

# Object

## Ch 7.2-7.4

# Topics

- 1) What does Object do for us?
  - a) toString()
  - b) equals()
  - c) hashCode()
  - d) clone()

Gee, that's it? That should be simple, eh?!

toString()

# toString()

- Automatically called when:
  - you concatenate an object with a string.
  - you print an object with `print()` or `println()`.
- Default implementation
  - `Object.toString()` prints class name and hash code.  
`ToStringExample@6876fb1b`

- Reasonable `toString()` Structure

```
@Override
public String toString() {
    return getClass().getName()
        + "["
        + ... fields=values....
        + "]" ;
}
```

`getClass()` gets the..

..

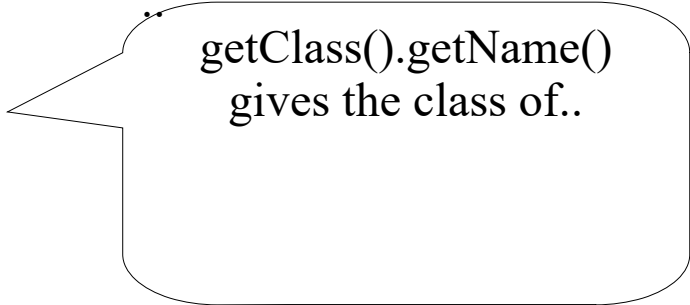
`getClass().getName()`  
returns the package name  
and full class name.

# toString() Guidelines

- toString() should:
  - return “a concise but informative representation that is easy for a person to read” [Javadocs for Object]
  - include all important fields in a human readable format
- Value Classes
  - For a “value” class (like PhoneNumber, Department) return..
    - “(604) 123-4567”, “CMPT”
  - Not for formatting complex objects for the UI:
    - “Minion ‘Horrible Harry’ has 5 evil deeds”
  - ..
- toString() not to replace getter functions;  
never get access to internals by parsing output of toString()

# toString() w/ Inheritance

```
class Employee {
    private String name;
    private int salary;
    @Override
    public String toString() {
        return getClass().getName()
            + "[name="+name
            + ",salary="+salary
            + "]";
    }
}
```



getClass().getName()  
gives the class of..

```
class Manager extends Employee {
    private int bonus;
    @Override
    public String toString() {
        return super.toString()
            + "[bonus="+bonus
            + "]";
    }
}
```

```
public static void main(String[] args) {
    Employee bob = new Employee(...);
    System.out.println(bob);

    Manager sarah = new Manager(...);
    System.out.println(sarah);
}
```

Employee[name=Bob,salary=50000]

Manager[name=Sarah,salary=100][bonus=200000]

equals()

# equals()

- equals() checks if..

- Common error:

```
Car a, b;  
if (a == b) {  
    ...  
}
```

Checks..  
do a and b point to the same  
address?

- Should use:

```
if (a.equals(b)) {  
    ...do something  
}
```

**But Object Provides:**

```
public boolean equals(Object obj) {  
    ..  
}
```

Override equals() only  
when necessary



# Equality

- Each class defines equals() as needed
  - Used by many classes to check equality.  
Ex: set checking for unique items.
  - Value classes (storing data) often need equals()
- Example: Strings
  - Two Strings equal if..
- Example: ICBC Database
  - Two Cars are equal if..  
even if different colours (likely one repainted)

# Casting Safety

```
class Car {  
    @Override  
    public boolean equals(Object o) {
```

To use o, must cast to Car.  
However, this can throw an exception.

