Department of Computer Science
University of Bristol

COMSM0086 – Object-Oriented Programming

Object's Life

Sion Hannuna | sh1670@bris.ac.uk
Simon Lock | simon.lock@bris.ac.uk

"From a very early age, we form concepts. Each concept is a particular idea or understanding we have about our world. These concepts allow us to make sense of and reason about the things in our world.

These things to which our concepts apply are called objects."

James Martin

Recap: Objects

(...and what they are made of...)

Recap: Attributes and Methods

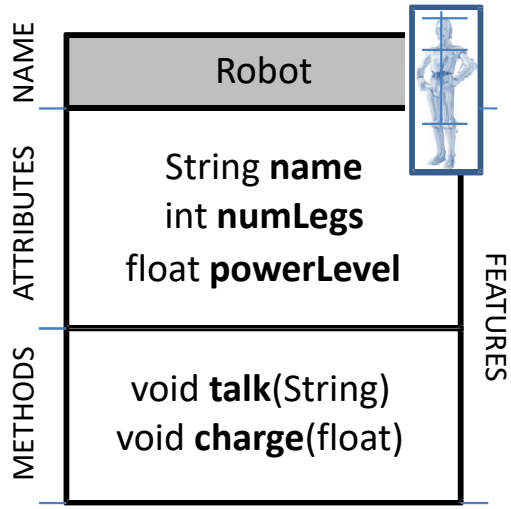
CONVENTION: method names are commonly verbs, attribute names are usually singular nouns (but plural if the attribute is a collection), or adjectives for boolean values

Attributes capture what objects can **be**.

- each object has its own copy of its attributes
- attributes can be plain datatypes:
  bool **char** **int** **float** **double**
- however, attributes can be more complex; they may even be references to other objects...

Methods capture what objects can **do**.

- methods take parameters
- methods return values
- methods can be *overloaded* (same method name, different parameters)
- methods are the main way to communicate with an object
- methods are, thus, sometimes called `messages'

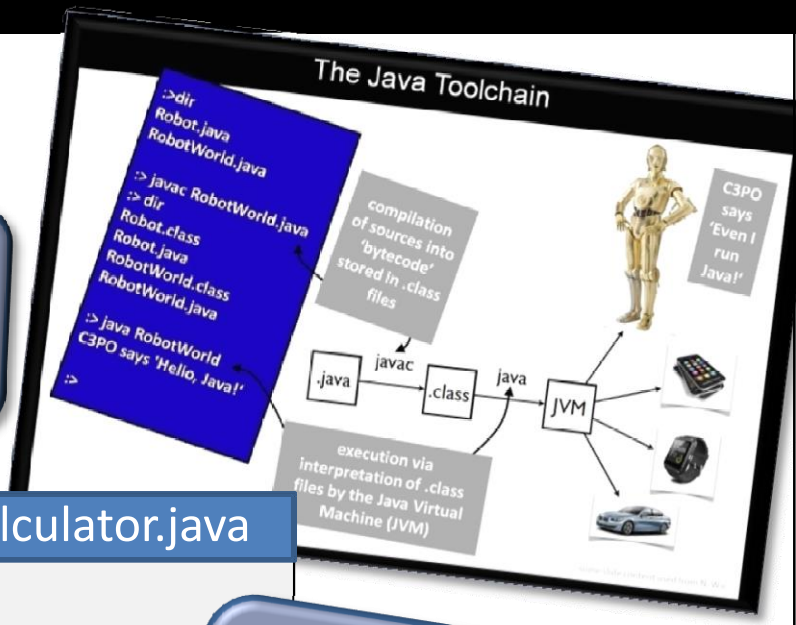


Recap: Construction, Instantiation, Exceptions, Toolchain

Adder.java

```
class Adder {
  int sum;

  Adder() {
    ...
  }
  void add(int summand) {
    sum += summand;
  }
}
```

Constructor can initialise an object's attributes



ExceptionalCalculator.java

```
public class ExceptionalCalculator {

  public static void main (String[] args) {
    Adder adder = new Adder();
    try {
      for (String arg : args) {
            adder.add(Integer.parseInt(arg));
      }
      System.out.println("Sum:"  + adder.sum);
    } catch (Exception e) {
      System.out.println("Something went wrong, but I can handle it!");
} } }
```

Instantiation generates a new object instance

- We know how to write classes and instantiate objects – today, we want to investigate the life of objects after their creation...

Exceptional Joke

Why does Yoda’s code always crash?

Exceptional Joke

Why does Yoda's code
always crash?

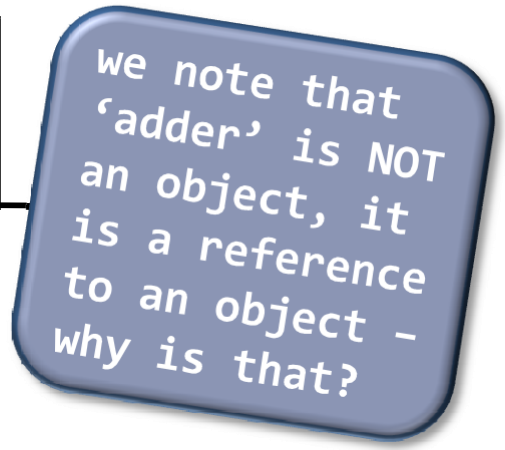
… there is no try.

References

(Philosophers may call them Signifiers)

References (standing in for the object itself)

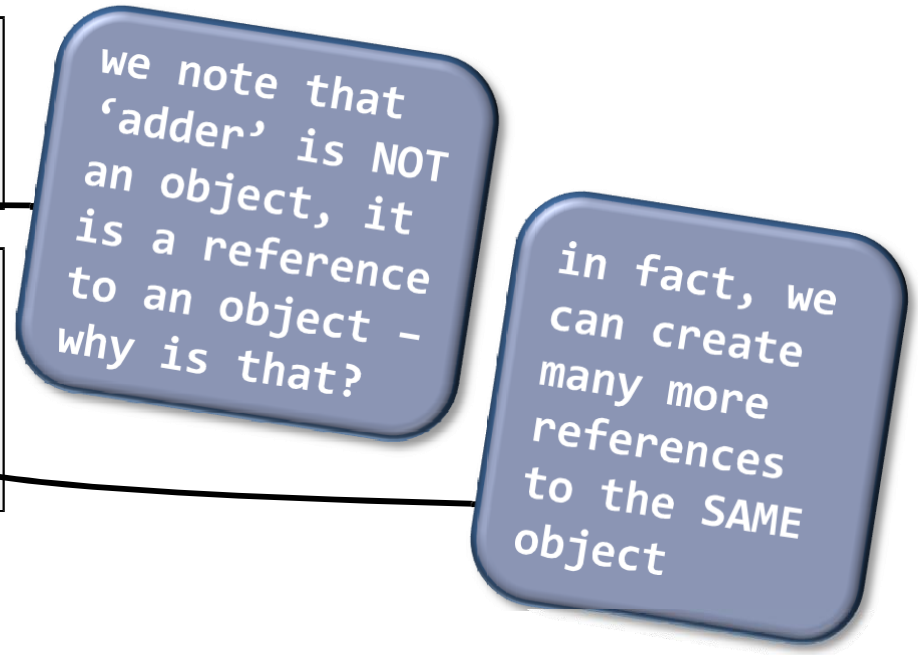
```
...
   Adder adder = new Adder();
...
```



References (standing in for the object itself)

```
...
   Adder adder = new Adder();
...
```

```
...
   Adder adder = new Adder();
   Adder adder2  = adder;
...
```



References (standing in for the object itself)

```
...
   Adder adder = new Adder();
...
```

we note that 'adder' is NOT an object, it is a reference to an object – why is that?

```
...
   Adder adder = new Adder();
   Adder adder2  = adder;
...
```

in fact, we can create many more references to the SAME object

...here the philosopher would step in and may say that references are really 'place holders' that signify the presence and/or usage of an object...

References (standing in for the object itself)

```
...
   Adder adder = new Adder();
...
```

we note that 'adder' is NOT an object, it is a reference to an object – why is that?

```
...
   Adder adder = new Adder();
   Adder adder2  = adder;
...
```

in fact, we can create many more references to the SAME object

...here the philosopher would step in and may say that references are really 'place holders' that signify the presence and/or usage of an object...

```
...
   Adder adder = null;
   adder.add(1); //oh no!
...
```

unfortunately, we are allowed to create a reference to nothing...

...at this point the philosopher may declare the computer scientists to be out of their mind...

Null References

"I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years."

— *Tony Hoare*

slide designed by N. Wu

The Heap

(...or where the wild objects live)

Stack vs the Heap

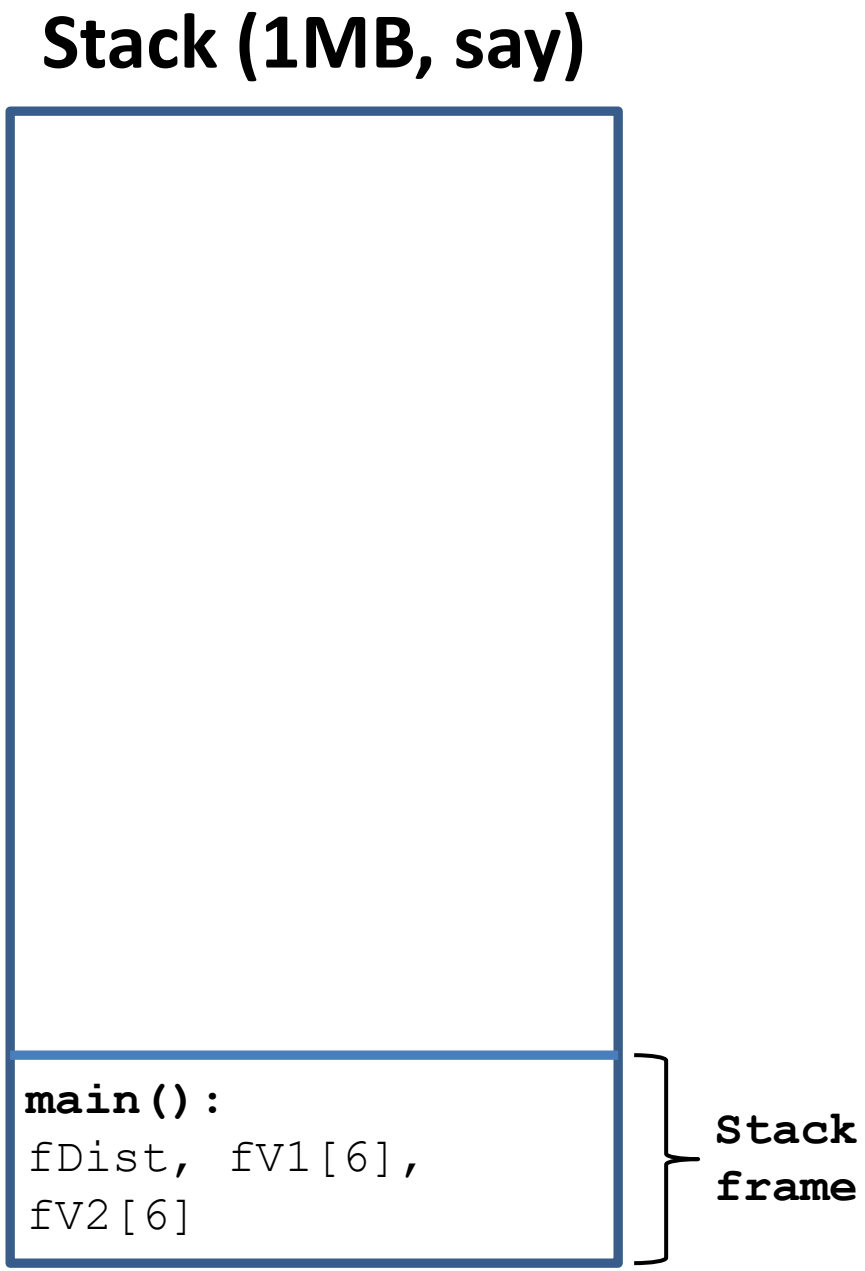
The stack is the memory set aside as scratch space for a thread of execution. When a function is called, a block is reserved on the top of the stack for local variables and some bookkeeping data. When that function returns, the block becomes unused and can be used the next time a function is called. The stack is always reserved in a LIFO (last in first out) order; the most recently reserved block is always the next block to be freed. This makes it really simple to keep track of the stack; freeing a block from the stack is nothing more than adjusting one pointer.

The heap is memory set aside for dynamic allocation. Unlike the stack, there's no enforced pattern to the allocation and deallocation of blocks from the heap; you can allocate a block at any time and free it at any time. This makes it much more complex to keep track of which parts of the heap are allocated or free at any given time; there are many custom heap allocators available to tune heap performance for different usage patterns.

