Software Development (cs2500)

Lecture 10: Reference Variables

M.R.C. van Dongen

October 18, 2010

Contents

1 Objectives **1**

2 The `Object` Class **2**
2.1 Object Comparison 2
2.2 The `Object` Class 3

3 Types Matter **3**

4 Sharing Object References **4**

5 Object Lifecycle **5**

6 Arrays **6**
6.1 Array Subscripts 6
6.2 Primitive Type Arrays 7
6.3 Object Type Arrays 7
6.4 Arrays do Not Grow 8

7 Life **8**

8 For Friday **10**

1 Objectives

The start of these notes is not in the book. The rest corresponds to Chapter 3. The main objectives are as follows.

- Study general properties of objects: the `Object` class.

- Revisit the importance of types in Java.
- Study the notion of *alias reference variables*.
- Explore Java's garbage collection mechanism.
- Study properties of objects in general and arrays in particular.
- Carry out a partial case study: Conway's Game of Life.

2 The `Object` Class

This section studies some object-related issues. It starts with a discussion about object comparison. It continues with a presention of some common methods that are defined in the `Object` class.

2.1 Object Comparison

Comparing object references works like as comparing primitive types. $\langle$`reference`$\rangle_1$ `==` $\langle$`reference`$\rangle_2$ if and only if $\langle$`reference`$\rangle_1$ and $\langle$`reference`$\rangle_2$ have the same value. "Having the same value" now means referring to the same object. $\langle$`reference`$\rangle_1$ `!=` $\langle$`reference`$\rangle_2$ if and only if `!` `(`$\langle$`reference`$\rangle_1$ `==` $\langle$`reference`$\rangle_2$`)`.

The following is an example.

```Java
Dog barney = null;
Dog fido   = new Dog( );
if (barney != fido) {
    System.out.println( ”This makes sense.” );
}
barney = fido;
if (barney == fido) {
    System.out.println( ”This also makes sense.” );
}
```

If it looks like a sheep, walks like a sheep, and bleats like a sheep, then it probably is a sheep. This is a reasonable assumption: what matters is the behaviour. In general, two things may be equal "modulo" an equivalence class. For example, 2 and 4 are both even. They are indistinguishable if we can only compare their evenness.

If $\langle$`sheep`$\rangle_1$ `==` $\langle$`sheep`$\rangle_2$ then $\langle$`sheep`$\rangle_1$ and $\langle$`sheep`$\rangle_2$ *definitely* behave the same. However, there may be *different* objects that behave the same. For example, `Sheep` clones. This is why object comparison with '==' is called *shallow* comparison. If $\langle$`object`$\rangle_1$ `==` $\langle$`object`$\rangle_2$ then $\langle$`object`$\rangle_1$ and $\langle$`object`$\rangle_2$ are said to be *shallowly* equal.

Two shallowly equal object references are *deeply* equal. Two shallowly different object references are also deeply equal if their attribute values are deeply equal.

2.2 The `Object` Class

Objects are first-class citizens in `Java`, so it should not come as a surprise that there is a special `Object` class, which all other classes are a subclass of. As you may recall from Lecture 6 subclasses inherit all attributes and methods from their superclass. Except for the `Object` class, any class therefore inherits any (`public`) attributes and methods which are defined in the `Object` class. As it turns out, the `Object` class does not define attributes but it *does* provide methods. The following are some of these methods.

`boolean equals( Object that )`: Indicates whether some other object is "deeply" equal to `this` one.

`int hashCode( )`: Returns the *hash code value* of `this` object. Here the *hash code* of an object is an `int` value which may be used to "partially" recognise the objects. If two objects are deeply equal, their hash codes should be equal.

`String toString( )`: Returns a string representation of the object.

For the moment `toString` is arguably one of the more important methods as it is used when the method is printed. The method gets called automatically when the object is printed.

```Java
Dog dog = new Dog( );
// Same as System.out.println( dog.toString( ) );
System.out.println( dog );
```

For the moment you may forget about the other methods.

3 Types Matter

In the following we shall assume the existence of a `Dog` and a `Giraffe` class. We've already seen that `Java` cares about its types when it came to primitive types. `Java` also cares about object reference types. When dealing with object reference variables, it is generally only allowed to assign like to like. So assigning `Dog` object references to `Dog` object reference variables is allowed. Likewise, assigning `Giraffe` object references to `Giraffe` object reference variables is also allowed. However, (in general) it is not possible to assign a `Giraffe` object reference to a `Dog` object reference variable. You can only assign an $\langle$`class`$\rangle_1$ object reference to a $\langle$`class`$\rangle_2$ object reference variable if $\langle$`class`$\rangle_1$ is a subclass of (a more specific class than) $\langle$`class`$\rangle_2$. The following is a valid `Java` program. Figure 1 depicts the resulting situation on the heap.

```Java
Dog barney = new Dog( );
Dog zeus   = new Dog( );
Giraffe giraffe = new Giraffe( );
```

The following, in general is not allowed. Figure 2 depicts this impossible situation: `Giraffe` objects don't `bark( )`.

