CSE201: Monsoon 2024
Advanced Programming

Lecture 07: The Object Class

Dr. Arun Balaji Buduru

Head, Center of Technology in Policing

Founding Head, Usable Security Group (USG)

Associate Professor, Dept. of CSE | HCD

IIIT-Delhi, India

This Lecture

- Class Object
  - equals method
  - Comparable and Comparator
  - Clonning

Can You Spot Any Similarities?

- Do you see any similarities between a Cat, Universe, and Furniture?
  - If you just look at their photographs then its hard to guess..

2

OK, Can You Spot Any Similarities NOW ?

```java
public class Cat {

    private String name;
    private String breed;

    public Cat() { ... }
    .......
}
```

```java
public class Universe {

    private List<Star> star;

    public Universe(){ ... }
    .......
}
```
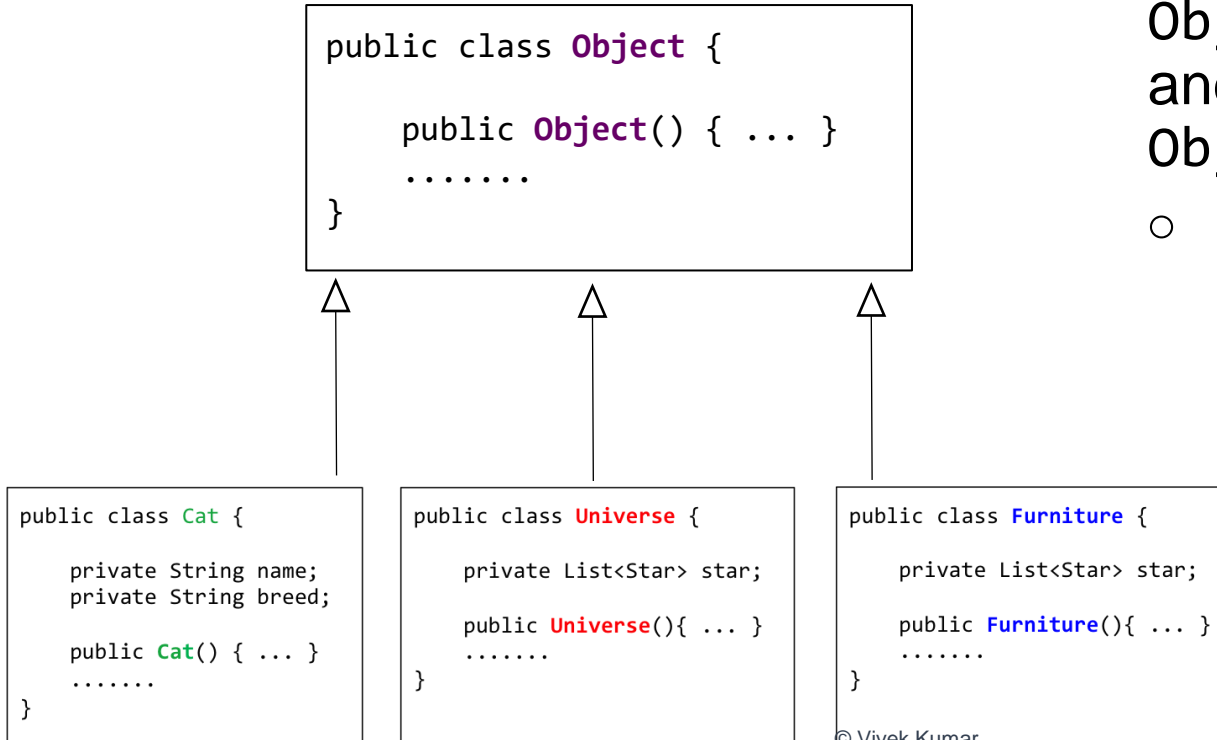
```java
public class Furniture {

    private List<Star> star;

    public Furniture(){ ... }
    .......
}
```

- Now we have a class representation of Cat, Universe and Furniture
  - Do you see any similarities now?

3

They Inherit from Someone!

● What if I tell you that although they look totally unrelated to each other, still they all inherit from a common class, i.e., they have a common parent!
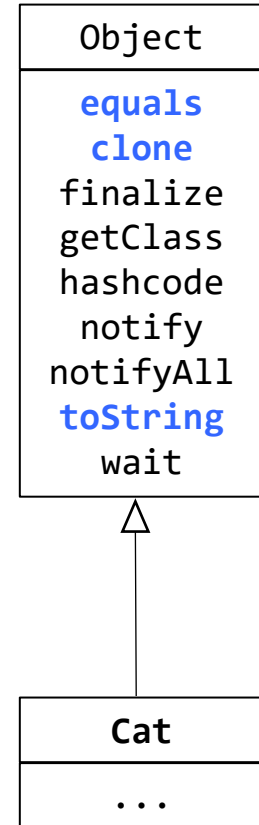
The Class Object in Java

```
public class Object {

    public Object() { ... }
    .......
}
```

```
public class Cat {

    private String name;
    private String breed;

    public Cat() { ... }
    .......
}
```

```
public class Universe {

    private List<Star> star;

    public Universe(){ ... }
    .......
}
```

```
public class Furniture {

    private List<Star> star;

    public Furniture(){ ... }
    .......
}
```

