Classes and Objects

EECS1022 Sections M & N:
Programming for Mobile Computing
Winter 2021

CHEN-WEI WANG

Assumptions

It is assumed that you also complete:

- **Java Tutorial Videos**:
  - ***Weeks 6*** [link]
  - ***Weeks 7*** [link]
  - ***Weeks 8*** [link]
- **Written Notes**:
  - ***Inferring Classes from JUnit Tests*** [link]
  - ***Manipulating Multi-Valued, Reference-Typed Attributes*** [link]

Learning Outcomes

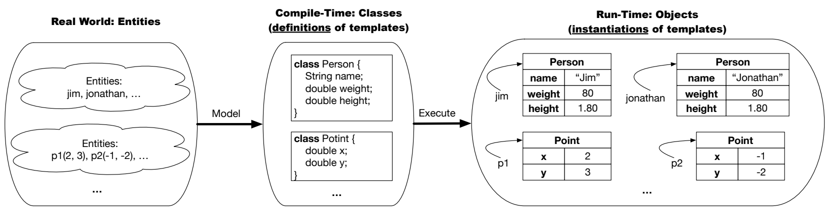
Understand:

- Object Orientation
- Classes as Templates:
  - attributes, constructors, (accessor and mutator) methods
  - use of `this`
- Objects as Instances:
  - use of `new`
  - the dot notation, method invocations
  - reference aliasing
- Reference-Typed Attributes: Single-Valued vs. Multi-Valued
- Non-Static vs. Static Variables
- Helper Methods

Where are we? Where will we go?

- We have developed the Java code within a `main` or utility method.
- In Java:
  - We may define more than one *classes*
  - Each class may contain more than one *methods*
- *object-oriented programming* in Java:
  - Use *classes* to define templates
  - Use *objects* to instantiate classes
  - At *runtime*, *create* objects and *call* methods on objects, to *simulate interactions* between real-life entities.

Object Orientation: Observe, Model, and Execute



- Study this tutorial video that walks you through the idea of *object orientation*.
- We *observe* how real-world *entities* behave.
- We *model* the common *attributes* and *behaviour* of a set of entities in a single *class*.
- We *execute* the program by creating *instances* of classes, which interact in a way analogous to that of real-world *entities*.

Object-Oriented Programming (OOP)

- In real life, lots of *entities* exist and interact with each other.
  e.g., *People* gain/lose weight, marry/divorce, or get older.
  e.g., *Cars* move from one point to another.
  e.g., *Clients* initiate transactions with banks.
- Entities:
  - Possess *attributes*;
  - Exhibit *bebaviour*; and
  - Interact with each other.
- Goals: Solve problems *programmatically* by
  - *Classifying* entities of interest
    Entities in the same class share *common* attributes and bebaviour.
  - *Manipulating* data that represent these entities
    Each entity is represented by *specific* values.

OO Thinking: Templates vs. Instances (1.1)

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.

- A template called `Point` defines the common
  - *attributes* (e.g., `x`, `y`) [≈ nouns]
  - *behaviour* (e.g., move up, get distance from) [≈ verbs]

OO Thinking: Templates vs. Instances (1.2)

- 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: $[\sqrt{(-4)^2 + (-2)^2}]$

OO Thinking: Templates vs. Instances (2.1)

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.

- A template called `Person` defines the common
  - *attributes* (e.g., `age`, `weight`, `height`) [≈ nouns]
  - *behaviour* (e.g., get older, gain weight) [≈ verbs]

OO Thinking: Templates vs. Instances (2.2)

- A *template* (e.g., class `Person`) defines what's **shared** by a set of related entities (i.e., persons).
  - Common *attributes* (`age`, `weight`, `height`)
  - Common *behaviour* (get older, lose weight, grow taller)
- Each template may be *instantiated* as multiple instances, each with *instance-specific* values for attributes `age`, `weight`, and `height`.
  - `Person` instance `jim` is 50-years old, 1.8-meters tall and 80-kg heavy
  - `Person` instance `jonathan` is 65-years old, 1.73-meters tall and 90-kg heavy
- Instances of the same template may exhibit *distinct behaviour*.
  - When `jim` gets older, he becomes 51
  - When `jonathan` gets older, he becomes 66.
  - `jim`'s BMI is based on his own height and weight [$\frac{80}{1.8^2}$]
  - `jonathan`'s BMI is based on his own height and weight [$\frac{90}{1.73^2}$]

OOP: Classes ≈ Templates

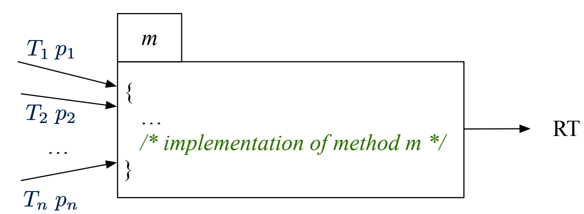
In Java, you use a *class* to define a *template* that enumerates *attributes* that are common to a set of *entities* of interest.

```
public class Person {
  private int age;
  private String nationality;
  private double weight;
  private double height;
}
```

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


OOP: Methods (1.1)

- 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 $T_1, T_2, \ldots, T_n$.

OOP: Methods (1.2)

- In the body of the method, you may
  - Declare new *local variables* (whose **scope** is within that method).
  - Use or change values of *attributes*.
  - Use values of *parameters*, if any.

```
public class Person {
  private String nationality;
  public void changeNationality(String newNationality) {
    nationality = newNationality; } }
```

- *Call* a *method*, with a **context object**, by passing *arguments*.

```
public class PersonTester {
  public static void main(String[] args) {
    Person jim = new Person(50, "British");
    Person jonathan = new Person(60, "Canadian");
    jim.changeNationality("Korean");
    jonathan.changeNationality("Korean"); } }
```


OOP: Methods (2)

- Each **class** `C` defines a list of methods.
  - A **method** `m` is a named block of code.
- We *reuse* the code of method `m` by calling it on an **object** `obj` of class `C`.

