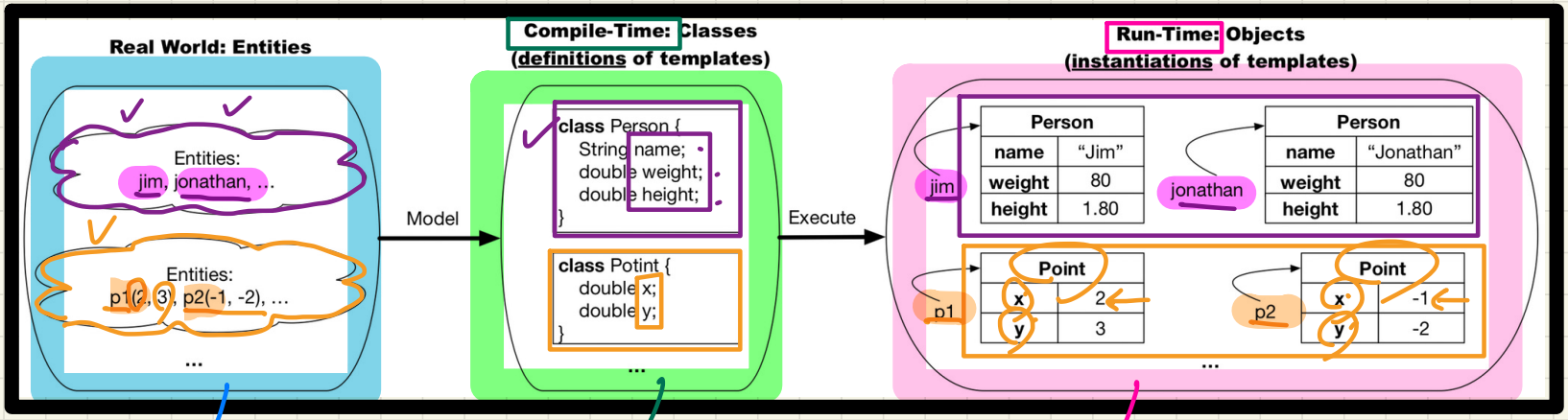
Lecture 4

Part A

Classes and Objects - Object Orientation

Observe-Model-Execute Process



observe

model

execute

Modelling: from Entities to Classes

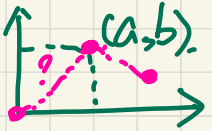
Identify Critical Nouns & Verbs

classes attributes

accessors mutators

Example 1

class Point



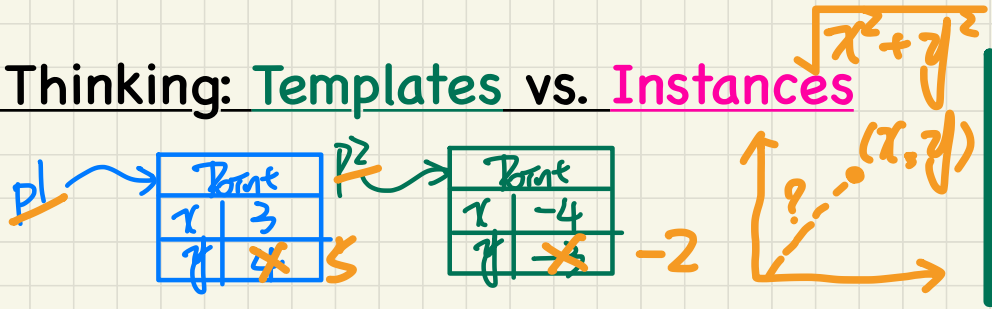
Points on a two-dimensional plane are identified by their signed distances from the X- and Y-axises. A point may move arbitrarily towards any direction on the plane. Given two points, we are often interested in knowing the distance between them.

attribute (x, y)

Example 2

A person is a being, such as a human, that has certain attributes and behaviour constituting personhood: a person ages and grows on their heights and weights.

OO Thinking: Templates vs. Instances




```
public class Point {
  private double x;
  private double y;
}
```

