Advanced Topics on Classes and Objects

EECS2030 B: Advanced
Object Oriented Programming
Fall 2018

CHEN-WEI WANG

Equality (1)

- 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*.

Equality (2.1)

- Implicitly:
  - Every class is a *child/sub* class of the `Object` class.
  - The `Object` class is the *parent/super* class of every class.
- There is a useful *accessor method* 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);
      }
      ```

    e.g., Say p1 and p2 are of type Point **V1** without the equals method redefined, then p1.**equals**(p2) boils down to (p1 **==** p2).
  - Very often when you define new classes, you want to
    redefine / *override* the inherited definition of equals.

```java
int i = 10;
int j = 12;
boolean sameValue = i.equals(j);
```

Compilation Error:

the equals method is only applicable to reference types.

Fix: write i == j instead.

Equality (3)

```
class PointV1 {
  double x; double y;
  PointV1(double x, double y) { this.x = x; this.y = y; }
}
```

```
1  PointV1 p1 = new PointV1(2, 3);
2  PointV1 p2 = new PointV1(2, 3);
3  System.out.println( p1 == p2 ); /* false */
4  System.out.println( p1.equals(p2) ); /* false */
```

- At **L4**, given that the equals method is not explicitly redefined/overridden in class PointV1, the <u>default version</u> inherited from class Object is called.
  Executing p1.equals(p2) boils down to (p1 == p2).
- If we wish to compare contents of two PointV1 objects, need to explicitly redefine/override the equals method in that class.

Requirements 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*
  $$x.equals(y) \land y.equals(z) \Rightarrow x.equals(z)$$

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

Equality (4.1)

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

```
class PointV2 {
  double x; double y;
  public boolean equals (Object obj) {
    if(this == obj) { return true; }
    if(obj == null) { return false; }
    if(this.getClass() != obj.getClass()) { return false; }
    PointV2 other = (PointV2) obj;
    return this.x == other.x && this.y == other.y; } }
```

```
String s = "(2, 3)";
PointV2 p1 = new PointV2(2, 3); PointV2 p2 = new PointV2(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 */
```

Equality (4.2)

- When making a method call `p.equals(o)`:
  - Variable `p` is declared of type `Point` **V2**
  - Variable `o` can be declared of any type (e.g., `Point` **V2**, `String`)
- 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` at runtime point to objects 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 PointV2 {
  boolean equals (Object obj) {
    if(this == null) { return false; }
```

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

Equality (4.3)

```
1  class PointV2 {
2    boolean  equals (Object obj) { ...
3      if(this.getClass() != obj.getClass()) { return false; }
4      PointV2 other = (PointV2) obj;
5      return this.x == other.x && this.y == other.y; } }
```

- `Object obj` at **L2** declares a parameter obj of type Object.
- `PointV2 other` at **L4** declares a variable p of type PointV2.
  We call such types declared at compile time as *static type*.
- The list of *applicable attributes/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**.

Equality (5)

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

- *Reference Equality* : use == to compare *addresses*

- *Object Equality* : define `equals` method to compare *contents*

```
1  PointV2 p1 = new PointV2(3, 4);
2  PointV2 p2 = new PointV2(3, 4);
3  PointV2 p3 = new PointV2(4, 5);
4  System.out.println(p1 == p1);  /* true */
5  System.out.println(p1.equals(p1));  /* true */
6  System.out.println(p1 == p2);  /* false */
7  System.out.println(p1.equals(p2));  /* true */
8  System.out.println(p2 == p3);  /* false */
9  System.out.println(p2.equals(p3));  /* false */
```

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

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().

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.

Equality (6.3)

LASSONDE

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.

Equality (6.4)

Person collectors are equal if containing equal lists of persons.

```
class PersonCollector {
  Person[] persons; int nop; /* number of persons */
  public PersonCollector() { ... }
  public void addPerson(Person p) { ... }
}
```

Redefine/Override the `equals` method in `PersonCollector`.

