Quick Quiz September 6, 2022

Test ID: 222710138

Question #1 of 50

Question ID: 1328015

Given the following:

```
int i = 5;
```

Which two code fragments will output `e = mc2`?

✗ **A)**
```
if ( i >= 3)
   System.out.print("e ");
if (i == 5) {
   System.out.print("= ");
} else {
   System.out.print("!= ");
   if (i < 6)
     System.out.print("mc2");
   else
     System.out.println("no energy");
}
```

✗ **B)**
```
if ( i >= 3)
   System.out.print("e ");
else if (i == 5)
   System.out.print("= ");
else if (i > 6)
   System.out.print("!= ");
else if (i < 6)
   System.out.print("mc2");
else
   System.out.println("no energy");
```

✓ **C)**
```
if ( i >= 3)
   System.out.print("e ");
if (i < 6)
   System.out.print("= ");
else
   System.out.print("!= ");
if (i == 5)
   System.out.print("mc2");
else
   System.out.println("no energy");
```

✗ **D)**
```
if ( i >= 3)
   System.out.print("e ");
else if (i == 5) {
   System.out.print("= ");
} else {
   System.out.print("!= ");
   if (i < 6)
     System.out.print("mc2");
   else
     System.out.println("no energy");
}
```

✓ **E)**
```
if ( i >= 3) {
   System.out.print("e ");
   if (i < 6)
     System.out.print("= ");
   else
     System.out.print("!= ");
   if (i == 5)
     System.out.print("mc2");
} else
   System.out.println("no energy");
```

Explanation

The following two code fragments will output `e = mc2`:

```
if ( i >= 3) {
   System.out.print("e ");
   if (i < 6)
     System.out.print("= ");
   else
     System.out.print("!= ");
```

```
   if (i == 5)
     System.out.print("mc2");
} else
   System.out.println("no energy");
```

```
if ( i >= 3)
   System.out.print("e ");
if (i < 6)
   System.out.print("= ");
else
   System.out.print("!= ");
if (i == 5)
   System.out.print("mc2");
else
   System.out.println("no energy");
```

In the first code fragment, the expression `i >= 3` in the first `if` statement must evaluate as `true` for the remaining `if` statements to be evaluated. Both expressions `i < 6` and `i== 5` are then evaluated as `true` as well. In the second code fragment, all three expressions are evaluated separately without being nested in each other.

The following code fragments will not output `e = mc2`, but output `e`, instead:

```
if ( i >= 3)
   System.out.print("e ");
else if (i == 5)
   System.out.print("= ");
else if (i > 6)
   System.out.print("!= ");
else if (i < 6)
   System.out.print("mc2");
else
   System.out.println("no energy");
```

```
if ( i >= 3)
   System.out.print("e ");
if (i == 5) {
   System.out.print("= ");
} else {
   System.out.print("!= ");
   if (i < 6)
     System.out.print("mc2");
   else
     System.out.println("no energy");
}
```

The output is `e` because expressions in `else if` statements and `if` statements nested in an `else` statement will only be evaluated when the first `if` statement is `false`.

The following code fragment will output `e=`, not `e = mc2`:

```
if ( i >= 3)
   System.out.print("e ");
else if (i == 5) {
   System.out.print("= ");
} else {
   System.out.print("!= ");
   if (i < 6)
     System.out.print("mc2");
   else
     System.out.println("no energy");
}
```

The output is `e=` because the expression `i < 6` is contained in an `if` statement nested in an `else` statement and will only be evaluated if the expression `i == 5` in the second `if` statement is `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 if-then Statement

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

---

Question #2 of 50

Question ID: 1328165

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

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

✗ **D)** Use the `volatile` keyword when declaring the Queue variable.

Explanation

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

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

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

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

Objective:
Concurrency

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

References:

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

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

---

Question #3 of 50

Question ID: 1328017

Given the following:

```
if (x != 0)
   System.out.print("She");
else
   System.out.print("Sally");
if (x < 5)
```

```
   System.out.print(" sells seashells");
if ( x > 10)
   System.out.print(" will sell all her seashore shells");
if (x < 15)
   System.out.print(" by the");
else if (x < 20)
   System.out.print(" on the");
if ( x < 10)
   System.out.print(" seashore");
else
    System.out.print(" seashell shore");
```

Which value for the variable x will output `She will sell all her seashore shells on the seashell shore`?

- ✓ **A) 15**
- ✗ **B)** 10
- ✗ **C)** 0
- ✗ **D)** 20

Explanation

The value 15 for the x variable will output `She will sell all her seashore shells on the seashell shore`. To output `She`, x must be not be 0 to meet the criteria of the first `if` statement. To output `will sell all her seashore shells`, x must be greater than 10 to meet the criteria of the third `if` statement. To output `on the`, x must be less than 20 but not less than 15 to reach and meet the criteria of the first `else if` statement. Finally, x must be greater than 10 to reach the second `else` statement and output `seashell shore`.

The value 0 for x will output `Sally`, not `She`. The complete output will be `Sally sells seashells by the seashore` because the criteria of the first `if` statement is met.

The value 10 for x will output `by the`, not `on the`. The complete output will be `She by the seashell shore`. This is because the criteria of the fourth `if` statement is met, but not the fifth `if` statement.

The value 20 for x will not output `on the`. The complete output will be `She will sell all her seashore shells seashell shore`. This is because the criteria of the `else if` statement is not met.

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

Objective:
Controlling Program Flow

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

References:

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

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

---

Question #4 of 50

Question ID: 1327947

Given the following:

```
char[][] charArray2D = {{'c','u','p'},{'o'},{'f'},{'j','a','v','a'}};
```

Which two expressions will evaluate to `false`?

- ✗ **A)** `charArray2D[1].getClass().isArray()`
- ✓ **B)** **`charArray2D.length > 4`**
- ✗ **C)** `charArray2D[1].length < 2`
- ✓ **D)** **`charArray2D[0].length < 2`**
- ✗ **E)** `charArray2D[0].getClass().isArray()`
- ✗ **F)** `charArray2D.getClass().isArray()`

Explanation

The expressions `charArray2D.length > 4` and `charArray2D[0].length < 2` will evaluate to `false`. The first expression compares the number of elements in the first dimension, which is 4. The second expression compares the number of elements in the first array, which is 3.

The expressions that use the `getClass().isArray()` invocation will evaluate to `true`. `charArray2D` is an array, as is each of its child elements.

The expression `charArray2D[1].length < 2` will evaluate to `true`. This is because the second array has only one element.

Objective:
Working with Arrays and Collections

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

References:

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

Question #5 of 50

Question ID: 1328064

Given the custom exception:

```
class OutsideStringException extends StringIndexOutOfBoundsException {}
class CannotFindPatternInStringException extends OutsideStringException {}
```

Given the method:

```
boolean searchString(String pattern, String str) throws OutsideStringException {
  //implementation omitted
}
```

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

- ✓ **A)** No handling action is required.
- ✗ **B)** **Catch `StringIndexOutOfBoundsException`.**
- ✗ **C)** Specify `CannotFindPatternInStringException`.
- ✗ **D)** Catch `CannotFindPatternInStringException`.
- ✗ **E)** Specify `StringIndexOutOfBoundsException`.

Explanation

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

Caching or specifying `StringIndexOutOfBoundsException` or `CannotFindPatternInStringException` is not required for the code to compile. Runtime exceptions are abnormal conditions internal to the application and often are unrecoverable. The compiler does not check catching and specifying `RuntimeException` and its subclasses.

Objective:
Exception Handling

Sub-Objective:
Create and use custom exceptions

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 > Specifying the Exceptions Thrown by a Method

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Essential Classes > Exceptions > Creating Exception Classes

---

Question #6 of 50

Question ID: 1327992

Which statement is true about JDBC drivers?

- ✓ **A) Vendor implementation must include the `Statement`, `ResultSet`, and `Connection` objects.**
- ✗ **B)** Type 4 drivers are dependent on native libraries and use bridging to connect to the data source.
- ✗ **C)** Vendor implementation must include the `SQLException` and `DriverManager` objects.
- ✗ **D)** Type 1drivers are pure Java and support use direct connections to the data source.

Explanation

Vendor implementation must include the `Statement`, `ResultSet`, and `Connection` objects. The implementation of these interfaces does not require all optional methods. Also, vendors must fully implement the `Driver`, `DatabaseMetaData`, `ParamaterMetaData`, `ResultMetaData`, `Wrapper`, `DataSource` interfaces for the JDBC 4.0 specification. The `PreparedStatement` and `CallableStatement` interfaces may also be required if supported by the data source.