  For each **method call** `obj.m(...)`:
  - `obj` is the ***context object*** of type `C`
  - `m` is a method defined in class `C`
  - We intend to apply the *code effect of method* `m` to object `obj`.

    e.g., `jim.getOlder()` vs. `jonathan.getOlder()`

    e.g., `p1.moveUp(3)` vs. `p2.moveUp(3)`
- All objects of class `C` share *the same definition* of method `m`.
- However:

  ∵ Each object may have *distinct attribute values*.

  ∴ Applying *the same definition* of method `m` has *distinct effects*.

OOP: Methods (3)

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: Class Constructors (1.1)

- The purpose of defining a *class* is to be able to create *instances* out of it.
- To *instantiate* a class, we use one of its **constructors**.
- A constructor
  - declares input *parameters*
  - uses input parameters to *initialize* **some or all** of its *attributes*

OOP: Class Constructors (1.2)

For each *class*, you may define *one or more* **constructors**:

- *Names* of all constructors must match the class name.
- *No return types* need to be specified for constructors.
- Constructor must have *distinct* lists of *parameter types*.
  - `Person(String n)`, `Person(String n, int age)` ✓
  - `Person(String n, int age)`, `Person(int age, String n)` ✓
  - `Person(String fN, int age)`, `Person(String lN, int id)` ×
- Each *parameter* that is used to initialize an attribute must have a *matching type*.
- The *body* of each constructor specifies how **some or all** *attributes* may be *initialized*.

OOP: Class Constructors (2.1)

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

  public Point(double initX, double initY) {
    x = initX;
    y = initY;
  }

  public Point(char axis, double distance) {
    if (axis == 'x') { x = distance; }
    else if (axis == 'y') { y = distance; }
    else { /* Error: invalid axis */ }
  }
}
```


OOP: Class Constructors (2.2)

```
public class Person {
  private int age;
  private String nationality;
  private double weight;
  private double height;
  public Person(int initAge, String initNat) {
    age = initAge;
    nationality = initNat;
  }
  public Person (double initW, double initH) {
    weight = initW;
    height = initH;
  }
  public Person(int initAge, String initNat,
           double initW, double initH) {
    ... /* initialize all attributes using the parameters */
  }
}
```

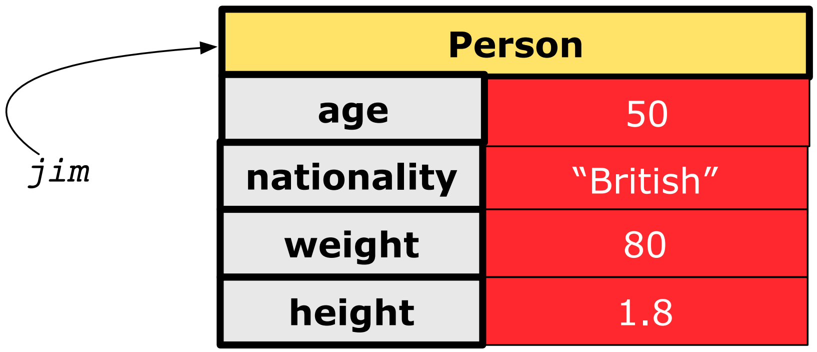

Visualizing Objects at Runtime (1)

- To trace a program with sophisticated manipulations of objects, it's critical for you to visualize how objects are:
  - Created using *constructors*
    `Person jim = new Person(50, "British", 80, 1.8);`
  - Inquired using *accessor methods*
    `double bmi = jim.getBMI();`
  - Modified using *mutator methods*
    `jim.gainWeightBy(10);`
- To visualize an object:
  - Draw a rectangle box to represent **contents** of that object:
    - Title indicates the *name of class* from which the object is instantiated.
    - Left column enumerates *names of attributes* of the instantiated class.
    - Right column fills in *values* of the corresponding attributes.
  - Draw arrow(s) for *variable(s)* that store the object's **address**.

Visualizing Objects at Runtime (2.1)

After calling a *constructor* to create an object:

```
Person jim = new Person(50, "British", 80, 1.8);
```

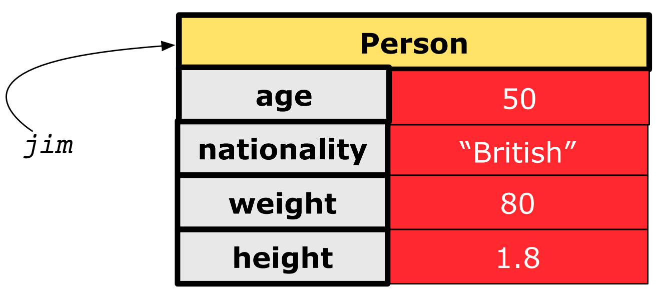


Visualizing Objects at Runtime (2.2)

After calling an *accessor* to inquire about context object `jim`:

```
double bmi = jim.getBMI();
```

- Contents of the object pointed to by `jim` remain intact.
- Retuned value $\frac{80}{(1.8)^2}$ of `jim.getBMI()` stored in variable `bmi`.

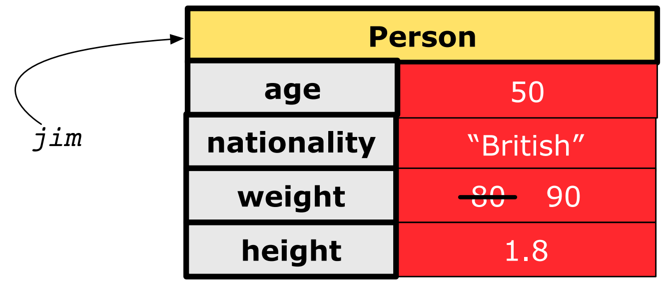


Visualizing Objects at Runtime (2.3)

After calling a *mutator* to modify the state of context object `jim`:

```
jim.gainWeightBy(10);
```

- *Contents* of the object pointed to by `jim` change.
- *Address* of the object remains unchanged.
  ⇒ `jim` points to the same object!

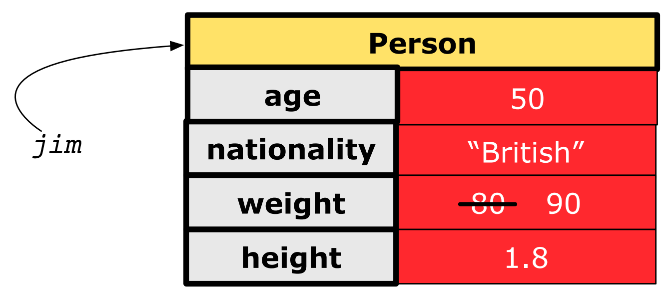


Visualizing Objects at Runtime (2.4)

After calling the same *accessor* to inquire the *modified* state of context object `jim`:

```
bmi = jim.getBMI();
```

- Contents of the object pointed to by `jim` remain intact.
- Retuned value $\frac{90}{(1.8)^2}$ of `jim.getBMI()` stored in variable `bmi`.



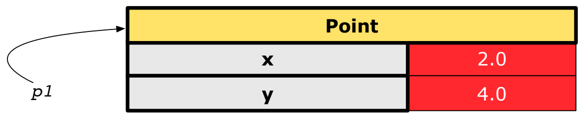
Object Creation (1.1)

```
Point p1 = new Point(2, 4);
```

1. **RHS (Source) of Assignment**: `new Point(2, 4)` creates a new *Point object* in memory.

Point	
x	2.0
y	4.0

2. **LHS (Target) of Assignment**: `Point p1` declares a *variable* that is meant to store the *address* of *some Point object*.
3. **Assignment**: Executing `=` stores new object's address in `p1`.



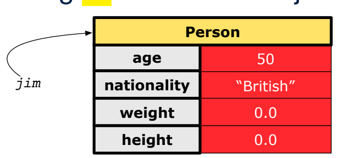
Object Creation (1.2)

```
Person jim = new Person(50, "British");
```

1. **RHS (Source) of Assignment**: `new Person(50, "British")` creates a new *Person object* in memory.

Person	
age	50
nationality	"British"
weight	0.0
height	0.0

2. **LHS (Target) of Assignment**: `Point jim` declares a *variable* that is meant to store the *address* of *some Person object*.
3. **Assignment**: Executing `=` stores new object's address in `jim`.



Object Creation (2)

```
Point p1 = new Point(2, 4);
System.out.println(p1);
```

```
Point@677327b6
```

By default, the address stored in `p1` gets printed.
Instead, print out attributes separately:

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

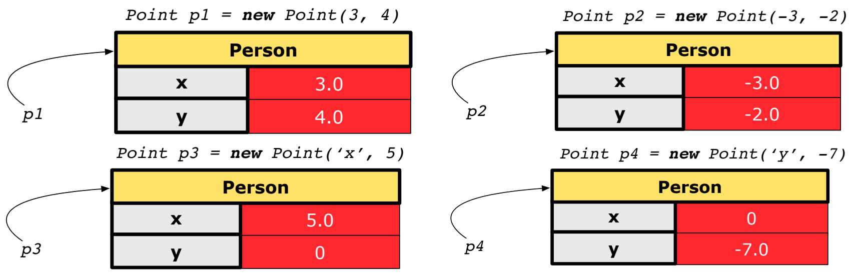
```
(2.0, 4.0)
```


OOP: Object Creation (3.1.1)

A constructor may only *initialize* some attributes and leave others *uninitialized*.

```
public class PointTester {
  public static void main(String[] args) {
    Point p1 = new Point(3, 4);
    Point p2 = new Point(-3 -2);
    Point p3 = new Point('x', 5);
    Point p4 = new Point('y', -7);
  }
}
```

OOP: Object Creation (3.1.2)

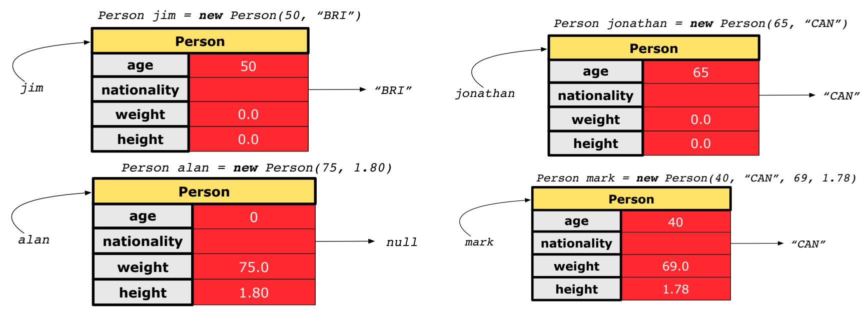


OOP: Object Creation (3.2.1)

A constructor may only *initialize* some attributes and leave others *uninitialized*.

```
public class PersonTester {
  public static void main(String[] args) {
    /* initialize age and nationality only */
    Person jim = new Person(50, "BRI");
    /* initialize age and nationality only */
    Person jonathan = new Person(65, "CAN");
    /* initialize weight and height only */
    Person alan = new Person(75, 1.80);
    /* initialize all attributes of a person */
    Person mark = new Person(40, "CAN", 69, 1.78);
  }
}
```


OOP: Object Creation (3.2.2)



OOP: Object Creation (4)

- When using the constructor, pass *valid* *argument values*:
  - The type of each argument value must match the corresponding parameter type.
  - e.g., `Person(50, "BRI")` matches `Person(int initAge, String initNationality)`
  - e.g., `Point(3, 4)` matches `Point(double initX, double initY)`
- When creating an instance, *uninitialized* attributes implicitly get assigned the *default values*.
  - Set *uninitialized* attributes properly later using **mutator** methods

```
Person jim = new Person(50, "British");
jim.setWeight(85);
jim.setHeight(1.81);
```

OOP: The Dot Notation (1)

- A binary operator:
  - **LHS** an object
  - **RHS** an attribute or a method
- Given a *variable* of some *reference type* that is **not** `null`:
  - We use a dot to retrieve any of its *attributes*. Analogous to 's in English e.g., `jim.nationality` means `jim`'s nationality
  - We use a dot to invoke any of its *mutator methods*, in order to *change* values of its attributes. e.g., `jim.changeNationality("CAN")` changes the `nationality` attribute of `jim`
  - We use a dot to invoke any of its *accessor methods*, in order to *use* the result of some computation on its attribute values. e.g., `jim.getBMI()` computes and returns the BMI calculated based on `jim`'s weight and height
  - Return value of an *accessor method* must be stored in a variable. e.g., `double jimBMI = jim.getBMI()`

The `this` Reference (1)

- Each *class* may be instantiated to multiple *objects* at runtime.

