Final Review Questions. Typos and mistakes are possible.

1. Which of the following statements will create a string "Review Session"?
   a) String s = "Review Session"; **works and this is most common**
   b) String s = new String("Review Session"); **works and ZyBooks section 7.10 might help**
   c) String s; s = "Review Session"; **works to initial memory location to point to String value**
   d) String s = "Review "; s += "Session"; **works to append Strings. Doing so will result in a different string instance because Strings are immutable, but the pointer will be updated to the new String.**
   e) all of the above because all the above works
   f) none of the above

2. What does the following statements print to console?

```
String s1 = "Review Session";
String s2 = s1;
String s3 = s2;
s2 += " for Final";
System.out.println(s1 == s2);
System.out.println(s1 == s3);
System.out.println(s2 == s3);
```

   a) false\nfalse\nfalse\n
   b) false\ntrue\nfalse\n because s1 s2 and s3 is all being set to point to same memory location, then s2 is modified to point to a different memory location. == with Strings compare memory locations.
   c) true\nfalse\ntrue\n
   d) true\ntrue\ntrue\n
   e) none of the above

3. Which of the following gives the number of characters in String variableName?
   a) variableName.length
   b) variableName.length() because the String class implemented the length as a method to return the length of the char[]. The char[] is used under the hood to implement String.
   c) variableName.size
   d) variableName.size()
   e) none of the above

4. Which of the following accesses the (i + 1)th character in String variableName?
   a) variableName[i]
   b) variableName(i)
   c) variableName.get(i)
   d) variableName.charAt(i) because that is how the String class implemented it.
   e) variableName.indexAt(i)
   f) none of the above

5. Which of the following gives the number of ints in int[] variableName?
   a) **variableName.length because array syntax.**
   b) variableName.length()
   c) variableName.size
   d) variableName.size()
   e) none of the above

6. Which of the following accesses the (i + 1)th int in int[] variableName?
   a) **variableName[i] because array syntax.**
   b) variableName(i)
   c) variableName.get(i)
   d) variableName.intAt(i)
   e) variableName.indexAt(i)
   f) none of the above

7. Which of the following gives the number of Objects in Object[] variableName?
   a) **variableName.length because array syntax.**
   b) variableName.length()
   c) variableName.size
   d) variableName.size()
   e) none of the above

8. Which of the following gives the number of Objects in ArrayList<Object> variableName?
   a) variableName.length
   b) variableName.length()
   c) variableName.size
   d) **variableName.size() because that's how the ArrayList class implemented it.**
   e) none of the above

9. Which of the following accesses the (i + 1)th Object in ArrayList<Object> variableName?
   a) variableName[i]
   b) variableName(i)
   c) **variableName.get(i) because that's how the ArrayList class implemented it.**
   d) variableName.objectAt(i)
   e) variableName.indexAt(i)
   f) none of the above

10. What does the following statements print to console?

```
public class Main {
   public static void giveMeFive(int input) {
      input = 5;
   }
   public static void main(String[] args) {
      int input = 0;
      giveMeFive(input);
      System.out.println(input);
   }
}
```

a) **0 because giveMeFive() is called with a copy of input = 0. Then giveMeFive() is only updating the copy.**
b) 5
c) void
d) null
e) none of the above

