Arrays

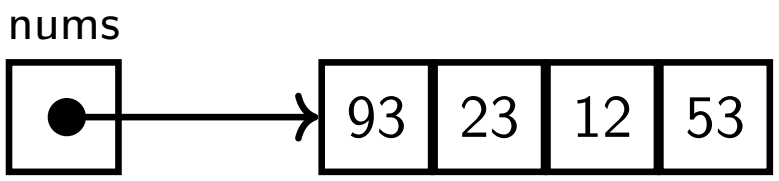
List as an array of a fixed length

The following Java code:

```
1 final int[] nums = new int[4];
```

becomes roughly the following in OS++

```
1 final nums = allocate_memory(4*1);
```



i	Mem[i]
⋮	⋮
3321	"Alice"
3322	"Bob"
3323	93
3324	23
3325	12
3326	53
3327	333
⋮	⋮

array `nums`

final `nums` = 3323

```
1 x = nums[3]; // 53
2 nums[3] = 4;
```

to OS++:

```
1 x = Mem[nums+3];
2 Mem[nums+3] = 4;
```

- `nums` is the address where the array starts in `Mem`; in our case it is equal to 3323.
- Notice that the value of `x` is 53. This is because the indexing of arrays starts from 0, i.e. the indices of elements in `nums` are $0, 1, 2, 3$.
- Every `int` in the array occupies one location in `Mem`. In the next slide we show an example of where every item in the array occupies more than just one location in `Mem`.
- The reason why `a[3]` is translated as `Mem[a+3]` is because Java (or C) knows that the type of `nums` is an array of integers, and that Java knows that an integer takes just one location in memory.

More complicated arrays in Mem[−]

```
1 class Student {
2     String name;
3     int id;
4 }
5 final Student[] students = new Student[3];
```

becomes

```
1 final students = allocate_memory(3*2);
```

i	Mem[i]
⋮	⋮
4029	"Sarah"
4030	1419238
4031	"Berry"
4032	2113812
4033	"Gale"
4034	1322813
⋮	⋮

final students = 4029

Java code:

```
1 students[1].name = "John";
2 students[1].id = 1419231;
```

OS++ code:

```
1 Mem[students+2*1+0] = "John";
2 Mem[students+2*1+1] = 1419231;
```

- To store one `Student` we need to allocate two locations in `Mem`.
- Note that our interpretation of `new Student[3]` is not exactly as in Java. In Java, this code creates an array of three pointers (i.e. addresses) and each of the pointers points to a newly allocated `Student` object.

Memory management

In Java

- memory allocation is automatic
- freeing memory is automatic (by the garbage collector)
- bounds of arrays are checked

In C or C++

- allocations is explicit (similar to OS++ and Mem[-])
- freeing memory is explicit (similar to OS++ and Mem[-])
- bounds are not checked

Java is slower and safe, C (or C++) is fast and dangerous.

A very common mistake is to forget to subtract `1`:

```
final int[] a = new int[5];
a[5] = 1000; // Kaboom!
```

This leads to an `ArrayIndexOutOfBoundsException` in Java whereas in C (or C++) this goes through without a warning and can lead to a corruption of data in memory!

Inserting by shifting in OS++ (NOT JAVA)

To insert a student at position `pos`, where $0 \leq pos \leq size$:

```
1  final Students[] students = new Students[maxsize];
2  int size = 0;     // number of students stored
3
4  void insert(int pos, String name, int id) {
5    if (size == maxsize) {
6        throw new ArrayFullException("Students array");
7    }
8    for (int i=size-1; i >= pos; i--) {
9      // Copy entry in pos i one pos towards the end
10     Mem[students + 2*i + 2] = Mem[students + 2*i    ];
11     Mem[students + 2*i + 3] = Mem[students + 2*i + 1];
12   }
13   Mem[students + 2*pos] = name;
14   Mem[students + 2*pos + 1] = id;
15   size++;
16 }
```

If we want to insert a value to an array (at a certain position) we can do this in two steps:

1. Create a new array, of size bigger by one.
2. Copy elements of the old array to the new one to the corresponding positions.

However, this requires to copy the whole array every single time. Instead, we can allocate a big array at the beginning (of size `maxsize`) and then always "only" shift elements whenever we are inserting/deleting one.