

Java Programming 2

Lecture #13

Mary Ellen Foster

MaryEllen.Foster@glasgow.ac.uk

4 November 2019

Outline for today

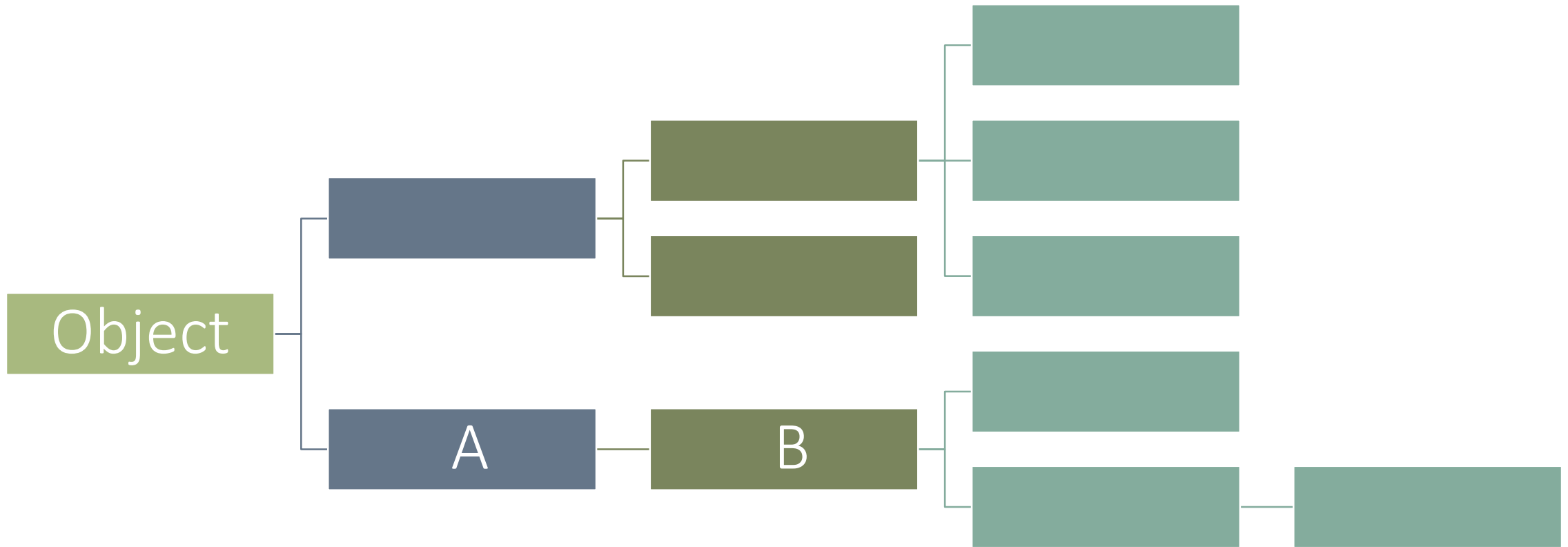
Writing an equals() method

Writing a hashCode() method

The Objects class

Using “varargs” methods

Inheritance in Java



Methods of java.lang.Object

`protected Object clone()`

`boolean equals (Object obj)`

`protected void finalize()`

`public Class<?> getClass()`

`public int hashCode()`

`public void notify() / notifyAll()`

`public String toString()`

`public void wait() / wait(long timeout) / wait(long timeout, int nanos)`

java.lang.Object.equals() documentation

Indicates whether some other object is “equal to” this one

The `equals` method implements an equivalence relation on non-null object references:

It is **reflexive**: for any non-null reference value `x`, `x.equals(x)` should return `true`.

It is **symmetric**: for any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.

It is **transitive**: for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.

It is **consistent**: for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in `equals` comparisons on the objects is modified.

For any non-null reference value `x`, `x.equals(null)` should return `false`.

Default implementation of equals()

“The most discriminating possible equivalence relation on objects”

Returns true **if and only if** x and y refer to the **same** object (i.e., x == y is true)

Gives the correct result for primitive types (int, double, char, etc.)

Does not check if objects are **equivalent** – i.e., if their contents are the same

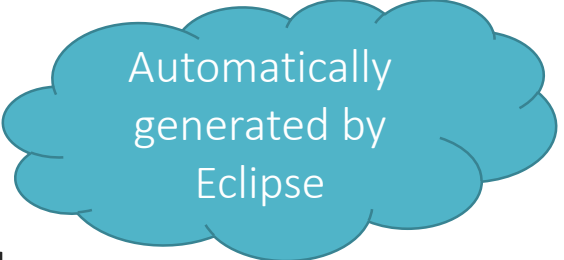
```
ArrayList<Integer> l1 = new ArrayList<>();  
l1.add (1);  
ArrayList<Integer> l2 = new ArrayList<>();  
l2.add (1);  
boolean result = l1.equals(l2); // Default would return false
```

Overriding equals()

Determine what equality means for your particular object

Should all fields be the same?