11. What does the following statements print to console?

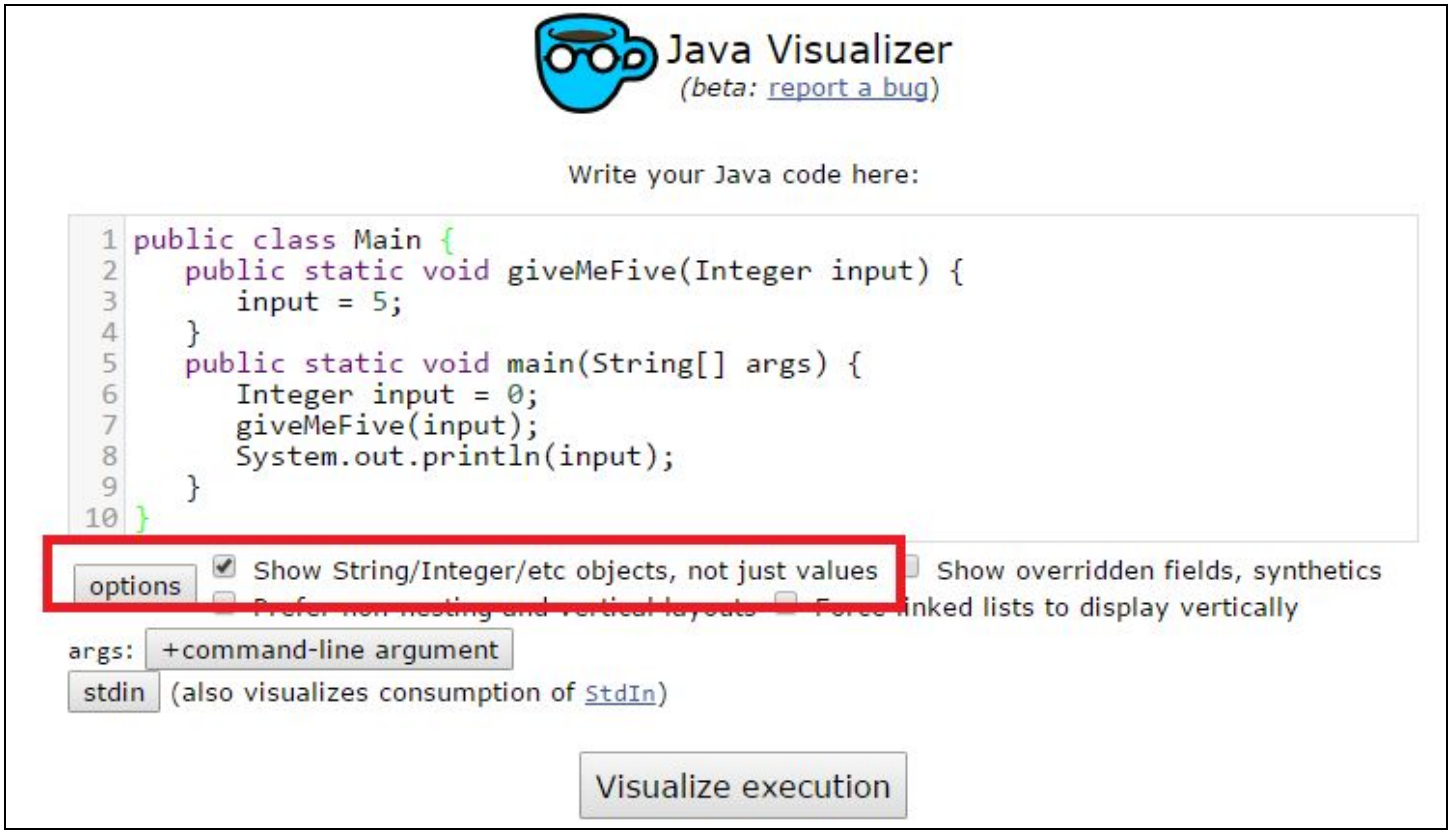
```
public class Main {
   public static int giveMeFive(int input) {
      return input = 5;
   }
   public static void main(String[] args) {
      int input = 0;
      System.out.println(giveMeFive(input) + " " + input);
   }
}
```

a) 0 0
b) 0 5
c) **5 0 because giveMeFive() is called with a copy of input = 0. Then giveMeFive() is only updating and returning the copy, which prints 5. The original input is unmodified, which prints 0.**
d) 5 5
e) none of the above

12. What does the following statements print to console?

```
public class Main {
   public static void giveMeFive(Integer input) {
      input = 5;
   }
   public static void main(String[] args) {
      Integer input = 0;
      giveMeFive(input);
      System.out.println(input);
   }
}
```

a) 0 because giveMeFive() is called with a copy of the memory location of input = 0. Then giveMeFive() is only updating the copy of the memory location to point to new Integer(5). The original memory location is still pointing to the original value, 0. For more details, run the code snippet in Java Visualizer with the show objects option turned on as shown in the screenshot below.



b) 5
c) void
d) null
e) none of the above

13. What does the following statements print to console?

```
public class Main {
   public static void giveMeFive(int[] input) {
      input[0] = 5;
   }
   public static void main(String[] args) {
      int[] input = {0}; // same as "int[] input = new int[] {0};"
      giveMeFive(input);
      System.out.println(input[0]);
   }
}
```

a) 0
b) **5 because arrays are passed in by reference, so methods can make modifications to elements of the array. See ZyBooks section 6.9 for more details.**
c) void
d) null
e) none of the above

