Majustic - Evaluation Test With Answers

Time: 1 hour 30 minutes

Version 1.0, 04/07/2015

Summary

1. JAVA Questions

1.1. Consider the following program:

```
class StrEqual {
 public static void main(String[] args) {
  String s1 = "hi";
  String s2 = new String("hi");
  String s3 = "hi";

  if (s1 == s2) {
   System.out.println("s1 and s2 equal");
  } else {
   System.out.println("s1 and s2 not equal");
  }

  if (s1 == s3) {
   System.out.println("s1 and s3 equal");
  } else {
   System.out.println("s1 and s3 not equal");
  }
 }
}
```

c)
s1 and s2 not equal
s1 and s3 equal
JVM sets a constant pool in which it stores all the string constants used in the type. If two references are declared with a constant, then both refer to the same constant object. The == operator checks the similarity of objects itself (and not the values in it). Here, the first comparison is between two distinct objects, so we get s1 and s2 not equal. On the other hand, since references of s1 and s3 refer to the same object, we get s1 and s3 equal.

1.2. Consider the following program:

```
class Point2D {
 private int x, y;
 public Point2D(int x, int y) {
  x = x;
 }

 public String toString() {
  return "[" + x + ", " + y + "]";
 }

 public static void main(String[] args) {
  Point2D point = new Point2D(10, 20);
  System.out.println(point);
 }
}
```

c) [0, 0]
The assignment x = x; inside the construct reassigns the passed parameter; it does not assign the member x in Point2D. The correct way to perform the assignment is this.x = x;. Field y is not assigned, so its value remains 0.

1.3. Consider the following program:

```
class Increment {
 public static void main(String[] args) {
  Integer i = 10;
  Integer j = 11;
  Integer k = ++i; // INCR
  System.out.println("k == j is " + (k == j));
  System.out.println("k.equals(j) is " + k.equals(j));
 }
}
```

d) When executed, this program prints
k == j is true
k.equals(j) is true
The Integer objects are immutable objects. If there is an Integer object for a value that already exists, then it does not create a new object again. In other words, Java uses sharing of immutable Integer objects, so two Integer objects are equal if their values are equal (no matter if you use == operators to compare the references or use equals() method to compare the contents).

1.4. Consider the following program:

```
class ArrayCompare {
 public static void main(String[] args) {
  int[] arr1 = {
   1, 2, 3, 4, 5
  };
  int[] arr2 = {
   1, 2, 3, 4, 5
  };
  System.out.println("arr1 == arr2 is " + (arr1 == arr2));
  System.out.println("arr1.equals(arr2) is " + arr1.equals(arr2));

  System.out.println("Arrays.equals(arr1, arr2) is " + java.util.Arrays.equals(arr1,
arr2));
 }
}
```

a) arr1 == arr2 is false
arr1.equals(arr2) is false
Arrays.equals(arr1, arr2) is true
The first comparison between two array objects is carried out using the == operator, which compares object similarity so it returns false here. The equals() method, which compares this array object with the passed array object, does not compare values of the array since it is inherited from the Object class. Thus we get another false. On the other hand, the Arrays class implements various equals() methods to compare two array objects of different types; hence we get true from the last invocation.

1.5. Consider the following program:

```
class Base {
 public static void foo(Base bObj) {
  System.out.println("In Base.foo()");
  bObj.bar();
 }

 public void bar() {
  System.out.println("In Base.bar()");
 }
}
class Derived extends Base {
 public static void foo(Base bObj) {
  System.out.println("In Derived.foo()");
  bObj.bar();
 }
 public void bar() {
  System.out.println("In Derived.bar()");
 }
}
class OverrideTest {
 public static void main(String[] args) {
  Base bObj = new Derived();
  bObj.foo(bObj);
 }
}
```

b)
In Base.foo()
In Derived.bar()
A static method is resolved statically. Inside the static method, a virtual method is invoked, which is resolved dynamically.

