Java Workshop Study Quiz

=== Week 1 ===
~Running Java Programs~
1. 'java' runs the program by passing bytecode to JVM, 'javac' compiles the program to bytecode.
2. At the top of the 'main' method.
3. Pikachu.java
4. "RaichuZap" is printed.

~Types of Errors~
1. Run-time errors are usually logic errors and occur only when the program is running. Compile-time errors are usually syntax errors and occur before the program is executed.
2. A compile-time error.
3. Add a semi-colon at end of print statement.

=== Week 2 ===
~Binary and Hexadecimal Numbers~
1. 1011
2. 1 or 0 (on or off)
3. 1B
4. 8
5. Hardware components in a circuit can conveniently hold one of two states, on or off. Hence, binary numbers are used because they neatly represent a sequence of on or off states.

~Variables~
1. To store and refer to information.
2. Byte, short, int, long, float, double, char, boolean
3. int pikachu = 25;
4. pikachu = 100;
5. String charmander = "Rawr!";

~The Scanner class~
1. Scanner reading = new Scanner(System.in);
2. String thisWord = reading.next();
3. double thisValue = reading.nextDouble();

~Command-line Arguments~
1. ==super meowzers==

~Arithmetic Operations~
1. 22 20
2. 11
3. 3
4. true (equivalent to 10-- > 2)
5. int magikarp = Math.pow(7,5);

=== Week 3 ===
~Boolean Expressions~
1. || and !
2. !=, >, >=, <
3. result = 0 < x && x < 10;
4. false
5. = is for assignment, == is for comparison
6. .equals() compares contents and == compares memory addresses
7. Based on order of characters in Unicode encoding

~Casting~

1. A way to explicitly treat a variable or result of an expression as another type.
2. 2.5
3. 2.0
4. Round towards zero.

~if-else Statements~
1. if (boolean expression) {
     if_body
   } else {
     else_body
   }
2. Hang in there!
3. Yes
4. We can check another condition if-and-only-if the previous one is true
5. We can check another condition if-and-only-if the previous one is false

~while Loops~
1. while (boolean_expression) {
      while_body
   }
2. (i) Evaluate condition.
   (ii) If true, go to (iii). If false, go to (iv).
   (iii) Run while_body, then go to (i).
   (iv) Go to statement after while-loop.
3. 0
   0
   0
   ... infinitely

=== Week 4 ===
~for Loops~
1. for(initialization; condition; update){
      for_body;
   }
3. for(;;){
      System.out.println("Meow");
   }
4. There is a semicolon after the for-loop which means the for-loop body is empty.

~do-while Loops~
1. do {
      do_while_body;
   } while(boolean_expression);
2. i) Run body code.
   ii) Evaluate condition.
   iii) If true, run body code again. If false, skip and go to next statement.
3. do-while loops runs body code at least once, where as while loops can skip the body completely.

~Terminating Loops~
1. Re-evaluates the condition of the innermost loop and decides whether to enter the loop_body or exit.
2. C
3.
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

```
    String input;

    while (true) {
        System.out.print("Enter a word: ");
        input = sc.next();

        if (!input.equals("woof")) {
            System.out.println("You entered ... " + input);
        } else {
            System.out.println("Okay!");
            break;
        }
    }
}
```

~Arrays~
1. type[] name = new type[size];
2. double fancy = new double[2];
   fancy[0] = 0.21;
   fancy[1] = 1.31;
3. Yes, pokemonList hasn't been initialised yet. Program will not compile.

=== Week 5 ===
~Method Basics~
1. return_type method_name(parameter_type parameter_name, ...){
      method_body;
   }
2. D
3. "Oh no! The Pokemon broke free!"
4. Nothing is returned by the method
5. To minimise code repetition, increase readability of code and make code easier to maintain/modify.

~Passing Parameters~
1.
```
public static boolean isOdd(int n){
      if(n%2 != 0)
         return true;
      return false;
   }
```
2.
```
public class Rawr {
      public static void main(String[] args){
         Scanner sc =  new Scanner(System.in);
         while(true){
            System.out.print("Enter an integer: ");
            int input = sc.nextInt();
            if(input < 0)
               break;
            System.out.println("Square is ... " + square(input));
         }
         System.out.println("Okay!");
      }

      public static int square(int n){
         return n*n;
      }
   }
```
3.

~Variable Scope~

1. A, B, C and D
2. A, B
3. A
4. C

=== Week 6 ===

~Strings~
1. String blank = "";
2. "A wild ZUBAT appears!"
3. "ub"
4. System.out.println("true\\false");
5. == compares memory addresses of variables, .equals() compares contents of Strings

~The StringBuilder class~
1. A String is immutable (constant) upon creation, a StringBuilder is like a String but not immutable (modifiable).
2. When concatenating a large number of Strings.
3. System.out.println(woof.toString());

~Exceptions and Exception Handling~
1. An event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
2. A checked exception checks during compile-time whether it is handled or not. An unchecked exception doees not need to be handled.
3.
```
public static void main(String[] args) {
  String a;
  a.trim();
}
```
4. (a)
```
try {
    int pokedexNo = 0;
    if (pokedexNo > 0) {
        System.out.println("Wild Pokemon!");
    } else {
        throw new Exception("Missingno!");
    }
}
```
(b)
```
public static void main(String[] args) throws Exception {
  ...
}
```
5. ArrayIndexOutOfBoundsException, NullPointerException, ArithmeticException, IOException, InputMismatchException.
6. Catch clauses are executed in a top-down order, and if the thrown exception is a child (subclass) of a preceding catch clause's exception then the specific catch clause will never be able to catch the Exception.