14. What does the following statements print to console?

```
import java.util.ArrayList;
public class Main {
   public static void start(ArrayList<Integer> input) {
      input.add(5);
   }

   public static void main(String[] args) {
      ArrayList<Integer> input = new ArrayList<Integer>();
      start(input);
      System.out.println(input);
   }
}
```

a) name of the class (java.util.ArrayList) followed by the object's hexadecimal address in memory
b) 5
c) []
d) **[5] because ArrayLists are passed in by reference, so methods can make modifications to elements of the ArrayList. ArrayLists are very similar to arrays with the difference being that ArrayLists are resizeable. See ZyBooks section 6.9 for more details. ArrayLists' toString() prints the []. Please see https://docs.oracle.com/javase/7/docs/api/java/util/AbstractCollection.html#toString() for more details.**
e) none of the above

15. What does the following statements print to console?

```
public class Main {
   int x = 1;

   public void start(int x) {
      x = 2;
      System.out.print(x + " " + this.x + " ");
   }

   public static void main(String[] args) {
      int x = 3;
      new Main().start(x);
      System.out.println(x);
   }
}
```

a) 1 2 3
b) 1 2 2
c) 2 1 1
d) 2 1 3

First line of main() declares local variable x = 3
Second line of main() calls the Main constructor, which declares object variable x = 1
Second line of main() calls the start() method, and passes in a copy of local x = 3
First line of start() changes the copy of local x = 2
Second line of start() prints local copy of x (2) and the object's x (1)
Third line of main() print out local x (3).

e) none of the above

16. What does the following statements print to console?

```
public class Main {
   int x = 1;
   int y = 3;

   public void first(int x) {
      int temp = x;
      y = x / 2;
      x = temp;
   }

   void second(int x, int z) {
      x = y + 2;
      first(x);
   }

   public void start() {
      int x = 2;
      y++;
      first(y);
      second(y, x);
      System.out.println(this.y);
   }

   public static void main(String[] args) {
      new Main().start();
   }
}
```

a) 1

b) 2

First line of main() just calls start().

First line of start() declares local variable x = 3.

Second line of start() increments class's y from 3 to 4. class variable y = 4;

Third line of start() calls first(), and passes in a copy of class variable y = 4.

First line of first() declares a new variable temp = x = copy of y = 4.

Second line of first() updates class's y. class variable y = x / 2 = (copy of y) / 2 = 4 / 2 = 2.

Third line of first() updates the local copy of x, which won't matter outside of the first().

Fourth line of start() calls second(), and passed in y = 2 and x = 2.

First line of second() creates updates local x = y + 2 = 2 + 2 = 4.

Second line of second() calls first(), and passes in x = 4.

First line of first declares a new variable temp = x = 4.

Second line of first() updates class's y. Class variable y = x / 2 = 4 / 2 = 2.

Third line of first() updates local copy of x, which won't matter outside of first().

Fifth line of start() prints class variable y, which was last updated to 2.

c) 4

d) 6

e) none of the above

17. What does the following statements print to console?

```
public class Main {
   int x = 1;
   int thisX = 2;
   Integer thatX = 3;
   public Main() {
      x = 4;
      this.thisX = 5;
      Integer thatX = 6;
   }
   public void method(Integer x) {
      int temp = this.thisX;
      this.thisX = x;
      x = temp;
      System.out.print("x = " + x + "; this.thisX = " + this.thisX + "; thatX = " + thatX);
   }
   public static void main(String[] args) {
      Main main = new Main();
      Integer thatOtherX = 7;
      main.method(thatOtherX);
      System.out.println("; thatOtherX = " + thatOtherX);
   }
}
```

a) x = 5; this.thisX = 7; thatX = 3; thatOtherX = 6

b) x = 5; this.thisX = 7; thatX = 3; thatOtherX = 7

First line of main() calls the constructor Main().

The first line of Main() updates class variable x from 1 to 4.

The second line of Main() update class variable thisX from 2 to 5.

The third line of Main updates a new Integer thatX, which won't matter outside of Main().

Therefore thatX remains unmodified. thatX = 3.

Note: It has nothing to do with the fact that the 3rd line uses a reference type and not a primitive type. If "Integer" was removed from the third line of Main(), then it would've updated the class variable thatX. On the other hand, if "int" was prepended to the first line, then updating x won't matter outside of Main() as well.

