Quick Quiz September 1, 2022

Test ID: 222268261

Question #1 of 50

Question ID: 1328129

Given:

```
Stream.of(
  "as", "as", "as",
  "by", "by", "by", "by", "by",
  "in", "in",
  "of", "of", "of", "of"
  )
  .collect(
  Collectors.groupingBy(s -> s, Collectors.counting())
  )
  .forEach((k, v) -> System.out.println(k + " : " + v));
```

What is the output?

✗ **A)**
```
as : [as, as, as]
by : [by, by, by, by, by]
in : [in, in]
of : [of, of, of, of]
```

✗ **B)**
```
as : as : as
in : in
of : of : of : of
by : by : by : by : by
```

✓ **C)**
```
as : 3
in : 2
of : 4
by : 5
```

✗ **D)**
```
as : [as, as, as]
in : [in, in]
of : [of, of, of, of]
by : [by, by, by, by, by]
```

✗ **E)**
```
2 : in, in
3 : as, as, as
4 : of, of, of, of
5 : by, by, by, by, by
```

Explanation

The output is the following:

```
as : 3
in : 2
of : 4
by : 5
```

The `groupingBy` collector builds a `Map`. The key for the `Map` is provided from the lambda expression, known as the *key extractor*, used on items coming down the stream. In this case, the key extractor is `s->s`, which means that each item is its own key.

In the simplest form of the `groupingBy` collector, the items in the stream are added to a `List` as the value portion of the `Map`. They are added to the `List` maintained as the value looked up by each item's key. This describes a `Map` that had with key as matched with same value, because each value is its own key, so the list would be `[as, as, as]`. In this code fragment, a second collector, called a *downstream* collector, is also provided. This is `Collectors.counting()`. This collector processes the items as they are added to the `List`, resulting in a single value. This value is the number of items matching the given key. Since the string as repeated three times, the value is 3, and so on.

The following is not the resulting output:

```
as : [as, as, as]
in : [in, in]
of : [of, of, of, of]
by : [by, by, by, by, by]
```

This would have been the output if the downstream collector `Collectors.counting()` were not provided as an argument. Without the downstream collector, a list of values would be returned for each key in the resulting `Map`.

The other output are not plausible based on the given code fragment.

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 #2 of 50

Question ID: 1327924

Which three code statements correctly instantiate and assign a generic collection?

✓ **A)** `Map<Integer, List<String>> maplist = new HashMap<>();`

✓ **B)** `Map<String, Integer> map = new HashMap<>();`

✗ **C)** `Map<<String>,<Integer>> map = new HashMap<,>();`

✓ **D)** `List<String> list = new ArrayList<>();`

✗ **E)** `Map<Integer, List<String>> maplist = new HashMap<Integer, ArrayList<String>>();`

✗ **F)** `List<> list = new ArrayList<String>();`

Explanation

The following code statements correctly instantiate and assign a generic collection:

```
List<String> list = new ArrayList<>();
  //Same as List<String> list = new ArrayList<String>();
Map<String, Integer> map = new HashMap<>();
  //Same as Map<String, Integer> map = new HashMap<String, Integer>();
Map<Integer, List<String>> maplist = new HashMap<>();
//Same as Map<Integer, List<String>> maplist = new HashMap<Integer, List<String>>();
```

These code statements rely on type inference by using an empty set of type parameters, known as the *diamond*. On line 1, the `list` variable is declared as a generic list for `String` element, so that the `String` type parameter is inferred when instantiating the generic `ArrayList` class. On line 2, the `map` variable is declared as a generic map with `String` keys and `Integer` values, so that the `String` and `Integer` type parameters are inferred when instantiating the generic `HashMap` class. On line 3, the `map` variable is declared as a generic map with `Integer` keys and `String` list values, so that the `Integer` and `String` list type parameters are inferred when instantiating the generic `HashMap` class.

The diamond cannot be used in variable declarations such as `list`, only when invoking constructors and other methods when the type parameters have been declared previously.

The parameter set cannot include a comma without type parameters when assigning the `map` variable. The diamond is an empty set of parameters without any characters between the angle brackets.

The type parameter `List<String>` in the `mapList` variable declaration does not match `ArrayList<String>` in the `HashMap` instantiation. 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 > Type Inference

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

---

Question #3 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 == 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.equals(str2) )
        System.out.println("str1 and str2 are equal");
```

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

✓ **E)**
```
if (str1.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 #4 of 50

Question ID: 1327914

What statement is true of a functional interface?

- ✗ **A)** Must be defined as extending `FunctionalInterface`
- ✗ **B)** Must be annotated with `@FunctionalInterface`
- ✓ **C) Must contain a single abstract method or inherit it from a parent interface**
- ✗ **D)** Must contain one abstract method that is not inherited from a parent interface
- ✗ **E)** Must be defined as implementing `FunctionalInterface`

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 #5 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)**
```
List<User> userIDSort = (u1, u2) -> Integer.compare(
        u1.returnUserID(), u2.returnUserID());
```

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

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

✗ **D)**
```
users.stream().map(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 #6 of 50

Question ID: 1328085

Which of the following operations on collections is the more efficient in terms of memory usage?

✗ **A)**
```
Predicate<Integer> p = i -> i == 2015;
p.test(1974);
```

✗ **B)**
```
List<Double> myList = new ArrayList<>();
list.add(2015);
```

✓ **C)**
```
IntPredicate intpred = i -> i == 2015;
ip.test(2015);
```

✗ **D)**
```
List<Float> myList = new ArrayList<>();
```

Explanation

The most memory-efficient code is the following:

```
IntPredicate intpred = i -> i == 2015;
ip.test(2015);
```

This code uses primitive forms of functional interfaces, in this case `IntPredicate`.

The other options all require boxing and unboxing to be done either manually by the programmer or automatically by Java. Both methods cause a considerable memory usage overhead that slows the program down.

Objective:
Working with Streams and Lambda expressions

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

References:

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

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

---

Question #7 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: 10`

✗ **B)** `x: 16`

✓ **C) x: 1**

✗ **D)** `x: 6`

Explanation

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

```
x: 1
```

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

The key differences between primitive variables and reference variables are:

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

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

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

Objective:
Working with Java Data Types

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

References:

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

Primitive vs Reference Data Types

Question #8 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)** 11
- ✗ **B)** 10
- ✗ **C) An infinite number**
- ✗ **D)** 9

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 #9 of 50

Question ID: 1327906

You define the following interface to handle photographic items:

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