Vendor implementation does not include the `SQLException` and `DriverManager` objects. The JDBC API implements these classes.

Type 4 drivers are not dependent on native libraries and use bridging to connect to the data source. Type 4 drivers are pure Java and support use direct connections to the data source. These drivers also support network protocols for connections.

Type 1 drivers are not pure Java and support use direct connections to the data source. Type 1 drivers are dependent on native libraries and use bridging to connect to the data source.

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 Community Process > Community Development of Java Technology Specifications > JSR-000221 JDBC 4.0 Final Release

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

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > JDBC(TM) Database Access > JDBC Basics > Processing Statements with JDBC

Question #7 of 50

Question ID: 1328143

Given the output:

```
Fred Jim Sheila
```

Which code fragment will generate this output?

✗ **A)**
```
System.out.println(
        Stream.of("Fred", "Jim", "Sheila")
        .reduce(" ", (a,b)->a+b));
```

✗ **B)**
```
System.out.println(
        Stream.of("Fred", "Jim", "Sheila")
        .reduce((a,b)->a + " " + b));
```

✗ **C)**
```
System.out.println(
        Stream.of("Fred", "Jim", "Sheila")
        .collect("", (a,b)->a+b, (a,b)->a+b));
```

✗ **D)**
```
Stream.of("Fred", "Jim", "Sheila")
        .reduce(" ", (a,b)->a+b)
        .ifPresent(System.out::println);
```

✓ **E)**
```
Stream.of("Fred", "Jim", "Sheila")
        .reduce((a,b)->a + " " + b)
        .ifPresent(System.out::println);
```

Explanation

The following code fragment will generate the required output:

```
Stream.of("Fred", "Jim", "Sheila")
  .reduce((a,b)->a + " " + b)
  .ifPresent(System.out::println);
```

This code fragment correctly reduces the data in the stream, concatenating the strings and separating them with a space character. The result of this reduction operation is an `Optional` object, and then extracts and prints the desired text from that optional.

Reduction operations are provided for the stream API in several forms. Two methods are provided for generalized reduction; these are `collect` and `reduce`. One of the `collect` methods accepts three arguments, like the one shown in the option built around the `collect` method. However, the arguments to that method are a `Supplier<R>`, and two `BiConsumer<R,T>` (where *R* is the final type of the collection, and *T* is the type in the stream). In the option offered, all three arguments are of the wrong type. The first is an actual `String`, rather than a `Supplier<String>` (note that in this option, the result type `R` and the stream type `T` both appear to be `String`). The second and third arguments seem to be `BinaryOperator<String>`, as they take two `String` arguments and return a `String`. This is incorrect, as the arguments should be `BiConsumer<String>`. Therefore, the option that uses the `collect` method is incorrect.

There are three variations of the `reduce` method. The two-argument version `reduce(" ", (a,b)->a+b)`, treats the first argument as an identity value. This version returns an actual object of the collected type, not an `Optional` of that type. This is because if the stream were empty, `reduce` would return the identity value instead of returning an empty `Optional`. Because of this behavior, the following option is incorrect:

```
.reduce(" ", (a,b)->a+b).ifPresent(System.out::println)
```

This code fragment is attempting to invoke the method `ifPresent` on a `String` object, and so will not compile.

The option that uses `.reduce(" ", (a,b)->a+b)` does compile, and does print an output. However, the identity value is used to start the reduction, and is not inserted in between each element of the stream. Therefore, the output of that option is:

```
FredJimSheila
```

It is important to note that the single space string used in this option as the identity value is a significant error anyway. The identity value should be something that does not affect the result, no matter where, when, or how many times it is used. Clearly an empty string does change the output, so semantically this would be a violation of the intention of the reduce operation, even if it worked. That said, this code does not produce the desired output, and is therefore incorrect.

Two options use the single-argument version of the reduce operation:

```
.reduce((a,b)->a + " " + b)
```

One, as already seen, is the correct answer. Remember that the result of this form of the `reduce` operation is an `Optional`. This is because, in the absence of an identity value, an empty stream will result in no value, and using `Optional` allows the API to express no value without using `null`. Therefore, the option that prints the result of this collection directly would result in:

```
Optional[Fred Jim Sheila]
```

It is therefore incorrect.

Objective:
Working with Streams and Lambda expressions

Sub-Objective:

Perform decomposition and reduction, including grouping and partitioning on sequential and parallel streams

References:

The Java Tutorials > Collections > Aggregate Operations > Reduction

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

---

Question #8 of 50

Question ID: 1328164

Given:

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

Which method provides the same synchronization as `concurrentMethod`?

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

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

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

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

Explanation

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

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

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

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

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

Objective:
Concurrency

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

References:

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

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

---

Question #9 of 50

Question ID: 1327793

Which two statements are true about the contents of a class?

- ✗ **A)** A class can include only non-`static` members.
- ✓ **B) A class can include nested enumerations and classes.**
- ✗ **C)** A class cannot include constructors.
- ✓ **D) A class cannot include `package` or `import` statements.**

Explanation

A class can include nested enumerations and classes but cannot include `package` or `import` statements. A class body can include `static` and non-`static` fields, methods, constructors, and nested enumerations and classes.

A class can include both `static` and non-`static` members.

A class can include constructors. If no constructor is included, then the compiler will provide a parameterless default constructor for the class.

Objective:
Java Object-Oriented Approach

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

References:

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

---

Question #10 of 50

Question ID: 1328071

Which statement(s) are true of the `UnaryOperator` interface? (Choose all that apply.)

- ✗ **A)** It declares a single abstract method that takes a single argument and returns void.
- ✗ **B)** It is intended for operation on primitive types.
- ✗ **C)** It declares a single abstract method that takes two arguments of the same reference type.
- ✓ **D)** **It specifies a single abstract method that constrains the argument and return types to be the same.**
- ✓ **E)** It inherits a single abstract method from a parent interface.

Explanation

The `UnaryOperator` interface inherits a single abstract method from a parent interface, and specifies a single abstract method that constrains the argument and return types to be the same. It is a sub-interface of `Function`. `Function` defines a single abstract method that takes a generic argument and returns a generic type, but the types are different. `UnaryOperator` only modifies the generic type variables, constraining them to be the same.

The `UnaryOperator` interface does not declare an abstract method that returns void.

The `UnaryOperator` interface does not declare a method that takes two arguments

The `UnaryOperator` interface can be used with primitive types (via autoboxing), but it is not intended for operations on primitive types. There are special interfaces to handle specific primitive types, like `IntUnaryOperator` and `DoubleUnaryOperator`.

Objective:
Working with Streams and Lambda expressions

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

References:

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

---

Question #11 of 50

Question ID: 1328001

Consider the following code:

```
public class Vader {
  public void speak() {
    System.out.println("Luke, I am your father.");
  }
}

public class Luke extends Vader {
  @Override
  public void speak(int x) {
    System.out.println("It’s not true!");
  }
}

public class Jedi {
  public static void main(String args[]) {
    Luke skyW = new Luke();
    skyW.speak();
  }
}
```

What would be the result of compiling the code above?

- ✓ **A) Compilation fails.**
- ✗ **B)** Runtime error occurs.
- ✗ **C)** `It’s not true!`

✗ **D)** `Luke, I am your father.`

Explanation

A compiler error would be generated because a method annotated as `@Override` does not override the method in the parent class correctly. An error of the following kind would be displayed:

`Method does not override or implement a method from a supertype.`

To ensure correct compilation, the parameter `int x` needs to be removed from the argument list in the overridden `speak()` method.

The other options are incorrect because a compiler error would occur, and so none of the messages in the other options would be displayed.

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

Objective:
Annotations

Sub-Objective:
Create, apply, and process annotations

References:

Oracle Technology Network > Java SE Documentation > Annotations

---

Question #12 of 50

Question ID: 1327898

Given the following classes:

```
class CongressionalCandidate implements Electable {
   public String getCitizenRequirement() { return "Naturalized Citizen"; }
}

class PresidentialCandidate implements Electable {
   public String getCitizenRequirement() { return "Birthright Citizen"; }
}
```

Which code fragment correctly defines Electable?

