Classes and Objects

Readings: Chapters 3 – 4 of the Course Notes

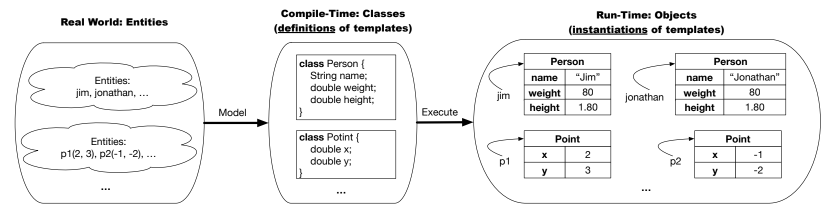
EECS2030: Advanced Object Oriented Programming
Fall 2017

CHEN-WEI WANG

Separation of Concerns: App vs. Model

- So far we have developed:
  Supplier: A single utility class.
  Client: A class with its `main` method using the utility methods.
- In Java:
  - We may define more than one (non-utility) *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



- 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)

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 [$\frac{80}{1.8^2}$]
  - `jonathan`'s BMI is based on his own height and weight [$\frac{90}{1.73^2}$]

OO Thinking: Templates vs. Instances (2.1)

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 (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 $(3, 4)$; `p1` 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.

```
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; } }
```

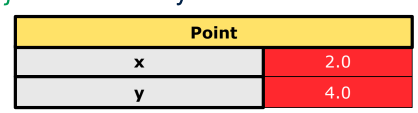
- In the corresponding tester class, each *call* to the `Point` constructor creates an instance of the `Point` template.

```
public class PersonTester {
  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); } }
```

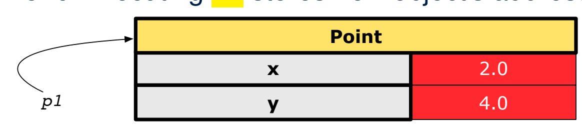

OOP: Define Constructors for Creating Objects (1.2)

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

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



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`.



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: 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); } }
```

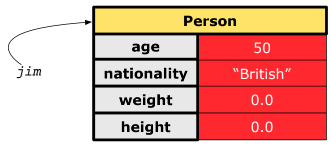
OOP: Define Constructors for Creating Objects (2.2)

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

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

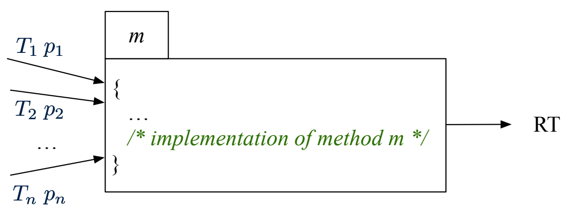
Person	
age	50
nationality	"British"
weight	0.0
height	0.0

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`.



OOP: Methods (1.1)

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



- The *Signature* 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 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: 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

```
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: 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)

```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 */
  }
}
```


OOP: Class Constructors (3)

```java
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 + ")");
```

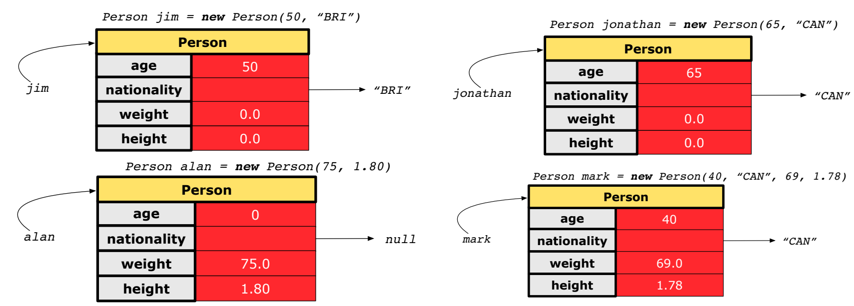
```
(2.0, 4.0)
```


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)

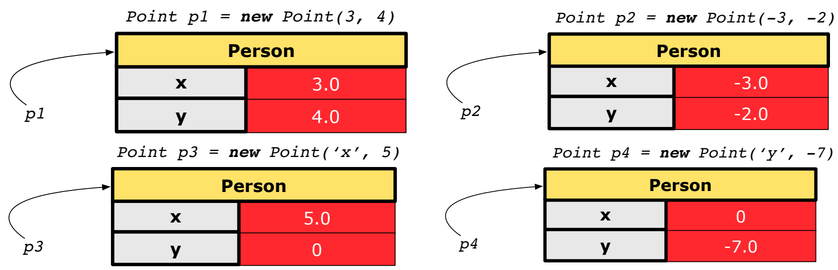


OOP: Object Creation (4)

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)



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)`

The `this` Reference (7.1): Exercise

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?

The `this` Reference (7.2): Exercise

```
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;
  }
}
```

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 be of type *ArrayList*, storing references to other objects.

```
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 *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 Data Types (3.2.2)

```
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) */
 }
}
```


Java Data Types (3.3.1)

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

```
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 Data Types (3.3.2)

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


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 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.

OO Program Programming: Object Alias (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]);
```


OO Program Programming: Object Alias (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.age);
System.out.println(alan.age);
System.out.println(persons2[0].age);
persons1[0] = jim;
persons1[0].setAge(75);
System.out.println(jim.age);
System.out.println(alan.age);
System.out.println(persons2[0].age);
```


Call by Value vs. Call by Reference (1)

- Consider the general form of a call to some ***mutator method*** `m`, with ***context object*** `co` and **argument value** `arg`:

  `co.m (arg)`

  - Argument variable **arg** is ***not*** passed directly for the method call.
  - Instead, argument variable **arg** is passed ***indirectly***: a *copy* of the value stored in **arg** is made and passed for the method call.
- What can be the type of variable **arg**? [Primitive or Reference]
  - **arg** is primitive type (e.g., `int`, `char`, `boolean`, *etc.*):
    Call by Value : Copy of **arg**'s *stored value* (e.g., `2`, `'j'`, `true`) is made and passed.
  - **arg** is reference type (e.g., `String`, `Point`, `Person`, *etc.*):
    Call by Reference : Copy of **arg**'s *stored reference/address* (e.g., `Point@5cb0d902`) is made and passed.

Call by Value vs. Call by Reference (2.1)

For illustration, let's assume the following variant of the `Point` class:

```
class Point {
  int x;
  int y;
  Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
  void moveVertically(int y){
    this.y += y;
  }
  void moveHorizontally(int x){
    this.x += x;
  }
}
```

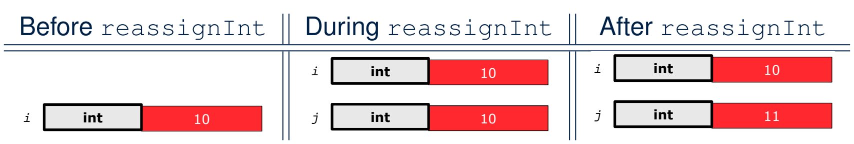
Call by Value vs. Call by Reference (2.2.1)

```
public class Util {
  void reassignInt(int j) {
    j = j + 1; }
  void reassignRef(Point q) {
    Point np = new Point(6, 8);
    q = np; }
  void changeViaRef(Point q) {
    q.moveHorizontally(3);
    q.moveVertically(4); } }
```

```
1  @Test
2  public void testCallByVal() {
3    Util u = new Util();
4    int i = 10;
5    assertTrue(i == 10);
6    u.reassignInt(i);
7    assertTrue(i == 10);
8  }
```

- ***Before*** the mutator call at **L6**, *primitive* variable `i` stores `10`.
- **When** executing the mutator call at **L6**, due to *call by value*, a copy of variable `i` is made.
  ⇒ The assignment `i = i + 1` is only effective on this copy, not the original variable `i` itself.
- ∴ ***After*** the mutator call at **L6**, variable `i` still stores `10`.