Check o is..  
(Must work for derived objects)

instanceof returns false  
if o is null

```
        if (!(o instanceof Car) return false;
```

```
        Car that = (Car) o;
```

```
    }
```

```
}
```

# Complete equals()

```
class Car {
    private String make;
    private Date year;
    private int seating;
    private double weight;

    @Override
    public boolean equals(Object o) {
        if (o == this) return true;

        if (!(o instanceof Car)) return false;

        Car that = (Car) o;
        return Objects.equals(make, that.make)
            && Objects.equals(year, that.year)
            && seating == that.seating
            && Double.compare(weight, that.weight) == 0;
    }
}
```

- Check if same object (efficiency)
- Check other object is correct class.
- Cast and check fields

# Checking fields for equal

- ..  
Objects.equals(field, that.field)
  - Correctly handles nulls and does not throw exception
  - Calls the field.equals() method.
- ..  
Double.compare(field, that.field) == 0
  - Correctly handles -0.0 vs +0.0, NaN, etc.
- int/short/long/bool/...  
field == that.field
- Skip fields which are not part of the object's logical state (cached values, etc).

```
class Car {
    private String make;
    private Date year;
    private int seating;
    private double wgt;

    @Override
    public boolean equals(Object o) {
        ...
        return Objects.equals(make, that.make)
            && Objects.equals(year, that.year)
            && seating == that.seating
            && Double.compare(wgt, that.wgt) == 0;
    }
}
```

# equals() contract

- For non-null x and y, equals() must be:
  - ..  
x.equals(x) must be true
  - ..  
x.equals(y) iff y.equals(x)  
“x and y must agree if they are equal”
  - ..  
x.equals(y) && y.equals(z) means x.equals(z)
  - ..  
value of x.equals(y) unchanged for multiple invocations
  - ..  
x.equals(null) == false

# Spot the Errors!

```
public class Truck {  
    private String make;  
    private Date year;  
    private double weight;  
  
    public boolean equals(Truck o) {  
        if (this == null) return false;  
  
        if (! (o instanceof Truck)) return false;  
  
        return make.endsWith(o.make)  
            && weight == o.weight  
            && year == o.year;  
    }  
}
```

# instanceof aside

- Use instanceof instead of .getClass()
- Checking identical class violates LSP:  
getClass() == o.getClass()  
only true for identical class
  - Cannot have a derived class..
  - Used with proxy classes, etc.
- With inheritance, instanceof can hit symmetry issues  
car.equals(sportsCar)  
sportsCar.equals(car)
  - equals() is hard with inheritance!

## Violates LSP

```
public boolean equals(Object o) {  
    if (o == null) return false;  
    if (o.getClass() != getClass())  
        return false;  
  
    Car that = (Car) o;  
    return ...;  
}
```

```
    if (!(o instanceof Car))  
        return false;  
vs  
    if (!(o instanceof SportsCar))  
        return false;
```

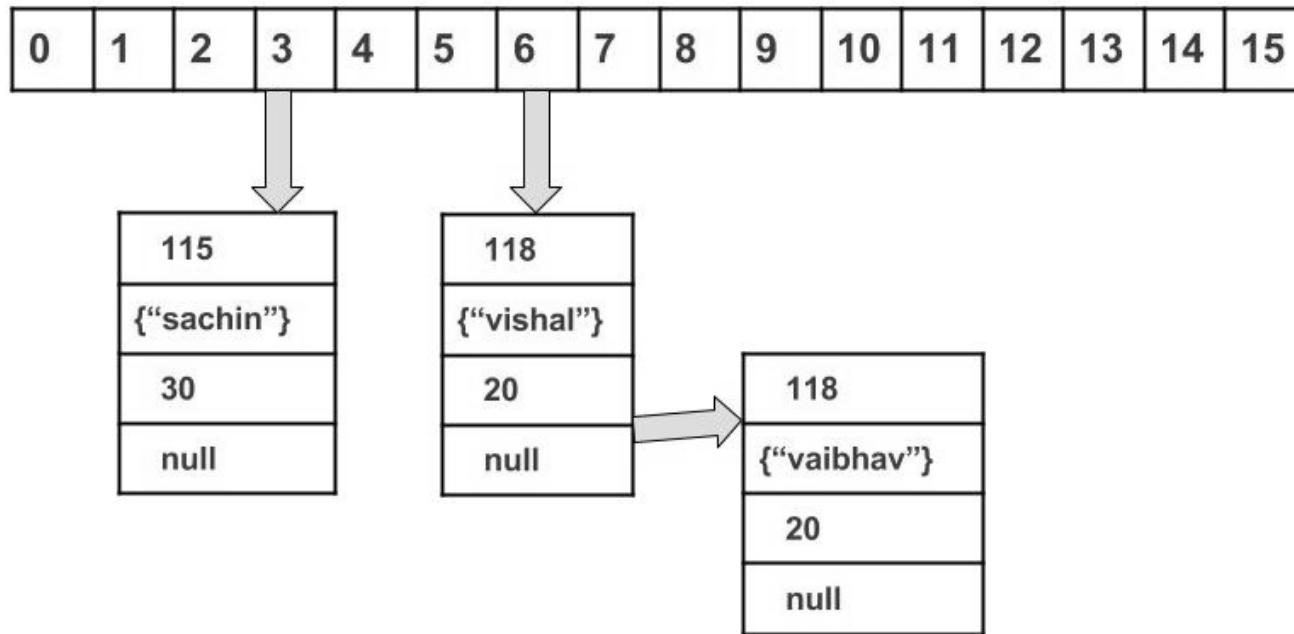
“There is no way to extend an instantiable class and add a value component while preserving the equals contract.”  
- Joshua Bloch (Effective Java 3<sup>rd</sup> ed; p42)