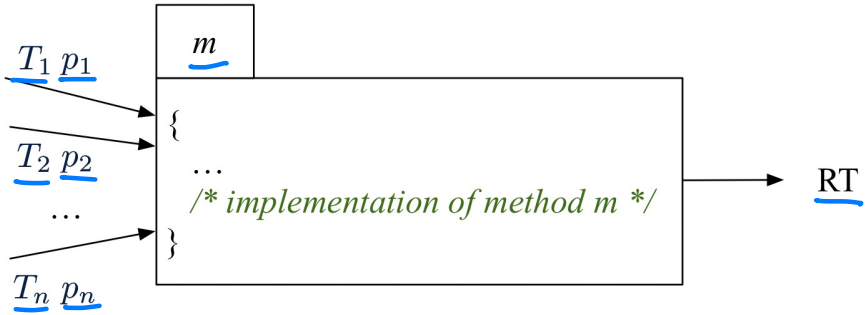
- A *template* (e.g., class `Point`) defines what's **shared** by a set of related entities (i.e., 2-D points).
  - Common *attributes* (`x`, `y`)
  - Common *behaviour* (move left, move up)
- Each template may be *instantiated* as multiple instances, each with *instance-specific* values for attributes `x` and `y`:
  - `Point` instance `p1` is located at $(3, 4)$
  - `Point` instance `p2` is located at $(-4, -3)$
- Instances of the same template may exhibit *distinct behaviour*.
  - When `p1` moves up for 1 unit, it will end up being at $(3, 5)$
  - When `p2` moves up for 1 unit, it will end up being at $(-4, -2)$
  - Then, `p1`'s distance from origin: $[\sqrt{3^2 + 5^2}]$
  - Then, `p2`'s distance from origin: $[\sqrt{(-4)^2 + (-2)^2}]$

What Is a Method?

Header (def.). parameters.

RT m(T1 p1, T2 p2, ..., Tn pn) { ... }

- A *method* is a named block of code, *reusable* via its name.



- The *Header* of a method consists of:
  - Return type [*RT* (which can be `void`)]
  - Name of method [*m*]
  - Zero or more *parameter names* [$p_1, p_2, \ldots, p_n$]
  - The corresponding *parameter types* [$T_1, T_2, \ldots, T_n$]
- A call to method *m* has the form: $m(a_1, a_2, \ldots, a_n)$
  Types of *argument values* $a_1, a_2, \ldots, a_n$ must match the the corresponding parameter types $T_1, T_2, \ldots, T_n$.

Usage

m(a1, a2, ... an)

arguments

Parameters vs. Arguments

parameters.

Template Definition

```
class Point {
   Point(double x, double y) {...}

   double getDistanceFrom(Point other) {...}

   void move(char direction, double units) {...}
}
```

argument

```
class PointTester {
   static void main(String[] args) {
      Point p1 = new Point(2.5, -3.6);
      Point p2 = new Point(-4.8, 5.9);
      double dist1 = p1.getDistanceFrom(p2);
      double dist2 = p2.getDistanceFrom(p1);
      p1.move('R', 7.6);
   }
}
```

argument

Method Usages

① Method declared in the context object's type? ✓

② Arguments compatible with param. types?

p1.getDistanceFrom(p2)

p1: context object

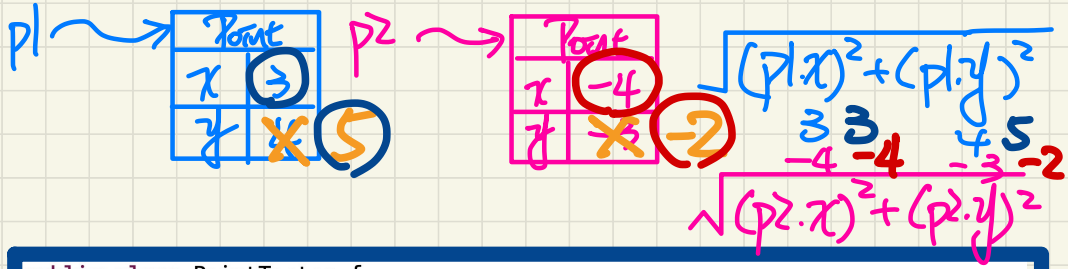
p2: argument

Kinds of Methods

1. *Constructor*
   - Same name as the class. No return type. *Initializes* attributes.
   - Called with the **new** keyword.
   - e.g., `Person jim = new Person(50, "British");`
2. *Mutator*
   - *Changes* (re-assigns) attributes
   - `void` return type
   - Cannot be used when a value is expected
   - e.g., `double h = jim.setHeight(78.5)` is illegal!
3. *Accessor*
   - *Uses* attributes for computations (without changing their values)
   - Any return type other than `void`
   - An explicit *`return` statement* (typically at the end of the method) returns the computation result to where the method is being used.
     e.g., `double bmi = jim.getBMI();`
     e.g., `println(p1.getDistanceFromOrigin());`