```
1  boolean equals (Object obj) {
2    if(this == obj) { return true; }
3    if(obj == null || this.getClass() != obj.getClass()) {
4      return false; }
5    PersonCollector other = (PersonCollector) obj;
6    boolean equal = false;
7    if(this.nop == other.nop) {
8      equal = true;
9      for(int i = 0; equal && i < this.nop; i ++) {
10       equal = this.persons[i].equals(other.persons[i]); } }
11   return equal;
12 }
```

Equality in JUnit (7.1)

- ***assertSame***(obj1, obj2)
  - Passes if obj1 and obj2 are references to the same object
  - ≈ assertTrue(obj1 == obj2)
  - ≈ assertFalse(obj1 != obj2)

  ```
  PointV1 p1 = new PointV1(3, 4); PointV1 p2 = new PointV1(3, 4);
  PointV1 p3 = p1;
  assertSame(p1, p3); /* pass */ assertSame(p2, p3); /* fail */
  ```

- ***assertEquals***(exp1, exp2)
  - ≈ exp1 == exp2 if exp1 and exp2 are **primitive** type

  ```
  int i = 10; int j = 20; assertEquals(i, j); /* fail */
  ```

  - ≈ exp1.equals(exp2) if exp1 and exp2 are **reference** type
    Q: What if equals is not explicitly defined in *obj*1's declared type?
    A: ≈ assertSame(obj1, obj2)

  ```
  PointV2 p4 = new PointV2(3, 4); PointV2 p5 = new PointV2(3, 4);
  assertEquals(p4, p5); /* pass */
  assertEquals(p1, p2); /* fail ∵ different PointV1 objects */
  assertEquals(p4, p2); /* fail ∵ different types */
  ```

```java
@Test
public void testEqualityOfPointV1() {
  PointV1 p1 = new PointV1(3, 4); PointV1 p2 = new PointV1(3, 4);
  assertFalse(p1 == p2); assertFalse(p2 == p1);
  /* assertSame(p1, p2); assertSame(p2, p1); */ /* both fail */
  assertFalse(p1.equals(p2)); assertFalse(p2.equals(p1));
  assertTrue(p1.x == p2.x && p2.y == p2.y);
}
@Test
public void testEqualityOfPointV2() {
  PointV2 p3 = new PointV2(3, 4); PointV2 p4 = new PointV2(3, 4);
  assertFalse(p3 == p4); assertFalse(p4 == p3);
  /* assertSame(p3, p4); assertSame(p4, p3); */ /* both fail */
  assertTrue(p3.equals(p4)); assertTrue(p4.equals(p3));
  assertEquals(p3, p4); assertEquals(p4, p3);
}
@Test
public void testEqualityOfPointV1andPointv2() {
  PointV1 p1 = new PointV1(3, 4); PointV2 p2 = new PointV2(3, 4);
  /* These two assertions do not compile because p1 and p2 are of different types. */
  /* assertFalse(p1 == p2); assertFalse(p2 == p1); */
  /* assertSame can take objects of different types and fail. */
  /* assertSame(p1, p2); */ /* compiles, but fails */
  /* assertSame(p2, p1); */ /* compiles, but fails */
  /* version of equals from Object is called */
  assertFalse(p1.equals(p2));
  /* version of equals from PointP2 is called */
  assertFalse(p2.equals(p1));
}
```

Equality in JUnit (7.3)

```java
@Test
public void testPersonCollector() {
 Person p1 = new Person("A", "a", 180, 1.8); Person p2 = new Person("A", "a", 180, 1.8);
 Person p3 = new Person("B", "b", 200, 2.1); Person p4 = p3;
 assertFalse(p1 == p2); assertTrue(p1.equals(p2));
 assertTrue(p3 == p4); assertTrue(p3.equals(p4));