```
public class Point {
  private double x; private double y;
  public void moveUp(double units) { y += units; }
}
```

- Each time when we call a method of some class, using the dot notation, there is a specific *target*/*context* object.

```
1 Point p1 = new Point(2, 3);
2 Point p2 = new Point(4, 6);
3 p1.moveUp(3.5);
4 p2.moveUp(4.7);
```

  - `p1` and `p2` are called the *call targets* or *context objects*.
  - **Lines 3 and 4** apply the same definition of the `moveUp` method.
  - But how does Java distinguish the change to `p1.y` versus the change to `p2.y`?

The `this` Reference (2)

- In the *method* definition, each *attribute* has an *implicit* `this` which refers to the *context object* in a call to that method.

```
public class Point {
  private double x;
  private double y;
  public Point(double newX, double newY) {
    this.x = newX;
    this.y = newY;
  }
  public void moveUp(double units) {
    this.y = this.y + units;
  }
}
```

- Each time when the *class* definition is used to create a new `Point` *object*, the `this` reference is substituted by the name of the new object.

The `this` Reference (3)

- After we create `p1` as an instance of `Point`

```
Point p1 = new Point(2, 3);
```

- When invoking `p1.moveUp(3.5)`, a version of `moveUp` that is specific to `p1` will be used:

```
public class Point {
  private double x;
  private double y;
  public Point(double newX, double newY) {
    p1.x = newX;
    p1.y = newY;
  }
  public void moveUp(double units) {
    p1.y = p1.y + units;
  }
}
```

The `this` Reference (4)

- After we create `p2` as an instance of `Point`

```
Point p2 = new Point(4, 6);
```

- When invoking `p2.moveUp(4.7)`, a version of `moveUp` that is specific to `p2` will be used:

```
public class Point {
  private double x;
  private double y;
  public Point(double newX, double newY) {
    p2.x = newX;
    p2.y = newY;
  }
  public void moveUp(double units) {
    p2.y = p2.y + units;
  }
}
```


The `this` Reference (5)

The `this` reference can be used to *disambiguate* when the names of *input parameters* clash with the names of *class attributes*.

```
public class Point {
  private double x;
  private double y;
  public Point(double x, double y) {
    this.x = x;
    this.y = y;
  }
  public void setX(double x) {
    this.x = x;
  }
  public void setY(double y) {
    this.y = y;
  }
}
```


The `this` Reference (6.1): Common Error

The following code fragment compiles but is problematic:

```
1  public class Person {
2    private String name;
3    private int age;
4    public Person(String name, int age) {
5      name = name;
6      age = age;
7    }
8    public void setAge(int age) {
9      age = age;
10   }
11 }
```

- Why? [variable **shadowing**]
  Target (LHS) of the assignment (L5) refers to parameter `name` (L4).
- Fix?

The `this` Reference (6.2): Common Error

Always remember to use `this` when *input parameter* names clash with *class attribute* names.

```
public class Person {
  private String name;
  private int age;
  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
  public void setAge(int age) {
    this.age = age;
  }
}
```

OOP: Mutator Methods

- These methods *change* values of attributes.
- We call such methods *mutators* (with `void` return type).

```
public class Person {
  ...
  public void gainWeight(double units) {
    this.weight = this.weight + units;
  }
}
```

```
public class Point {
  ...
  public void moveUp() {
    this.y = this.y + 1;
  }
}
```


OOP: Accessor Methods

- These methods *return* the result of computation based on attribute values.
- We call such methods *accessors* (with non-`void` return type).

```
public class Person {
  ...
  public double getBMI() {
    double bmi = this.height / (this.weight * this.weight);
    return bmi;
  } }
```

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


OOP: Method Calls

```
1 Point p1 = new Point(3, 4);
2 Point p2 = new Point(-4, -3);
3 System.out.println(p1.getDistanceFromOrigin());
4 System.out.println(p2.getDistanceFromOrigin());
5 p1.moveUp(1);
6 p2.moveUp(1);
7 System.out.println(p1.getDistanceFromOrigin());
8 System.out.println(p2.getDistanceFromOrigin());
```

- **Lines 1 and 2** create two different instances of `Point`
- **Lines 3 and 4:** invoking the same accessor method on two different instances returns *distinct* values
- **Lines 5 and 6:** invoking the same mutator method on two different instances results in *independent* changes
- **Lines 3 and 7:** invoking the same accessor method on the same instance *may* return *distinct* values, why? **Line 5**

OOP: Use of Mutator vs. Accessor Methods

- 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();` ✓

OOP: 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)`

OOP: Reference Aliasing (1)

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

- **Line 2** copies the number stored in `i` to `j`.
- After **Line 4**, `i`, `j`, `k` refer to three separate integer placeholder, which happen to store the same value `3`.

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

- **Line 2** copies the *address* stored in `p1` to `p2`.
- Both `p1` and `p2` refer to the same object in memory!
- `p3`, whose *contents* are same as `p1` and `p2`, refer to a different object in memory.

OOP: Reference Aliasing (2.1)

Problem: Consider assignments to *primitive* variables:

```
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]);
```


OOP: Reference Aliasing (2.2)

Problem: Consider assignments to *reference* variables:

```
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());
```

Java Data Types (1)

A (data) type denotes a set of related *runtime values*.

1. ***Primitive Types***
   - *Integer* Type
     - `int` [set of 32-bit integers]
     - `long` [set of 64-bit integers]
   - *Floating-Point Number* Type
     - `double` [set of 64-bit FP numbers]
   - *Character* Type
     - `char` [set of single characters]
   - *Boolean* Type
     - `boolean` [set of `true` and `false`]
2. ***Reference Type*** : *Complex Type with Attributes and Methods*
   - *String* [set of references to character sequences]
   - *Person* [set of references to Person objects]
   - *Point* [set of references to Point objects]
   - *Scanner* [set of references to Scanner objects]

Java Data Types (2)

- A variable that is declared with a *type* but *uninitialized* is implicitly assigned with its ***default value***.
  - **Primitive Type**
    - `int i;` [`0` is implicitly assigned to `i`]
    - `double d;` [`0.0` is implicitly assigned to `d`]
    - `boolean b;` [`false` is implicitly assigned to `b`]
  - **Reference Type**
    - `String s;` [`null` is implicitly assigned to `s`]
    - `Person jim;` [`null` is implicitly assigned to `jim`]
    - `Point p1;` [`null` is implicitly assigned to `p1`]
    - `Scanner input;` [`null` is implicitly assigned to `input`]
- You *can* use a primitive variable that is *uninitialized*. Make sure the ***default value*** is what you want!
- Calling a method on a *uninitialized* reference variable crashes your program. [*NullPointerException*]
  Always initialize reference variables!

Java Data Types (3.1)

- An ***attribute*** may store the reference to another object.

