Designing classes: Methods and properties

Lecture 5

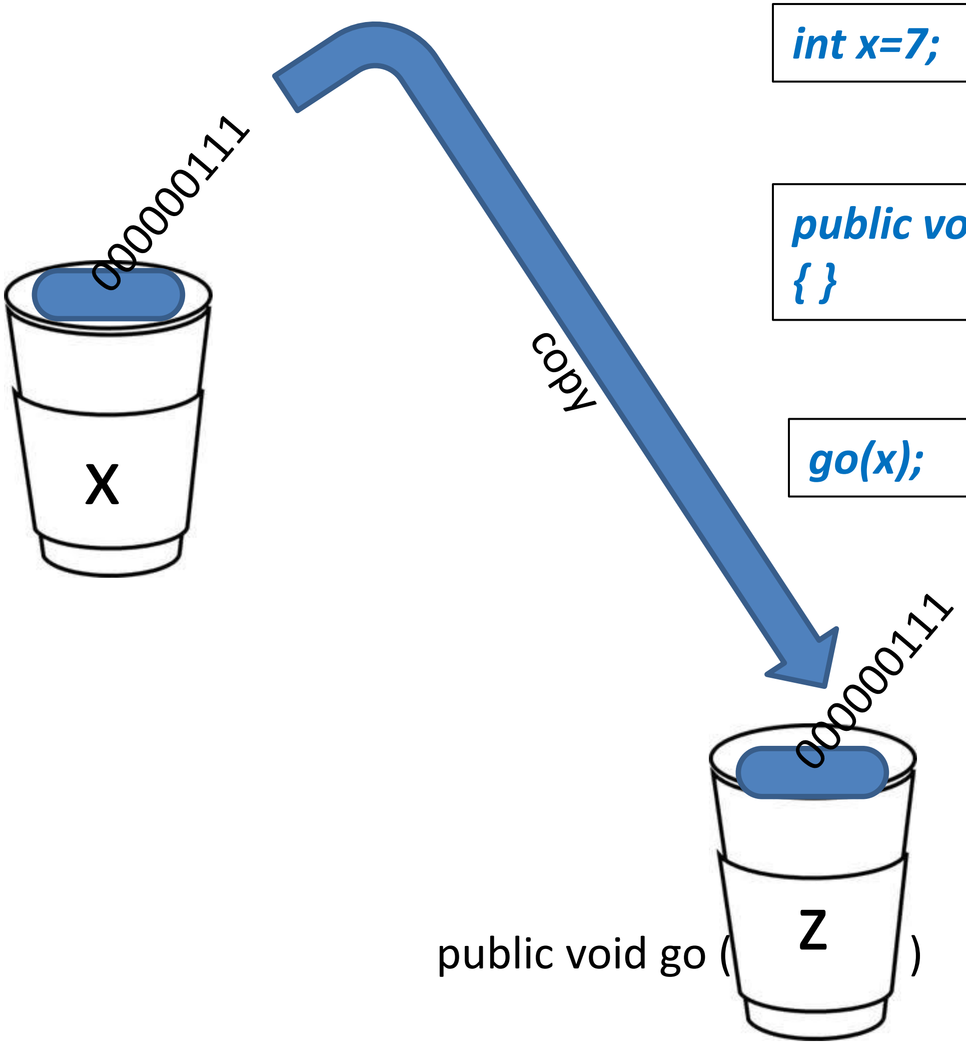
Objects have state and behavior

- State: Instance variables, fields or properties and their current values
- Behavior: methods

Method arguments and return values

- Each method may have >=0 parameters (arguments)
- Each method may have only 1 return value

