

CSE201: Advanced Programming

# Lecture 08: The Object Class

Vivek Kumar

Computer Science and Engineering

IIIT Delhi

[vivekk@iiitd.ac.in](mailto:vivekk@iiitd.ac.in)

# This Lecture

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

# 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..

# OK, Can You Spot Any Similarities NOW ?

```
public class Cat {  
  
    private String name;  
    private String breed;  
  
    public Cat() { ... }  
    .....  
}
```

```
public class Universe {  
  
    private List<Star> star;  
  
    public Universe(){ ... }  
    .....  
}
```

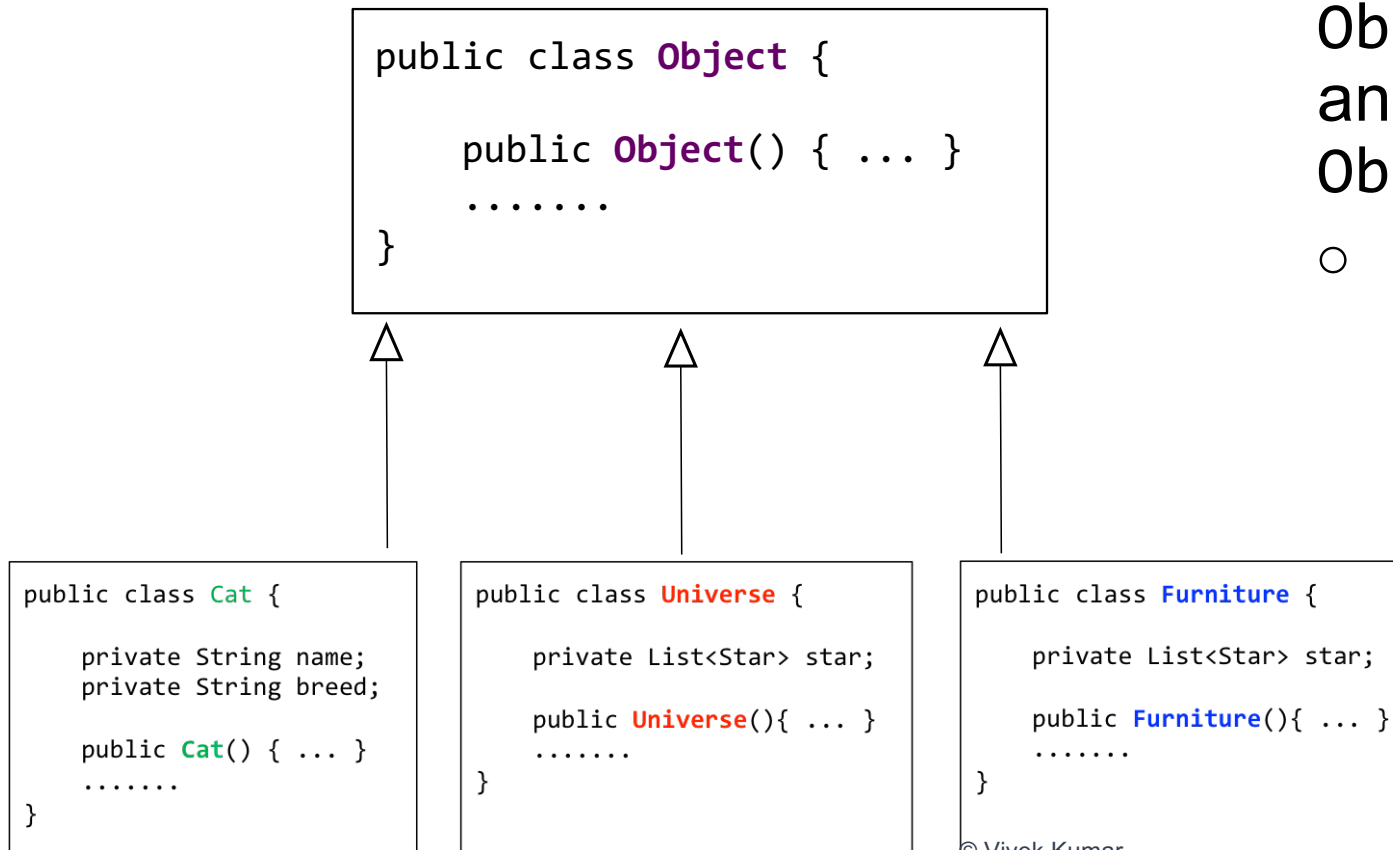
```
public class Furniture {  
  
    private List<Star> star;  
  
    public Furniture(){ ... }  
    .....  
}
```