Source: https://stackoverflow.com/users/14742/jeff-hill

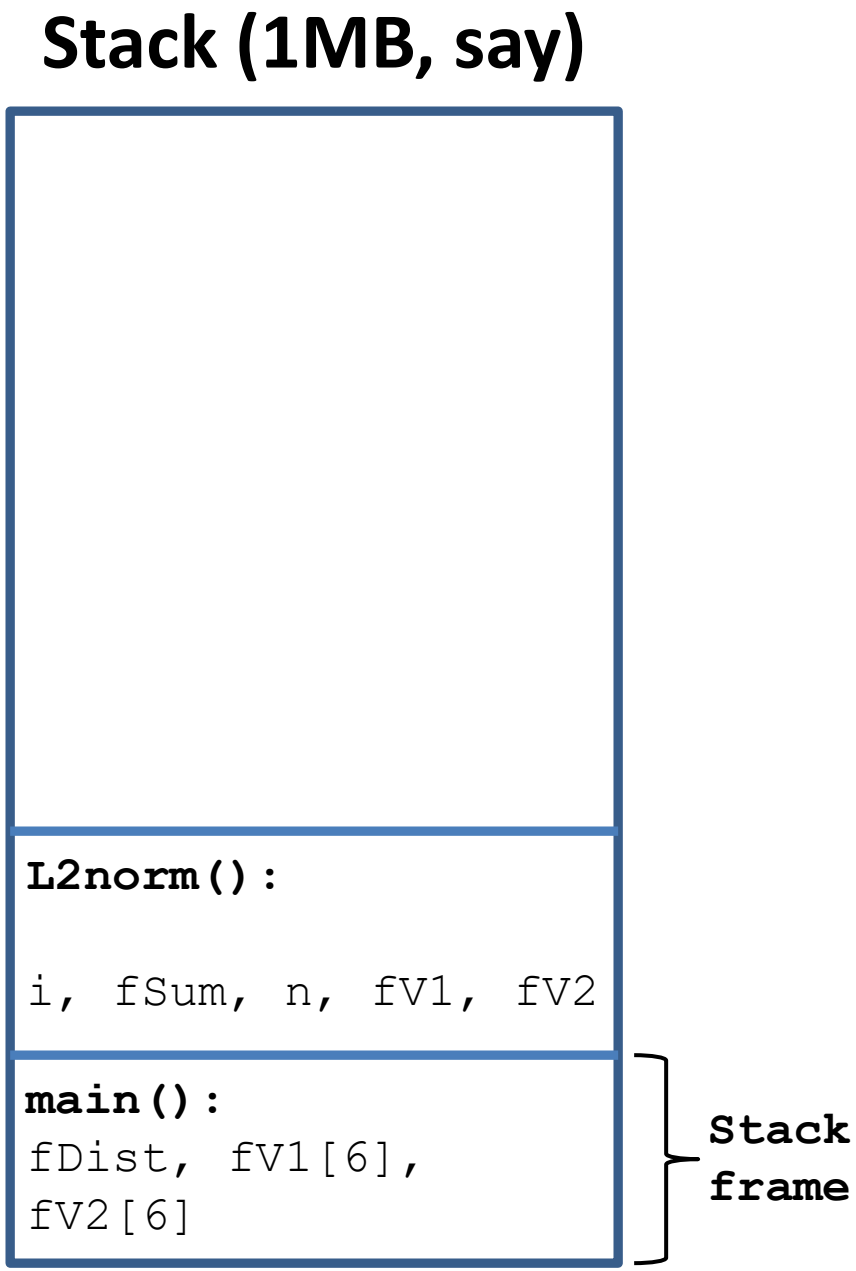
The Heap and the Stack

```
#include "stdafx.h"
#include <math.h>

float squaredDiff(float f1, float f2)
{
    float fSquareDiff;

    fSquareDiff = (f1 - f2) * (f1 - f2);

    return fSquareDiff;
}

float L2norm(float fV1[], float fV2[], int n)
{
    int i;
    float fSum = 0;

    for (i = 0; i < n; ++i)
        fSum += squaredDiff(fV1[i], fV2[i]);

    return (float)sqrt((double)fSum);
}


void main(void)
{
    float fDist;
    float fV1[] = { 4,10,12,23,56,87 };
    float fV2[] = { 14,1,22,23,66,97 };

    fDist = L2norm(fV1, fV2, 6);
    printf("L2 norm  = %0.2f\n", fDist);
}
```



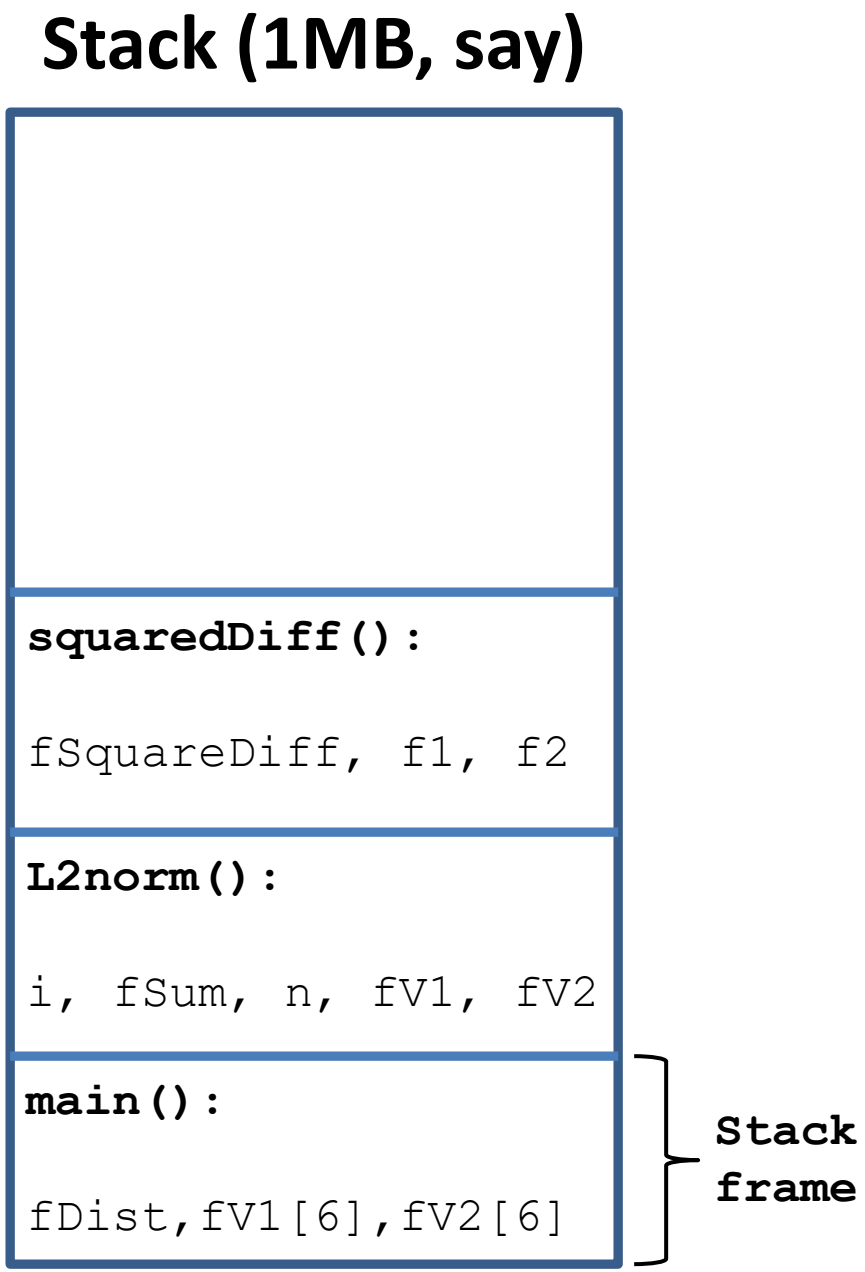
The Heap and the Stack

```
#include "stdafx.h"
#include <math.h>

float squaredDiff(float f1, float f2)
{
    float fSquareDiff;

    fSquareDiff = (f1 - f2) * (f1 - f2);

    return fSquareDiff;
}

float L2norm(float fV1[], float fV2[], int n)
{
    int i;
    float fSum = 0;

    for (i = 0; i < n; ++i)
        fSum += squaredDiff(fV1[i], fV2[i]);

    return (float)sqrt((double)fSum);
}


void main(void)
{
    float fDist;
    float fV1[] = { 4,10,12,23,56,87 };
    float fV2[] = { 14,1,22,23,66,97 };

    fDist = L2norm(fV1, fV2, 6);
    printf("L2 norm  = %0.2f\n", fDist);
}
```



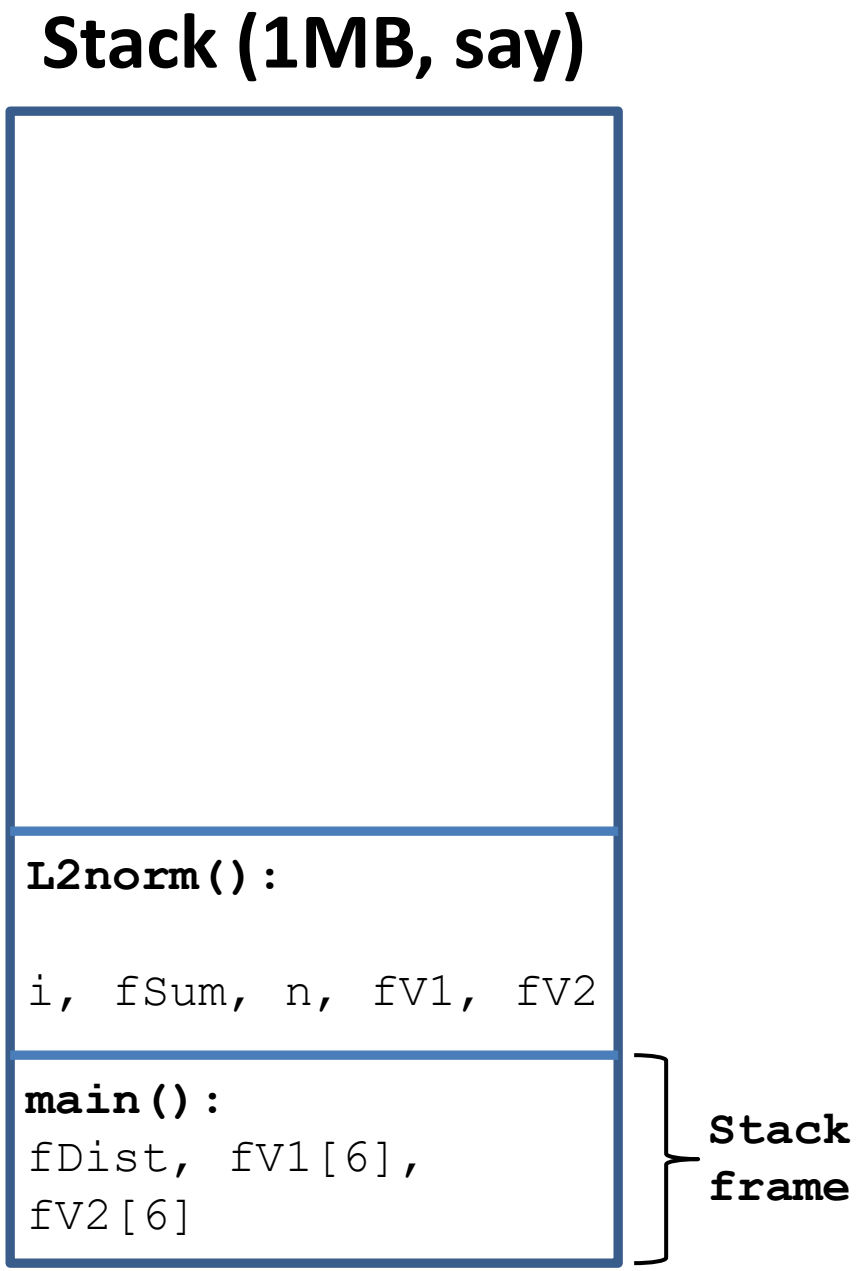
The Heap and the Stack

```
#include "stdafx.h"
#include <math.h>

float squaredDiff(float f1, float f2)
{
    float fSquareDiff;

    fSquareDiff = (f1 - f2) * (f1 - f2);

    return fSquareDiff;
}

float L2norm(float fV1[], float fV2[], int n)
{
    int i;
    float fSum = 0;

    for (i = 0; i < n; ++i)
        fSum += squaredDiff(fV1[i], fV2[i]);

    return (float)sqrt((double)fSum);
}


void main(void)
{
    float fDist;
    float fV1[] = { 4,10,12,23,56,87 };
    float fV2[] = { 14,1,22,23,66,97 };

    fDist = L2norm(fV1, fV2, 6);
    printf("L2 norm  = %0.2f\n", fDist);
}
```



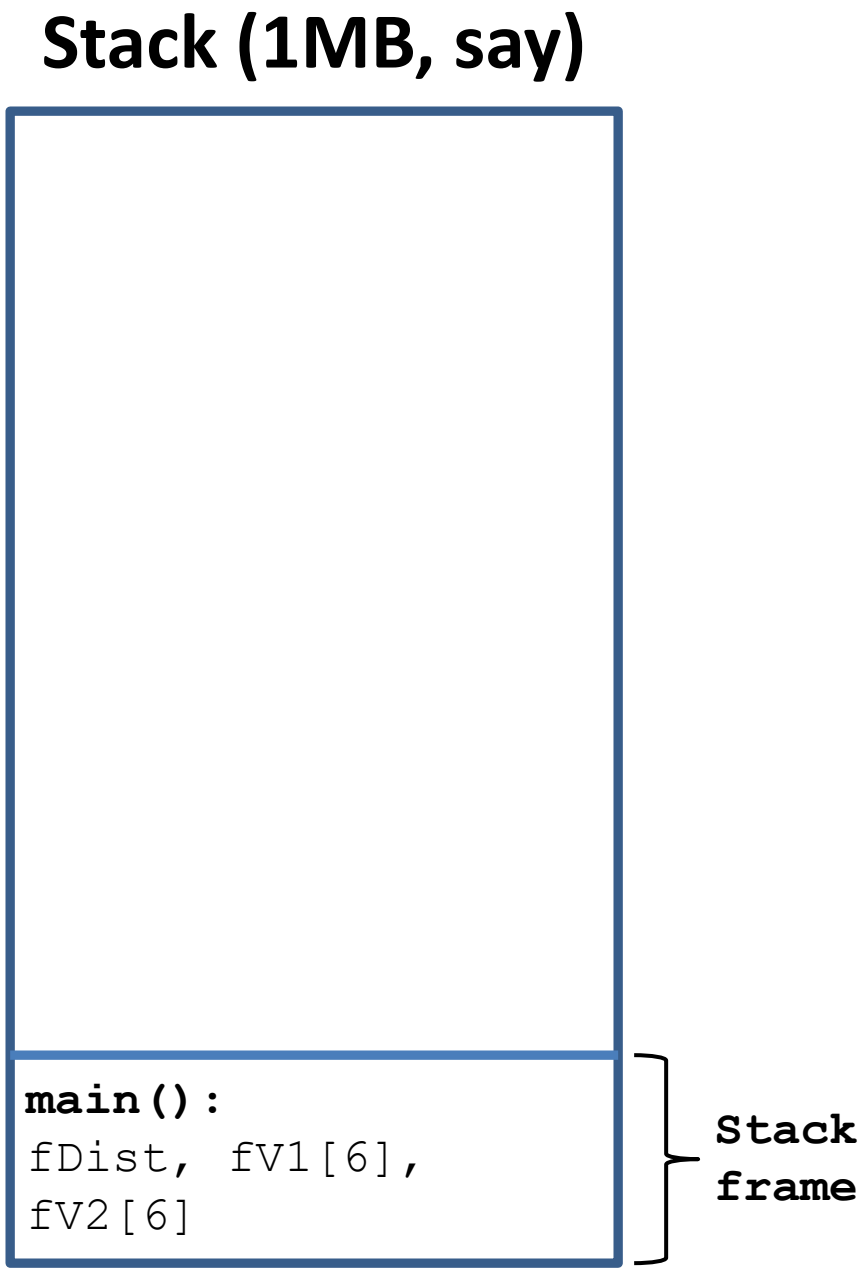
The Heap and the Stack

```
#include "stdafx.h"
#include <math.h>

float squaredDiff(float f1, float f2)
{
    float fSquareDiff;

    fSquareDiff = (f1 - f2) * (f1 - f2);

    return fSquareDiff;
}

float L2norm(float fV1[], float fV2[], int n)
{
    int i;
    float fSum = 0;

    for (i = 0; i < n; ++i)
        fSum += squaredDiff(fV1[i], fV2[i]);

    return (float)sqrt((double)fSum);
}


void main(void)
{
    float fDist;
    float fV1[] = { 4,10,12,23,56,87 };
    float fV2[] = { 14,1,22,23,66,97 };

    fDist = L2norm(fV1, fV2, 6);
    printf("L2 norm  = %0.2f\n", fDist);
}
```