Java is pass by value – this means pass by copy



```
int x=7;
```

1. Declare variable x, assign value 7.

```
public void go (int z)
{ }
```

2. Declare method go with its own variable for a method parameter

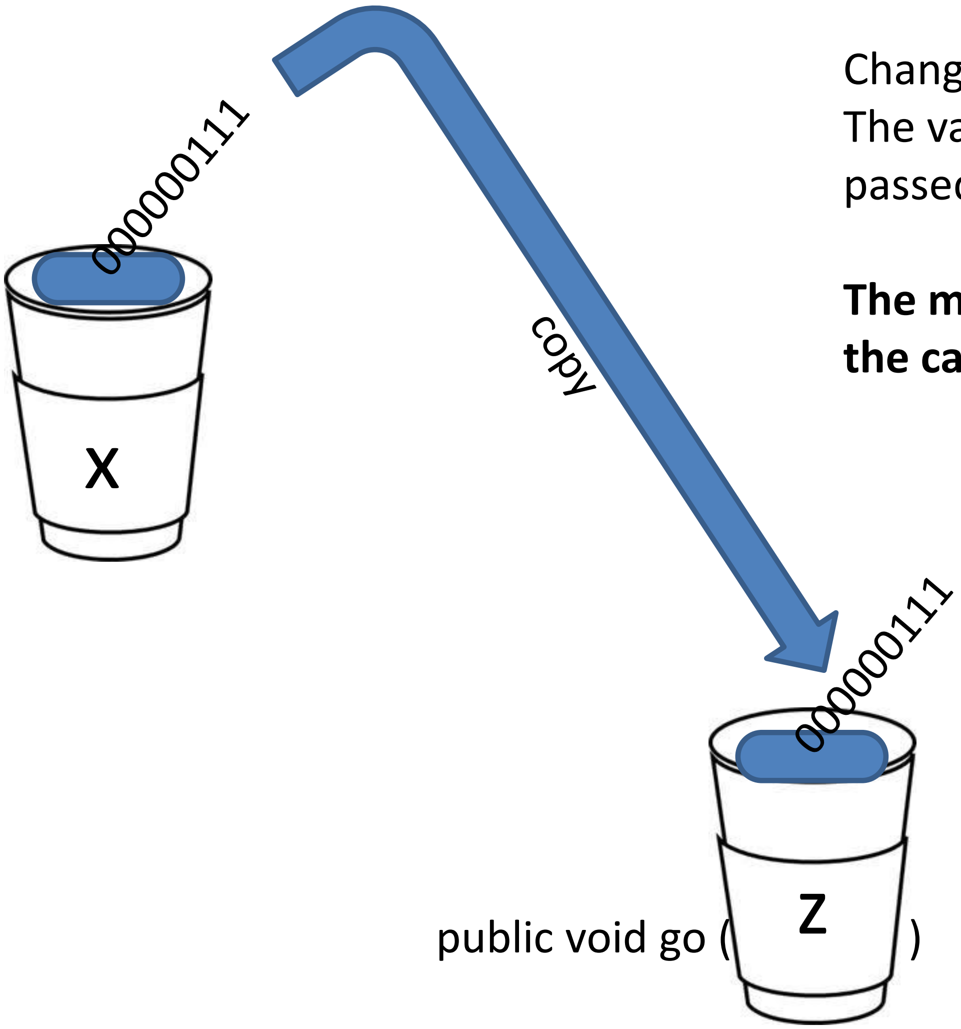
```
go(x);
```

3. Call method passing x as an argument – the bits are copied from x to z

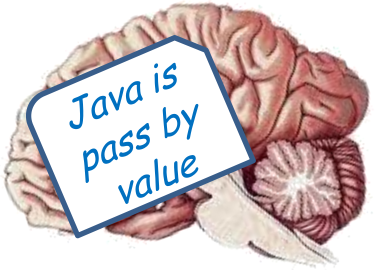
Java is pass by value
– this means pass by copy

Change the value of z inside the method.
The value of x doesn’t change! The argument passed to the z parameter was only a copy of x.

The method can’t change the bits that were in the calling variable x.



public void go ( )



Make it stick

Roses are red,
this poem is choppy,
passing by value
is passing by copy

What happens with arguments-objects?