Second line of main() declares new variable thatOtherX = 7.

Third line of main() calls method() with a copy to the reference to thatOtherX = 7.

First line of method() declares new variable temp = class variable thisX = 5.

Second line of method() updates class variable thisX = x = 7.

Third line of method() updates a copy to the reference to thatOtherX, x = temp = 5.

Updates to the copy to the reference to thatOtherX won't matter outside of method().

Fourth line of method() prints x = 5; thisX = 7; thatX = 3;

Fourth line of main prints thatOtherX, which is unmodified from the method() call. thatOtherX = 7

c) x = 5; this.thisX = 7; thatX = 6; thatOtherX = 6

d) x = 5; this.thisX = 7; thatX = 6; thatOtherX = 7

e) none of the above

18. What does the following statements print to console?

```
java.util.ArrayList<Character> arrayList = new
   java.util.ArrayList<Character>();
arrayList.add('a');
arrayList.add('c');
arrayList.add('c');
arrayList.add('d');
arrayList.add('c');
for (int i = 0; i < arrayList.size(); ++i) {
   if (arrayList.get(i) == 'c') {
      arrayList.remove(i);
   }
}
System.out.println(arrayList);
```

a) [a, d]

b) [a, c, d]

After add()s, arrayList = [a, c, c, d, c];
When i == 0, arrayList.get(i) != 'c', so nothing gets removed, and arrayList = [a, c, c, d, c];
When i == 1, arrayList.get(i) == 'c', so remove the element at index 1, and arrayList = [a, c, d, c];
When i == 2, arrayList.get(i) != 'c' (actually == 'd'), so nothing gets removed, and arrayList = [a, c, d, c];
When i == 3, arrayList.get(i) == 'c', so remove the element at index 3, and arrayList = [a, c, d];
Note: To remove all characters 'c' from an ArrayList, then either

1) **Start from the end of the ArrayList to the beginning,**
2) **Replace the if block with a while loop, or**
3) **Adjust the index accordingly when an element of the ArrayList is removed.**

c) [a, c, d, c]

d) name of the class (java.util.ArrayList) followed by the object's hexadecimal address in memory

e) none of the above

19. What does the following statements print to console?

```
class Animal {
   public Animal () {
      System.out.print("animalConstructor ");
   }
}
class Dog extends Animal {
   public Dog () {
      System.out.print("dogConstructor ");
   }
}
public class Main {
   public static void main(String[] args) {
      Animal dogAnimal = new Dog();
   }
}
```

a) dogConstructor
b) animalConstructor
c) animalConstructor dogConstructor

The first line of main() calls new Dog(), the Dog constructor.
Since Dog extends Animal, Animal has to be created first, which calls the Animal constructor.
The Animal constructor prints "animalConstructor "
The Dog constructor prints "dogConstructor "

d) dogConstructor animalConstructor
e) none of the above

20. What does the following statements print to console?

```
class Animal {
}
class Dog extends Animal {
}
public class Main {
   public static void main(String[] args) {
      Animal dogAnimal = new Dog();
      if (dogAnimal instanceof Dog) {
            System.out.print("dog ");
      }
      if (dogAnimal instanceof Animal) {
            System.out.print("animal ");
      }
   }
}
```

a)
b) dog
c) animal
d) dog animal

new Dog() is an instance of Dog, so the program prints "dog "
new Dog() is an instance of Animal because Dog extends Animal, so the program prints "animal "

e) none of the above

21. What does the following statements print to console?

```
class Animal {
   public boolean equals(Object other) { return false; }
}
class Dog extends Animal {
   public boolean equals(Animal other) { return true; }
}
public class Main {
   public static void main(String[] args) {
      Animal dogAnimal = new Dog();
      System.out.println(dogAnimal.equals(new Animal()));
   }
}
```

a) true
b) **false since the dog's equal method is overloads the Animal's equal method. Since dogAnimal is declared as an Animal on compile-time, the compiler sees that the Animal's equal method is the closest match. If one wants to use the Dog's equal method, then declare it as a Dog with "`Dog dogAnimal = new Dog();`".**
c) null
d) void
e) none of the above