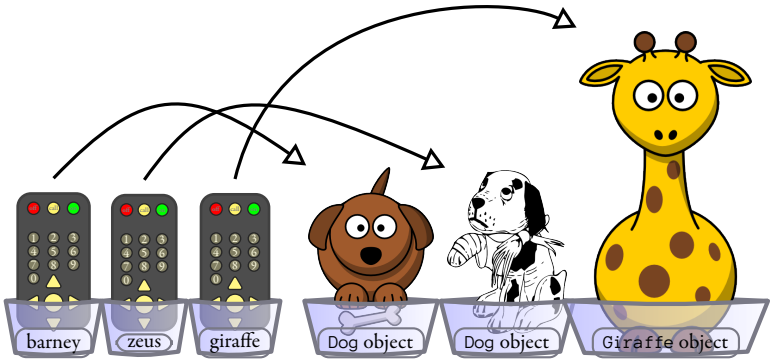

Figure 1: Object reference variable values and objects on the heap after creating two `Dog` objects and one `Giraffe` object.

Don't Try this at Home

```
Dog barney = new Giraffe( ); // Impossible
barney.bark( ); // ???
```

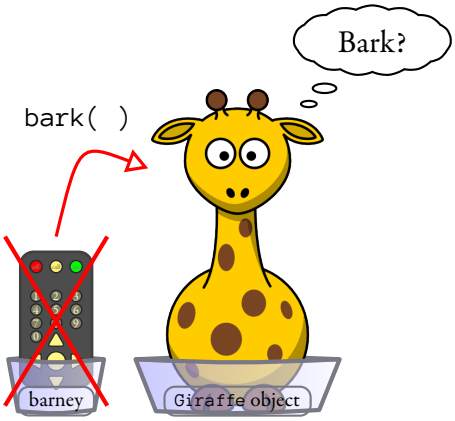

Figure 2: Attempt to assign `Giraffe` object reference to `Dog` object reference variable.

4 Sharing Object References

We've already seen that it's allowed to assign object reference values to object reference variables. However, in all examples the (object reference) values have been created with `new`. With variables of primitive

types we can assign the value of one variable to that of another. The following program demonstrates that this is also allowed with object reference variables. In the example, the last assignment assigns `barney`'s value to the variable `alias`. As is the case with primitive type variables, the net effect of such an assignment is as follows.

- The assignment does not involve the creation of new objects; and
- After the assignment the values of the variables are the same. *If the variable on the right hand side of the assignment refers to an object* before *the assignment, then both variables will refer to that object* after *the assignment*.

Using our TV remote control analogy, the assignment lets `barney` and `alias` control the same TV. In computer science two object reference values that refer to the same object are called *aliases*. Figure 3 depicts the example's heap configuration.

```Java
Dog barney = new Dog( );
Dog zeus   = new Dog( );
Dog alias  = barney;
alias.bark( ); // Makes barney's reference bark too.
```

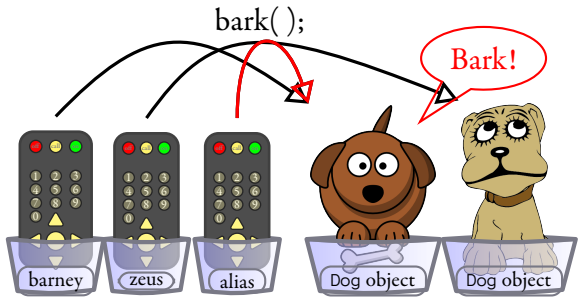

Figure 3: Object reference value sharing. The `Dog` references `barney` and `alias` share the reference to the `Dog` object to the left.

5 Object Lifecycle

Life on the Java heap is a struggle. Due to the huge demand for object allocation space, the JVM (Java Virtual Machine) is on the constant lookout for *garbage collectible* (also known as dead) objects, i.e. objects which are no longer referenced by any object reference variable. Any garbage collectible object may be *garbage collected* at any time by the JVM. Garbage collection is an object's worst nightmare: when this happens to an object it ceases to exist.

