

# Object Equality and Call by Value



EECS2030 B & E: Advanced  
Object Oriented Programming  
Fall 2021

CHEN-WEI WANG

# Learning Outcomes

This module is designed to help you learn about:

- **Object equality**: To **Override** or **Not** to Override
- Asserting **Object Equality**: `assertSame` vs. `assertEquals`
- Short-Circuit Effect (SCE): `&&` vs. `||`
- Equality for Array-, Reference-Typed Attributes
- **Call by Value** : Primitive vs. Reference Argument Values

# 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 **values** 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:

- `public boolean equals(Object obj)`

- Indicates whether some other object `obj` is “equal to” this one.
- The default definition inherited from `Object`:

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

e.g., Say `p1` and `p2` are of type `Point` **v1**  
in which the `equals` method is not **redefined/overridden**,  
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`.

## Equality (2.2): Common Error

```
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)

```
public class PointV1 {  
    private int x; private int y;  
    public PointV1(int x, int y) { this.x = x; this.y = y; }  
}
```

```
1 String s = "(2, 3)";  
2 PointV1 p1 = new PointV1(2, 3);  
3 PointV1 p2 = new PointV1(2, 3);  
4 PointV1 p3 = new PointV1(4, 6);  
5 System.out.println(p1 == p2); /* false */  
6 System.out.println(p2 == p3); /* false */  
7 System.out.println(p1.equals(p1)); /* true */  
8 System.out.println(p1.equals(null)); /* false */  
9 System.out.println(p1.equals(s)); /* false */  
10 System.out.println(p1.equals(p2)); /* false */  
11 System.out.println(p2.equals(p3)); /* false */
```

- The `equals` method is not explicitly redefined/overridden in class `PointV1` ⇒ The default version inherited from class `Object` is called. e.g., Executing `p1.equals(null)` boils down to `(p1 == null)`.
- To compare contents of `PointV1` objects, **redefine/override** `equals`.

# 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**  
 $x.equals(y) \wedge y.equals(z) \Rightarrow x.equals(z)$

## Equality (4.1)

To compare **contents** rather than addresses, override `equals`.

```
public class PointV2 {
    private int x; private int 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;
    }
}
```

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



## Equality (4.2)

- When making a method call `p.equals(o)`:
  - Say 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` does **not** store the `null` address.
    - `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 {  
    public boolean equals(Object obj) {  
        if(this == null) { return false; }  
    }  
}
```

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

## Equality (4.3)

```

1  public class PointV2 {
2      public boolean equals(Object obj) {
3          ...
4          if(this.getClass() != obj.getClass()) { return false; }
5          PointV2 other = (PointV2) obj;
6          return this.x == other.x && this.y == other.y;
7      }
8  }

```

- `Object obj` at **L2** declares a parameter `obj` of type `Object`.
- `PointV2 other` at **L5** declares a variable `p` of type `PointV2`.  
We call such types declared at compile time as **static type**.
- *Applicable attributes/methods callable* upon 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 `PointV2`).
- If we are certain 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 `PointV2`.
- The **cast** `(PointV2) obj` creates an **alias** of `obj`, upon which (or upon its alias such as `other`) more methods can be invoked.

## 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.

## Equality in JUnit (1.1)

- **assertSame**(exp1, exp2)
  - Passes if exp1 and exp2 are references to the same object
    - ≈ **assertTrue**(exp1 == exp2)
    - ≈ **assertFalse**(exp1 != exp2)

```
Point v1 p1 = new Point v1(3, 4);  
Point v1 p2 = new Point v1(3, 4);  
Point v1 p3 = p1;  
assertSame(p1, p3); ✓  
assertSame(p2, p3); ✗
```

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

```
int i = 10;  
int j = 20;  
assertEquals(i, j); ✗
```

## Equality in JUnit (1.2)

- **assertEquals**(exp1, exp2)

- $\approx$  `exp1.equals(exp2)` if exp1 and exp2 are **reference** type

**Case 1:** If equals is **not** explicitly overridden in exp1's dynamic type  
 $\approx$  **assertSame**(exp1, exp2)

```
PointV1 p1 = new PointV1(3, 4);
PointV1 p2 = new PointV1(3, 4);
PointV2 p3 = new PointV2(3, 4);
assertEquals(p1, p2);    ✗ /* :: different PointV1 objects */
assertEquals(p2, p3);    ✗ /* :: different object addresses */
```