22. What does the following statements print to console?

```
class Animal {
   public boolean equals(Object other) { return false; }
}
class Dog extends Animal {
   public boolean equals(Object other) { return true; }
}
public class Main {
   public static void main(String[] args) {
      Animal dogAnimal = new Dog();
      System.out.println(dogAnimal.equals(new Animal()));
   }
}
```

a) **true since the two methods have the same signature and the Dog's equals() method override the Animal's equals() method. If one wants to use the Animal's equal method, then avoid overriding it by instantiating an Animal with "`Animal dogAnimal = new Animal();`".**
b) false
c) null
d) void
e) none of the above

23. What does the following statements print to console?

```
class Animal {
   public int height = 1;
}
class Dog extends Animal {
   public int height = 2;
}
public class Main {
   public static void main(String[] args) {
      Animal dogAnimal = new Dog();
      Animal secondDogAnimal = dogAnimal;
      secondDogAnimal.height = 3;
      System.out.println(dogAnimal.height + " " + secondDogAnimal.height);
   }
}
```

a) 1 3
b) 2 2
c) 2 3
d) 3 3

secondDogAnimal is pointing to the same Dog. Therefore, modifying one dog's height will modify both. To have each Dog be a separate instance so that modifying one dog's height will not modify both, use "`Animal secondDogAnimal = new Dog();`" so that secondDogAnimal refers to a separate dog.

e) none of the above

24. What does the following statements print to console?

```
class Animal {
   public int height = 1;
   public void grow() {
      height++;
   }
}
public class Main {
   public static void main(String[] args) {
      Animal animal = new Animal();
      Animal secondAnimal = animal;
      secondAnimal.grow();
      System.out.println(animal.height + " " + secondAnimal.height);
   }
}
```

a) 1 1
b) 1 2
c) 2 1
d) 2 2

secondAnimal is pointing to the same Animal. Therefore, modifying one animal's height will modify both.
To have each Animal be a separate instance so that modifying one animal's height will not modify both, use "`Animal secondAnimal = new Animal();`" so that secondAnimal refers to a separate animal.

e) none of the above

25. What does the following statements print to console?

```
class Animal {
   public int height = 1;
   public void grow() {
      height++;
   }
}
public class Main {
   public static void main(String[] args) {
      Animal animal = new Animal();
      Animal secondAnimal = new Animal();
      secondAnimal.grow();
      System.out.println(animal.height + " " + secondAnimal.height);
   }
}
```

a) 1 1

b) 1 2

secondAnimal is pointing to a separate instance of Animal. Therefore, modifying one animal's height will only modify that one animal's height as expected.
To have either animal or secondAnimal modify the same height, either set them both to point to the same Animal or use the static keyword.

c) 2 1

d) 2 2

e) none of the above

26. What does the following statements print to console?

```
class Animal {
   public static int height = 1;
   public void grow() {
      height++;
   }
}
public class Main {
   public static void main(String[] args) {
      Animal animal = new Animal();
      Animal secondAnimal = new Animal();
      secondAnimal.grow();
      System.out.println(animal.height + " " + secondAnimal.height);
   }
}
```

a) 1 1

b) 1 2

c) 2 1

d) 2 2

Since height is static, all Animals share the same height variable. Modifying the height in one Animal instance will modify the height in all Animal instances.

e) none of the above

27. An object is an instance of a

a) class

Please refer to ZyBooks section 7.2.

b) program

c) method

d) data

e) none of the above

28. Multiple true/false question: Which of the followings are true about `System.out.println(...)` statement?

a) `System` is a class that we need to define

System is already defined and automatically imported from the java.lang library

b) `out` is a private reference field in `System`

out is not private because since "System.out" works (as opposed to having to do "System.getOut()").

c) `out` is a static reference field in `System`

out is static because "System.out" works without have to create an instance of System.

d) `println` is a method belongs to `System`

println is a method is belongs to PrintSteam System.out

e) `println` is an overloaded method

println accepts nothing, a boolean, a char, a char[], a double, a float, an int, a long, an Object, or a String

f) **When** passing **a** reference **variable** to **`println`,** it **prints** the **variable** class **followed** by **@** and **the** object's **address** in **memory**

Depends on the toString() method

g) Whatever passed to the `println` goes to a buffer first

Please refer to ZyBooks section 9.1.

