Platform Standard Edition 11 > API > java.util.logging > Logger

---

Question #38 of 50

Question ID: 1327812

Given the following code line:

```
printToConsole(1.2);
```

Which overloaded method is invoked?

A)
```
void printToConsole(Double dblObj) {
   System.out.println("My double object is " +
dblObj.toString());
}
```

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

C)
```
void printToConsole(float fl) {
   System.out.println("My float is " + fl);
}
```

D)
```
void printToConsole(Integer intObj) {
   System.out.println("My integer object is " +
intObj.toString());
}
```

Explanation

The following overloaded method is invoked:

```
void printToConsole(Double dblObj) {
   System.out.println("My double object is " + dblObj.toString());
}
```

This method is invoked because the default primitive type for a literal fractional value is `double`, and this type is automatically boxed as a `Double` object. Thus, the overloaded method with a `Double` parameter is invoked. 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 a `float` parameter is not invoked. The default primitive type for a literal fraction value is `double`, not `float`. This overloaded method would be executed if the invocation contains a valid `float` value such as 1 or 1.2f.

The overloaded method with an `Integer` parameter is never invoked. The default primitive type for a literal fraction value is `double`, not `int`. Neither the `int` nor its box class `Integer` support fractional amounts. They support only whole numbers. This overloaded method is not invoked because the method with the `float` parameter will execute when an `int` value or variable is specified as an argument.

The overloaded method with an `Object` parameter is not invoked. 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.

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 #39 of 50

Question ID: 1327916

Given:

```
(a)-> a.intValue() + 10
```

Which two functional interfaces could this lambda expression implement? (Choose two.)

A) `Consumer`

B) `LongFunction`

C) `UnaryOperator`

D) `Supplier`

E) `ToIntFunction`

Explanation

The two functional interfaces that could be implemented are `UnaryOperator` and `ToIntFunction`. The `lambda (a)-> a.intValue() + 10` takes a single argument and returns a numeric (integer type) result. The single argument must be something that supports the `intValue` operation, which must therefore be an object, not a primitive.

`UnaryOperator<Number>` or `UnaryOperator<Integer>` both provide for a single argument that supports the `intValue` method. The returned int expression (that is, the result of adding the two int values) is supported by autoboxing. Therefore, `UnaryOperator` is a correct answer.

`ToIntFunction<E>` specifically takes a single argument of type `E`, which must be either `Number` or `Integer`, and returns a primitive `int` value. Therefore, that option is correct and does not require autoboxing for success.

The interface cannot be `Consumer` because the method of that interface does not return anything.

The interface cannot be `Supplier` because the method of that interface does not accept any arguments.

The interface cannot be a `LongFunction` because its argument is a primitive long value. It would not be possible to invoke `intValue` (or any other method) on this interface.

Objective:
Java Object-Oriented Approach

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

References:

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

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

Java Platform Standard Edition 11 > API > java.util.function > Interface ToIntFunction<T>

Java Language Specification > Java SE 11 Edition >Conversions and Contexts > Kinds of Conversion > Boxing Conversion

---

Question #40 of 50

Question ID: 1327833

Which access modifier will permit access to class members only from the same package?

A) `protected`

B) `private`

C) no modifier

D) `public`

Explanation

Default access, by specifying no modifier, will restrict access to class members only in the same package. This access level is referred to as *package-private*. Subclasses and any other class in the same package will have access to these members.

The access modifier `protected` will not permit access to members from all classes in the same package. This modifier will restrict access to subclasses within the same package.

The access modifier `private` will not permit access to classes in the same package. This modifier will restrict access to members only within the same class.

The access modifier `public` will not restrict access to members only from classes in the same package. This modifier will allow access to members from any class.

Objective:
Java Object-Oriented Approach

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

References:

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

---

Question #41 of 50

Question ID: 1328002

Consider the following annotation:

```
public class ModernRevelations {
// INSERT ANNOTATION HERE
 public int numberOfTheB() {
  return 666;

 }

 public int neighborOfTheB(){
  return 665;
 }
}
```

What type of annotation will you insert for the method `numberOfTheB` to indicate is no longer in use?

