

Classes and Objects



EECS2030 B: Advanced
Object Oriented Programming
Fall 2018

CHEN-WEI WANG

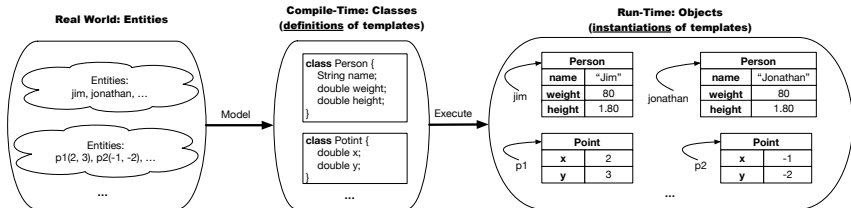
Separation of Concerns: App/Tester vs. Model

- In EECS1022:
 - **Model Component**: One or More Java Classes
e.g., Person vs. SMS, Student, CourseRecord
 - Another Java class that “manipulates” the model class (by creating instances and calling methods):
 - **Controller** (e.g., BMIActivity, BankActivity). Effects? Visualized (via a GUI) at connected tablet
 - **Tester** with main (e.g., PersonTester, BankTester). Effects? Seen (as textual outputs) at console
- 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



- o Study this tutorial video that walks you through the idea of **object orientation**.
- o We **observe** how real-world *entities* behave.
- o We **model** the common *attributes* and *behaviour* of a set of entities in a single *class*.
- o 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 *behaviour*; and
 - Interact with each other.
- Goals: Solve problems *programmatically* by
 - *Classifying* entities of interest
Entities in the same class share *common* attributes and behaviour.
 - *Manipulating* data that represent these entities
Each entity is represented by *specific* values.

OO Thinking: Templates vs. Instances (1.1)

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 (1.2)

- 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
 - `jonathan`'s BMI is based on his own height and weight

$$\left[\frac{80}{1.8^2} \right]$$
$$\left[\frac{90}{1.73^2} \right]$$

OO Thinking: Templates vs. Instances (2.1)

Points on a two-dimensional plane are identified by their signed distances from the X- and Y-axes. 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 (2.2)

- 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 (3)

- A **template** defines what's **shared** by a set of related entities.
 - Common **attributes** (age in `Person`, `x` in `Point`)
 - Common **behaviour** (get older for `Person`, move up for `Point`)
- Each template may be **instantiated** into multiple instances.
 - `Person` instances: jim and jonathan
 - `Point` instances: p1 and p2
- 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
 - Each `Point` instance has a location:
p1 is at (3,4), p2 is at (-3,-4)
- 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.
 - 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 \approx 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 {  
    int age;  
    String nationality;  
    double weight;  
    double height;  
}
```

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

OOP:

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); } }  
}
```

OOP:

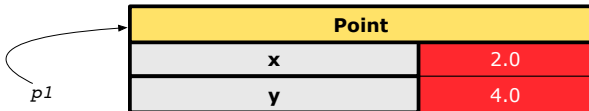
Define Constructors for Creating Objects (1.2)

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

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

Point	
x	2.0
y	4.0

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

OOP:

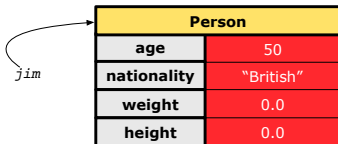
Define Constructors for Creating Objects (2.2)

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

- 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

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



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*

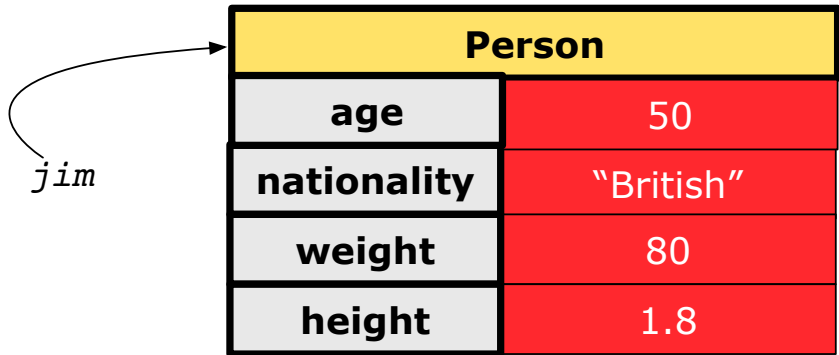
```
double bmi = jim.getBMI();
```
 - Modified using *mutator methods*

```
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);
```