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

# 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!

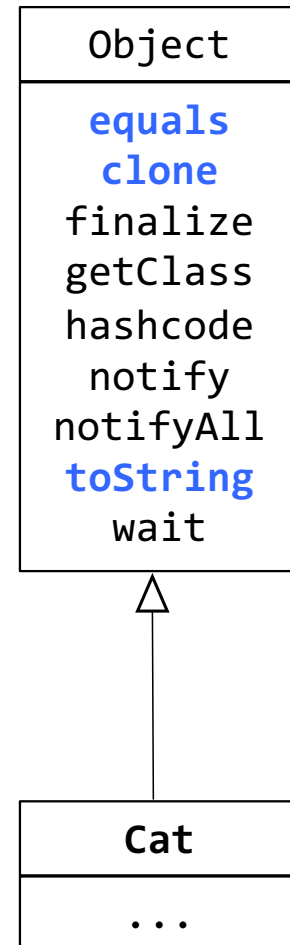
# The Class Object in Java



- 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`

# The Class Object

- The class Object forms the root of the overall inheritance tree of all Java classes.
  - Every class is implicitly a subclass of Object
  - No need to explicitly say “extends Object”
- The Object class defines several methods that become part of every class you write. For example:
  - `public String toString()`  
Returns a text representation of the object, usually so that it can be printed.



# Object Methods

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

**Question:** `speak()` is a method in `Cat` class, is this correct?

- 1) `o1.speak()`
- 2) `o1.toString()`

- 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`.

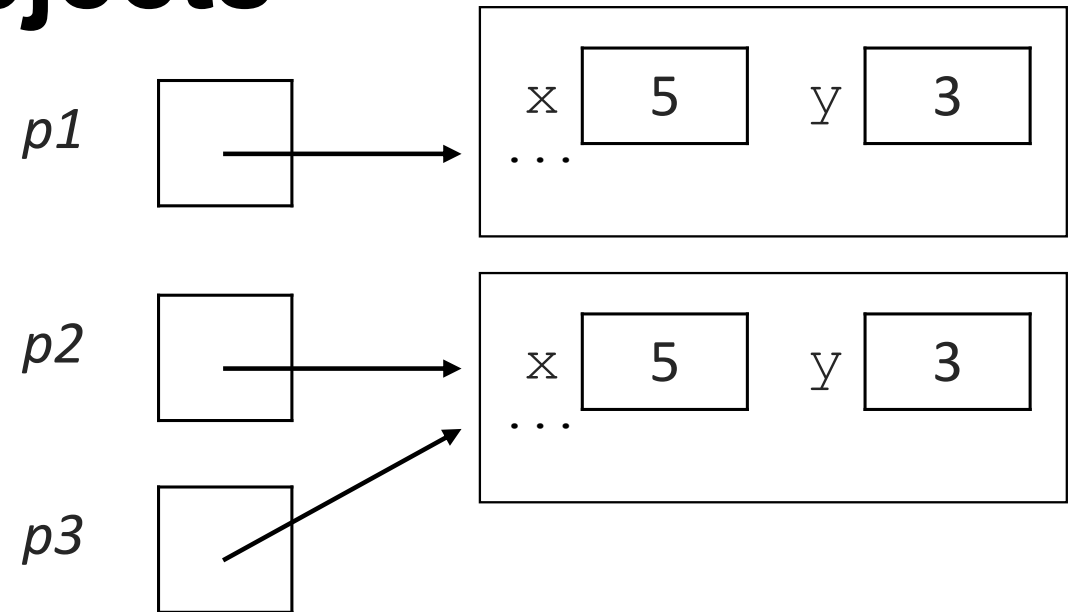
```
Object[] a = new Object[5];  
a[0] = "hello";  
a[1] = new Cat();  
List<Object> list = new ArrayList<Object>();
```

# 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) ?
```



- 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

# 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.
  - Subclasses can *override* equals to test for equality in their own way

# Is this Correctly Implemented

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

- Wrong Implementation !
  - **Flaw-1**
    - Signature of equals method doesn't match with that in class Object
    - Compilation error as we are not overriding!
  - The parameter to equals method is not of type Object but is of type Point
    - **This is method overloading and not overriding**

# 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 o) {
6.         return (x==o.x && y==o.y);
7.     }
8. }
9.
```

- Still incorrect !