The following JAVA code snippet may help you understand the object lifecycle a bit better.

```Java
Dog nosey = new Dog( );
Dog pity  = new Dog( );
Dog puppy = new Dog( );
pity  = nosey;
nosey = puppy;
pity  = nosey;
nosey = pity = puppy = null;
```

- After the fourth assignment statement `pity` and `nosey` have become aliases, but the object which was previously referenced by `pity` is no longer referenced. It is lost forever and cannot be recovered by the program any more. Since there are no more references to the object, the JVM is allowed to garbage collect it at any time.
- After the fifth assignment, `nosey` and `puppy` have become aliases but `pity` and `nosey` are no longer aliases. No additional object has become garbage collectible as a result of this assignment.
- After the second last assignment all object reference variables refer to the same object. They have all become aliases of each other and their value is the original value of `puppy`. As a result of this assignment the object which was originally referred to by `nosey` has become garbage collectible.
- After the last assignment all our `Dog` objects have become garbage collectible. They can be sent to doggy heaven at any time.

6 Arrays

Arrays are objects too. This section is a short introduction to arrays.

6.1 Array Subscripts

Arrays in `Java` are different from `php` arrays. In `Java`:

- An array is a sequence of ℓ things, where ℓ is the *length* of the array. You get the length of the array using the instance attribute `length`. So writing ⟨`array`⟩`.length` gives you the length of ⟨`array`⟩.
- You can look up the ith thing in the array, where i is an integer such that $0 \leq i < \ell$.
- You write ⟨`array`⟩`[ `i` ]` to get the thing at position i in the array ⟨`array`⟩. If i is negative or greater than or equal to ⟨`array`⟩`.length` you get a run-time error.
- You may use ⟨`array`⟩`[ `i` ]` just like you use any variable. The best thing is to regard ⟨`array`⟩`[ `i` ]` as the variable that stores the value of the ith thing in ⟨`array`⟩.

The following is an example. In the example, it is assumed that `nums` is an array that stores `int` values.

```Java
for ( int index = 0; index != nums.length; index ++ ) {
    nums[ index ] = index * index;
}
for ( int index = 0; index != nums.length; index ++ ) {
    String str = "nums[ " + index + " ] = " + nums[ index ];
    System.out.println( str );
}
```

6.2 Primitive Type Arrays

When an array of primitive type elements is created, all its members are initialised to a default value. For `boolean` the default value is `false`, for `char` it is `'\u0000'`, and for all other types it is `0`. The spell '`new` ⟨`type`⟩`[` ⟨`size`⟩`]`' creates an array with ⟨`size`⟩ members. The type of the members in the array is ⟨`type`⟩.

The following is an example. Figure 4 depicts the state of the array object at the end of this example.

```Java
byte[] nums = new byte[ 5 ];
nums[ 1 ] = 4;
nums[ 4 ] = 17;
```

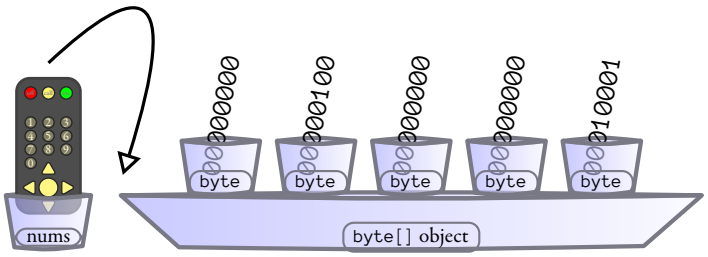

Figure 4: A primitive array object on the heap. The array object reference variable `nums` refers to the array. The members of the array are all `int`s.

6.3 Object Type Arrays

When an array of object type elements is created, all its members are also initialised to a default value. Here the default value is `null`, which is a special value, which can be assigned to *any* object reference variable. However, the value `null` does *not* correspond to any object. This explains why you are not allowed to use a `null`-valued object reference variable's attributes and methods. Any such attempt will lead to a run-time error. For example, if the value of the `Dog` object reference variable `dog` is `null`, then writing '`dog.bark( )`' is not allowed.

```Java
Dog[] dogs = new Dog[ 3 ];
dogs[ 1 ] = new Dog( );
dogs[ 1 ].bark( );
dogs[ 0 ].bark( ); // Run-time error!
```

6.4 Arrays do Not Grow

The `length` attribute of a `Java` array is `final`, i.e. its value remains the same for the entire lifetime of the array. The only things that may change are the values of the things in the array. Since the size of arrays doesn't change, it should not come as a surprise that it is not allowed to assign values to ⟨`array`⟩`.length`: `length` is a `final` attribute.

The maximum size of any array is `0`. The maximum size of any array is given by the constant `Integer.MAX_VALUE`, which is the largest possible value of an `int`.