The Heap and the Stack

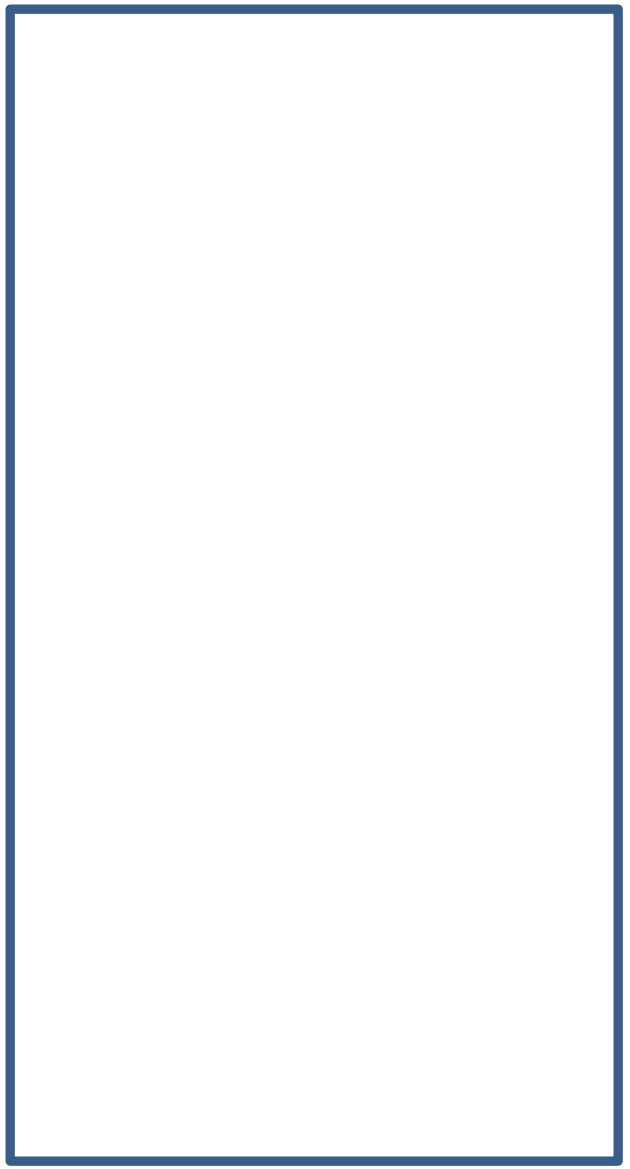
```
#include "stdafx.h"
#include <math.h>

float squaredDiff(float f1, float f2)
{
    float fSquareDiff;

    fSquareDiff = (f1 - f2) * (f1 - f2);

    return fSquareDiff;
}

float L2norm(float fV1[], float fV2[], int n)
{
    int i;
    float fSum = 0;

    for (i = 0; i < n; ++i)
        fSum += squaredDiff(fV1[i], fV2[i]);

    return (float)sqrt((double)fSum);
}


void main(void)
{
    float fDist;
    float fV1[] = { 4,10,12,23,56,87 };
    float fV2[] = { 14,1,22,23,66,97 };

    fDist = L2norm(fV1, fV2, 6);
    printf("L2 norm  = %0.2f\n", fDist);
}
```



The Heap and the Stack

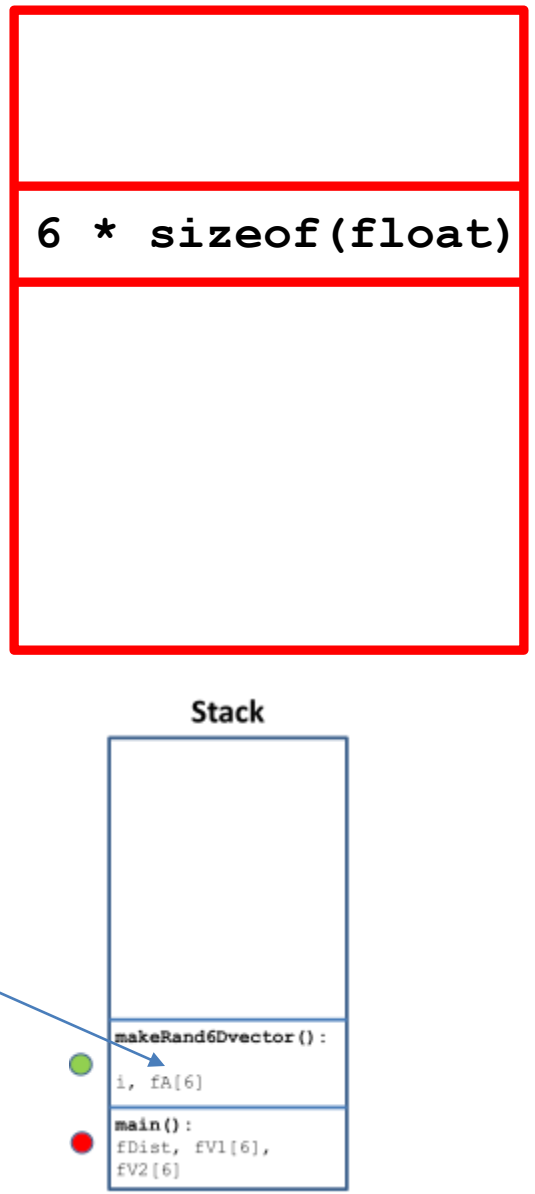
```
#include "stdafx.h"
#include <math.h>

float squaredDiff(float f1, float f2)
{
    float fSquareDiff;

    fSquareDiff = (f1 - f2) * (f1 - f2);

    return fSquareDiff;
}

float L2norm(float fV1[], float fV2[], int n)
{
    int i;
    float fSum = 0;

    for (i = 0; i < n; ++i)
        fSum += squaredDiff(fV1[i], fV2[i]);

    return (float)sqrt((double)fSum);
}


void main(void)
{
    float fDist;
    float fV1[] = { 4,10,12,23,56,87 };
    float fV2[] = { 14,1,22,23,66,97 };

    fDist = L2norm(fV1, fV2, 6);
    printf("L2 norm  = %0.2f\n", fDist);
}
```

Stack (1MB, say)



The Heap and the Stack

```
...

float * makeRand6DvectorRIGHT()
{
    int i;
    float * fA = (float*)malloc(sizeof(float)*6);

    for (i = 0; i < 6; ++i)
        fA[i] = (float)(rand() % 100);

    return fA;
}

float * makeRand6DvectorWRONG()
{
    int i;
    float fA[6];

    for (i = 0; i < 6; ++i)
        fA[i] = (float)(rand() % 100);

    return fA;
}

void main(void)
{
    float fDist;
    srand(time(NULL));
    float * fV1 = makeRand6DvectorRIGHT();
    float * fV2 = makeRand6DvectorWRONG();

    fDist = L2norm(fV1, fV2, 6);
    printf("L2 norm  = %0.2f\n", fDist);
}
```



Why does dynamic allocation succeed here?

```c
void main(void)
{
    int arr[1000000]; // Doesn't work
    int * arr = (int*)malloc(sizeof(int) * 1000000); //Works
}
```

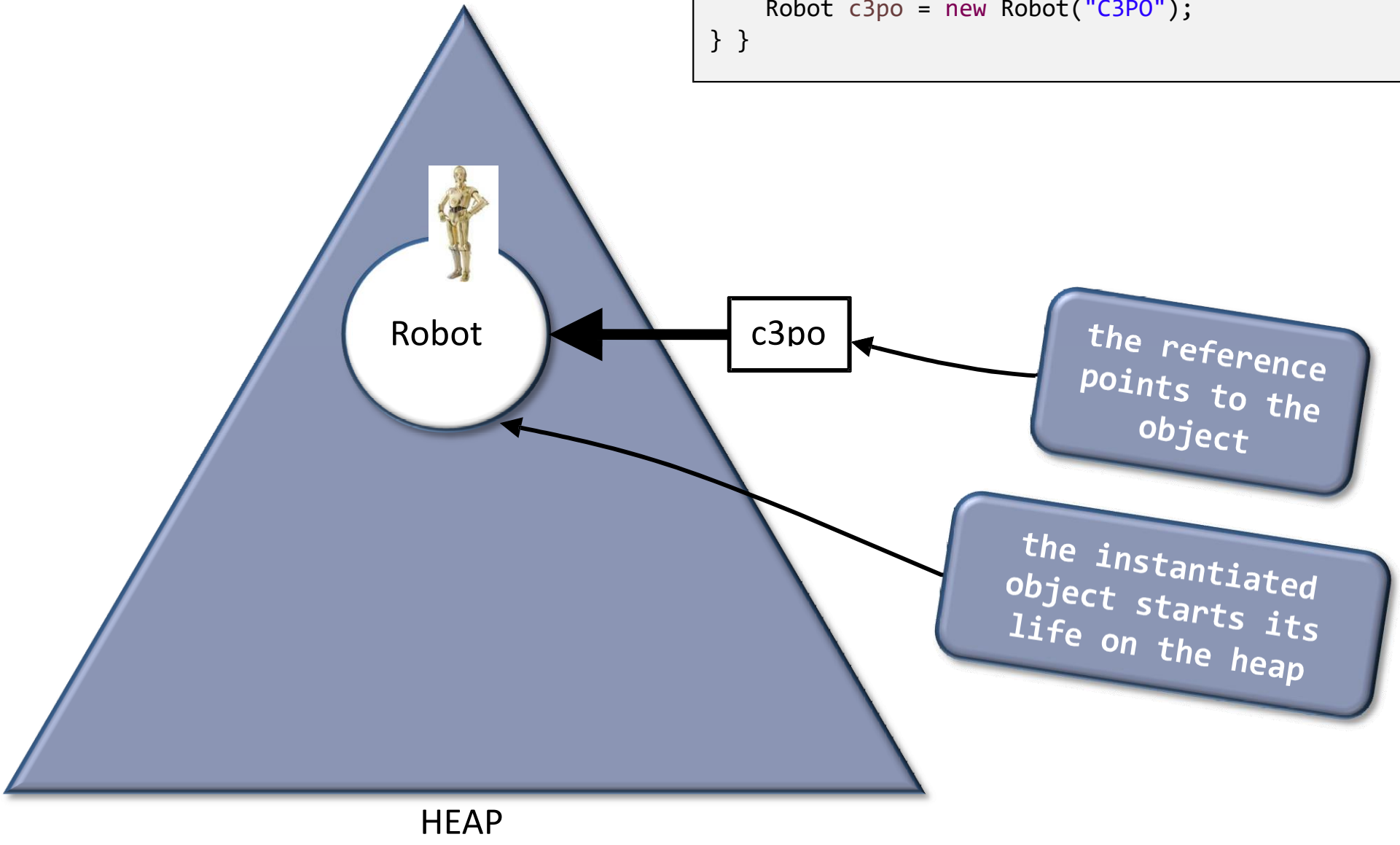
Why does dynamic allocation succeed here?

```
void main(void)
{
    int arr[1000000]; // Doesn't work
    int * arr = (int*)malloc(sizeof(int) * 1000000); //Works
}
```

