Review Test Submission: Quiz 2: Class concepts & applications; Android basics

User	Khasiano Webb
Course	WTC-DeepDiveJava_Android-57588-nbennett-Sept2019
Test	Quiz 2: Class concepts & applications; Android basics
Started	10/9/19 3:49 PM
Submitted	10/9/19 3:51 PM
Status	Completed
Attempt Score	130 out of 130 points
Time Elapsed	2 minutes
Results Displayed	Submitted Answers, Feedback, Incorrectly Answered Questions

Question 1

10 out of 10 points

Given that **java.lang.RuntimeException** is a subclass of **java.lang.Exception**, which of the following is a valid (compilable) **try/catch/finally** statement?

Selected Answer:

```
try {
  throw new RuntimeException();
} catch (RuntimeException ex) {
  ex.printStackTrace();
} catch (Exception ex) {
  ex.printStackTrace();
} finally {
  System.out.println("Whew! Made it!");
}
```

Response Feedback: When catching exceptions using multiple **catch** blocks, and where one (or more) of the exception types being caught is a subclass of another exception type being caught, the block for the more specific type (i.e. the subclass) *must* appear before the block for the more general type (the superclass); if not, the block for the more specific type will be unreachable, and will not compile.

When catching multiple exception types using a *multi-catch* in a single catch block (e.g. **catch (Type1 | Type2 ex) {…}**), the exception types specified in the multi-catch *must not* have a subclass-superclass relationship; otherwise, the **try** statement will not compile.

If a **try** statement includes **finally**, the **finally** block must be the last block of the statement; that is, if any **catch** block follows the **finally** block, the statement cannot be compiled.

Question 2

10 out of 10 points

```
public class WeekDays {

 public enum Day {
    MON, TUE, WED, THU, FRI, SAT
 };

  public static void main(String[] args) {
    Day[] days = Day.values();
    System.out.println(days[2]);
  }

}
```

What is the output of this code?

Selected Answer: `WED`

Response The static **values()** method of an **enum** returns an array of all the values of the **enum** in the order they are specified

← OK

Feedback: When printing (or using in some other way) an **enum** as a string the default **toString()** method returns simply the name of that value exactly as specified in the **enum** class definition.

Question 3

10 out of 10 points

What is printed to the console when the following code is run?

```
String result = (true ? "Hello" : "Goodbye");
System.out.println(result);
```

Selected Answer: Hello

Response Feedback: In a ternary operator, the first expression after the question mark is returned when the first expression evaluates to true.

Question 4

10 out of 10 points

```
class Car {

  double mpg = 0;
  double horsePower = 0;

  public Car(double mpg, double horsePower) {
    this.mpg = mpg;
    this.horsePower = horsePower;
  }

  public Car(int mpg, int horsePower) {
    this.mpg = mpg;
    this.horsePower = horsePower;
  }
}

class Main {

  public static void main(String args[]) {
    Car car1 = new Car();
    System.out.println("Car1: mpg = " + car1.mpg + ", horsePower = " + car1.horsePower);
    Car car2 = new Car(25.4, 250);
    System.out.println("Car2: mpg = " + car2.mpg + ", horsePower = " + car2.horsePower);
  }
}
```

What is the output when this code is run?

Selected Answer: Code does not compile

Response Feedback: When we use the **new** keyword to create an instance of a (non-array) class, we follow that keyword with a constructor invocation. Just as a method invocation must match the signature of a method, the same is true of a constructor invocation.

If we don't define a constructor for a class, the compiler implicitly creates (in the compiled byte code, not in the original source code) a *default* constructor, with no parameters. If we define any constructors (with or without parameters), a default constructor for the class is not created by the compiler.

Question 5

10 out of 10 points

Which of the following is typically inflated to construct a tree of view components making up the UI of an Android activity?

Selected Answer: A layout resource, composed as an XML document.

Response Feedback: An Android activity may be defined purely as a Java class, or as a class that inflates an XML layout resource, in which the UI elements of the activity are defined.

10 out of 10 points

Question 6

```
class T {

  int t = 20;

  T() {
    t = 40;
  }
}

class Main {

  public static void main(String args[]) {
    T t1 = new T();
    System.out.println(t1.t);
  }
}
```

What is the output when running this code?

Selected Answer: 40

Response Feedback: The code in the constructor will be run after any fields of the class are initialized. Therefore the value that the field was initially set to was overwritten.

Question 7

10 out of 10 points

Which of the following is a valid variable declaration and assignment?

Selected Answer: a. **`float f = 'a';`**

Response Feedback: A **`char`** is actually an integer-type primitive variable, holding a 2-byte unsigned value (the only unsigned integer intrinsic type in Java, in fact). A literal **`char`** value is written either as a non-negative numeric literal (in the range from 0 to 65535, inclusive), as a single character in single quotes, or as a 2-byte Unicode code point in single quotes (e.g. **`'\u0061'`**). Thus, the literal value **`'a'`**, which we would generally think of as a **`char`**, actually represents the integral numeric value 97. (The ASCII code of the letter "a" is 97.)

