Classes and Objects

EECS1021:
Object Oriented Programming:
from Sensors to Actuators
Winter 2019

CHEN-WEI WANG

- We have developed the Java code *solely* within `main` 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
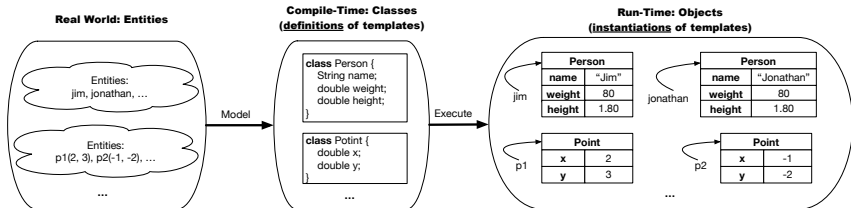
○ Study this tutorial video that walks you through the idea of *object orientation* .

○ 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.

LASSONDE
SCHOOL OF ENGINEERING

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]

- Persons share these common *attributes* and *behaviour*.
  - Each person possesses an age, a weight, and a height.
  - Each person's age, weight, and height might be *distinct*
    e.g., `jim` is 50-years old, 1.8-meters tall and 80-kg heavy
    e.g., `jonathan` is 65-years old, 1.73-meters tall and 90-kg heavy

- Each person, depending on the `specific values` of their
  attributes, might 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 $\left[\frac{80}{1.8^2}\right]$
  - `jonathan`'s BMI is based on his own height and weight $\left[\frac{90}{1.73^2}\right]$

OO Thinking: Templates vs. Instances (1.3)

- 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* into multiple instances.
  - `Person` instance `jim`
  - `Person` instance `jonathan`
- Each *instance* may have *specific values* for the attributes.
  - Each `Person` instance has an age:
    - `jim` is 50-years old
    - `jonathan` is 65-years old
- Therefore, instances of the same template may exhibit *distinct behaviour*.
  - Each `Person` instance can get older:
    - `jim` getting older from 50 to 51
    - `jonathan` getting older from 65 to 66

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]

- Points share these common *attributes* and *behaviour*.
  - Each point possesses an x-coordinate and a y-coordinate.
  - Each point's location might be *distinct*
    e.g., $p1$ is located at $(3, 4)$
    e.g., $p2$ is located at $(-4, -3)$
- Each point, depending on the **specific values** of their attributes (i.e., locations), might 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 (2.3)

- 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* into multiple instances.
  - `Point` instance `p1`
  - `Point` instance `p2`
- Each *instance* may have *specific values* for the attributes.
  - Each `Point` instance has an age:
    - `p1` is at `(3, 4)`
    - `p2` is at `(-3, -4)`
- Therefore, instances of the same template may exhibit *distinct behaviour*.
  - Each `Point` instance can move up:
    - `p1` moving up from `(3, 3)` results in `(3, 4)`
    - `p2` moving up from `(-3, -4)` results in `(-3, -3)`

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.

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

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

Define Constructors for Creating Objects (1.1)

- Within class Point, you define *constructors* , specifying how instances of the Point template may be created.

```
public class Point {
  ... /* attributes: x, y */
  Point(double newX, double newY) {
    x = newX;
    y = newY; } }
```

- In the corresponding tester class, each *call* to the Point constructor creates an instance of the Point template.

```
public class PointTester {
  public static void main(String[] args) {
    Point p1 = new Point(2, 4);
    println(p1.x + " " + p1.y);
    Point p2 = new Point(-4, -3);
    println(p2.x + " " + p2.y); } }
```

```
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`.

Point	
x	2.0
y	4.0

p1
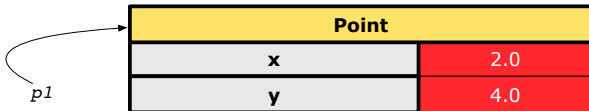
OOP:
Define Constructors for Creating Objects (2.1)

- Within class `Person`, you define *constructors*, specifying how instances of the `Person` template may be created.

