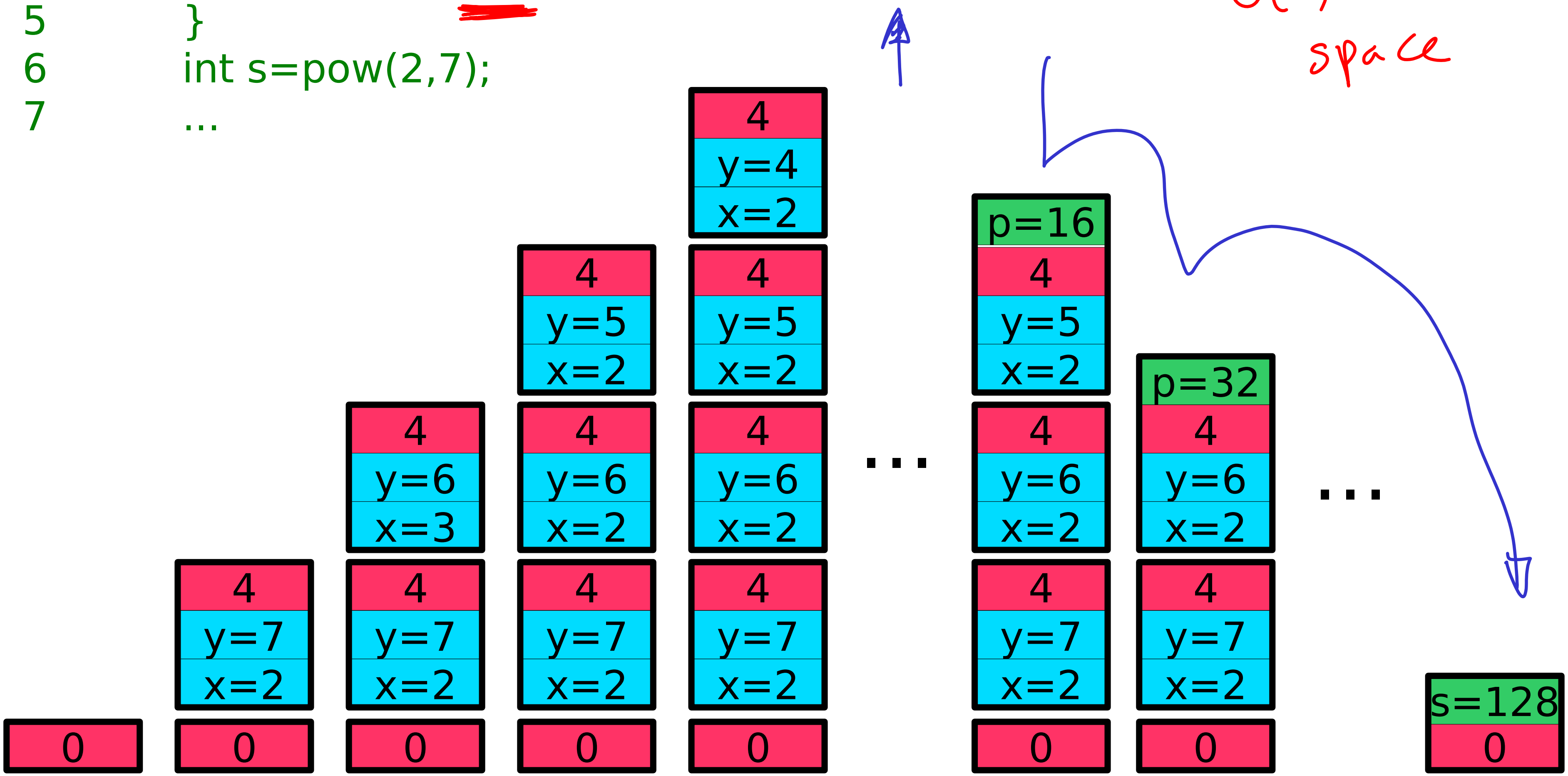
Recursive Functions

ML notes

```
1        int pow (int x, int y) {
2                if (y==0) return 1;
3                int p = pow(x,y-1);
4                return x*p;
5        }
6        int s=pow(2,7);
7        ...
```

O(n) space



0

4
y=7
x=2
0

4
y=6
x=3
4
y=7
x=2
0

4
y=5
x=2
4
y=6
x=2
4
y=7
x=2
0

4
y=4
x=2
4
y=5
x=2
4
y=6
x=2
4
y=7
x=2
0

...