- Pass by value
- Value is bits inside the variable

- Bits in the reference variable are the remote control (address?) of an object.
- When they are copied into a method argument, we are pointing to the same object, and thus we are changing the same object

If we need to change int value

Pass a wrapper class

```
int c1, c2;
Integer oCounter1=new Integer(c1);
Integer oCounter2=new Integer(c2);
incrementAllCounters(oCounter1, oCounter2)

c1=oCounter1.intValue();
c2=oCounter2.intValue();

public void incrementAllCounters (Integer counter1, Integer counter2)
{
        counter1.intValue++;
        counter2.intValue++;
}
```

Passing *this* as an argument

```
class Person {
        public void eat(Apple apple) {
                Apple peeled = apple.getPeeled();
                System.out.println("Yummy");
        }
}

class Peeler {
        static Apple peel(Apple apple) {
                // ... remove peel
                return apple; // Peeled
        }
}

class Apple {
        Apple getPeeled() { return Peeler.peel(this); }
}
```

```
public class PassingThis {
public static void main(String[] args) {
    new Person().eat(new Apple());
  }
} /* Output:
Yummy
```

Passing *this* to create an association

- Usually associations are done in the constructor

```
package Demos.Car;

/**
 * This class models a CSCI331Mobile that knows about
 * its City. Again, the instance variables,
 * constructor, and other methods that we defined
 * in earlier slides are elided.
 */

public class CSCI331Mobile {
  private City _city;

  public CS15Mobile(City myCity) {
    _city = myCity; // store association
    // More code elided
  }
}
```

Now the `CSCI331Mobile` can call any of `City`'s public methods on `_city`.

Syntax: City

```
package Demos.Car;

/**
 * This class models a city where CSCI331Mobiles
 * exist. Because the City contains the
 * CSCI331Mobile, it can send the CSCI331Mobile the
 * reference to an instance of itself.
 */

public class City {

  private CSCI1331Mobile _331mobile;

  public City() {
    _331mobile = new CSCI1331Mobile (this);
  }
  // … Other methods of City elided
} // End of class City
```

Method overloading

- Method overloading is having two methods with the same name but different lists of parameters

- There is no operator overloading in Java

```
public class Overloads {
        String uniqueID;
        public int addNums(int a, int b) {
                return a + b;
        }

        public double addNums(double a, double b) {
                return a + b;
        }

        public void setUniqueID(String theID) {
                // lots of validation code, and then:
                uniqueID = theID;
        }
        public void setUniqueID(int ssNumber) {
                String numString = “” + ssNumber;
                setUniqueID(numString);
        }
}
```

Overloading on return values

Illegal

```
void f();
int f() { return 1; }
```

Calling overriden constructor from within constructor

```
public class Flower {
        int _petalCount = 0;
        String _name = "No name";
        Flower(int petalCount) {
                _petalCount = petalCount;
                System.out.println("Created flower "+ _name+" with "+_petalCount+" petals");
        }
        Flower(String name) {
                this();
                _name = name;
                System.out.println("Created flower "+ _name+" with "+_petalCount+" petals");
        }
        Flower(String name, int petalCount) {
                this (petalCount);
                //! this(name); // Can’t call two!
                this._name = name; // Another use of "this"
                System.out.println("Created flower "+ _name+" with "+_petalCount+" petals");
        }
        Flower() {
                this ("Artificial flower", 2);
                System.out.println("Created flower "+ _name+" with "+_petalCount+" petals");
        }
}
```

What is printed?