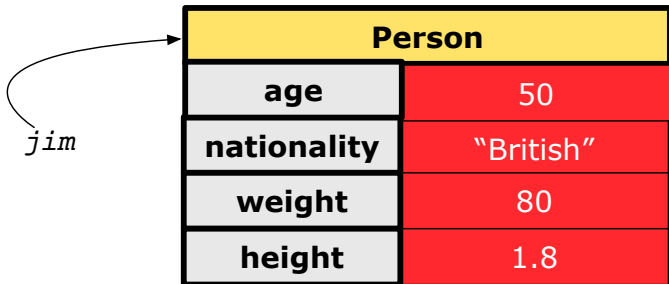


Visualizing Objects at Runtime (2.2)

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

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

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

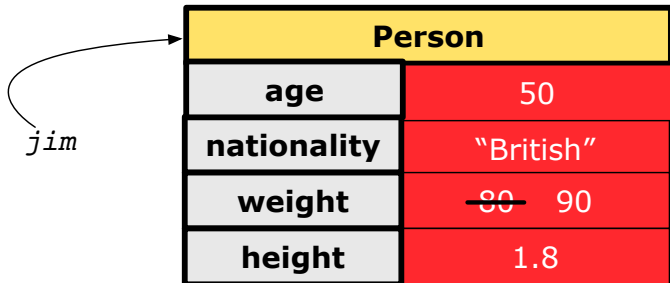


Visualizing Objects at Runtime (2.3)

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!

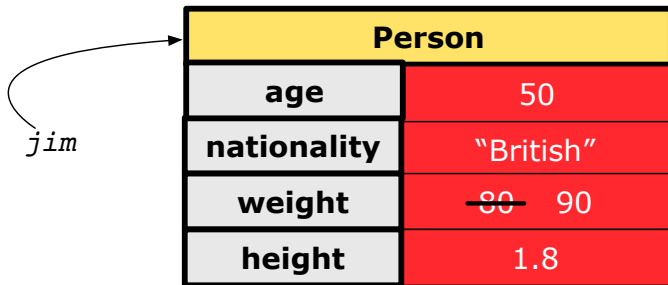


Visualizing Objects at Runtime (2.4)

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

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

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



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

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.

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:

```
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 `this` Reference (6.1): Common Error

The following code fragment compiles but is problematic:

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

Why? Fix?

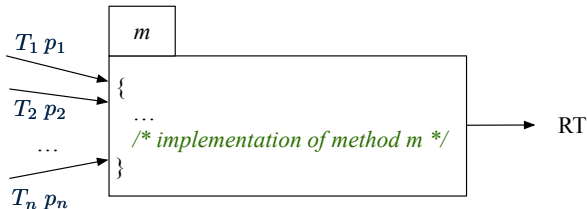
The `this` Reference (6.2): Common Error

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

```
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: [see here]
 - Return type [RT (which can be void)]
 - Name of method [m]
 - Zero or more *parameter names* [p₁, p₂, ..., p_n]
 - The corresponding *parameter types* [T₁, T₂, ..., T_n]
- A call to method *m* has the form: $m(a_1, a_2, \dots, a_n)$
 Types of **argument values** a_1, a_2, \dots, a_n must match the the corresponding parameter types T_1, T_2, \dots, T_n .

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** $obj.m(\dots)$:
 - 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(pl.getDistanceFromOrigin());`

OOP: The Dot Notation (1.1)

A binary operator:

- LHS stores an address (which denotes an object)
- RHS the name of an attribute or a method
- LHS . RHS means:

Locate the context object whose address is stored in **LHS**,
then apply RHS.

What if LHS stores `null`?

`[NullPointerException]`

OOP: The Dot Notation (1.2)

- 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

OOP: Class Constructors (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 (2)

```
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 */
    }
}
```

OOP: Class Constructors (3)

```
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.") }  
    }  
}
```

OOP: Class Constructors (4)

- 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.
 - 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 *some or all* *attributes* may be *initialized*.

