COMP6700/2140 Object Equality and *all that*

Alexei B Khorev and Josh Milthorpe

Research School of Computer Science, ANU

March 2017

Topics

1. Overriding and hiding
2. Equality of objects and `equals()` method:
    - When equality makes sense?
    - How to define equals?
    - Difficulty to follow inheritance path to equality
    - Composition as the solution
    - Once overrode `equals()` then do the same to `hashCode()`
    - When equality makes sense?
3. Object doppelganger: to `clone()` or not to `clone()`
4. Wrapper classes and Auto In-boxing/Un-boxing
5. OO Glossary

Overriding and hiding

If inherited *instance* methods can be *overridden*, inherited fields (if a child class introduces a field with the *name but not necessarily type* identical to a field in the parent class) are *hidden*. For a field in the subclass with the same name as a field in the superclass, the latter still exists, but it's no longer accessible by its simple name. The reference must be cast to the superclass type to access it.

```java
class SuperShow {
    public String str = "SuperString";
    public void show() { System.out.println("Super.show: " + str);}
}
class ExtendShow extends SuperShow {
    public String str = "ExtendString";  // hiding the field
    public void show() {                 // overriding the method
        System.out.println("Extend.show: " + str);
    }
}
```

Run *InheritanceTest* class (which involves the parent-child pair *SuperShow* and *ExtendShow*):

```
Extend.show: ExtendString // method is selected by the object class
Extend.show: ExtendString
sup.str = SuperString // field is selected by the reference type
ext.str = ExtendString
```

Class method hiding

Static methods behave similarly to fields: they are *hidden, not overridden*.

Some overriding does not make sense: overriding a class method into instance method (stripping `static`) doesn't make sense, and vice-versa — overriding an instance method into a static one. Both attempts result in the compile errors. (To make sense of these rules, remember **Is-A** relationship between parent and child.) Study the example in A.java, B.java and C.java.

Defining a Method with the Same Signature as a Superclass's Method

Kind of Inheritance	**Superclass Instance Method**	**Superclass Static Method**
Subclass Instance Method	Overrides	Illegal (Compile Error)
Subclass Static Method	Illegal (Compile Error)	Hides

Note: In a subclass, you can *overload* methods inherited from the superclass. Such methods neither hide nor override the superclass methods — they are new methods, unique to the subclass.

Note: When overriding a method, you might want to use the `@Override` annotation that instructs the compiler that you intend to override a method from the superclass. When the compiler detects that the method does not exist in one of the superclasses, it will generate an error.

Reference type, actual class and super

The super can be invoked in any non-static methods. It acts as a reference to the current object *as an instance of its superclass*. When you need to select a parental implementation even if the reference is attached to an instance of the child class, use super.

```
class That {
    protected String getName() { return "That"; } //return the class name
}
class More extends That {
    protected String getName() { return "More"; } //overrides the superclass method
    void printName() {
        That sref = (That) this; // no need to do the cast, though
        System.out.println("1   this.getName() = " + this.getName());
        System.out.println("2   sref.getName() = " + sref.getName());
        System.out.println("3   super.getName() = " + super.getName());
    }
    public static void main(String[] args) { (new More()).printName(); }
}
```

Both sref and super refer to the same object of the type *That*, but super will ignore the real class of the object and use the superclass implementation.

```
1   this.getName()  = More
2   sref.getName()  = More
3   super.getName() = That
```

Equality of objects

1. When equality makes sense?

Equality of objects

1. When equality makes sense?
2. How to define equals?

Equality of objects

1. When equality makes sense?
2. How to define equals?
3. Difficulty to follow inheritance path to equality

Equality of objects

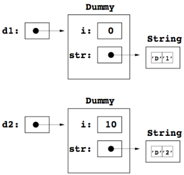
1. When equality makes sense?
2. How to define equals?
3. Difficulty to follow inheritance path to equality
4. Composition as the solution

Equality of objects

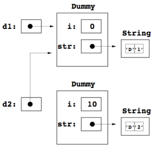
1. When equality makes sense?
2. How to define equals?
3. Difficulty to follow inheritance path to equality
4. Composition as the solution
5. Once overrode `equals()` then do the same to `hashCode()`

Equality of references and objects

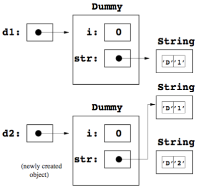
When two objects of the same class can considered *equal* but independent? This depends on how we define the object equality.



```
Dummy d1, d2;
d1 = new Dummy(0,"D1");
d2 = new Dummy(10,"D2");
(d1 != d2) &&
(!d1.equals(d2))
```



```
d2 = d1;
(d1 == d2) &&
(d1.equals(d2))
```



```
d2 = new Dummy(0,"D1");
(d1 != d2)
   but is
d1.equals(d2) ?
```

equals()