● Every Java class has `Object` as its superclass and thus inherits the `Object` methods
   ○ Due to this, although Cat, Universe and Furniture are totally unrelated, they still inherit from class `Object`

© Vivek Kumar

5

The Class Object

● The class `Object` forms the root of the overall inheritance tree of all Java classes.
   o Every class is implicitly a subclass of `Object`
   o No need to explicitly say "extends Object"

● The `Object` class defines several methods that become part of every class you write. For example:

   o `public String toString()` Returns a text representation of the object, usually so that it can be printed.

Object
equals
clone
finalize
getClass
hashcode
notify
notifyAll
toString
wait

Cat
...

Object Methods

method	description
`protected Object` **`clone`**`()`	creates a copy of the object
`public boolean` **`equals`**`(Object o)`	returns whether two objects have the same state
`protected void` **`finalize`**`()`	called during garbage collection
`public Class<?>` **`getClass`**`()`	info about the object's type
`public int` **`hashCode`**`()`	a code suitable for putting this object into a hash collection
`public String` **`toString`**`()`	text representation of the object

`public void` **`notify`**`()`
`public void` **`notifyAll`**`()`
`public void` **`wait`**`()`
`public void` **`wait`**`(...)`
	methods related to concurrency and locking (seen later)

Using the Object Class

- You can store any object in a variable of type `Object`.

```
Object o1 = new Cat("Meau", "Indian Cat");
Object o2 = "hello there";
```

- You can write methods that accept an `Object` parameter.

```
public void example(Object o) {
    if (o != null) {
        System.out.println("o is " + o.toString());
    }
}
```

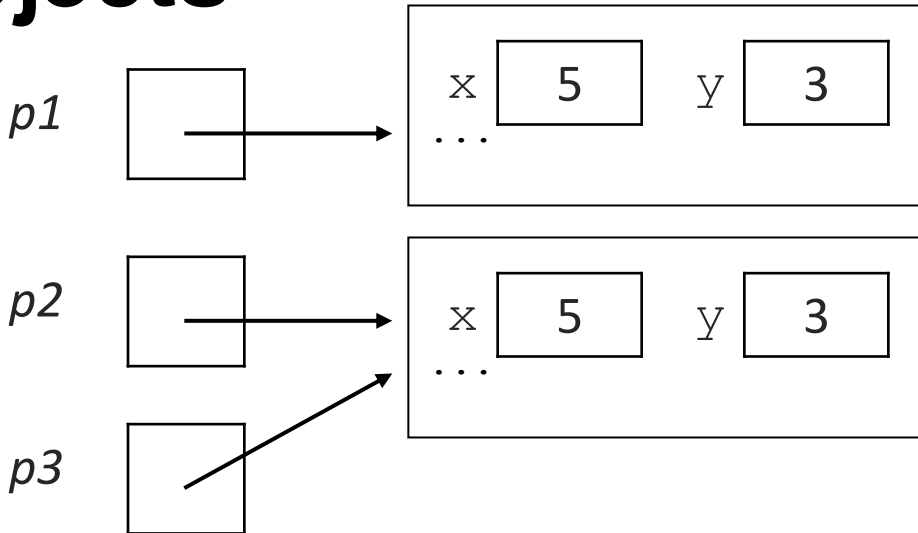- You can make arrays or collections of `Objects`.

```
Object[] a = new Object[5];
a[0] = "hello";
a[1] = new Cat();
List<Object> list = new ArrayList<Object>();
```

8

© Vivek Kumar

Equality Test on Objects

```
Point p1 = new Point(5,3);
Point p2 = new Point(5,3);
Point p3 = p2;

// p1 == p2 is false;
// p1 == p3 is false;
// p2 == p3 is true

// p1.equals(p2)?
// p2.equals(p3)?
```

p1

| x | 5 | y | 3 |
...

p2

| x | 5 | y | 3 |
...

p3

- The == operator does not work well with objects.

  == tests for **referential equality**, not state-based equality.