In the previous case a stack overflow occurred

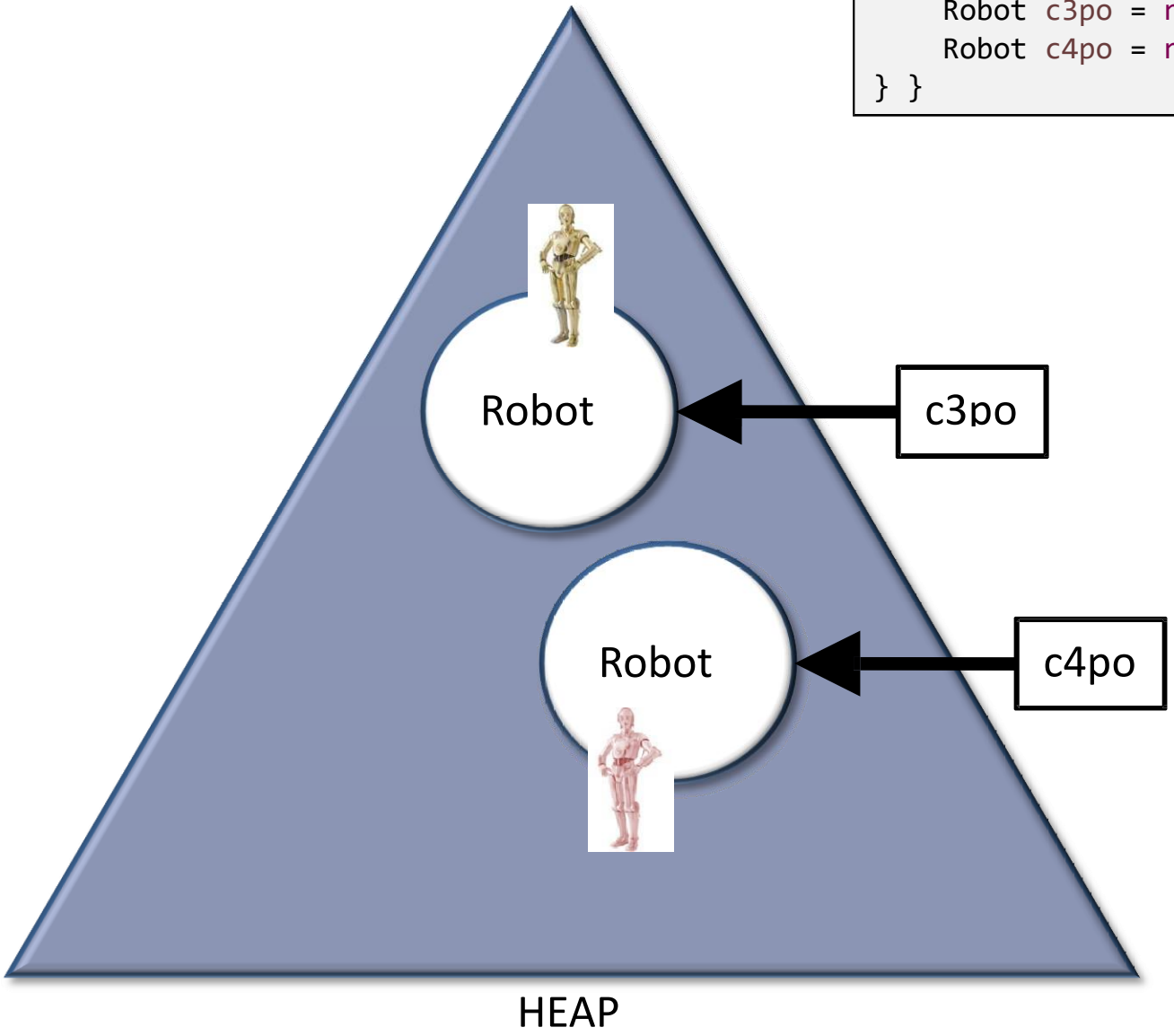
The Lifecycle of Objects

```
class RobotHeap {
  public static void main (String[] args) {
    Robot c3po = new Robot("C3PO");
} }
```



The Lifecycle of Objects

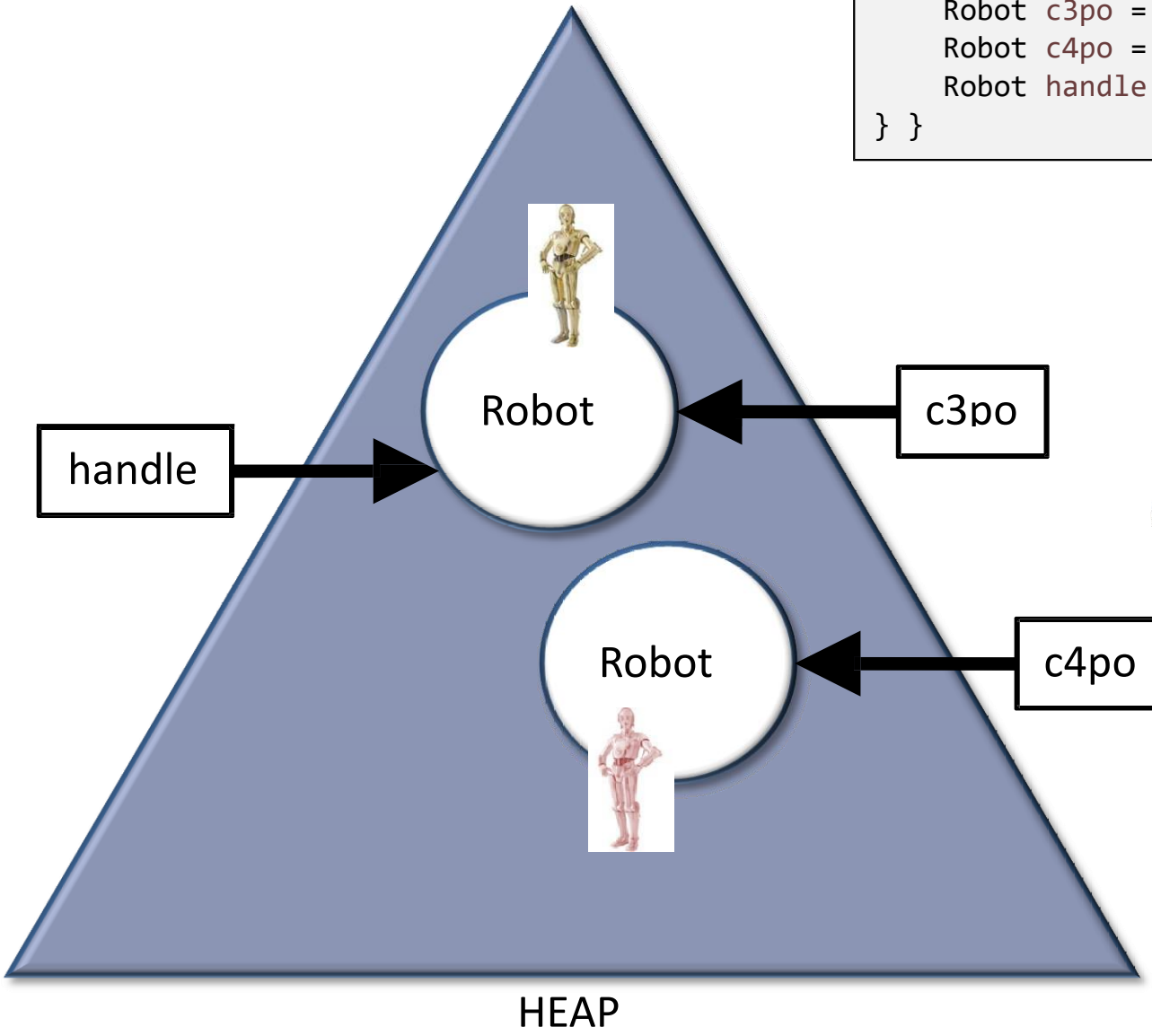
```
class RobotHeap {
  public static void main (String[] args) {
    Robot c3po = new Robot("C3PO");
    Robot c4po = new Robot("C4PO");
} }
```



another instantiated object starts its life on the heap

The Lifecycle of Objects

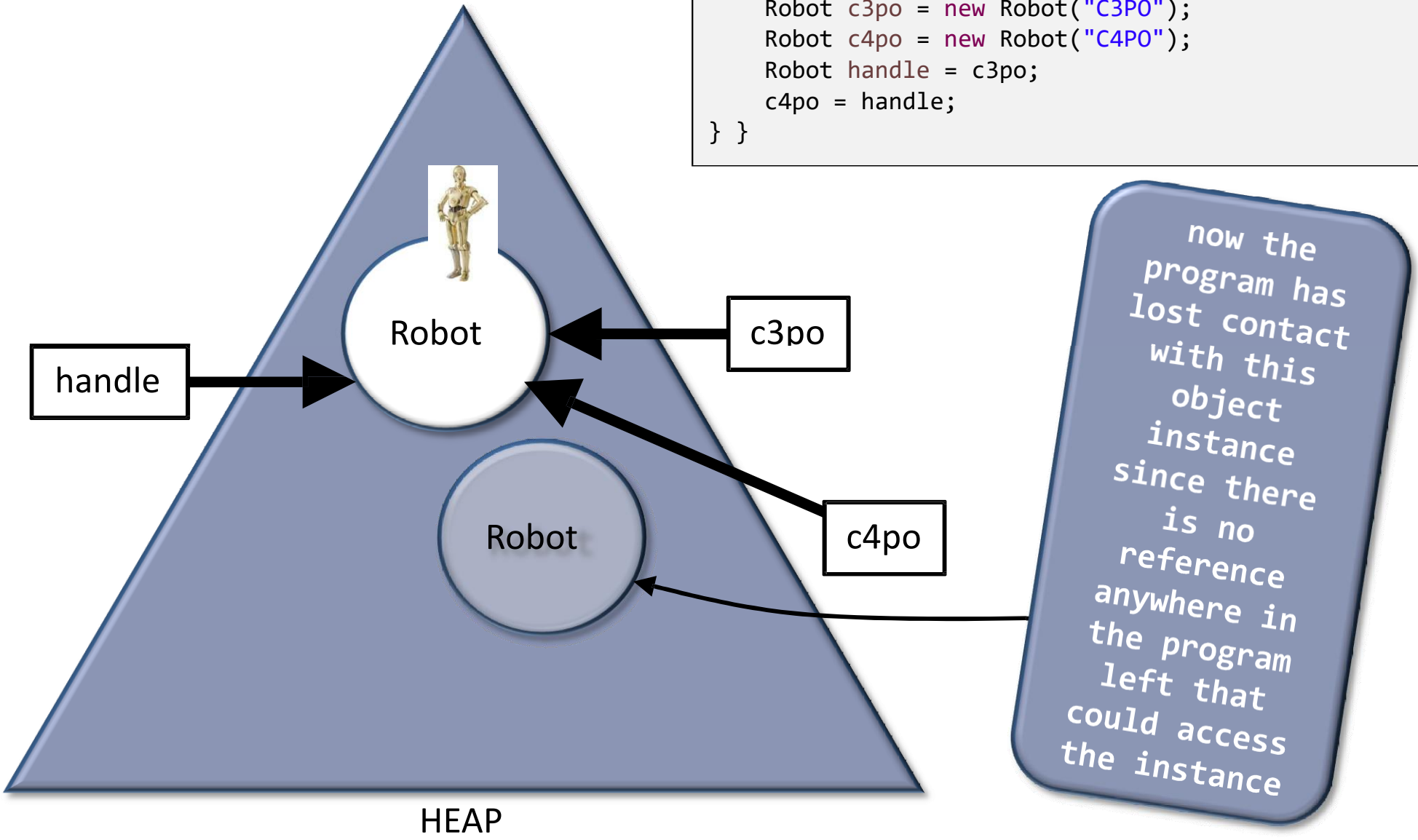
```
class RobotHeap {
  public static void main (String[] args) {
    Robot c3po = new Robot("C3PO");
    Robot c4po = new Robot("C4PO");
    Robot handle = c3po;
} }
```



Various references (c3po, handle,...) may point to the same object instance on the heap

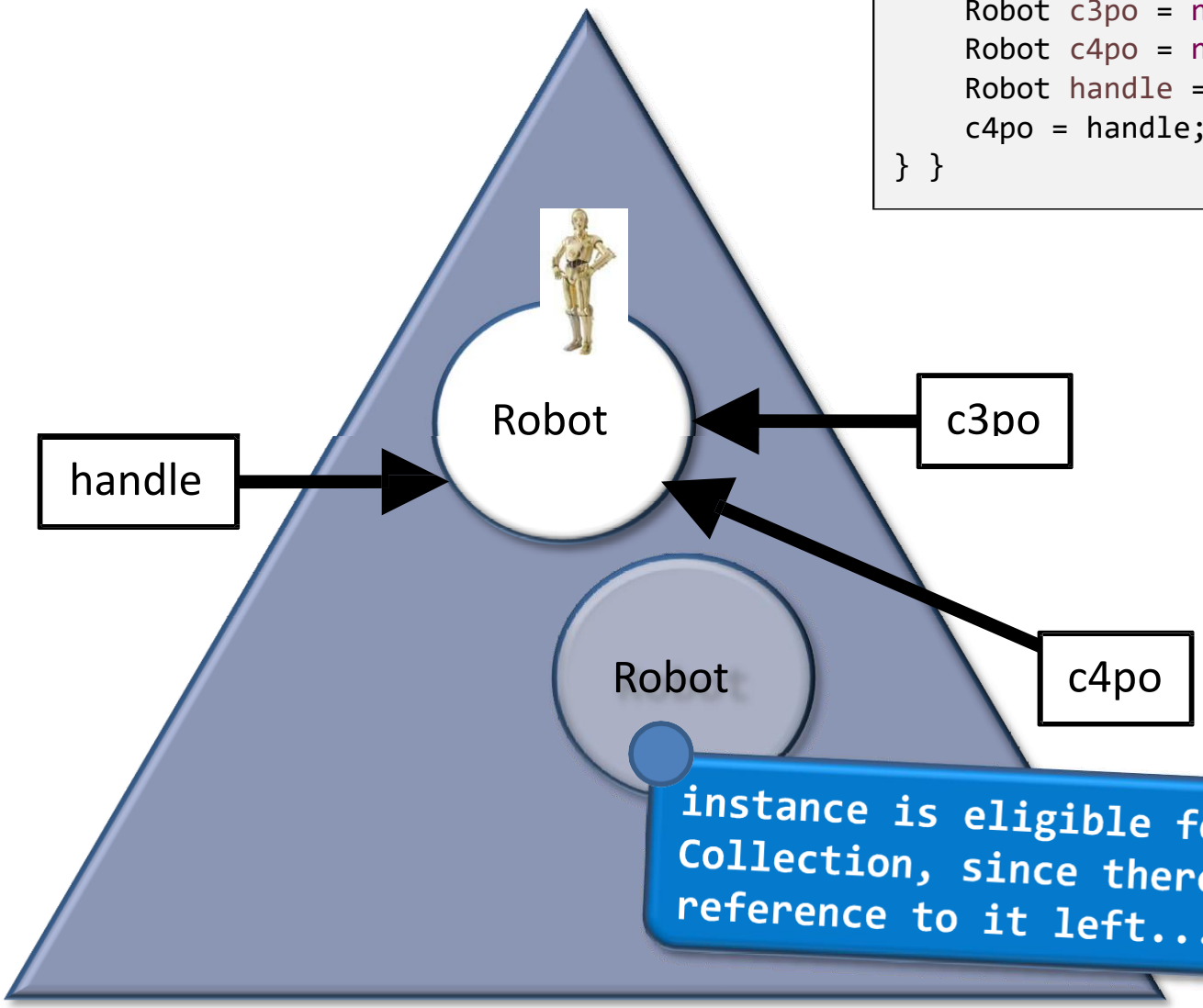
The Lifecycle of Objects

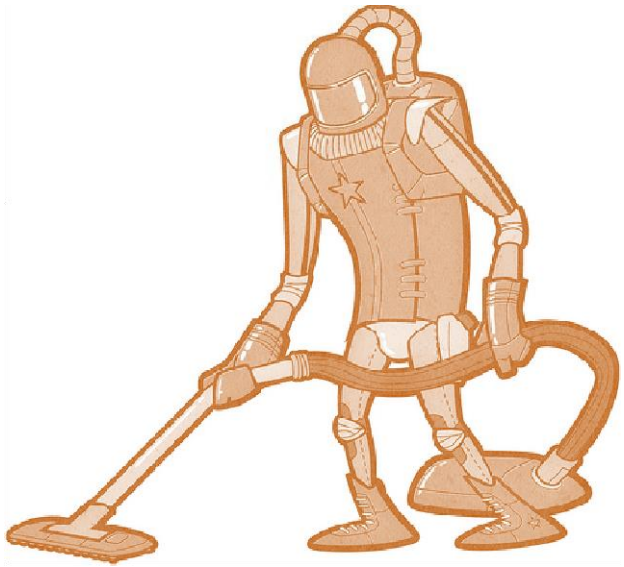
```
class RobotHeap {
  public static void main (String[] args) {
    Robot c3po = new Robot("C3PO");
    Robot c4po = new Robot("C4PO");
    Robot handle = c3po;
    c4po = handle;
} }
```



