Quick Quiz September 7, 2022

Test ID: 222846403

Question #1 of 50

Question ID: 1328016

Given the following:

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

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

✗ **A)** 1

✗ **B)** 10

✓ **C) 0**

✗ **D)** 5

Explanation

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

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

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

Objective:
Controlling Program Flow

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

References:

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

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

---

Question #2 of 50

Question ID: 1328031

Given the following output:

```
8
5
2
-1
```

Which code fragment generates this output?

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

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

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

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

Explanation

The following code fragment will generate the required output:

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

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

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

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

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

Objective:
Controlling Program Flow

Sub-Objective:

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

References:

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

---

Question #3 of 50

Question ID: 1327840

Given:

```
public class Pedometer {
   private double stride;
   private double[] measurements;
}
```

Which code fragment is a method that meets good encapsulation principles?

✗ **A)**
```
public void getStride(double stride) {
   stride = this.stride;
}
```

✗ **B)**
```
public double[] getMeasurements() {
   return this.measurements;
}
```

✓ **C)**
```
public double[] getMeasurements() {
   return measurements.clone();
}
```

✗ **D)**
```
public void getStride(double stride) {
   this.stride = stride;
}
```

Explanation

The following code fragment is a method that meets good encapsulation principles:

```
public double[] getMeasurements() {
   return measurements.clone();
}
```

In this code fragment, the accessor method uses the `clone` method to return a copy of the `measurements` field, rather than a reference to the field itself. Accessor methods should return copies of field objects. A copy of a field will prevent other classes from modifying fields directly and will require going through mutator methods.

The two versions of the accessor method `getStride` do not meet good encapsulation principles. Neither method returns the value of the `stride` field as expected, but instead modifies the `stride` field like a mutator method. Also, the statement `stride = this.stride;` overwrites the `stride` parameter, rather than assigning the `stride` parameter to the `stride` field as expected.

The accessor method `getMeasurements` that returns a direct reference to the `measurements` field object does not meet good encapsulation principles. Accessor methods should not return direct references to field objects. Direct access will allow other classes to modify fields directly without going through mutator methods.

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 #4 of 50

Question ID: 1328139

You are writing code for a social networking website. Your site has over a million subscribers. You create a stream of User objects from a collection of over 1 million visitors:

```
visitors.stream()
```

Which `Stream` method(s) will return a single value from the entire stream?

- ✗ **A)** `parallelStream`
- ✗ **B)** `readAllLines`
- ✓ **C)** **`max`**
- ✓ **D)** **`min`**
- ✓ **E)** **`collect`**
- ✗ **F)** **`get`**
- ✓ **G)** **`count`**

Explanation

The correct options are the `collect`, `min`, `max`, and `count` methods. These are reduction operations that combine streams to return a single value. Reduction operations also include the methods `average` and `sum`.

`parallelStream` is not a correct option as it actually allows you to create parallel streams in your code in place of the regular stream method. Like the standard stream method, `parallelStream` returns a sequential series of values, but the Java streams API decompose queries automatically to make use of multiple processor cores.

The option `readAllLines` is incorrect because it is a method used by the `Files` class to read data from a file path.

The option `get` is incorrect because it is method of the `Path` class used to get the file path of a specified file.

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 #5 of 50

Question ID: 1327876

Given:

```
public class SuperString {
   public String toString() {
     return "Super String 1";
   }
   public Object toString(String str) {
     return "Super String 2";
   }
}

class SubString extends SuperString {
   public String toString() {
     return "Sub String 1";
   }
   public String toString(String str) {
     return "Sub String 2";
   }
   public static void main(String[] args) {
     SuperString string = new SubString();
     System.out.println(string);
```

```
    }
}
```

What is the result?

- ✗ **A)** An exception is thrown at run time.
- ✗ **B)** `Super String 1`
- ✓ **C)** **`Sub String 1`**
- ✗ **D)** `Super String 2`
- ✗ **E)** Compilation fails.
- ✗ **F)** `Sub String 2`

Explanation

The result is the output `Sub String 1`. The `String` object is a `SuperString` reference to an instantiated `SubString` object. To output the `SubString` object, the parameterless `toString` method is invoked. Because the actual object is a `SubString` object, the implementation in the `SubString` class is executed.

The output is not `Sub String 2` because no argument is specified. The second version of the overloaded `toString` method would be invoked only if there was an argument provided.

The output is not `Super String 1` or `Super String 2` because the actual object is a `SubString` object. The reference type does not determine which overridden method is executed.

Compilation does not fail, nor is an exception thrown at run time. The code is syntactically and semantically correct. An overriding method can return a subclass of the return type as is the case for the second version of the overloaded `toString` method.

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 #6 of 50

Question ID: 1327912

Which two types are valid functional interfaces?

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

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

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

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

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

Explanation

The following two types are valid functional interfaces:

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

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

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

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

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

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

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

Objective:
Java Object-Oriented Approach

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

References:

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

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

---

Question #7 of 50

Question ID: 1327880

Given:

```
public abstract class Writer {
   public static void write() {System.out.println("Writing...");}
}

public class Author extends Writer {
   public static void write() {System.out.println("Writing book");}
}

public class Programmer extends Writer {
   public static void write() {System.out.println("Writing code");}
   public static void main(String[] args) {
     Writer w = new Programmer();
     w.write();
   }
}
```

What is the result?

- ✗ **A)** `Writing code`
- ✗ **B)** `Writing book`
- ✗ **C)** An exception is thrown at run time.
- ✗ **D)** **Compilation fails.**
- ✓ **E)** `Writing...`

Explanation

The result is the output `Writing ...`. This is because the reference type determines which implementation of the `write` method is executed if class hiding is used. Because the reference type is `Writer` and the write method is `static`, the implementation that is in the `Writer` class is executed.

The result is not the output `Writing code`. This is because polymorphism does not apply when members are hidden. Polymorphism applies when instance methods are overridden.

The result is not the output `Writing book`. This is because `Author` is not the reference type.

Compilation does not fail, nor is an exception thrown at run time. The code is syntactically and semantically correct. A reference type can be a supertype of the object type.