```
public class Person { private Person spouse; }
```

- Methods may take as ***parameters*** references to other objects.

```
public class Person {
  public void marry(Person other) { ... } }
```

- ***Return values*** from methods may be references to objects.

```
public class Point {
  public void moveUpBy(int i) { y = y + i; }
  Point movedUpBy(int i) {
    Point np = new Point(x, y);
    np.moveUpBy(i);
    return np;
  }
}
```


Java Data Types (3.2.1)

An attribute may be ***multi***-valued, ***reference***-typed
e.g., of type `Point[]`, storing references to `Point` objects.

```
public class PointCollector {
 private Point[] points; private int nop; /* number of points */
 public PointCollector() { this.points = new Point[100]; }
 public void addPoint(double x, double y) {
  this.points[this.nop] = new Point(x, y); this.nop++; }
 public Point[] getPointsInQuadrantI() {
  Point[] ps = new Point[this.nop];
  int count = 0; /* number of points in Quadrant I */
  for(int i = 0; i < this.nop; i++) {
   Point p = this.points[i];
   if(p.x > 0 && p.y > 0) { ps[count] = p; count++; } }
  Point[] q1Points = new Point[count];
  /* ps contains null if count < nop */
  for(int i = 0; i < count; i++) { q1Points[i] = ps[i] }
  return q1Points;
 } }
```

Required Reading: Point and PointCollector

Java Data Types (3.2.2)

```java
public class PointCollectorTester {
  public static void main(String[] args) {
    PointCollector pc = new PointCollector();
    System.out.println(pc.getNumberOfPoints()); /* 0 */
    pc.addPoint(3, 4);
    System.out.println(pc.getNumberOfPoints()); /* 1 */
    pc.addPoint(-3, 4);
    System.out.println(pc.getNumberOfPoints()); /* 2 */
    pc.addPoint(-3, -4);
    System.out.println(pc.getNumberOfPoints()); /* 3 */
    pc.addPoint(3, -4);
    System.out.println(pc.getNumberOfPoints()); /* 4 */
    Point[] ps = pc.getPointsInQuadrantI();
    System.out.println(ps.length);  /* 1 */
    System.out.println("(" +
      ps[0].getX() + ", " + ps[0].getY() + ")"); /* (3, 4) */
  }
}
```


Java Data Types (3.3.1)

An attribute may be of type `ArrayList<Point>`, storing references to `Point` objects.

```java
1  class PointCollector {
2    ArrayList<Point> points;
3    PointCollector() { points = new ArrayList<>(); }
4    void addPoint(Point p) {
5      points.add (p); }
6    void addPoint(double x, double y) {
7      points.add (new Point(x, y)); }
8    ArrayList<Point> getPointsInQuadrantI() {
9      ArrayList<Point> q1Points = new ArrayList<>();
10     for(int i = 0; i < points.size(); i ++) {
11       Point p = points.get(i);
12       if(p.x > 0 && p.y > 0) { q1Points.add (p); } }
13     return q1Points;
14   } }
```

L8 & L9 may be replaced by:

```java
for(Point p : points) { q1Points.add(p); }
```


Java Data Types (3.3.2)

```java
class PointCollectorTester {
  public static void main(String[] args) {
    PointCollector pc = new PointCollector();
    System.out.println(pc.points.size());  /* 0 */
    pc.addPoint(3, 4);
    System.out.println(pc.points.size());  /* 1 */
    pc.addPoint(-3, 4);
    System.out.println(pc.points.size());  /* 2 */
    pc.addPoint(-3, -4);
    System.out.println(pc.points.size());  /* 3 */
    pc.addPoint(3, -4);
    System.out.println(pc.points.size());  /* 4 */
    ArrayList<Point> ps = pc.getPointsInQuadrantI();
    System.out.println(ps.length);  /* 1 */
    System.out.println("(" + ps[0].x + ", " + ps[0].y + ")");
    /* (3, 4) */
  }
}
```


Anonymous Objects (1)

- What's the difference between these two fragments of code?

```java
1  double square(double x) {
2    double sqr = x * x;
3    return sqr; }
```

```java
1  double square(double x) {
2    return x * x; }
```

After **L2**, the result of `x * x`:

- LHS: it can be reused (without recalculating) via the name `sqr`.
- RHS: it is not stored anywhere and returned right away.

- Same principles applies to objects:

```java
1  Person getP(String n) {
2    Person p = new Person(n);
3    return p; }
```

```java
1  Person getP(String n) {
2    return new Person(n); }
```

new Person(n) is an object whose address is not stored in a variable.

- LHS: **L2** stores the address of this anonymous object in `p`.
- RHS: **L2** returns the address of this anonymous object directly.

Anonymous Objects (2.1)

Anonymous objects can also be used as *assignment sources* or *argument values*:

```
class Member {
  private Order[] orders;
  private int noo;
  /* constructor ommitted */
  public void addOrder(Order o) {
    this.orders[this.noo] = o;
    this.noo++;
  }
  public void addOrder(String n, double p, double q) {
    this.addOrder( new Order(n, p, q) );
    /* Equivalent implementation:
     * this.orders[this.noo] = new Order(n, p, q); noo ++;
     */
  }
}
```


Anonymous Objects (2.2)

One more example on using anonymous objects:

```
public class MemberTester {
  public static void main(String[] args) {
    Member m = new Member("Alan");
    Order o = new Order("Americano", 4.7, 3);
    m.addOrder(o);
    m.addOrder( new Order("Cafe Latte", 5.1, 4) );
  }
}
```


The `this` Reference (7.1): Exercise

Consider the `Person` class

```
public class Person {
  private String name;
  private Person spouse;
  public Person(String name) {
    this.name = name;
  }
}
```

How do you implement a mutator method `marry` which marries the current `Person` object to an input `Person` object?

The `this` Reference (7.2): Exercise

```
public void marry(Person other) {
  if(this.spouse != null || other.spouse != null) {
    /* Error: both must be single */
  }
  else { this.spouse = other; other.spouse = this; }
}
```

When we call `jim.marry(elsa)`: `this` is substituted by the ***context object*** `jim`, and `other` by the ***argument*** `elsa`.