 PersonCollector pc1 = new PersonCollector(); PersonCollector pc2 = new PersonCollector();
 assertFalse(pc1 == pc2); assertTrue(pc1.equals(pc2));

 pc1.addPerson(p1);
 assertFalse(pc1.equals(pc2));

 pc2.addPerson(p2);
 assertFalse(pc1.persons[0] == pc2.persons[0]);
 assertTrue(pc1.persons[0].equals(pc2.persons[0]));
 assertTrue(pc1.equals(pc2));

 pc1.addPerson(p3); pc2.addPerson(p4);
 assertTrue(pc1.persons[1] == pc2.persons[1]);
 assertTrue(pc1.persons[1].equals(pc2.persons[1]));
 assertTrue(pc1.equals(pc2));

 pc1.addPerson(new Person("A", "a", 175, 1.75));
 pc2.addPerson(new Person("A", "a", 165, 1.55));
 assertFalse(pc1.persons[2] == pc2.persons[2]);
 assertFalse(pc1.persons[2].equals(pc2.persons[2]));
 assertFalse(pc1.equals(pc2));
}
```

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`.

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

```java
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.

```
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:
  - $\boxed{\texttt{ce1.compareTo(ce2) > 0}}$ [ce1 "is greater than" ce2]
  - $\boxed{\texttt{ce1.compareTo(ce2) == 0}}$ [ce1 "is equal to" ce2]
  - $\boxed{\texttt{ce1.compareTo(ce2) < 0}}$ [ce1 "is smaller than" ce2]
- Say ces is an array of CEmployee1 (CEmployee1[] ces),
  calling Arrays.sort(ces) re-arranges ces, so that:

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

```
@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!

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

```
1  class CEmployee2  implements Comparable <CEmployee2> {
2    ... /* attributes, constructor, mutator similar to Employee */
3    @Override
4    public int compareTo(CEmployee2 other) {
5      if(this.salary > other.salary) {
6        return -1;
7      }
8      else if (this.salary < other.salary) {
9        return 1;
10     }
11     else { /* equal salaries */
12       return this.id - other.id;
13     }
14 }
```

```
1   @Test
2   public void testComparableEmployees_2() {
3     /*
4      * CEmployee2 implements the Comparable interface.
5      * Method compareTo first compares salaries, then
6      * compares id's for employees with equal salaries.
7      */
8     CEmployee2 alan = new CEmployee2(2);
9     CEmployee2 mark = new CEmployee2(3);
10    CEmployee2 tom = new CEmployee2(1);
11    alan.setSalary(4500.34);
12    mark.setSalary(3450.67);
13    tom.setSalary(3450.67);
14    CEmployee2[] es = {alan, mark, tom};
15    Arrays.sort(es);
16    CEmployee2[] expected = {alan, tom, mark};
17    assertArrayEquals(expected, es);
18  }
```

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* :

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

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

- *Transitive* :
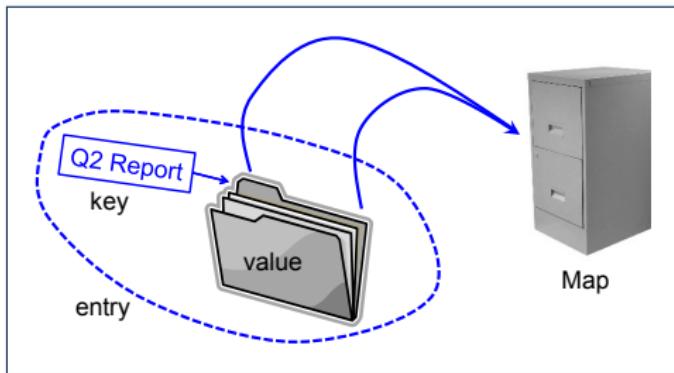
$$c1.compareTo(c2) < 0 \wedge c2.compareTo(c3) < 0 \quad \Rightarrow \quad c1.compareTo(c3) < 0$$
$$c1.compareTo(c2) > 0 \wedge c2.compareTo(c3) > 0 \quad \Rightarrow \quad c1.compareTo(c3) > 0$$

∴ We have $c1 < c2 \wedge c2 < c3 \Rightarrow c1 < c3$

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

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?

