1 Pass-by-What?

```java
public class Pokemon {
    public String name;
    public int level;

    public Pokemon(String name, int level) {
        this.name = name;
        this.level = level;
    }

    public static void main(String[] args) {
        Pokemon p = new Pokemon("Pikachu", 17);
        int level = 100;
        change(p, level);
        System.out.println("Name: " + p.name + ", Level: " + p.level);
    }

    public static void change(Pokemon poke, int level) {
        poke.level = level;
        level = 50;
        poke = new Pokemon("Gengar", 1);
    }
}
```

(a) Draw the box-and-pointer diagram after Java evaluates the `main` method. What would Java print?

```
main:13
  p
  level 100
Pokemon Instance
  name "Pikachu"
  level 100
String
Name: Pikachu, Level: 100
```

For a step by step walkthrough of this box and pointer diagram, see https://tinyurl.com/yyknozdd

(b) On line 19, we set `level` equal to 50. What level do we mean? An instance variable of the `Pokemon` class? The local variable containing the parameter to the `change` method? The local variable in the `main` method? Something else?

It is the local variable in the `change` method and does not have any effect on the other variables of the same name in the `Pokemon` class or the `main` method.
2 Static Methods and Variables

```java
public class Cat {
    public String name;
    public static String noise;

    public Cat(String name, String noise) {
        this.name = name;
        this.noise = noise;
    }

    public void play() {
        System.out.println(noise + " I'm " + name + " the cat!");
    }

    public void nickname(String newName) {
        name = newName;
    }

    public static void anger() {
        noise = noise.toUpperCase();
    }

    public static void calm() {
        noise = noise.toLowerCase();
    }
}
```

(a) Write what will happen after each call of `play()` in the following method.

```java
public static void main(String[] args) {
    Cat a = new Cat("Cream", "Meow!");
    Cat b = new Cat("Tubbs", "Nyan!");
    a.play();
    b.play();
    Cat.anger();
    a.calm();
    a.play();
    b.play();
    a.nickname("Kitty");
    a.play();
    b.play()
}
```

Nyan! I'm Cream the cat!
Nyan! I'm Tubbs the cat!
nyan! I'm Cream the cat!
nyan! I'm Tubbs the cat!
nyan! I'm Kitty the cat!
nyan! I’m Tubbs the cat!

Explanation: Notice that the variable `noise` was declared to be a static variable. What this means is that there is only one `noise` variable for the entire `Cat` class. In contrast, every time a `Cat` object is created, it gets its own `name`.

Another common use of static variables is for storing the total number of objects that have been created of a class. There needs to be only one variable per class for storing something like this!

Since there is only one `noise` variable, it first gets set to `Meow!` in line 2. Then it changes to `Nyan!` in line 3 and `Meow!` is forgotten forever.

Line 6 changes our `noise` from `Nyan!` to `NYAN!`. Then, Line 7 eventually changes our one and only `noise` to `nyan!`.

Line 10 looks at an instance method of the `Cat` class. When we call `nickname` on `a`, it changes `a`’s name to `Kitty`, but `b`’s name should stay the same.

(b) If we were to add `Cat.nickname("KitKat")` to the end of our main function, what would happen?

If we were to add this line to our main function, it would error. In the class, `nickname` is an instance function. What would it mean to rename `Cat` as opposed to a specific cat? It doesn’t really make sense. So when we try to run this function on our class, it errors.

One more thing to note is the functions `anger` and `calm` are declared static themselves. Static methods can be called using the name of the class, as in line 7, whereas non-static methods cannot. The golden rule for static methods to know is that **static methods can only modify static variables.**
3 Practice with Linked Lists

Draw the box-and-pointer diagram that results from running the following code. A StringList is similar to an IntList. It has two instance variables, first and rest.

```java
StringList L = new StringList("eat", null);
L = new StringList("shouldn't", L);
L = new StringList("you", L);
L = new StringList("sometimes", L);
StringList M = L.rest;
StringList R = new StringList("many", null);
R = new StringList("potatoes", R);
R.rest.rest = R;
M.rest.rest.rest = R.rest;
L.rest.rest.rest = L.rest.rest.rest;
L = M.rest;
```

For a step by step walkthrough of this box and pointer diagram, see https://tinyurl.com/y38jkzpj
4 Squaring a List Extra

Implement `square` and `squareDestructive` which are static methods that both take in an `IntList` `L` and return an `IntList` with its integer values all squared. `square` does this non-destructively with recursion by creating new `IntLists` while `squareDestructive` uses an iterative approach to change the instance variables of the input `IntList` `L`.

```java
public static IntList square(IntList L) {
  if (L == null) {
    return L;
  } else {
    IntList rest = square(L.rest);
    IntList M = new IntList(L.first * L.first, rest);
    return M;
  }
}
```

Explanation: This is a recursive function relying on the famous recursive leap of faith. Lines 1-2 take care of the base case. Line 4 takes the recursive leap of faith. It assumes that the `square` function correctly squares the rest of the linked list. Line 5 then uses the correct result from line 4 to create a new `IntList` with the first element squared.

```java
public static IntList squareDestructive(IntList L) {
  IntList B = L;
  while (B != null) {
    B.first *= B.first;
    B = B.rest
  }
  return L;
}
```

Explanation: This method walks through the linked list, one part at a time, and squares each element in place. `B` is used as a position variable to keep track of where we are in the linked list. Once `B` becomes `null`, we have hit the end of the linked list.

Extra: Now, implement `square` iteratively, and `squareDestructive` recursively.

```java
public static IntList square(IntList L) {
  if (L == null) {
    return L;
  }
  IntList B = L.rest;
  IntList LSquared = new IntList(L.first * L.first, null);
  IntList C = LSquared;
  while (B != null) {
    C.rest = new IntList(B.first * B.first, null);
    B = B.rest;
    C = C.rest;
  }
  return LSquared;
}
```
public static IntList squareDestructive(IntList L) {
    if (L == null) {
        return L;
    } else {
        L.first = L.first * L.first;
        squareDestructive(L.rest);
    }
    return L;
}