```
public void marry(Person other elsa) {
  ...
    jim.spouse = elsa;
    elsa.spouse = jim;
  ...
}
```

OOP: The Dot Notation (2)

- **LHS** of dot *can be more complicated than a variable* :
  - It can be a *path* that brings you to an object

```
public class Person {
  private String name;/* public accessor: name() */
  private Person spouse;/* public accessor: spouse() */
}
```

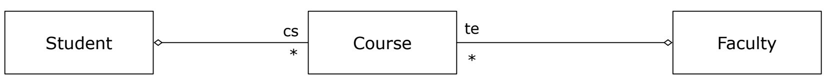
  - Say we have `Person jim = new Person("Jim Davies")`
  - Inquire about `jim`'s name? [`jim.name()`]
  - Inquire about `jim`'s spouse's name? [`jim.spouse().name()`]
  - But what if `jim` is single (i.e., `jim.spouse() == null`)?
    Calling `jim.spouse().name()` will cause *NullPointerException*!!
  - **Quesion.** Assuming that:
    - `jim` is not single. [`jim.spouse() != null`]
    - The marriage is mutual. [`jim.spouse().spouse() != null`]

    What does `jim.spouse().spouse().name()` mean?

    Answer. `jim.name()`

OOP: The Dot Notation (3.1)

In real life, the relationships among classes are sophisticated.




```
class Student {
  String id;
  Course[] cs;
}
```

```
class Course {
  String title;
  Faculty prof;
}
```

```
class Faculty {
  String name;
  Course[] te;
}
```

- Assume: All attributs are ***private*** with the corresponding ***public*** accessor methods.
- In the context of class `Student`:
  - Writing ***cs*** denotes the array of registered courses.
  - Writing ***cs[i]*** (where `i` is a valid index) navigates to the class `Course`, which changes the context to class `Course`.

OOP: The Dot Notation (3.2)

```
class Student {
  String id;
  Course[] cs;
}
```

```
class Course {
  String title;
  Faculty prof;
}
```

```
class Faculty {
  String name;
  Course[] te;
}
```

```
class Student {
  ... /* attributes */
  /* Get the student's id */
  String getID() { return this.id; }
  /* Get the title of the ith course */
  String getTitle(int i) {
    return this.cs[i].getTitle();
  }
  /* Get the instructor's name of the ith course */
  String getName(int i) {
    return this.cs[i].getProf.getName();
  }
}
```


OOP: The Dot Notation (3.3)


```
class Student {
  String id;
  Course[] cs;
}
```

```
class Course {
  String title;
  Faculty prof;
}
```

```
class Faculty {
  String name;
  Course[] te;
}
```


```
class Course {
  ... /* attributes */
  /* Get the course's title */
  String getTitle() { return this.title; }
  /* Get the instructor's name */
  String getName() {
    return this.prof.getName();
  }
  /* Get title of ith teaching course of the instructor */
  String getTitle(int i) {
    return this.prof.getTE()[i].getTitle();
  }
}
```

OOP: The Dot Notation (3.4)

```
class Student {
  String id;
  Course[] cs;
}
```

```
class Course {
  String title;
  Faculty prof;
}
```

```
class Faculty {
  String name;
  Course[] te;
}
```

```
class Faculty {
  ... /* attributes */
  /* Get the instructor's name */
  String getName() {
    return this.name;
  }
  /* Get the title of ith teaching course */
  String getTitle(int i) {
    return this.te[i].getTitle();
  }
}
```


OOP: Equality (1)

```
Point p1 = new Point(2, 3);
Point p2 = new Point(2, 3);
boolean sameLoc = ( p1 == p2 );
System.out.println("p1 and p2 same location?" + sameLoc);
```

```
p1 and p2 same location?  false
```


OOP: Equality (2)

- Recall that
  - A *primitive* variable stores a primitive *value*
    e.g., `double d1 = 7.5; double d2 = 7.5;`
  - A *reference* variable stores the *address* to some object (rather than storing the object itself)
    e.g., `Point p1 = new Point(2, 3)` assigns to `p1` the address of the new `Point` object
    e.g., `Point p2 = new Point(2, 3)` assigns to `p2` the address of *another* new `Point` object
- The binary operator `==` may be applied to compare:
  - *Primitive* variables: their *contents* are compared
    e.g., `d1 == d2` evaluates to *true*
  - *Reference* variables: the *addresses* they store are compared (**rather than** comparing contents of the objects they refer to)
    e.g., `p1 == p2` evaluates to *false* because `p1` and `p2` are addresses of *different* objects, even if their contents are *identical*.

Static Variables (1)

```
public class Account {
  private int id;
  private String owner;
  public int getID() { return this.id; }
  public Account(int id, String owner) {
    this.id = id;
    this.owner = owner;
  }
}
```

```
class AccountTester {
  Account acc1 = new Account(1, "Jim");
  Account acc2 = new Account(2, "Jeremy");
  System.out.println(acc1.getID() != acc2.getID());
}
```

But, managing the unique id's *manually* is *error-prone* !

Static Variables (2)

```
class Account {
  private static int globalCounter = 1;
  private int id; String owner;
  public Account(String owner) {
    this.id = globalCounter;
    globalCounter ++;
    this.owner = owner; } }
```