The Lifecycle of Objects

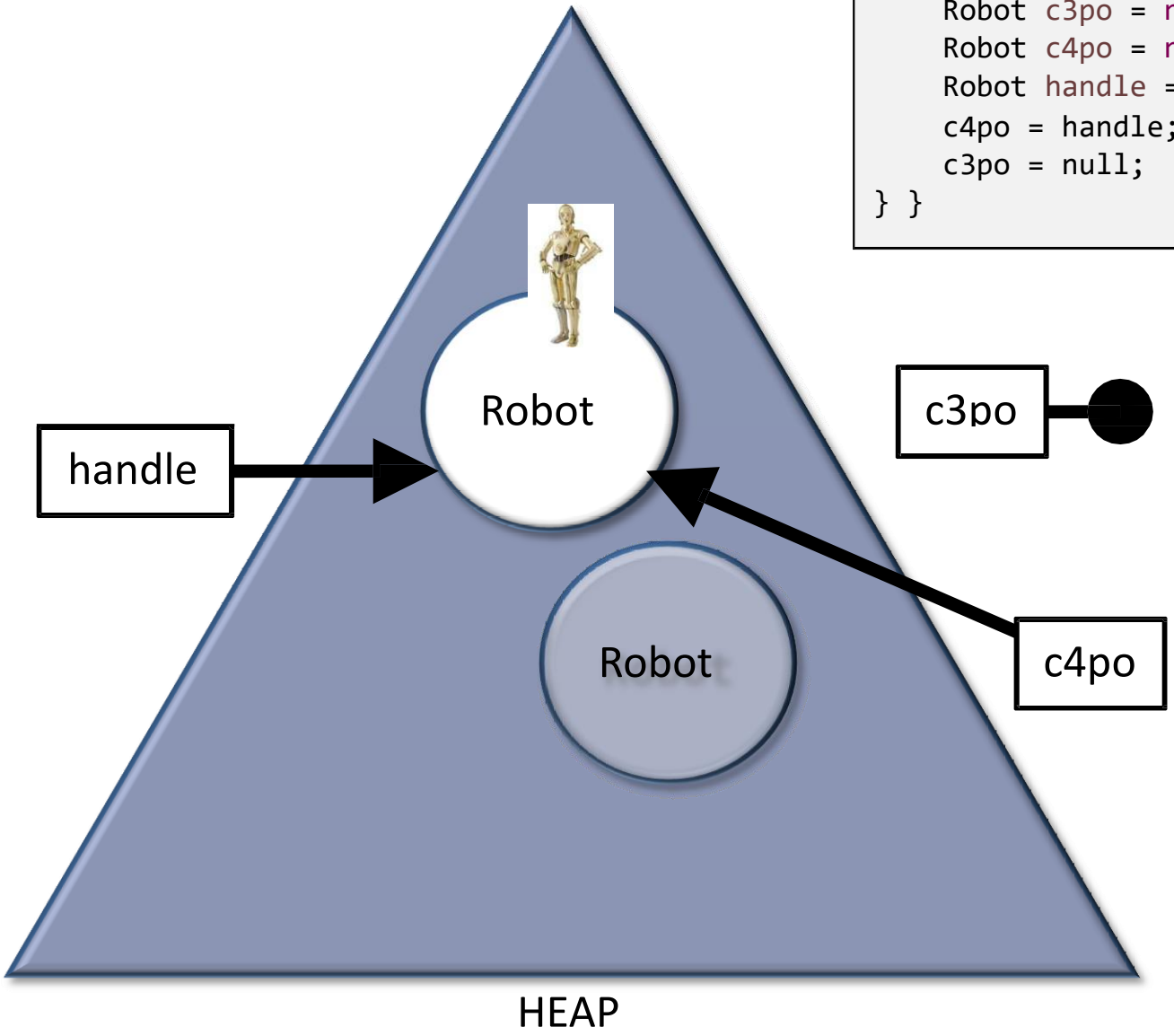
```
class RobotHeap {
  public static void main (String[] args) {
    Robot c3po = new Robot("C3PO");
    Robot c4po = new Robot("C4PO");
    Robot handle = c3po;
    c4po = handle;
} }
```





The Lifecycle of Objects

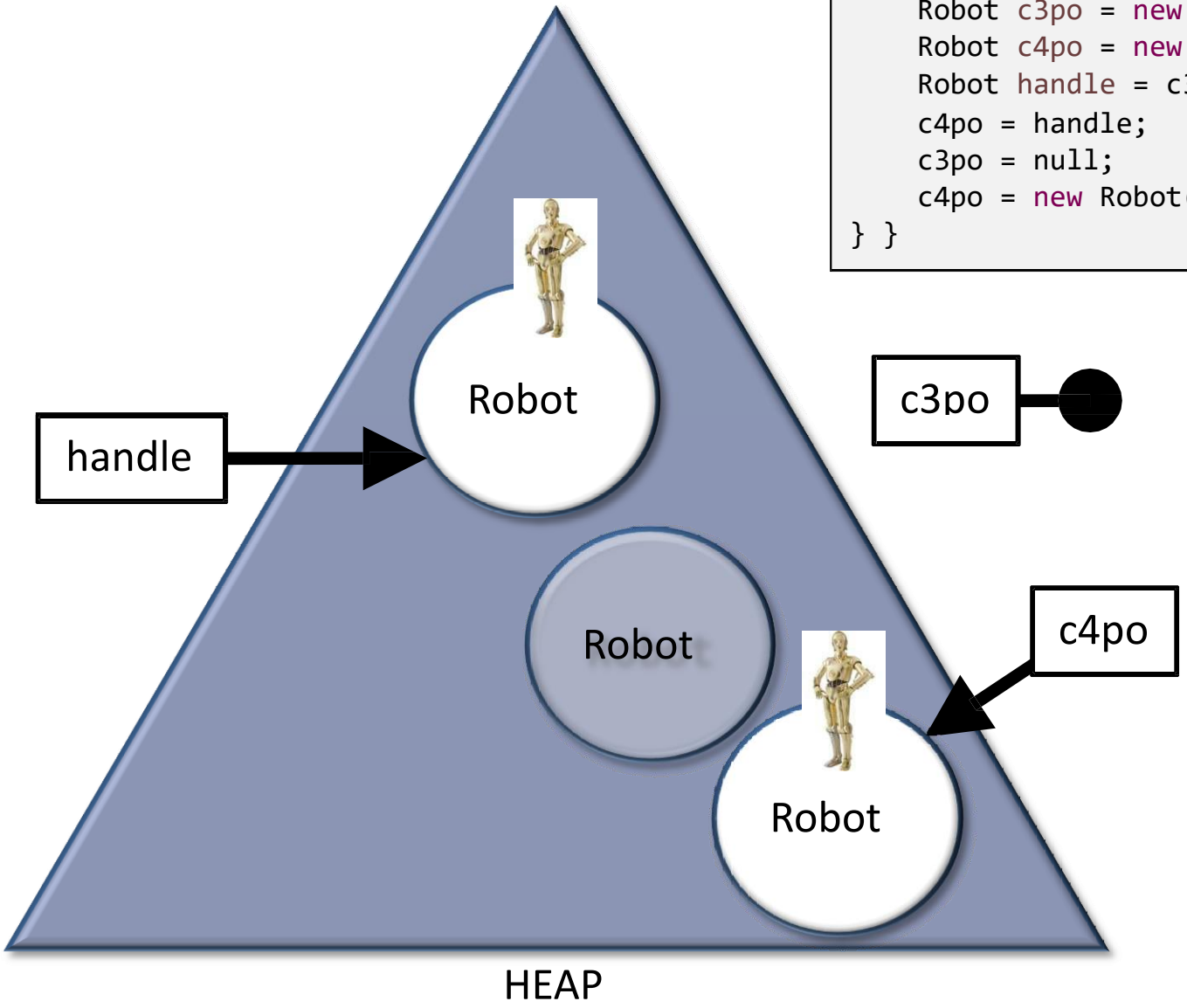
```
class RobotHeap {
  public static void main (String[] args) {
    Robot c3po = new Robot("C3PO");
    Robot c4po = new Robot("C4PO");
    Robot handle = c3po;
    c4po = handle;
    c3po = null;
} }
```



setting a reference to null disconnects the reference from the referred object; the reference now points to no object

The Lifecycle of Objects

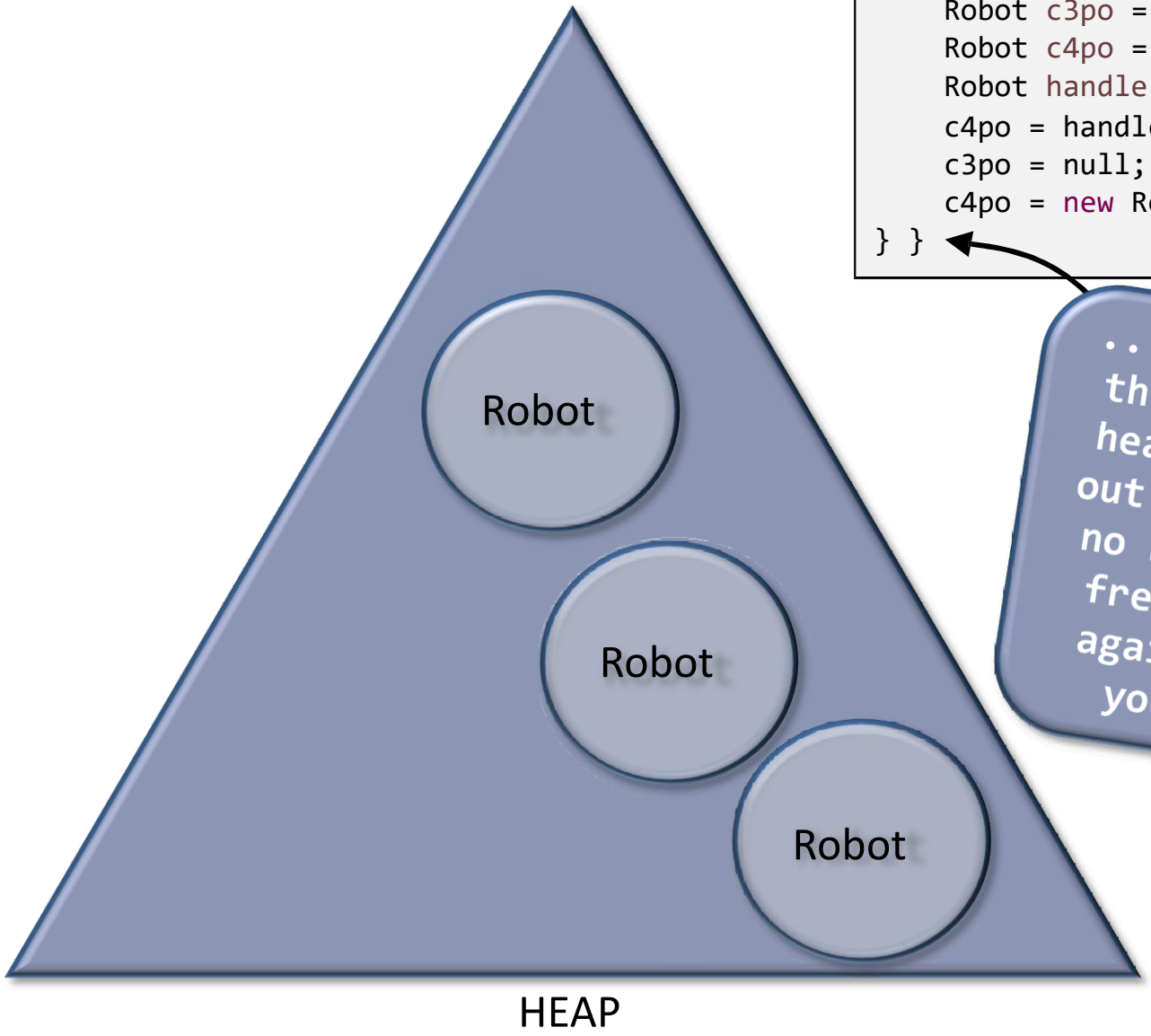
```
class RobotHeap {
  public static void main (String[] args) {
    Robot c3po = new Robot("C3PO");
    Robot c4po = new Robot("C4PO");
    Robot handle = c3po;
    c4po = handle;
    c3po = null;
    c4po = new Robot("C3PO");
} }
```



two separately accessible objects now exist on the heap, be it with identical attribute values at this point in time

The Lifecycle of Objects

```
class RobotHeap {
  public static void main (String[] args) {
    Robot c3po = new Robot("C3PO");
    Robot c4po = new Robot("C4PO");
    Robot handle = c3po;
    c4po = handle;
    c3po = null;
    c4po = new Robot("C3PO");
} }
```



...at the end of the program the heap is cleaned out in any case, no need to ever free instances again... thank you Java...