✗ **A)**
```
interface Electable {
   public String getCitizenRequirement() { return "General
Citizen"; }
}
```

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

✗ **C)**
```
abstract class Electable {
   public String getCitizenRequirement() { return "General
Citizen"; }
}
```

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

Explanation

The following code fragment correctly defines `Electable`:

```
interface Electable {
   String getCitizenRequirement();
}
```

Because `CongressionalCandidate` and `PresidentialCandidate` both use the `implements` keyword, `Electable` must be declared as an interface. `Electable` should contain the method `getCitizenRequirement` overridden in `CongressionalCandidate` and `PresidentialCandidate`.

By default, all methods declared in an interface are implicitly abstract and public, so implementing classes must use the `public` keyword as well. Since Java 9, interfaces can also support private methods for code reuse without providing visibility to implementing classes.

The code fragments that declare abstract classes do not correctly define `Electable`. The `implements` keyword requires an interface, not an abstract class, for implementing classes. The `extends` keyword is required for an abstract class because implementing classes must use inheritance.

The code fragment that declares an interface with a method body does not correctly define `Electable`. Methods in an interface cannot contain a body and are implicitly abstract and public.

Objective:
Java Object-Oriented Approach

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

References:

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

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

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

---

Question #13 of 50

Question ID: 1328049

Given:

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

Which exception is thrown by running the given code?

✓ **A)** `NullPointerException`

✗ **B)** `IndexOutOfBoundsException`

✗ **C)** `StringIndexOutOfBoundsException`

✗ **D)** `ArrayIndexOutOfBoundsException`

Explanation

`NullPointerException` is the exception thrown by running the given code. This exception is thrown whenever an object is required, such as when accessing instance members. In this code, the `charAt` method is invoked on a non-existing object, so a `NullPointerException` is thrown.

The exceptions `IndexOutOfBoundsException`, `ArrayIndexOutOfBoundsException`, or `StringIndexOutOfBoundsException` will not be thrown by running the given code. This is because the object is not available to attempt executing the `charAt` method. If the `charAt` method is invoked with an invalid index for a string, then a `StringIndexOutOfBoundsException` will be thrown.

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 NullPointerException

---

Question #14 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 declared using the final keyword.
- ✓ **C)** The variable must be effectively final.
- ✗ **D)** **Value of the variable can be changed within the lambda statement.**

Explanation

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

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

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

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

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

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

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

A lambda expression's body contains a scope which is the same as a regular nested block of code. Additionally, lambda expressions can access local variables from a scope which are functionally final. This means that these

variables must either declared as `final` or are not modified.

Objective:
Working with Java Data Types

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

References:

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

Lambda Expressions and Variable Scope

Question #15 of 50

Question ID: 1328166

Which statement is true about the `ReadWriteLock` interface?

- ✗ **A)** A read lock can be used by multiple reader threads, while a writer thread can use the write lock.
- ✓ **B)** **There are two locks: one for read-only operations and one for write operations.**
- ✗ **C)** The shared lock is not exclusive.
- ✗ **D)** There is one shared lock used for both read-only and write operations.

Explanation

The `ReadWriteLock` interface has two locks: one for read-only operations and one for write operations. The read lock is not exclusive and can be held by multiple threads that perform read operations, if no write operations are performed. The write lock is exclusive, so that only a single thread can perform write operations.

There is not one shared lock for read-only and write operations. The `ReadWriteLock` interface has two locks.

A read lock cannot be used by multiple threads if a writer thread is using the write lock. If a writer thread is using the write lock, then the read lock cannot be shared between multiple threads.

The shared lock is exclusive if performing a write operation. If no write operation is being performed, then the read lock can be shared across multiple threads.

Objective:
Concurrency

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

References:

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

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

---

Question #16 of 50

Question ID: 1328144

Given:

```
int [] res = IntStream.of(1,2,3,4,5,6,7,8,9,10)
  .parallel()
  .collect( ()->new int[2],  // line n1
  (a,b)->{if (b%2==0) a[1]+=b; else a[0]+=b;}, // line n2
  (a,b)->{a[0]+=b[0];a[1]+=b[1];});
System.out.println("Odd sum = " + res[0] + " even sum = " + res[1]);
```

What is the result?

- ✓ **A)** `Odd sum = 25 even sum = 30`
- ✗ **B)** Compilation fails at line `n1`
- ✗ **C)** Compilation fails at line `n2`
- ✗ **D)** `Odd sum = 55 even sum = 55`
- ✗ **E)** **Non-deterministic output**

Explanation

The result is the following output:

```
Odd sum = 25 even sum = 30
```

This collect method takes three arguments. The first is a `Supplier<R>` where *R* is the result type of the collection operation. In this case, the supplier creates an array of two `int` values, and that is consistent with the result type for the operation.

The second argument for this collector is a `BiConsumer<R,T>` where *T* is the stream type. Since the stream contains integers, these might be `int` primitives or `Integer` objects. The behavior of the block lambda is to test if the second argument (the item from the stream) is odd or even, and to add that stream value to one or the other element of the array depending on whether the stream value is odd or even. The arguments are `int[2]` and `int`/`Integer`, and are used correctly for those types.

The return type of the `BiConsumer` is `void`, and the block of the lambda does not return anything. The resulting behavior is a collection of the sums for odd and even numbers seen in the two elements of the array of int, and to

output the odd sum, *25*, and the even sum, *30*.

The output will not have the same value of 55 for both odd and even numbers because the lambda expression in the second argument checks for whether items are odd or even and third argument increments these values independently.

The output is deterministic. The collect operation differs from a reduce operation in that each thread that is created in a parallel stream situation is given its own mutable storage for collecting intermediate results. Because each thread is given its own mutable storage, the resulting output is predictable and expected.

The code at line `n1` is correct. It defines a supplier of `int[2]`, which is appropriate as the first argument to the collector and compatible with the rest of the collector arguments.

The code at line `n2` is also correct. It defines a lambda that is compatible with `BiConsumer<int[2], int>`.

Objective:
Working with Streams and Lambda expressions

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

References:

The Java Tutorials > Collections > Aggregate Operations > Parallelism

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

---

Question #17 of 50

Question ID: 1328087

Given:

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

What is the result?

✓ **A)**
```
Fred
Jim
```

✗ **B)**
```
Fred
Jim
Sheila
```

✗ **C)**
```
Fred
```

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

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

Explanation

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

```
Fred
Jim
```

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

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

Objective:
Working with Streams and Lambda expressions

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

References:

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

The Java Tutorials > Collections > Lesson: Aggregate Operations

---

Question #18 of 50

Question ID: 1328133

You need to perform stream operations on a collection of `User` objects. Consider the code below:

```
01  List<Integer> userIDs =
02  // INSERT CODE
03  .filter(u -> u.getDep() == User.IT)
04  .sorted(comparing(User::getValue).reversed())
05  .map(User::getUserID)
06  .collect(toList());
```

Which code snippet should you insert at line `02` to make the stream processing as efficient as possible?

✗ **A)** `users.users.stream()`

✓ **B)** **`users.parallelStream()`**

✗ **C)** `users.stream().users.stream()`

✗ **D)** `users.stream().stream()`

Explanation

You should insert `users.parallelStream()` at line `02` to make the stream processing as efficient as possible:.

```
01  List<Integer> userIDs =
02  users.parallelStream()
03  .filter(u -> u.getDep() == User.IT)
04  .sorted(comparing(User::getValue).reversed())
05  .map(User::getUserID)
06  .collect(toList());
```

The other options are syntactically incorrect and will not create parallel streams. Only the `parallelStream` method creates a parallel stream in Java. When you call the appropriate method for creating a parallel stream, the Java streams API decompose queries internally automatically to make use of multiple cores on a computer. This is what results in a more efficient processing by the system.

The option `users.stream().stream()` is incorrect as it tries to create a stream from a stream.

The option `users.stream().users.stream()` is incorrect as the syntax for streams does not allow using a collection after a `Stream` method.

The option `users.users.stream()` is incorrect because you cannot join two instances of the same collection using the dot operator. This is syntactically incorrect.

Objective:
Working with Streams and Lambda expressions

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

References:

Oracle Technology Network > Java SE > Articles > Processing Data with Java 8 Streams

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

---

Question #19 of 50

Question ID: 1328177

Consider the following code:

```
01  class exceptional {
02    public static void main(String[] args) throws FileNotFoundException {
03      FileInputStream input = new FileInputStream(System.getenv("APPDATA") + args[0]);
04    }
05  }
```

Which line of the code contains a possible security vulnerability?