```
class AccountTester {
  Account acc1 = new Account("Jim");
  Account acc2 = new Account("Jeremy");
  System.out.println(acc1.getID() != acc2.getID()); }
```

- Each instance of a class (e.g., `acc1`, `acc2`) has a *local* copy of each attribute or instance variable (e.g., `id`).
  - Changing `acc1.id` does not affect `acc2.id`.
- A ***static*** variable (e.g., `globalCounter`) belongs to the class.
  - All instances of the class share a *single* copy of the ***static*** variable.
  - Change to `globalCounter` via `acc1` is also visible to `acc2`.

Static Variables (3)

```
public class Account {
  private static int globalCounter = 1;
  private int id; private String owner;
  public Account(String owner) {
    this.id = globalCounter;
    globalCounter ++;
    this.owner = owner;
  } }
```

- *Static* variable `globalCounter` is not instance-specific like *instance* variable (i.e., attribute) `id` is.
- To access a *static* variable:
  - ***No*** context object is needed.
  - Use of the class name suffices, e.g., `Account.globalCounter`.
- Each time `Account`'s constructor is called to create a new instance, the increment effect is *visible to all existing objects* of `Account`.

Static Variables (4.1): Common Error

```
public class Client {
  private Account[] accounts;
  private static int numberOfAccounts = 0;
  public void addAccount(Account acc) {
    accounts[this.numberOfAccounts] = acc;
    this.numberOfAccounts ++;
  } }
```

```
public class ClientTester {
  Client bill = new Client("Bill");
  Client steve = new Client("Steve");
  Account acc1 = new Account();
  Account acc2 = new Account();
  bill.addAccount(acc1);
    /* correctly added to bill.getAccounts()[0] */
  steve.addAccount(acc2);
    /* mistakenly added to steve.getAccounts()[1]! */
}
```


Static Variables (4.2): Common Error

- Attribute `numberOfAccounts` should **not** be declared as `static` as its value should be specific to the client object.
- If it were declared as `static`, then every time the `addAccount` method is called, although on different objects, the increment effect of `numberOfAccounts` will be visible to all `Client` objects.
- Here is the correct version:

```
public class Client {
  private Account[] accounts;
  private int numberOfAccounts;
  public void addAccount(Account acc) {
    accounts[this.numberOfAccounts] = acc;
    this.numberOfAccounts ++;
  }
}
```

Static Variables (5.1): Common Error

```
1 public class Bank {
2   private string branchName;
3   public String getBrachName() { return this.branchName; }
4   private static int nextAccountNumber = 0;
5   public static String getInfo() {
6     nextAccountNumber++;
7     return this.branchName + nextAccountNumber;
8   }
9 }
```

- *Non-static method cannot be referenced from a static context*
- **Line 4** declares that we *can* call the method `getInfo` without instantiating an object of the class `Bank`.
- However, in **Line 7**, the *static* method references a *non-static* attribute, for which we *must* instantiate a `Bank` object.

Static Variables (5.2): Common Error

```
1 public class Bank {
2   private String branchName;
3   public String getBrachName() { return this.branchName; }
4   private static int nextAccountNumber = 0;
5   public static String getInfo() {
6     nextAccountNumber++;
7     return this.branchName + nextAccountNumber;
8   }
9 }
```

- To call `getInfo()`, no instances of `Bank` are required:

```
Bank.getInfo();
```

- *Contradictorily*, to access `branchName`, a *context object* is required:

```
Bank b = new Bank(); b.setBranch("Songdo IBK");
System.out.println(b.getBranchName());
```


Static Variables (5.3): Common Error

There are two possible ways to fix:

1. Remove all uses of *non-static* variables (i.e., `branchName`) in the *static* method (i.e., `useAccountNumber`).
2. Declare `branchName` as a *static* variable.
   - This does not make sense.
     ∵ `branchName` should be a value specific to each `Bank` instance.

OOP: Helper Methods (1)

- After you complete and test your program, feeling confident that it is *correct*, you may find that there are lots of *repetitions*.
- When similar fragments of code appear in your program, we say that your code "*smells*"!
- We may eliminate *repetitions* of your code by:
  - *Factoring out* recurring code fragments into a new method.
  - This new method is called a *helper method*:
    - You can replace every occurrence of the recurring code fragment by a *call* to this helper method, with appropriate argument values.
    - That is, we *reuse* the body implementation, rather than repeating it over and over again, of this helper method via calls to it.
- This process is called *refactoring* of your code:
  Modify the code structure **without** compromising *correctness*.

OOP: Helper (Accessor) Methods (2.1)

```
public class PersonCollector {
  private Person[] ps;
  private final int MAX = 100;/* max # of persons to store */
  private int nop; /* number of persons */
  public PersonCollector() {
    this.ps = new Person[MAX];
  }
  public void addPerson(Person p) {
    this.ps[this.nop] = p;
    this.nop++;
  }
  /* Tasks:
   * 1. An accessor: boolean personExists(String n)
   * 2. A mutator: void changeWeightOf(String n, double w)
   * 3. A mutator: void changeHeightOf(String n, double h)
   */
}
```


OOP: Helper (Accessor) Methods (2.2.1)

```
public class PersonCollector {
  /* ps, MAX, nop, PersonCollector(), addPerson */
  public boolean personExists(String n) {
    boolean found = false;
    for(int i = 0; i < nop; i ++) {
      if(ps[i].getName().equals(n)) { found = true; } }
    return found;
  }
  public void changeWeightOf(String n, double w) {
    for(int i = 0; i < nop; i ++) {
      if(ps[i].getName().equals(n)) { ps[i].setWeight(w); } }
  }
  public void changeHeightOf(String n, double h) {
    for(int i = 0; i < nop; i ++) {
      if(ps[i].getName().equals(n)) { ps[i].setHeight(h); } }
  }
}
```


OOP: Helper (Accessor) Methods (2.2.2)

```
public class PersonCollector {/* code smells:repetitions! */
  /* ps, MAX, nop, PersonCollector(), addPerson */
  public boolean personExists(String n) {
    boolean found = false;
    for(int i = 0; i < nop; i ++) {
      if(ps[i].getName().equals(n)) { found = true; } }
    return found;
  }
  public void changeWeightOf(String n, double w) {
    for(int i = 0; i < nop; i ++) {
      if(ps[i].getName().equals(n)) {ps[i].setWeight(w);} }
  }
  public void changeHeightOf(String n, double h) {
    for(int i = 0; i < nop; i ++) {
      if(ps[i].getName().equals(n)) {ps[i].setHeight(h);} }
  }
}
```


OOP: Helper (Accessor) Methods (2.3)