Objective:
Java Object-Oriented Approach

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

References:

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

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

---

Question #8 of 50

Question ID: 1328150

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

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

Explanation

Both the `submit` and `execute` methods support a single `Runnable` argument. When executing a task using the `ExecutorService` class, you can invoke either the `execute` or `submit` methods. The `execute` method supports `Runnable` objects that do not return a value, while the `submit` method supports both `Runnable` and `Callable` objects. The `submit` method returns a `Future` object representing the pending results of the task.

The `call` method is not provided by the `ExecutorService` class. The `call` method of the `Callable` interface returns a result or throws an exception.

The `run` method is not provided by the `ExecutorService` class. The `run` method of the `Runnable` interface performs a task but does not return a result or contain an exception in its declaration.

The `start` method is not provided by the `ExecutorService` class. The `start` method of the `Thread` class executes a thread as a child of the current thread.

Objective:

Concurrency

Sub-Objective:

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

References:

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

Oracle Technology Network > Java SE > Java SE Documentation > Java Platform Standard Edition 11 API Specification > java.util.concurrent > Interface Callable<V>

Question #9 of 50

Question ID: 1327827

Given:

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

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

What is the result?

✗ **A)** `Sub 1`

✗ **B)** `Sub 3`

✓ **C)**
```
Super
Sub 3
```

✗ **D)**
```
Super
Sub 1
```

Explanation

The result is the following output:

```
Super
Sub 3
```

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

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

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

Objective:
Java Object-Oriented Approach

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

References:

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

---

Question #10 of 50

Question ID: 1328075

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

✗ **A)** Declares a method with a single parameter

✗ **B)** Declares a method called `produce`

✗ **C)** Declares a method with a `void` return type

✓ **D) Declares exactly one method**

✗ **E)** Declares a method called `supply`

Explanation

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

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

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

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

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

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

Objective:
Working with Streams and Lambda expressions

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

References:

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

---

Question #11 of 50

Question ID: 1327800

Given:

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

and:

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

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

✗ **A)** `new Duck()`

✗ **B)** `Turkey.new Duck()`

✗ **C)** `new Turkey.Duck()`

✓ **D)** **`new Turkey().new Duck()`**

Explanation

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

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

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

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

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

Objective:
Java Object-Oriented Approach

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

References:

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

---

Question #12 of 50

Question ID: 1328054

Given:

```
public class ExceptionSelection {
   private Exception ex;
   public ExceptionSelection(Exception ex) {
     this.ex = ex;
```

```
  }
  public void throwException() throws Exception {
    System.out.println("Method started...");
    throw ex;
  }
  public static void main (String[] args) throws Exception {
    ExceptionSelection exObj =
     new ExceptionSelection(new UnsupportedOperationException());
    exObj.throwException();
    System.out.println("Method ended.");
  }
}
```

What is the most likely output?

✗ **A)**
```
Exception in thread "main"
java.lang.UnsupportedOperationException
  at java11.ExceptionSelection.main(ExceptionSelection.java:21)
Method ended.
```

✗ **B)**
```
Exception in thread "main"
java.lang.UnsupportedOperationException
  at java11.ExceptionSelection.main(ExceptionSelection.java:21)
```

✗ **C)**
```
Method started...
Exception in thread "main"
java.lang.UnsupportedOperationException
  at java11.ExceptionSelection.main(ExceptionSelection.java:21)
Method ended.
```

✓ **D)**
```
Method started...
Exception in thread "main"
java.lang.UnsupportedOperationException
  at java11.ExceptionSelection.main(ExceptionSelection.java:21)
```

Explanation

The following output is most likely:

```
Method started...
Exception in thread "main" java.lang.UnsupportedOperationException
  at java11.ExceptionSelection.main(ExceptionSelection.java:21)
```

In this code, the `throwException` method throws an `UnsupportedOperationException` after printing the output `Method started...`. By throwing an exception, the `throwException` method returns prematurely. Because the

exception is not caught in the `main` method, the exception forces the execution to end prematurely before the last `println` method is invoked.

The output that does not include `Method started...` is not likely. This is because the code executes normally and prints this output, until an exception is thrown and propagated up the call stack.

The output that includes `Method ended.` is not likely. This is because code prematurely exits the `main` method before this output is printed because the exception is not caught beforehand.

Objective:
Exception Handling

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

References:

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

---

Question #13 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 ");
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");
```

✗ **B)**
```
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");
}
```

✓ **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

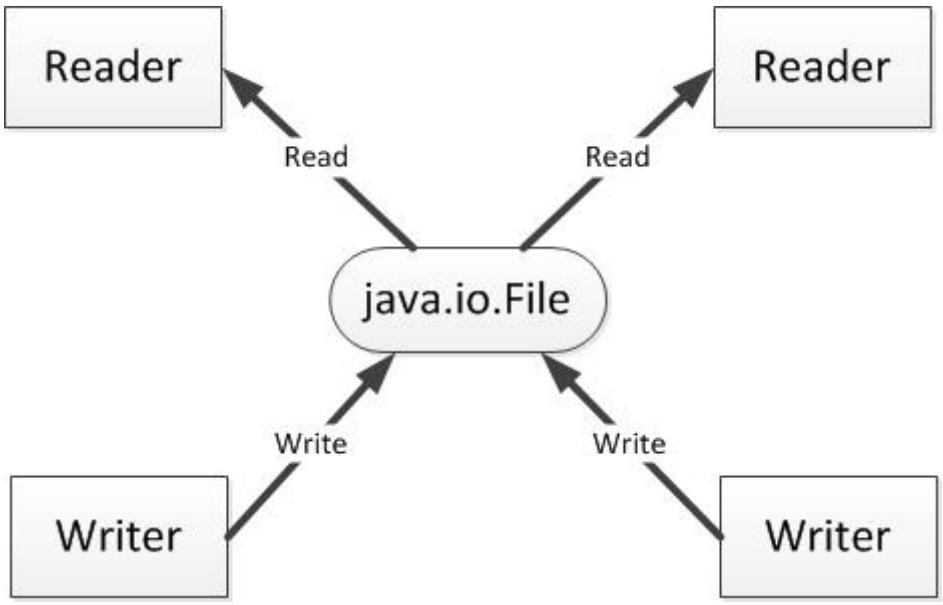
Oracle Technology Network > Java SE > Java SE Documentation > The Java Tutorials > Learning the Java Language > Language Basics > Equality, Relational and Conditional Operators

---

Question #14 of 50

Question ID: 1328173

Given:



The methods that the `Reader` and `Writer` threads require are synchronized. The `Reader` and `Writer` threads are blocked from accessing the `File` are unable to complete their required tasks. Which term best describes the threading issue?

- ✗ **A)** Atomicity
- ✗ **B)** Starvation
- ✓ **C)** Deadlock
- ✗ **D)** **Livelock**

Explanation

Deadlock best describes the situation. Deadlock occurs when two or more threads are blocked and waiting for access to the same resource and are unable to complete their tasks, because they are waiting for each other to complete.

Atomicity does not describe the situation. Atomicity describes actions that are indivisible and cannot be stopped prematurely before they complete. Atomicity reduces the likelihood of memory consistency errors.

Livelock does not describe the situation. Livelock occurs when two or more threads are so busy responding to each other that they are unable to complete their tasks. There is no actual blocking that occurs in a livelock scenario.

Starvation does not describe the situation. Starvation occurs when one or more threads access a shared resource so frequently that other threads are unable to gain needed access.

Objective:
Concurrency

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

References:

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

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

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

---

Question #15 of 50

Question ID: 1328063

Given:

```
public class BadFileLocationException extends InvalidPathException {
  public BadFileLocationException(String input) { super(input, null); }
}
```

Which two catch blocks catch the exceptions `FileNotFoundException` and `BadFileLocationException`?

✓ **A)**
```
catch (FileNotFoundException | BadFileLocationException ex) {
  //handle exceptions
}
```

✗ **B)**
```
catch (InvalidPathException & FileNotFoundException ex) {
  //handle exceptions
}
```

✗ **C)**
```
catch (IllegalArgumentException ex) {
  //handle exceptions
}
```

✗ **D)**
```
catch (FileNotFoundException & BadFileLocationException ex) {
  //handle exceptions
}
```

✗ **E)**
```
catch (IOException ex) {
  //handle exceptions
}
```

✓ **F)**
```
catch (InvalidPathException | FileNotFoundException ex) {
  //handle exceptions
}
```

Explanation

The following catch blocks catch the exceptions `FileNotFoundException` and `BadFileLocationException`:

```
catch (FileNotFoundException | BadFileLocationException ex) {
  //handle exceptions
}

catch (InvalidPathException | FileNotFoundException ex) {
  //handle exceptions
}
```

Because `BadFileLocationException` extends `InvalidPathException`, the catch clause can specify either `BadFileLocationException` or one of its superclasses, such as `InvalidPathException`. Java 8 has introduced the vertical bar (`|`) to separate exception classes in a multi-catch statement, so that more than one exception type can be handled by the same catch block. Exception classes specified in a multi-catch statement cannot be subclasses. The order of the exception classes in the multi-catch statement does not affect its execution.

The ampersand (`&`) is not a valid operator for separating exception classes in a catch statement. The ampersand (`&`) is used for bitwise AND operations.

The `IOException` should not be the only exception type specified in the catch statement. A `FileNotFoundException` will be caught, but a `BadFileLocationException` will not be caught.

The `IllegalArgumentException` should not be the only exception type specified in the catch statement. A `BadFileLocationException` will be caught, but a `FileNotFoundException` will not be caught.

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 Blocks

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

Oracle Technology Network > Java SE > Java SE Documentation > Java Platform Standard Edition

Oracle Technology Network > Java SE > Java SE Documentation > Java Platform Standard Edition 8 API Specification > java.nio.file > Class InvalidPathException

---

Question #16 of 50

Question ID: 1328161

Given the following code:

```
import java.util.*;
import java.util.concurrent.*;

public class ListThread extends Thread {
  private List<Integer> list_copy;
  public ListThread (List<Integer> list) {
    list_copy = list;
  }
  @Override
  public void run() {
    for(Integer i : list_copy) {
      try {
        String thName = Thread.currentThread().getName();
        list_copy.remove(i);
        System.out.println( list_copy + "(" + thName + ") ");
      } catch (Exception ex) {
        System.err.println(ex.getClass());
      }
    }
  }
  public static void main(String[] args) {
    List<Integer> list = new ArrayList<>();
    list.add(1);list.add(2);list.add(3);
    list.add(4);list.add(5);list.add(6);
    Thread th1 = new ListThread(list);
    Thread th2 = new ListThread(list);
    th1.start();th2.start();
  }
```

```
}
```

Which output fragment is the most likely included in the result?

✗ **A)**
```
class java.lang.ArrayIndexOutOfBoundsException
```

✗ **B)**
```
[2, 3, 4, 5, 6](Thread-0)
[3, 4, 5, 6](Thread-1)
[4, 5, 6](Thread-0)
[5, 6](Thread-1)
[6](Thread-0)
](Thread-1)
```

✓ **C)**
```
Exception in thread "Thread-1"
java.util.ConcurrentModificationException
Exception in thread "Thread-0"
java.util.ConcurrentModificationException
```

✗ **D)**
```
[2, 3, 4, 5, 6](Thread-0)
[3, 4, 5, 6](Thread-0)
[4, 5, 6](Thread-0)
[5, 6](Thread-0)
[6](Thread-0)
](Thread-0)
[2, 3, 4, 5, 6](Thread-1)
](Thread-1)
```

✗ **E)**
```
[2, 3, 4, 5, 6](Thread-0)
[3, 4, 5, 6](Thread-0)
[4, 5, 6](Thread-0)
[2, 3, 4, 5, 6](Thread-1)
[5, 6](Thread-1)
[5, 6](Thread-1)
[5, 6](Thread-1)
[6](Thread-1)
](Thread-1)
[5, 6](Thread-0)
](Thread-0)
```

✗ **F)**
```
class java.lang.IndexOutOfBoundsException
```

Explanation

The following output is most likely to be included in the result:

```
Exception in thread "Thread-1" java.util.ConcurrentModificationException
Exception in thread "Thread-0" java.util.ConcurrentModificationException
```

This exception is thrown by both threads because modification operations are not thread-safe in an `ArrayList` when using an iterator. To use an iterator that supports thread safety, you can either create a derived class with synchronization code or use the thread-safe variant of `ArrayList`: `CopyOnWriteArrayList`. The `CopyOnWriteArrayList` class copies its instance, so that changes are not committed until all modification actions occur. If the `CopyOnWriteArrayList` class is used in this code, then the `ConcurrentModificationException` will not be thrown and included in the output.

The output must include an exception because the `ArrayList` class does not support thread-safe operations using an iterator.

The output will not include an `IndexOutOfBoundsException` and `ArrayIndexOutOfBoundsException` because this code uses iterators, not manual index-based access.

Objective:
Concurrency

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

References:

Oracle Technology Network > Java SE > Java SE Documentation > Java Platform Standard Edition 11 API Specification > java.util.concurrent > Class CopyOnWriteArrayList<E>

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

---

Question #17 of 50

Question ID: 1328020

Examine the code fragment:

```
public static String getGradeFB (String grade) {
   String comment = null;
   switch (grade) {
     case "A":
     case "B":
       comment = "Excellent job!";
       break;
     case "C":
     case "D":
       comment = "You should try again.";
       break;
     case "F":
```

```
      comment = "You should study more.";
    default:
      throw new RuntimeException();
  }
  return comment;
}
```

Which three code statements will throw an exception?

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

Explanation

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

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

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

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

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

Objective:
Controlling Program Flow

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

References:

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

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

Question #18 of 50

Question ID: 1327950

Given the code fragment:

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

You want to produce this output:

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

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

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

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

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

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

Explanation

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

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

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

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

how elements are sorted.

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

Objective:
Working with Arrays and Collections

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

References:

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

---

Question #19 of 50

Question ID: 1327834

Given:

```
class StandardMethods {
   private StandardMethods() { System.out.println("Constructor invoked!"); }
   public static void printPerimeter(double... sides) {
     double result = 0;
     for (double side: sides) {
       result += side;
     }
     System.out.println("Perimeter is " + result);
   }
}

class MainClass {
   public static void main(String[] args) {
     new StandardMethods().printPerimeter(5,5);
   }
}
```

What is the result?

✓ **A)** Compilation fails.

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

✗ **C)**
```
Constructor invoked!
Perimeter is 10.0
```

✗ **D)** `Constructor invoked!`

✗ **E)** `Perimeter is 10.0`

Explanation

Compilation fails because the statement `new StandardMethods().printPerimeter(5,5);` attempts to access the `private` constructor of `StandardMethods`. If a class has a constructor declared as private, then that class cannot be directly instantiated. Private members are only available within the same class.

The result is not output, because compilation fails. The following output would be the result if the `private` keyword were removed:

```
Constructor invoked!
Perimeter is 10.0
```

The following output would be the result if the constructor were removed:

```
Perimeter is 10.0
```

An exception is not thrown at runtime because compilation fails before reaching the Java runtime.

Objective:
Java Object-Oriented Approach

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

References:

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

---

Question #20 of 50

Question ID: 1327831

Given:

```
class Job {
   private String name;
   private String[] reqs;
   public Job(String name, String... reqs) {
     this.name = name;
     this.reqs = reqs;
   }
   public void post() {/*Implementation omitted*/}
```

```
   //insert code here
}

class Programmer extends Job {
   private String language;
   public Programmer(String name, String... reqs) {super(name, reqs);}
   public void post(String language) {
     post();
     post(language.toCharArray());
   }
}
```

Which method, when inserted in the code, will compile?

✓ **A)** **`protected void post(char[] rawData) {/*Implementation omitted*/}`**

✗ **B)** `void post(String rawData) {/*Implementation omitted*/}`

✗ **C)** `private void post(char[] rawData) {/*Implementation omitted*/}`

✗ **D)** `public void post(String rawData) {/*Implementation omitted*/}`

Explanation

The following method, when inserted in the code, will compile:

`protected void post(char[] rawData) {/*Implementation omitted*/}`

Because this overloaded method is invoked in the subclass `Programmer` , it must be declared with an access modifier other than `private` . This method is declared with the access modifier `protected`, so that it is available only to subclasses, such as `Programmer`.

The method declared with the `private` keyword will not compile when inserted in the code. This is because the method is unavailable to the `Programmer` class. The access modifier `private` indicates that a class member is only available to that class.

The following methods, when inserted in the code, will not compile:

```
public void post(String rawData) {/*Implementation omitted*/}
void post(String rawData) {/*Implementation omitted*/}
```

Although both methods are available to the `Programmer` class, they do not match the signature required by the `Programmer` class. The statement `post(language.toCharArray());` requires a method that specifies a `char` array parameter.

Objective:
Java Object-Oriented Approach

Sub-Objective:

Understand variable scopes, apply encapsulation and make objects immutable

References:

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

---

Question #21 of 50

Question ID: 1327924

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

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

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

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

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

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

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

Explanation

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

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

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

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

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

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

Objective:
Working with Arrays and Collections

Sub-Objective:
Use generics, including wildcards

References:

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

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

---

Question #22 of 50

Question ID: 1327757

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

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

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 #23 of 50

Question ID: 1327995

```
01  try {
02    String dbURL = " jdbc:derby://localhost:1527/sample;user=test;password=p@$$w0rd";
03    Connection cn = DriverManager.getConnection(dbURL);
04    String query = "SELECT Customer_ID, Name, City, State, Zip FROM Customer";
05    Statement stmt = cn.createStatement(
06    ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
07    ResultSet rs = stmt.executeQuery(query);
08    rs.next(); rs.previous(); rs.next();
09    System.out.println(rs.getInt(1) + "-" + rs.getString(2));
10  } catch (SQLException ex) {
11    System.err.println("ERROR!");
12  }
```

Assume the data table is named `Customer` and accessible using the `dbURL` variable. What is the result of compiling and executing this code?

- ✗ **A)** `ERROR!`
- ✓ **B)** **`1-JumboCom`**
- ✗ **C)** Compilation fails.
- ✗ **D)** `2-Livermore Enterprises`

Explanation

The result of compiling and executing this code is as follows:

```
1-JumboCom
```

On lines `05-06`, the result set is declared as scrollable and read-only. A scrollable result set supports arbitrary backward and forward movements. By default, the cursor of a result set points before the first row. On line `08`, the first `next` method moves the cursor to the first row, the `previous` method moves the cursor before the first row, and then the second `next` method moves the cursor back to the first row again. Thus, the retrieved column values are from the first row.

Compilation will not fail because this code fragment is syntactically correct.

The result will not be `2-Livermore Enterprises`. This would be the result if the `previous` method was not invoked on line `08`.

The result will not be `ERROR!` because no exception is thrown. An exception would be thrown if the result set was not scrollable as declared on lines `05-06` or the `next` method was invoked only once on line `08`.

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 > Retrieving and Modifying Values from Result Sets

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

---

Question #24 of 50

Question ID: 1327820

Given:

```
class Fruit {}

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

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

- ✓ **A)** `new Coconut(3);`
- ✓ **B)** **`new Fruit();`**
- ✗ **C)** `new Fruit(3, "Costa Rica");`
- ✓ **D)** `new Coconut(3, "Costa Rica");`
- ✗ **E)** `new Fruit(3);`
- ✗ **F)** `new Coconut();`

Explanation

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

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

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

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

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

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

Objective:
Java Object-Oriented Approach

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

References:

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

---

Question #25 of 50

Question ID: 1327879

Given:

```
public interface StringInterface {
   public String toString();
}

public class SuperString implements StringInterface {
   public String toString() {
     return "Super String 1";
   }
```

```
   public Object toString(String str) {
     return "Super String 2";
   }
}

class SubString extends SuperString {
   public String toString() {
     return "Sub String 1";
   }
   public String toString(String str) {
     return "Sub String 2";
   }
   public static void main(String[] args) {
     StringInterface string = new SubString();
     System.out.println(string.toString("test"));
   }
}
```

What is the result?

- ✓ **A)** Compilation fails.
- ✗ **B)** `Super String 1`
- ✗ **C)** `Super String 2`
- ✗ **D)** **`Sub String 2`**
- ✗ **E)** `Sub String 1`
- ✗ **F)** An exception is thrown at run time.

Explanation

Compilation fails, because `StringInterface` does not declare a version of `toString` that includes an `input` parameter. `StringInterface` only declares a parameterless version of `toString`.

The result is not output, because compilation fails. If the variable `string` were declared as type `SuperString` or `SubString`, then the output would be `Sub String 2`. This is because in polymorphism, the instantiated class determines which implementation method is executed.

The code does not throw an exception at run time. The code cannot be executed because it fails compilation.

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

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

---

Question #26 of 50

Question ID: 1328097

Given the code fragment:

```
/* insert here */
  .forEach(System.out::println);
```

Which code added at `/* insert here */` would result in the output 1, 2, 3, and 4?

✓ **A)** `IntStream.range(1, 5)`

✗ **B)** `IntStream.range(1, 4)`

✗ **C)** `IntStream.rangeClosed(1, 5)`

✗ **D)** `IntStream.iterate(1, 4)`

✗ **E)** `IntStream.of(1, 4)`

Explanation

The code `IntStream.range(1, 5)` will result in the output 1, 2, 3, and 4. The `IntStream.range` method takes two `int` arguments. The first argument specifies the initial value of the stream. Each subsequent stream element is then calculated as the previous value increased by one for as long as the value remains less than the second argument.

`IntStream.range(1, 4)` will result in a stream containing the numbers 1, 2, and 3.

`IntStream.of(1, 4)` creates a stream with the elements 1 and 4. The method `IntStream.of` takes a variable length argument list and creates a stream of int values containing the same elements that were provided in the array.

`IntStream.iterate(1, 4)` is not a function, so it will fail to compile. The iterate method takes two arguments. The first is an initial value, which becomes the first item in the stream. The second argument is a function that is called repeatedly to produce the next stream value based on the previous one.

`IntStream.rangeClosed(1, 5)` will output the numbers from 1 to 5 inclusive. The `rangeClosed` method is like the `range` method except that it includes the value of the second argument in the generated stream.

Objective:
Working with Streams and Lambda expressions

Sub-Objective:

Use Java Streams to filter, transform and process data

References:

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

---

Question #27 of 50

Question ID: 1328204

You are writing code for an airport terminal ticker. You output time information using the following code:

```
01  import java.time.format.DateTimeFormatter;
02  import java.time.ZonedDateTime;

03  public class airportTicker {
04    public static void main(String[] args) {
05      ZonedDateTime myZone =ZonedDateTime.from(ZonedDateTime.now());
06      String currentTime;
07      //Insert code here
08      System.out.println(currentTime);
09    }
10  }
```

The time should be in the following output format:

```
Sun, 30 Aug 2020 11:00:20 GMT
```

Which enumeration value should you specify with the `format` method on line `07`?

- ✗ **A)** `DateTimeFormatter.ISO_INSTANT`
- ✓ **B)** `DateTimeFormatter.RFC_1123_DATE_TIME`
- ✗ **C)** `DateTimeFormatter.ISO_LOCAL_DATE`
- ✗ **D)** `DateTimeFormatter.ISO_DATE`

Explanation

You should use `DateTimeFormatter.RFC_1123_DATE_TIME` as follows:

```
currentTime = myZone.format(DateTimeFormatter.RFC_1123_DATE_TIME);
```

The format `RFC_1123_DATE_TIME` provides date time information with the day of the week, date, and current time in GMT, which is useful in an international airport.

The option `DateTimeFormatter.ISO_DATE` is incorrect because it will output the date as:

```
2020-08-30+01:00
```

The option `DateTimeFormatter.ISO_LOCAL_DATE` is incorrect because it will output:

```
2020-08-30
```

The option `DateTimeFormatter.ISO_INSTANT` is incorrect because it will output:

```
2015-08-30T12:15:30Z
```

Objective:
Localization

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

References:

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

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

---

Question #28 of 50

Question ID: 1328034

Given the following:

```
public class Java11_Looping {
   public static void main(String[] args) {
     int intArray[] = { 1, 2, 3, 4, 5};
     int index = 0;
     do {
       while (index < 10)
         System.out.print(index++ + " ");
     } while (index < intArray.length);
   }
}
```

What is the result?