✗ **A)** There are no obvious vulnerabilities in this code.

✗ **B)** Line 01

✓ **C) Line 02**

✗ **D)** Line 03

Explanation

Line 02 has a security risk. Throwing a `FileNotFoundException` can disclose sensitive file structure information to an attacker.

One way to make code more security-compliant is to issue error messages that do not expose any underlying file or directory information. An example would be an I/O error message like “File not valid”.

Also, you can use the `File.getCanonicalFile()` method that first canonicalizes the file name so that filepath name comparisons can be made in a more simplified way. This is illustrated in the following code:

```
File myfile = null;
try {
  myfile = new File(System.getenv("APPDATA") +
  args[0]).getCanonicalFile();
  if (!file.getPath().startsWith("e:\\mypath")) {
    System.out.println("File not valid");
    return;
  }
} catch (IOException excep) {
  System.out.println("File not valid");
}
```

The other lines of code do not contain a security risk.

To protect confidential information from illegal access, you need to follow these guidelines:

- Remove sensitive data from exceptions.
- Never log sensitive data.
- Remove sensitive data from memory after use.

Exceptions like the `FileNotFoundException` can contain data like filenames or pathnames that can be used by attackers to infiltrate a system. This can happen if the `java.io.FileInputStream` constructor is called and it tries to read a system file that may not exist. The thrown exception can expose the underlying files or directory structure of the system, which can then be a target for further attacks.

Exceptions thrown may also change their outputs in future when underlying libraries that they access may change with more information. This means that a safe exception could in future expose system details that could cause a vulnerability.

Certain sensitive data, like Social Security Numbers (SSNs), must never be kept in memory longer than necessary or logged. If an application uses a character array to store SSNs, those arrays need to be purged immediately after use. Similarly, parsing libraries might be logging the data they parse which could include SSNs. For this reason, programmers need to be careful about what data is being parsed by the chosen libraries and ensure it isn't confidential information.

After any processing of sensitive data, memory containing it must be zeroed right away so that the confidential data is not the target of debugging, confidentiality or debugging attacks. However, this may not always be possible given that certain libraries may keep copies of this data in other parts of memory. Also, adding these security measures to code can reduce the quality of the code.

Objective:
Secure Coding in Java SE Application

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

References:

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

Examples of Non-Secure Code

Question #20 of 50

Question ID: 1327797

Which class correctly uses the `final` keyword?

✗ **A)**
```
final class ClassB {
  final String value;
  void setValue(String value) { this.value = value; }
}
```

✓ **B)**
```
final class ClassD {
  final String value;
  ClassD() {value = "default"; }
  ClassD(String value) {this.value = value; }
}
```

✗ **C)**
```
final class ClassC {
  final String value = "default";
  abstract void setValue(String value);
}
```

✗ **D)**
```
final abstract class ClassA {
  final String value = "default";
}
```

Explanation

`ClassD` correctly uses the final keyword as follows:

```
final class ClassD {
  final String value;
  ClassD() {value = "default"; }
  ClassD(String value) {this.value = value; }
}
```

The `final` keyword in the class declaration indicates that `ClassD` cannot be extended. The `final` keyword, when applied to the instance field value, indicates its value cannot be modified after initialization. A final instance field does not need to be set when it is declared, only before the object is initialized. In `ClassD`, the value field is set in one of the overloaded constructors.

`ClassA` does not correctly use the `final` keyword. The `final` and `abstract` keywords cannot be applied to the same class or method, because an abstract class must be extended for instantiation and its abstract methods must be overridden. The declaration of the final instance field value is correct, however.

`ClassB` does not correctly use the final keyword. In the `setValue` method, the final instance field value is set. A final instance field cannot be set after initialization. Also, since value is not assigned at declaration or initialization, `ClassB` will not compile.

`ClassC` does not correctly use the `final` keyword. The `final` keyword cannot be applied to a class that has an abstract method because the class must be extended for a concrete subclass to override the `setValue` method. Also, a class with an abstract method must be declared with the `abstract` keyword.

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 #21 of 50

Question ID: 1328182

You have a Java application that provides directory listing using the following code:

```
class MisterLister {
  public static void main(String[] args) throws Exception {
    String directory = System.getProperty("dir");
    Runtime run = Runtime.getRuntime();
    Process prc = run.exec("cmd.exe /C dir " + directory);
    int res = prc.waitFor();
    if (res != 0) {
      System.out.println("Process failed by: " + res);
    }
    InputStream input = (res == 0) ? prc.getInputStream() : prc.getErrorStream();
    int i;
    while ((i = input.read()) != -1) {
     System.out.print((char) ch);
    }
  }
}
```

What would you use to secure this application? (Choose all that apply.)

✗ **A)** `Collections.unmodifiable`

✗ **B)** `Identity.hashmap`

✓ **C)** `File.list()`

✗ **D)** `ServiceLoader`

✓ **E)** **`Pattern.matches()`**

Explanation

You can use either `Pattern.matches()` to sanitize user input or use `File.list()` instead of using `Runtime.exec()`.

To sanitize user input, you would use the following code before the arguments are passed to the `Runtime.exec()` method:

```
if (!Pattern.matches("[0-9A-Za-z@.]+", directory)) {
  // Error handling code
```

```
}
```

The following code illustrates a more secure directory listing application using the `File.list()` method:

```
import java.io.File;

class MisterLister2 {
  public static void main(String[] args) throws Exception {
    File directory = new File(System.getProperty("dir"));
    if (!directory.isDirectory()) {
      System.out.println("Directory Error!");
    } else {
      for (String filename : dir.list()) {
        System.out.println(filename);
      }
    }
  }
}
```

Using `Collections.unmodifiable` is an incorrect option. You do this to protect collections.

Using `ServiceLoader` is an incorrect option. For any service to be used successfully, each of its service providers is required to be found and then loaded. For this purpose, the `ServiceLoader` class exists and takes care of the same.

Using `Identity.hashmap` is an incorrect option. You use this when needing to secure identity equality operations.

You need to validate the parameters to methods for securing applications. Three ways to do this are to validate inputs to methods, validate outputs from untrusted objects as inputs, and create wrappers for native methods.

- When validating inputs, take care to avoid integer overflows in input values and addition of (`../`) as arguments, which can be used to traverse directories.
- Return values of certain methods need to be checked; for example, attackers could use `ClassLoader` instances passed as return values.
- Native code needs to be placed in a wrapper so that only its functionality is exposed and not its internal working and parameters. This way input validation for the native code can be done by the Java wrapper. The following demonstrates how this can be implemented:

```
public final class NativeCodeWrapper {
  // native method kept private
  private native void nativeMethod(byte[] data, int offset, int len);
  // wrapper method performs checks
  public void methodChecker(byte[] data, int offset, int len) {
    // Here we check the parameters passed to the native method and throw
    // an argument if there is a problem with the arguments
    // we now call the native method and pass the validated arguments
```

```
    nativeMethod(data, offset, len);
  }
}
```

Mutability can cause security issues in a Java application. Some ways to ensure that doesn't happen is to follow these guidelines:

- Keep value types immutable. To do this, you declare fields `final` so that the object cannot be modified post construction.
- Copy mutable output values. You should create a copy of an internal mutable object if a method returns a reference to it.
- Make copies of input values input values for mutables and subclasses. Attackers can take advantage of a TOCTOU (Time-of-check Time-of-use) inconsistency where an object from an untrusted source may pass a `SecurityManager` check but different values may then be used by the object. To avoid this risk, you need to create a copy of the input object and then perform method logic on the copy and not the actual object.
- Provide copy functionality for mutable classes. You need to create a means to safely copy instances of a mutable class. You do this by making a static creation method, a copy constructor, or a `public` copy method for all final classes.
- Check identity equality. Identity equality methods like `Object.equals` could be overidden so that an object can be made to look *equal* to another object even when it isn't. A way to safeguard against this is to use `Identity.HashMap` or other collection implementations for ensuring identity equality.
- Check input and output to and from untrusted objects. When an object is passed to untrusted objects, the data in the object being passed should be cloned before passing by an appropriate `clone()` method. Similarly, when an object is received from an untrusted source it needs to be copied and validated before being passed on.
- Create wrapper methods for internal state code. If a class has a state that will be accessed publicly, then you need to create a `public` wrapper method around it and make it a `private` field. Similarly, if you have a state in a class that will be accessed by subclasses then you need to make it a `private` field and create a `protected` wrapper method for it.
- Finalize `public static` fields. `public static` fields need to be declared as `final` to ensure that they are not accessed and modified by attackers. When using arrays or lists, use methods like `unmodifiableList` to ensure that the list isn't modified and so secured.
- Hide mutable statics. `private static` members should be treated as if they were `public` to ensure safety of code.
- Hide collections that can be modified. Collections exposed to other classes should be made read-only or unmodifiable. You create unmodifiable collections using `of/ofEntries`, `copyOf`, or `Collections.unmodifiable` methods.