```
public class PersonCollector { /* Code Smell Eliminated */
  /* ps, MAX, nop, PersonCollector(), addPerson */
  private int indexOf (String n) { /* Helper Methods */
    int i = -1;
    for(int j = 0; j < nop; j ++) {
      if(ps[j].getName().equals(n)) { i = j; }
    }
    return i; /* -1 if not found; >= 0 if found. */
  }
  public boolean personExists(String n) {
    return this.indexOf (n) >= 0; }
  public void changeWeightOf(String n, double w) {
    int i = indexOf (n); if(i >= 0) { ps[i].setWeight(w); }
  }
  public void changeHeightOf(String n, double h) {
    int i = indexOf (n); if(i >= 0) { ps[i].setHeight(h); }
  }
}
```

OOP: Helper (Accessor) Methods (3.1)

Problems:

- A `Point` class with `x` and `y` coordinate values.
- Accessor `double` *`getDistanceFromOrigin`*`()`.
  `p.getDistanceFromOrigin()` returns the distance between `p` and `(0, 0)`.
- Accessor `double` *`getDistancesTo`*`(Point p1, Point p2)`.
  `p.getDistancesTo(p1, p2)` returns the sum of distances between `p` and `p1`, and between `p` and `p2`.
- Accessor `double` *`getTriDistances`*`(Point p1, Point p2)`.
  `p.getDistancesTo(p1, p2)` returns the sum of distances between `p` and `p1`, between `p` and `p2`, and between `p1` and `p2`.

OOP: Helper (Accessor) Methods (3.2)

```
class Point { /* code smells:repetitions! */
  double x; double y;

  double getDistanceFromOrigin() {
    return Math.sqrt(Math.pow(this.x - 0, 2) + Math.pow(this.y - 0, 2)); }

  double getDistancesTo(Point p1, Point p2) {
    return
      Math.sqrt(Math.pow(this.x - p1.x, 2) + Math.pow(y - p1.y, 2))
      +
      Math.sqrt(Math.pow(this.x - p2.x, 2) + Math.pow(y - p2.y, 2)); }

  double getTriDistances(Point p1, Point p2) {
    return
      Math.sqrt(Math.pow(this.x - p1.x, 2) + Math.pow(y - p1.y, 2))
      +
      Math.sqrt(Math.pow(this.x - p2.x, 2) + Math.pow(y - p2.y, 2))
      +
      Math.sqrt(Math.pow(p1.x - p2.x, 2) + Math.pow(p1.y - p2.y, 2));
  }
}
```


OOP: Helper (Accessor) Methods (3.3)

- The code pattern

  `Math.sqrt(Math.pow(... - ..., 2) + Math.pow(... - ..., 2))`

  is written down explicitly every time we need to use it.
- Create a *helper method* out of it, with the right *parameter* and *return* types:

```
double getDistanceFrom(double otherX, double otherY) {
  return Math.sqrt(
    Math.pow(ohterX - this.x, 2)
    +
    Math.pow(otherY - this.y, 2));
}
```


OOP: Helper (Accessor) Methods (3.4)

```
public class Point { /* Code Smell Eliminated */
  private double x; private double y;
  double getDistanceFrom(double otherX, double otherY) {
    return Math.sqrt(Math.pow(ohterX - this.x, 2) +
           Math.pow(otherY - this.y, 2));
  }
  double getDistanceFromOrigin() {
    return this.getDistanceFrom(0, 0);
  }
  double getDistancesTo(Point p1, Point p2) {
    return this.getDistanceFrom(p1.x, p1.y) +
           this.getDistanceFrom(p2.x, p2.y);
  }
  double getTriDistances(Point p1, Point p2) {
    return this.getDistanceFrom(p1.x, p1.y) +
           this.getDistanceFrom(p2.x, p2.y) +
           p1.getDistanceFrom(p2.x, p2.y)
  }
}
```

OOP: Helper (Mutator) Methods (4.1)

```
public class Student {
  private String name;
  private double balance;
  public Student(String n, double b) {
    name = n;
    balance = b;
  }

  /* Tasks:
   * 1. A mutator void receiveScholarship(double val)
   * 2. A mutator void payLibraryOverdue(double val)
   * 3. A mutator void payCafeCoupons(double val)
   * 4. A mutator void transfer(Student other, double val)
   */
}
```


OOP: Helper (Mutator) Methods (4.2.1)

```
public class Student {
  /* name, balance, Student(String n, double b) */
  public void receiveScholarship(double val) {
    balance = balance + val;
  }
  public void payLibraryOverdue(double val) {
    balance = balance - val;
  }
  public void payCafeCoupons(double val) {
    balance = balance - val;
  }
  public void transfer(Student other, double val) {
    balance = balance - val;
    other.balance = other.balance + val;
  }
}
```


OOP: Helper (Mutator) Methods (4.2.2)

```
public class Student { /* code smells:repetitions! */
  /* name, balance, Student(String n, double b) */
  public void receiveScholarship(double val) {
    balance = balance + val;
  }
  public void payLibraryOverdue(double val) {
    balance = balance - val;
  }
  public void payCafeCoupons(double val) {
    balance = balance - val;
  }
  public void transfer(Student other, double val) {
    balance = balance - val;
    balance = other.balance + val;
  }
}
```


OOP: Helper (Mutator) Methods (4.3)

```
public class Student { /* Code Smell Eliminated */
  /* name, balance, Student(String n, double b) */
  public void deposit (double val) { /* Helper Method */
    balance = balance + val;
  }
  public void withdraw (double val) { /* Helper Method */
    balance = balance - val;
  }
  public void receiveScholarship(double val) { this.deposit (val); }
  public void payLibraryOverdue(double val) { this.withdraw (val); }
  public void payCafeCoupons(double val) { this.withdraw (val) }
  public void transfer(Student other, double val) {
    this.withdraw (val);
    other.deposit (val);
  }
}
```

Index (1)

Index (2)

Index (3)

Index (4)

Index (5)

Index (6)

Index (7)

Index (8)