  $\qquad$ 3 entries: `(0, "A")`, `(1, "B")`, `(2, "C")`

- *Search keys* are the set of numerical index values.
- The set of index values are *unique* $\qquad$ [e.g., $0 .. (a.length - 1)$]
- Given a **valid** index value i, we can
  - *Uniquely* determines where the object is $\qquad$ [$(i + 1)^{th}$ item]
  - *Efficiently* retrieves that object $\qquad$ [$a[i] \approx$ fast memory access]
- Maps in general may have **non-numerical** key values:
  - Student ID $\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ [student record]
  - Social Security Number $\qquad\qquad\qquad\qquad\qquad$ [resident record]
  - Passport Number $\qquad\qquad\qquad\qquad\qquad\qquad$ [citizen record]
  - Residential Address $\qquad\qquad\qquad\qquad\qquad$ [household record]
  - Media Access Control (MAC) Address $\qquad\quad$ [PC/Laptop record]
  - Web URL $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ [web page]

  $\cdots$

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.
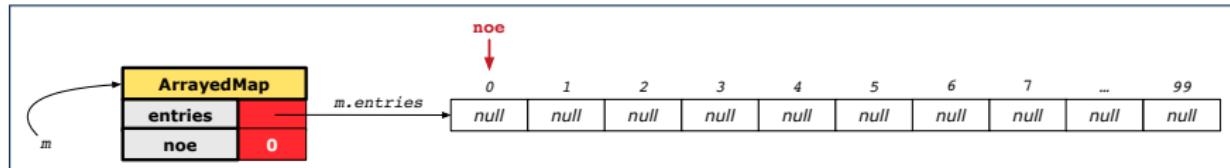
- **Naive Solution**:

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

Hashing: Naive Implementation of Map (0)

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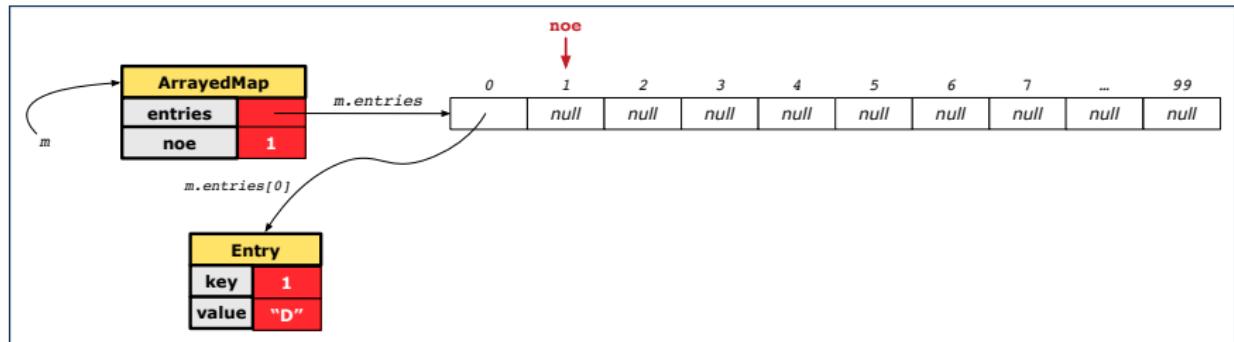
- Attribute `m.entries` initialized as an array of 100 `null` slots.
- Attribute `m.noe` is 0, meaning:
  - Current number of entries stored in the map is 0.
  - Index for storing the next new entry is 0.