A) `@Override`

B) `@Deprecated`

C) `@SafeVarargs`

D) `@SupressWarnings`

Explanation

`@Deprecated` is a marker annotation indicating that the associated declaration is has now been replaced with a newer one.

`@Override` is an incorrect option. This is a marker annotation only to be used with methods that override methods from the parent class. It helps ensure methods are overridden and not just overloaded.

`@SupressWarnings` is an incorrect option. This specifies warnings in string form that the compiler must ignore.

`@Safevarargs` is an incorrect option. This is an annotation to demarcate code that does not perform any unsafe operations on its `varargs` parameter.

Marker annotations are used to *mark* declarations. They do not contain any data or members. The `@Override` annotation is an example of a marker annotation.

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

Objective:
Annotations

Sub-Objective:
Create, apply, and process annotations

References:

Oracle Technology Network > Java SE Documentation > Annotations

---

Question #42 of 50

Question ID: 1328052

Given:

```
public class ExceptionFun {
   public ExceptionFun(Object obj) {
     if (obj == null)
       throw new IOException("Provide an object!");
     System.out.println(obj + " created!");
   }
   public static void createObject() {
     try {
```

```
      ExceptionFun obj = new ExceptionFun(null);
     } finally {
      System.out.println("Was the object created?");
     }
   }
   public static void main(String[] args) {
     createObject();
   }
}
```

What is the result?

A) `ExceptionFun is created!`
`Was the object created?`

B) Compilation fails.

C) `is created!`
`Was the object created?`

D) `ExceptionFun is created!`

E) `is created!`

F) An exception is thrown at run time.

Explanation

The result is that compilation fails because `IOException` is a checked exception. The compiler verifies that checked exceptions are handled by being specified or caught in code. Checked exceptions are only `Exception` and its subclasses, excluding `RuntimeException` and its subclasses.

The result is not output because the code fails compilation. If there were no check in the constructor that threw an exception, then the result would be the following output:

```
is created!
Was the object created?
```

This is because the code in the `finally` block is executed whether or not an exception is thrown.

The result is not a runtime exception because `IOException` must be specified or caught in code.

Objective:
Exception Handling

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

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Essential Classes > Exceptions > Putting it All Together

Question #43 of 50

Question ID: 1328069

You developed software that maintains a list of clients, where a `Client` object represents a customer or similar business partner. You are working on the following method to take that list and produce an output list of clients meeting certain criteria:

```
public static <E> List<E> filterList(
  List<E> input,
  /* Point A */ condition) {
    List<E> output = new ArrayList<>();
    for (E e : input) {
      if (/* Point B */) {
        output.add(e);
      }
    }
  return output;
}
```

Which actions should you perform to complete the `filterList` method?

A) Define the parameter at `Point A` as `ToBooleanFunction<E>`, then complete `Point B` with `condition.apply(e)`

B) Define a new interface describing a test that operates on an `E` and returns a `boolean`, use that interface type to define the parameter type at `Point A`, and invoke the test defined by that interface on each item at `Point B`

C) Use an interface from the `java.util.function` package to define the parameter type at `Point A`, then invoke `condition.test(e)` at `Point B`

D) Define the parameter at `Point A` as `Function<E, Boolean>`, then complete `Point B` with `condition.test(e)`

Explanation

You should use an interface from the `java.util.function` package to define the parameter type at `Point A`, then invoke `condition.test(e)` at `Point B`. Tests of this type are the reason that the `Predicate` interface exists. The `Predicate` interface defines a single generic method that takes an argument of the generic type and returns `boolean`. The method is called `test`.

The `filterList` method could be modified as follows:

```
public static <E> List<E> filterList(
  List<E> input,
  /* Point A */ Predicate<E> condition) {
    List<E> output = new ArrayList<>();
    for (E e : input) {
      if (* Point B */ condition.test(e)) {
        output.add(e);
      }
    }
  return output;
}
```

Defining a new interface is completely unnecessary because the JDK defines several standard functional interfaces in the `java.util.function` package.

Using a `Function<E, Boolean>` would potentially be workable, but the method defined in the `Function` interface is `apply`, not `test`.

Using a `ToBooleanFunction` is impossible because there no such interface exists in the JDK. The Predicate interface fulfils the purpose of taking an object and returning a `boolean`.