Here are links to the javadocs for System
https://docs.oracle.com/javase/7/docs/api/java/lang/System.html and System.out
https://docs.oracle.com/javase/7/docs/api/java/lang/System.html#out

29. Multiple true/false question: Which of the followings are true about System.in.read() statement?

a) It can be used to read strings

Reads the next byte/char/int. Please refer to ZyBooks section 9.2.

b) It can cause the program to throw `FileNotFoundException`

Reads from console or OS buffer, not file

c) It reads 8-bits ASCII value when available

Reads the next byte and 8-bits is 1 byte. Please refer to ZyBooks section 9.2.

d) `System.in` can be used as an argument to a `Scanner` object to read strings

System.in reads from console or OS buffer, not Strings. Please refer to ZyBooks section 9.4 (Figure 9.4.1) to read from Strings.

30. What does the following statements print to console?

```
try {
   System.out.print("try ");
   throw new ArithmeticException();
} catch (ArithmeticException excpt) {
   System.out.print("catchArithmetic ");
   throw new ArrayIndexOutOfBoundsException();
} catch (ArrayIndexOutOfBoundsException excpt) {
   System.out.print("catchBounds ");
} finally {
   System.out.print("finally ");
}
```

a) try catchArithmetric catchBounds
b) try catchArithmetric catchBounds finally
c) try catchArithmetric
d) try catchArithmetric finally
e) none of the above

The program will first try, which throws an ArithmeticException.
Catching the ArithmeticException throws an ArrayIndexOutOfBoundsException.
Before leaving the try-catch block, finally always execute.
The program crashes due to an uncaught ArrayIndexOutOfBoundsException.
Note: Use nested try-catch blocks to handle nested exceptions.

31. What does the following statements print to console?

```
try {
   try {
      System.out.print("try ");
      throw new ArithmeticException();
   } catch (ArithmeticException excpt) {
      System.out.print("catchArithmetic ");
      throw new ArrayIndexOutOfBoundsException();
   } catch (ArrayIndexOutOfBoundsException excpt) {
      System.out.print("catchBounds ");
   } finally {
      System.out.print("finally ");
   }
} catch (Exception excpt) {
   System.out.print("catchExcpt ");
}
```

a) try catchArithmetric catchBounds catchExcpt
b) try catchArithmetric catchBounds catchExcpt finally
c) try catchArithmetric catchBounds finally
d) try catchArithmetric catchBounds finally catchExcpt
e) try catchArithmetric catchExcpt
f) try catchArithmetric catchExcpt finally
g) try catchArithmetric finally catchExcpt

The program will first try, which throws an ArithmeticException.
Catching the ArithmeticException throws an ArrayIndexOutOfBoundsException.
Before leaving the try-catch block, finally always execute.
Finally, the program catches the (ArrayIndexOutOfBounds)Exception.
Please see https://piazza.com/class/ion8bfpex9d2wo?cid=154 for more discussion.

h) none of the above

32. Which of the following lines of code to successfully write "something" to a file given PrintWriter printWriter that has been declared and initialized.
a) printWriter("something");
b) printWriter.println("something"); because that is how the PrintWriter class implemented it.
c) printWriter.out.println("something");
d) printerWriter.writeln("something");
e) none of the above

33. True or False: If private instance variables were made public, we would not need the *get* (accessor) and *set* (mutator) methods.
a) True because getters and setters won't needed if they are already visible
b) False

34. Which accessSpecifier for height will compile?

```
class Animal {
   [accessSpecifier] int height = 1;
}
public class Main {
   public static void main(String[] args) {
      System.out.println(new Animal().height);
   }
}
```

a) public
b) public and protected
c) public, protected, and no specifier

Please refer to ZyBooks section 10.2.

d) public, protected, no specifier, and private

35. Which accessSpecifier for height will compile?

Animal.java:

```
package firstPackage;

public class Animal {
   [accessSpecifier] int height = 1;
}
```

Dog.java:

```
package secondPackage;
import firstPackage.Animal;

public class Dog extends Animal {
   public Dog() {
         height = 2;
   }
}
```

Main.java:

```
package firstPackage;
import secondPackage.Dog;

public class Main {
   public static void main(String[] args) {
         System.out.println(new Dog().height);
   }
}
```

a) public
b) public and protected

Please refer to ZyBooks section 10.2.

c) public, protected, and no specifier
d) public, protected, no specifier, and private
e) none of the above