Reference Equality

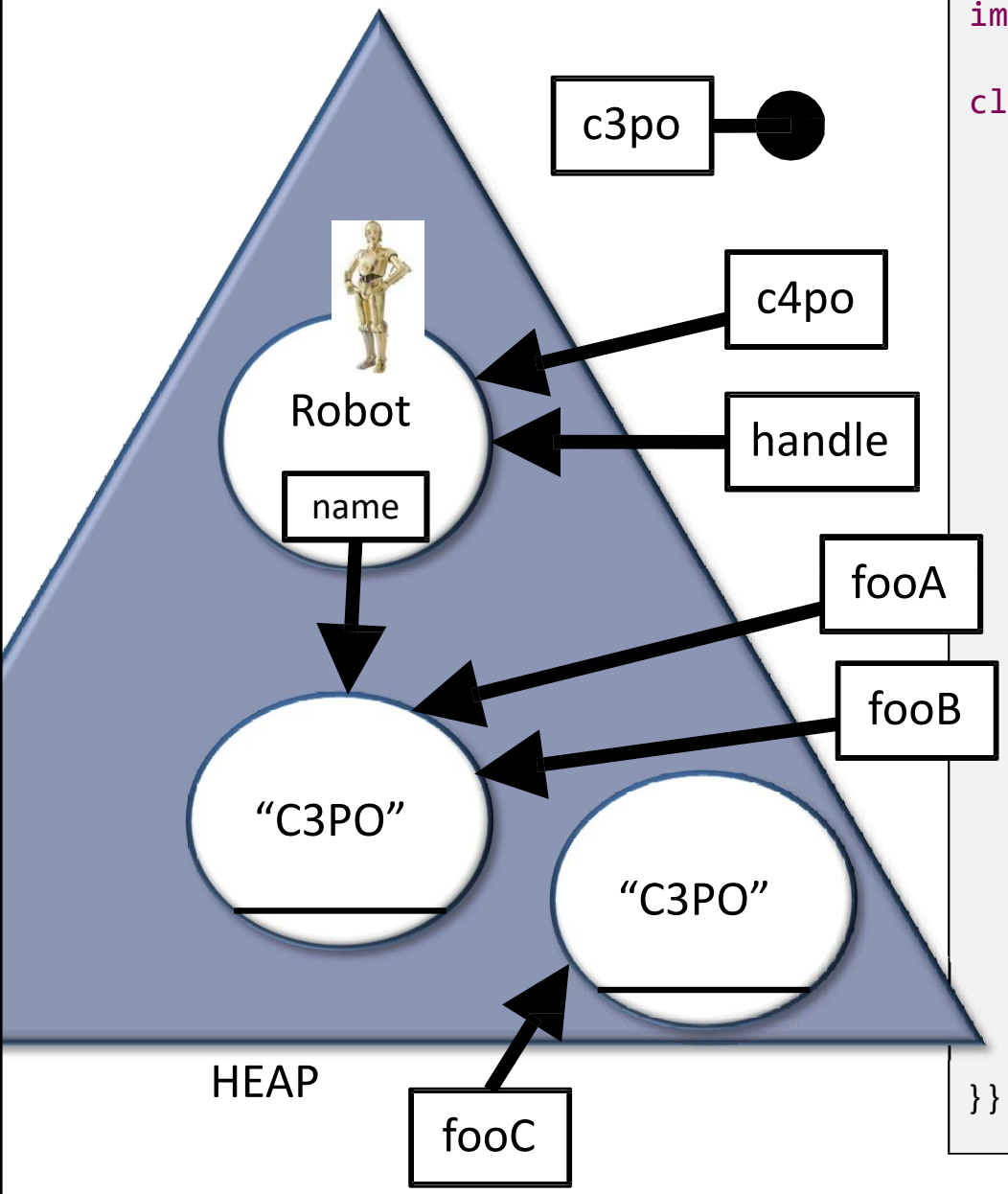
The == Operator in Java

- on plain old data types (PODs), == checks for equality just as you might expect:

```
System.out.println(7 == 5 + 2);    //true
```

- since == compares values: with references, == refers to reference equality, not the equality of object attributes
- to compare underlying attributes of objects that are referenced, use or implement the **equals()** method
- for instance, since String is a class in Java, comparing strings w.r.t. their character content must use the **equals()** method (using == you will compare whether two **String** object references are equal)

Reference Equality



RobotHeap.java

```
import java.io.*;

class RobotHeap {

  public static void main (String[] args) {
    Robot c3po = new Robot("C3PO");
    Robot c4po = new Robot("C4PO");
    Robot handle = c3po;
    c4po = handle;
    c3po = null;
    PrintStream out = System.out;
    String fooA = "C3PO";
    String fooB = "C3PO";
    String fooC = new String("C3PO");
    //visualisation snapshot  taken here
    out.println(fooA==fooB); //true (look at constants pool)
    out.println(fooA==fooC);                    //false
    out.println(fooC.equals(fooA));             //true
    out.println(c3po==handle);                  //false
    out.println(c4po==handle);                  //true
    out.println(c3po==null);                    //true
    out.println(c4po==handle);                  //true
    out.println(c4po.name==fooC);               //false
    out.println(c4po.name==fooB);               //true
    out.println(fooC.equals(c4po.name));//true
}}
```

fooA and fooB are references to a particular static String "C3PO"

equals compares attributes of the object

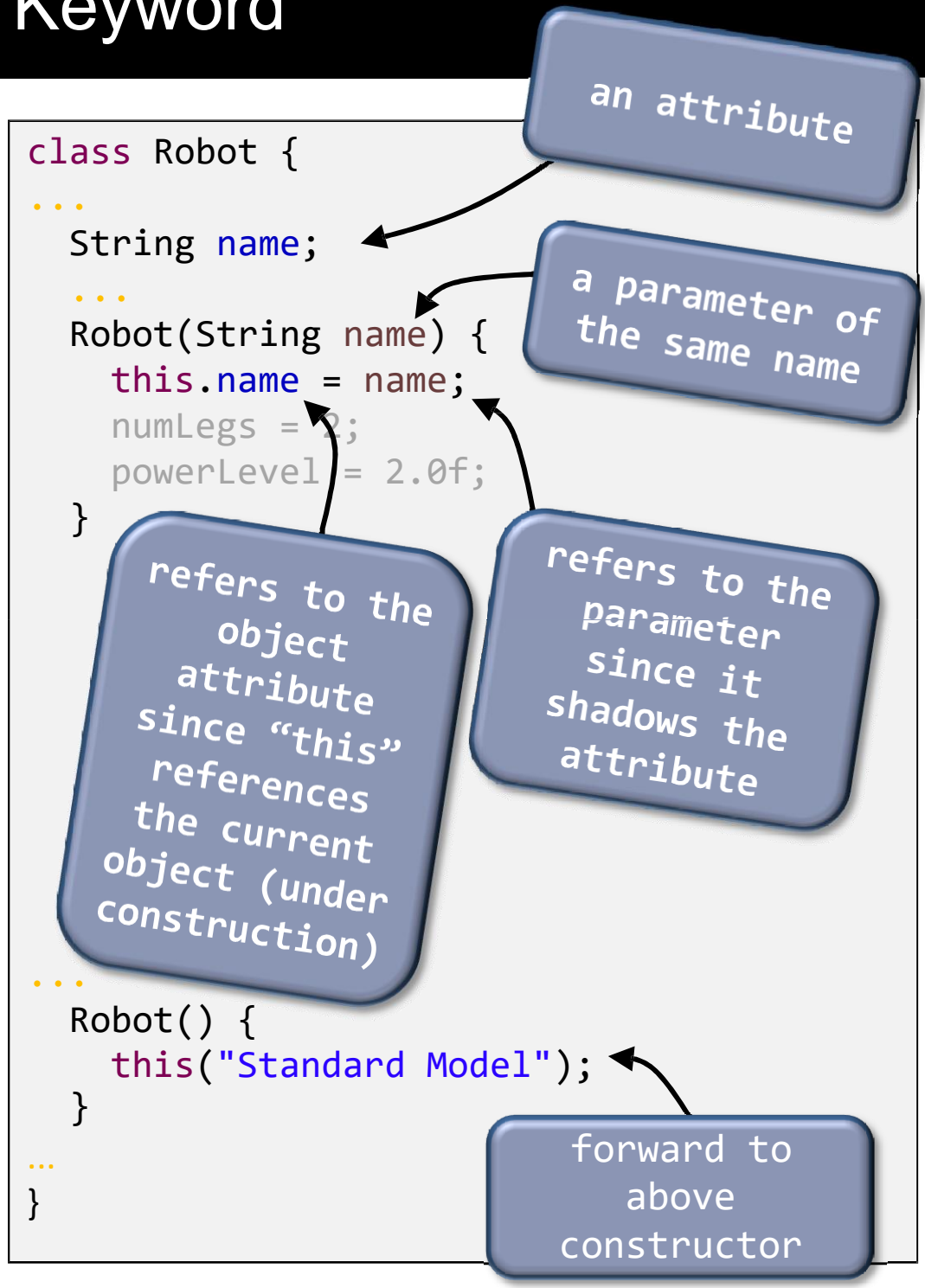
== compares values, which are references

THIS

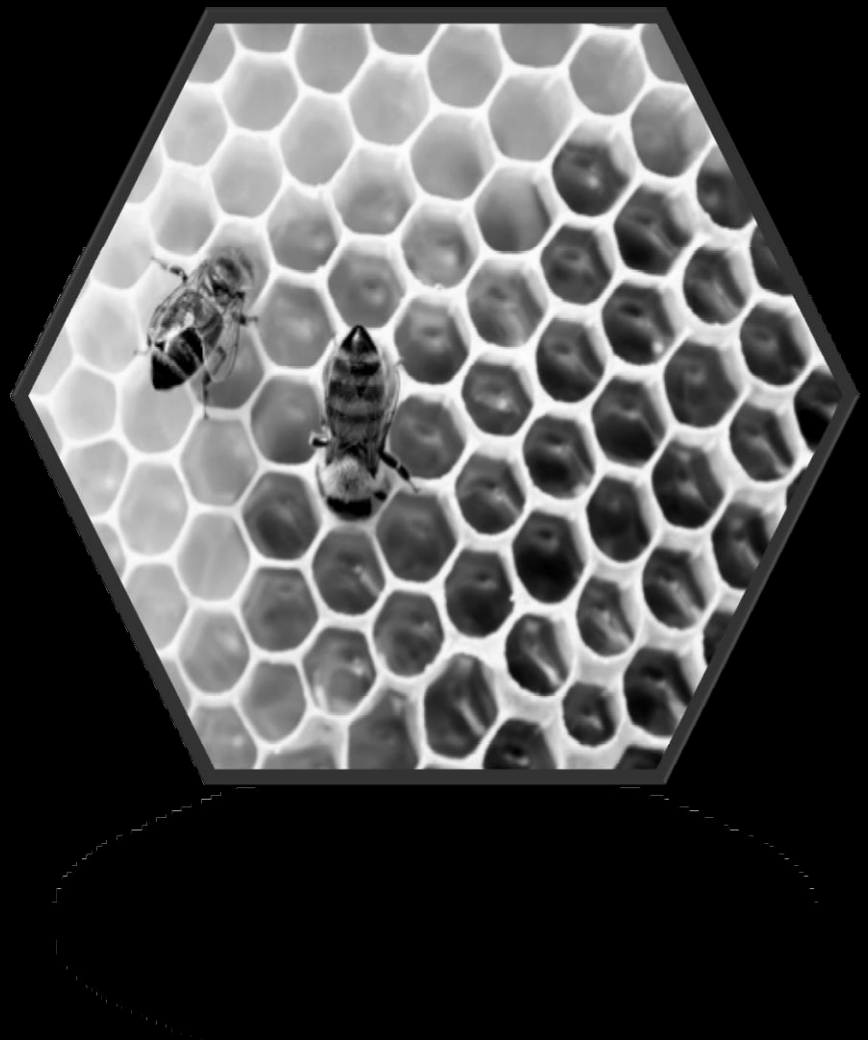
(OR THE SELF-REFERENCE)

The ”this” Keyword

- this provides a reference to the current object whose method is being executed
- it can be used inside a method or constructor, but not in a static element
- within a constructor, you can also use this in order to refer to another constructor method of the current object under construction
- particularly useful to access shadowed attributes or provide default values in constructors



Arrays



Arrays in Focus: Creating, Initialising and Iterating

- arrays (in Java) are objects too...

```
public class RobotArrays {
  public static void main (String[] args) {
    Robot[] robotsA = new Robot[3]; //instantiate array of references to 3 Robot objects
    System.out.println(robotsA[0]); //at start, array locations carry null
    robotsA[0] = new Robot("C3PO"); //initialise entry at index 0
    robotsA[1] = new Robot("C4PO"); //initialise entry at index 1
    robotsA[2] = robotsA[0];         //initialise with same reference as index 0
    Robot[] robotsB = {              //neat initialisation syntax using {...}
              new Robot("C5PO"),
              robotsA[0],
              robotsA[1]
            };
    System.out.println(robotsB.length); //print size of array robotsB
    for (Robot robot : robotsB)          //loop through entries, assign current to robot
      System.out.println(robot.name);    //print name of current element
  }
```

REMEMBER: This is an Iterator using the ":" notation, which provides a reference "robot" to each object held in the array turn-by-turn (The reference is not a one to the array location itself, which is not an object.)