OOP: Object Creation (1)

```
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)
```

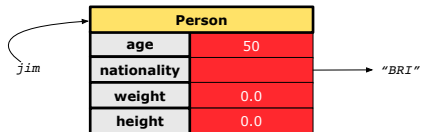
OOP: Object Creation (2)

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

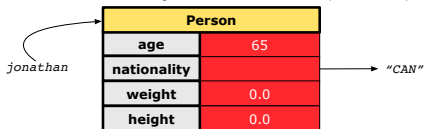
```
public class PersonTester {  
    public static void main(String[] args) {  
        /* initialize age and nationality only */  
        Person jim = new Person(50, "BRI");  
        /* initialize age and nationality only */  
        Person jonathan = new Person(65, "CAN");  
        /* initialize 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)

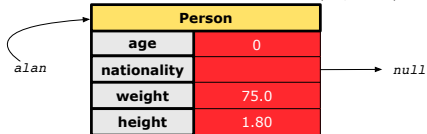
`Person jim = new Person(50, "BRI")`



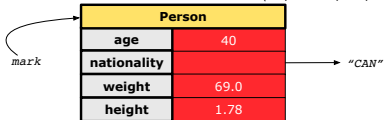
`Person jonathan = new Person(65, "CAN")`



`Person alan = new Person(75, 1.80)`



`Person mark = new Person(40, "CAN", 69, 1.78)`



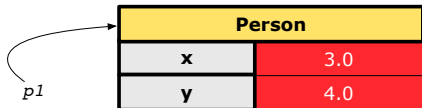
OOP: Object Creation (4)

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

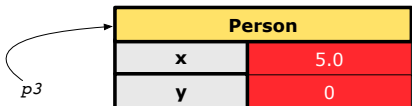
```
public class PointTester {  
    public static void main(String[] args) {  
        Point p1 = new Point(3, 4);  
        Point p2 = new Point(-3 -2);  
        Point p3 = new Point('x', 5);  
        Point p4 = new Point('y', -7);  
    }  
}
```

OOP: Object Creation (5)

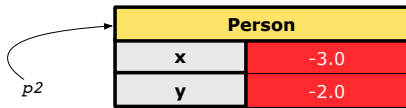
Point p1 = new Point(3, 4)



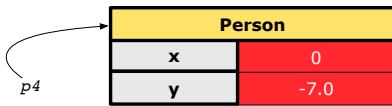
Point p3 = new Point('x', 5)



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



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



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 **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: Object Alias (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 System.out.println(p3 == p1 || p3 == p2); /* false */
5 System.out.println(p3.x == p1.x && p3.y == p1.y); /* true */
6 System.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.

OO Program Programming: Object Alias (2.1)

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]);
```

OO Program Programming: Object Alias (2.2)

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 + 1) % persons1.length]; }
9 persons1[0].setAge(70);
10 System.out.println(jim.age); /* 0 */
11 System.out.println(alan.age); /* 70 */
12 System.out.println(persons2[0].age); /* 0 */
13 persons1[0] = jim;
14 persons1[0].setAge(75);
15 System.out.println(jim.age); /* 75 */
16 System.out.println(alan.age); /* 70 */
17 System.out.println(persons2[0].age); /* 0 */
```

OO Program Programming: Object Alias (3)

```
Person tom = new Person("TomCruise");
Person ethanHunt = tom;
Person spy = ethanHunt;
tom.setWeight(77); print(tom.weight); /* 77 */
ethanHunt.gainWeight(10); print(tom.weight); /* 87 */
spy.loseWeight(10); print(tom.weight); /* 77 */
Person prof = new Person("Jackie"); prof.setWeight(80);
spy = prof; prof = tom; tom = spy;
print(prof.name+" teaches 2030");/*TomCruise teaches 2030*/
print("EthanHunt is "+ethanHunt.name);/*EthanHunt is TomCruise*/
print("EthanHunt is "+spy.name);/*EthanHunt is Jackie*/
print("TomCruise is "+tom.name);/*TomCruise is Jackie*/
print("Jackie is "+prof.name);/*Jackie is TomCruise*/
```

- An *object* at runtime may have *more than one identities*. Its *address* may be stored in multiple *reference variables*.
- Calling a *method* on one of an object's identities has the *same effect* as calling the same method on any of its other identities.

Anonymous Objects (1)

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

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

```
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:

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

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

`new Person(n)` denotes an object without a name reference.