OOP: Creating and Manipulating Objects

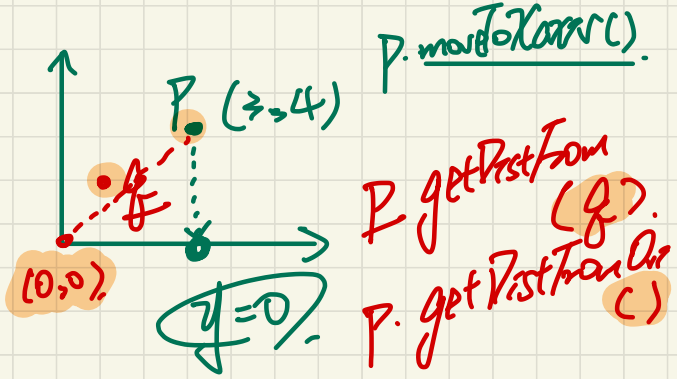
```java
public class Point {
    private double x;
    private double y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public void moveUp(double units) {
        this.y += units;
    }

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

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

    public double getDistanceFromOrigin() {
        double dist =
            Math.sqrt(this.x * this.x
                + this.y * this.y);
        return dist;
    }
}
```




```java
public class PointTester {
    public static void main(String[] args) {
        Point p1 = new Point(3, 4);
        Point p2 = new Point(-4, -3);

        System.out.println("p1 " + "(" + p1.getX() + ", " + p1.getY() + ")");
        System.out.println("p2 " + "(" + p2.getX() + ", " + p2.getY() + ")");
        System.out.println(p1.getDistanceFromOrigin());
        System.out.println(p2.getDistanceFromOrigin());

        p1.moveUp(1);
        p2.moveUp(1);

        System.out.println("p1 " + "(" + p1.getX() + ", " + p1.getY() + ")");
        System.out.println("p2 " + "(" + p2.getX() + ", " + p2.getY() + ")");
        System.out.println(p1.getDistanceFromOrigin());
        System.out.println(p2.getDistanceFromOrigin());
    }
}
```

Use of Accessors vs. Mutators

```
class Person {
  void setWeight(double weight) { ... }
  double getBMI() { ... }
}
```



- Calls to *mutator methods* *cannot* be used as values.
  - e.g., `System.out.println(jim.setWeight(78.5));` ×
  - e.g., `double w = jim.setWeight(78.5);` ×
  - e.g., `jim.setWeight(78.5);` ✓
- Calls to *accessor methods* *should* be used as values.
  - e.g., `jim.getBMI();` ×
  - e.g., `System.out.println(jim.getBMI());` ✓
  - e.g., `double w = jim.getBMI();` ✓

Method Parameters

- **Principle 1:** A *constructor* needs an *input parameter* for every attribute that you wish to initialize.

  e.g., `Person(double w, double h)` vs. `Person(String fName, String lName)`

- **Principle 2:** A *mutator* method needs an *input parameter* for every attribute that you wish to modify.

  e.g., In `Point`, `void moveToXAxis()` vs. `void moveUpBy(double unit)`

- **Principle 3:** An *accessor method* needs *input parameters* if the attributes alone are not sufficient for the intended computation to complete.

  e.g., In `Point`, `double getDistFromOrigin()` vs. `double getDistFrom(Point other)`

Lecture 4

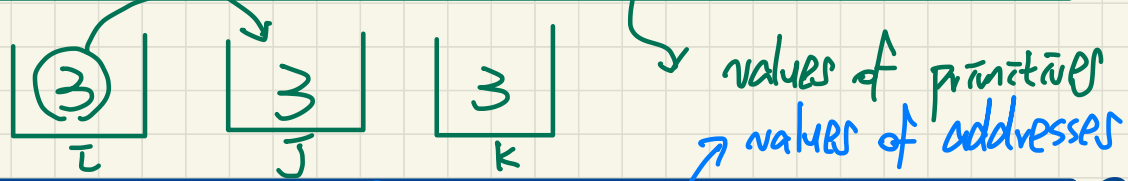
Part B

Classes and Objects - Reference Aliasing

Copying Primitive vs. Reference Values

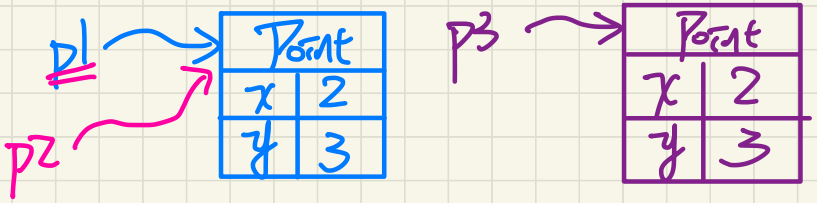
Primitive

```
int i = 3;
int j = i;   System.out.println(i == j);/*true*/
int k = 3;   System.out.println(k == i && k == j);/*true*/
```



Reference

```
Point p1 = new Point(2, 3);
Point p2 = p1;   System.out.println(p1 == p2);/*true*/
Point p3 = new Point(2, 3);
Systme.out.println(p3 == p1 || p3 == p2);/*false*/
Systme.out.println(p3.x == p1.x && p3.y == p1.y);/*true*/
Systme.out.println(p3.x == p2.x && p3.y == p2.y);/*true*/
```