**Case 2:** If equals is explicitly **overridden** in exp1's dynamic type  
 $\approx$  exp1.**equals**(exp2)

```
PointV1 p1 = new PointV1(3, 4);
PointV1 p2 = new PointV1(3, 4);
PointV2 p3 = new PointV2(3, 4);
assertEquals(p1, p2);    ✗ /* ≈ p1.equals(p2) ≈ p1 == p2 */
assertEquals(p2, p3);    ✗ /* ≈ p2.equals(p3) ≈ p2 == p3 */
assertEquals(p3, p2);    ✗ /* ≈ p3.equals(p2) ≈ p3.getClass() == p2.getClass() */
```

## Equality in JUnit (2)

```
@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.getX() == p2.getX() && p1.getY() == p2.getY());
}

@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 (6.1)

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

```
1 public class Person {
2     private String firstName; private String lastName;
3     private double weight; private double height;
4     public boolean equals(Object obj) {
5         if(this == obj) { return true; }
6         if(obj == null || this.getClass() != obj.getClass()) { return false; }
7         Person other = (Person) obj;
8         return
9             this.weight == other.weight
10            && this.height == other.height
11            && this.firstName.equals(other.firstName)
12            && this.lastName.equals(other.lastName);
13     }
14 }
```

**Q:** At L6, will we get a *NullPointerException* if `obj` is `null`?

**A:** **No** ∴ Short-Circuit Effect of `||`

`obj == 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()`.

## Equality (6.2)

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

```
1 public class Person {
2     private String firstName; private String lastName;
3     private double weight; private double height;
4     public boolean equals(Object obj) {
5         if(this == obj) { return true; }
6         if(obj == null || this.getClass() != obj.getClass()) { return false; }
7         Person other = (Person) obj;
8         return
9             this.weight == other.weight
10            && this.height == other.height
11            && this.firstName.equals(other.firstName)
12            && this.lastName.equals(other.lastName);
13     }
14 }
```

**Q:** At L6, if swapping the order of two operands of disjunction:

`this.getClass() != obj.getClass() || obj == null`

Will we get a **NullPointerException** if `obj` is `null`?

**A: Yes** ∴ Evaluation of operands is from left to right.



## Equality (6.3)

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

```
1 public class Person {
2     private String firstName; private String lastName;
3     private double weight; private double height;
4     public boolean equals(Object obj) {
5         if(this == obj) { return true; }
6         if(obj == null || this.getClass() != obj.getClass()) { return false; }
7         Person other = (Person) obj;
8         return
9             this.weight == other.weight
10            && this.height == other.height
11            && this.firstName.equals(other.firstName)
12            && this.lastName.equals(other.lastName);
13     }
14 }
```

**Q:** At L11 & L12, where is the `equals` method defined?

**A:** The `equals` method **overridden** in the `String` class.

When implementing the `equals` method for your own class, **reuse** the `equals` methods **overridden** in other classes wherever possible.

## Equality (6.4)

Person collectors are equal if containing equal lists of persons.

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

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

```
1 public boolean equals(Object obj) {
2     if(this == obj) { return true; }
3     if(obj == null || this.getClass() != obj.getClass()) { return false; }
4     PersonCollector other = (PersonCollector) obj;
5     boolean equal = false;
6     if(this.nop == other.nop) {
7         equal = true;
8         for(int i = 0; equal && i < this.nop; i++) {
9             equal = this.persons[i].equals(other.persons[i]);
10        }
11    }
12    return equal;
13 }
```

## Equality in JUnit (3)

```
@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.getPersons()[0] == pc2.getPersons()[0]);
    assertTrue(pc1.getPersons()[0].equals(pc2.getPersons()[0]));
    assertTrue(pc1.equals(pc2));
    pc1.addPerson(p3);
    pc2.addPerson(p4);
    assertTrue(pc1.getPersons()[1] == pc2.getPersons()[1]);
    assertTrue(pc1.getPersons()[1].equals(pc2.getPersons()[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.getPersons()[2] == pc2.getPersons()[2]);
    assertFalse(pc1.getPersons()[2].equals(pc2.getPersons()[2]));
    assertFalse(pc1.equals(pc2));
}
```

## Beyond this lecture...

---

- Play with the source code

`ExampleEqualityPointsPersons.zip`

**Tip.** Use the debugger to step into executing the various versions of `equals` method.

- Go back to your Review Tutorial: Extend the `Product`, `Entry`, and `RefurbishedStore` classes by *overridden* versions of the `equals` method.

# Index (1)

---

## Learning Outcomes

Equality (1)

Equality (2.1)

Equality (2.2): Common Error

Equality (3)

Requirements of equals

Equality (4.1)

Equality (4.2)

Equality (4.3)

Equality (5)

Equality in JUnit (1.1)

## **Index (2)**

---

**Equality in JUnit (1.2)**

**Equality in JUnit (2)**

**Equality (6.1)**

**Equality (6.2)**

**Equality (6.3)**

**Equality (6.4)**

**Equality in JUnit (3)**

**Beyond this lecture. . .**