hashCode()



# hashCode()

- A hash code is..
  - Used in hash-maps/hash-sets.
  - Object's hashCode() hashes the memory address.



# Full Example

```
class Car {
    private String make;
    private Date year;
    private int seating;
    private double weight;

    @Override
    public boolean equals(Object o) {
        if (o == this) return true;
        if (!(o instanceof Car)) return false;

        Car that = (Car) o;
        return Objects.equals(make, that.make)
            && Objects.equals(year, that.year)
            && seating == that.seating
            && Double.compare(weight, that.weight) == 0;
    }

    @Override
    public int hashCode() {
        return Objects.hash(make, year, seating, weight);
    }
}
```

- Use `Objects.hash()`
  - Pass all fields which store state.

# hashCode() and equals()

- If overriding equals(),...
  - Hash tables use both methods to find elements:  
Use hashCode() to find the “bin”  
Use equals() to find the object inside the “bin”.
- ..
  - Otherwise collections may not work correctly with the class!
  - Use the same set of fields for computing equals() as for hashCode().

# What is wrong with the following?

```
@Override  
public int hashCode() {  
    return 42;  
}
```

Works!  
But terrible  
efficiency!

Missing  
@Override  
(optional)

Depending on field not used  
in equals!

x and y might show as equal();  
however have different  
hashCode() values!

```
public class Bucket {  
    private String label;  
    private double cost;  
  
    public boolean equals(Object o) {  
        // ... some code omitted...  
        return label.equals(o.label);  
    }  
  
    public int hashCode() {  
        return Objects.hash(cost, label);  
    }  
}
```

# Inheritance with equals() & hashCode()

- Defer to base class the work on the base class's fields.

```
// Note: Poor use of inheritance to add a value to
//       an instantiable class... but anyway.
class SportsCar extends Car{
    private int topSpeed;

    @Override
    public boolean equals(Object o) {
        if (!super.equals(o)) return false;

        if (!(o instanceof SportsCar)) return false;
        SportsCar sportsCar = (SportsCar) o;

        return topSpeed == sportsCar.topSpeed;
    }

    @Override
    public int hashCode() {
        return Objects.hash(super.hashCode(), topSpeed);
    }
}
```

Breaks  
symmetry

clone()

# DANGER!

- clone() is for duplicating objects.
- Want to know about clone?
  - Go read Effective Java 3<sup>rd</sup> ed by Joshua Bloch, Item 13

(Not covered here!)

- Use judiciously!



# Summary

- Subtleties with Object:
  - toString():
    - For UI only if a simple data object
    - Call super.toString() when needed
  - equals():
    - Check type, cast, check equals
    - Watch for null fields
  - hashCode():
    - Override with equals()
    - Consistent with equals()