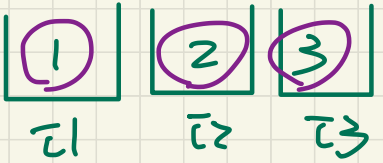


Copying Primitive Values

```
int i1 = 1;
int i2 = 2;
int i3 = 3;
int[] numbers1 = {i1, i2, i3};
int[] numbers2 = new int[numbers1.length];
for(int i = 0; i < numbers1.length; i ++) {
  numbers2[i] = numbers1[i];
}
numbers1[0] = 4;
System.out.println(numbers1[0]);
System.out.println(numbers2[0]);
```

4

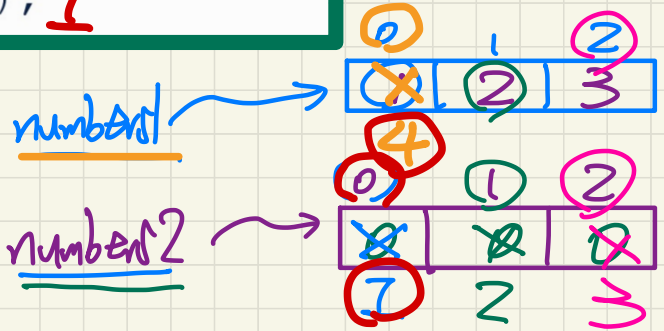
1



1st: numbers2[0] = numbers1[0];

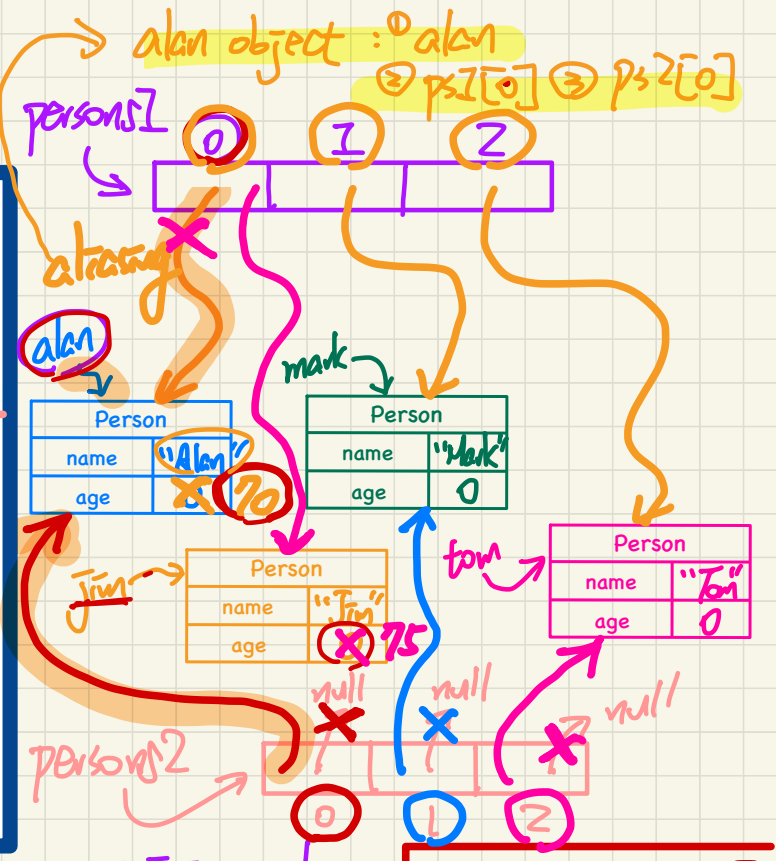
2nd: numbers2[1] = numbers1[1];

3rd: numbers2[2] = numbers1[2];



Copying Reference Values: Aliasing

```
Person alan = new Person("Alan");
Person mark = new Person("Mark");
Person tom = new Person("Tom");
Person jim = new Person("Jim");
Person[] persons1 = {alan, mark, tom};*
Person[] persons2 = new Person[persons1.length];
for(int i = 0; i < persons1.length; i ++) {
  persons2[i] = persons1[i]; }
persons1[0].setAge(70);
System.out.println(jim.getAge());
System.out.println(alan.getAge());
System.out.println(persons2[0].getAge());
persons1[0] = jim;
persons1[0].setAge(75);
System.out.println(jim.getAge());
System.out.println(alan.getAge());
System.out.println(persons2[0].getAge());
```



* persons1 is an array of size 3 where each index stores the address of some Person object

```
persons1[0] = alan;
persons1[1] = mark;
persons1[2] = tom;
```

1st: ps2[0] = ps1[0]
2nd: ps2[1] = ps1[1]
2nd: ps2[2] = ps1[2]