Call by Value vs. Call by Reference (2.2.2)



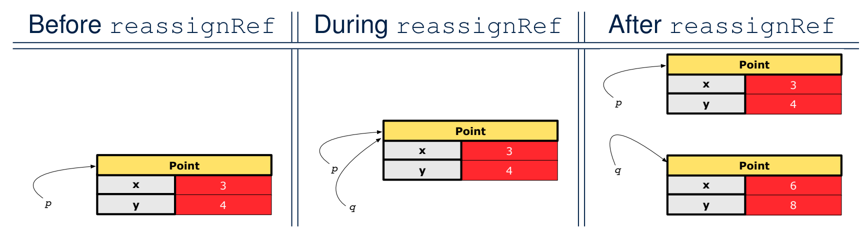
Call by Value vs. Call by Reference (2.3.1)

```
public class Util {
  void reassignInt(int j) {
    j = j + 1; }
  void reassignRef(Point q) {
    Point np = new Point(6, 8);
    q = np; }
  void changeViaRef(Point q) {
    q.moveHorizontally(3);
    q.moveVertically(4); } }
```

```
1  @Test
2  public void testCallByRef_1() {
3    Util u = new Util();
4    Point p = new Point(3, 4);
5    Point refOfPBefore = p;
6    u.reassignRef(p);
7    assertTrue(p==refOfPBefore);
8    assertTrue(p.x==3 && p.y==4);
9  }
```

- ***Before*** the mutator call at **L6**, *reference* variable `p` stores the *address* of some `Point` object (whose `x` is `3` and `y` is `4`).
- **When** executing the mutator call at **L6**, due to *call by reference*, a *copy of address* stored in `p` is made.
  ⇒ The assignment `p = np` is only effective on this copy, not the original variable `p` itself.
- ∴ ***After*** the mutator call at **L6**, variable `p` still stores the original address (i.e., same as `refOfPBefore`).

Call by Value vs. Call by Reference (2.3.2)



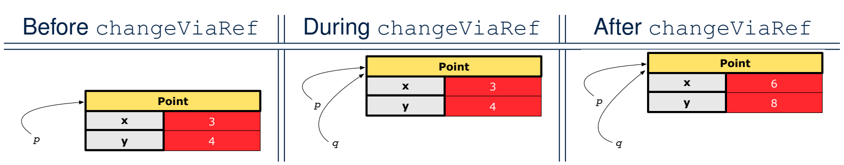
Call by Value vs. Call by Reference (2.4.1)

```java
public class Util {
  void reassignInt(int j) {
    j = j + 1; }
  void reassignRef(Point q) {
    Point np = new Point(6, 8);
    q = np; }
  void changeViaRef(Point q) {
    q.moveHorizontally(3);
    q.moveVertically(4); } }
```

```java
1 @Test
2 public void testCallByRef_2() {
3   Util u = new Util();
4   Point p = new Point(3, 4);
5   Point refOfPBefore = p;
6   u.changeViaRef(p);
7   assertTrue(p==refOfPBefore);
8   assertTrue(p.x==6 && p.y==8);
9 }
```

- ***Before*** the mutator call at **L6**, *reference* variable `p` stores the *address* of some `Point` object (whose `x` is `3` and `y` is `4`).
- **When** executing the mutator call at **L6**, due to *call by reference*, a *copy of address* stored in `p` is made. **[Alias:** `p` and `q` store same address.**]**
  ⇒ Calls to `q.moveHorizontally` and `q.moveVertically` are effective on both `p` and `q`.
- ∴ ***After*** the mutator call at **L6**, variable `p` still stores the original address (i.e., same as `refOfPBefore`), but its `x` and `y` have been modified via `q`.

Call by Value vs. Call by Reference (2.4.2)



Aggregation vs. Composition: Terminology

Container object: an object that contains others.

Containee object: an object that is contained within another.

- e.g., Each course has a faculty member as its instructor.
  - ***Container***: `Course` ***Containee***: `Faculty`.
- e.g., Each student is registered in a list of courses; Each faculty member teaches a list of courses.
  - ***Container***: `Student`, `Faculty` ***Containees***: `Course`.

    e.g., `eecs2030` taken by `jim` (student) and taught by `tom` (faculty).

    ⇒ ***Containees*** **may** be *shared* by different classes of ***containers***.

    e.g., When EECS2030 is finished, `jim` and `jackie` still exist!

    ⇒ ***Containees*** **may** exist *independently* without their ***containers***.
- e.g., In a file system, each directory contains a list of files.
  - ***Container***: `Directory` ***Containees***: `File`.

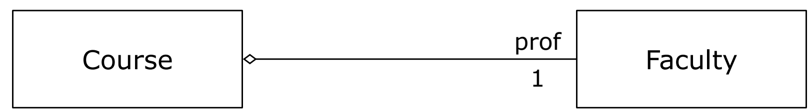
    e.g., Each file has exactly one parent directory.

    ⇒ A ***containee*** **may** be *owned* by only one ***container***.

    e.g., Deleting a directory also deletes the files it contains.

    ⇒ ***Containees*** **may** *co-exist* with their ***containers***.

Aggregation: Independent Containees Shared by Containers (1.1)




```java
class Course {
  String title;
  Faculty prof;
  Course(String title) {
    this.title = title;
  }
  void setProf(Faculty prof) {
    this.prof = prof;
  }
  Faculty getProf() {
    return this.prof;
  }
}
```

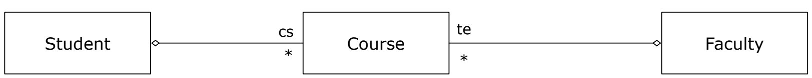
```java
class Faculty {
  String name;
  Faculty(String name) {
    this.name = name;
  }
  void setName(String name) {
    this.name = name;
  }
  String getName() {
    return this.name;
  }
}
```

Aggregation: Independent Containees Shared by Containers (1.2)

```java
@Test
public void testAggregation1() {
  Course eecs2030 = new Course("Advanced OOP");
  Course eecs3311 = new Course("Software Design");
  Faculty prof = new Faculty("Jackie");
  eecs2030.setProf(prof);
  eecs3311.setProf(prof);
  assertTrue(eecs2030.getProf() == eecs3311.getProf());
  /* aliasing */
  prof.setName("Jeff");
  assertTrue(eecs2030.getProf() == eecs3311.getProf());
  assertTrue(eecs2030.getProf().getName().equals("Jeff"));

  Faculty prof2 = new Faculty("Jonathan");
  eecs3311.setProf(prof2);
  assertTrue(eecs2030.getProf() != eecs3311.getProf());
  assertTrue(eecs2030.getProf().getName().equals("Jeff"));
  assertTrue(eecs3311.getProf().getName().equals("Jonathan"));
}
```


Aggregation: Independent Containees Shared by Containers (2.1)




```java
class Student {
  String id; ArrayList<Course> cs; /* courses */
  Student(String id) { this.id = id; cs = new ArrayList<>(); }
  void addCourse(Course c) { cs.add(c); }
  ArrayList<Course> getCS() { return cs; }
}
```

```java
class Course { String title; }
```

```java
class Faculty {
  String name; ArrayList<Course> te; /* teaching */
  Faculty(String name) { this.name = name; te = new ArrayList<>(); }
  void addTeaching(Course c) { te.add(c); }
  ArrayList<Course> getTE() { return te; }
}
```

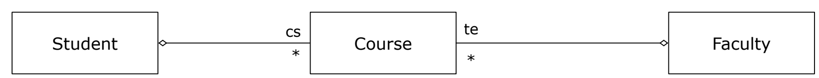

Aggregation: Independent Containees Shared by Containers (2.2)

```java
@Test
public void testAggregation2() {
  Faculty p = new Faculty("Jackie");
  Student s = new Student("Jim");
  Course eecs2030 = new Course("Advanced OOP");
  Course eecs3311 = new Course("Software Design");
  eecs2030.setProf(p);
  eecs3311.setProf(p);
  p.addTeaching(eecs2030);
  p.addTeaching(eecs3311);
  s.addCourse(eecs2030);
  s.addCourse(eecs3311);

  assertTrue(eecs2030.getProf() == s.getCS().get(0).getProf());
  assertTrue(s.getCS().get(0).getProf() == s.getCS().get(1).getProf());
  assertTrue(eecs3311 == s.getCS().get(1));
  assertTrue(s.getCS().get(1) == p.getTE().get(1));
}
```