Note that there's a subtle pitfall when assigning a floating point literal value (i.e one with a decimal point) to a **`float`**: Unlike integral literals, the compiler will not automatically perform a range check on a floating point value to see if it can be represented as a **`float`**. Instead, if the literal ends with "f" or "F", the compiler will attempt to treat it as a **`float`**; otherwise, it treats it as a **`double`** - and a **`double`** value cannot be assigned to a **float** without casting or conversion. Thus, a statement such as

```
float f = -1.0;
```

will not compile.

Question 8

5 out of 5 points

The bytecode produced by the Java compiler can be interpreted directly by the Android runtime environment.

Selected Answer: False

Response Feedback: Android apps can be built (at least in part) from code compiled by the Java compiler. However, Android does not use a Java VM, and thus cannot interpret the Java bytecode directly. Instead, bytecode produced by the Java compiler is translated at build time to Dalvik bytecode, and stored in **`.dex`** files (instead of **`.class`** and **`.jar`** files). The original virtual machine used on Android was the Dalvik VM; it has since been replaced by the Android Runtime (ART) virtual machine, but the bytecode format is still that used by Dalvik.

Question 9

10 out of 10 points

What is produced when the following code is compiled and executed as a Java application?

```
public class Test {

  private static int initialize(int value) {
    System.out.println("initialize");
    return value;
  }

  private static boolean test(int value, int limit) {
    System.out.println("test");
    return value < limit;
  }

  private static int next(int value) {
    System.out.println("next");
    return value + 1;
  }

  public static void main(String[] args) {
    for (int i = initialize(0); test(i, 3); i = next(i)) {
    }
    System.out.println("done");
  }

}
```

Selected Answer:

```
initialize
test
next
test
next
test
next
test
done
```

Response Feedback: In a **`for`** loop, the initialization is performed first, then the condition is tested. If the condition evaluates to **`true`**, the statement(s) controlled by the **`for`** are executed. After each such execution, the update portion of the **`for`** is executed, and the condition is tested again. When the condition evaluates to false (if ever), iteration terminates.

Question 10

10 out of 10 points

```
class Test {
  int i;
}

class Main {

  public static void main(String args[]) {
    Test t = new Test();
    System.out.println(t.i);
  }

}
```

What is the output of this code?

Selected Answer: 0

Response Feedback: Numeric primitive fields (not local variables) have a default value of 0.

Question 11

5 out of 5 points

`102` and **`0b01100110`** are literal representations (recognized by the Java compiler) of the same value.

Selected Answer: True

Response Feedback: Numeric literal values beginning with **`0b`** or **`0B`** are recognized by the Java compiler's parser as an integer value expressed in base-2 form. Without a preceding **`0`**, a numeric literal is parsed as a base-10 value.

10 out of 10 points

Question 12

```
class First {

  void display() {
    System.out.println("Inside First");
  }
}

class Second extends First {

  void display() {
    System.out.println("Inside Second");
  }
}


class Test {

  public static void main(String[] args) {
    First obj1 = new First();
    Second obj2 = new Second();

    First ref;
    ref = obj1;
    ref.display();

    ref = obj2;
    ref.display();
  }
}
```

What is the output of this program?

Selected Answer:

```
Inside First
Inside Second
```

Response Feedback: The implementation used when a method is invoked on an object depends on the object instance type, not its reference type.

Question 13

10 out of 10 points

Which of the following subclasses of **`android.view.ViewGroup`** is intended to be used for display of, and interaction with, a scrolling list of items?

Selected Answer: **`android.widget.ListView`**

Response Feedback: A **`ViewGroup`** is a subclass of **`View`** that is able to contain other views within it. The standard and support libraries for Android provide literally dozens of of different **`ViewGroup`** subclasses (including all of the standard layouts). However, there are a few that are specifically intended to present a collection of (usually) homogeneous items in a regular, one- or two-dimensional, scrollable arrangement. Among these are **`ListView`**, **`RecyclerView`**, **`GridView`**, and **`ViewPager`**.

Question 14

10 out of 10 points

Which of the following functional interfaces is most closely associated with writing concurrent (i.e. multi-threaded) Java code?

Selected Answer: d. **`java.lang.Runnable`**

Response Feedback: We can think of a **`java.lang.Runnable`** instance as a task that can be assigned to run on an existing thread or a new thread. We usually implement that interface, and create instances of the implementations, when writing code that will run on multiple threads – including Android, Swing, and JavaFX applications, most of which use one thread for UI interaction, and additional threads for long-running, non-UI tasks.

Wednesday, October 9, 2019 3:51:37 PM MDT