```
public class Person {
 ... /* attributes: age, nationality, weight, height */
 Person(int newAge, String newNationality) {
   age = newAge;
   nationality = newNationality; } }
```

- In the corresponding tester class, each *call* to the `Person` constructor creates an instance of the `Person` template.

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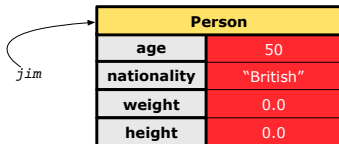
```
Person jim = new Person(50, "British");
```

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`.

jim

Person	
age	50
nationality	"British"
weight	0.0
height	0.0

- 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* .

After calling a *constructor* to create an object:
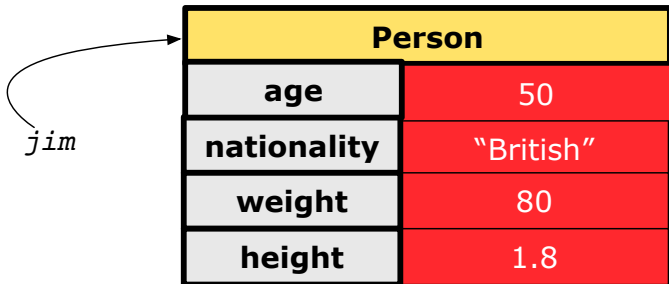
```
Person jim = new Person(50, "British", 80, 1.8);
```

Person	
age	50
nationality	"British"
weight	80
height	1.8

jim

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

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

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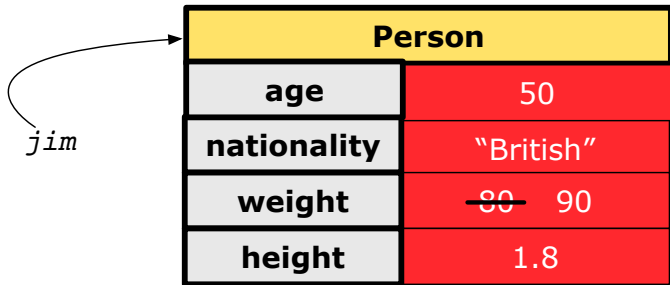- Retuned value $\frac{80}{(1.8)^2}$ of `jim.getBMI()` stored in variable `bmi`.

Person	
age	50
nationality	"British"
weight	80
height	1.8

jim
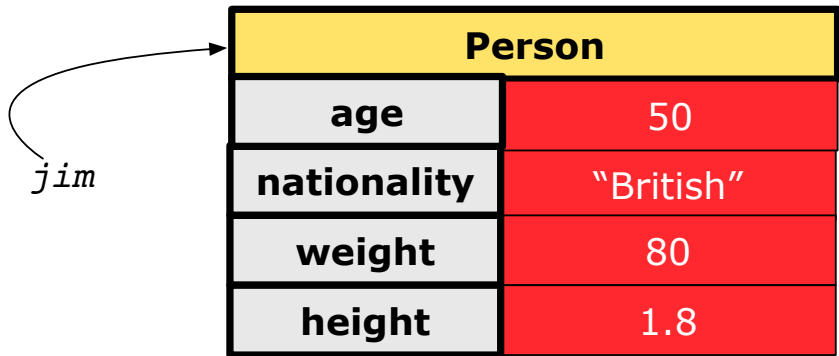
Visualizing Objects at Runtime (2.3)

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.
  $\Rightarrow$ `jim` points to the same object!

Person	
age	50
nationality	"British"
weight	~~80~~ 90
height	1.8

jim

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

```
bmi = p.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.

Person	
age	50
nationality	"British"
weight	~~80~~ 90
height	1.8

jim

The `this` Reference (1)

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

```
class Point {
  double x; double y;
  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.

```
class Point {
  double x;
  double y;
  Point(double newX, double newY) {
    this.x = newX;
    this.y = newY;
  }
  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.

- 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:

```
class Point {
  double x;
  double y;
  Point(double newX, double newY) {
    p1 .x = newX;
    p1 .y = newY;
  }
  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:

```
class Point {
  double x;
  double y;
  Point(double newX, double newY) {
    p2.x = newX;
    p2.y = newY;
  }
  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*.