OOP: The Dot Notation (3.1)

In real life, the relationships among classes are sophisticated.




```java
class Student {
  String id;
  ArrayList<Course> cs;
}
```

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

```java
class Faculty {
  String name;
  ArrayList<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 list 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;
  ArrayList<Course> cs;
}
```

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

```
class Faculty {
  String name;
  ArrayList<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.get(i).title;
  }
  /* Get the instructor's name of the ith course */
  String getInstructorName(int i) {
    return this.cs.get(i).prof.name;
  }
}
```


OOP: The Dot Notation (3.3)

```
class Student {
  String id;
  ArrayList<Course> cs;
}
```

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

```
class Faculty {
  String name;
  ArrayList<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.get(i).title;
  }
}
```


OOP: The Dot Notation (3.4)

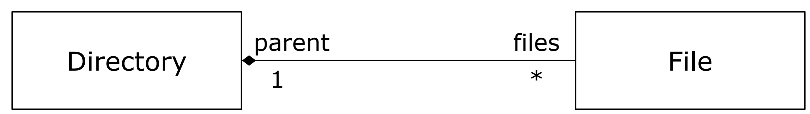
```
class Student {
  String id;
  ArrayList<Course> cs;
}
```

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

```
class Faculty {
  String name;
  ArrayList<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.get(i).title;
  }
}
```


Composition: Dependent Containees Owned by Containers (1.1)

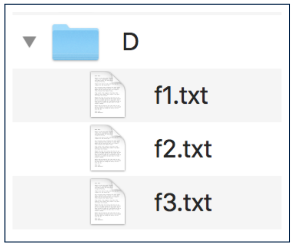


Assumption: Files are not shared among directories.

```
class File {
  String name;
  File(String name) {
    this.name = name;
  }
}
```

```
class Directory {
  String name;
  File[] files;
  int nof; /* num of files */
  Directory(String name) {
    this.name = name;
    files = new File[100];
  }
  void addFile(String fileName) {
    files[nof] = new File(fileName);
    nof ++;
  }
}
```

Composition: Dependent Containees Owned by Containers (1.2.1)

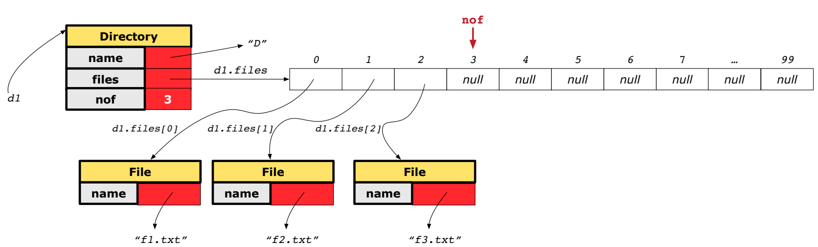



```
1 @Test
2 public void testComposition() {
3   Directory d1 = new Directory("D");
4   d1.addFile("f1.txt");
5   d1.addFile("f2.txt");
6   d1.addFile("f3.txt");
7   assertTrue(
8     d1.files[0].name.equals("f1.txt"));
9 }
```

- **L4**: a 1st `File` object is created and ***owned exclusively*** by `d1`. No other directories are sharing this `File` object with `d1`.
- **L5**: a 2nd `File` object is created and ***owned exclusively*** by `d1`. No other directories are sharing this `File` object with `d1`.
- **L6**: a 3rd `File` object is created and ***owned exclusively*** by `d1`.

Composition: Dependent Containees Owned by Containers (1.2.2)

Right before test method `testComposition` terminates:



Composition: Dependent Containees Owned by Containers (1.3)

Problem: How do you implement a *copy instructor* for the `Directory` class?

```
class Directory {
  Directory(Directory other) {
    /* ?? */
  }
}
```

Hints:

- The implementation should be consistent with the effect of copying and pasting a directory.
- Separate copies of files are created.

Composition: Dependent Containees Owned by Containers (1.4.1)

Version 1: *Shallow Copy* by copying all attributes using =.

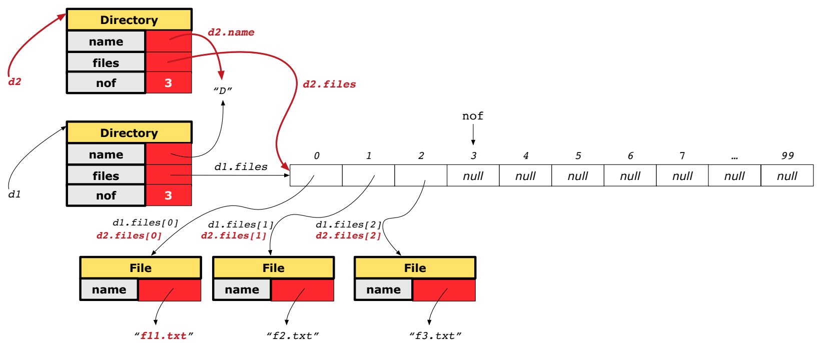
```
class Directory {
  Directory(Directory other) {
    /* value copying for primitive type */
    nof = other.nof;
    /* address copying for reference type */
    name = other.name; files = other.files; } }
```

Is a shallow copy satisfactory to support composition?
i.e., Does it still forbid sharing to occur? [***NO***]

```
@Test
void testShallowCopyConstructor() {
  Directory d1 = new Directory("D");
  d1.addFile("f1.txt"); d1.addFile("f2.txt"); d1.addFile("f3.txt");
  Directory d2 = new Directory(d1);
  assertTrue(d1.files == d2.files); /* violation of composition */
  d2.files[0].changeName("f11.txt");
  assertFalse(d1.files[0].name.equals("f1.txt")); }
```

Composition: Dependent Containees Owned by Containers (1.4.2)

Right before test method `testShallowCopyConstructor` terminates:



Composition: Dependent Containees Owned by Containers (1.5.1)

Version 2: a *Deep Copy*

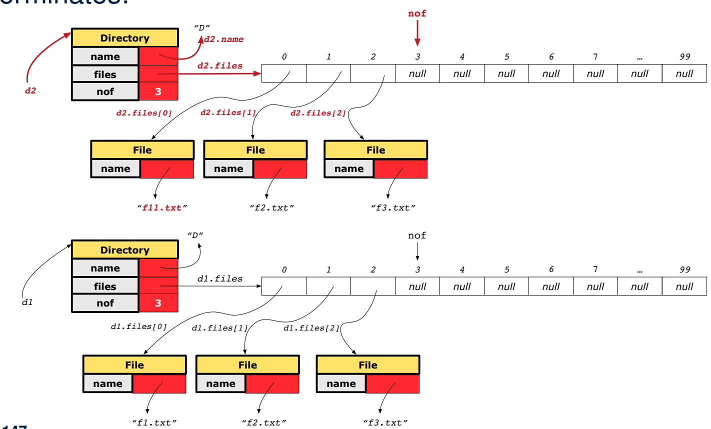
```
class File {
  File(File other) {
    this.name =
      new String(other.name);
  }
}
```

```
class Directory {
  Directory(String name) {
    this.name = new String(name);
    files = new File[100]; }
  Directory(Directory other) {
    this (other.name);
    for(int i = 0; i < nof; i ++) {
      File src = other.files[i];
      File nf = new File(src);
      this.addFile(nf); } } }
```

```
@Test
void testDeepCopyConstructor() {
  Directory d1 = new Directory("D");
  d1.addFile("f1.txt"); d1.addFile("f2.txt"); d1.addFile("f3.txt");
  Directory d2 = new Directory(d1);
  assertTrue(d1.files != d2.files); /* composition preserved */
  d2.files[0].changeName("f11.txt");
  assertTrue(d1.files[0].name.equals("f1.txt")); }
```


Composition: Dependent Containees Owned by Containers (1.5.2)

Right before test method `testDeepCopyConstructor` terminates:



Composition: Dependent Containees Owned by Containers (1.6)

Exercise: Implement the accessor in class `Directory`

```
class Directory {
  File[] files;
  int nof;
  File[] getFiles() {
    /* Your Task */
  }
}
```

so that it ***preserves composition***, i.e., does not allow references of files to be shared.