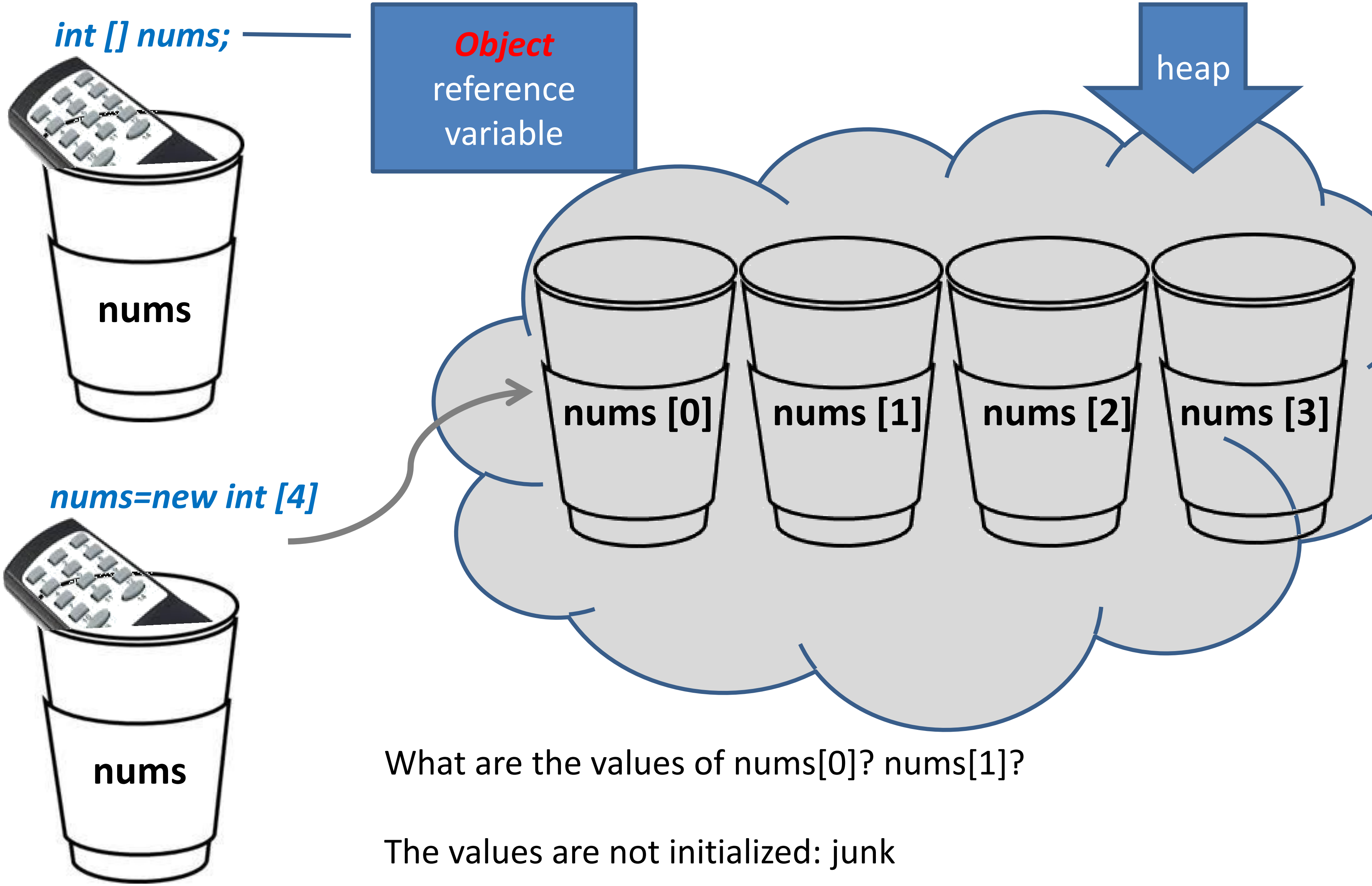
```
Flower f=new Flower(“Rose”);
Flower f=new Flower( 5));
Flower f=new Flower(“Rosa glauca”, 5));
```