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

The following code fragment compiles but is problematic:

```java
class Person {
  String name;
  int age;
  Person(String name, int age) {
    name = name;
    age = age;
  }
  void setAge(int age) {
    age = age;
  }
}
```

Why? Fix?

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

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

OOP: Methods (1.1)

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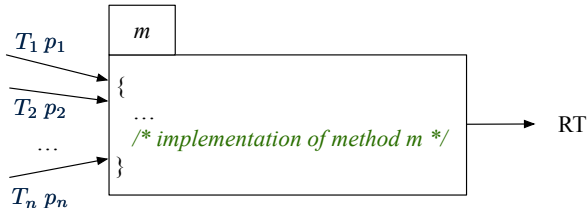
- The *Header* of a method consists of:
  - Return type   [*RT* (which can be `void`)]
  - Name of method   [*m*]
  - Zero or more *parameter names*   [$p_1, p_2, \ldots, p_n$]
  - The corresponding *parameter types*   [$T_1, T_2, \ldots, T_n$]
- A call to method *m* has the form: $m(a_1, a_2, \ldots,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

28 of 94

OOP: Methods (1.2)

- In the body of the method, you may
  - Declare and use new *local variables*
    Scope of local variables is only within that method.
  - Use or change values of *attributes*.
  - Use values of *parameters*, if any.

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

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

```
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: 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()`

OOP: Method Calls

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

- 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 (2)

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

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

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

  Point(char axis, double distance) {
    if (axis == 'x') { x = distance; }
    else if (axis == 'y') { y = distance; }
    else { System.out.println("Error: invalid axis.") }
  }
}
```

- For each *class*, you may define *one or more* <mark>*constructors*</mark> :
  - ○ *Names* of all constructors must match the class name.
  - ○ *No return types* need to be specified for constructors.
  - ○ Each constructor must have a *distinct* list of *input parameter types*.
  - ○ Each *parameter* that is used to initialize an attribute must have a *matching type*.
  - ○ The *body* of each constructor specifies how <mark>*some or all attributes*</mark> may be *initialized*.

```
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.x + ", " + p1.y + ")");
```
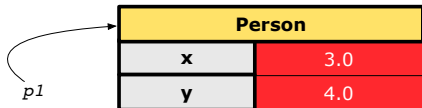
```
(2.0, 4.0)
```

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

```java
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);
 }
}
```
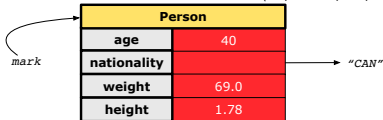
A constructor may only *initialize* some attributes and leave others *uninitialized*.

```java
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);
  }
}
```
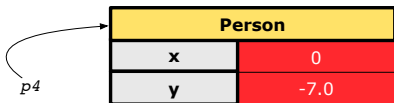
*Point p1 = **new** Point(3, 4)*

Person	
x	3.0
y	4.0

p1

*Point p2 = **new** Point(-3, -2)*

Person	
x	-3.0
y	-2.0

p2

*Point p3 = **new** Point('x', 5)*

Person	
x	5.0
y	0

p3

*Point p4 = **new** Point('y', -7)*

Person	
x	0
y	-7.0

p4
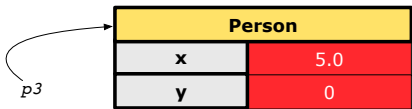
OOP: Object Creation (6)

- 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: Mutator Methods

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

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

```
public class Point {
  ...
  void moveUp() {
    y = 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 {
  ...
  double getBMI() {
    double bmi = height / (weight * weight);
    return bmi;
  }
}
```

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

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 <mark>*constructor*</mark> 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 <mark>*mutator*</mark> 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 <mark>*accessor method*</mark> 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)`

```
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.

Problem: Consider assignments to *primitive* variables:

```
1   int i1 = 1;
2   int i2 = 2;
3   int i3 = 3;
4   int[] numbers1 = {i1, i2, i3};
5   int[] numbers2 = new int[numbers1.length];
6   for(int i = 0; i < numbers1.length; i ++) {
7     numbers2[i] = numbers1[i];
8   }
9   numbers1[0] = 4;
10  System.out.println(numbers1[0]);
11  System.out.println(numbers2[0]);
```

Problem: Consider assignments to *reference* variables:

```
1  Person alan = new Person("Alan");
2  Person mark = new Person("Mark");
3  Person tom = new Person("Tom");
4  Person jim = new Person("Jim");
5  Person[] persons1 = {alan, mark, tom};
6  Person[] persons2 = new Person[persons1.length];
7  for(int i = 0; i < persons1.length; i ++) {
8    persons2[i] = persons1[i]; }
9  persons1[0].setAge(70);
10 System.out.println(jim.age);
11 System.out.println(alan.age);
12 System.out.println(persons2[0].age);
13 persons1[0] = jim;
14 persons1[0].setAge(75);
15 System.out.println(jim.age);
16 System.out.println(alan.age);
17 System.out.println(persons2[0].age);
```

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]

- 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* <u>reference</u> variable crashes your program.   [*NullPointerException*]

  Always initialize <u>reference</u> variables!

Java Data Types (3.1)

- An attribute may store the reference to some object.

```
class Person { Person spouse; }
```

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

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

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

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

An attribute may be of type *Point[]* , storing references to
`Point` objects.

```java
class PointCollector {
  Point[] points; int nop; /* number of points */
  PointCollector() { points = new Point[100]; }
  void addPoint(double x, double y) {
    points[nop] = new Point(x, y); nop++; }
  Point[] getPointsInQuadrantI() {
    Point[] ps = new Point[nop];
    int count = 0; /* number of points in Quadrant I */
    for(int i = 0; i < nop; i ++) {
      Point p = 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
1   class PointCollectorTester {
2    public static void main(String[] args) {
3      PointCollector pc = new PointCollector();
4      System.out.println(pc.nop);  /* 0 */
5      pc.addPoint(3, 4);
6      System.out.println(pc.nop);  /* 1 */
7      pc.addPoint(-3, 4);
8      System.out.println(pc.nop);  /* 2 */
9      pc.addPoint(-3, -4);
10     System.out.println(pc.nop);  /* 3 */
11     pc.addPoint(3, -4);
12     System.out.println(pc.nop);  /* 4 */
13     Point[] ps = pc.getPointsInQuadrantI();
14     System.out.println(ps.length);  /* 1 */
15     System.out.println("(" + ps[0].x + ", " + ps[0].y + ")");
16     /* (3, 4) */
17    }
18  }
```

Java Data Types (3.3.1)

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

```
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:

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

```java
1  class PointCollectorTester {
2   public static void main(String[] args) {
3     PointCollector pc = new PointCollector();
4     System.out.println(pc.points.size()); /* 0 */
5     pc.addPoint(3, 4);
6     System.out.println(pc.points.size()); /* 1 */
7     pc.addPoint(-3, 4);
8     System.out.println(pc.points.size()); /* 2 */
9     pc.addPoint(-3, -4);
10    System.out.println(pc.points.size()); /* 3 */
11    pc.addPoint(3, -4);
12    System.out.println(pc.points.size()); /* 4 */
13    ArrayList<Point> ps = pc.getPointsInQuadrantI();
14    System.out.println(ps.length); /* 1 */
15    System.out.println("(" + ps[0].x + ", " + ps[0].y + ")");
16    /* (3, 4) */
17   }
18 }
```

Consider the Person class

```
class Person {
 String name;
 Person spouse;
 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?