Objects of the same type are often compared on equality with one another. The method `Object.equals(Object o)` returns `true` only if the objects are *one and the same* (the default implementation is the test `o == this`). Some classes do require this kind of behaviour (like *Thread*, which represents a process, not a value). But often `equals()` is required as the test of *logical equality*, when two instances of a *value class* are considered equal not only when they not refer to the same object, but also when the objects can be substituted for one another without altering the computational environment. Such `equals()` methods are important for search and placement of elements in instances of *Collection* classes. Demo with two versions of `equals()` in the class A.java (the test running program is TestingEquals.java).

To work correctly, the overridden `equals()` must satisfy the *equivalence relations*:

- be *reflexive*, `x.equals(x)` returns `true`
- be *symmetric*, `y.equals(x)` and `x.equals(y)` return the same value
- be *transitive* and *consistent* (returns the same value over the two objects life if they are subjected the same manipulations)
- `x.equals(null)` returns `false`

Problems and solutions with equals()

Overriding equals() can occur in two ways:

- Inheritance: Extend a class by adding new aspects (fields), *eg Point* → *ColourPoint*
- Composition: Combine all aspects (old and new) into one new class

The above equivalence relations *cannot* be satisfied all at once if is is done on the way of inheritance — there is simply no way to extend a class and add an aspect (a new field) while preserving the equals() contract" (for proof see Joshua Bloch's book "The Effective Java"). However, equals() can be defined with the above properties on the way of composition.

```java
class ColourPoint {
    Point point; Colour colour;
    public boolean equals(Object o) {
        if (!(o instanceof ColourPoint)) return false;
        ColourPoint cp = (ColourPoint) o;
        return cp.point.equals(point) && cp.colour.equals(colour);
    }
}
```

One case, when there is no need to override equals, is when the a class is defined in such a way that at most one object of it can be instantiated (*Singleton* pattern). Another example of types for which equals() is equivalent to == is *Enum* (they allow only a finite number of instances which are defined as a part of the enum type declaration).

Like `equals()` like `hashCode()`

Importance of `hashCode`

If a new class has its `equals()` method overridden, so should be another "primordial" method `java.lang.Object.hashcode()`. This method is used every time an object is inserted in a data structure like `java.util.Map` (where the implementation is done using a hash-table algorithm which calculates an integer using the object state).

The `hashCode` contract

If the method `equals()` has been overridden in a new class, the two different by reference but equal by state objects *must* be placed in the same bucket (which index is returned by the `hashCode()` method). If `hashCode()` is not overridden (or overridden incorrectly), an attempt to store an object in a map data structure will result in placing it in one (wrong) bucket, while an attempt to retrieve the object will likely fail because the look up will be performed in the different (wrong) bucket.

How to create a doppelganger? call `clone()`

Sometimes, the client code needs to create a copy of an object which has the *same state* as the prototype object. This procedure is called **cloning**. The method which can do such creation, `clone()`, is defined in the *Object* class; it is a `native` method. The `Object.clone()` method returns a reference to the Object type object which must be appropriately cast. However, the returned object must be otherwise *independent* from the original one such that subsequent changes to the newly cloned object do not affect the original object (*deep clone*). This task cannot be achieved by simply calling the inherited `clone()` — the `Object.clone()` is declared `protected`, and every subclass needs to explicitly override it, and either keep it `protected`, or promote it to `public` (not always a good idea). When overriding the `clone()` method in a derived class, one should:

- make the class implement a dummy interface *Cloneable* (otherwise `CloneNotSupportedException` is thrown); `Object.clone()` checks whether the object on which it was invoked implements the *Cloneable* interface and throws *CloneNotSupportedException* if it does not; `clone()` always returns an object of the ambient class
- the call for the superclass `clone()` must be supplemented by additional statements insuring that all reference type fields are appropriately initialised
- declare the overridden `clone()` to throw no `CloneNotSupportedException` (this is simplification — the decision to implement *Cloneable* and to (not) throw the exception depends of the class policy)

Good clone()

