

1 Switcheroo

The **Golden Rule of Equals** says:

Given variables **b** and **a**, the assignment statement **b = a** copies all the bits from **a** into **b**.

Passing parameters obeys the same rule: copy the bits to the new scope.

1.1 What is wrong with this definition of `swap`? How can we fix it?

```
class SimpleSwap {  
    public static void swap(int a, int b) {  
        int temp = b;  
        b = a;  
        a = temp;  
    }  
    public static void main(String[] args) {  
        int x = 2, y = 5;  
        System.out.println("x: " + x + ", y: " + y);  
        swap(x, y);  
        System.out.println("x: " + x + ", y: " + y);  
    }  
}
```

x: 2, y: 5

x: 2, y: 5

In the main method, **x** and **y** won't actually be swapped. Within `swap`, we can change what **a** and **b** point to, but we can't change the variables that were declared in `main`. We can fix this by either in-lining the `swap` functionality in the main method or returning and reassigning the swapped values using an object.

Meta: Emphasize “Golden Rule of Equals” (pass-by-value). The bits are copied over. It is helpful to talk about the 8 types of primitive variable types (**byte**, **short**, **int**, **long**, **float**, **double**, **boolean**, **char**), and that Java stores the actual value in the variable. So when a primitive variable is passed to a function, its value itself is copied over.

1.2 How is this implementation of swap different?

```
class Coordinate {
    int x, y;
    Coordinate(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

class SwapObject {
    public static void swap(Coordinate p) {
        int temp = p.x;
        p.x = p.y;
        p.y = temp;
    }
    public static void main(String[] args) {
        Coordinate p = new Coordinate(2, 5);
        System.out.println("p.x: " + p.x + ", p.y: " + p.y);
        swap(p);
        System.out.println("p.x: " + p.x + ", p.y: " + p.y);
    }
}
```

When calling `swap` with a `Coordinate` object, we're passing a reference to the original `Coordinate` object. The object's instance variables can be changed from within `swap` and will remain changed after we exit from the function.

Meta: It can be useful to mention that Java stores only the address of an object in a variable of non-primitive type, and so when passed to a function, the address of the original object gets passed around.

2 Flatter Me

Arrays are ordered sequences of fixed length. Unlike Python lists, the length must be known when creating an array.

```
int[] a = new int[3];
```

It is possible to initialize and fill an array in a single expression.

```
int[] b = new int[]{1, 2, 3};
```

Java can infer the type of the array from its context, yielding this shorthand.

```
int[] c = {1, 2, 3};
```

Uninitialized values have a default value like `0`, **false**, or **null**.

```
String[] c = new String[1];  
c[0] == null;
```

- 2.1 Implement `middle`, which takes in `int[]` and returns the middle element. If no element is in the exact middle, return the element to the left middle.

```
public static int middle(int[] data) {  
    return data[(data.length - 1) / 2];  
}
```

Meta: It is helpful here to review the fact that `length` is a field and not a method.

- 2.2 Write a method `flatten` that takes in a two-dimensional array `data` and returns a one-dimensional array that contains all of the arrays in `data` concatenated together.

```
public static int[] flatten(int[][] data) {  
    int size = 0;  
    for (int[] row : data) {  
        size += row.length;  
    }  
    int[] result = new int[size];  
    int i = 0;  
    for (int[] row : data) {  
        for (int value : row) {  
            result[i] = value;  
            i += 1;  
        }  
    }  
    return result;  
}
```

Meta: This may be one of the first times students see this kind of a for loop syntax, so it will be helpful to review it. Additionally, if students are stuck, one starting point is to figure out what the size of the new flattened array should be. Also, a discussion on why the array size needs to be determined at the beginning is in order since in Python, one can use the `.append` method to dynamically change the size of the array.

3 Dogs Yay

```

3.1 class Dog {
    public void walk() {
        System.out.println("The dog is walking");
    }
}
class Beagle extends Dog {
    @Override
    public void walk() {
        System.out.println("The beagle is walking");
    }
}

```

What would Java display?

(a) `Dog fido1 = new Dog();`
`fido1.walk();`

The dog is walking

(b) `Beagle fido2 = new Beagle();`
`fido2.walk();`

The beagle is walking

(c) `Beagle fido3 = new Dog();`
`fido3.walk();`

Compile-time error.

A container meant for Beagles can't contain Dogs.

(d) `Dog fido4 = new Beagle();`
`fido4.walk();`

The beagle is walking

A container for Dogs can contain Beagles. At compile time, `fido.walk()` is linked to `Dog.walk()` but at runtime, this method is overridden by `Beagle.walk()`.

Meta: It can be useful to talk about the static types of variables as boxes with labels and the dynamic type of an object as what you place in the box. So long as what you place in the box fits the label (i.e. a **Beagle** is a **Dog**, so a dynamic **Beagle** can be placed in a static **Dog**) then no compiler errors

occur. Remember to mention dynamic method lookup.

- 3.2 What would each call in `Poodle.main` print? If a line would cause an error, determine if it is a compile-error or runtime-error.

```

class Dog {
    void bark(Dog dog) {
        System.out.println("bark");
    }
}

class Poodle extends Dog {
    void bark(Dog dog) {
        System.out.println("woof");
    }
    void bark(Poodle poodle) {
        System.out.println("yap");
    }
    void play(Dog dog) {
        System.out.println("no");
    }
    void play(Poodle poodle) {
        System.out.println("bowwow");
    }
    public static void main(String[] args) {
        Dog dog = new Poodle();
        Poodle poodle = new Poodle();

        dog.play(dog)           // Compile-error
        dog.play(poodle)        // Compile-error
        poodle.play(dog)         // no
        poodle.play(poodle)      // bowwow

        poodle.bark(dog)         // woof
        poodle.bark(poodle)      // yap
        dog.bark(dog)            // woof
        dog.bark(poodle)         // woof

    }
}

```

Meta: This requires a walkthrough. First, setup the rules: dynamic method lookup, method overriding, static and dynamic types, and compile-error vs. runtime error. Give students a chance to work, then clear misconceptions.

4 Pokemon *Extra Practice*

4.1 Identify the errors that occur when running the code to the right.

```
public class Pokemon {
    public int hp, power;
    public String cry;
    public String secret;
    public Pokemon() {
        hp = 50;
        cry = "Poke?";
    }
    public Pokemon(String c, int hp) {
        cry = c;
        this.hp = hp;
    }
    public void attack(Pokemon p) {
        p.hp -= power;
    }
    public void eat() {
        System.out.println("nom nom");
    }
}

public class Pikachu extends Pokemon {
    public Pikachu() {
        hp = 100;
    }
    public Pikachu(int hp) {
        super("Pika pika pikachu", hp);
    }
    public void attack(Pokemon p) {
        p.hp = 0;
    }
    public void eat() {
        System.out.println("nom Pika nom");
    }
}

public class Squirtle extends Pokemon {
    public void attack() {
        System.out.println("Water gun!!!");
    }
}
```

```
Pikachu p = new Pikachu();
Pokemon a = p;
// (1)
// p = a;
a.eat();
a = new Squirtle();
// (2)
// a.attack();
((Squirtle) a).attack();
Pokemon z = new Pikachu();
// (3)
// Squirtle s = (Squirtle) z;
((Pokemon) p).attack(z);
```


(1) is a compile-time error since you can't assign a static type `Pokemon` to a variable who has a static type `Pikachu`. Can be solved with a cast.

(2) is a compile-time error because `Pokemon` does not have a method with the signature `attack()`.

(3) is a run-time error because the dynamic type of `z` (`Pikachu`) cannot be cast to a `Squirtle`.