

The Object Superclass

Fitting Classes Into the Java Hierarchy

`java.lang.Object` defines several methods that are designed to be overridden in subclasses (JLS §4.3.2:)

- ▶ The method `equals(Object)` defines a notion of object equality, which is based on value, not reference, comparison.
- ▶ The method `hashCode` is very used together with `equals(Object)` in hashtables such as `java.util.HashMap`.
- ▶ The method `toString` returns a `String` representation of the object.
- ▶ The method `clone` is used to make a duplicate of an object (don't touch).
- ▶ The method `finalize` is run just before an object is destroyed (don't touch).

A class hierarchy is also sometimes called a **framework**.

When to Override the equals(Object) Method

The default implementation of equals(Object) in java.lang.Object is object identity - each object equals(Object) only itself.

When should a class override equals(Object)?

- ▶ When logical equality differs from object identity, as is the case for **value** classes like Date
- ▶ When classes don't implement instance control, e.g., enum values.
- ▶ When a suitable equals(Object) method is not provided by a superclass.

More important than recognizing **when** to override equals(Object) is knowing **how** to override equals(Object).

How to Override the equals(Object) Method

Obey the general contract of `equals(Object)` (JLS), which says that `equals(Object)` defines an equivalence relation. So, for non-null references, `equals(Object)` is

- ▶ reflexive - any object equals(Object) itself
- ▶ symmetric - if a.equals(Object)(b) is true then b.equals(a) must be true
- ▶ transitive - if a.equals(b) and b.equals(c) are true then a.equals(c) must be true
- ▶ consistent - if a and b do not change between invocations of a.equals(b) or b.equals(a) then each invocation must return the same result
- ▶ a.equals(null) must always return false.

A Recipe for Implementing `equals(Object)`

Obeying the general contract of `equals(Object)` is easier if you follow this recipe ¹.

1. Check for reference equality with `==` (are we comparing to self?).
2. Check that the other object is an `instanceof` this object's class.
3. Cast the other object to this's type (guaranteed to work after `instanceof` test)
4. Check that each “significant” field in the other object `equals(Object)` the corresponding field in this object.

After seeing an example applicaiton of this recipe we'll motivate the proper implementation of `equals(Object)` methods by introducing our first collection class, `ArrayList`.

¹<http://www.informit.com/store/effective-java-9780134685991>

An Example equals(Object) Method

Assume we have a Person class with a single name field.

1. Check for reference equality with == (are we comparing to self?).
2. Check that the other object is an instanceof this object's class.
3. Cast the other object to this's type (guaranteed to work after instanceof test)
4. Check that each "significant" field in the other object equals(Object) the corresponding field in this object.

Applying the recipe:

```
public boolean equals(Object other) {  
1:     if (this == other) { return true; }  
2:     if (!(other instanceof Person)) { return false; }  
3:     Person that = (Person) other;  
4:     return this.name.equals(that.name);  
}
```


ArrayList Basics

Create an ArrayList with operator new:

```
ArrayList tasks = new ArrayList();
```

Add items with add():

```
tasks.add("Eat");  
tasks.add("Sleep");  
tasks.add("Code");
```

Traverse with for-each loop:

```
for (Object task: tasks) {  
    System.out.println(task);  
}
```

Note that the for-each loop implicitly uses an iterator.

The equals(Object) Method and Collections

- ▶ A class whose instances will be stored in a collection must have a properly implemented equals(Object) method.
- ▶ The contains method in collections uses the equals(Object) method in the stored objects.
- ▶ The default implementation of equals(Object) (object identity - true only for same object in memory) only rarely gives correct results.
- ▶ Note that hashCode() also has a default implementation that uses the object's memory address. As a rule, whenever you override equals(Object), you should also override hashCode².

² hashCode() is used in hash-based collections, which we'll learn in a few weeks.

equals(Object) Method Examples

In this simple class hierarchy, FoundPerson has a properly implemented equals(Object) method and LostPerson does not. See [ArrayListEqualsDemo.java](#).

```
abstract static class Person {
    public String name;
    public Person(String name) {
        this.name = name;
    }
}
static class LostPerson extends Person {
    public LostPerson(String name) { super(name); }
}
static class FoundPerson extends Person {
    public FoundPerson(String name) { super(name); }

    public boolean equals(Object other) {
        if (this == other) { return true; }
        if (!(other instanceof Person)) { return false; }
        return ((Person) other).name.equals(this.name);
    }
}
```


Covariant Returns

covariantReturn is covariant in Person and LostPerson:

```
abstract static class Person {  
    public String name;  
    public Person(String name) { this.name = name; }  
    public Object covariantReturn() { return new Object(); }  
}  
static class LostPerson extends Person {  
    public LostPerson(String name) { super(name); }  
    @Override public LostPerson covariantReturn(){return this;}  
}
```

... because LostPerson is a subtype of Object.

But SubLostPerson won't compile:

```
static class SubLostPerson extends LostPerson {  
    public SubLostPerson(String name) { super(name); }  
    @Override public Person covariantReturn() {return this;}  
}
```

... because its covariantReturn's return type, Person, is a supertype of LostPerson.

Non-Covariant Return

NoncompilingPerson won't compile because `int` is not a subtype of `boolean` and because return-type covariance only applies to reference types, not primitives.

```
static class NonCompilingPerson extends Person {
    public NonCompilingPerson(String name) { super(name); }

    /**
     * This method won't compile because int is not a subtype of
     * boolean.
     */
    public int equals(Object other) {
        if (this == other) return 1;
        if (!(other instanceof Person)) return 0;
        return ((Person) other).name.equals(this.name) ? 1 : 0;
    }
}
```

Accidental Overloading

It's easy to make this mistake:

```
static class OverloadedPerson extends Person {  
    public OverloadedPerson(String name) { super(name); }  
  
    public boolean equals(OverloadedPerson other) {  
        if (this == other) { return true; }  
        if (!(other instanceof OverloadedPerson)) { return false; }  
        return ((OverloadedPerson) other).name.equals(this.name);  
    }  
}
```

- ▶ Signature of equals in Object is
`public boolean equals(Object other)` - parameter type is `Object`.
- ▶ In `OverloadedPerson` we've accidentally **overloaded** equals instead of **overriding** equals by making the parameter type `OverloadedPerson`.

Using the `@Override` annotation helps you avoid this mistake.