~Reading/Writing Files~
1. FileNotFoundException
2.
```
public static void main(String[] args) {
    try {
        File text = new File("Jolteon.txt");
        Scanner sc = new Scanner(text);
        while (sc.hasNextLine()) {
            System.out.println(sc.nextLine());
        }
```

```
    } catch (FileNotFoundException e ) {
        System.out.println("File wasn't found!");
    }
}
3. Close it
4.
public static void main(String[] args) {
    try {
        File file = new File("awesome.txt");
        Scanner sc = new Scanner(System.in);
        PrintWriter pw = new PrintWriter(file);

        System.out.println("Enter a sentence: ");
        pw.append(sc.nextLine());
        pw.close();
    } catch (FileNotFoundException e ) {
        System.out.println("File wasn't found!");
    }
}
```

=== Week 7 ===
~Classes and Objects~
1. You can store more than one kind of value in one place, you can control access to the values, you can control operations possible with the values.
2. An object belongs to a class and is an instance of a class.
3.
```
modifiers class name {
  methods_and_variables_go_here
}
```
4.
```
class Item {
  String name;
  String description;
  int quantity;
}
```

~Constructors~
1. To initialise variables and perform operations when an object is first instantiated.
2.
```
modifiers class_name (parameters) {
  constructor_body
}
```
3.
```
public Pikachu () {
  name = "Pikachu";
  HP = 10;
  ATK = 8;
  SPA = 14;
  DEF = 6;
  SPD = 7;
  SPE = 20;
}
```
4. Add the 'this' keyword to the lefthand-side 'name' variable.

~Static vs. Non-static~
1. A member variable, or method, can be accessed without requiring an instantiation of the class to which it belongs.
2. A non-static String variable. Since every instance of a Person could have a different name.

3. A static String variable. Since every Pikachu can evolve into a Raichu, a shared trait.
4. A class variable is a global static variable. An instance variable is a global non-static variable.
5. Static methods can only use class variables (global static) or local variables. Non-static methods can use any kind of variable: class, instance and local variables.

~Enumerations~
1. Special classes that are used to represent a fixed number of constants.
2. Allows you to control what values are legal or illegal.
3.

```
enum Direction {
    NORTH, SOUTH, EAST, WEST;
}

class Lost {
    public void travel(Direction direction) {
        switch(direction) {
            case NORTH:
                System.out.println("Keep going!");
                break;
            case SOUTH:
                System.out.println("That's backwards!");
                break;
            case EAST:
            case WEST:
                System.out.println("Wrong way!");
                break;
        }
    }
}
```

~Encapsulation~
1. The technique of making global variables in a class private and providing access to the variables via public methods.
2. public allows access from any other file or class. private only allows access from the class in which it was declared (same class).
3. Both the variable and its accessor method are public which contradicts the principle of encapsulation. The name variable should be private to fix the issue.

=== Week 8 ===
~More Constructors and Methods~
1. Allows for more appropriate initialisations based on given parameters.
2. There is a duplication of method signatures (method name + parameters).
sum(2, 4) could return either an int or String, ambiguity which is illegal in Java.

~Testing~
1.
(a) Giving some input to a program and investigating the output.
(b) A way of developing software where tests are written for a method before the method is written.
(c) A route through the flow of the program.
(d) Testing a program by only inspecting the input and output.
(e) Testing a program by inspecting execution paths.

(f) Running all tests after a significant change to ensure no previously functioning code is affected.
(g) A condition (boolean expression) written at some point in a program
(h) An assertion placed before some piece of code
(i) An assertion placed after some piece of code
(j) An assertion of the class
2.

```
public class MagikarpTester {
  public static void main(String[] args) {
    Magikarp m = new Magikarp(19);
    m.useRareCandy();
    if (m.canEvolve()) {
      System.out.println("Passed!");
    } else {
      System.out.println("Failed!");
    }
  }
}
```

=== Week 9 ===
~Designing Classes~
Re-do Week 10 tutorial questions.

=== Week 10 ===
~Inheritance~
1. Significant code re-use (much less code repetition).
2. High school is a subclass of school.
3. Meow is the subclass (child) and Woof is the superclass (parent).
4. It prevents access of a method or variable from any other class, unless it is a subclass or the same class as the protected method or variable.
5.

```
class Pokemon {
  public int level;
  public String name, type;
  public int[] stats;
}

class Bulbasaur extends Pokemon {
  public void useTackle() { ... }
}

class Charmander extends Pokemon {
  public void useScratch() { ... }
}
```

6. super(parameters);

~Polymorphism~
1. A feature wherein subclasses of a class can define their own unique behaviours and yet share some of the same functionality of the parent class.
2. Allows methods with the same signature (name and parameters) to have multiple code bodies depending on the class it belongs to.
3. To override the parent class's method when the child's version is executed.
4. No, it is a concept that is demonstrated through inheritance and method overriding.