What if a field is null?



Automatically
generated by
Eclipse

Override the equals() method

Check for self-equality

Check for other object being null

Check that other object has correct type, and cast if it is

Compare field-by-field, testing for null

```
public boolean equals(Object obj) {  
    if (this == obj) {  
        return true;  
    }  
    if (obj == null) {  
        return false;  
    }  
    if (!(obj instanceof Song)) {  
        return false;  
    }  
    Song other = (Song) obj;  
    if (artist == null) {  
        if (other.artist != null) {  
            return false;  
        }  
    } else if (!artist.equals(other.artist)) {  
        return false;  
    }  
    if (title == null) {  
        if (other.title != null) {  
            return false;  
        }  
    } else if (!title.equals(other.title)) {  
        return false;  
    }  
    return true;  
}
```

Remember method signature!

It's tempting to write something like
`public boolean equals(Song other)`

But this WILL NOT override the `equals()` method – it will OVERLOAD it instead with a different signature

Many Java library methods (e.g., Collections things like `ArrayList.contains()`) make use of `Object.equals()` – will not use your implementation!

*Solution: use **@Override** annotation*

Remember to check for null and class!

Check that the **argument** is not null before accessing its fields

Check that each **field** is not null before calling methods on them or accessing fields

Check that argument is of the correct **type** using `instanceof` or similar

`equals()` should never throw a `NullPointerException` or `ClassCastException`

```
public boolean equals(Object obj) {  
    if (this == obj) {  
        return true;  
    }  
    if (obj == null) {  
        return false;  
    }  
    if (!(obj instanceof Song)) {  
        return false;  
    }  
    Song other = (Song) obj;  
    if (artist == null) {  
        if (other.artist != null) {  
            return false;  
        }  
    } else if (!artist.equals(other.artist)) {  
        return false;  
    }  
    if (title == null) {  
        if (other.title != null) {  
            return false;  
        }  
    } else if (!title.equals(other.title)) {  
        return false;  
    }  
    return true;  
}
```

java.lang.Object.hashCode() documentation

Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by `HashMap`.

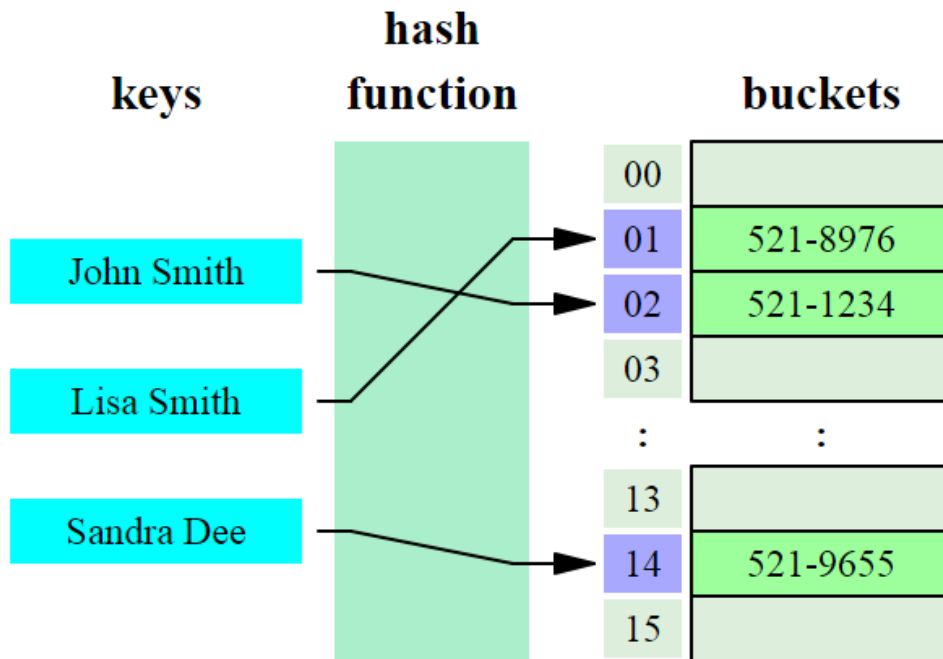
The general contract of `hashCode` is:

Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in `equals` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.

If two objects are equal according to the `equals (Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.

It is not required that if two objects are unequal according to the `equals (java.lang.Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

(Aside: hash tables???)