Objective:
Secure Coding in Java SE Application

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

References:

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

Sanitizing User Inputs

Question #22 of 50

Question ID: 1327775

Given:

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

Which code fragment will output `Good, Dog`?

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

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

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

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

Explanation

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

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

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

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

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

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

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

Objective:
Working with Java Data Types

Sub-Objective:
Handle text using String and StringBuilder classes

References:

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Numbers and Strings > Manipulating Characters in a String

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

Question #23 of 50

Question ID: 1327891

Given these classes in independent files:

```
public class PlayableCharacter {
   protected int level;
   public void levelUp(int experience) {
     level = Math.round(experience / 1_000);
   }
   public int getLevel() { return level;}
}

public class StandardClass extends PlayableCharacter {
   __________ void levelUp(int experience) {
     level = Math.round((float) Math.sqrt((experience + 500) / 500));
   }
}
```

Which access modifier should be inserted to compile `StandardClass`?

✓ **A)** `public`

✗ **B)** `protected`

✗ **C)** `private`

✗ **D)** `package`

Explanation

The access modifier `public` should be inserted to compile `StandardClass`. When overriding a method, the access modifier must be the same as or less restrictive than the original method. Public access is the least restrictive, so the overriding the `levelUp` method requires the `public` access modifier.

The access modifiers `private` or `protected` should not be inserted to compile `StandardClass` because the access modifier for the overriding method cannot be more restrictive than the original method. The `private` modifier restricts access to the member within the containing class, while the `protected` modifier expands access to subclasses and classes within the same package.

There is no `package` access modifier. If no access modifier is specified, then the effective access is package-level. The `package` keyword is reserved for declaring package names.

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 > Controlling Access to Members of a Class

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

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

---

Question #24 of 50

Question ID: 1328044

Given the following code:

```
public void copy(Path srcFile, Path destFile) throws _____________{
   byte[] readBytes = Files.readAllBytes(srcFile);
   Files.write(destFile, readBytes);
}
```

Which insertion will allow the code to compile?

✗ **A)** `IOError`

✗ **B)** **`FileNotFoundException`**

✓ **C)** `IOException`

✗ **D)** `FileSystemNotFoundException`

✗ **E)** `Error`

Explanation

To allow the code to compile, the insertion should specify the class `IOException` or its superclass `Exception`. The `readAllBytes` and write methods specify that they throw `IOException`, so any invocations of those methods must 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 `Error`, `RuntimeException`, and their subclasses.

The insertion should not specify `Error` or `IOError` to allow the code to compile. The `readAllBytes` and `write` methods require handling the `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 insertion should not specify `FileNotFoundException` to allow the code to compile. The `readAllBytes` and `write` methods require handling `IOException` or its superclass `Exception`. Subclasses of the `IOException` can be specified in the `throws` clause, but the more general `IOException` must be also specified, or the code will not compile.

The insertion should not specify `FileSystemNotFoundException` to allow the code to compile. 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 > Specifying the Exceptions Thrown by a Method

---

Question #25 of 50

Question ID: 1328003

Consider the following code:

```
public abstract class Jedi {
 public abstract void useTheForce();
 @Deprecated
 public void fightTheEmpire() { /*Implementation omitted */}
 }
}
```

```
class LastJedi extends Jedi {
 @Override
 public void useTheForce() { /*Implementation omitted */}

 public static void main(String[] args) {
  Jedi rey = new LastJedi();
  rey.fightTheEmpire();
 }
}
```

During compilation, a warning is generated. What should you do to compile without warning messages displayed?

- ✗ **A)** Decorate the main method with `@SuppressWarnings("overrides")`
- ✓ **B)** Decorate the `main` method with `@SuppressWarnings("deprecations")`
- ✗ **C)** Decorate the `fightTheEmpire` method with `@SuppressWarnings("overrides")`
- ✗ **D)** Decorate the `fightTheEmpire` method with `@SuppressWarnings("deprecations")`

Explanation

You would decorate the `main` method with `@SuppressWarnings("deprecations")` as follows:

```
@SuppressWarnings("deprecations")
public static void main(String[] args) {
 Jedi rey = new LastJedi();
 rey.fightTheEmpire();
}
```

This is because the `fightTheEmpire` method in the `Jedi` class is marked with the `@Deprecated` annotation and the method is being used by the `rev` object of the `Jedi` class. This would generate a warning during compilation. Marking the `main` method with `@SupressWarnings` will allow compilation without any alerts.

The other options are incorrect because the `main` method needs to be marked with `@SupressWarnings` for deprecations not overrides for compilation warnings to be avoided. You would use `@SupressWarnings` for overrides if there was a compiler warning regarding overriding methods. In this scenario, the `useTheForce` method is already decorated by the `@Override` annotation.

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

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

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

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

Objective:
Annotations

Sub-Objective:
Create, apply, and process annotations

References:

Oracle Technology Network > Java SE Documentation > Annotations

---

Question #26 of 50

Question ID: 1327836

Given:

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

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

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

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

Explanation

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

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

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

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

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

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

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

Objective:
Java Object-Oriented Approach

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

References:

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

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

Question #27 of 50

Question ID: 1327757

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

✗ **A)** `int i1 = 40.4;`

✗ **B)** `boolean b2 = 1;`

✗ **C)** `int i2 = -6,000;`

✗ **D)** `var v;`

✓ **E)** `float f2 = 10.01E2f;`

✓ **F)** `boolean b1 = (6 < 4);`

✓ **G)** `var v = true;`

✗ **H)** `float f1 = 10.01;`

Explanation

The following three declaration statements correctly initialize their variables:

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

float f2 = 10.01E2f;

var v = true;
```

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

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

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

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

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

initialization.

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

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

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

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

Objective:
Working with Java Data Types

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

References:

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

OpenJDK > Java Local Variable Type Inference: Frequently Asked Questions

---

Question #28 of 50

Question ID: 1327900

Given the following interfaces:

```
interface Chargeable {
   void plugin();
   void unplug();
}

interface BluetoothSupport {
   void connectDevice(String device);
   void disconnectDevice();
}
```

```
interface WiFiSupport {
   void connectNetwork(String sid);
   void disconnectNetwork();
}

interface Touchable {
   void tap();
   void swipe();
}

interface TouchScreen extends Touchable, Chargeable {}
```

Which of the following classes will compile?

✗ **A)**
```
class Tablet2 implements TouchScreen {
        void tap() {}
        void swipe() {}
        void plugin() {}
        void unplug() {}
    }
```

✗ **B)**
```
class Tablet4 implements TouchScreen, WiFiSupport {
        public void tap() {}
        public void swipe() {}
        public void plugin() {}
        public void unplug() {}
    }
```

✓ **C)**
```
abstract class Tablet1 implements TouchScreen, WiFiSupport,
    BluetoothSupport {}
```

✗ **D)**
```
abstract class Tablet3 implements TouchScreen {
        abstract void tap();
        abstract void swipe();
        abstract void plugin();
        abstract void unplug();
    }
```

Explanation

The following class will compile:

```
abstract class Tablet1 implements TouchScreen, WiFiSupport, BluetoothSupport {}
```

`Tablet1` will compile because it is declared `abstract` and does not require explicit implementation of the `TouchScreen`, `WiFiSupport`, and `BluetoothSupport` interfaces. Abstract classes cannot be instantiated because they rely on concrete subclasses to override any abstract methods.

To be a concrete class, all methods from the `TouchScreen`, `BluetoothSupport`, and `WiFiSupport` interface must be overridden. Because `TouchScreen` inherits the abstract methods from `Touchable` and `Chargeable`, concrete classes must override these methods as well.

`Tablet2` and `Tablet3` will not compile. The methods of `TouchScreen` are implemented as required in `Tablet2` and declared as abstract in `Tablet3`, but interface methods are implicitly public. An overriding method cannot restrict access further than the overridden method. By default, class members without an access modifier are accessible only within the same package.

`Tablet4` will not compile. `Tablet4` must also override the methods in `WiFiSupport`, namely `connectNetwork` and `disconnectNetwork`. Otherwise, `Tablet4` must be declared as abstract.

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 #29 of 50

Question ID: 1327875

Given:

```
public class SuperString {
   public String toString() {
     return "Super String";
   }
   public Object toString(String str) {
     return "Super " + str;
   }
}

class SubString extends SuperString {
   public String toString() {
     return "Sub String";
```

```
   }
   public String toString(String str) {
     return "Sub " + str;
   }
}
```

Which two statements will generate the output `Super String`?

✗ **A)** `System.out.println(new SubString().toString());`

✗ **B)** `System.out.println(((Object) new SubString()).toString());`

✓ **C)** `System.out.println(new SuperString().toString("String"));`

✗ **D)**
```
System.out.println(((Object) new
SubString()).toString("String"));
```

✓ **E)** `System.out.println(((Object) new SuperString()).toString());`

✗ **F)**
```
System.out.println(((Object) new
SuperString()).toString("String"));
```

Explanation

The following two statements will generate the output `Super String`:

```
System.out.println(new SuperString().toString("String"));
```

```
System.out.println(((Object) new SuperString()).toString());
```

The first statement instantiates the `SuperString` class and then invokes the second overloaded version of the `toString` method. By feeding the literal `String` as the argument for the parameter, the output is `Super String`. The second statement instantiates the `SuperString` class, casts the reference as an `Object` type, and then invokes the first version of the overloaded `toString` method. Because the actual object is a `SuperString` object, the implementation in the `SuperString` class is executed.

The statements that instantiate the `SubString` class and invoke the parameterless version of `toString` will not generate the output `Super String`. This is because no versions of the `toString` method in `SubString` class return the text `Super, only Sub`.

The statements that cast the `SuperString` or `SubString` object to an `Object` type and then invoke the `toString` method with a `String` argument will not generate the output `Super String`. The statements will fail to compile because the `Object` class does not provide a `toString` method with a `String` parameter.

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 > Overriding and Hiding Methods

---

Question #30 of 50

Question ID: 1327776

Given:

```
String str1 = "salt";

String str2 = "sAlT";
```

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

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

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

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

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

✗ **E)**
```
if (str1 == str2.toLowerCase() )
        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 #31 of 50

Question ID: 1328200

Consider the following code:

```
LocalDate day = LocalDate.of(2020, Month.AUGUST, 28);
day.plusDays(30);
System.out.println(day);
```

Which of the following is the correct output?

✗ **A)** `FRIDAY`

✓ **B)** `2020-08-28`

✗ **C)** `MONDAY`

✗ **D)** `2020-09-28`

Explanation

The output is `2020-08-28` because Java date time classes are immutable. Once created, the object is not changed. Even though the day object of the `LocalDate` class has the `plusDays()` method performed on it, it still references the object that was instantiated with the date of August 28th , 2020. You could reassign `LocalDate` variable to the return value from the `plusDays` method, however.

The other options are incorrect because the `LocalDate` object is not modified in this example. Also, the output would not be the day of the week. You would use the following code to retrieve the day of the week using the `LocalDate` variable:

`System.out.println(day.getDayOfWeek());`

Because the `LocalDate` object is not modified, the value would be the day of the week for August 28th, 2020, which is the output `FRIDAY`.

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 #32 of 50

Question ID: 1327824

Given:

```
public class PrinterClass {
   private String arg;
   public PrinterClass() { System.out.println("no args"); }
   public PrinterClass(String arg) {
     this();
     this.arg = arg;
     System.out.println("arg");
   }
   public void print() {
     System.out.println(arg);
   }
   public static void main(String[] args) {
      new PrinterClass().print();
   }
}
```

What is the result?

✗ **A)** `arg`

✓ **B)** `no args`
`null`

✗ **C)** `no args`

✗ **D)** `arg`
`null`

Explanation

The result is the following output:

```
no args
null
```

The code in the main method invokes the parameterless `PrinterClass` constructor and then invokes the `print` method. The constructor prints `no args`, and because this constructor does not set the `arg` field, the `print` method prints `null`.

The result will not include the output `arg` because the overloaded constructor with a `String` parameter is not invoked. The parameterless `PrinterClass` constructor is invoked.

The result will not omit the output `null` because the `arg` field is not set. The default value is printed because the parameterless `PrinterClass` constructor is invoked.

Objective:
Java Object-Oriented Approach

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

References:

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

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

---

Question #33 of 50

Question ID: 1327906

You define the following interface to handle photographic items:

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

```
    }
    return result;
  }
}
```

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

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

- ✗ **A)** `List<Photograph> makePhotographs(Scene ... scenes)`
- ✗ **B)** `public List<Photograph> makePhotographs(Scene ... scenes)`
- ✓ **C)** `default List<Photograph> makePhotographs(Scene ... scenes)`
- ✗ **D)** `List<Photograph> makePhotographs(Scene ... scenes);`
- ✗ **E)** `static 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 #34 of 50

Question ID: 1327818

Given:

```
public class Circle {
   public double getCircumference(double radius ) {
     return java.lang.Math.PI * 2 * radius;
   }
   public static double getArea(double radius) {
     return java.lang.Math.PI * radius * radius;
   }
}
```

Which two code fragments will fail compilation?

✗ **A)** `new Circle().getCircumference(10.5);`

✗ **B)** `new Circle().getArea(5.5);`

✗ **C)** `double a = new Circle().getArea(5.5);`

✗ **D)** `Circle.getArea(5.5);`

✓ **E)** `Circle.getCircumference(10.5);`

✓ **F)** `double c = Circle.getCircumference(10.5);`

Explanation

The following code fragments will fail compilation:

```
Circle.getCircumference();
double c = Circle.getCircumference(10.5);
```

These code fragments fail compilation because the `getCircumference` method is an instance method and cannot be invoked as a `static` class method. The code must instantiate the `Circle` class before accessing the `getCircumference` method.

The other code fragments will compile because the `getCircumference` method is accessible from an instance context and the `getArea` method is accessible from either a `static` or instance context.

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 #35 of 50

Question ID: 1328131

Consider the following stream of electronic component objects:

```
parts.stream
  .filter(part =>
  part.isWattage("1/4'))
  .collect(Collectors.toList());
```

Which of the following should you use to store the returned results of the above code?

✗ **A)** `LocalDate diode`

✗ **B)** `char diodes`

✗ **C)** `int diodes`

✓ **D)** `List<parts> diodes`

Explanation

You should use `List<parts> diodes` to store the returned results of the code. The `collect` method returns a collection of objects filtered from a stream using a `Predicate`.

You should not use either `int` or `char` because the `collect` method returns a collection of filtered objects which cannot be stored in primitive data types.

You should not use the `LocalDate` object because `LocalDate` is used to store date and time information and would not be compatible with our scenario.

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 #36 of 50

Question ID: 1327966

Which statement is true about standard streams?

✗ **A)** The standard output and error streams cannot be diverted from the console.

✗ **B)** `System.err` is the only stream available for console output.

✓ **C)** The Java platform supports three standard streams.

✗ **D)** All standard streams are character streams.

Explanation

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

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

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

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

Objective:
Java File I/O

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

References:

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

Question #37 of 50

Question ID: 1328023

Given the following:

```
String s1 = "espresso";
String s2 = "java";
```

```
int i = 1;
boolean b = true;
```

Which two expressions are valid in a `while` statement?

✗ **A)** `b != "false"`

✗ **B)** `i = 2`

✓ **C)** `i == 2`

✓ **D)** `s1.equals(s2)`

✗ **E)** `i <= b`

✗ **F)** `s1.compareTo(s2)`

Explanation

The expressions `s1.equals(s2)` and `i == 2` are valid in a `while` statement. To be a valid expression in a `while` statement, the expression must evaluate to a `boolean` value. Each of these expressions checks for equality, using the `equals` method and equality operator (==), respectively.

The expression `s1.compareTo(s2)` is not valid in a `while` statement because this expression will not evaluate to a `boolean` value. The `compareTo` method returns 0 if both objects are equal, positive if the specified object comes before the instance, or negative if the specified object comes after the instance.

The expression `i = 2` is not valid in a `while` statement because this expression will not evaluate to a `boolean` value. This expression uses the assignment operator (=), not the equality operator (==).