Aggregation vs. Composition (1)

Terminology:

- **Container** object: an object that contains others.
- **Containee** object: an object that is contained within another.

Aggregation :

- Containees (e.g., `Course`) may be *shared* among containers (e.g., `Student`, `Faculty`).
- Containees *exist independently* without their containers.
- When a container is destroyed, its containees still exist.

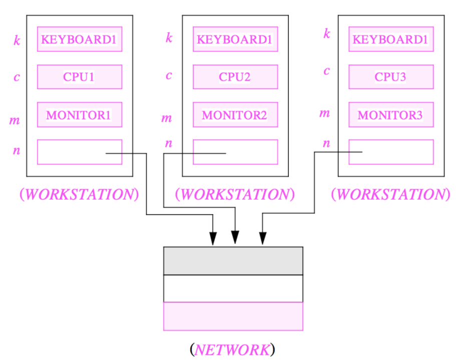
Composition :

- Containers (e.g, `Directory`, `Department`) *own* exclusive access to their containees (e.g., `File`, `Faculty`).
- Containees cannot exist without their containers.
- Destroying a container destroys its containeees *cascadingly*.

Aggregation vs. Composition (2)

Aggregations and *Compositions* may exist at the same time!
e.g., Consider a workstation:

- Each workstation owns CPU, monitor, keyword. [***compositions***]
- All workstations share the same network. [***aggregations***]



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*.

OOP: Equality (3)

- Implicitly:
  - Every class is a *child/sub* class of the `Object` class.
  - The `Object` class is the *parent/super* class of every class.
- There are two useful *accessor methods* that every class *inherits* from the `Object` class:
  - `boolean equals(Object other)`
    Indicates whether some other object is "equal to" this one.
    - The default definition inherited from `Object`:

```
boolean equals(Object other) {
  return (this == other); }
```

  - `String toString()`
    Returns a string representation of the object.
- Very often when you define new classes, you want to *redefine* / *override* the inherited definitions of `equals` and `toString`.

OOP: Contract of `equals`

Given that reference variables `x`, `y`, `z` are not `null`:

- $$\neg\, x.equals(null)$$
- *Reflexive*:
  $$x.equals(x)$$
- *Symmetric*
  $$x.equals(y) \iff y.equals(x)$$
- *Transitive*
  $$x.equals(y) \wedge y.equals(z) \Rightarrow x.equals(z)$$

API of `equals` Inappropriate Def. of `equals` using `hashCode`

OOP: Equality (4.1)

- How do we compare *contents* rather than addresses?
- Define the *accessor method* `equals`, e.g.,

```
class Point {
  double x; double y;
  boolean equals (Object obj) {
    if(this == obj) { return true; }
    if(obj == null) { return false; }
    if(this.getClass() != obj.getClass()) { return false; }
    Point other = (Point) obj;
    return this.x == other.x && this.y == other.y; } }
```

```
class PointTester {
  String s = "(2, 3)";
  Point p1 = new Point(2, 3); Point p2 = new Point(2, 3);
  System.out.println(p1. equals (p1)); /* true */
  System.out.println(p1.equals(null)); /* false */
  System.out.println(p1.equals(s)); /* false */
  System.out.println(p1 == p2); /* false */
  System.out.println(p1. equals (p2)); /* true */ }
```


OOP: Equality (4.2)

- When making a method call `p.equals(o)`:
  - Variable `p` is of type `Point`
  - Variable `o` can be any type
- We define `p` and `o` as ***equal*** if:
  - Either `p` and `o` refer to the same object;
  - Or:
    - `o` is not null.
    - `p` and `o` are of the same type.
    - The `x` and `y` coordinates are the same.
- **Q**: In the `equals` method of `Point`, why is there no such a line:

```
class Point {
  boolean equals (Object obj) {
    if(this == null) { return false; }
```

A: If `this` is `null`, a `NullPointerException` would have occurred and prevent the body of `equals` from being executed.

OOP: Equality (4.3)

```
1  class Point {
2   boolean equals (Object obj) {
3     ...
4     Point other = (Point) obj;
5     return this.x == other.x && this.y == other.y; } }
```

- `Object obj` at **L2** declares a parameter `obj` of type `Object`.
- `Point p` at **L4** declares a variable `p` of type `Point`.
  We call such types declared at compile time as ***static type***.
- The list of applicable methods that we may call on a variable depends on its ***static type***.
  e.g., We may only call the small list of methods defined in `Object` class on `obj`, which does not include `x` and `y` (specific to `Point`).
- If we are SURE that an object's "actual" type is different from its ***static type***, then we can *cast* it.
  e.g., Given that `this.getClass() == obj.getClass()`, we are sure that `obj` is also a `Point`, so we can cast it to `Point`.
- Such cast allows more attributes/methods to be called upon `(Point) obj` at **L5**.

OOP: Equality (5.1)

Exercise: Persons are *equal* if names and measures are equal.

```
1  class Person {
2   String firstName; String lastName; double weight; double height;
3   boolean equals (Object obj) {
4     if(this == obj) { return true }
5     if(obj == null || this.getClass() != obj.getClass()) {
6       return false; }
7     Person other = (Person) obj;
8     return
9         this.weight == other.weight && this.height == other.height
10      && this.firstName. equals (other.firstName)
11      && this.lastName. equals (other.lastName) } }
```

Q: At **L5**, will we get `NullPointerException` if `obj` is `Null`?
A: *No* ∵ Short-Circuit Effect of `||`
`obj` is `null`, then `obj == null` evaluates to ***true***
⇒ no need to evaluate the RHS

The left operand `obj == null` acts as a *guard constraint* for the right operand `this.getClass() != obj.getClass()`.

OOP: Equality (5.2)

Exercise: Persons are *equal* if names and measures are equal.

```
1  class Person {
2   String firstName; String lastName; double weight; double height;
3   boolean equals (Object obj) {
4     if(this == obj) { return true }
5     if(obj == null || this.getClass() != obj.getClass()) {
6       return false; }
7     Person other = (Person) obj;
8     return
9         this.weight == other.weight && this.height == other.height
10      && this.firstName. equals (other.firstName)
11      && this.lastName. equals (other.lastName) } }
```

Q: At **L5**, if swapping the order of two operands of disjunction:
`this.getClass() != obj.getClass() || obj == null`
Will we get `NullPointerException` if `obj` is `Null`?
A: ***Yes*** ∵ Evaluation of operands is from left to right.

OOP: Equality (5.3)

Exercise: Persons are *equal* if names and measures are equal.

```
1  class Person {
2   String firstName; String lastName; double weight; double height;
3   boolean equals (Object obj) {
4     if(this == obj) { return true }
5     if(obj == null || this.getClass() != obj.getClass()) {
6       return false; }
7     Person other = (Person) obj;
8     return
9         this.weight == other.weight && this.height == other.height
10      && this.firstName. equals (other.firstName)
11      && this.lastName. equals (other.lastName) } }
```

L10 & L11 call `equals` method defined in the `String` class.

When defining `equals` method for your own class, *reuse* `equals` methods defined in other classes wherever possible.