Rosa glauca

Method return values

- Only one return value
- If need more – return array, return object

Java arrays: array of primitives



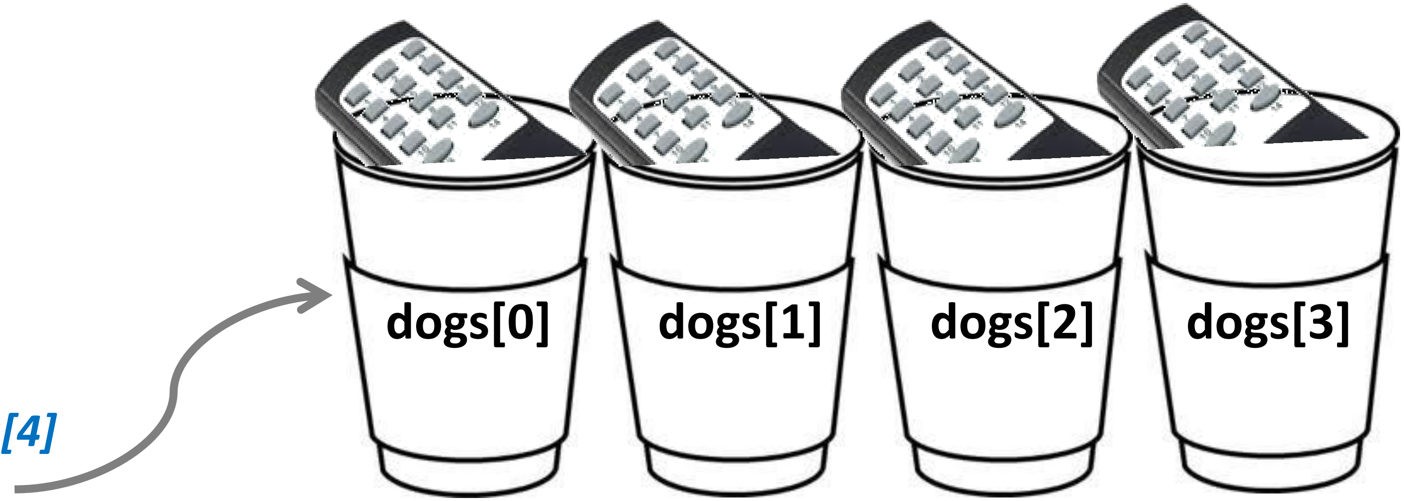
What are the values of nums[0]? nums[1]?

The values are not initialized: junk

Java arrays: array of objects

Dog [] dogs;

dogs=new Dog[4]