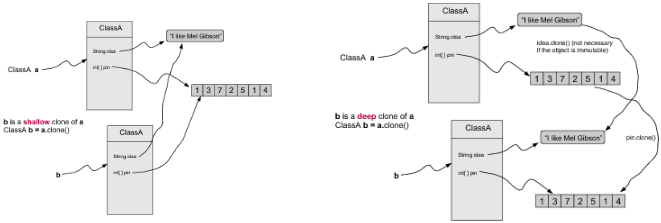
An example, given in B.java class, is demonstrated with both — naive (incorrect) and correct versions of clone(). Class *B* uses a private buffer field (a simple array of int) to provide a *stack type data structure* (for details, see **A6**) which allows to push() a value into the stack, pop() the latest added value, and read the latest added value with getTop(). What if we attempt to clone an existing stack object of *B* which then could be used as *independent* stack? A very important aspect of cloning is to make sure that buffer is correctly cloned too:

```
public B clone() {
  try {
    // recreating the old object with shared reference fields
    B tmp = (B) super.clone();
    // calling the corresponding field's clone
    tmp.buffer = buffer.clone(); /* omit this and you're in trouble! */
    return tmp; // provided buffer.clone() is already correct
  } catch (CloneNotSupportedException e) {
     // Cannot happen -- 'cause we supported the clone
      throw new InternalError(e.toString());
  }
}
```

Shallow and deep clone

The cloning problem is a delicate one and it is dealt with differently by languages:

- *Eiffel* provides the deep clone as a language feature
- *Python* has a package called copy to support both shallow and deep cloning (best decision?)
- *Java* follows a rather quirky approach (feature/library hybrid), when the developer has leverage of whether and how to clone



Implementing `clone()` is a messy business (in Java). Often, a much better way to program object creation in a given state is to define a *copy constructor*; this provides a simpler alternative (*eg*, it can deal with `final` fields).

Turning Java into a pure OO language: Wrapper classes

Inclusion of primitive types in Java is a performance “hack”, not a necessary feature (Smalltalk, or Eiffel, which predate Java, are *pure* OO, without primitives): primitive type variables do not incur initialisation overhead like objects. Also a factor is to maintain the type system familiar to C/C++ practitioners. The trade-off is to sacrifice the expressiveness and uniformity of type system. This artificial division between two kinds of type is not only illogical, but also caused practical limitation (*eg*, collection types can be only contain references as elements). To address these issues, Java provides a *wrapper class* for each primitive type: *Boolean*, *Character* and the abstract *Number* (with concrete subclasses to represent the number types). These classes can be instantiated to carry the data which the corresponding primitive type do. They also provide additional services (conversion, parsing values, *etc*) and type information (like range *etc*).

```java
int i = 10; Integer j = new Integer(i); // wrapping a primitive value
i = j.intValue(); // getting it back from an object
double k = j.doubleValue();
Integer l = Integer.decode("0xAAA"); //decodes string representing a hex number
i = Integer.parseInt((new Scanner(System.in)).next());
```

Purists argued that coexistence of reference and non-reference types is a flaw in language design (eg, by Nick Ourusoff, *Comm. ACM* **45** (8) 2002): “…expression evaluation for primitive types breaks the OO paradigm, data representation is confused with object encapsulation, the machine domain is confused with the application domain…” Yet, currently Java plans to introduce *value classes*.

Automatic Boxing/Unboxing

It used to be awkward to convert from primitives to wrapper objects and back. If `numbers` is an object of `ArrayList<Integer>` class, its every element **is** an *Integer* object reference, which when extracted must be converted before assignment to an `int` variable can be made:

```
int i = 10;
numbers.add(new Integer(i));
Integer j = numbers.get(4); // getting the copy of the element at index 4
int k = j.intValue();
```

Since *Java SE 5*, such explicit conversion is unnecessary:

```
Integer val = 3; // in-boxing conversion
int i = numbers.get(4); // un-boxing conversion
```

The class *Freq* created the word-frequency map reading from the command line:

```
public class Freq {
   public static void main(String[] args) {
      Map m = new TreeMap();
      for (String word : args)
         m.put(word, m.getOrDefault(word, 0) + 1); // new in Java SE 8
      System.out.println(m);
   }
}
```

Object-Oriented Glossary

Basic OO terms and concepts

- Interface (user perspective) — set of methods which can be called on an object
- Encapsulation — concealing implementation details behind object's interface
- Polymorphism — ability to treat objects of different classes by the type of their reference
- Dynamic binding — choosing method implementation at run time based the object's class
- Overloading — using same name for multiple methods
- Overriding — changing method's implementation in a subclass
- Hiding — using same name for a field added in a subclass
- Abstract class — class with incomplete implementation
- Interface (code construct) — type which only declares behaviour and no implementation
- Extension (subclassing, specialising) — reusing existing class in defining a new one
- Implementation — turning an interface type into concrete class

Where to look for this topic in the textbook?

- Hortsmann's Core Java for the Impatient, Ch. 2.2.5, 3.5, 4.1, 4.2