OOP: Equality (6)

Two notions of **equality** for variables of *reference* types:

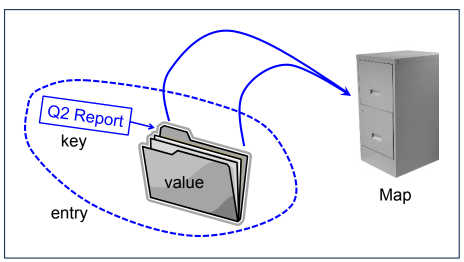
- **Reference Equality**: use == to compare *addresses*
- **Object Equality**: define `equals` method to compare *contents*

```
Point p1 = new Point(3, 4);
Point p2 = new Point(3, 4);
Point p3 = new Point(4, 5);
System.out.println(p1 == p1);  /* true */
System.out.println(p1.equals(p1));  /* true */
System.out.println(p1 == p2);  /* false */
System.out.println(p1.equals(p2));  /* true */
System.out.println(p2 == p3);  /* false */
System.out.println(p2.equals(p3));  /* false */
```

- Being *reference*-equal implies being *object*-equal
- Being *object*-equal does **not** imply being *reference*-equal

Hashing: What is a Map?

- A **map** (a.k.a. table or dictionary) stores a collection of *entries*.



ENTRY	
(SEARCH) KEY	VALUE
1	D
25	C
3	F
14	Z
6	A
39	C
7	Q

- Each **entry** is a pair: a *value* and its *(search) key*.
- Each **search key**:
  - *Uniquely* identifies an object in the map
  - Should be used to *efficiently* retrieve the associated value
- Search keys must be *unique* (i.e., do not contain duplicates).

Hashing: Arrays are Maps

- Each array *entry* is a pair: an object and its *numerical* index.
  e.g., say `String[] a = {"A", "B", "C"}`, how many entries?
  3 entries: `(0, "A")`, `(1, "B")`, `(2, "C")`
- *Search keys* are the set of numerical index values.
- The set of index values are *unique* [e.g., $0 .. (a.length - 1)$]
- Given a **valid** index value i, we can
  - *Uniquely* determines where the object is [$(i + 1)^{th}$ item]
  - *Efficiently* retrieves that object [`a[i]` ≈ fast memory access]
- Maps in general may have **non-numerical** key values:
  - Student ID [student record]
  - Social Security Number [resident record]
  - Passport Number [citizen record]
  - Residential Address [household record]
  - Media Access Control (MAC) Address [PC/Laptop record]
  - Web URL [web page]

  ...

Hashing: Naive Implementation of Map

- **Problem**: Support the construction of this simple map:

ENTRY	
(SEARCH) KEY	VALUE
1	D
25	C
3	F
14	Z
6	A
39	C
7	Q

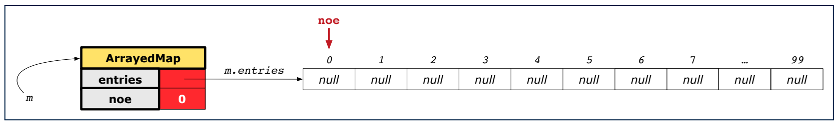
  Let's just assume that the maximum map capacity is 100.
- **Naive Solution**:

  Let's understand the expected runtime structures before seeing the Java code!

Hashing: Naive Implementation of Map (0)

After executing `ArrayedMap m = new ArrayedMap()`:

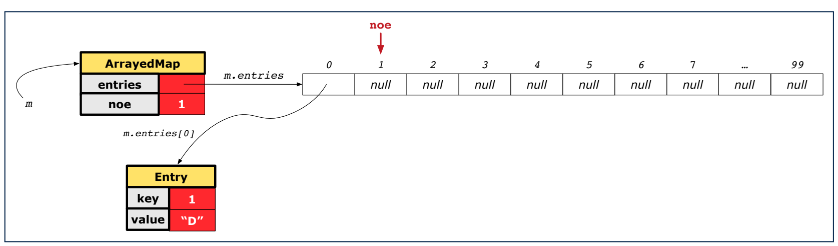
- Attribute `m.entries` initialized as an array of 100 `null` slots.
- Attribute `m.noe` is 0, meaning:
  - Current number of entries stored in the map is 0.
  - Index for storing the next new entry is 0.



Hashing: Naive Implementation of Map (1)

After executing `m.put(new Entry(1, "D"))`:

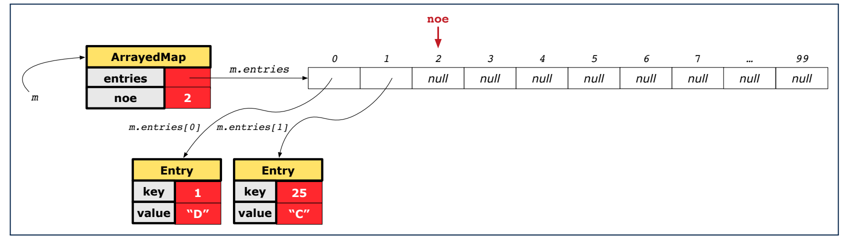
- Attribute `m.entries` has 99 `null` slots.
- Attribute `m.noe` is 1, meaning:
  - Current number of entries stored in the map is 1.
  - Index for storing the next new entry is 1.



Hashing: Naive Implementation of Map (2)

After executing `m.put(new Entry(25, "C"))`:

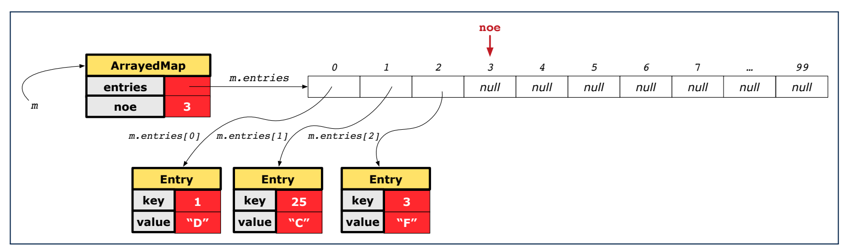
- Attribute `m.entries` has 98 `null` slots.
- Attribute `m.noe` is 2, meaning:
  - Current number of entries stored in the map is 2.
  - Index for storing the next new entry is 2.



Hashing: Naive Implementation of Map (3)

After executing `m.put(new Entry(3, "F"))`:

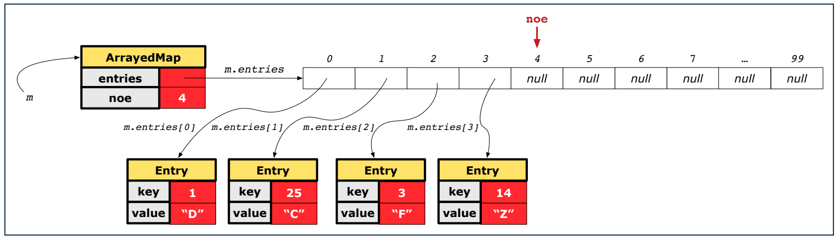
- Attribute `m.entries` has 97 `null` slots.
- Attribute `m.noe` is 3, meaning:
  - Current number of entries stored in the map is 3.
  - Index for storing the next new entry is 3.



Hashing: Naive Implementation of Map (4)

After executing `m.put(new Entry(14, "Z"))`:

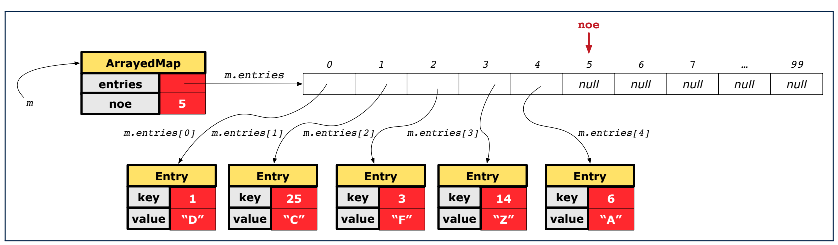
- Attribute `m.entries` has 96 `null` slots.
- Attribute `m.noe` is 4, meaning:
  - Current number of entries stored in the map is 4.
  - Index for storing the next new entry is 4.