5. Method overloading is creating methods with the same name but different parameters. Method overriding involves overriding a parent class's method with the same signature.
6.
```
public static void main(String[] args) {
  Animal[] animals = new Animal[3];
  animals[0] = new Animal();
  animals[1] = new Dog(); // <--- OK because Inheritance
  animals[2] = new Cat(); // <--- OK because Inheritance

  for (int i = 0; i < animals.length; i++) {
    animals[i].talk(); // <--- Will print Graah! Woof! Meow!
  }
}
```

~Common Array Algorithms~
1.
```
public static void printReverseEntry(String[] entry) {
    for (int i = entry.length-1; i >= 0; i--) {
        System.out.println(entry[i]);
    }
}
```
2.
```
public static int findMin(int[] array){
    int min = array[0];
    for (int i = 0; i < array.length; i++) {
        if (array[i] < min)
            min = array[i];
    }
    return min;
}
```
3.
```
@Override
public String toString(){
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < currentBox.length; i++) {
        sb.append(currentBox[i]);
        sb.append(" ");
    }
    return sb.toString().trim();
}
```

=== Week 11 ===
~Primitive Wrapper Classes~
1. Allows primitive data types to be accessed as objects and has many built-in methods for convenience.
2. Integer.MAX_VALUE, Integer.MIN_VALUE, Integer.parseInt(String s)

~The ArrayList~
1. ArrayList<reference_type> name = new ArrayList<reference_type>();
2. The array list will automatically expand when it reaches maximum capacity where as arrays do not.
3.
(a) .get(i)
(b) .remove(0)
(c) .size()
(d) .add(meow)
4.
```
list.add(kitty);
list.add(puppy);
```

```
array[0] = kitty;
array[1] = puppy;
System.out.println(list.get(0).getSecret());
System.out.println(array[1].getSecret());
```

~Sets and Maps~
1. Sets don't store duplicates, but Lists can.
2. TreeSets keep objects in order if they are comparable. HashSets have very fast access but don't keep things in order.
3. A map is able to store an association between a key and its value.
4. Every key has a unique value.

~For-each Loop~
1.
```
for (list_stored_type name : list_name) {
  for_each_body
}
```
2. Allows for easy iterating over lists.
3. List must implement the Iterable interface.
4.
```
for(Pokemon current : party) {
  current.heal();
}
```

~Recursion~
1. Often times easier and more elegant to implement.
2. Usually slower than an iterative approach.
3. A base case and a recurrence relation.
4.
```
public static int factorial (int n) {
    if (n == 0)
        return 1;
    return n * factorial(n-1);
}
```
5.
```
public static int factorial (int n) {
    int res = 1;
    for (int i = 2; i <= n; i++) {
        res *= i;
    }
    return res;
}
```
6.
```
public static int fibonacci(int n) {
    if (n == 0)
        return 0;
    else if (n <= 2)
        return 1;
    return fibonacci(n-1) + fibonacci(n-2);
}
```

=== Week 12 ===
~Abstract Classes~
1. To group related classes together and improve program organization.
2. Fruit with subclasses Apple, Orange, Pear
3.
```
abstract class Character {
  protected int x, y;
```

```
  abstract void move(int newX, int newY);

  public int getX() {
    return this.x;
  }

  public int getY() {
    return this.y;
  }
}

class PlayerCharacter extends Character {
    public void move(int newX, int newY){
        this.x = newX;
        this.y = newY;
    }
}

class NonPlayerCharacter extends Character {
    private int pushback = 5;

    public void move(int newX, int newY){
         this.x = newX - pushback;
         this.y = newY - pushback;
    }
}
```

~Interfaces~
1. To abstract away the details of an object and show only the necessary functions to be used.
2. The class must implement all the methods specified in the interface.
3. Interfaces can be used as a type to perform grouping. E.g. Talkable[] array;
4.

```
interface Talkable {
  void talk();
}
```

5.
(a) Abstract classes CANNOT be instantiated but interfaces CAN.
(b) Abstract classes CAN have methods but interfaces can only have METHOD SIGNATURES.
(c) Abstract classes CAN have variables but interfaces can only have 'static final' variables.
(d) Abstract classes CAN have a constructor but interfaces CANNOT.

~2D Arrays~
1. To represent a grid of items.
2. Nope, it's unlimited.
3.

```
public static int[] summate(int[][] data){
    int[] sums = new int[data.length];

    for(int i = 0; i < data.length; i++){
           int total = 0;
           for(int j = 0; j < data[i].length; j++){
                  total += data[i][j];
           }
```

```
            sums[i] = total;
    }
    return sums;
}
```

```
~switch-statements~
1.
switch(variable) {
  case value_1:
    code_to_do
    [break;]
  case value_2:
    code_to_do
    [break;]
  [default:
    default_to_do]
}
2. It's not very effective.
3. It hit!
```