```
  }
}
```

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

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

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

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

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

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

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

Explanation

You should declare the `makePhotographs` method as follows:

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

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

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

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

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

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

Objective:
Java Object-Oriented Approach

Sub-Objective:

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

References:

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

---

Question #10 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 and write permissions on the `protected.enc` file.
- ✗ **B)** A runtime exception is thrown.
- ✗ **C)** The user jhester controls permission assignment on the `protected.enc` file.
- ✗ **D)** **The user jhester has read, write, and execute permissions on the `protected.enc` file.**
- ✓ **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 #11 of 50

Question ID: 1328041

Given:

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

Which insertion will allow the code to compile?

✓ **A)** `IOException`

✗ **B)** `IOError`

✗ **C)** `FileSystemNotFoundException`

✗ **D)** `Error`

✗ **E)** `FileNotFoundException`

Explanation

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

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

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

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

Objective:
Exception Handling

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

References:

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

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

---

Question #12 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)** `String s->"Message is: " + s`

✓ **B)** `s->s`

✗ **C)** `(final s) -> s + "."`

✗ **D)** `s->s.length()`

✗ **E)** `()->f.apply("")`

Explanation

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

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

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

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

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

Objective:
Working with Streams and Lambda expressions

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

References:

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

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

---

Question #13 of 50

Question ID: 1327935

Given the following code fragment:

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

What is the result?

- ✗ **A)** Code throws a runtime exception.
- ✗ **B)** `Ruth`
- ✗ **C)** `Todd`

✗ **D)** `Jason`

✓ **E) Code compilation fails.**

Explanation

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

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

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

Objective:
Working with Arrays and Collections

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

References:

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

Question #14 of 50

Question ID: 1328185

Consider the following code:

```
class Client implements Serializable {
  double account = 90876543;
  public String toString() {
    return String.valueOf(account);
  }
}
```

How would you ensure that sensitive data like an account number is protected during serialization of this class?

✗ **A)** Use `readObject`

✓ **B)** Using a `serialVersionUID`

✗ **C)** Use `clone`

✗ **D) Use `readObjectNoData`**

Explanation

You should use a `serialVersionUID`. The following code illustrates the use of a serial version UID for securing the serializable class:

```
class Client implements Serializable, Externalizable {
  private static final long serialVersionUID = 21L;
  transient double account = 90876543;
  public String toString() {
    return String.valueOf(numOfWeapons);
  }
}
```

`clone`, `readObject`, and `readObjectNoData` are incorrect because they do not aid in securing a serializable class. These methods should not be called from within a class constructor for security reasons because each of them is an overridable method.

To protect sensitive fields in serializable classes, the following need to be done:

- Fields need to be made `transient`
- The `Externalizable` interface must be implemented
- The `serialPeristentFields` array fields need to be defined correctly
- The `writeReplace` method must be implemented as it replaces the instance with a proxy
- The `writeObject` method must be implemented

The process of serialization circumvents the field access control aspect of Java that provides security. For this reason, serialization must be done very carefully. Deserialization must be avoided altogether when dealing with untrusted data.

When a class is made serializable, a `public` interface is invariably created for all fields of that class. A `public` constructor is added to a class when it is serialized. Additionally, lambdas and functional interfaces come with security issues and so should be used with care.

Objective:
Secure Coding in Java SE Application

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

References:

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

Question #15 of 50

Question ID: 1327910

Consider the following Java interface:

```
public interface music {
    default void opera(String note) {
        playMusic(note, "THIS IS OPERA");
    }
    default void rocknroll(String note) {
        playMusic(note, "THIS IS ROCKNROLL");
    }
    default void jazz(String note) {
        playMusic(note, "THIS IS JAZZ");
    }
    default void country(String note) {
        playMusic(note, "THIS IS COUNTRY");
    }
    private void playMusic(String note, String musicType);
}
```

What needs to be added to make this code compile correctly?

- ✗ **A)** Add the keyword `public` to the `playMusic()` method
- ✗ **B)** Add the keyword `abstract` to the `playMusic()` method
- ✗ **C) Implement the `playMusic()` method in a derived class**
- ✓ **D)** Implement the `private` method `playMusic()`in the `music` interface

Explanation

You should provide an implementation of the `private` method `playMusic()` in the `music` interface. 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.

Adding the keyword `abstract` to the `playMusic()` method is incorrect because the `abstract` and `private` keywords cannot be used together in a Java interface. The `private` and `abstract` both have separate uses in Java. When a method is deemed `private` it indicates that the method is accessible only *inside* the interface and cannot be accessed from this interface by other interfaces or classes. The purpose of this is to expose only the method implementations that are intended to be exposed. However, when a method is deemed `abstract`, it needs to be inherited and overridden by subclasses.

Adding the keyword `public` to the `playMusic()` method is incorrect because a `private` method inside an interface cannot be given a lesser access modifier. It is not meant to be inherited by a subclass that may implement the interface. When a method inside an interface is deemed `private`, it indicates that the method is accessible only *inside* the interface and cannot be accessed from this interface by other interfaces or classes. The purpose of this is to expose only the method implementations that are intended to be exposed. Additionally, method declarations inside interfaces with a body represented by a semicolon are `public` and `abstract` implicitly.

Implementing the `playMusic()` method in a derived class is incorrect because `playMusic()` is a `private` method inside an interface, and therefore cannot be inherited by a subclass. `private` methods inside interfaces can only be accessed within the interface and cannot be inherited by another interface or class.

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

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

---

Question #16 of 50

Question ID: 1328025

Given the following:

```
int x = 0;
```

Which code fragment increments x to 10?

✗ **A)** `while (x < 10 ? 1 : 0) { x++; }`

✗ **B)** `while (x < 11) { x++; }`

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

✓ **D)** **`while (x < 10) { x++; }`**

Explanation

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

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

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

Objective:

Controlling Program Flow

Sub-Objective:

Create and use loops, if/else, and switch statements

References:

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

---

Question #17 of 50

Question ID: 1327796

Which two keywords are required to declare a constant?

✗ **A)** `public`

✗ **B)** `const`

✗ **C)** `volatile`

✓ **D)** **`final`**

✓ **E)** **`static`**

✗ **F)** `abstract`

Explanation

The two keywords required to declare a constant are `final` and `static`. The `final` keyword indicates the field value cannot change, while the `static` keyword limits initialization to the class, rather than object initialization.

Once set, a constant cannot be reassigned or the code will fail compilation. The common way to set a constant variable is by assignment in the declaration, but the value can also be assigned in a static initializer block. The following code demonstrates these two ways of setting a constant:

```
class ConstClass {
  final static int ULTIMATE_ANSWER = 42;
  final static double AVOGADRO_NUMBER;
  static {
  AVOGADRO_NUMBER = 6.0221415e23;
  }
}
```

The `const` keyword is not required to declare a constant. Although const is reserved as a keyword in Java, it is not used by the language. In C++, `const` is used for declaring read-only variables or pointers.

The `public` keyword is not required to declare a constant. `public` can be used to provide the broadest access to a constant, but a constant does not require this keyword.

The `volatile` keyword is not required to declare a constant. This keyword is used to manage concurrency when a field value changes often, so that write operations by threads must precede any read operations performed by other threads.

The `abstract` keyword is not required to declare a constant. This keyword is used to describe a class that cannot be instantiated or a method that must be overridden by a subclass.

The `final` keyword can be used on a class, field, or method, and specifies that a class cannot be extended, a field is a constant value, or a method cannot be overridden. You cannot inherit from a class marked as `final`. You should declare all methods called from constructors as `final`. Otherwise, a non-final method called by a constructor may be overridden by a subclass, which may cause unwanted behavior in the 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:

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

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Interfaces and Inheritance > Writing Final Classes and Members

---

Question #18 of 50

Question ID: 1327829

Which statement is true about constructor overloading?

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

Explanation

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

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

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

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

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

Objective:
Java Object-Oriented Approach

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

References:

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

---

Question #19 of 50

Question ID: 1328136

Which feature(s) are provided by the parallel streams mechanism? (Choose all that apply.)

- ✓ **A)** Utilization of multiple CPUs on a single solution
- ✓ **B)** Minimum loss of concurrency due to thread contention
- ✓ **C)** Avoidance of issues caused by memory consistency
- ✗ **D)** Identification of efficient algorithms for parallelization
- ✗ **E)** Automatic addition of synchronized blocks to binary code

Explanation

The parallel streams mechanism utilizes multiple CPUs on a single solution, minimizing loss of concurrency and avoiding issues related to memory consistency. The parallel streams mechanism is intended to provide a simple-to-use yet reasonably efficient concurrency model that will allow software to benefit from modern hardware-parallel systems without requiring the exceptional skill levels traditionally necessary to create correct code in concurrent situations. Several problems must be addressed.

Java provides rules which specify how changes made to memory by one thread become visible to another thread, which is known as its *memory model*. It is carefully designed to permit high levels of optimization on very different

execution platforms. However, partly as a result of this, and partly because thread interactions involving mutating data are always prone to misunderstanding, the model is hard for normal programmers to reason about reliably. The parallel streams API seeks to mitigate this issue by specifying a predictable, easy to implement model for inter-thread data sharing. Further, for a great many situations, the sharing (which typically occurs during reduction or collection) may be handled by pre-written library code.

The programming model of parallel streams seeks to avoid the need for synchronization in any of the programmer-supplied code. The implementations also seek to minimize such behaviors, which are a common cause of reduced scalability. Because of this, there is no expectation of synchronization (either hand written or automatic) in code written to run on the parallel streams system. The parallel streams programming model generally seeks to minimize the use of synchronization, precisely because that technique causes a loss of concurrency, and that loss of concurrency causes a loss of scalability.

The parallel streams mechanism does not provide automatic addition of synchronized blocks to binary code. There is no expectation of synchronization whatsoever.

Unfortunately, the parallel streams mechanism does not identify efficient algorithms for parallelization. Although the parallel streams mechanism can be correctly applied to a wide variety of stream-processing situations, there is a non-trivial overhead involved in managing the system. That overhead is largely built into the library APIs and implementation and is nearly impossible to estimate simply by looking at the programmer supplied code. Some programmer-supplied code is a poor return on the CPU effort spent executing the programmer's algorithm. Such code typically does not benefit from the parallel streams mechanism. However, while qualitative guidance can be given as to the properties of code that make parallel streams more, or less, likely to be effective, no automated system can yet make that determination.

Objective:
Working with Streams and Lambda expressions

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

References:

The Java Tutorials > Collections > Aggregate Operations > Parallelism

---

Question #20 of 50

Question ID: 1327993

Given the following incorrect program:

```
public class JDBCApp {
  public static void main(String[] args) throws Exception {
    Connection con =
DriverManager.getConnection("jdbc:mysql://localhost:3306/sakila","root","vi$t@"))
    //Perform code
```

```
  }
}
```

Which change will make the program work correctly?

- ✗ **A)** Add the `com.mysql.jdbc.Driver` class to the class path of the program.
- ✗ **B)** **Use the `Class.forName` method to load the `com.mysql.jdbc.Driver` class.**
- ✓ **C)** Add the JAR file for the MySQL JDBC driver to the class path of the program.
- ✗ **D)** Use the `DriverManager.getDriver` method to load the JAR file for the MySQL JDBC driver.

Explanation

To make the program work correctly, you should add the JAR file for the MySQL JDBC driver to the class path of the program. Without adding the JDBC driver to the class path, the program will throw a `SQLException` with the message No suitable driver found for `jdbc:mysql://localhost:3306/sakila`. The classes provided for the MySQL database implement the JDBC API and are automatically loaded in JDBC 4.

You do not need to use the `Class.forName` method to load a database driver in JDBC 4. This method retrieves a class reference based on the fully qualified class name. In previous version of JDBC, this method had to be invoked to load JDBC drivers before establishing the database connection.

You do not need to use the `DriverManager.getDriver` method to load a JAR file for a database driver because database drivers are automatically loaded in JDBC 4 based on connection URLs.

You should not add the `com.mysql.jdbc.Driver` class to the class path of the program, because this includes only the `Driver` interface implementation, not all classes associated with the JDBC driver.

Objective:
Database Applications with JDBC

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

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > JDBC(TM) Database Access > JDBC Basics > Getting Started

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > JDBC(TM) Database Access > JDBC Basics > Establishing a Connection

---

Question #21 of 50

Question ID: 1328026

Given the code fragment:

```
int[] grades = {73,82,97,91,67};
```

Which two sets of expressions are valid in a `for` statement?

✗ **A)** `;i < 5; i++`

✗ **B)** `grades : var i`

✗ **C)** `;; i++`

✓ **D)** `var i : grades`

✓ **E)** `int i = 0;i < 5; i++`

Explanation

The two sets of expressions `int i = 0; i < 5; i++` and `var i : grades` are valid in a `for` statement. The first set follows the syntax of a traditional `for` statement, where the first expression is the initialization, the second expression is the termination, and the last expression is a loop modification. The second set follows the syntax of an enhanced `for` statement, using type inference.

The expression sets `;; i++` and `;i < 5; i++` are not valid in a `for` statement. Neither expression set declares the variable `i`, although the termination and/or loop modification expressions reference that variable. The code that uses these expression sets will fail compilation.

The expression `grades : var i` is not valid in a `for` statement. This expression set reverses the order of the placeholder and array variables in the syntax required in an enhanced `for` statement.

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 #22 of 50

Question ID: 1327758

Given the following:

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

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

What is the result when this program is executed?

✓ **A)** `x,y,z: 1,1,10`

✗ **B)** `x,y,z: 10,10,10`

✗ **C)** `x,y,z: 1,10,10`

✗ **D)** `x,y,z: 1,1,1`

Explanation

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

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

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

The key difference between primitive variables and reference variables are:

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

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

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

Objective:
Working with Java Data Types

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

References:

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

Primitive vs Reference Data Types

Question #23 of 50

Question ID: 1328098

Given the code fragment:

```
Arrays.stream(new int[]{1,3,5,7,9,11}, 1, 4)
  .forEachOrdered(System.out::println);
```

What is the result?

✓ **A)** 3
5
7

✗ **B)** 3
9

✗ **C)** 1
2
3
4

✗ **D)** 3
5
7
9

✗ **E)** 1
2
3

Explanation

The correct result is the following output:

```
3
5
7
```

There are several overloaded `Arrays.stream` methods. Generally, they use the contents of an array as the items that will be sent down the stream. The type of the stream is determined by the type of the array, though that is not significant in this question.

The overload used in this example uses the second and third arguments to select a sub-range of the array contents. The second argument specifies the first array element to be used (in this example, subscript 1) and therefore the first value to send to the stream is 3. The third argument defines the first array element ***not*** to use. In this example, the third argument is 4, which means that the last item to be sent down stream will be at subscript 3.

Therefore, the elements from the array that are sent to the stream will be 3, 5, and 7.

All the other outputs are incorrect because they do not correctly identify the elements in the stream.

Objective:
Working with Streams and Lambda expressions

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

References:

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

Question #24 of 50

Question ID: 1328090

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

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

Explanation

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

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

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

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

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

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

Objective:
Working with Streams and Lambda expressions

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

References:

The Java Tutorials > Collections > Lesson: Aggregate Operations

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

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

---

Question #25 of 50

Question ID: 1327990

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

```
01  Path filePath = Paths.get(nameOfFile);
02  List<String> liner = new ArrayList<>();
03  Charset encoding = StandardCharsets.UTF_8;
04  Object get = new Object(filePath, encoding);
```

```
05  while (get.hasNextLine())
06   liner.add(get.nextLine());
07  get.close();
08  return liner;
```

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

- ✓ **A)** Replace `Object` with `Scanner` at line `04`
- ✗ **B)** Replace `Path` with `File` at line `01`
- ✗ **C)** Replace `List` with `Collection` at line `02`
- ✗ **D)** **Replace `Object` with `File` at line `04`**

Explanation

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

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

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

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

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

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

Objective:
Java File I/O

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

References:

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

Question #26 of 50

Question ID: 1327817

Given the following code:

```
public class VarScope {
   public int i1;
   public static int i2;
}
```

And given the following output:

```
10 + 5 = 15
```

Which code fragments will generate the required output? (Choose all that apply.)

✗ **A)**
```
VarScope v1 = new VarScope();
VarScope v2 = new VarScope();
v1.i1 = 10; v2.i1 = 5;
System.out.format("%d + %d = %d", v1.i2, v1.i2, v1.i2 + v1.i2);
```

✗ **B)**
```
VarScope v1 = new VarScope();
VarScope v2 = new VarScope();
v1.i2 = 10; v2.i2 = 5;
System.out.format("%d + %d = %d", v1.i2, v2.i2, v1.i2 + v2.i2);
```

✓ **C)**
```
VarScope v1 = new VarScope();
VarScope v2 = new VarScope();
v1.i1 = 10; v2.i1 = 5;
System.out.format("%d + %d = %d", v1.i1, v2.i1, v1.i1 + v2.i1);
```

✓ **D)**
```
VarScope v1 = new VarScope();
VarScope v2 = new VarScope();
v1.i1 = 10; v2.i2 = 5;
System.out.format("%d + %d = %d", v1.i1, v1.i2, v1.i1 + v1.i2);
```

Explanation

The following two code fragments will generate the required output:

```
VarScope v1 = new VarScope();
VarScope v2 = new VarScope();
v1.i1 = 10; v2.i1 = 5;
System.out.format("%d + %d = %d", v1.i1, v2.i1, v1.i1 + v2.i1);
```

```
VarScope v1 = new VarScope();
VarScope v2 = new VarScope();
v1.i1 = 10; v2.i2 = 5;
System.out.format("%d + %d = %d", v1.i1, v1.i2, v1.i1 + v1.i2);
```

The first code fragment uses only the instance fields `i1` for the `v1` and `v2` objects. The second code fragment sets the static field `i2`. The value for field `i2` will be the same across both objects because it is a `static` member. Thus, the value for `v1.i2` will equal `v2.i2`.

The code fragment that sets `i2` twice, once for `v1` and then for `v2`, will not generate the required output. Because `i2` is a `static` field, it will have the same value across both objects. The output will be as follows:

```
5 + 5 = 10
```

The code fragment that retrieves `i2` from both `var1` and `var2` variables will not generate the required output. Because `i2` is never set, the default value is 0 for both `var1` and `var2` objects. The output will be as follows:

```
0 + 0 = 0
```

Objective:
Java Object-Oriented Approach

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

References:

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

---

Question #27 of 50

Question ID: 1327811

Given the following code line:

```
overloadedMethod(10.5);
```

Which two statements are true about matching overloaded methods?

- ✗ **A)** An overloaded method that declares a `float` parameter may match.
- ✓ **B) An overloaded method that declares an `Object` parameter may match.**
- ✗ **C)** An overloaded method that declares an `Object` return type may match.
- ✓ **D) An overloaded method that declares a `Double` parameter may match.**
- ✗ **E)** An overloaded method that declares a `float` return type may match.
- ✗ **F)** An overloaded method that declares a `Double` return type may match.

Explanation

An overloaded method that declares either a `Double` parameter or an `Object` parameter may match. The default primitive type for a literal fractional value is `double`, and this type is automatically boxed as a `Double` object. Thus, an overloaded method with a `Double` parameter will be matched. If this overloaded method does not exist, then an overloaded method with an `Object` parameter will be matched. 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.

An overloaded method that declares a `float` parameter will not match. The default primitive type for a literal fraction value is `double`, not `float`.

An overloaded method that declares a `float`, `Double`, or `Object` return type will not match. This is because only the list of parameter data types determines which overloaded method is executed.

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 #28 of 50

Question ID: 1328047

Given the following code:

```
String number1="Eleven";
Object newObj = number1;
Integer number2=(Integer)newObj;
System.out.println(number2);
```

What will be the output?

- ✗ **A)** `Eleven`
- ✗ **B)** `11`
- ✗ **C)** `ArithmeticException`
- ✓ **D)** **`ClassCastException`**

Explanation

The code will generate a runtime error due to a `ClassCastException`. In this case, a `String` object is attempted to be cast into an `Integer` object. The compiler does not detect that the `Object` variable `newObj` contains a `String` object. At runtime, Java sees that the `String` class is not a super class of `Integer`. So, attempting to cast a `String` object to an `Integer` results in a `ClassCastException` being thrown.

The output will not be `11` nor `Eleven` because the code will not run correctly due to the `ClassCastException`.

An `ArithmeticException` will not be thrown in this case. An `ArithmeticException` is thrown when there is a division by zero that occurs in the code.

Objective:
Exception Handling

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

References:

Oracle Documentation > Java SE 11 API > Class ArithmeticException

Oracle Documentation > Java SE 11 API > Class ClassCastException

---

Question #29 of 50

Question ID: 1327883

Given the code fragment:

```
CharSequence obj = new StringBuilder ( new String("String") );
```

Which is the reference type?

- ✗ **A)** `StringBuilder`
- ✗ **B)** `Object`
- ✓ **C)** `CharSequence`
- ✗ **D)** `String`

Explanation

The reference type is `CharSequence`. The reference type corresponds to the type in the variable declaration.

The reference type is not `Object`. Although all classes implicitly extend the `Object` class, the reference type is the explicit data type declared for the variable.

The reference type is not `String`. Although `StringBuilder` uses the `String` class, the reference type is the explicit data type declared for the variable.

The reference type is not `StringBuilder`. `StringBuilder` is the object type because the object type corresponds to the instantiated class.

Objective:
Java Object-Oriented Approach

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

References:

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

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

---

Question #30 of 50

Question ID: 1328130

Consider the following stream of users:

```
Stream<User> userstream = users.stream();
```

You create the following filter to find all users based in Houston, TX:

```
users.filter(user -> user.isLocated("Houston"));
```

Which of the following code fragments would put all Houston-based users into one collection?

✗ **A)**
```
List<User> houston_users = users.stream().filter( user ->
    user.isLocated("Houston") );
```

✗ **B)**
```
List<User> houston_users = users.stream().map( user ->
    user.isLocated("Houston") );
```

✓ **C)**
```
List<User> houston_users = users.stream().filter( user ->
    user.isLocated("Houston") ).collect(Collectors.toList());
```

✗ **D)**
```
List<User> houston_users = users.stream().filter( user ->
    user.isLocated("Houston") ).map(Collectors.toList());
```

Explanation

You should use the code that collects all user objects that correspond to users from Houston into a `List` called `houston_users` using the `collect` method:

```
List<User> houston_users = users.stream().filter( user ->
  user.isLocated("Atlanta") ).collect(Collectors.toList());
```

You should not use the following code because the `map` method does not collect objects into a collection:

```
List<User> houston_users = users.stream().filter( user ->
  user.isLocated("Houston") ).map(Collectors.toList());
```

The `map` method maps the elements of a stream based on a given criterion. The `map` method accepts a stream and returns a mapped element.

The following code options are incorrect because they both are missing the `collect` method:

```
List<User> houston_users = users.stream().filter( user ->
  user.isLocated("Houston") )
```

```
List<User> houston_users = users.stream().map( user ->
  user.isLocated("Houston") )
```

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 #31 of 50

Question ID: 1328158

Which statement is true about collections in the `java.util.concurrent` package, such as `BlockingQueue` and `CopyOnWriteArrayList`?

- ✗ **A)** Read access to these collections are not thread-safe.
- ✓ **B)** **Read and write access to these collections is thread-safe.**
- ✗ **C)** Deletion of elements in these collections is not thread-safe.
- ✗ **D)** Copying these collections is thread-safe.

Explanation

Read and write access to collections in the `java.util.concurrent` package is thread-safe. To prevent memory consistency errors, a resource must ensure that write operations are visible to all threads when they occur, so that subsequent operations are consistent known as a *happens-before* relationship. The `java.util.concurrent` package includes collections that define this relationship when adding an element for subsequent reading and deleting operations. These collections and their operations are safe to be used by multi-threaded applications.

Read access, in addition to write access, to these collections is thread-safe.

Deletion of elements is included in write access. Deletion of elements in these collections is thread-safe.

Copying these collections is not thread-safe. Only read/write access is thread-safe, not copying the collection itself.

Objective:
Concurrency

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

References:

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

---

Question #32 of 50

Question ID: 1328167

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

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

Explanation

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

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

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

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

Objective:
Concurrency

Sub-Objective:

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

References:

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

---

Question #33 of 50

Question ID: 1327823

Given:

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

Which constructor is the default constructor?

✗ **A)** `CardDeck (int)`

✓ **B)** Not provided.

✗ **C)** `CardDeck (int, boolean)`

✗ **D)** **`CardDeck()`**

Explanation

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

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

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

Objective:

Java Object-Oriented Approach

Sub-Objective:

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

References:

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

---

Question #34 of 50

Question ID: 1327882

Given the code fragment:

```
Readable reader = new BufferedReader(new FileReader("file.txt"));
```

Which is the object type for `reader`?

- ✓ **A)** `BufferedReader`
- ✗ **B)** `Readable`
- ✗ **C)** `String`
- ✗ **D)** `FileReader`

Explanation

The object type for reader is `BufferedReader`. The object type corresponds to the instantiated class.

The object type is not `String` or `FileReader`. Although `FileReader` uses the `String` class and `BufferedReader` uses the `FileReader`, the object type is the outermost instantiated class.

The object type is not `Readable`. `Readable` is the reference type because it corresponds to the type in the variable declaration.

Objective:
Java Object-Oriented Approach

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

References:

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

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

---

Question #35 of 50

Question ID: 1328004

Consider the following code:

```
import java.util.ArrayList;
import java.util.List;

public class store{
 //Annotation goes here
 private void show(List<String>... items) {
  for (List<String> item : items) {
   System.out.println(item);
  }
 }

 public static void main(String[] args) {
  store s = new store();
  List<String> itemlist = new ArrayList<String>();
  itemlist.add("Piano");
  itemlist.add("Electric Guitar");
  s.show(itemlist);
 }
}
```

This generates the following output:

```
Note: store.java uses unchecked or unsafe operations.
```

Which annotation should you add to ensure the program runs without warnings?

✗ **A)** `@SupressWarnings`

✗ **B)** `@Override`

✗ **C)** **`@Deprecated`**

✓ **D)** `@SafeVarargs`

Explanation

You should use the `@SafeVarargs` annotation. The annotation needs to be used as indicated below:

```
public class store{
 @SafeVarargs
 private void show(List<String>... items) {
  for (List<String> item : items) {
   System.out.println(item);
  }
 }
 //Other code omitted
}
```

The `show` method uses a List of items as arguments which the compiler perceives as a potentially unsafe operation. The `@SafeVarargs` annotation is used for methods or constructors that use `varargs` parameters. This annotation is used to ensure that unsafe operations are not performed by the method on the `varargs` parameters. This annotation can be used on `final` or `static` methods or constructors. It can be applied only to those methods that cannot be overridden. Using this annotation on any other method will result in a compilation error.

`@Deprecated` is incorrect because it does not resolve unsafe operation warnings. 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 resolve unsafe operation warnings. 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 resolve unsafe operation warnings for arguments. It is an annotation that specifies warnings in string form that the compiler must ignore.

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 #36 of 50

Question ID: 1327783

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

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

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

Explanation

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

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

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

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

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

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

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

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

Objective:
Working with Java Data Types

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

References:

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

Lambda Expressions and Variable Scope

Question #37 of 50

Question ID: 1327820

Given:

```
class Fruit {}

class Coconut extends Fruit {
   Coconut(int size) {}
```

```
   Coconut(int size, String region) {this(size);}
}
```

Which lines of code invoke a default constructor? (Choose all that apply.)

✗ **A)** `new Fruit(3);`

✓ **B)** `new Coconut(3, "Costa Rica");`

✗ **C)** `new Coconut();`

✓ **D)** **`new Fruit();`**

✓ **E)** `new Coconut(3);`

✗ **F)** `new Fruit(3, "Costa Rica");`

Explanation

The following three lines of code will invoke a default constructor:

```
new Fruit();
new Coconut(3);
new Coconut(3, "Costa Rica");
```

If no constructor is defined for a class, then the compiler will automatically provide the default constructor. The default constructor specifies no parameters and invokes the parameterless constructor of the superclass. In this scenario, the `Fruit` class will have a default constructor. The first line of code explicitly invokes the default constructor by instantiating a `Fruit` object.

If any constructor is defined in a class, then the compiler will not provide the default constructor. In this scenario, the `Coconut` class will not have a default constructor. When invoking subclass constructors that do not explicitly invoke a superclass constructor, the default constructor of the superclass will be invoked implicitly at runtime. The second and third lines of code invoke the default constructor in the `Fruit` class because `Coconut` is a subclass of `Fruit`, which does have a default constructor.

The line of code that invokes a parameterless constructor for `Coconut` will not invoke a default constructor. The `Coconut` class will not have a default constructor because there are already constructors defined. Because there is no default constructor to invoke explicitly, this code will fail compilation.

The lines of code that invoke `Fruit` constructor using parameters will not invoke a default constructor. The `Fruit` class does not have any constructors defined with parameters, so these code lines will fail compilation.

Objective:
Java Object-Oriented Approach

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

References:

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

Question #38 of 50

Question ID: 1328201

You create code for an airline reservation system where you need account for a flight from Atlanta, Georgia, USA, to New Delhi, India. The flight is 1140 minutes in duration and spans across multiple time zones:

```
01  public class TimeZone {
02    public static void main(String[] args) {
03      DateTimeFormatter tzFormat =
04       DateTimeFormatter.ofPattern("MMM d yyyy hh:mm a");
05      LocalDateTime dateofFlight =
06       LocalDateTime.of(2015, Month.AUGUST, 30, 15, 30);
07      ZoneId ziDeparture = ZoneId.of("America/Chicago");
08      ZonedDateTime dtDeparture =
09       ZonedDateTime.of(dateofFlight, ziDeparture);
10      String dateDisplay1 = dtDeparture.format(tzFormat);
11      System.out.printf("DEPARTURE: %s (%s)%n",
12       dateDisplay1, ziDeparture);
13       ZoneId ziArrival = ZoneId.of("Asia/New_Dehi");
14       //Insert code here
15       String dateDisplay2 = dtArrival.format(tzFormat);
16       System.out.printf("ARRIVAL: %s (%s)%n",
17        dateDisplay2, ziArrival);
18    }
19  }
```

Which code statement should you insert at line 14 to add the flight duration to the date information?

✗ **A)**
```
DateTimeFormatter dtArrival =
    dtDeparture.withZoneSameInstant(ziArrival).plusMinutes(1140);
```

✗ **B)**
```
ZoneId dtArrival =
    dtDeparture.withZoneSameInstant(ziArrival).plusMinutes(1140);
```

✓ **C)**
```
ZonedDateTime dtArrival =
    dtDeparture.withZoneSameInstant(ziArrival).plusMinutes(1140);
```

✗ **D)**
```
String dtArrival =
    dtDeparture.withZoneSameInstant(ziArrival).plusMinutes(1140);
```

Explanation

You should use the `ZonedDateTime` class as shown below:

```
ZonedDateTime dtArrival =
dtDeparture.withZoneSameInstant(ziArrival).plusMinutes(1140);
```

The `ZonedDateTime` class includes a date and time in a specific time zone and can support daylight savings changes. It works in correlation with the `java.util.Calendar` class.

The other options will not work in this scenario because only the `ZonedDateTime` class correctly evaluates time zone differences. The return value must be a `ZonedDateTime` object.

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 > Java Platform Standard Edition 11 API Specification > java.util.timezone > Class TimeZone

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

---

Question #39 of 50

Question ID: 1327899

Which code segment correctly defines an interface?

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

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

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

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

Explanation

The following code segment correctly defines an interface:

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

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

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

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

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

Objective:
Java Object-Oriented Approach

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

References:

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

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

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

---

Question #40 of 50

Question ID: 1327756

Given the following:

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

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

✗ **A)** `boolean b = null;`

✗ **B)** `boolean b = 0;`

✗ **C)** **`byte b = 1101;`**

✓ **D)** `byte b = 0b1101;`

Explanation

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

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

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

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

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

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

When declaring and initializing variables in Java, you should delineate each declaration with a semi colon (`;`). In Java SE 10 and above, you also use the `var` keyword to infer the data type of the variable based on its initialization. For example, line 3 could have been written as follows:

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

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

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

Objective:
Working with Java Data Types

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

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Language Basics > Primitive Data Types

OpenJDK > Java Local Variable Type Inference: Frequently Asked Questions

---

Question #41 of 50

Question ID: 1328065

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

✗ **A)** `(s, n) -> n`

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

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

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

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

✓ **F)** **`(s, n) -> s + n`**

Explanation

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

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

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

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

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

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

Objective:
Working with Streams and Lambda expressions

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

References:

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

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

---

Question #42 of 50

Question ID: 1328021

Which statement is true about using a `String` object in a `switch` statement?

- ✗ **A)** Execution falls through if `break` statements are specified in case labels.
- ✓ **B)** **`String` comparisons in case labels are case-sensitive.**
- ✗ **C)** Execution terminates if `break` statements are not specified in case labels.
- ✗ **D)** `String` comparisons in case labels are case-insensitive.

Explanation

When using a `String` object in a `switch` statement, `String` comparisons in case labels are case-sensitive. The comparison in each case label represents an invocation of the `String.equals` method, which compares each case-sensitive character in the string literals. To work around case sensitivity, you could invoke the `toLowerCase()`

or `toUpperCase()` methods on the `String` object and specify a lower-case or upper-case expression for each case label.

`String` comparisons are not case-insensitive. The comparison in each case label represents an invocation of the `String.equals` method.

Execution will not fall through if `break` statements are specified in case labels. `break` statements terminate execution when specified in a case label.

Execution will not terminate if `break` statements are not specified in case labels. Without `break` statements, execution will fall through case labels.

Objective:
Controlling Program Flow

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

References:

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

---

Question #43 of 50

Question ID: 1328115

You create a list of part names for an auto parts company:

```
01  List<String> partNames = new ArrayList<>();
02  memberNames.add("engine");
03  memberNames.add("wheels");
04  memberNames.add("bumpers");
05  memberNames.add("windshield");
06  memberNames.add("lights");
07  memberNames.add("pedals");
```

To find the first occurrence of an engine, you use the following code:

```
08  String match = partNames.stream()
09  .filter((s) -> s.startsWith("e"))
10  // INSERT CODE
11  System.out.println(match);
```

Which of the following statement(s) can you insert at line `10` to make this code function correctly? (Choose all that apply.)

✓ **A)** `.findFirst().get();`

✗ **B)** `.allMatch().get();`

✗ **C)** `.noneMatch().get();`

✓ **D)** `.findAny().get();`

Explanation

Because there is only one part that begins with the letter *e*, you could use either the `findFirst` or `findAny` methods as shown below:

```
String match = partNames.stream()
.filter((s) -> s.startsWith("e"))
.findFirst().get();
System.out.println(match);
```

Java provides various methods to search for data in a stream. The `findFirst` method returns an `Optional` object that describes the very first element of the stream. It returns an empty `Optional` object if the stream is empty. Also, if the stream does not have an encounter order, then any element can be returned by the `findFirst` method. The `findAny` method returns an `Optional` object that describes any element of the stream. It returns an empty `Optional` object if the stream is empty. The `anyMatch` method returns a `boolean` value of true if any elements of the stream match the predicate, and false otherwise.

You should not use `allMatch` because `allMatch` returns true when all elements of a stream match the predicate which in this case is not possible because only one part starts with an *e*.

You should not use `noneMatch` because it returns true when none of the elements match the predicate. Some of the parts in the database start with *e*, which means they will match the predicate.

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 #44 of 50

Question ID: 1327934

Which statement is true about using `ArrayList` objects?

✗ **A)** `ArrayList` objects are fixed in size and not resizable like arrays.

✓ **B)** **`ArrayList` objects require less low-level implementation than arrays.**

✗ **C)** `ArrayList` objects provide better performance than arrays.

✗ **D)** `ArrayList` objects require less memory than arrays.

Explanation

`ArrayList` objects require less low-level implementation than arrays. Classes in the Collections Framework, like `ArrayList`, provide implementation for useful data structures and algorithms. For example, rather than implementing the logic for creating a new array when more elements are required, an `ArrayList` object automatically resizes to accommodate additional elements.

`ArrayList` objects do not require less memory than arrays. `ArrayList` objects provide a variety of additional functionality over arrays, which increases its memory footprint.

`ArrayList` objects do not provide better performance than arrays. Although the Collections Framework provides implementation to reduce development time, collections do not provide better performance than simple arrays. However, complex algorithms for sorting and handling elements provide better performance in collections than if they were implemented manually using an array.

`ArrayList` objects are not fixed in size and arrays are not resizable. Arrays are fixed in size, while `ArrayList` objects are resizable.

Objective:
Working with Arrays and Collections

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

References:

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

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

Question #45 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)** `03`

✗ **B)** `02`

✗ **C)** **`01`**

✗ **D)** `04`

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 #46 of 50

Question ID: 1327763

Given the following:

```
int x = 5;
```

Which two expressions evaluate to 5?

- ✓ **A)** `10 - x--`
- ✗ **B)** `10 - --x`
- ✗ **C)** `x-- + 10`
- ✗ **D)** `10 - -x`
- ✓ **E)** `-x + 10`
- ✗ **F)** `--x + 10`

Explanation

The following two expressions evaluate to 5:

```
-x + 10
```

```
10 – x- -
```

The first expression uses the unary minus operator to negate x, so that the expression becomes -5 + 10 = 5. The second expression uses the unary decrement operator as a postfix operation. Postfix operations do not change a variable until after the overall expression is evaluated. Thus, the expression is 10 -5 = 5. The only side effect of this operation is that after the expression is evaluated, the value of x is 4.

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

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

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

The expressions `x-- + 10` and `10 - -x` do not evaluate to 5. Both expressions evaluate to 15. In the first expression, x is decremented only after its evaluation, so that the expression becomes 5 + 10 = 15. In the second expression, x is negated to -5, so that the expression becomes 10 - -5 = 10 + 5 = 15.

The expressions `--x + 10` and `10 - --x` do not evaluate to 5. In both expressions, x is decremented to 4 and evaluates to 6. The first expression becomes -4 + 10 = 6. The second expression becomes 10 - 4 = 6.

Objective:
Working with Java Data Types

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

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Language Basics > Assignment, Arithmetic, and Unary Operators

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

---

Question #47 of 50

Question ID: 1328039

Which of these situations best illustrate the advantages of exception handling in Java? (Choose all that apply.)

- ✗ **A)** Reducing the number of syntax errors
- ✗ **B)** Sending errors down the method call stack
- ✓ **C)** **Grouping errors into general types**
- ✓ **D)** **Keeping program and error code separate**

Explanation

Exception handling in Java helps keep program code and error handling code separate. It also groups errors into different types, which is another advantage.

The option stating that Java exception handling sends errors *down* the method call stack is incorrect. In Java, exceptions are sent *up* a method call stack.

The option stating that Java exception handling reduces the number of syntax errors is incorrect. Exceptions occur at runtime, not during compilation.

Objective:
Exception Handling

Sub-Objective:

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

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Essential Classes > Advantages of Exception Handling in Java

---

Question #48 of 50

Question ID: 1327872

Given these classes:

```
abstract class Player {
   String type = "Human";
   public void printType() { System.out.println(type);}
}

class GamePlayer extends Player {
   GamePlayer() {
     type = "Computer";
   }
}

class TurnBasedPlayer extends GamePlayer {
   String type = "Turn-based";
}

class ChessPlayer extends TurnBasedPlayer {
   ChessPlayer() {
     type = "Chess";
   }
   public void printType() { System.out.println(type);}
}

class CheckersPlayer extends TurnBasedPlayer {
   CheckersPlayer() {
     type = "Checkers";
   }
   public void printType() { System.out.println(type);}
}
```

Which code fragment will output `Computer`?

✗ **A)**
```
GamePlayer p = new Player();
p.printType();
```

✗ **B)**
```
GamePlayer p = new CheckersPlayer();
p.printType();
```

✓ **C)**
```
Player p = new TurnBasedPlayer();
p.printType();
```

✗ **D)**
```
Player p = new ChessPlayer();
p.printType();
```

Explanation

The following code fragment will output Computer:

```
Player p = new TurnBasedPlayer();
p.printType();
```

This code fragment instantiates `TurnBasedPlayer` and sets it to a `Player` type. `TurnBasedPlayer` hides the `type` field declared in `Player`. Because polymorphism does not apply to hidden members, using a `Player` reference will output `Computer`, rather than the value set in `TurnBasedPlayer`.

The following code fragment will not output `Computer`:

```
GamePlayer p = new Player();
p.printType();
```

This code would not compile because `Player` is abstract and cannot be instantiated.

The following code fragment will not output `Computer`:

```
Player p = new ChessPlayer();
p.printType();
```

Although hidden members do not support polymorphism, overridden methods do use polymorphism. The method `printType` in `ChessPlayer` overrides the method defined in `Player`. The `printType` method in `ChessPlayer` references the hidden field `type` in `TurnBasedPlayer`, not `type` in `Player`. This code fragment will output `Chess`.

The following code fragment will not output `Computer`:

```
GamePlayer p = new CheckersPlayer();
p.printType();
```

Although hidden members do not support polymorphism, overridden methods do use polymorphism. The method `printType` in `CheckersPlayer` overrides the method defined in `Player`. The `printType` method in `CheckersPlayer` references the hidden field `type` in `TurnBasedPlayer`, not `type` in `Player`. This code fragment will output `Checkers`.

Objective:
Java Object-Oriented Approach

Sub-Objective:

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

References:

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

---

Question #49 of 50

Question ID: 1327790

Given the following:

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

Which statement(s) should be inserted in the code to output `20`?

✗ **A)** `System.out.println(modify(i + 10));`

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

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

✓ **D)** **`modify(i); System.out.println(i + 10);`**

Explanation

The statements `modify(i); System.out.println(i + 10);` should be inserted in the code to output `20`. Although the object reference is copied as an argument for the `modify` method, `Integer` objects are immutable, as are the `String` and type wrapper classes. With autoboxing between primitive type and type wrapper classes, the code for the modify method may seem to change the value of variable `i`. The compiler is rendering the code modify method as follows:

```
static int modify (Integer i) {
   i = new Integer(i.intValue() + 10);
   return new Integer(i.intValue() + 10);
}
```

As this code indicates, the parameter `i` no longer references the original `Integer` argument, so no changes affect the original `Integer` object. Because the local variable `i` is 10, then the output will be 10 + 10 (`20`).

The statements `System.out.println(modify(i));` and `System.out.println(modify(i + 10));` should not be inserted in the code to output `20`. This is because `Integer` objects, like `String` objects, are immutable, and the `modify` method is reassigning the variable `i` to new `Integer` objects, not modifying the original argument object. Both statements invoke the `modify` method with the primitive value 10 to get the return value 30.The first statement will directly output `30`, while the second statement will output `40` (30 + 10).

The statements `modify(i); System.out.println(i);` should not be inserted in the code to output `20`. This is because `Integer` objects, like `String` objects, are immutable, and the `modify` method is reassigning the variable `i` to new `Integer` objects, not modifying the original argument object. The value of `i` remains at 10, so that the output is `10`.

Objective:
Java Object-Oriented Approach

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

References:

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

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

---

Question #50 of 50

Question ID: 1327944

Given the following:

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

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

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

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

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

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

Explanation

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

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

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

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

Objective:
Working with Arrays and Collections

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

References:

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