  It produces true only when you compare an object to itself

9

Default equals Method

- The `Object` class's `equals` implementation is very simple:

```
public class Object {
    ...
    public boolean equals(Object o) {
        return this == o;
    }
}
```

- The Object class is designed for inheritance.
  o Subclasses can *override* `equals` to test for equality in their own way

Is this Correctly Implemented NOW ?

```
1. public class Point {
2.     private int x, y;
3.     public Point(int _x, int _y) { ... }
4.     @Override
5.     public boolean equals(Object o1) {
6.         Point o = (Point) o1; //type casting
7.         return (x==o.x && y==o.y);
8.     }
9. }
10.
```

- Still incorrect !
  - **Flaw-3**
    - It compiles and works fine if Point type objects are passed but fail to compile if non-Point type objects are passed
  - The typecasting will be an issue for following statement
    ```
    Object o1=new Point(1,2);
    Object o2="hello";
    boolean cond=o1.equals(o2);
    ```
  - The flaw is in line 6 as not every Object will be of Point type:
    ```
    Point o = (Point) o1;
    ```
    ClassCastException!!

11

The `instanceof` Keyword

```
if (variable instanceof type) {

        statement;

}
```

- Tests whether **variable** refers to an object of class **type** (or any subclass of **type**)

```
String s = "hello";
Point p = new Point();
```

expression	result
s instanceof Point	false
s instanceof String	true
p instanceof Point	true
p instanceof String	false
p instanceof Object	true
s instanceof Object	true
null instanceof String	false
null instanceof Object	false

(null is a reference and is not an object)

Is this Correctly Implemented NOW ?

```
1. public class Point {
2.     private int x, y;
3.     public Point(int _x, int _y) { ... }
4.     @Override
5.    public boolean equals(Object o1) {
6.        if(o1 instanceof Point) {
7.           Point o = (Point) o1; //type casting
8.           return (x==o.x && y==o.y);
9.        }
10.       else {
11.          return false;
12.       }
13.    }
14. }
15. // subclass of Point
16. class Point3D extends Point {
17.     private int z;
18.     public Point3D(int_x,int _y,int _z) {...}
19.     .....
20. }
```

● Still incorrect !
  ○ **Flaw-4**
    ▪ The method equals will not behave correctly if Point class is extended

```
Point3D p1 = new Point3D(1,2,0);
Point3D p2 = new Point3D(1,2,3);
Point p3 = new Point(1,2);
p1.equals(p2); // true
p2.equals(p3); // true
p3.equals(p1); // true
```

Is this Correctly Implemented NOW ?

```
1. public class Point {
2.     private int x, y;
3.     public Point(int _x, int _y) { ... }
4.     @Override
5.     public boolean equals(Object o1) {
6.         if(o1 instanceof Point) {
7.             Point o = (Point) o1; //type casting
8.             return (x==o.x && y==o.y);
9.         }
10.        else {
11.            return false;
12.        }
13.     }
14. }
15. // subclass of Point
16. class Point3D extends Point {
17.     private int z;
18.     public Point3D(int _x, int _y, int _z) { ... }
19.     @Override
20.     public boolean equals(Object o1) {
8.         if(o1 instanceof Point3D) {
9.             Point3D o = (Point3D) o1; //type casting
8.             return (super.equals(o1) && z==o.z);
9.         }
10.        else {
11.            return false;
12.        }
13.     }
14. }
```

● Still incorrect !

  ○ **Flaw-5**

    ▪ It produces *asymmetric* results when Point and Point3D are mixed

```
Point p1 = new Point(1,2);
Point3D p2 = new Point3D(1,2,3);
p1.equals(p2); // true
p2.equals(p1); // false
```

Equality should be symmetric !!

Rules of Equality for Any Two Objects

- Equality is reflexive:
  - `a.equals(a)` is `true` for every object `a`

- Equality is symmetric:
  - `a.equals(b)` ↔ `b.equals(a)`

- Equality is transitive:
  - `(a.equals(b) && b.equals(c))` ↔ `a.equals(c)`

- No non-`null` object is equal to `null`:
  - `a.equals(null)` is `false` for every object `a`

Finally, the Correct Implementation

```
1. public class Point {
2.     private int x, y;
3.     public Point(int _x, int _y) { ... }
4.     @Override
5.      public boolean equals(Object o1) {
6.         if(o1 != null && getClass() == o1.getClass()) {
7.             Point o = (Point) o1; //type casting
8.             return (x==o.x && y==o.y);
9.         }
10.         else {
11.             return false;
12.         }
13.     }
14. }
15.  // subclass of Point
16. class Point3D extends Point {
17.     private int z;
18.     public Point3D(int _x, int _y, int _z) { ... }
19.     @Override
20.     public boolean equals(Object o1) {
8.         if(o1 != null && getClass() == o1.getClass()) {
9.             Point3D o = (Point3D) o1; //type casting
8.             return (super.equals(o1) && z==o.z);
9.         }
10.         else {
11.             return false;
12.         }
13.     }
14. }
```

- getClass returns information about the type of an object
  - Stricter than instanceof; subclasses return different results
- getClass should be used when implementing equals
  - Instead of instanceof to check for same type, use getClass
  - This will eliminate subclasses from being considered for equality
  - Caution: Must check for null before calling getClass

16

Comparing Objects

Comparing Objects in Java

.equals() = true

.equals() = false

Can we use `equals` to get the above arrangement?