...beware, it is the wrath of the null that you need to defend against in your programming...

Java Arrays as Arguments and Return Values

- since arrays are objects, you can also pass them to methods or return them

---

- note that the **java.util.Arrays** class provides static helper functions particularly useful for sorting, comparing and finding things in arrays of PODs
- however, we will discuss this soon and also learn about **Collections**, which provide extended functionality beyond arrays in a true Java fashion

RobotArrays.java

```
public class RobotArrays {

  static Robot[] makeRobotArray() {
    Robot[] robots = {
                new   Robot("C3PO"),
                new   Robot("C4PO"),
                new Robot("C5PO")
              };
    return robots;
  }

  static void printArray(Robot[] robots) {
    for (Robot robot : robots)
      System.out.println(robot.name);
  }

  public static void main (String[] args) {
    Robot[] robots = makeRobotArray();
    printArray(robots);
  }
}
```

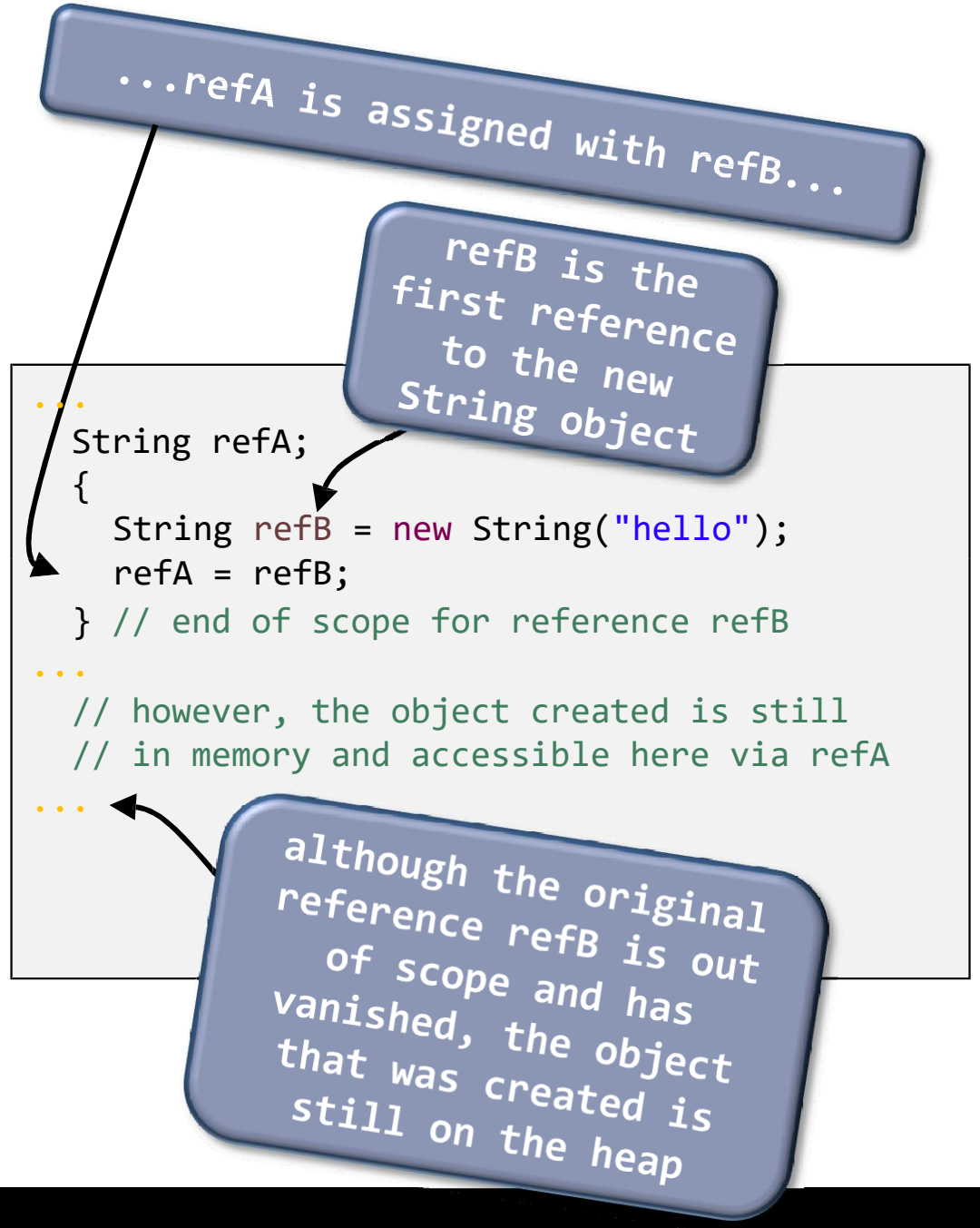
return of an array

an array parameter

Object Scope

Scope and Object Persistence

- the scope of argument variables and local variables is the same as in C
- objects do not vanish when the creation reference goes out of scope
- as long as there is a single reference somewhere in the program to the object it is kept on the heap
- Garbage Collection may only be triggered for `dead' objects not referenced anywhere anymore



Objects are Entities with their Independent State

- objects share nothing implicitly, other than their class and static features
  - objects store their own attributes
  - but do objects execute their own copies of methods?
- objects are therefore stateful entities and in order to exist they require that their state is supported by underlying heap space

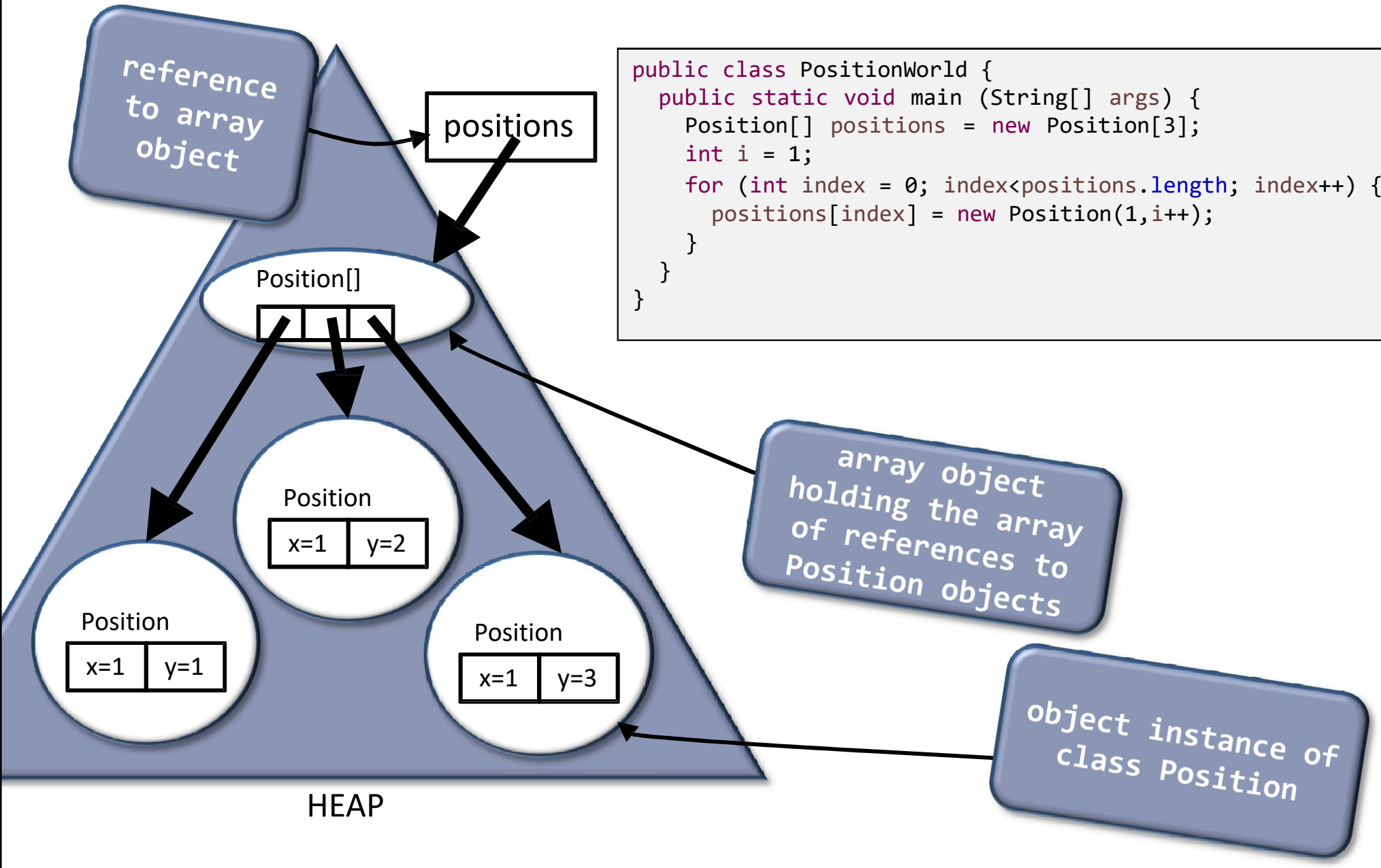
Position.java

```
class Position {
  int x;
  int y;

  Position(int x, int y) {
    this.x =  x;
    this.y =  y;
  }

  void move(int dx, int dy) {
    x += dx:
    y += dy;
  }
}
```

```
public class PositionWorld {
  public static void main (String[] args) {
    Position[] positions = new Position[3];
    int i = 1;
    for (Position p : positions)
      p = new Position(1,i++);
    ... //oh  no  -  nulls again
} }
```

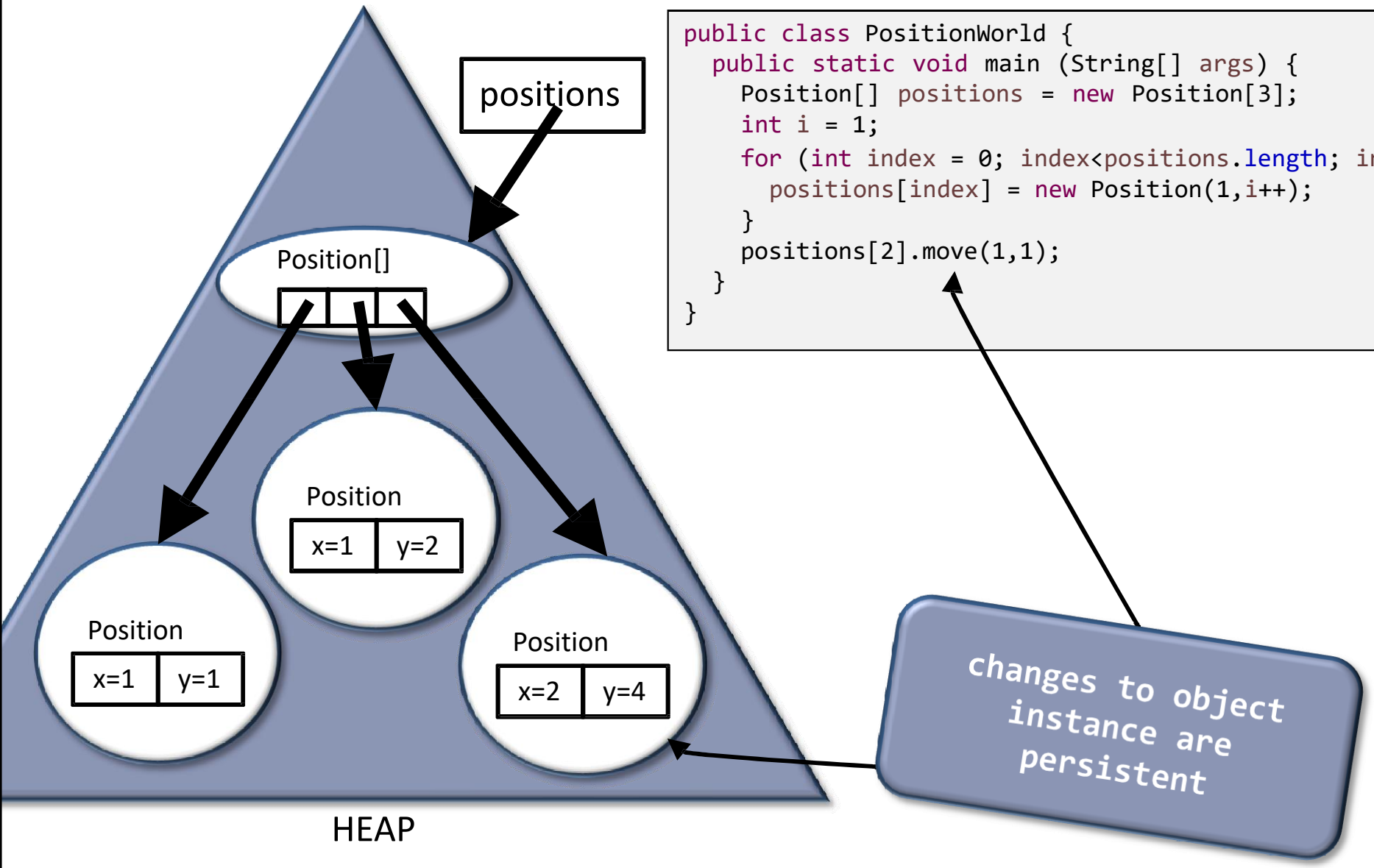
Why does this code NOT work for initialisation?