Hashing: Naive Implementation of Map (1)

After executing `m.put(new Entry(1, "D"))`:
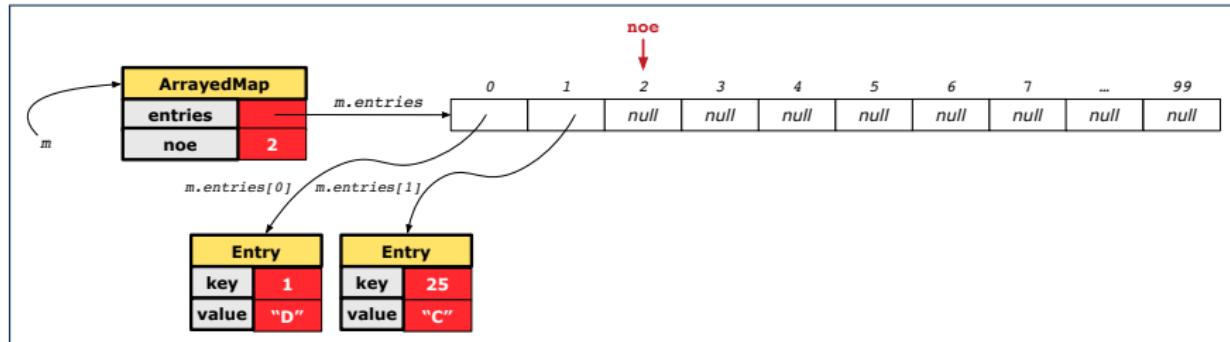
- Attribute `m.entries` has 99 null slots.
- Attribute `m.noe` is 1, meaning:
  - Current number of entries stored in the map is 1.
  - Index for storing the next new entry is 1.

After executing `m.put(new Entry(25, "C"))`:
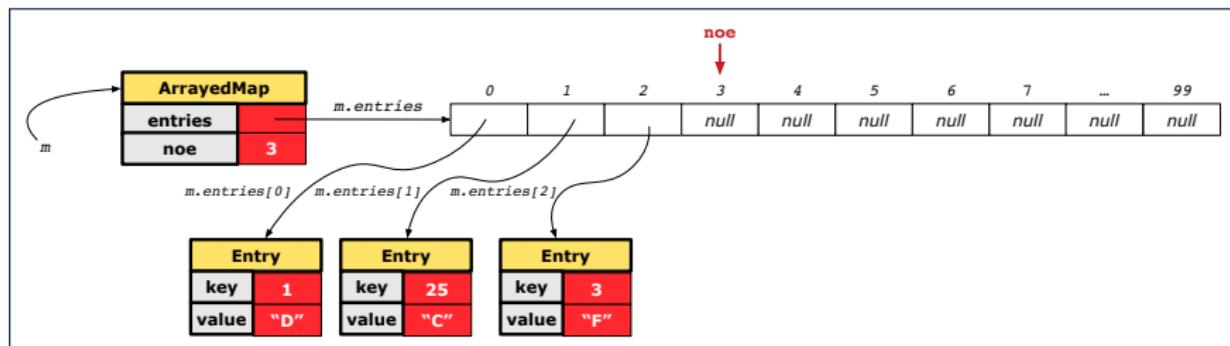
- Attribute `m.entries` has 98 `null` slots.
- Attribute `m.noe` is 2, meaning:
  - Current number of entries stored in the map is 2.
  - Index for storing the next new entry is 2.

After executing `m.put(new Entry(3, "F"))`:
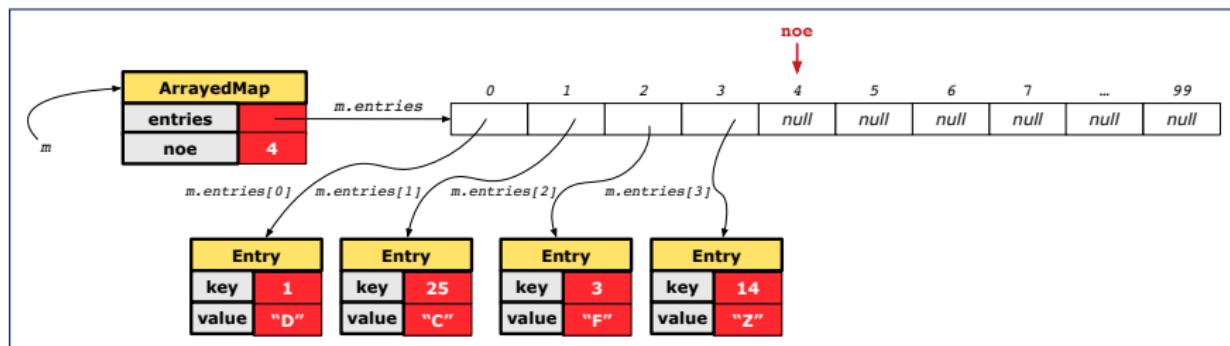
- Attribute `m.entries` has 97 null slots.
- Attribute `m.noe` is 3, meaning:
  - Current number of entries stored in the map is 3.
  - Index for storing the next new entry is 3.