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 > Package java.util.function

The Java Tutorials > Learning the Java Language > Classes and Objects > Approach 6: Use Standard Functional Interfaces with Lambda Expressions

---

Question #44 of 50

Question ID: 1328198

You have the following code:

```
LocalTime thisMoment = LocalTime.of(12, 10, 30);
LocalDate thisDay = LocalDate.of(2015, Month.MAY, 27);
LocalDateTime today = LocalDateTime.of(thisDay, thisMoment);
System.out.println(today.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));
```

What is the result of compiling and running the code?

A) The code generates a compiler error.

B) `12:10:30`

C) `2015-05-27`

D) `12:10`

E) `2015-05-27T12:10:30`

Explanation

The correct output is:

`2015-05-27T12:10:30`

The output includes all date and time information.

The output would not be `12:10:30`. This would be the output if using the `DateTimeFormatter` constant `ISO_LOCAL_TIME`.

The output would not be `2015-05-27`. This would be the output if using the `DateTimeFormatter` constant `ISO_LOCAL_DATE`.

The output would not be `12:10`, because it does not include the second-of-day value.

The code will not generate a compiler error, because it is syntactically correct.

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 > The Java Tutorials > Java Date Time Formatter

---

Question #45 of 50

Question ID: 1328123

Consider the following code for a user database:

```
public class User {
  Integer userID;
  String firstName;
  String lastName;
```

```
  LocalDate dateofHire;
}

public class CreateUsersList {
  List<User> users;
  public void setup() {
    users = new ArrayList<>();
    users.add(new User(001, "Sean", "Benjamin", LocalDate.of(1990, Month.MAY, 20)));
    users.add(new User(002, "Sally", "Donner", LocalDate.of(2010, Month.JANUARY, 10)));
    users.add(new User(003, "Richard", "Anderson", LocalDate.of(2004, Month.JULY, 10)));
    users.add(new User(004, "Jessica", "Winters", LocalDate.of(2006, Month.JULY, 20)));
    users.add(new User(005, "Jonathan", "Steele", LocalDate.of(1990, Month.MAY, 20)));
    users.add(new User(006, "Cindy", "Summer", LocalDate.of(2008, Month.MAY, 15)));
    //Insert code here
  }
}
```

Which code statements that will enable you to sort this list by employee number using Java Stream API? (Choose all that apply.)

A)
```
users.stream().map(userIDSort)
    .forEach(s -> System.out.println(s));
```

B)
```
Comparator<User> userIDSort = (u1, u2) -> Integer.compare(
  u1.returnUserID(), u2.returnUserID());
```

C)
```
List<User> userIDSort = (u1, u2) -> Integer.compare(
    u1.returnUserID(), u2.returnUserID());
```

D)
```
users.stream().sorted(userIDSort)
  .forEach(s -> System.out.println(s));
```

Explanation

The correct options are:

```
Comparator<User> userIDSort = (u1, u2) -> Integer.compare(
  u1.returnUserID(), u2.returnUserID());
```

and

```
users.stream().sorted(userIDSort)
  .forEach(s -> System.out.println(s));
```

The `sorted` method returns a stream made of the elements of the stream on which this method was run, but sorted in a natural order.

The following option is incorrect because it implements `List`:

```
List<User> userIDSort = (u1, u2) ->  Integer.compare(  u1.returnUserID(), u2.returnUserID());
```

You need to create a `Comparator` interface that implements the comparison needed to perform the sorting.

The following option is incorrect because it uses the `map` method:

```
users.stream().map(userIDSort)
  .forEach(s -> System.out.println(s));
```

The map method does not organize the order of elements, but translates or modifies them.

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 #46 of 50

Question ID: 1328051

Which statement is true about exception propagation?

A) If an exception is caught, then that exception is propagated down the call stack automatically.

B) If an exception is thrown in a `try` block, the code continues its normal execution.

C) If an exception is specified, then that exception will not be propagated up the call stack.

D) Whether an exception is thrown or not thrown in a `try` block, the code in the `finally` block will execute.

Explanation

Whether an exception is thrown or not in a `try` block, the code in the `finally` block will execute. If no exception is thrown, then the code in the `try` block will execute normally and then the code in the `finally` block will execute. If an exception is thrown, then execution will move from the `try` block to any matching `catch` blocks for that exception, and then continue execution in the `finally` block.