References and Object Scope




```
public class PositionWorld {
  public static void main (String[] args) {
    Position[] positions = new Position[3];
    int i = 1;
    for (int index = 0; index<positions.length; index++) {
      positions[index] = new Position(1,i++);
    }
  }
}
```

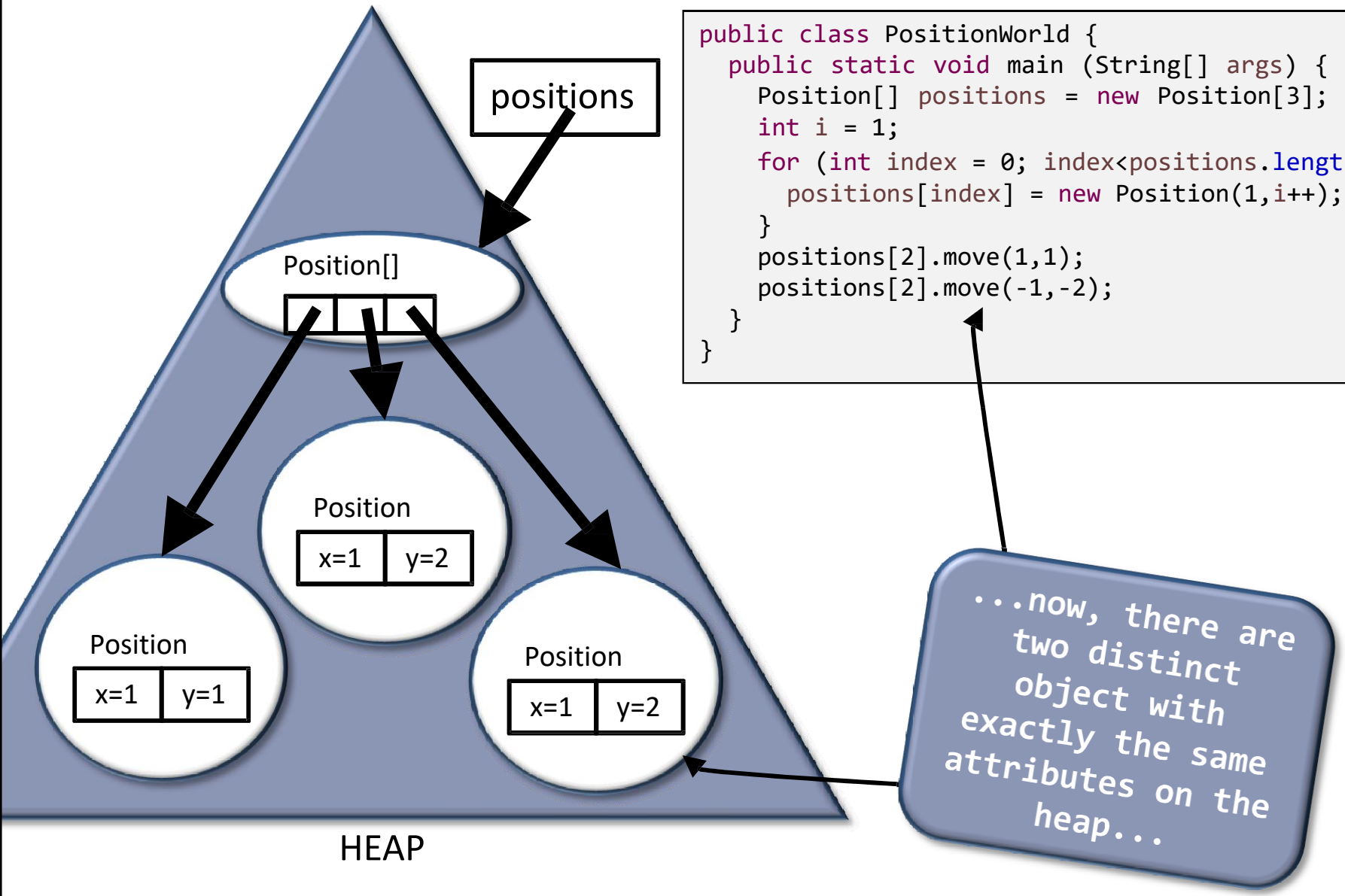
References and Object Scope




```
public class PositionWorld {
  public static void main (String[] args) {
    Position[] positions = new Position[3];
    int i = 1;
    for (int index = 0; index<positions.length; index++) {
      positions[index] = new Position(1,i++);
    }
    positions[2].move(1,1);
  }
}
```

changes to object instance are persistent

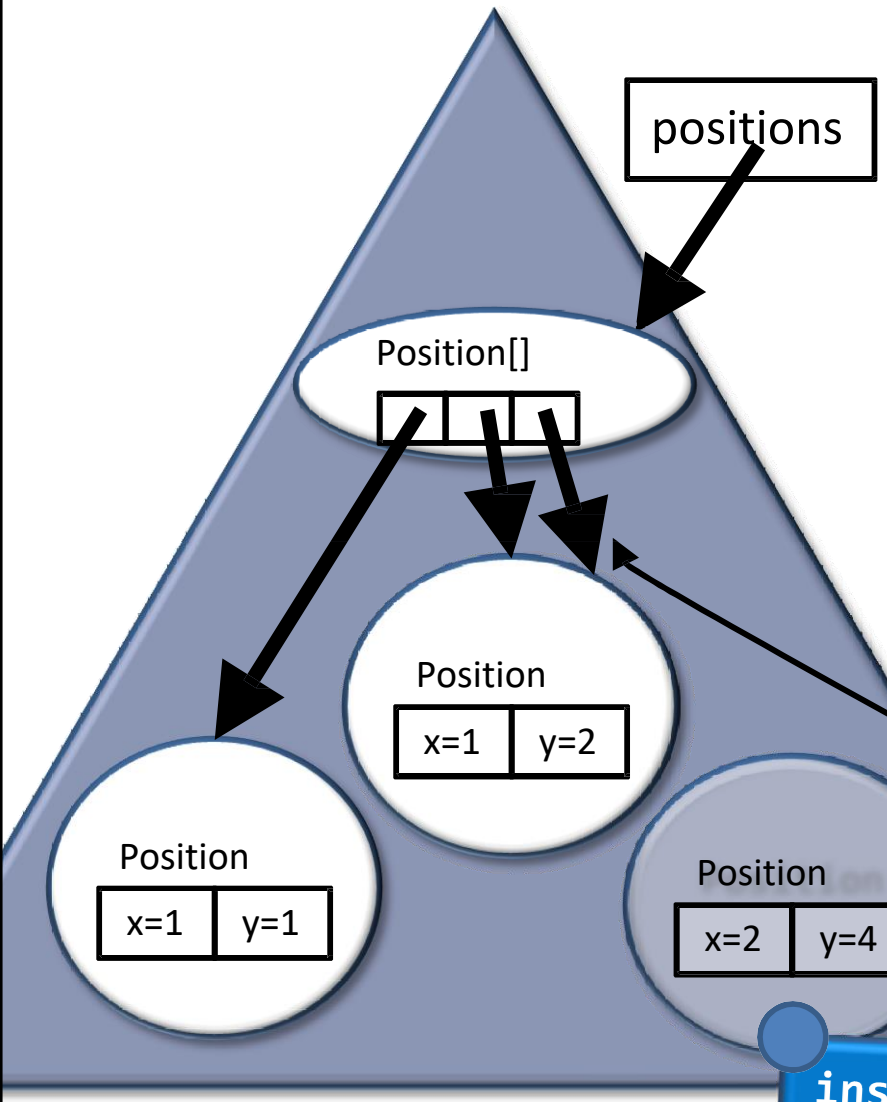
References and Object Scope




```
public class PositionWorld {
  public static void main (String[] args) {
    Position[] positions = new Position[3];
    int i = 1;
    for (int index = 0; index<positions.length; index++) {
      positions[index] = new Position(1,i++);
    }
    positions[2].move(1,1);
    positions[2].move(-1,-2);
  }
}
```

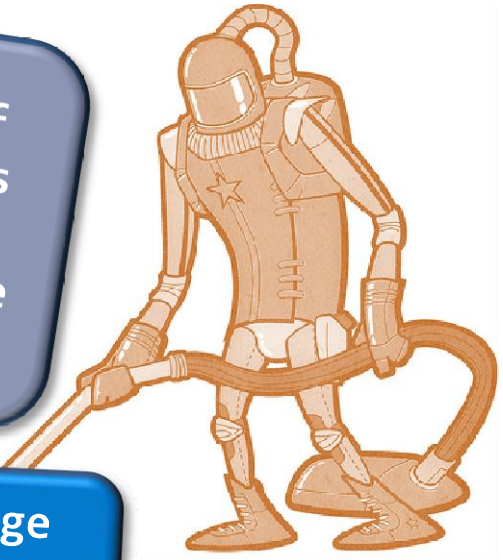
...now, there are two distinct object with exactly the same attributes on the heap...

References and Object Scope



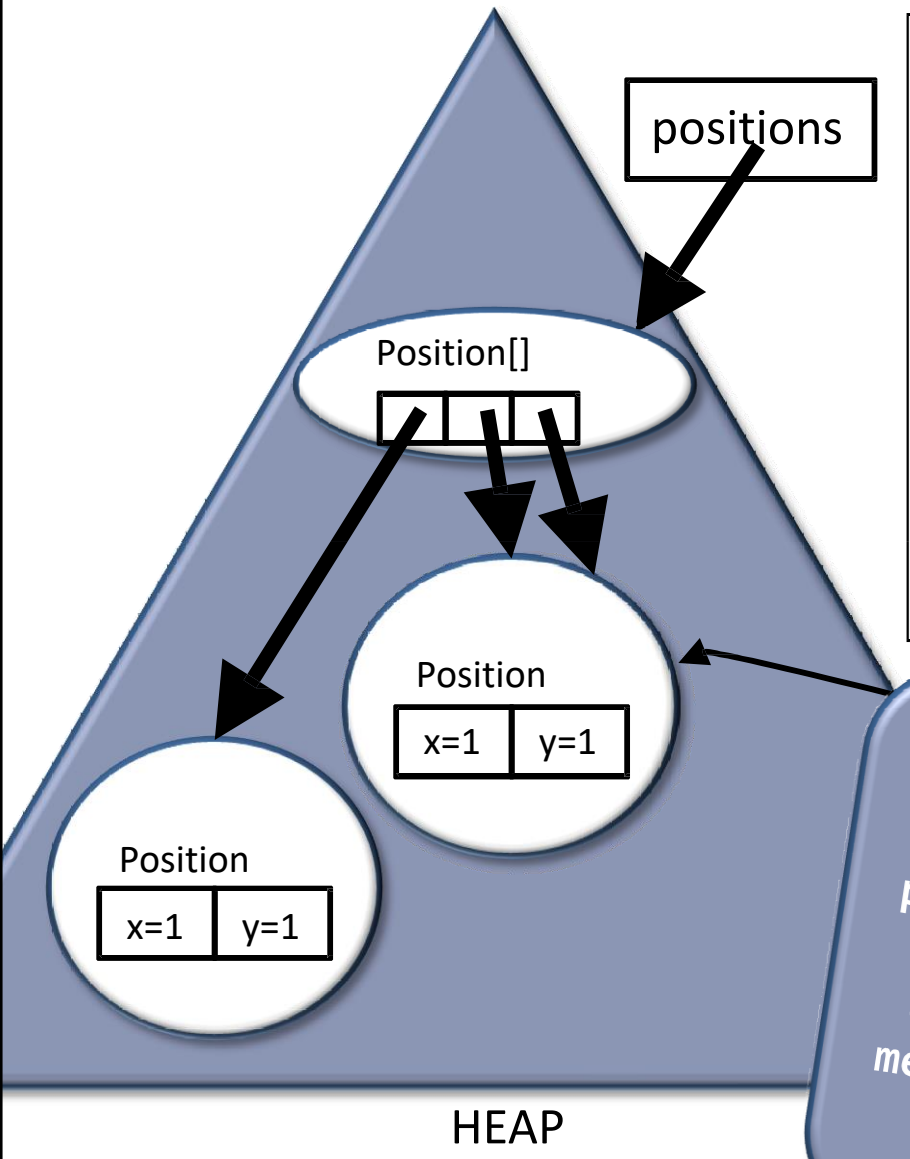

```
public class PositionWorld {
  public static void main (String[] args) {
    Position[] positions = new Position[3];
    int i = 1;
    for (int index = 0; index<positions.length; index++) {
      positions[index] = new Position(1,i++);
    }
    positions[2].move(1,1);
    positions[2].move(-1,-2);
    positions[2] = positions[1];
  }
}
```

reassignment of references leads to two identical references in the positions array



instance is eligible for Garbage Collection, since there is no reference to it left...

References and Object Scope



PositionWorld.java

```
public class PositionWorld {

  public static void main (String[] args) {
    Position[] positions = new Position[3];
    int i = 1;
    for (int index = 0; index<positions.length; index++) {
      positions[index] = new Position(1,i++);
    }
    positions[2].move(1,1);
    positions[2].move(-1,-2);
    positions[2] = positions[1];
    positions[1].move(0,-1);
  }
}
```

changes to objects may happen through various routes: positions[2].y has now changed although through means of using the reference positions[1]

again, there are now two distinct objects on the heap that have identical attributes

Calling

More Implications: Call-by-Value vs. Call-by-Reference

- since objects are accessed via references you can only use an object as a parameter in the **call-by-reference** mechanism, not like primitives that support **call-by-value** (as shown on the right)

- in case you would like to simulate a **call-by-value** mechanism for objects you'd need to create a (deep) clone of your object (a copy of the object with all referenced objects also cloned) and supply a reference to this new instance as parameter in the method call – you would not use this clone again after the call

CallWorld.java

```
public class CallWorld {

  static int square(Number n) {
    n.x  *= n.x;
    return n.x;
  }

  static int square(int x) {
    x *= x;
    return x;
  }

  public static void main (String[] args) {
    int x = 10;
    System.out.println(square(x));
    System.out.println(x);
    Number n = new Number(10);
    System.out.println(square(n));
    System.out.println(n.x);
  }
}
```

call-by-ref: altering attributes of n is persistent

call-by-value: altering the parameter x does not alter the int x at the caller end

Number.java

```
public class Number {
  int x;

  Number(int x) {
    this.x = x;
  }
}
```

this class wraps around an int, instances will be accessed by references as usual

To Do

- recap content and check out the unit website
- write, compile, run and understand all the tiny programs from the lectures so far **(use a text editor and terminal for these)**

- ➢ note that we expect you to manage your own device and may not be able to help with issues on your own devices