Hashing: Naive Implementation of Map (4)

After executing `m.put(new Entry(14, "Z"))`:
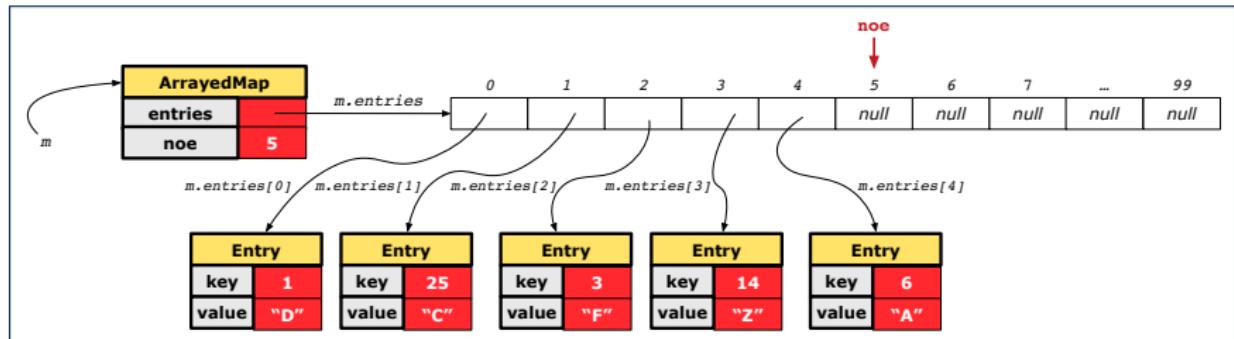
- Attribute `m.entries` has 96 null slots.
- Attribute `m.noe` is 4, meaning:
  - Current number of entries stored in the map is 4.
  - Index for storing the next new entry is 4.

After executing `m.put(new Entry(6, "A"))`:
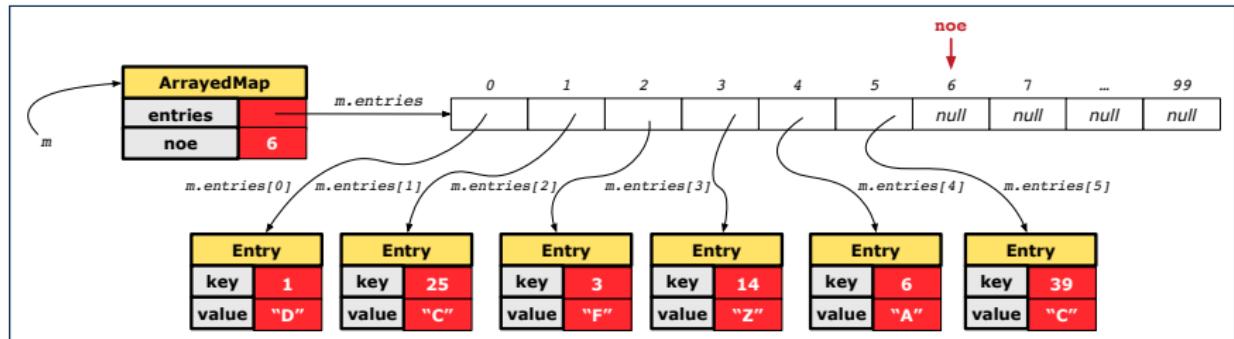
- Attribute `m.entries` has 95 null slots.
- Attribute `m.noe` is 5, meaning:
  - Current number of entries stored in the map is 5.
  - Index for storing the next new entry is 5.