p=16
4
y=5
x=2
4
y=6
x=2
4
y=7
x=2
0

p=32
4
y=6
x=2
4
y=7
x=2
0

...

s=128
0

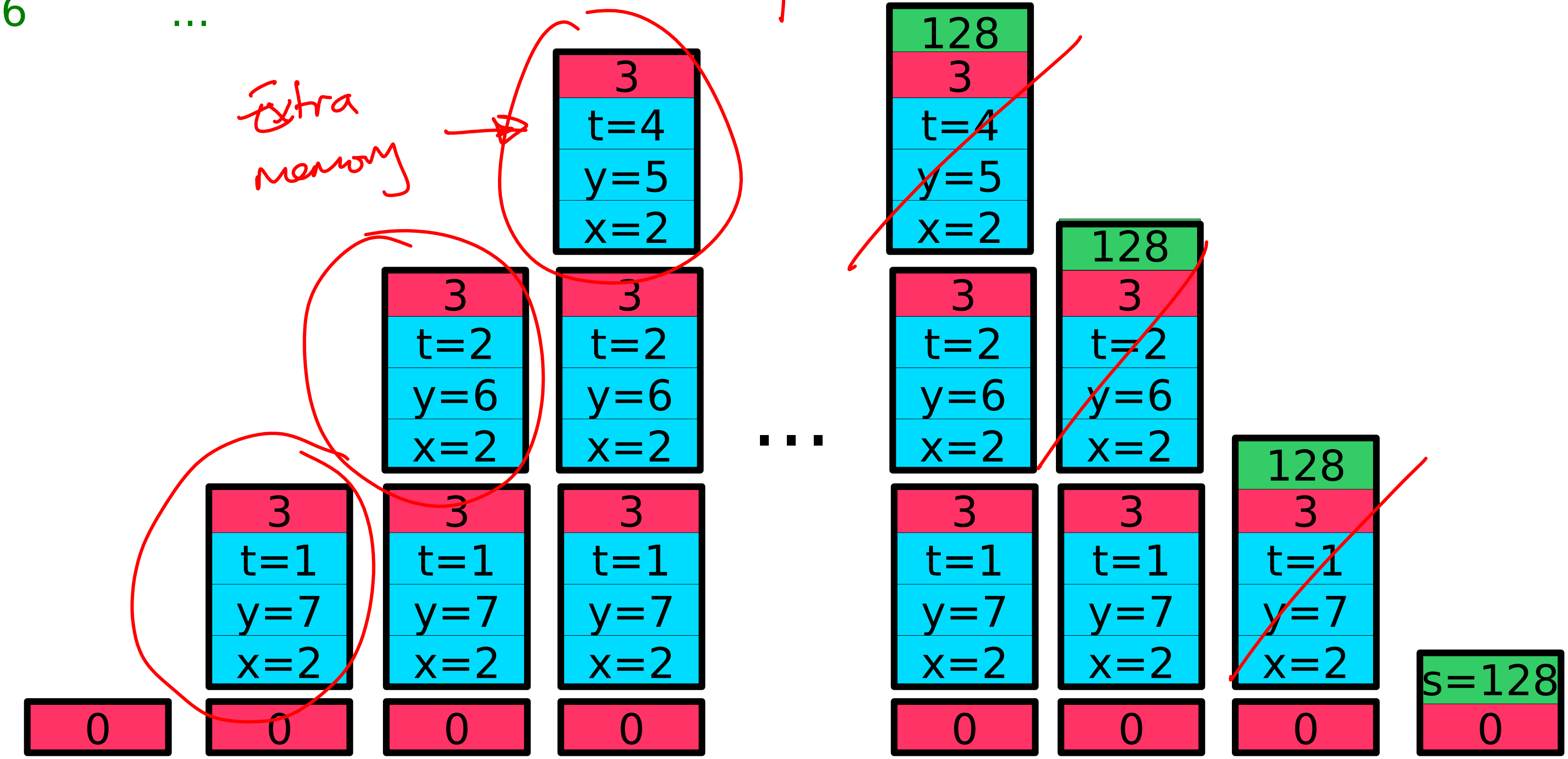
Tail-Recursive Functions I

Accumulator

```
int pow (int x, int y, int t) {
        if (y==0) return t;
        return pow(x,y-1, t*x);
}
int s = pow(2,7,1);
...
```

