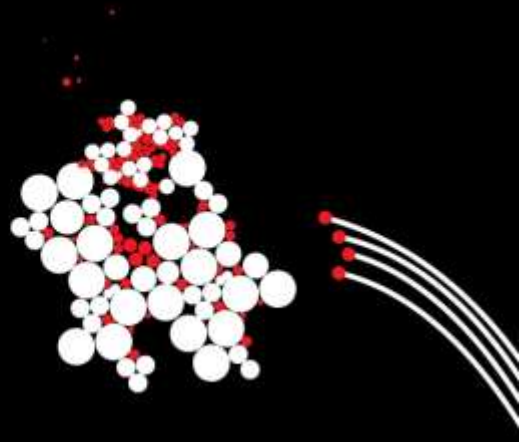
UNIVERSITY OF TWENTE.

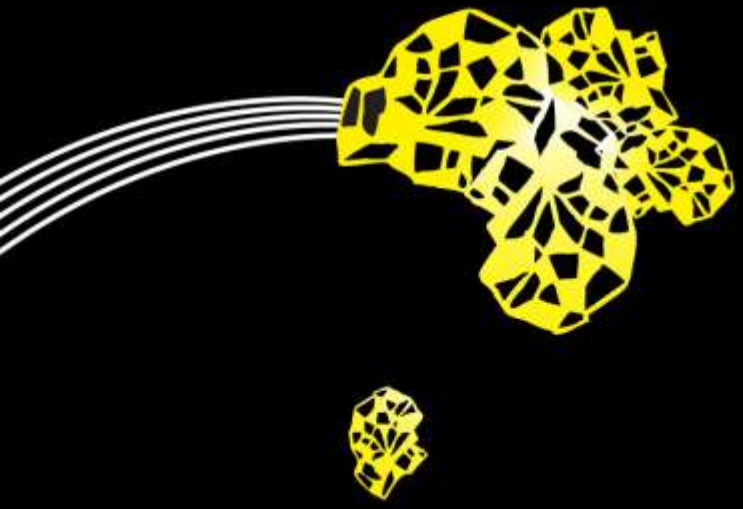


# COMPILER CONSTRUCTION:

# CC 6-2: MEMORY USAGE (2)

29 MAY 2019

MODULE 8: PROGRAMMING PARADIGMS



# PASSING PARAMETERS

- Parameters are part of the activation record
- Call-by-value (Java)
  - The argument is put in the activation record
  - For the callee, this can be treated as a local variable
  - Changes are *not* copied back to the caller
- Call-by-reference
  - An address of the location holding the argument is put in the activation record
  - The callee has to dereference (look up the value in memory)
  - Any change made by callee is observable by caller
- Call-by-name (out of fashion)
  - The argument expression is not evaluated but copied to the callee
    - Modulo appropriate renaming of variables
  - Every reference by callee evaluates the argument
  - Evaluation may depend on variables that also change

# PASSING PARAMETERS: EXAMPLE

```
1   int a[];
2   int i;
3   …
4   void m(int b, int j) {
5      b = 2;
6      j = 0;
7      a[i] = 10;
8      b = 4;
9   }
10  …
11  a = { 1, 3, 5 };
12  i = 1;
13  m(a[i+1], i);
14  // value of a and i?
```

ref to a[2]

ref to i

Value of `a` and `i` after executing main code?

- If `a[i+1]` and `i` are passed *by value*
  - `a` = {1, 10, 5}
  - `i` = 1
- If `a[i+1]` and `i` are passed *by reference*
  - `a` = {10, 3, 4}
  - `i` = 0
- If `a[i+1]` and `i` are passed *by name*
  - `a` = {10, 4, 2}
  - `i` = 0

# PASSING PARAMETERS: EXAMPLE

```
1   int a[];
2   int i;
3   …
4   void m(int b, int j) {
5      b = 2;
6      j = 0;
7      a[i] = 10;
8      b = 4;
9   }
10  …
11  a = { 1, 3, 5 };
12  i = 1;
13  m(a[i+1], i);
14  // value of a and i?
```


```
// what kind of parameter
// could i+1 be?
m(a[i+1], i+1);
```

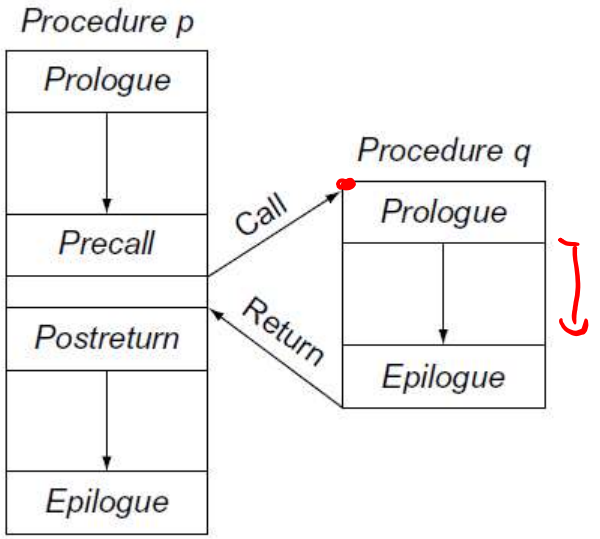
Value of `a` and `i` after executing main code?