After executing `m.put(new Entry(39, "C"))`:
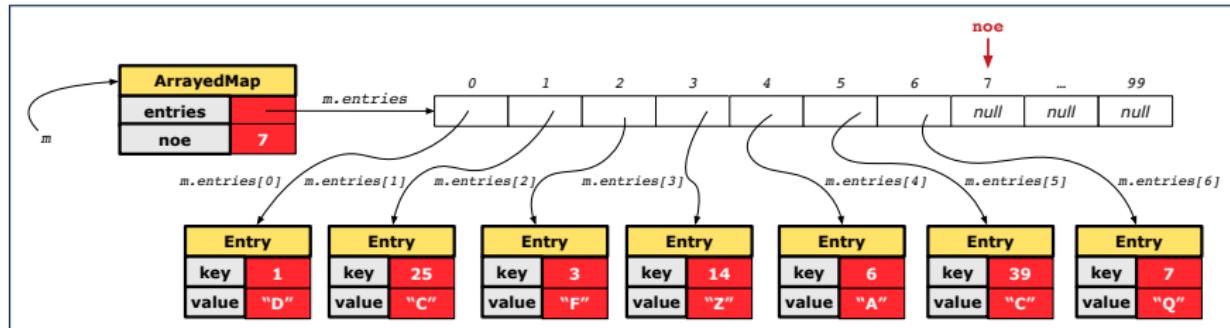
- Attribute `m.entries` has 94 null slots.
- Attribute `m.noe` is 6, meaning:
  - Current number of entries stored in the map is 6.
  - Index for storing the next new entry is 6.

Hashing: Naive Implementation of Map (7)

After executing $\boxed{\text{m.put(new Entry(7, "Q"))}}$:

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

```
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 */
}
```

```
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

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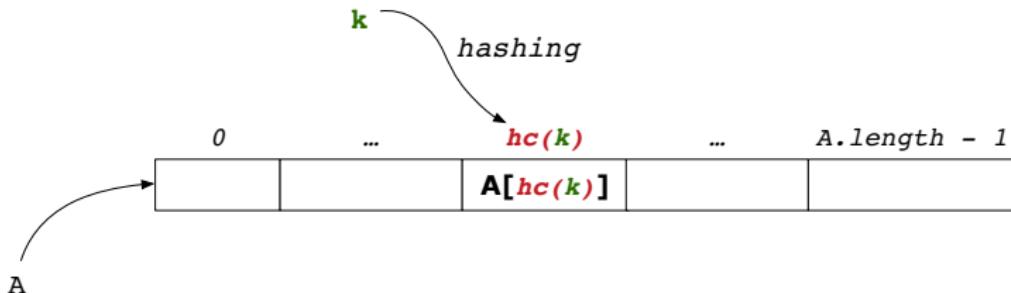
```java
public class ArrayedMap {
 private final int MAX_CAPCAITY = 100;
 public String get (int key) {
   for(int i = 0; i < noe; i ++) {
     Entry e = entries[i];
     int k = e.getKey();
     if(k == key) { return e.getValue(); }
   }
   return null;
 }
```

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!
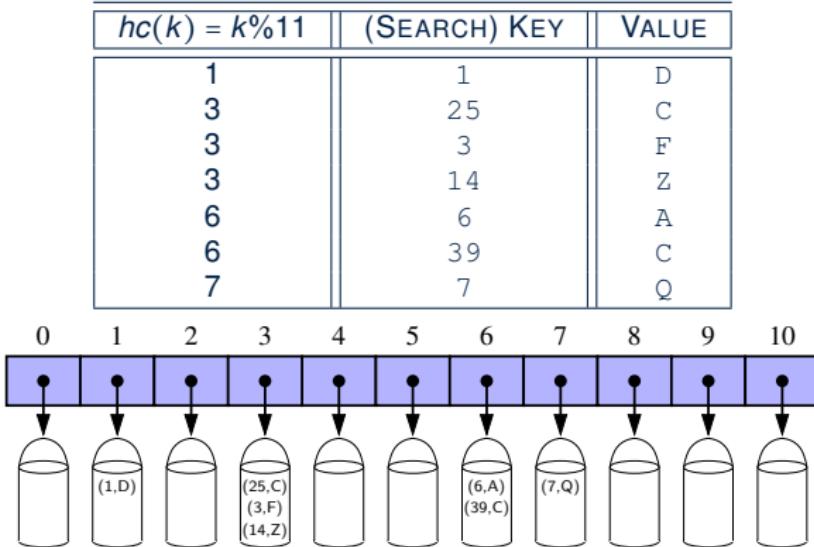
Hashing: Hash Table (1)