t=128 at top

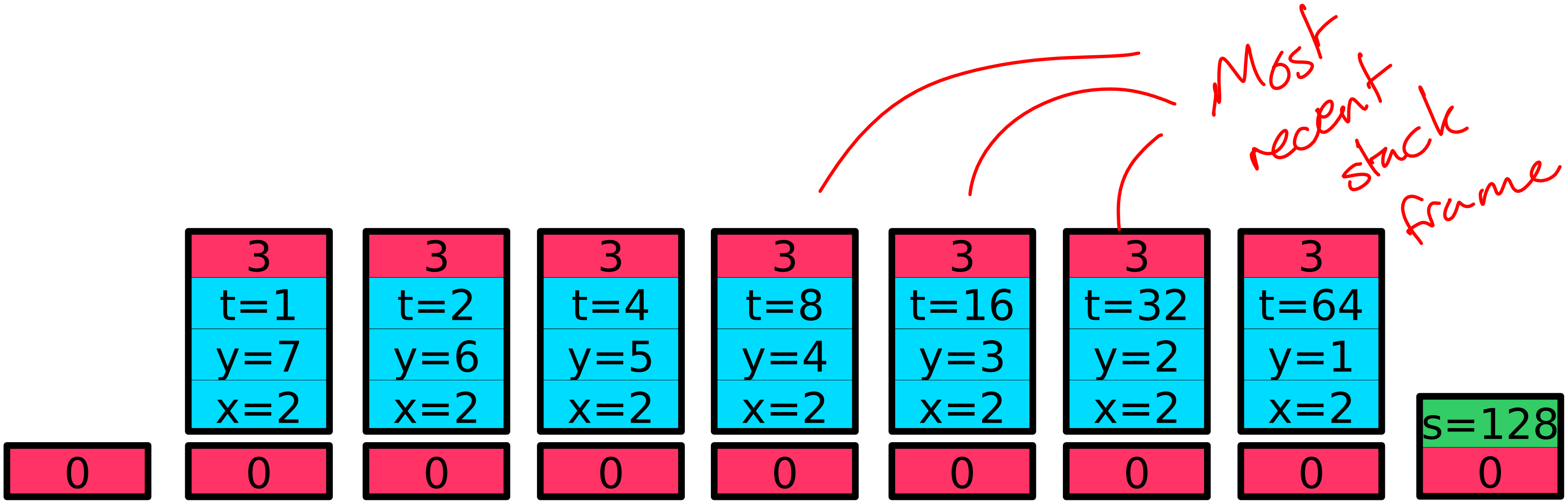
Extra memory



Tail-Recursive Functions II

```
int pow (int x, int y, int t) {
        if (y==0) return t;
        return pow(x,y-1, t*x);
}
int s = pow(2,7,1);
...
```

O(1)
space



Most recent stack frame

Control Flow: for and while

for(*init; boolean_expression; step* **)**

Empty

```
for (int i=0; i<8; i++) ...

int j=0;  for(; j<8; j++) ...

for(int k=7;k>=0; j--) ...
```

Iteration

j++

j=j+1

while(*boolean_expression* **)**

```
int i=0;  while (i<8) { i++; ...}

int j=7; while (j>=0) { j--; ...}
```

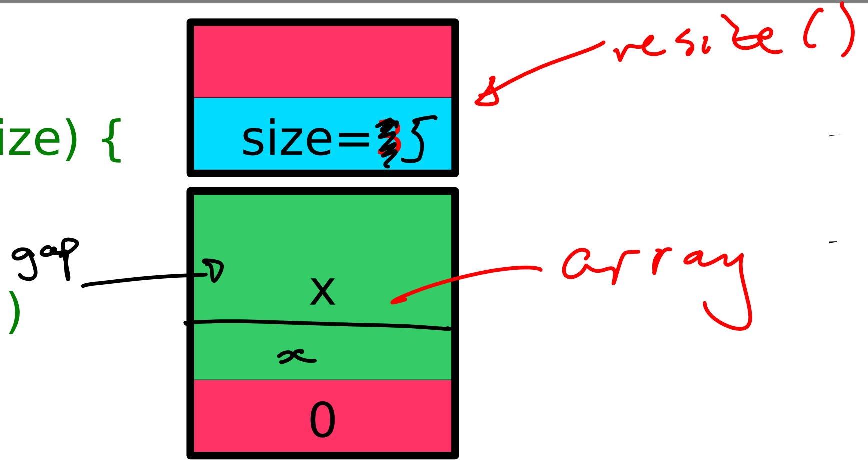
j--

j=j-1

The Heap

```
int[] x = new int[3];
public void resize(int size) {
    int tmp=x;
    x=new int[size];
    for (int i=0; i<3; i++)
        x[i]=tmp[i];
}
resize(5);
```