7 Life

This section is a case study in programming with arrays. The subject is a game which was developed by the British mathematician Arthur Conway as early as 1970see e.g. `http://en.wikipedia.org/wiki/Conway's_Game_of_Life`.

The rules of the $n \times n$ version of the game are surprisingly simple.

- Life is played on an $n \times n$ grid. A square on the grid is called a *cell*. A cell is either dead or alive.
- The game starts with an initial configuration of cells. The configuration can be either random or given.
- The initial configuration is called the initial generation g_0.
- The game continues by computing the generations g_1, g_2, and so on.
- Generation g_{i+1} is completely determined by generation g_i. This is done as follows.

  Underpopulation: A live cell with fewer than two live neighbours dies;

  Overcrowding: A live cell with more than three live neighbours dies;

  Reproduction: Dead cells with three live neighbours become live; and

  Invariance: All other cells remain the same.

  These rules are applied simultaneously to all cells of the grid, so the liveness of the cells in the next generation is completely determined by the liveness of the previous generation.

The remainder of the notes are a partial implementation. You will complete this implementation as Assignment 3.

The following is an outline of the class. The private attribute `cells` represents the cells of the game. A cell is alive if and only if its cell is `true`.

```Java
public class Life {
    private final boolean[][] cells;

    public static void main( String[] args ) {
        Life generation = new Life( 70 );
        for (int i = 0; i != 10000; i++) {
           System.out.println( generation );
           generation.next( );
        }
    }

    // Constructors

    // Public methods and helper methods.
}
```

The notation '`boolean[][]`' means 'array of array of `boolean`': a 2-dimensional array.

We shall implement two constructors. The first constructor is the default constructor. It creates a `Life` object by configuring it as a "Glider" pattern on a 6×6 grid. The remaining constructor takes a size of the grid and generates a random `Life` object of that size.

The following is the default constructor.

```Java
// Initialises 6 * 6 grid with Glider pattern.
public Life( ) {
    final int SIZE = 6;
    // Arguably, this should have been a class (static) array.
    final int[][] pairs = {{2,4},{3,3},{1,2},{2,2},{3,2}};
    cells = new boolean[ SIZE ][ ];
    for (int row = 0; row < SIZE; row ++) {
        cells[ row ] = new boolean[ SIZE ];
    }
    for (int pair = 0; pair < pairs.length; pair ++) {
        final int row = pairs[ pair ][ 0 ];
        final int col = pairs[ pair ][ 1 ];
        cells[ row ][ col ] = true;
    }
}
```

The main task of the constructor is to construct the array `cells` and initialise the values in the array. Since `cells` is a two-dimensional array of `boolean` the things at the top level are one-dimensional `boolean` arrays. For example, writing '`cells[ index ]`' corresponds to an array of `boolean`. The first thing the constructor does is create an array of array of `boolean`: `new boolean[ size ][ ]`. This makes sure that `cells` can hold `size` arrays of `boolean`. Next it continues by creating each of the `size`

boolean arrays and assign them to the cells in the top-level array: `cells[ row ] = new boolean[ SIZE ]`. Finally, each cell is initialised. This is done in the last `for` loop.

The following is the random constructor. The following implementation forces about half of the cells to be live. To see how this works, notice that that `rand.nextInt( 2 )` returns `0` or `1`, each with equal probability. Therefore the condition '`(rand.nextInt( 2 ) == 0)`[1] is `true` about half of the times.

```Java
// Initialise size * size grid with random cells.
public Life( int size ) {
    final Random rand = new Random( );
    cells = new boolean[ size ][ ];
    for (int row = 0; row < size; row ++) {
        cells[ row ] = new boolean[ size ];
        for (int col = 0; col < size; col ++) {
            cells[ row ][ col ] = (rand.nextInt( 2 ) == 0);
        }
    }
}
```

Remember that the method `toString( )` returns a printable representation of the object. The default implementation of the method is provided by the class `Object`. This implementation is not very useful. The nice thing about Java is that you can provide a more specific implementation for instances of the current class by *overriding* the method.

The following demonstrates how this is done. You start by writing the magic spell `@Override` and continue by writing the more specific definition of `toString( )`.

```Java
@Override
public String toString( ) {
    String result = "";
    for (int row = 0; row < cells.length; row ++) {
        final boolean[] column = cells[ row ];
        for (int col = 0; col < column.length; col ++) {
            result = result + (column[ col ] ? "x" : ".");
        }
        result = result + "\n";
    }
    return result;
}
```

8 For Friday

Study the notes and study Chapter 3.

[1] The parentheses are added for clarity: they are not needed.