- 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 {
    Order[] orders;
    int noo;
    /* constructor omitted */
    void addOrder(Order o) {
        orders[noo] = o;
        noo++;
    }
    void addOrder(String n, double p, double q) {
        addOrder(new Order(n, p, q));
        /* Equivalent implementation:
        * orders[noo] = new Order(n, p, q);
        * noo++; */
    }
}
```

Anonymous Objects (2.2)

One more example on using anonymous objects:

```
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));  
    }  
}
```

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 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;  
    }  
}
```

Java Data Types (3.2.1)

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

```
1 class PointCollector {
2     Point[] points; int nop; /* number of points */
3     PointCollector() { points = new Point[100]; }
4     void addPoint(double x, double y) {
5         points[nop] = new Point(x, y); nop++; }
6     Point[] getPointsInQuadrantI() {
7         Point[] ps = new Point[nop];
8         int count = 0; /* number of points in Quadrant I */
9         for(int i = 0; i < nop; i ++ ) {
10            Point p = points[i];
11            if(p.x > 0 && p.y > 0) { ps[count] = p; count ++; } }
12        Point[] qlPoints = new Point[count];
13        /* ps contains null if count < nop */
14        for(int i = 0; i < count; i ++ ) { qlPoints[i] = ps[i] }
15        return qlPoints;
16    } }
```

Java Data Types (3.2.2)

```
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 }
```

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

Static Variables (3)

```
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`.

Static Variables (4.1): Common Error

```
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:

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


Static Variables (5.1): Common Error

```
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 `useAccountNumber` 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.

Static Variables (5.2): Common Error

```
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);
```

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.

Index (1)

Separation of Concerns: App/Tester vs. Model

Object Orientation:

Observe, Model, and Execute

Object-Oriented Programming (OOP)

OO Thinking: Templates vs. Instances (1.1)

OO Thinking: Templates vs. Instances (1.2)

OO Thinking: Templates vs. Instances (2.1)

OO Thinking: Templates vs. Instances (2.2)

OO Thinking: Templates vs. Instances (3)

OOP: Classes \approx Templates

OOP:

Define Constructors for Creating Objects (1.1)

OOP:

Define Constructors for Creating Objects (1.2)

Index (2)

OOP:

Define Constructors for Creating Objects (2.1)

OOP:

Define Constructors for Creating Objects (2.2)

Visualizing Objects at Runtime (1)

Visualizing Objects at Runtime (2.1)

Visualizing Objects at Runtime (2.2)

Visualizing Objects at Runtime (2.3)

Visualizing Objects at Runtime (2.4)

The `this` Reference (1)

The `this` Reference (2)

The `this` Reference (3)

The `this` Reference (4)

The `this` Reference (5)

Index (3)

The `this` Reference (6.1): Common Error

The `this` Reference (6.2): Common Error

OOP: Methods (1.1)

OOP: Methods (1.2)

OOP: Methods (2)

OOP: Methods (3)

OOP: The Dot Notation (1.1)

OOP: The Dot Notation (1.2)

OOP: Method Calls

OOP: Class Constructors (1)

OOP: Class Constructors (2)

OOP: Class Constructors (3)

OOP: Class Constructors (4)

OOP: Object Creation (1)

Index (4)

- OOP: Object Creation (2)**
- OOP: Object Creation (3)**
- OOP: Object Creation (4)**
- OOP: Object Creation (5)**
- OOP: Object Creation (6)**
- OOP: Mutator Methods**
- OOP: Accessor Methods**
- OOP: Use of Mutator vs. Accessor Methods**
- OOP: Method Parameters**
- OOP: Object Alias (1)**
- OOP: Object Alias (2.1)**
- OOP: Object Alias (2.2)**
- OOP: Object Alias (3)**
- Anonymous Objects (1)**

Index (5)

Anonymous Objects (2.1)

Anonymous Objects (2.2)

Java Data Types (1)

Java Data Types (2)

Java Data Types (3.1)

Java Data Types (3.2.1)

Java Data Types (3.2.2)

Static Variables (1)

Static Variables (2)

Static Variables (3)

Static Variables (4.1): Common Error

Static Variables (4.2): Common Error

Static Variables (5.1): Common Error

Static Variables (5.2): Common Error

Index (6)

Static Variables (5.3): Common Error