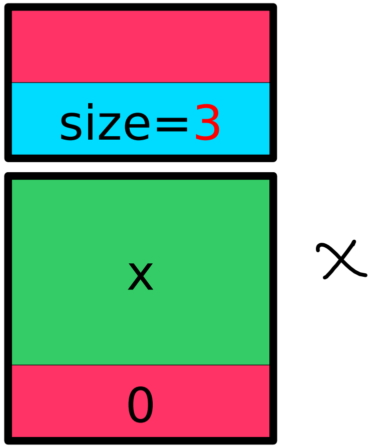
int j=0

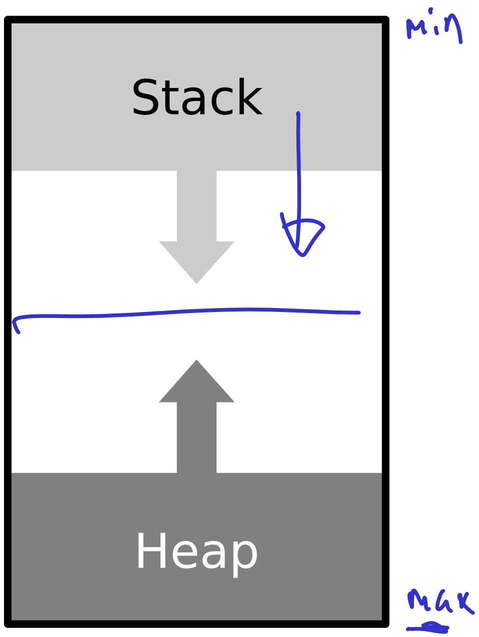


The Heap

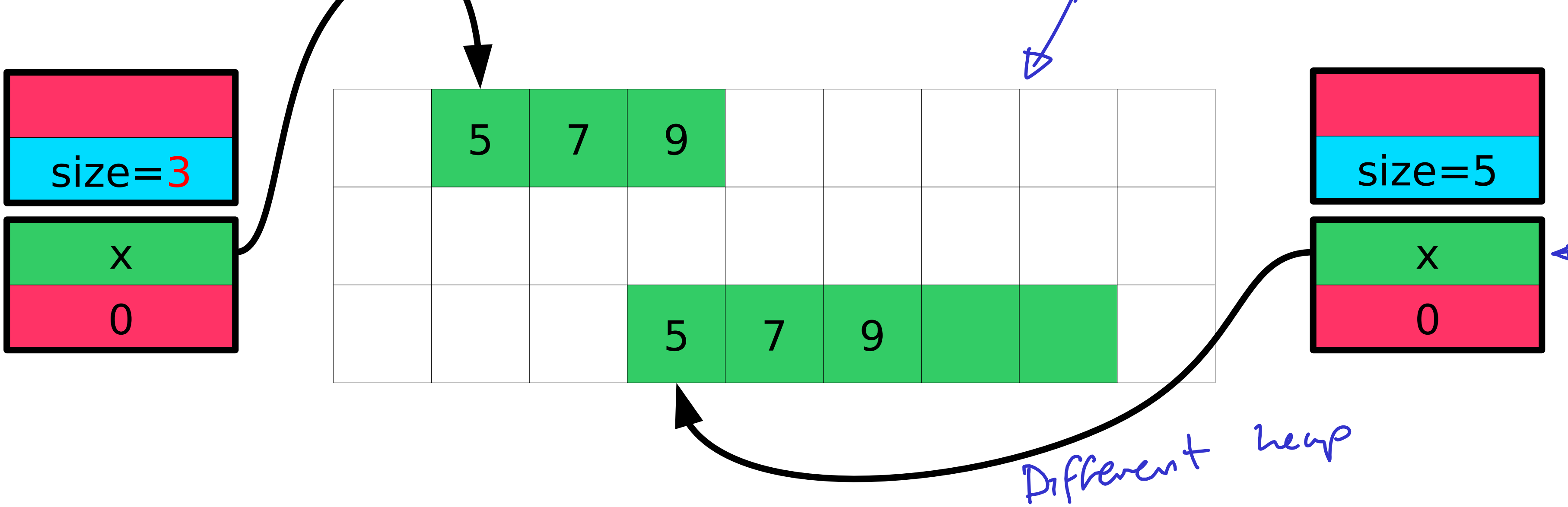
Reference

```
int[] x = new int[3];
public void resize(int size) {
    int tmp=x;
    x=new int[size];
    for (int=0; i<3; i++)
          x[i]=tmp[i];
}
resize(5);
```