Hashing: Naive Implementation of Map (5)

After executing `m.put(new Entry(6, "A"))`:

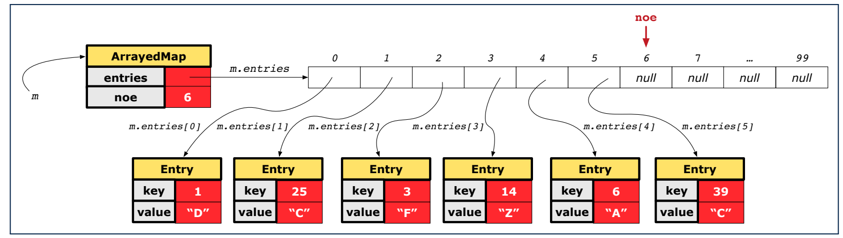
- Attribute `m.entries` has 95 `null` slots.
- Attribute `m.noe` is 5, meaning:
  - Current number of entries stored in the map is 5.
  - Index for storing the next new entry is 5.



Hashing: Naive Implementation of Map (6)

After executing `m.put(new Entry(39, "C"))`:

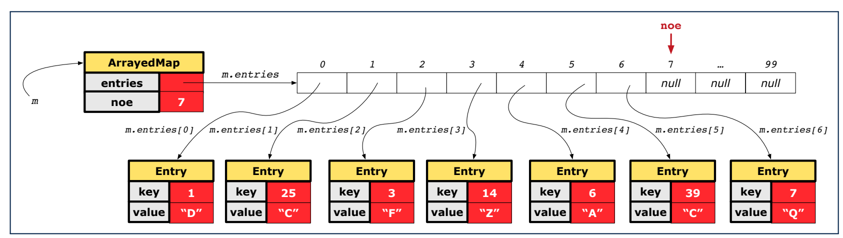
- Attribute `m.entries` has 94 `null` slots.
- Attribute `m.noe` is 6, meaning:
  - Current number of entries stored in the map is 6.
  - Index for storing the next new entry is 6.



Hashing: Naive Implementation of Map (7)

After executing `m.put(new Entry(7, "Q"))`:

- Attribute `m.entries` has 93 `null` slots.
- Attribute `m.noe` is 7, meaning:
  - Current number of entries stored in the map is 7.
  - Index for storing the next new entry is 7.



Hashing: Naive Implementation of Map (8.1)

```
public class Entry {
  private int key;
  private String value;

  public Entry(int key, String value) {
    this.key = key;
    this.value = value;
  }
  /* Getters and Setters for key and value */
}
```


Hashing: Naive Implementation of Map (8.2)

```
public class ArrayedMap {
  private final int MAX_CAPCAITY = 100;
  private Entry[] entries;
  private int noe; /* number of entries */
  public ArrayedMap() {
    entries = new Entry[MAX_CAPCAITY];
    noe = 0;
  }
  public int size() {
    return noe;
  }
  public void put(int key, String value) {
    Entry e = new Entry(key, value);
    entries[noe] = e;
    noe ++;
  }
```

Required Reading: Point and PointCollector

Hashing: Naive Implementation of Map (8.3)

```
@Test
public void testArrayedMap() {
  ArrayedMap m = new ArrayedMap();
  assertTrue(m.size() == 0);
  m.put(1, "D");
  m.put(25, "C");
  m.put(3, "F");
  m.put(14, "Z");
  m.put(6, "A");
  m.put(39, "C");
  m.put(7, "Q");
  assertTrue(m.size() == 7);
  /* inquiries of existing key */
  assertTrue(m.get(1).equals("D"));
  assertTrue(m.get(7).equals("Q"));
  /* inquiry of non-existing key */
  assertTrue(m.get(31) == null);
}
```


Hashing: Naive Implementation of Map (8.4)

```
public class ArrayedMap {
  private final int MAX_CAPCAITY = 100;
  public String getValue (int key) {
    for(int i = 0; i < noe; i ++) {
      Entry e = entries[i];
      int k = e.getKey();
      if(k == key) { return e.getValue(); }
    }
    return null;
  }
```

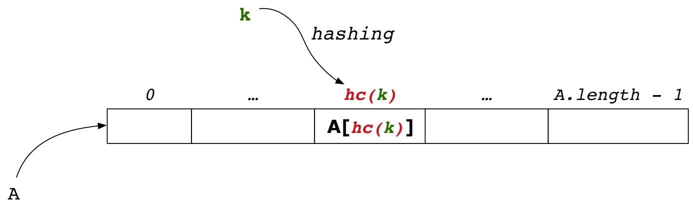
Say `entries` is: {(1, D), (25, C), (3, F), (14, Z), (6, A), (39, C), (7, Q), null, ...}

- How efficient is `m.get(1)`? [1 iteration]
- How efficient is `m.get(7)`? [7 iterations]
- If `m` is full, worst case of `m.get(k)`? [100 iterations]
- If `m` with 10^6 entries, worst case of `m.get(k)`? [10^6 iterations]

  ⇒ `get`'s worst-case performance is *linear* on size of `m.entries`!

A much *faster* (and ***correct***) solution is possible!

Hashing: Hash Table (1)

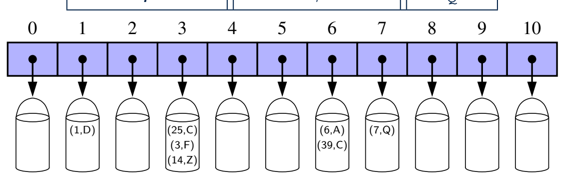


- Given a (numerical or non-numerical) search key ***k***:
  - Apply a function ***hc*** so that ***hc(k)*** returns an integer.
    - We call ***hc(k)*** the *hash code* of key ***k***.
    - Value of ***hc(k)*** denotes a ***valid index*** of some array `A`.
  - Rather than searching through array `A`, go directly to `A[` ***hc(k)*** `]` to get the associated value.
- Both computations are fast:
  - Converting ***k*** to ***hc(k)***
  - Indexing into `A[` ***hc(k)*** `]`

Hashing: Hash Table as a Bucket Array (2)

For illustration, assume `A.length` is 10 and $hc(k) = k\%11$.

$hc(k) = k\%11$	(SEARCH) KEY	VALUE
1	1	D
3	25	C
3	3	F
3	14	Z
6	6	A
6	39	C
7	7	Q



- **Collision**: unequal keys have same hash code (e.g., 25, 3, 14)
  ⇒ Unavoidable as number of entries ↑, but a *good* hash function should have sizes of the buckets uniformly distributed.

Hashing: Contract of Hash Function

- Principle of defining a hash function ***hc***:

$$k1.equals(k2) \Rightarrow hc(k1) == hc(k2)$$

  Equal keys always have the same hash code.
- Equivalently, according to contrapositive:

$$hc(k1) \neq hc(k2) \Rightarrow \neg k1.equals(k2)$$

  Different hash codes must be generated from unequal keys.

inconsistent `hashCode` and `equals`

Hashing: Defining Hash Function in Java (1)

The `Object` class (common super class of all classes) has the method for redefining the hash function for your own class:

```
public class IntegerKey {
  private int k;
  public IntegerKey(int k) { this.k = k; }
  @Override
  public int hashCode() { return k % 11; }
  @Override
  public boolean equals(Object obj) {
    if(this == obj) { return true; }
    if(obj == null) { return false; }
    if(this.getClass() != obj.getClass()) { return false; }
    IntegerKey other = (IntegerKey) obj;
    return this.k == other.k;
  } }
```

Q: Can we define `equals` as `return this.hashCode == other.hashCode()`? [*No* ∵ Collision; see contract of `equals`]

Hashing: Defining Hash Function in Java (2)

```java
@Test
public void testCustomizedHashFunction() {
  IntegerKey ik1 = new IntegerKey(1);
  /* 1 % 11 == 1 */
  assertTrue(ik1.hashCode() == 1);

  IntegerKey ik39_1 = new IntegerKey(39);
  /* 39 % 11 == 3 */
  assertTrue(ik39_1.hashCode() == 6);

  IntegerKey ik39_2 = new IntegerKey(39);
  assertTrue(ik39_1.equals(ik39_2));
  assertTrue(ik39_1.hashCode() == ik39_2.hashCode());
}
```


Hashing: Using Hash Table in Java

```java
@Test
public void testHashTable() {
  Hashtable<IntegerKey, String> table = new Hashtable<>();
  IntegerKey k1 = new IntegerKey(39);
  IntegerKey k2 = new IntegerKey(39);
  assertTrue(k1.equals(k2));
  assertTrue(k1.hashCode() == k2.hashCode());
  table.put(k1, "D");
  assertTrue(table.get(k2).equals("D"));
}
```


Hashing: Defining Hash Function in Java (3)

- When you are given instructions as to how the `hashCode` method of a class should be defined, override it manually.
- Otherwise, use Eclipse to generate the `equals` and `hashCode` methods for you.
  - Right click on the class.
  - Select `Source`.
  - Select `Generate hashCode() and equals()`.
  - Select the relevant attributes that will be used to compute the hash value.