- If `a[i+1]` and `i` are passed *by value*
  - `a = {1, 10, 5}` and `i = 1`
- If `a[i+1]` and `i` are passed *by reference*
  - Assignment to `b` (lines 5, 8) changes `a[2]`
  - Assignment to `j` (line 6) changes `i`
  - `a = {10, 3, 4}` and `i = 0`
- If `a[i+1]` and `i` are passed *by name*
  - Assignment to `b` (lines 5, 8) changes `a[i+1]`
  - Assignment to `j` (line 6) changes `i`
  - `a = {10, 4, 2}` and `i = 0`

- By value? OK
- By reference? no, `i+1` does not have reference
- By name? only if 2nd parameter not used as assignment target

# ROLES IN PROCEDURE LINKAGE

- **Precall**:
  What the caller does to prepare the call
  - Callee's environment: arguments, return address, return value
  - Stores addressability (incl. global display)
  - Saves caller-saves registers
- **Prologue**:
  What the callee does to set up the local data
  - Reserves space for local data
  - Saves callee-saves registers
- **Epilogue**:
  What the callee does to prepare for return
  - Stores return value
  - Restores callee-saves registers
- **Postcall**:
  What the caller does to conclude the return
  - Restores addressability link
  - Restores caller-saves registers



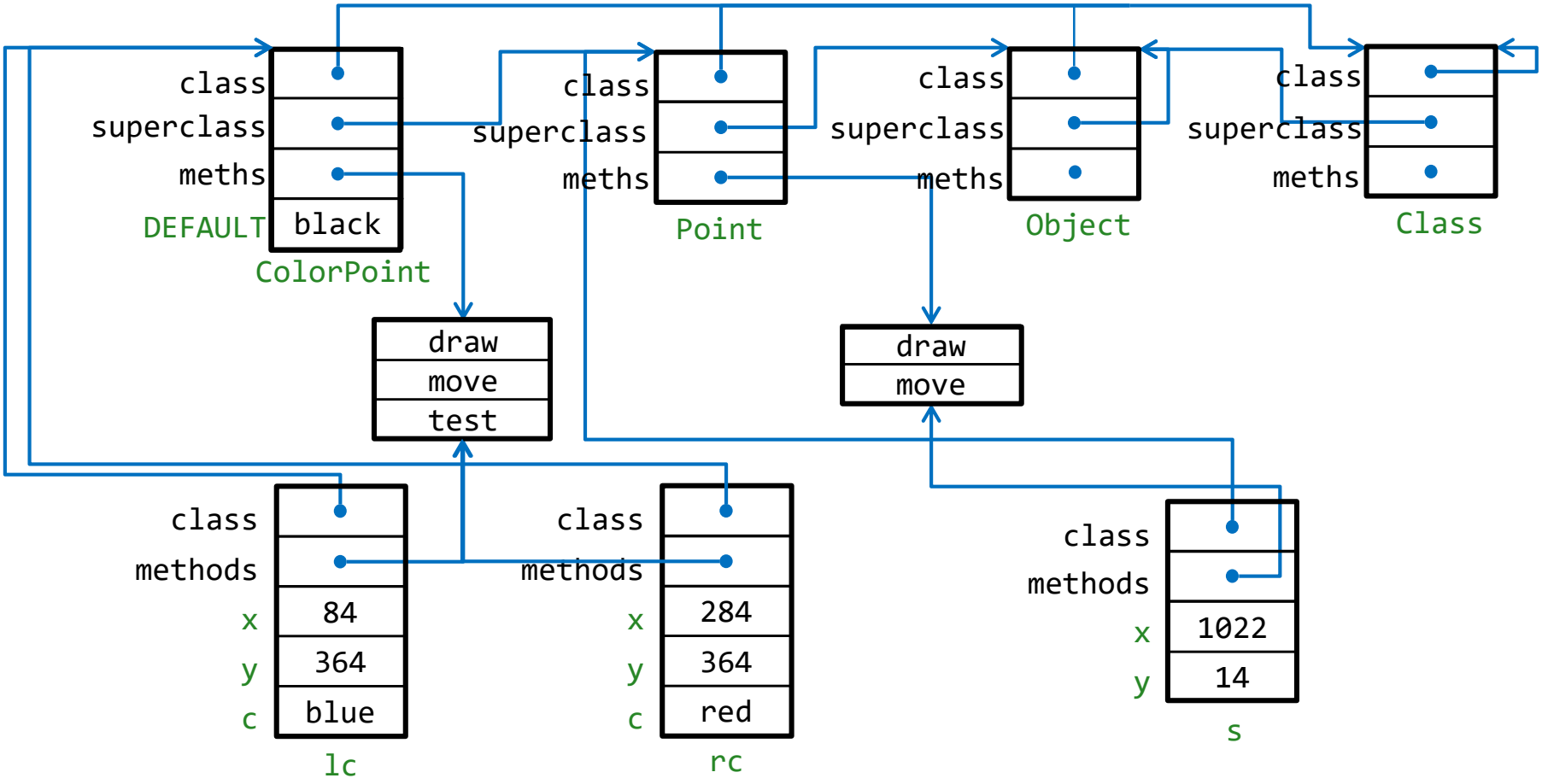
# MEMORY LAYOUT FOR OOLS

```
class Point {
   public int x, y;
   public void draw() { … }
   public void move() { … }
}
class ColorPoint extends Point {
   static final Color DEFAULT = …;
   Color c;
   public void draw() { … }
   public void test() { draw(); }
}
class C {
   int x, y;
   public void m() {
      Point s = new Point();
      Point lc = new ColorPoint();
      Point rc = new ColorPoint();
   }
}
```

What do we need to store?

- Class objects: in static block
  - One for each class (Point, …)
  - Pointer to (class object of) superclass
  - Table of non-static method addresses
  - Table of static method addresses
  - Static variables
- "Proper" objects: on heap
  - One for each instantiated object
  - Pointer to class (object)
  - Table of non-static method addresses
  - Instance variables
- Values of class types
  - Always stored as pointer to heap

# OOL MEMORY LAYOUT: EXAMPLE




Note: this example does not show static methods