Ref. to array



References

- Pointers are useful but dangerous
- References can be thought of as restricted pointers
  - Still just a memory address
  - But the compiler limits what we can do to it
- C, C++: pointers *and* references
- Java: references *only*
- ML: references *only*

References vs Pointers

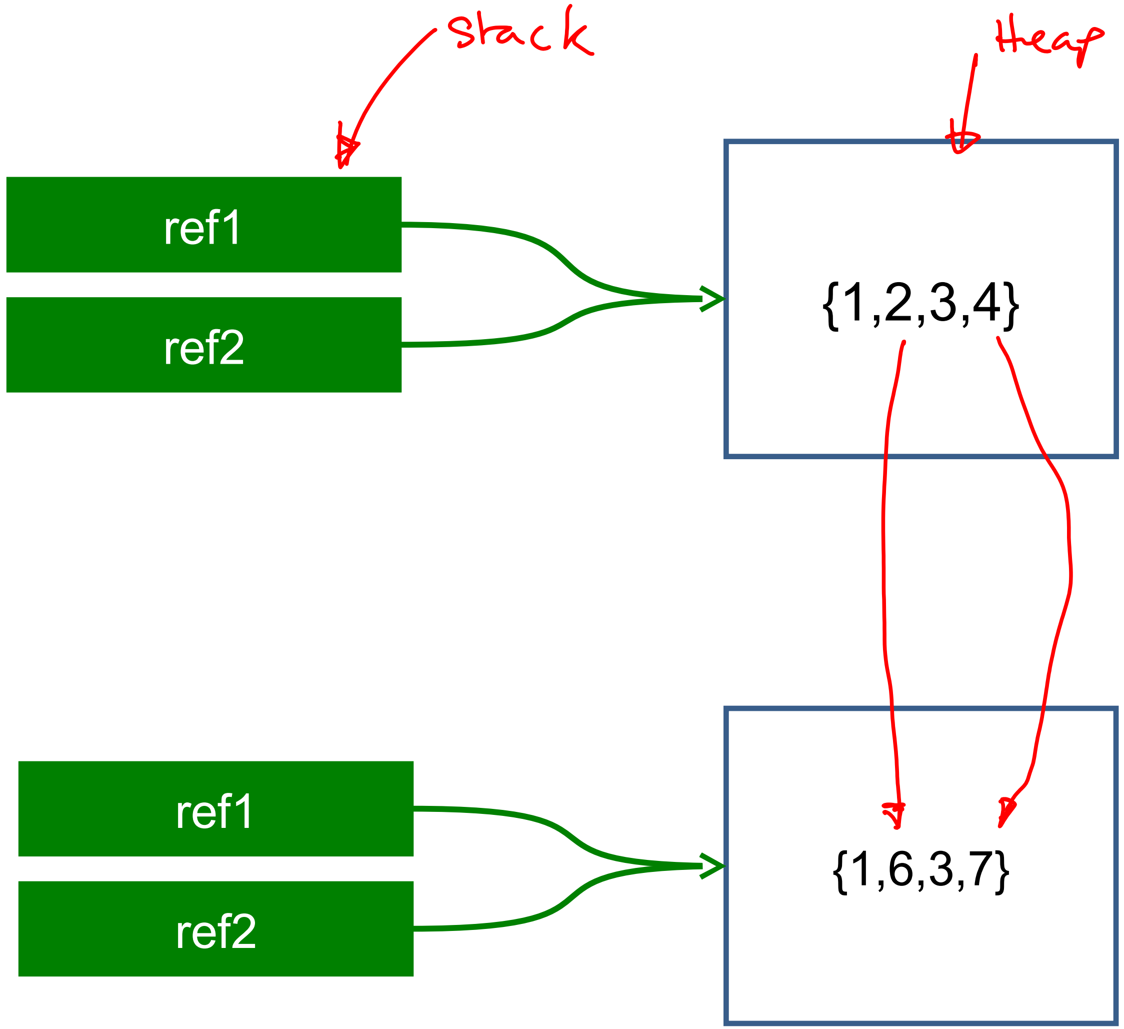
	Pointers	References
Represents a memory address	Yes	Yes
Can be randomly assigned	Yes	No
Can be assigned to established object	Yes	Yes
Can be tested for validity	No	Yes
Pointer arithmetic	Yes	No

Danger

References Example (Java)

Reference

```
int[] ref1 = null;
ref1 = new int[]{1,2,3,4};
int[] ref2 = ref1;
```