The expressions `b != "false"` and `i <= b` are not valid in a `while` statement because these expressions will not compile. In the first expression, `boolean` and `String` are incomparable types. In the second expression, `int` and `boolean` are incomparable types.

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 #38 of 50

Question ID: 1328010

Consider the following code:

```
public @interface Wizards {
 int age() default 300;
 String wizardName();
}

//Insert code here
public class Merlin {
 void magic() {
  System.out.println("abracadabra!");
 }
```

What code will you use to apply the annotation `Wizards` to `Merlin`?

✗ **A)** `@Override Wizards (age=777, wizardName="Merlin")`

✓ **B)** `@Wizards (age=777, wizardName="Merlin ")`

✗ **C)** `@interface Wizards (age=777, wizardName="Merlin")`

✗ **D)** `@Inherited Wizards (age=777, wizardName="Merlin")`

Explanation

You should use the following code:

`@Wizards(age=777, wizardName="Merlin")`

This allows you to set specific element values for the custom annotation `Wizards`.

The `@interface` option is incorrect. You use `@interface` when defining a custom annotation.

`@Inherited` is an incorrect option. You use the `@Inherited` annotation when you want subclasses to inherit custom annotations of a parent class.

`@Override` is an incorrect option. 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.

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

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

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

The other options referencing `@Deprecated`, `@Override`, and `@Inherited` annotations are syntactically incorrect and do not provide custom annotations. These are among Java's predefined annotation types.

Objective:

Annotations

Sub-Objective:

Create, apply, and process annotations

References:

Oracle Technology Network > Java SE Documentation > Annotations

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

---

Question #39 of 50

Question ID: 1327974

Given:

```
01  import java.io.*;
02  public class FileCopier {
03    public static void main(String[] args) throws IOException{
04      try (
05       FileInputStream in = new FileInputStream(args[0]);
06       FileOutputStream out = new FileOutputStream(args[1])) {
07        int b;
08        while ((b = in.read()) != 0) out.write(b);
09      } catch (IOException ex) {
10        throw ex;
11      }
12    }
13  }
```

And the commands:

```
javac FileCopier.java
java FileCopier orig.txt dest.txt
```

Assuming that `orig.txt` and `dest.txt` exist and are not empty, what is the most likely result?

- ✗ **A)** An exception is thrown at runtime.
- ✓ **B)** The program goes into an infinite loop.
- ✗ **C)** The content of `dest.txt` is overwritten with the content of `orig.txt`.
- ✗ **D)** The content of `orig.txt` is appended to the content of `dest.txt`.

Explanation

The most likely result is this program goes into an infinite loop. The `read` method in `FileInputStream` returns the next byte in the file or -1 for the end of file. The `while` statement on line `08` checks for the value 0, rather than the value -1. This statement will loop until the ASCII null character is read, even if the end of the file is already reached. Most text documents do not contain the ASCII null character, so the program will continue past the end of the file in most cases.

The most likely result is not the content of `dest.txt` is overwritten with the content of `orig.txt`, because neither stream is closed while the program goes into an infinite loop. This would be the result if the `while` statement on line `08` was rewritten as follows:

```
while ((b = in.read()) != -1) out.write(b);
```

The most likely result is not the content of `orig.txt` is appended to the content of `dest.txt`, because a `boolean` type is not specified in the `FileOutputStream` constructor on line `06` for appending data. By default, the content of the target file `dest.txt` will be overwritten, rather than appended.

The most likely result is not an exception is thrown at runtime. This could occur if either file is unavailable or the security manager denies read or write access to a respective file.

Objective:
Java File I/O

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

References:

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

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Essential Classes > Basic I/O > Reading, Writing, and Creating Files

---

Question #40 of 50

Question ID: 1327778

Given:

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

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

✗ **A)** 4
✓ **B)** 20
✗ **C)** 16
✗ **D)** 12

Explanation

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

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

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

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

Objective:
Working with Java Data Types

Sub-Objective:
Handle text using String and StringBuilder classes

References:

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

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

---

Question #41 of 50

Question ID: 1327803

Given the following class:

```
1.   public class Machine {
2.     static String manufacturer;
3.     public static void main (String[] args) {
4.       //Insert code here
5.     }
6.   }
```

Which three code fragments correctly assign a value to the manufacturer field at line 4?

✓ **A)** `Machine.manufacturer = "Oracle";`

✓ **B)** `manufacturer = "Oracle";`

✗ **C)** `this.manufacturer = "Oracle";`

✓ **D)**
```
Machine myMachine = new Machine();
myMachine.manufacturer = "Oracle";
```

✗ **E)**
```
super.manufacturer = "Oracle";
```

Explanation

The following code fragments correctly assign a value to the manufacturer field at line 4:

```
manufacturer = "Oracle";
```

```
Machine.manufacturer = "Oracle";
```

```
Machine myMachine = new Machine();
myMachine.manufacturer = "Oracle";
```

All three code fragments access the `manufacturer` field as a `static` member. `static` members do not require class instantiation for access.

The first code fragment accesses the `manufacturer` field directly within the same class because the main method is also a `static` member. The second code fragment is the preferred approach to accessing `static` members outside the class. This code fragment specifies the class name `Machine` and `manufacturer` field using dot notation.

The third code fragment is confusing because it accesses the `static` member as if it were an instance member. Although this is syntactically correct, this approach is not recommended. This code fragment will compile because both `static` and instance members are available to objects.

The code fragment that specifies the keyword `this` does not correctly assign a value to the `manufacturer` field. The `this` keyword references the current instance, which is unavailable in a static context.

The code fragment that specifies the keyword `super` does not correctly assign a value to the `manufacturer` field. The `super` keyword references the parent class associated with the current instance, which is unavailable in a static context.

Objective:
Java Object-Oriented Approach

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

References:

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

---

Question #42 of 50

Question ID: 1327860

Given these classes in independent files:

```
01   abstract public class Animal {
02    public String name;
03    public abstract void makeNoise();
04   }
05   public class Monkey extends Animal {
06    public void makeNoise() {System.out.println("ooh ooh ahh ahh"); }
07    public static void main (String[] args) {
08     Animal a = new Monkey();
09     a.makeNoise();
10    }
11   }
```

What is the result?

- ✓ **A)** `ooh ooh ahh ahh`
- ✗ **B)** `null`
- ✗ **C)** Compilation fails due to an error on line 01.
- ✗ **D)** Compilation fails due to an error on line 09.
- ✗ **E)** Compilation fails due to an error on line 08.
- ✗ **F)** Compilation fails due to an error on line 02.

Explanation

The result is the output `ooh ooh ahh ahh`. The `Animal` class is declared as abstract, requiring a concrete subclass to implement `makeNoise`. The `Monkey` class implements `makeNoise` on line 06 and invokes that method using polymorphism in lines 08 and 09.

The result is not `null`. The `Monkey` class is instantiated, and its overridden method is invoked in lines 08 and 09.

Compilation will not fail due to an error on line 01 or 02. Although convention has the `abstract` keyword preceding the access modifier `public`, Java syntax allows the keywords to be in either order.

Compilation will not fail due to an error on line 08. Although `Animal` cannot be instantiated because it is abstract, this class is a valid variable type.

Compilation will not fail due to an error on line 09. Using polymorphism, the implementation of `makeNoise` in `Monkey` will be invoked.

Objective:
Java Object-Oriented Approach

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

References:

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

---

Question #43 of 50

Question ID: 1328103

Given this class:

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

Which of the following two code statements will compile?

✗ **A)** `BiFunction<String, Integer> cH = Tests::countLower;`

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

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

✓ **D)** `ToIntFunction<String> cP = Tests::countLower;`

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

✗ **F)** `String s = "abcdEFGH"; Supplier<Integer> s1 = s::countLower;`

Explanation

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

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

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

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

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

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

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

Objective:
Working with Streams and Lambda expressions

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

References:

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

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

Question #44 of 50

Question ID: 1327987

Given the code fragment:

```
Path sPath = Paths.get("C:\\Documents\\JavaProjects\\Java11\\Upgrade.txt");
Path dPath = Paths.get("C:\\Documents\\JavaProjects\\Java11\\NIO\\src");
try {
  Files.move(sPath, dPath, StandardCopyOption.ATOMIC_MOVE);
} catch (IOException ex) {
  System.err.println("Error Happened!");
}
```

Assuming the file `Upgrade.txt` exists in both the source and destination path, what is the result of compiling and executing this code?