- **Flaw-2**

- Compilation error as the parameter to equals is of Object type but then x and y is not defined in class Object

- Can we can do the following:

```
Object o1 = new Point(1, 2);
// Type casting below
Point p = (Point) o1;
```

# 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!!
```

# 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. }
```

© Vivok Kumar

- 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

# 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

# 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

# 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

```
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

```
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;
        }
    }
}
```

```
// 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 Rectangles with same area could still have different values for sideA and sideB

# 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
  - .....

# Comparator Interface

```
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

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

© Vivek Kumar

- Using Comparators, two objects could be compared in different possible ways
- For creating different comparison, implement different objects of Comparator type

```
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);
    }
}
```

# 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());
```

# Copying Objects

# Copying objects

- In other languages (common in C++), to enable clients to easily make copies of an object, you can supply a *copy constructor* :

```
// in client code
Point p1 = new Point(-3, 5);
Point p2 = new Point(p1);      // make p2 a copy of p1
```

```
// in Point.java
public Point(Point blueprint) {    // copy constructor
    this.x = blueprint.x;
    this.y = blueprint.y;
}
```

- Java has some copy constructors but also has a different way...

# Object clone method

```
protected Object clone()  
    throws CloneNotSupportedException
```

- Creates and returns a copy of this object. General intent:
  - `x.clone() != x`
  - `x.clone().equals(x)`
  - `x.clone().getClass() == x.getClass()`
    - (though none of the above are absolute requirements)



# clone() must be Implemented

- If we want to clone **Point** type objects, **Point** class must implement **clone()** method

```
Point.java:11: error: incompatible types: Object cannot be converted to Point
    Point p2 = p.clone();
                  ^
1 error
```

- You must also make your class implement the `Cloneable` interface to signify that it is allowed to be cloned

# The Cloneable interface

```
public interface Cloneable {}
```

- Why would there ever be an interface with no methods?
  - Another example: `Set` interface, a sub-interface of `Collection`
- **Tagging/marker interface:** One that does not contain/add any methods, but is meant to mark a class as having a certain quality or ability.
  - Generally a wart in the Java language; a misuse of interfaces.
  - Now largely unnecessary thanks to *annotations* (seen later).
  - But we still must interact with a few tagging interfaces, like this one.
- Let's implement clone for a `Point` class...

# What's Wrong with the Below Method?

```
public class Point implements Cloneable {
    private int x, y;
    ...
    public Point clone() {
        Point copy = new Point(this.x, this.y);
        return copy;
    }
}
```

# The flaw

```
// also implements Cloneable and inherits clone()  
public class Point3D extends Point {  
    private int z;  
    ...  
}
```

- The above `Point3D` class's `clone` method produces a `Point`!
  - This is undesirable and unexpected behavior.
  - The only way to ensure that the clone will have exactly the same type as the original object (even in the presence of inheritance) is to call the `clone` method from class `Object` with `super.clone()`.

# Proper clone method (1/2)

```
public class Point implements Cloneable {
    private int x, y;
    ...
    public Point clone() {
        try {
            Point copy = (Point) super.clone();
            return copy;
        } catch (CloneNotSupportedException e) {
            // this will never happen
            return null;
        }
    }
}
```

- To call Object's clone method, you must use try/catch.
  - But if you implement Cloneable, the exception will not be thrown.