1.6. Consider the following program:

```
class TestSwitch {
 public static void main(String[] args) {
  String[] cards = {
   "Club", "spade", " diamond ", "hearts"
  };
  for (String card: cards) {
   switch (card) {
    case "Club":
     System.out.print(" club ");
     break;
    case "Spade":
     System.out.print(" spade ");
     break;
    case "diamond":
     System.out.print(" diamond ");
     break;
    case "heart":
     System.out.print(" heart ");
     break;
    default:
     System.out.print(" none ");
   }
  }
 }
}
```

b) club none none none
Here is the description of matches for the four enumeration values:

- “club” matches with the case “Club”.
- For “Spade”, the case “spade” does not match because of the case difference (switch case match is case sensitive).
- does not match with “diamond” because case statements should exactly match and there are extra whitespaces in the original string.
- “hearts” does not match the string “heart”.

1.7. Consider the following program:

```
class Outer {
 static class Inner {
  public final String text = "Inner";
 }
}

class InnerClassAccess {
 public static void main(String[] args) {
  System.out.println( /*CODE HERE*/ );
 }
}
```

a) new Outer.Inner().text
The correct way to access fields of the static inner class is to use the inner class instance along with the outer class, so new Outer.Inner().text will do the job.

1.8. Consider the following enumeration definition:

```
enum Cards {
 CLUB, SPADE, DIAMOND, HEARTS
};

class CardsEnumTest {
 public static void main(String[] args) {
  /* TRAVERSE */
 }
}
```

a) for(Cards card : Cards.values())
System.out.print(card + " ");
The values() method of an enumeration returns the array of enumeration members.

1.9. Given these three definitions

```
interface I1 {}
interface I2 {}
abstract class C {}
```

d) class CI12 extends C implements I1, I2 {}
A class inherits another class using the extends keyword and inherits interfaces using the implements keyword.

1.10. Given these two definitions

```
interface I1 {}
interface I2 {}
```

d) interface II extends I1, I2 {}
It is possible for an interface to extend one or more interfaces. In that case, we need to use the extends keyword and separate the list of super-interfaces using commas.

1.11. Consider the following program:

```
abstract class AbstractBook {
 public String name;
}

interface Sleepy {
 public String name = "undefined";
}

class Book extends AbstractBook implements Sleepy {
 public Book(String name) {
  this.name = name;   // LINE A
 }
 public static void main(String[] args) {
  AbstractBook mathsBook = new Book("Principia Mathematica");
  System.out.println("The name of the book is " + mathsBook.name);  // LINE B
 }
}
```

c) The program will not compile and will result in a compiler error “ambiguous reference to name” in LINE A.
Since name is defined in both the base interface and the abstract class, any reference to the member name is ambiguous. The first reference to name is in the line marked with comment LINE A, so the error is flagged in this line by the compiler.

1.12. Which one of the following relationships describes the OO design concept of “composition”?

c) has-a
Composition is a design concept that refers to the has-a relationship.

1.13. Consider the following program:

```
class SimpleCounter <T> {
 private static int count = 0;
 public SimpleCounter() {
  count++;
 }
 static int getCount() {
  return count;
 }
}

class CounterTest {
 public static void main(String[] args) {
  SimpleCounter < Double > doubleCounter = new SimpleCounter < Double > ();
  SimpleCounter < Integer > intCounter = null;
  SimpleCounter rawCounter = new SimpleCounter(); // RAW
  System.out.println("SimpleCounter<Double> counter is " + doubleCounter.getCount());
  System.out.println("SimpleCounter<Integer> counter is " + intCounter.getCount());
  System.out.println("SimpleCounter counter is " + rawCounter.getCount());
 }
}
```

e) When executed, this program will print
SimpleCounter<Double> counter is 2
SimpleCounter<Integer> counter is 2
SimpleCounter counter is 2
Count is a static variable, so it belongs to the class and not to an instance. Each time constructor is invoked, count is incremented. Since two instances are created, the count value is two.

1.14. Consider the following program:

```
import java.util.regex.Pattern;

class Split {
 public static void main(String[] args) {
  String date = "10-01-2012"; // 10th January 2012 in dd-mm-yyyy format
  String[] dateParts = date.split("-");
  System.out.print("Using String.split method: ");
  for (String part: dateParts) {
   System.out.print(part + " ");
  }
  System.out.print("\nUsing regex pattern: ");
  Pattern datePattern = Pattern.compile("-");
  dateParts = datePattern.split(date);
  for (String part: dateParts) {
   System.out.print(part + " ");
  }
 }
}
```

b)
Using String.split method: 10 01 2012
Using regex pattern: 10 01 2012
Using str.split(regex) is equivalent to using Pattern.compile(regex).split(str).