- We have seen how to check equality between two objects:
  - `Obj1 == Obj2`
  - `Obj1.equals(Obj2)`
- But how to check the following:
  - `Obj1 < Obj2`
  - `Obj1 > Obj2`
- Operators like < and > do not work with objects in Java

18

© Vivek Kumar

Comparing Objects in Java

■.compareTo( ) < 0

■.compareTo(■) > 0

■.compareTo( ) = 0

A call of **A**.compareTo(**B**) should return:

    // if **A** comes "before" **B** in

    // the ordering, a `value < 0`

    // if **A** comes "after" **B** in

    // the ordering, a `value > 0`

    // or exactly 0 if **A** and **B**

    // are "equal" in the ordering

19

The Comparable Interface

- The standard way for a Java class to define a comparison function for its objects is to implement the Comparable interface.

```
public interface Comparable<T> {
    public int compareTo(T other);
}
```

compareTo Example

```java
public class Rectangle implements Comparable<Rectangle> {
    private int sideA, sideB, area;
    public Rectangle (int _a, int _b) { ... }

    @Override
    public int compareTo(Rectangle o) {
      if(area == o.area) return 0;
      else if(area < o.area) return -1;
      else return 1;
    }
}
```

- In this Rectangle class, the compareTo method compares the Rectangle objects as per their area

- You can choose your own comparison algorithm!

compareTo v/s equals

```java
public class Rectangle implements Comparable<Rectangle> {
    private int sideA, sideB, area;
    public Rectangle (int _a, int _b) { ... }

    @Override
    public int compareTo(Rectangle o) {
        if(area == o.area) return 0;
        else if(area < o.area) return -1;
        else return 1;
    }
    @Override
    public boolean equals(Object o1) {
        if(o1 != null && getClass() == o1.getClass()) {
            Rectangle o = (Rectangle) o1; //type casting
            return (sideA==o.sideA && sideB==o.sideB);
        }
        else {
            return false;
        }
    }
}
```

```java
// Area1 = 2 x 32 = 64

Rectangle r1=Rectangle(2, 32);

// Area2 = 4 x 16 = 64

Rectangle r2=Rectangle(4, 16);

if(r1.compareTo(r2)==0) {

    // is this true??

}

if(r1.equals(r2)) {

    // is this true??

}
```

Recall, that two `Rectangle`s with same `area` could still have different values for `sideA` and `sideB`

22

How to Compare Two Objects in Different Styles ?

- Our Rectangle class can only implement one compareTo method and hence only one comparison algorithm (style)

- We may want to compare two Rectangles differently
  - Based on sides
  - Based on area
  - ……

23

Comparator Interface

```java
public interface Comparator<T> {
    public int compare(T first, T second);
}
```

- Interface `Comparator` is an external object that specifies a comparison function over some other type of objects.
  - Allows you to define multiple orderings for the same type.
  - Allows you to define a specific ordering for a type even if there is no obvious "natural" ordering for that type

Comparator Example

- Using Comparators, two objects could be compared in different possible ways

- For creating different comparison, implement different objects of Comparator type

```
public class RectangleAreaComparator

                        implements Comparator<Rectangle> {

    @Override

    public int compare(Rectangle r1, Rectangle r2) {

        return r1.getArea() - r2.getArea();

    }

}
```

```
public class RectangleSidesComparator

                        implements Comparator<Rectangle> {

    @Override

    public int compare(Rectangle r1, Rectangle r2) {

        if (r1.getSideA() != r2.getSideA()) {

            return r1.getSideA() - r2.getSideA();

        } else {

            return r1.getSideB() - r2.getSideB();

        }

    }

}
```

```
Class Main {

public static void main(String[] args) {

  Rectangle r1=Rectangle(2, 32);

  Rectangle r2=Rectangle(4, 16);

  RectangleAreaComparator rac = new RectangleAreaComparator();

  RectangleSidesComparator rsc = new RectangleSidesComparator();

  int area_result = rac.compare(r1, r2);

  int sides_result = rsc.compare(r1, r2);

}

}
```

© Vivek Kumar

25

Benefits of Comparator

- Java Collections class (*covered later*) provide method for sorting elements of collections

  public static <T> void sort(List<T> list, Comparator(? super T> c)

- You can sort list of Rectangles based on different criteria using the Comparator interface

  Collections.sort(list, new RectangleAreaComparator());

  Collections.sort(list, new RectangleSidesComparator());