- ✓ **A)** The `Upgrade.txt` file is removed from the `Java11` directory, placed in the `NIO` sub-directory, and renamed `src` in a single operation.
- ✗ **B)** The `Upgrade.txt` file is copied from the `Java11` directory, placed in the `NIO` sub-directory, and renamed `src` in three distinct operations.
- ✗ **C)** The `Upgrade.txt` file is copied from the `Java11` directory, placed in the `NIO` sub-directory, and renamed `src` in a single operation.
- ✗ **D)** The `Upgrade.txt` file is removed from the `Java11` directory and placed in the `NIO\src` sub-directory in a single operation. The existing `Upgrade.txt` file is appended.
- ✗ **E)** The `Upgrade.txt` file is removed from the `Java11` directory, placed in the `NIO` sub-directory, and renamed `src` in three distinct operations.
- ✗ **F)** The `Upgrade.txt` file is removed from the `Java11` directory and placed in the `NIO\src` sub-directory in two distinct operations. The existing `Upgrade.txt` file is overwritten.

Explanation

The result of compiling and executing this code is the `Upgrade.txt` file is removed from the `Java11` directory, placed in the `NIO` sub-directory, and renamed `src` in a single operation. The `Files.move` operation removes the original file, while the `StandardCopyOption` value `ATOMIC_MOVE` ensures that the performed steps are performed as a single operation. In the `Files.move` and `Files.copy` methods, both `Path` arguments include the filename. In this scenario, the last entry in the destination `Path` object is *src*, so this is the new name of the moved file.

The `Upgrade.txt` file is not removed from the `Java11` directory and placed in the `NIO\src` sub-directory, because `src` is the destination filename. Both `Path` arguments include the filename, so that the last entry is the filename for both source and destination. If the same filename existed in the destination, the existing file would be overwritten only if the `StandardCopyOption` value `REPLACE_EXISTING` was specified. Otherwise, an exception would be thrown.

The result is not performed in two or three distinct operations, because `StandardCopyOption.ATOMIC_MOVE` is specified as an argument for the `Files.move` method. The `StandardCopyOption` value `ATOMIC_MOVE` ensures that the performed steps are performed as a single operation.

The `Upgrade.txt` file is not copied from the `Java11` directory because the `Files.move` method removes the source file. The `Files.copy` method does not affect the source file, while the `Files.move` does.

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 > Enum StandardCopyOption

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

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Essential Classes > Basic I/O > Reading, Writing, and Creating Files

Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Essential Classes > Basic I/O > Managing Metadata

---

Question #45 of 50

Question ID: 1328138

Given the code fragment:

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

Which best describes the result?

- ✗ **A)** The numbers 1 to 1000 in an unpredictable order
- ✗ **B)** The numbers 0 to 999 in sequential order
- ✓ **C)** The numbers 0 to 999 in an unpredictable order
- ✗ **D)** The numbers 1 to 1000 in sequential order

Explanation

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

```
119
120
121
122
123
124
0
1
2
3
4
5
```

```
6
7
8
9
10
```

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

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

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

Objective:
Working with Streams and Lambda expressions

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

References:

The Java Tutorials > Collections > Aggregate Operations > Parallelism

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

---

Question #46 of 50

Question ID: 1327812

Given the following code line:

```
printToConsole(1.2);
```

Which overloaded method is invoked?

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

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

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

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

Explanation

The following overloaded method is invoked:

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

This method is invoked because the default primitive type for a literal fractional value is `double`, and this type is automatically boxed as a `Double` object. Thus, the overloaded method with a `Double` parameter is invoked. When a method is invoked on an overloaded method, only the list of parameter data types determines which method is executed.

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

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

The overloaded method with an `Object` parameter is not invoked. All objects inherit from the `Object` class, so this overloaded method is an effective catch-all for an argument whose data type is not explicitly declared in another overloaded method.

Objective:
Java Object-Oriented Approach

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

References:

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

Question #47 of 50

Question ID: 1327765

Given the following:

```
int i = 10;
```

Which two expressions evaluate to 3?

✗ **A)** `(i + 5 - 6) * 10 / 5`

✗ **B)** `i + (5 - 6 * 10) / 5`

✓ **C)** `(i + 5) - 6 * 10 / 5`

✓ **D)** `(i + 5) - 6 * (10 / 5)`

✗ **E)** `i + (5 - 6) * 10 / 5`

✗ **F)** `((i + 5 - 6) * 10) / 5`

Explanation

The following two expressions evaluate to 3:

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

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

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

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

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

1. Postfix unary: `num++`, `num--` (value change only occurs *after* overall expression is evaluated)
2. Prefix unary: `++num`, `--num`, `+num`, `-num`, `~` `!`
3. Multiply, Divide, Modulus: `*` `/` `%`
4. Add, Subtract: `+` `-`
5. Shift: `<<` `>>` `>>>`

6. Relational: < > <= >= instanceof
7. Equality: == !=
8. Bitwise AND: &
9. Bitwise exclusive OR: ^
10. Bitwise inclusive OR: |
11. Logical AND: &&
12. Logical OR: ||
13. Ternary: ? :
14. Assignment: = += -= *= /= %= &= ^= |= <<= >>= >>>=

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

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

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

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

Objective:
Working with Java Data Types

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

References:

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

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

---

Question #48 of 50

Question ID: 1327789

Given the following:

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

Which statement(s) should be inserted in the code to output `30`?

✓ **A)** `System.out.println(modify(i));`

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

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

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

Explanation

The statement `System.out.println(modify(i));` should be inserted in the code to output `30`. This code invokes the `modify` method with the primitive value 10 and outputs the return value of 30. Inside the modify method, the argument is incremented by 10 (20) and then returns 10 plus its value (30).

The statement `System.out.println(modify(i + 10));` should not be inserted in the code to output `30`. This code will output `40` because the argument for the `i` parameter in the `modify` method is 20.

The statements `modify(i); System.out.println(i);` should not be inserted in the code to output `30`. The output of this code will be `10` because the local variable `i` is unchanged after invoking the `modify` method. Primitive values are copied from variable to variable, not references.

The statements `modify(i); System.out.println(i + 10);` should not be inserted in the code to output `30`. The output of this code will be `20` because the local variable `i` is unchanged after invoking the `modify` method and the argument for the `println` method is 10 + 10 (20).

Objective:
Java Object-Oriented Approach

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

References:

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

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

---

Question #49 of 50

Question ID: 1328122

Given:

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

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

✗ **A)** `.sum(System.out::println)`

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

✓ **C)** `:.count()`

✗ **D)** `.max()`

Explanation

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

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

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

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

Objective:
Working with Streams and Lambda expressions

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

References:

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

---

Question #50 of 50

Question ID: 1327965

Which code fragment outputs `160.11` to the console?

✓ **A)**
```
String s = "1";
System.out.print(10.1 + 100 + 30 + 20 + s);
```

✗ **B)**
```
String s = new String("1");
System.out.print(10.1 + 100 + s + 30 + 20);
```

✗ **C)**
```
String s = new String("1");
System.out.print(10.1 + 100 + 30 + s + 20);
```

✗ **D)**
```
String s = "1";
System.out.print(s + 10.1 + 100 + 30 + 20);
```

Explanation

The following code fragment will output `160.11` to the console:

```
String s = "1";
System.out.print(10.1 + 100 + 30 + 20 + s);
```

The additive operator (+) performs both arithmetic addition and string concatenation, depending on the data type of its operands. The first four numeric literals will be added together with the result of `160.1`, but because the last value is of type `String`, the value 1 is appended to 160.1 before output is produced.

The following code fragments will not output `160.11`:

```
String s = "1";
System.out.print(s + 10.1 + 100 + 30 + 20);
```

This code fragment will output `110.11003020`. Because the first operand is of type `String`, all remaining operands will be concatenated, not added as numerical values.

The following code fragments will not output `160.11`:

```
String s = new String("1");
System.out.print(10.1 + 100 + s + 30 + 20);
```

This code fragment will output `110.113020`. Because the third operand is of type `String`, all remaining operands will be concatenated, not added as numerical values. Only the first two arguments will be added before the concatenation operation.

The following code fragments will not output `160.11`:

```
String s = new String("1");
System.out.print(10.1 + 100 + 30 + s + 20);
```

This code fragment will output `140.1120`. Because the fourth operand is of type `String`, the remaining operand will be concatenated, not added as a numerical value. The first three arguments will be added before the concatenation operation.

Objective:
Java File I/O

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

References:

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

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