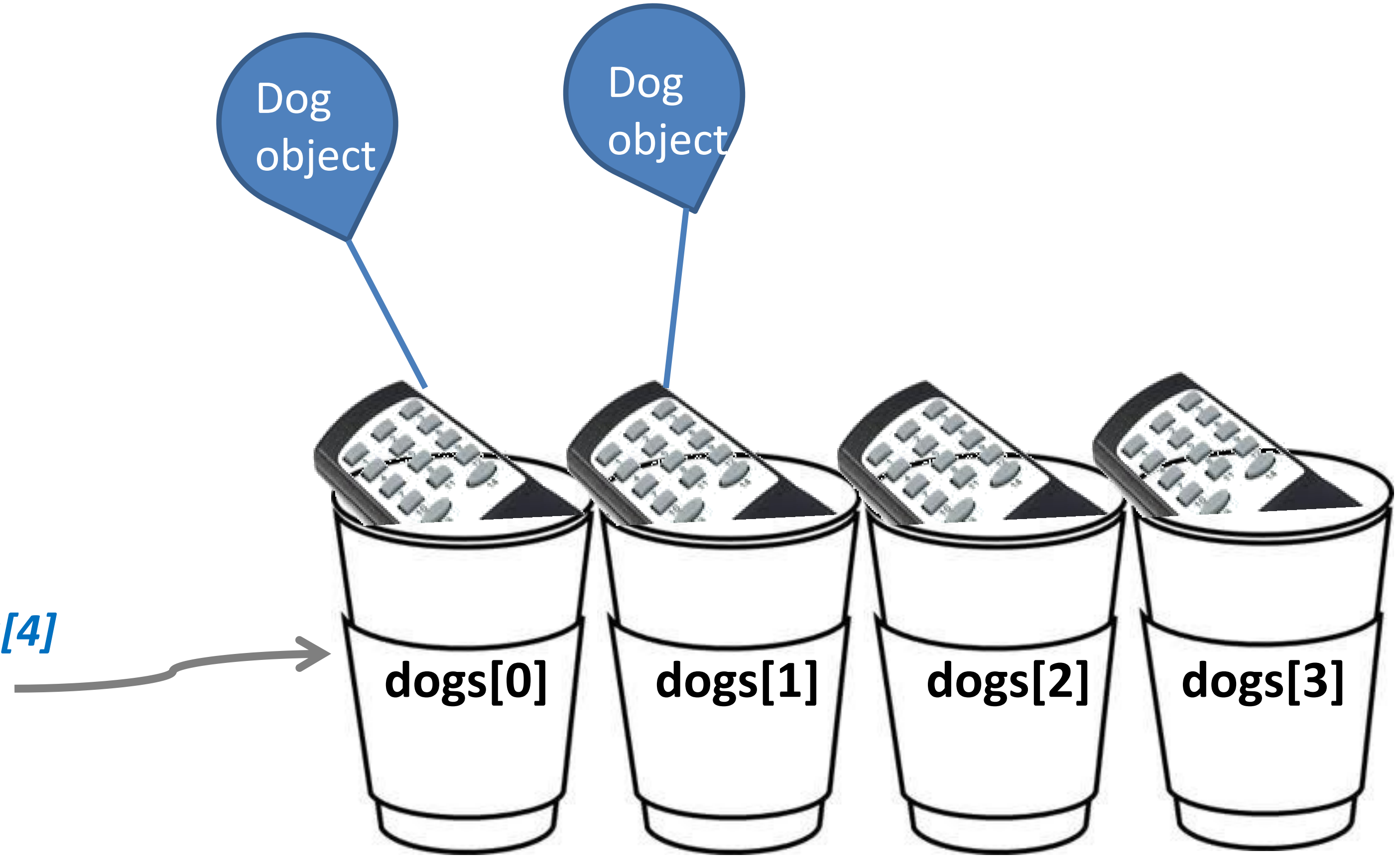
Can we call methods of dogs[0]?
What's missing?

Dogs!
We have an array of references but no actual Dog objects

Java arrays: array of objects

```
Dog [] dogs;
```

```
dogs=new Dog[4]
```



```
dogs[0]=new Dog;
dogs[1]=new Dog;
```

Initialize each reference with new Dog object

Instance variables: initialization

If initial state is not set in the constructor, all instance variables are automatically initialized to their default values:

- Numeric primitives – zero
- Boolean primitive – false
- String and other objects - null

Local variables

- Local, stack-variables, scope-challenged variables
- Their life is short – inside the curly brackets of the method
- The objects created inside the method and referenced by a local variable are destroyed when the method execution ends
- Local variables are not automatically initialized, but their initialization is enforced by a compiler

Does not compile

```
class Foo {
        public void go() {
                int x;
                int z = x + 3;
        }
}
```

```
% javac Foo.java
Foo.java:4: variable x might
not have been initialized
int z = x + 3;
1 error ^
```

Variable lifespan

- The life of an object depends on the life of the reference variable controlling it.
- But what is the lifespan of a reference variable?