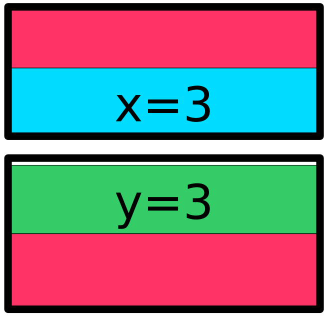
```
ref1[3]=7;
ref2[1]=6;
```

Argument Passing

- **Pass-by-value**. Copy the object into a new value in the stack

Java

```
void test(int x) {…}
int y=3;
test(y);
```



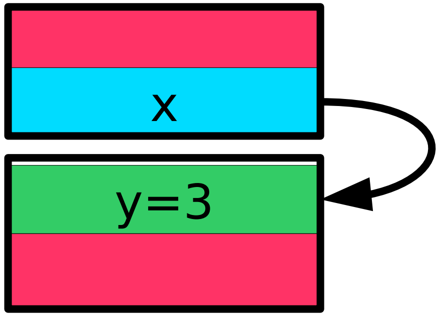
fn() acts on this version

- **Pass-by-reference**. Create a reference to the object and pass that.

C++

```
void test(int &x) {…}
int y=3;
test(y);
```

pass by reference



fn() act on original

Passing Procedure Arguments In Java

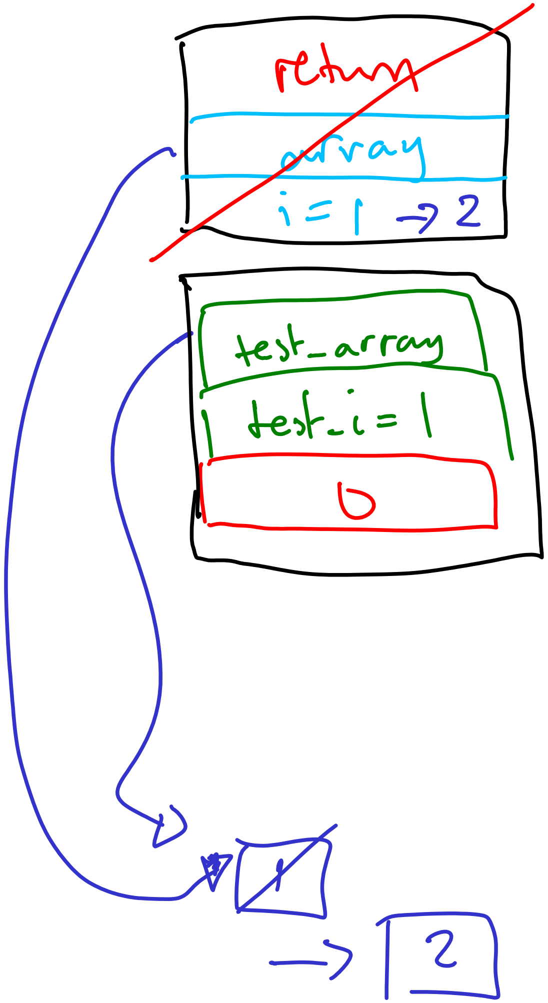
```
class Reference {

   public static void update(int i, int[] array) {
      i++;
      array[0]++;
   }

   public static void main(String[] args) {
      int test_i = 1;
      int[] test_array = {1};
      update(test_i, test_array);
      System.out.println(test_i);
      System.out.println(test_array[0]);
   }

}
```