A data structure used to map **keys** to **values**

Like a dictionary in Python

Uses a **hash function** to compute an index into an array of slots

Ideally, hash function should assign each key to a unique bucket – in practice, collisions occur, but should be minimised for efficiency

Much more about this in ADS2

By Jorge Stolfi - Own work, CC BY-SA 3.0
<https://commons.wikimedia.org/w/index.php?curid=6471238>

Default implementation

“As much as is reasonably practical, the `hashCode` method defined by class `Object` does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the Java™ programming language.)”

equals() and hashCode()

“If two objects are equal according to the `equals (Object)` method, then calling the `hashCode` method on each of the two objects **must produce the same integer result.**”

So if you override `equals ()`, **you must also override `hashCode ()`**

Overriding hashCode()

Take into account the field values used in the equals() calculation

How to do it is up to you; Eclipse will auto-generate a version if you ask it to

```
public int hashCode() {  
    final int prime = 31;  
    int result = 1;  
    result = prime * result + ((artist == null) ? 0 : artist.hashCode());  
    result = prime * result + ((title == null) ? 0 : title.hashCode());  
    return result;  
}
```

equals() and compareTo()

Documentation for Comparable:

It is strongly recommended, but *not* strictly required that $(x.compareTo(y) == 0) == (x.equals(y))$. Generally speaking, any class that implements the Comparable interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."

Making it easier with `java.util.Objects`

“This class consists of static utility methods for operating on objects. These utilities include `null`-safe or `null`-tolerant methods for computing the hash code of an object, returning a string for an object, and comparing two objects.”

Added in Java 1.7

Allows for greatly simplified `equals()` and `hashCode()` implementations

Actual implementations from Song

```
public boolean equals(Object obj) {  
    if (obj == this) return true;  
    if (obj instanceof Song) {  
        Song s = (Song) obj;  
        return Objects.equals(s.artist, this.artist) && Objects.equals(s.title, this.title);  
    }  
    return false;  
}  
  
public int hashCode() {  
    return Objects.hash(artist, title);  
}
```

Objects.equals()

“Returns true if the arguments are equal to each other and false otherwise. Consequently, if both arguments are null, true is returned and if exactly one argument is null, false is returned. Otherwise, equality is determined by using the equals method of the first argument.”

Implementation is:

```
public static boolean equals(Object a, Object b) {  
    return (a == b) || (a != null && a.equals(b));  
}
```

Objects.hash()

Provides a hash code based on all input arguments

(Using `Arrays.hashCode()` internally if you're curious)

Deals appropriately with `null` arguments

Example invocations:

```
// Current Song example
```

```
Objects.hash(artist, title);
```

```
// Monster class
```

```
Objects.hash(type, hitPoints, attackPoints,  
weaknesses);
```

Objects.hash () signature?!?!

Q: How many arguments does it have?

A: A **variable** number!

```
// Current Song example  
Objects.hash (artist,  
title);
```

```
// Monster class  
Objects.hash (type,  
hitPoints, attackPoints,  
weaknesses);
```

“varargs”?

Varargs = variable *[number of]* arguments

Useful when you don't know how many arguments will be passed to a method

```
System.out.printf("%s: %d, %s%n", name, idnum, address);
```

```
System.out.printf("%s: %d, %s, %s, %s%n", name, idnum,  
                address, phone, email);
```

```
System.out.printf("No format");
```

Vararg syntax

Put ellipsis (three dots, "...") after the type of the last parameter

```
public PrintStream printf (String format, Object... args)
```

You can only have **one** varargs parameter, and it has to be the **last**

Thought experiment: why?

Argument(s) get treated as an array of the given type

Varargs sample code

```
public void myFunction (int num1, double num2, String... strings) {  
    System.out.println (num1);  
    System.out.println (num2);  
    System.out.println ("Number of strings: "  
                        + strings.length);  
  
    for (String s : strings) {  
        System.out.println (s);  
    }  
}
```

Implementation of Objects.hash

```
public static int hash(Object... values) {  
    return Arrays.hashCode(values);  
}
```

Signature of Arrays.hashCode:

```
public static int hashCode(Object[] a)
```


Implementation of varargs

The compiler basically **removes** ellipsis and **replaces** it with an array

```
Objects.hash("one", "two") ->  
Objects.hash(new Object[] { "one", "two" } );
```

This means that **usually** you can just use arrays and arg lists interchangeably

Many methods were changed to use varargs when they were introduced, e.g.,
`Arrays.asList()`

Passing null

There can be unexpected behaviour when passing `null`

```
static void count(Object... objs) {  
    System.out.println(objs.length);  
}
```

```
count(null, null, null); // prints "3"
```

```
count(null, null); // prints "2"
```

```
count(null); // throws java.lang.NullPointerException!!!
```

```
Count ((Object)null); // prints "1"
```

<http://stackoverflow.com/a/2926653>

Arrays of primitives

```
static void count(Object... objs) {  
    System.out.println(objs.length);  
}
```

```
count (1, 2); // Returns 2  
count (new int[] { 1, 2 }); // Returns 1  
count (new Integer[] { 1, 2 }); // Returns 2; eclipse warning  
count (new Object[] { 1, 2 }); // Returns 2
```

Based on <http://stackoverflow.com/a/2926653>

Coming up ...

Wednesday: GUI programming with SWING (guest lecture by Mireilla Bikanga Ada)

Friday: **NO TUTORIAL** (I am away)