Hashing: Defining Hash Function in Java (4.1)

Caveat: Always make sure that the `hashCode` and `equals` are redefined/overridden to work together consistently.
e.g., Consider an alternative version of the `IntegerKey` class:

```java
public class IntegerKey {
  private int k;
  public IntegerKey(int k) { this.k = k; }
  /* hashCode() inherited from Object NOT overridden. */
  @Override
  public boolean equals(Object obj) {
    if(this == obj) { return true; }
    if(obj == null) { return false; }
    if(this.getClass() != obj.getClass()) { return false; }
    IntegerKey other = (IntegerKey) obj;
    return this.k == other.k;
  } }
```

Problem? [**Hint**: Contract of `hashCode()`]

Hashing: Defining Hash Function in Java (4.2)

```
1  @Test
2  public void testDefaultHashFunction() {
3    IntegerKey ik39_1 = new IntegerKey(39);
4    IntegerKey ik39_2 = new IntegerKey(39);
5    assertTrue(ik39_1.equals(ik39_2));
6    assertTrue(ik39_1.hashCode() != ik39_2.hashCode()); }
7  @Test
8  public void testHashTable() {
9    Hashtable<IntegerKey, String> table = new Hashtable<>();
10   IntegerKey k1 = new IntegerKey(39);
11   IntegerKey k2 = new IntegerKey(39);
12   assertTrue(k1.equals(k2));
13   assertTrue(k1.hashCode() != k2.hashCode());
14   table.put(k1, "D");
15   assertTrue(table.get(k2) == null); }
```

L3, 4, 11, 12: Default version of `hashCode`, inherited from `Object`, returns a *distinct* integer for every new object, *despite its contents*. [***Fix***: Override `hashCode` of your classes!]

Why Ordering Between Objects? (1)

Each employee has their numerical id and salary.

e.g., (*alan*, 2, 4500.34), (*mark*, 3, 3450.67), (*tom*, 1, 3450.67)

- ***Problem***: To facilitate an annual review on their statuses, we want to arrange them so that ones with smaller id's come before ones with larger id's.s

  e.g., ⟨*tom*, *alan*, *mark*⟩
- Even better, arrange them so that ones with larger salaries come first; only compare id's for employees with equal salaries.

  e.g., ⟨*alan*, *tom*, *mark*⟩
- *Solution*:
  - Define ***ordering*** of `Employee` objects. [`Comparable` interface, `compareTo` method]
  - Use the library method `Arrays.sort`.

Why Ordering Between Objects? (2)

```
class Employee {
  int id; double salary;
  Employee(int id) { this.id = id; }
  void setSalary(double salary) { this.salary = salary; } }
```

```
1  @Test
2  public void testUncomparableEmployees() {
3    Employee alan = new Employee(2);
4    Employee mark = new Employee(3);
5    Employee tom = new Employee(1);
6    Employee[] es = {alan, mark, tom};
7    Arrays.sort(es);
8    Employee[] expected = {tom, alan, mark};
9    assertArrayEquals(expected, es); }
```

L8 triggers a ***java.lang.ClassCastException***:
Employee cannot be cast to java.lang.Comparable

∵ `Arrays.sort` expects an array whose element type defines a precise ***ordering*** of its instances/objects.

Defining Ordering Between Objects (1.1)

```
class CEmployee1 implements Comparable<CEmployee1> {
  ... /* attributes, constructor, mutator similar to Employee */
  @Override
  public int compareTo(CEmployee1 e) { return this.id - e.id; }
}
```

- Given two `CEmployee1` objects `ce1` and `ce2`:
  - `ce1.compareTo(ce2) > 0` [`ce1` "is greater than" `ce2`]
  - `ce1.compareTo(ce2) == 0` [`ce1` "is equal to" `ce2`]
  - `ce1.compareTo(ce2) < 0` [`ce1` "is smaller than" `ce2`]
- Say `ces` is an array of `CEmployee1` (`CEmployee1[] ces`), calling `Arrays.sort(ces)` re-arranges `ces`, so that:

$$\underbrace{\texttt{ces[0]}}_{\texttt{CEmployee1}\ \textit{object}} \leq \underbrace{\texttt{ces[1]}}_{\texttt{CEmployee1}\ \textit{object}} \leq \ldots \leq \underbrace{\texttt{ces[ces.length - 1]}}_{\texttt{CEmployee1}\ \textit{object}}$$

Defining Ordering Between Objects (1.2)

```
@Test
public void testComparableEmployees_1() {
  /*
   * CEmployee1 implements the Comparable interface.
   * Method compareTo compares id's only.
   */
  CEmployee1 alan = new CEmployee1(2);
  CEmployee1 mark = new CEmployee1(3);
  CEmployee1 tom = new CEmployee1(1);
  alan.setSalary(4500.34);
  mark.setSalary(3450.67);
  tom.setSalary(3450.67);
  CEmployee1[] es = {alan, mark, tom};
  /* When comparing employees,
   * their salaries are irrelevant.
   */
  Arrays.sort(es);
  CEmployee1[] expected = {tom, alan, mark};
  assertArrayEquals(expected, es);
}
```


Defining Ordering Between Objects (2.1)

Let's now make the comparison more sophisticated:

- Employees with higher salaries come before those with lower salaries.
- When two employees have same salary, whoever with lower id comes first.

```
1 class CEmployee2 implements Comparable<CEmployee2> {
2   ... /* attributes, constructor, mutator similar to Employee */
3   @Override
4   public int compareTo(CEmployee2 other) {
5     int salaryDiff = Double.compare(this.salary, other.salary);
6     int idDiff = this.id - other.id;
7     if(salaryDiff != 0) { return - salaryDiff; }
8     else { return idDiff; } } }
```

- **L5**: `Double.compare(d1, d2)` returns
  - (`d1 < d2`), 0 (`d1 == d2`), or + (`d1 > d2`).
- **L7**: Why inverting the sign of `salaryDiff`?
  - *this.salary* > *other.salary* ⇒ `Double.compare(`*this.salary*, *other.salary*`)` > 0
  - But we should consider employee with *higher* salary as "smaller".
    ∵ We want that employee to come *before* the other one!