✓ **A)** `0 1 2 3 4 5 6 7 8 9`

✗ **B)** `0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9`

✗ **C)** Nothing is printed.

✗ **D)** `0 1 2 3 4`

Explanation

The result is the output `0 1 2 3 4 5 6 7 8 9`. The `do-while` block will execute at least once, so that the code in the inner `while` block is reached. The code in the `while` block will print the value of `index` and then increment it before the next iteration. The `while` block will end once `index` equals 10. Then the expression in the `while` statement of the `do-while` block will be evaluated. Because there are only five elements in `intArray` and the value of `index` is 10, the execution exits the `do-while` block.

The result is not that nothing is printed because code in a `do-while` block will execute at least once.

The result is not the output `0 1 2 3 4`. This output would be the result if the expressions for the `do-while` statement and inner `while` statement were reversed.

The result is not the output `0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9`. This output would not be the result unless the value for the `index` variable were reset outside of the inner `while` loop.

Objective:
Controlling Program Flow

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

References:

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

---

Question #29 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
```

✓ **B)**
```
Fred
Jim
```

✗ **C)**
```
Fred
Jim
Sheila
```

✗ **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 #30 of 50

Question ID: 1328052

Given:

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

```
   }
   public static void main(String[] args) {
     createObject();
   }
}
```

What is the result?

- ✗ **A)** An exception is thrown at run time.
- ✗ **B)** `is created!`
  `Was the object created?`
- ✗ **C)** `ExceptionFun is created!`
- ✓ **D)** Compilation fails.
- ✗ **E)** `ExceptionFun is created!`
  `Was the object created?`
- ✗ **F)** `is created!`

Explanation

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

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

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

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

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

Objective:
Exception Handling

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

References:

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

---

Question #31 of 50

Question ID: 1328013

Given the following code fragment:

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

What is the output?

✗ **A)** `10`

✗ **B)** `No`

✓ **C)** Compiler error

✗ **D)** `0`

Explanation

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

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

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

Objective:
Controlling Program Flow

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

References:

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

---

Question #32 of 50

Question ID: 1328147

Given the following code fragment:

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

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

✗ **A)** A new thread will be created for each task, with five threads available initially.

✓ **B)** Five threads will be created, each thread supporting a single task.

✗ **C)** A single thread will be created for up to five tasks.

✗ **D)** A new thread will be created for each task, up to five threads simultaneously.

✗ **E)** A new thread will be created for each fifth task.

Explanation

Because the `newScheduledThreadPool` factory method is invoked with the argument 5, five threads will be created, each thread supporting a single task. Threads not being used will be idle. The number of threads created for an `Executor` object is determined by the method used to obtain it. This thread pool supports tasks that run after a delay or are executed periodically.

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

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

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

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

Objective:
Concurrency

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

References:

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

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

---

Question #33 of 50

Question ID: 1327792

Given the following:

```
public class MyBasicClass {
   //Insert code here
}
```

Which three lines of code can be included in the class?

✗ **A)** `module basicModule {}`

✓ **B)** `void BasicMethod() {}`

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

✗ **D)** `package basicPackage;`

✗ **E)** `import java.text.*;`

✓ **F)** `static final int VAL=1000;`

Explanation

The three lines of code can be included in the class as follows:

```
public class MyBasicClass {
  enum ClassType {basic, advanced}
  void BasicMethod() {}
  static final int VAL=1000;
}
```

A class body can include static and non-static fields, methods, constructors, and nested enumerations and classes.

The code line `package basicPackage;` cannot be included in the class because a `package` statement must be the first executable line in the source file, not inside a class body.

The code line `import java.text.*;` cannot be included in the class because `import` statements are not allowed in class bodies.

The code line `module basicModule {}` cannot be included in the class source file, but must be included in a separate `module-info.java` file.

Objective:
Java Object-Oriented Approach

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

References:

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

Oracle.com > Java 9 | Excerpt > Understanding Java 9 Modules

Question #34 of 50

Question ID: 1327756

Given the following:

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

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

✓ **A)** `byte b = 0b1101;`

✗ **B)** `byte b = 1101;`

✗ **C)** `boolean b = 0;`

✗ **D)** `boolean b = null;`

Explanation

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

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

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

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

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

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

2e64-1.

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

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

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

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

Objective:
Working with Java Data Types

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

References:

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

OpenJDK > Java Local Variable Type Inference: Frequently Asked Questions

---

Question #35 of 50

Question ID: 1328099

Given the code fragment:

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

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

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

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

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

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

Explanation

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

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

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

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

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

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

Objective:
Working with Streams and Lambda expressions

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

References:

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

Question #36 of 50

Question ID: 1327807

Given the following code fragment:

```
public class StandardMethods {
   public static void printPerimeter(double... sides) {
     double result = 0;
     for (double side: sides) {
       result += side;
     }
     System.out.println("Perimeter is " + result);
```

```
    }
}
```

Assuming the given code is in the same package, which code lines will compile? (Choose all that apply.)

✓ **A)** `StandardMethods.printPerimeter();`

✓ **B)** `StandardMethods.printPerimeter(7.5, 9.8, 11);`

✗ **C)** `double perimeter = StandardMethods.printPerimeter();`

✗ **D)** `double perimeter = StandardMethods.printPerimeter(7.5, 9.8, 11);`

Explanation

The following code lines will compile:

```
StandardMethods.printPerimeter();
StandardMethods.printPerimeter(7.5, 9.8, 11);
```

Both code lines invoke the `printPerimeter` method without expecting a return value because the method declares `void`. Because varargs (`...`) is used for the `printPerimeter` method, an arbitrary number of arguments can be specified during invocation. The first code line invokes `printPerimeter` with no arguments, while the second code line invokes `printPerimeter` with three arguments.

The code lines that attempt to retrieve a return value will not compile. The `getSurfaceArea` method declares `void`, so no return value is expected. The compiler will indicate that the variable `perimeter` of the `double` type is not compatible with the `void` type.

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 > 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 > Defining Methods

---

Question #37 of 50

Question ID: 1327822

Which statement is true about default constructors?

✗ **A)** All classes are provided a default constructor by the compiler.

✗ **B)** Default constructors include only a single parameter.

✗ **C)** Superclass constructors automatically invoke default constructors of subclasses.

✓ **D)** Default constructors include an invocation of the superclass.

Explanation

Default constructors include an invocation of the superclass. If no constructor is defined for a class, then the compiler will automatically provide the default constructor. The default constructor specifies no parameters and invokes the parameterless constructor of the superclass.

Default constructors do not include a single parameter. Default constructors are parameterless.

All classes are not provided a default constructor by the compiler. If any constructor is defined by a class, then the compiler will not provide a default constructor.

Superclass constructors do not automatically invoke default constructors of subclasses. Subclass constructors that do not explicitly invoke a superclass's default constructor will automatically invoke the default constructor of the superclass.

Objective:
Java Object-Oriented Approach

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

References:

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

---

Question #38 of 50

Question ID: 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)** Line 02

✗ **B)** Line 03

✗ **C)** Line 01

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

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 #39 of 50

Question ID: 1327763

Given the following:

```
int x = 5;
```

Which two expressions evaluate to 5?

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

Explanation

The following two expressions evaluate to 5:

```
-x + 10
```

```
10 – x- -
```

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

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

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

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

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

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

Objective:
Working with Java Data Types

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

References:

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

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

Question #40 of 50

Question ID: 1327786

Given the following code fragment:

```
Customer cust = new Customer();
```

Which part of the statement instantiates the `Customer` object?

✗ **A)** `cust`

✗ **B)** `Customer cust`

✓ **C)** `new`

✗ **D)** `Customer()`

Explanation

The part of the statement that instantiates the `Customer` object is `new`. The `new` keyword is required to create the object.

`cust` is not the part of the statement that instantiates the `Customer` object. The variable name is `cust`.

`Customer cust` is not the part of the statement that instantiates the `Customer` object. The declaration part of the statement is `Customer cust`.

`Customer()` is not the part of the statement that instantiates the `Customer` object. The initialization part of the statement is `Customer()`.

Objective:
Java Object-Oriented Approach

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

References:

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

---

Question #41 of 50

Question ID: 1327936

Given the following code fragment:

```
public class TestArrayList {
   public static void main (String[] args) {
     ArrayList<String> names = new ArrayList<>(2);
```

```
    names.add("Amy");
    names.add("Anne");
    names.add("Jason");
    System.out.println(names.get(3));
  }
}
```

What is the result?

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

Explanation

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

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

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

Objective:
Working with Arrays and Collections

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

References:

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

Question #42 of 50

Question ID: 1328065

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

- ✗ **A)** `(String s, n) -> String.format(s, n)`
- ✗ **B)** `s, n -> String.format(s, n)`
- ✗ **C)** `(final s, n) -> String.format(s, n)`
- ✓ **D)** `(final String s, Number n) -> String.format(s, n)`
- ✗ **E)** `(s, n) -> n`
- ✓ **F)** `(s, n) -> s + n`

Explanation

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

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

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

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

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

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

Objective:
Working with Streams and Lambda expressions

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

References:

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

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

---

Question #43 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.getClass().isArray()`

✗ **B)** `charArray2D[1].length < 2`

✗ **C)** `charArray2D[0].getClass().isArray()`

✓ **D)** `charArray2D.length > 4`

✓ **E)** `charArray2D[0].length < 2`

✗ **F)** `charArray2D[1].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 #44 of 50

Question ID: 1328183

You have written a class to store employee records including bank account numbers.

```
public abstract class Employee {
  protected Employee() {
    //Insert code here
  }
  private Employee(Void ignored) {
    // Implementation omitted
  }
  private static Void securityCheck() {
    SecurityManager secured = System.getSecurityManager();
    if (secured != null) {
```

```
      secured.checkCreateClassLoader();
    }
    return null;
  }
}
```

What line of code will you insert to disallow malicious subclassing of this class?

✗ **A)** `Runtime run = Runtime.getRuntime();`

✗ **B)** `PreparedStatement ps = con.prepareStatement(sql);`

✓ **C)** `this(securityManagerCheck());`

✗ **D)** `System.out.println("Check this!");`

Explanation

The following line of code needs to be inserted:

`this(securityManagerCheck());`

The other options are incorrect because they do not invoke the `SecurityManager` check, which is required for securing constructors.

Secure construction of confidential classes by ensuring a check by `SecurityManager` each time a class can be instantiated. This check should be done at the beginning of each `public` or `protected` constructor in the class. Checks must also be enforced before any `public static` factory methods and `readObject` or `readObjectNoData` methods of serializable classes.

To secure sensitive objects during construction, you need to hide constructors by using `static` factory methods instead of using `public` constructors. Also, inheritance should be implemented using delegation instead of using inheritance. You also need to avoid cloning and using implicit constructors.

Objective:
Secure Coding in Java SE Application

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

References:

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

Question #45 of 50

Question ID: 1327781

Given:

```
public class Java11 {
   public static void main (String[] args) {
     StringBuilder sb = new StringBuilder("DataDataDataDataData");
     sb.delete(0,sb.length());
     System.out.println(sb.capacity());
   }
}
```

What is the output when this program is executed?

✗ **A)** `0`

✗ **B)** `16`

✓ **C)** `36`

✗ **D)** `20`

Explanation

The output is `36`. Because additional characters beyond the initial capacity have not forced the `StringBuilder` object to resize, its capacity has not changed.

The capacity of a `StringBuilder` object is 16 if not specified explicitly in the constructor as an `int` argument. 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 `DataDataDataDataData` is 20 characters in length, so the capacity is 16 + 20 or 36.

The output is not `0`. Removing characters does not modify the capacity of a `StringBuilder` object. Because the `delete` method removes all characters in the underlying `String` object, the `length` method would return 0, not the `capacity` method.

The output is not `16` or `20`. The initial capacity is 36 and would not be less unless specified explicitly in the constructor.

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 #46 of 50

Question ID: 1327869

Given the classes:

```
class Costume {
   int totalTreats;
   public void trickOrTreat() {System.out.println("Trick or treat!"); }
   public void tallyTreats() {System.out.println(totalTreats + " treats.");}
}
class ScaryCostume extends Costume {
   int totalTreats = 10;
   public void trickOrTreat() {System.out.println("Boo!"); }
}
class SuperHeroCostume extends Costume {
   int totalTreats = 20;
   public void tallyTreats() {System.out.println(totalTreats + " treats.");}
}
```

Which two statements are true?

- ✗ **A)** `ScaryCostume` and `SuperHeroCostume` set the `totalTreats` field in `Costume` to different values.
- ✓ **B)** `SuperHeroCostume` overrides the `tallyTreats` method from `Costume`.
- ✗ **C)** The `tallyTreats` method will output `0 treats for SuperHeroCostume`.
- ✗ **D)** `ScaryCostume` inherits the `trickOrTreat` method from `Costume`, but not the `tallyTreats` method.
- ✓ **E)** `ScaryCostume` and `SuperHeroCostume` hide the `totalTreats` field in `Costume`.
- ✗ **F)** The `tallyTreats` method will output `10 treats for ScaryCostume`.

Explanation

The following two statements are true about these classes:

- `SuperHeroCostume` overrides the `tallyTreats` method from `Costume`.
- `ScaryCostume` and `SuperHeroCostume` hide the `totalTreats` field in `Costume`.

Because the `trickOrTreat` method has the same signature in `SuperHeroCostume` as `trickOrTreat` in `Costume`, this method is overridden. When the `trickOrTreat` method is invoked on a `SuperHeroCostume` object, the output will be `Boo!` When `trickOrTreat` is invoked on `Costume`, the output will be `Trick or treat!`

Because `totalTreats` is declared as a field in both `ScaryCostume` and `SuperHeroCostume`, the field in `Costume` is effectively hidden in these classes. This means that any reference to `totalTreats` in `ScaryCostume` and `SuperHeroCostume` will resolve to their version of `totalTreats`, not the original field declared in `Costume`.

The `tallyTreats` method will not output `10 treats for ScaryCostume`. `ScaryCostume` inherits the `tallyTreats` method from `Costume`. Although the `totalTreats` field is hidden in `ScaryCostume`, this class does not reference the field directly. The value of `totalTreats` in `Costume` is referenced by the `tallyTreats` method. The output will be `0 treats for ScaryCostume`.

The `tallyTreats` method will not `output 0 treats for SuperHeroCostume`. `SuperHeroCostume` overrides the `tallyTreats` method from `Costume`. Because the `totalTreats` field is hidden in `SuperHeroCostume`, this class uses its version of `totalTreats` in the overridden `tallyTreats` method. The output for `SuperHeroCostume will be 20 treats`.

`ScaryCostume` does inherit the `tallyTreats` method from `Costume`. By default, all non-private fields and methods of a superclass are inherited by a subclass. `ScaryCostume` overrides `trickOrTreat`, but it also inherits `tallyTreats` from `Costume`.

`ScaryCostume` and `SuperHeroCostume` do not set the `totalTreats` field in `Costume` to different values. `ScaryCostume` and `SuperHeroCostume` hide the `totalTreats` field in `Costume`. This means that any reference to `totalTreats` in `ScaryCostume` and `SuperHeroCostume` will resolve to their version of `totalTreats`, not the original field declared in `Costume`.

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 > Inheritance

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

---

Question #47 of 50

Question ID: 1328091

Which statements are true of `Stream` operations? (Choose all that apply.)

✗ **A)** A `map` operation that receives *n* input items results in m output items where *m* <= *n*.

✗ **B)** The return type of a lambda expression provided as the argument to a `map` operation must be different from that lambda's argument type.

✓ **C)** A `map` operation that receives *n* input items results in *n* output items.

✗ **D)** A `flatMap` operation that receives *n* input items results in *m* output items, where *m* >= *n*.

✓ **E)** A lambda expression provided as the argument to a `flatMap` operation must return a single item of type `Stream`.

Explanation

A `map` operation that receives *n* input items results in *n* output items, and a lambda expression provided as the argument to a `flatMap` operation must return a single item of type Stream. The `Stream.map` operation produces one output item for each input item that it processes, but the type of that item can be the same or different. Thus, there is an *n-to-n* (1 to 1) relationship between input and output items in a map operation. The `flatMap` operation requires that the programmer provide code (usually, but not necessarily, as a lambda expression) that extracts any number of items from the input item. The items must be presented back to the `flatMap` operation as a `Stream`.

A `map` operation does not have a number of input items less than the number of output items. The number of element output by the `map` operation must be the same as the number of input elements.

The output type of a `map` operation does not have to differ from the input type. The output type of the `map` operation is independent of the input type, but it can be different, or the same.

A `flatMap` operation does not have a number of output items greater than the number of input items. The `flatMap` operation uses its argument, broadly speaking, to extract any number of items from a single item. It is common for the number of output items from the `flatMap` operation to be greater than the number of input items, but it is also possible that any given input item might result in zero output items.

Objective:
Working with Streams and Lambda expressions

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

References:

The Java Tutorials > Collections > Lesson: Aggregate Operations

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

---

Question #48 of 50

Question ID: 1327959

Consider the following code for a service client application:

```
package applic;

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

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

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

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

✗ **A)** `package package1;`

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

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

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

Explanation

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

```
uses package1.SpeakerInterface;
```

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

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

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

Objective:
Java Platform Module System

Sub-Objective:

Declare, use, and expose modules, including the use of services

References:

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

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

---

Question #49 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)** Compilation fails due to an error on line 09.
- ✗ **B)** Compilation fails due to an error on line 01.
- ✗ **C)** Compilation fails due to an error on line 08.
- ✗ **D)** `null`
- ✗ **E)** Compilation fails due to an error on line 02.
- ✓ **F)** `ooh ooh ahh ahh`

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 #50 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)** The content of `dest.txt` is overwritten with the content of `orig.txt`.

✗ **B)** The content of `orig.txt` is appended to the content of `dest.txt`.

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

✓ **D)** The program goes into an infinite loop.

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