# Proper clone method (2/2)

```
public class Point implements Cloneable {
    private int x, y;
    ...
    public Point clone() {
        try {
            Point copy = (Point) super.clone();
            return copy;
        } catch (CloneNotSupportedException e) {
            // this will never happen
            return null;
        }
    }
}
```

```
public class Point3D extends Point {
    int z;
    ...
    public Point3D clone() {
        Point3D p = (Point3D) super.clone();
        p.z = this.z;
        return p;
    }
}
```

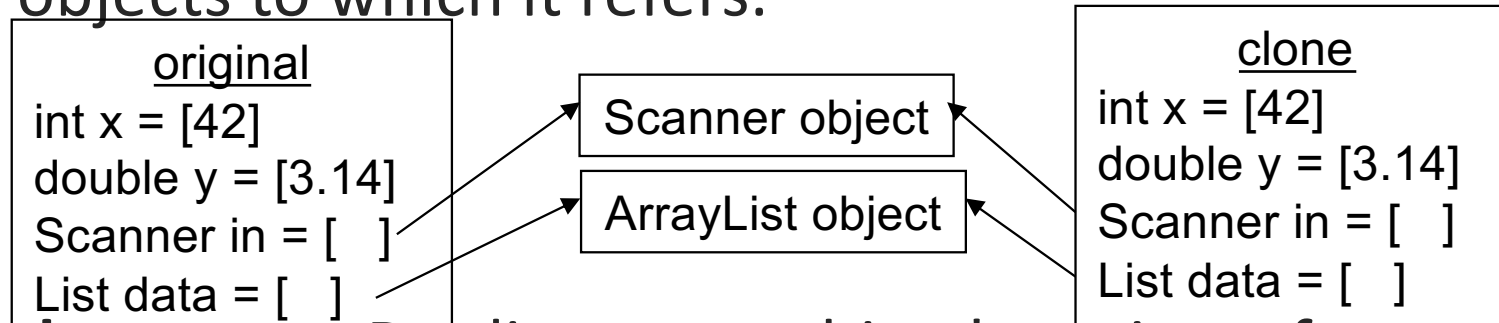
- Every subclass should re-implement clone and must call super.clone() internally
- Only the topmost class in parent-child hierarchy should call super.clone() inside try/catch exception handling block

# What's Still Wrong with the Below Method?

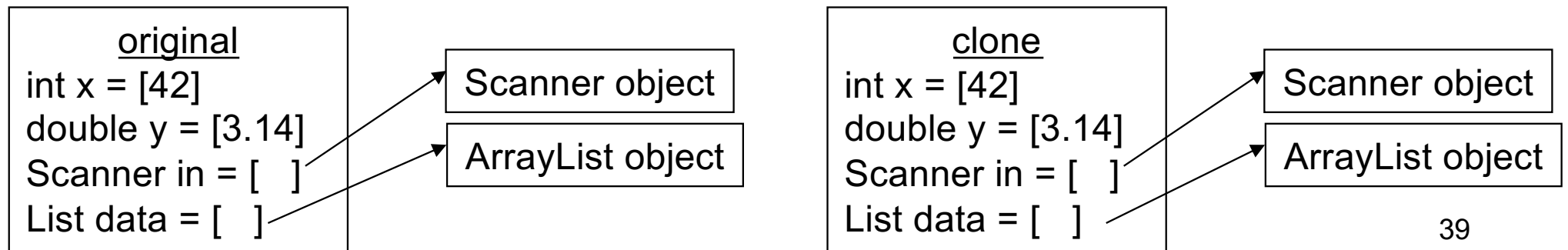
```
public class BankAccount implements Cloneable {
    private String name;
    private List<String> transactions;
    ...
    public BankAccount clone() {
        try {
            BankAccount copy = (BankAccount) super.clone();
            return copy;
        } catch (CloneNotSupportedException e) {
            return null;    // won't ever happen
        }
    }
}
```

# Shallow vs. deep copy

- **shallow copy:** Duplicates an object without duplicating any other objects to which it refers.



- **deep copy:** Duplicates an object's entire *reference graph*: copies itself and deep copies any other objects to which it refers.





# Proper clone method 2

```
public class BankAccount implements Cloneable {
    private String name;
    private List<String> transactions;
    ...
    public BankAccount clone() {
        try {
            // deep copy
            BankAccount copy = (BankAccount) super.clone();
            copy.transactions = new ArrayList<String>(transactions);
            return copy;
        } catch (CloneNotSupportedException e) {
            return null; // won't ever happen
        }
    }
}
```

- Copying the list of transactions (and any other modifiable reference fields) produces a deep copy that is independent of the original

# Next Lecture

- Generic programming
- Quiz-2
  - Syllabus: Lectures 04-07
- Assignment-3 (inheritance & polymorphism)
  - Assignment-4 will build on assignment-3, so please submit assignment-3 for attempting assignment-4