```
         k
              hashing

 0        …       hc(k)       …      A.length - 1
┌──────┬──────┬──────────┬──────┬──────┐
│      │      │ A[hc(k)] │      │      │
└──────┴──────┴──────────┴──────┴──────┘

A
```

- 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*)]

For illustration, assume A.length is 11 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)
  $\Rightarrow$ Unavoidable as number of entries $\uparrow$, but a *good* hash
  function should have sizes of the buckets uniformly distributed.

Hashing: Hash Table as a Bucket Array (2.2)

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)
  ⇒ When there are *multiple entries* in the *same bucket*, we
  distinguish between them using their ***unequal*** keys.

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.

- What if $\neg k1.equals(k2)$?
  - $hc(k1) == hc(k2)$ [collision e.g., 25 and 3]
  - $hc(k1) \neq hc(k2)$ [no collision e.g., 25 and 1]
- What if $hc(k1) == hc(k2)$?
  - $\neg k1.equals(k2)$ [collision e.g., 25 and 3]
  - $k1.equals(k2)$ [sound hash function]

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:

```java
 1  public class IntegerKey {
 2    private int k;
 3    public IntegerKey(int k) { this.k = k; }
 4    @Override
 5    public int hashCode() { return k % 11; }
 6    @Override
 7    public boolean equals(Object obj) {
 8      if(this == obj) { return true; }
 9      if(obj == null) { return false; }
10      if(this.getClass() != obj.getClass()) { return false; }
11      IntegerKey other = (IntegerKey) obj;
12      return this.k == other.k;
13    } }
```

Q: Can we replace **L12** by `return this.hashCode() == other.hashCode()`?

A: *No* ∵ When collision happens, keys with same hash code (i.e., in the same bucket) cannot be distinguished.

```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 == 6 */
  IntegerKey ik39_2 = new IntegerKey(39);
  IntegerKey ik6 = new IntegerKey(6); /* 6 % 11 == 6 */

  assertTrue(ik39_1.hashCode() == 6);
  assertTrue(ik39_2.hashCode() == 6);
  assertTrue(ik6.hashCode() == 6);

  assertTrue(ik39_1.hashCode() == ik39_2.hashCode());
  assertTrue(ik39_1.equals(ik39_2));

  assertTrue(ik39_1.hashCode() == ik6.hashCode());
  assertFalse(ik39_1.equals(ik6));
}
```

```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"));
}
```

- 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 (4.1.1)

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;
  } }
```

Hashing:
Defining Hash Function in Java (4.1.2)

```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**?

* Default implementation of hashCode() from the Object class:

  Objects with *distinct* addresses have *distinct* hash code values.

* Violation of the Contract of hashCode():

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

○ What about equal objects with different addresses?

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, 10, 11: Default version of `hashCode`, inherited from `Object`, returns a *distinct* integer for every new object, *despite its contents*.   [*Fix*: Override `hashCode` of your classes!]

Index (1)

Index (2)

Index (4)