```
void marry(Person other) {
 if(this.spouse != null || other.spouse != null) {
   System.out.println("Error: both must be single.");
 }
 else { this.spouse = other; other.spouse = this; }
}
```

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

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

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

```
class Person {
  String name;
  Person 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 trigger *NullPointerException*!!
  - Assuming that:
    - `jim` is not single. [jim.spouse != null]
    - The marriage is mutual. [jim.spouse.spouse != null]
    What does `jim.spouse.spouse.name` mean? [jim.name]

OOP: The Dot Notation (3.1)

In real life, the relationships among classes are sophisticated.
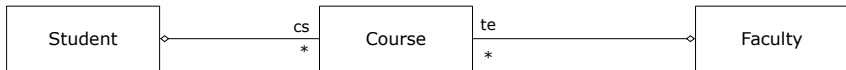

```
class Student {
  String id;
  Course[] cs;
}
```

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

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

Aggregation links between classes constrain how you can *navigate* among these classes.

e.g., 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.

```
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 getCourseTitle(int i) {
  return this.cs[i].title;
 }
 /* Get the instructor's name of the ith course */
 String getInstructorName(int i) {
  return this.cs[i].prof.name;
 }
}
```

```
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 getInstructorName() {
  return this.prof.name;
 }
 /* Get title of ith teaching course of the instructor */
 String getCourseTitleOfInstructor(int i) {
  return this.prof.te.[i].title;
 }
}
```

```
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 getCourseTitle(int i) {
  return this.te[i].title;
 }
}
```

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)

```
class Account {
  int id;
  String owner;
  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.id != acc2.id);
}
```

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

Static Variables (2)

```
class Account {
  static int globalCounter = 1;
  int id; String owner;
  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.id != acc2.id); }
```

- 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 <u>share</u> a *single* copy of the *static* variable.
  ○ Change to globalCounter via c1 is also visible to c2.

```
class Account {
  static int globalCounter = 1;
  int id; String owner;
  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.

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

```
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.accounts[0] */
 steve.addAccount(acc2);
  /* mistakenly added to steve.accounts[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:

```java
class Client {
  Account[] accounts;
  int numberOfAccounts = 0;
  void addAccount(Account acc) {
    accounts[numberOfAccounts] = acc;
    numberOfAccounts ++;
  }
}
```

```
1  public class Bank {
2     public string branchName;
3     public static int nextAccountNumber = 1;
4     public static void useAccountNumber() {
5        System.out.println (branchName + ...);
6        nextAccountNumber ++;
7     }
8  }
```

- *Non-static method cannot be referenced from a static context*
- **Line 4** declares that we *can* call the method userAccountNumber without instantiating an object of the class Bank.
- However, in **Lined 5**, the *static* method references a *non-static* attribute, for which we *must* instantiate a Bank object.

```
1  public class Bank {
2     public string branchName;
3     public static int nextAccountNumber = 1;
4     public static void useAccountNumber() {
5        System.out.println (branchName + ...);
6        nextAccountNumber ++;
7     }
8  }
```

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

```
Bank .useAccountNumber();
```

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

```
Bank b1 = new Bank(); b1.setBranch("Songdo IBK");
System.out.println( b1 .branchName);
```

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)

- <u>After</u> 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 <u>every occurrence</u> 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*.

```java
class PersonCollector {
 Person[] ps;
 final int MAX = 100; /* max # of persons to be stored */
 int nop; /* number of persons */
 PersonCollector() {
   ps = new Person[MAX];
 }
 void addPerson(Person p) {
   ps[nop] = p;
   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)
  */
}
```

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

OOP: Helper (Accessor) Methods (2.2.2)

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

```
class PersonCollector { /* Eliminate code smell.   */
 /* ps, MAX, nop, PersonCollector(), addPerson */
 int indexOf (String n) { /* Helper Methods */
   int i = -1;
   for(int j = 0; j < nop; j ++) {
     if(ps[j].name.equals(n)) { i = j; }
   }
   return i; /* -1 if not found; >= 0 if found. */
 }
 boolean personExists(String n) { return indexOf (n) >= 0; }
 void changeWeightOf(String n, double w) {
   int i = indexOf (n); if(i >= 0) { ps[i].setWeight(w); }
 }
 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)

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

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

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

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

- 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));
}
```

```java
class Point {
 double x; 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)

```
class Student {
  String name;
  double balance;
  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)

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

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

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

Index (2)

Index (5)

Index (6)