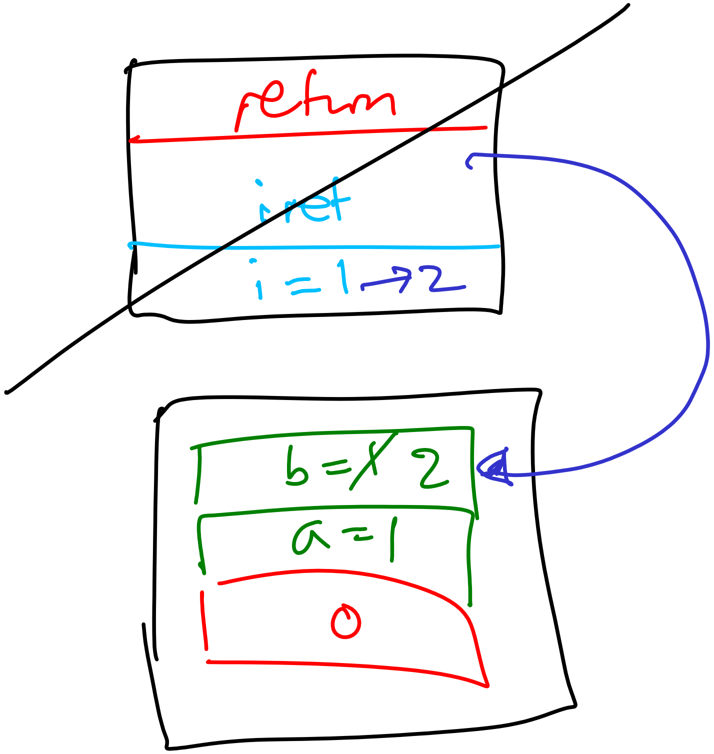
Ref



Passing Procedure Arguments In C++

```
void update(int i, int &iref){
  i++;
  iref++;
}

int main(int argc, char** argv) {
  int a=1;
  int b=1;
  update(a,b);
  printf("%d %d\n",a,b);
}
```

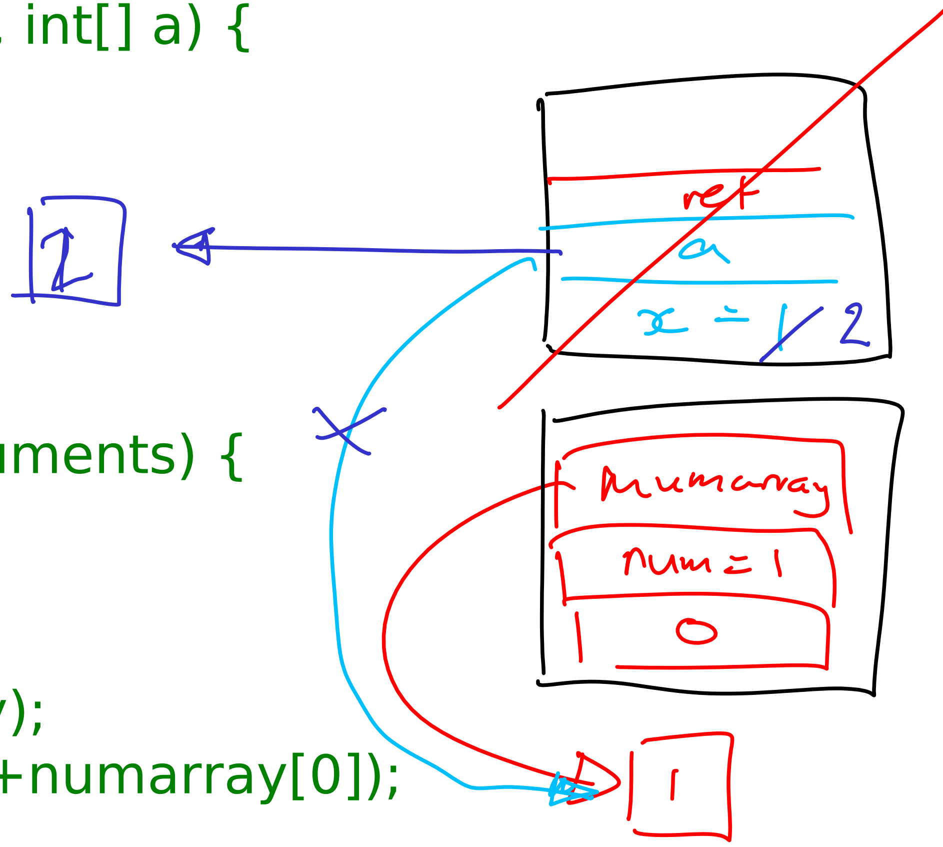


Check…

```
public static void myfunction2(int x, int[] a) {
        x=1;
        x=x+1;
        a = new int[]{1};
        a[0]=a[0]+1;
}

public static void main(String[] arguments) {
        int num=1;
        int numarray[] = {1};

        myfunction2(num, numarray);
        System.out.println(num+" "+numarray[0]);
}
```



A. “1 1” 30
B. “1 2” 10
C. “2 1” 0
D. “2 2” 1

Section: The Java Virtual Machine (JVM)

The Java Approach

- *Java* was born in an era of internet connectivity. SUN wanted to distribute programs to internet machines
  - But many architectures were attached to the internet – how do you write one program for them all?
  - And how do you keep the size of the program small (for quick download)?