If an exception is thrown in a `try` block, the code will not continue its normal execution. Execution will move into an associated `catch` block that matches the exception and/or move into a `finally` block if associated with the `try` block.

If an exception is specified, then the exception will be propagated up the call stack. Invokers of the method must either catch or specify the exception until it reaches the JVM. The JVM will shut down the process.

If an exception is caught, then that exception is not propagated *down* the call stack. Catching an exception allows execution to continue execution outside of its associated try block. Propagation of exceptions occurs *up* the call stack, not *down* the call stack.

Objective:
Exception Handling

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

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Essential Classes > Exceptions > Putting it All Together

---

Question #47 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) `java`

B) `javac -d`

C) `javac -s`

D) `javac -g`

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 #48 of 50

Question ID: 1328016

Given the following:

```
if ( x < 10) {
   if (x > 0)
     System.out.print("She");
   else
     System.out.print("Sally");
   if (x < 5)
     System.out.print(" sells seashells");
   if ( x > 10)
     System.out.print(" will sell all her seashore shells");
   else if (x < 15)
     System.out.print(" by the");
   else if (x < 20)
     System.out.print(" on the");
   if ( x < 10)
     System.out.print(" seashore");
   else
     System.out.print(" seashell shore");
} else {
   System.out.print("Of that I'm sure");
}
```

Which value for the variable x will output `Sally sells seashells by the seashore`?

A) 10

B) 1

C) 0

D) 5

Explanation

The value 0 for the `x` variable will output `Sally sells seashells by the seashore`. To meet the criteria of the first `if` statement, `x` must be less than `10`. To output `Sally`, `x` must be less than or equal to `0` to reach the second `else` statement. To output `sells seashells`, `x` must be less than `5` to meet the criteria of the third `if` statement. To output `by the`, `x` must be less than 15 but not greater than 10 to reach and meet the criteria of the first `else if` statement. Finally, `x` must be less than `10` to meet the criteria of the fifth `if` statement and output `seashore`.

The values 1 and 5 for `x` will not output `Sally`, but will output `She`. If `x` is set to 1, then the output will be `She sells seashells by the seashore`. If `x` is set to 5, then the output will be `She by the seashore`.

The value 10 for `x` will output `Of that I'm sure`, not `Sally`. This is because the criteria of the first `if` statement is not met.

Objective:
Controlling Program Flow

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

References:

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

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

---

Question #49 of 50

Question ID: 1328128

Given:

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

What is the result?

A) `<A,B>`

B) `A,B`

C) `>A,B<`

D) Compilation fails at line `n1`.

E) `AB`

Explanation

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

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

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

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

There are no compilation errors at line `n1`.

Objective:
Working with Streams and Lambda expressions

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

References:

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

---

Question #50 of 50

Question ID: 1327799

Given:

```
01  interface Reportable {
02    void report();
03  }
04  class CommandLine {
05    public static void print(Reportable r) {
06      r.report();
07    }
08    public static void main(String[] args) {
09      print(new Reportable() {
10        public void report() {
11          System.out.println("anonymous");
```

```
12         }
13       });
14     }
15  }
```

What is the result?

A) Compilation fails due to an error on line 09.

B) Compilation fails due to an error on line 06.

C) Compilation fails due to an error on line 10.

D) `null`

E) `anonymous`

Explanation

The result is the output `anonymous`. The `Reportable` interface is implemented by an anonymous inner class on lines 09-13. Because an anonymous inner class does not require a name, you only need to reference the interface name that is implemented.

The result is not the output `null`. The report method is implemented in lines 10-12 to output `anonymous`.

Compilation does not fail due to an error on line 06. The `Reportable` interface declares a report method, so it is a valid reference type on which to invoke report.

Compilation does not fail due to an error on line 09. Although an interface cannot be instantiated, this code declares an anonymous inner class that implements `Reportable`. Anonymous inner classes are allowed in the Java language.

Compilation does not fail due to an error on line 10. The report method signature matches the method declared in the `Reportable` interface.

Objective:
Java Object-Oriented Approach

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

References:

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

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

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