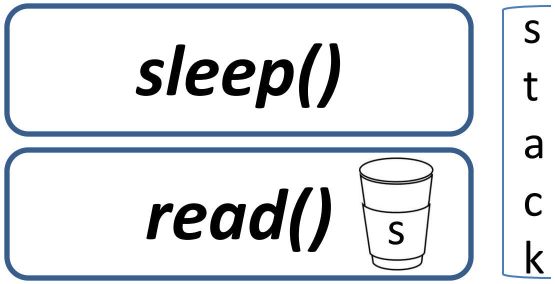
Variable scope

```
public class Student {
        public void read() {
                int s = 42;
                sleep();
        }

        public void sleep() {
                s = 7;
        }
}
```

Does not compile

sleep() cannot see variable ***s***. Since it is not in its own stack frame, sleep() does not know anything about it



Is ***s*** still alive when the program is performing ***sleep()*** method?

Yes, when ***sleep()*** completes and ***read()*** is on the top of the stack, it still can access the value of ***s***

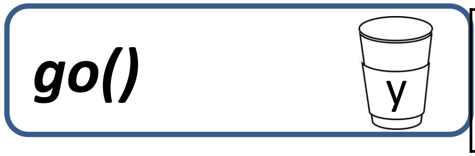
When ***read()*** completes and is popped off the stack, ***s*** is dead

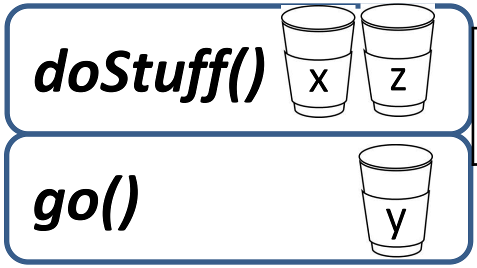
Life and scope

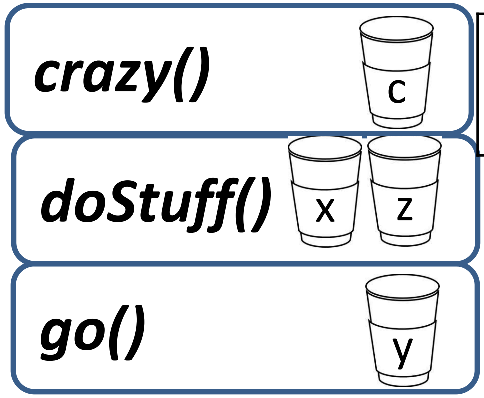
```
public void go() {
            int y = 3;
            doStuff(y);
}

public void doStuff (int x) {
            int z = x + 24;
            crazy();
            // imagine more code here
}

public void crazy() {
            char c = ‘a’;
}
```







alive	In scope	alive	In scope	alive	In scope	alive	In scope
y		x		z		c	
v	v	-	-	-	-	-	-
v	-	v	v	v	v		
v	-	v	-	v	-	v	v

What about reference variables?

An object becomes eligible for GC when its last live reference disappears. If you do not release your objects, you will run out of memory

3 ways to release your object

1. The reference goes out of scope, permanently

```
void go() {
        Life z = new Life();
}
```

2. The reference is assigned another object

```
Life z = new Life();
z = new Life();
```

3. The reference is explicitly set to null

```
Life z = new Life();
z = null;
```

Exercise

```
public class GC {
        public static GC doStuff() {
                GC newGC = new GC();
                doStuff2(newGC);
                return newGC;
        }

        public static void main(String [] args) {
                GC gc1;
                GC gc2 = new GC();
                GC gc3 = new GC();
                GC gc4 = gc3;
                gc1 = doStuff();
        ★

                // call more methods
        }

public static void doStuff2(GC copyGC) {
                GC localGC = copyGC;
        }
}
```

How many total GC objects were allocated in this program? 3

How many references? 6

Which of the following lines will release exactly one additional object when inserted in place of star?

1. ***copyGC = null;***
2. ▶ ***gc2 = null;***
3. ***newGC = gc3;***
4. ▶ ***gc1 = null;***
5. ***newGC = null;***
6. ***gc4 = null;***
7. ***gc3 = gc2;***
8. ▶ ***gc1 = gc4;***
9. ***gc3 = null;***