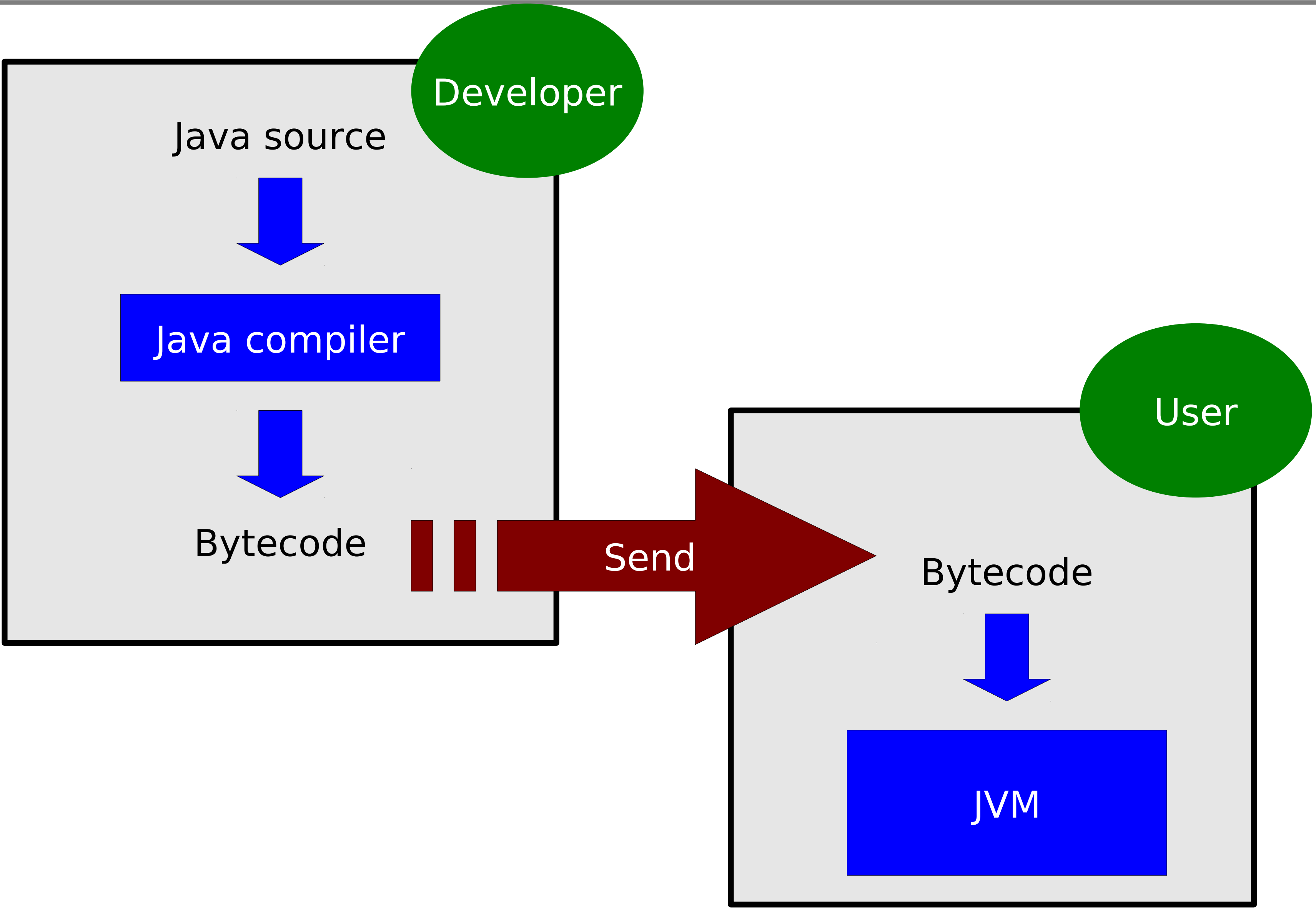
- Could use an interpreter (→ Javascript). But:
  - High level languages not very space-efficient
  - The source code would implicitly be there for anyone to see, which hinders commercial viability.

- Went for a clever hybrid interpreter/compiler

Java Bytecode I

- SUN envisaged a hypothetical Java Virtual Machine (JVM). Java is compiled into machine code (called bytecode) for that (imaginary) machine. The bytecode is then distributed.
- To use the bytecode, the user must have a JVM that has been specially compiled for their architecture.
- The JVM takes in bytecode and spits out the correct machine code for the local computer. i.e. is a bytecode interpreter

Java Bytecode II



Developer

Java source

Java compiler

Bytecode

Send

User

Bytecode

JVM

Java Bytecode III

+ Bytecode is compiled so not easy to reverse engineer

+ The JVM ships with tons of libraries which makes the bytecode you distribute small

+ The toughest part of the compile (from human-readable to computer readable) is done by the compiler, leaving the computer-readable bytecode to be translated by the JVM (→ easier job → faster job)

- Still a performance hit compared to fully compiled (“native”) code