36. What does the following statements print to console?

```
abstract class Animal {
   abstract void makeSound();
}
class Dog extends Animal {
   @Override
   void makeSound() {
      System.out.print("woof ");
   }
}
class Cat extends Animal {
   @Override
   void makeSound() {
      System.out.print("meow ");
   }
}
public class Main {
   public static void main(String[] args) {
      Dog dog = new Dog();
      Cat cat = new Cat();
      Animal dogAnimal = dog;
      dog.makeSound();
      cat.makeSound();
      dogAnimal.makeSound();
   }
}
```

a) will not compile
b) meow meow meow
c) woof meow meow
d) woof meow woof because dog goes woof and cat goes meow. One cannot instantiate an instance of an abstract class and upcasting is always ok, so dogAnimal is really just a dog that goes woof as well.
e) none of the above

37. True or False: The following will compile and run despite instantiating an abstract class.

```
abstract class Animal {
   abstract void makeSound();
}
public class Main {
   public static void main(String[] args) {
      Animal animal = new Animal();
   }
}
```

a) True
b) False because abstract classes cannot be instantiated. Please refer to ZyBooks section 11.2.

38. True or False: The following will compile despite extending an abstract class without implementing the abstract method.

```
abstract class Animal {
   abstract void makeSound();
}
class CookieMonster extends Animal {
   void eat() {
      System.out.print("nom nom nom ");
   }
}
```

a) True

b) False because concrete classes that extends an abstract class must implement all inherited abstract methods. Please refer to ZyBooks section 11.2.

39. True or False: The following will compile despite creating an abstract class that extends another abstract class without implementing the abstract method.

```
abstract class Animal {
   abstract void makeSound();
}
abstract class Bird extends Animal {
   void sing() {
      System.out.print("chirp ");
   }
}
```

a) True because abstract classes can extend another abstract class without implementing all inherited abstract methods. However, the concrete class that extends either Animal or Bird will need to implement all inherited abstract methods.

b) False

40. What does the following statements print to console?

Animal.java

```
public interface Animal {
   abstract void makeSound();
}
```

Main.java

```
class Dog implements Animal {
   @Override
   public void makeSound() {
         System.out.print("woof ");
   }
}
class Cat implements Animal {
   @Override
   public void makeSound() {
         System.out.print("meow ");
   }
}
public class Main {
   public static void main(String[] args) {
         Dog dog = new Dog();
         Cat cat = new Cat();
         Animal dogAnimal = dog;
         dog.makeSound();
         cat.makeSound();
         dogAnimal.makeSound();
   }
}
```

a) will not compile
b) meow meow meow
c) woof meow meow
d) woof meow woof because dog goes woof and cat goes meow. One cannot instantiate an instance of an interface class and upcasting is always ok, so dogAnimal is really just a dog that goes woof as well.
e) none of the above

41. True or False: The following will compile despite implementing an interface without implementing the abstract method.

Animal.java

```
public interface Animal {
   abstract void makeSound();
}
```

Main.java

```
class CookieMonster implements Animal {
   void eat() {
        System.out.print("nom nom nom ");
   }
}
```

a) True

b) False because concrete classes that implemented an interface must implement all inherited abstract methods. Please refer to ZyBooks section 11.5.

42. Running the following piece of code

```
// print absolute path to file
System.out.println(file.getAbsolutePath());
// print current working directory
System.out.println(System.getProperty("user.dir"));
```

prints the following console:

```
C:\Users\usrName\workspace\ProjectName\folder\file.txt
C:\Users\usrName\workspace\ProjectName
```

Multiple true/false question: Which of the following works for declaring and initializing the file variable to the same file?

a) File file = new File("folder\file.txt");

Does not work because '\' is an escape character. Please refer to ZyBooks section 2.10.

b) File file = new File("folder\\file.txt");

Works as a relative path.

c) File file = new File("C:\\Users\\usrName\\workspace\\ProjectName\\folder\\file.txt");

Works as an absolute path.

d) File file = new File("C:\\Users\\usrName\\workspace\\ProjectName\\.\\folder\\file.txt");

Works as an absolute path. "." represents current directory so it's okay to stick them in the path.

e) File file = new File("package\\..\\file.txt");

Does not work. ".." represents parent directory.

The absolute path is then equated to "C:\\Users\\usrName\\workspace\\ProjectName\\file.txt", which does not go into the folder (called "folder") inside of the folder called "ProjectName".

43. True or False: The "static" keyword means that the variable is constant and cannot be changed.

a) True

b) False

The *static* keyword is used for fields and methods that belong to the class, rather than to an instance of the class. The *final* keyword is used for fields and methods that are constant and cannot be changed/overridden.

44. Placeholder