1.15. Consider the following program

```
import java.util.regex.Pattern;

class Regex {
 public static void main(String[] args) {
  String pattern = "a*b+c{3}";
  String[] strings = {
   "abc", "abbccc", "aabbcc", "aaabbbccc"
  };
  for (String str: strings) {
   System.out.print(Pattern.matches(pattern, str) + " ");
  }
 }
}
```

d) false true false true
Here are the following regular expression matches for the character x:

- x* means matches with x for zero or more times.

- x+ means matches with x for one or more times.
- x{n} means match x exactly n times.
  The pattern a*b+c{3} means match a zero or more times, followed by b one or more times, and c exactly three times.
  So, here is the match for elements in the strings array:
- For "abc", the match fails, resulting in false.
- For "abbccc", the match succeeds, resulting in true.
- For "aabbcc", the match fails, resulting in false.
- For "aaabbbccc", the match succeeds, resulting in true.

1.16. Consider the following program:

```
import java.lang.*;

class InvalidValueException extends IllegalArgumentException {}
class InvalidKeyException extends IllegalArgumentException {}

class BaseClass {
 void foo() throws IllegalArgumentException {
  throw new IllegalArgumentException();
 }
}

class DeriClass extends BaseClass {
 public void foo() throws IllegalArgumentException {
  throw new InvalidValueException();
 }
}

class DeriDeriClass extends DeriClass {
 public void foo() { // LINE A
  throw new InvalidKeyException();
 }
}

class EHTest {
 public static void main(String[] args) {
  try {
   BaseClass base = new DeriDeriClass();
   base.foo();
  } catch (RuntimeException e) {
   System.out.println(e);
  }
 }
}
```

a) The program prints the following: InvalidKeyException.
It is not necessary to provide an Exception thrown by a method when the method is overriding a method defined with an exception (using the throws clause). Hence, the given program will compile successfully and it will print InvalidKeyException.

1.17. Consider the following program:

```
class Worker extends Thread {
 public void run() {
  System.out.println(Thread.currentThread().getName());
 }
}

class Master {
 public static void main(String[] args) throws InterruptedException {
  Thread.currentThread().setName("Master ");
  Thread worker = new Worker();
  worker.setName("Worker ");
  worker.start();
  Thread.currentThread().join();
  System.out.println(Thread.currentThread().getName());
 }
}
```

b) When executed, the program prints "Worker" and then the program hangs (i.e., does not terminate). The statement Thread.currentThread() in the main() method refers to the “Master” thread. Calling the join() method on itself means that the thread waits itself to complete, which would never happen, so this program hangs (and does not terminate).

2. SQL Questions

2.1. Which command is used to SELECT only one copy of each set of duplicable rows.

2.2. What is the difference between inner and outer join? Explain with example.

2.3. What is the difference between WHERE clause and HAVING clause?

2.4. How to select first 5 records from a table? Give an example.

3. Hibernate Questions

3.1. What is Hibernate Framework and what are the benefits of using it?

3.2. What is Java Persistence API (JPA)?

3.3. What is the difference between JPA and Hibernate

3.4. Name some important annotations used for Hibernate mapping?

3.5. What are different states of an entity bean?

3.6. What is difference between Hibernate save(), saveOrUpdate() and persist() methods?

4. Spring Questions

4.1. What is Spring Framework?

4.2. What do you understand by Dependency Injection?

4.3. Name some of the important Spring Modules with description?

4.4. What is a Spring Bean?

4.5. What are different scopes of Spring Bean?

4.6. What are some of the important Spring annotations you have used?

4.7. What is @Autowired annotation?

4.8. What are different types of Spring Bean autowiring?

4.9. What's the difference between @Component, @Controller, @Repository & @Service annotations in Spring?

4.10. How can we use Spring to create Restful Web Service returning JSON response (De/Serialization mechanism)?