Defining Ordering Between Objects (2.2)

Alternatively, we can use extra `if` statements to express the logic more clearly.

```
class CEmployee2 implements Comparable<CEmployee2> {
  ... /* attributes, constructor, mutator similar to Employee */
  @Override
  public int compareTo(CEmployee2 other) {
    if(this.salary > other.salary) {
      return -1;
    }
    else if (this.salary < other.salary) {
      return 1;
    }
    else { /* equal salaries */
      return this.id - other.id;
    }
}
```


Defining Ordering Between Objects (2.3)

```
@Test
public void testComparableEmployees_2() {
  /*
   * CEmployee2 implements the Comparable interface.
   * Method compareTo first compares salaries, then
   * compares id's for employees with equal salaries.
   */
  CEmployee2 alan = new CEmployee2(2);
  CEmployee2 mark = new CEmployee2(3);
  CEmployee2 tom = new CEmployee2(1);
  alan.setSalary(4500.34);
  mark.setSalary(3450.67);
  tom.setSalary(3450.67);
  CEmployee2[] es = {alan, mark, tom};
  Arrays.sort(es);
  CEmployee2[] expected = {alan, tom, mark};
  assertArrayEquals(expected, es);
}
```

Defining Ordering Between Objects (3)

When you have your class `C` implement the interface `Comparable<C>`, you should design the `compareTo` method, such that given objects `c1`, `c2`, `c3` of type `C`:

- *Asymmetric* :

$$\neg(c1.compareTo(c2) < 0 \land c2.compareTo(c1) < 0)$$
$$\neg(c1.compareTo(c2) > 0 \land c2.compareTo(c1) > 0)$$

$\therefore$ We don't have $c1 < c2$ and $c2 < c1$ at the same time!

- *Transitive* :

$$c1.compareTo(c2) < 0 \land c2.compareTo(c3) < 0 \Rightarrow c1.compareTo(c3) < 0$$
$$c1.compareTo(c2) > 0 \land c2.compareTo(c3) > 0 \Rightarrow c1.compareTo(c3) > 0$$

$\therefore$ We have $c1 < c2 \land c2 < c3 \Rightarrow c1 < c3$

Q. How would you define the `compareTo` method for the `Player` class of a rock-paper-scissor game? [**Hint**: Transitivity]

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 `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.

Static Variables (5.2): Common Error

```
public class Bank {
   public string branchName;
   public static int nextAccountNumber = 1;
   public static void useAccountNumber() {
      System.out.println (branchName + ...);
      nextAccountNumber ++;
   }
}
```

- 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.

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)

```
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)
   */
}
```


OOP: Helper (Accessor) Methods (2.2.1)

```
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)

```
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); } }
  }
}
```


OOP: Helper (Accessor) Methods (2.3)

```
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)

```
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));
  } }
```

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)

```
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;
  }
}
```

OOP: Helper (Mutator) Methods (4.2.2)

```
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;
  }
}
```


OOP: Helper (Mutator) Methods (4.3)

```
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 (1)

Separation of Concerns: App 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 ≈ Templates
OOP:
Define Constructors for Creating Objects (1.1)
OOP:
Define Constructors for Creating Objects (1.2)
The `this` Reference (1)

Index (2)

The `this` Reference (2)
The `this` Reference (3)
The `this` Reference (4)
The `this` Reference (5)
The `this` Reference (6.1): Common Error
The `this` Reference (6.2): Common Error
OOP:
Define Constructors for Creating Objects (2.1)
OOP:
Define Constructors for Creating Objects (2.2)
OOP: Methods (1.1)
OOP: Methods (1.2)
OOP: Methods (2)
OOP: Methods (3)

Index (3)

OOP: The Dot Notation (1)
OOP: The Dot Notation (2)
OOP: Method Calls
OOP: Class Constructors (1)
OOP: Class Constructors (2)
OOP: Class Constructors (3)
OOP: Class Constructors (4)
OOP: Object Creation (1)
OOP: Object Creation (2)
OOP: Object Creation (3)
OOP: Object Creation (4)
OOP: Object Creation (5)
OOP: Object Creation (6)
OOP: Mutator Methods

Index (4)

OOP: Accessor Methods
OOP: Use of Mutator vs. Accessor Methods
OOP: Method Parameters
The `this` Reference (7.1): Exercise
The `this` Reference (7.2): Exercise
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)
Java Data Types (3.3.1)
Java Data Types (3.3.2)
OOP: Object Alias (1)
OOP: Object Alias (2.1)

Index (5)

OOP: Object Alias (2.2)
Call by Value vs. Call by Reference (1)
Call by Value vs. Call by Reference (2.1)
Call by Value vs. Call by Reference (2.2.1)
Call by Value vs. Call by Reference (2.2.2)
Call by Value vs. Call by Reference (2.3.1)
Call by Value vs. Call by Reference (2.3.2)
Call by Value vs. Call by Reference (2.4.1)
Call by Value vs. Call by Reference (2.4.2)
Aggregation vs. Composition: Terminology
Aggregation: Independent Containees
Shared by Containers (1.1)
Aggregation: Independent Containees
Shared by Containers (1.2)

Index (6)

Aggregation: Independent Containees
Shared by Containers (2.1)
Aggregation: Independent Containees
Shared by Containers (2.2)
OOP: The Dot Notation (3.1)
OOP: The Dot Notation (3.2)
OOP: The Dot Notation (3.3)
OOP: The Dot Notation (3.4)
Composition: Dependent Containees
Owned by Containers (1.1)
Composition: Dependent Containees
Owned by Containers (1.2.1)
Composition: Dependent Containees
Owned by Containers (1.2.2)

Index (7)

Composition: Dependent Containees Owned by Containers (1.3)
Composition: Dependent Containees Owned by Containers (1.4.1)
Composition: Dependent Containees Owned by Containers (1.4.2)
Composition: Dependent Containees Owned by Containers (1.5.1)
Composition: Dependent Containees Owned by Containers (1.5.2)
Composition: Dependent Containees Owned by Containers (1.6)
Aggregation vs. Composition (1)
Aggregation vs. Composition (2)
OOP: Equality (1)

Index (8)

OOP: Equality (2)
OOP: Equality (3)
OOP: Contract of `equals`
OOP: Equality (4.1)
OOP: Equality (4.2)
OOP: Equality (4.3)
OOP: Equality (5.1)
OOP: Equality (5.2)
OOP: Equality (5.3)
OOP: Equality (6)
Hashing: What is a Map?
Hashing: Arrays are Maps
Hashing: Naive Implementation of Map
Hashing: Naive Implementation of Map (0)

Index (9)

Hashing: Naive Implementation of Map (1)
Hashing: Naive Implementation of Map (2)
Hashing: Naive Implementation of Map (3)
Hashing: Naive Implementation of Map (4)
Hashing: Naive Implementation of Map (5)
Hashing: Naive Implementation of Map (6)
Hashing: Naive Implementation of Map (7)
Hashing: Naive Implementation of Map (8.1)
Hashing: Naive Implementation of Map (8.2)
Hashing: Naive Implementation of Map (8.3)
Hashing: Naive Implementation of Map (8.4)
Hashing: Hash Table (1)
Hashing: Hash Table as a Bucket Array (2)
Hashing: Contract of Hash Function

Index (10)

Hashing: Defining Hash Function in Java (1)
Hashing: Defining Hash Function in Java (2)
Hashing: Using Hash Table in Java
Hashing: Defining Hash Function in Java (3)
Hashing: Defining Hash Function in Java (4.1)
Hashing: Defining Hash Function in Java (4.2)
Why Ordering Between Objects? (1)
Why Ordering Between Objects? (2)
Defining Ordering Between Objects (1.1)
Defining Ordering Between Objects (1.2)
Defining Ordering Between Objects (2.1)
Defining Ordering Between Objects (2.2)
Defining Ordering Between Objects (2.3)
Defining Ordering Between Objects (3)

Index (11)

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
Static Variables (5.3): Common Error
OOP: Helper Methods (1)
OOP: Helper (Accessor) Methods (2.1)
OOP: Helper (Accessor) Methods (2.2.1)
OOP: Helper (Accessor) Methods (2.2.2)
OOP: Helper (Accessor) Methods (2.3)
OOP: Helper (Accessor) Methods (3.1)

Index (12)

OOP: Helper (Accessor) Methods (3.2)
OOP: Helper (Accessor) Methods (3.3)
OOP: Helper (Accessor) Methods (3.4)
OOP: Helper (Mutator) Methods (4.1)
OOP: Helper (Mutator) Methods (4.2.1)
OOP: Helper (Mutator) Methods (4.2.2)
OOP: Helper (Mutator) Methods (4.3)