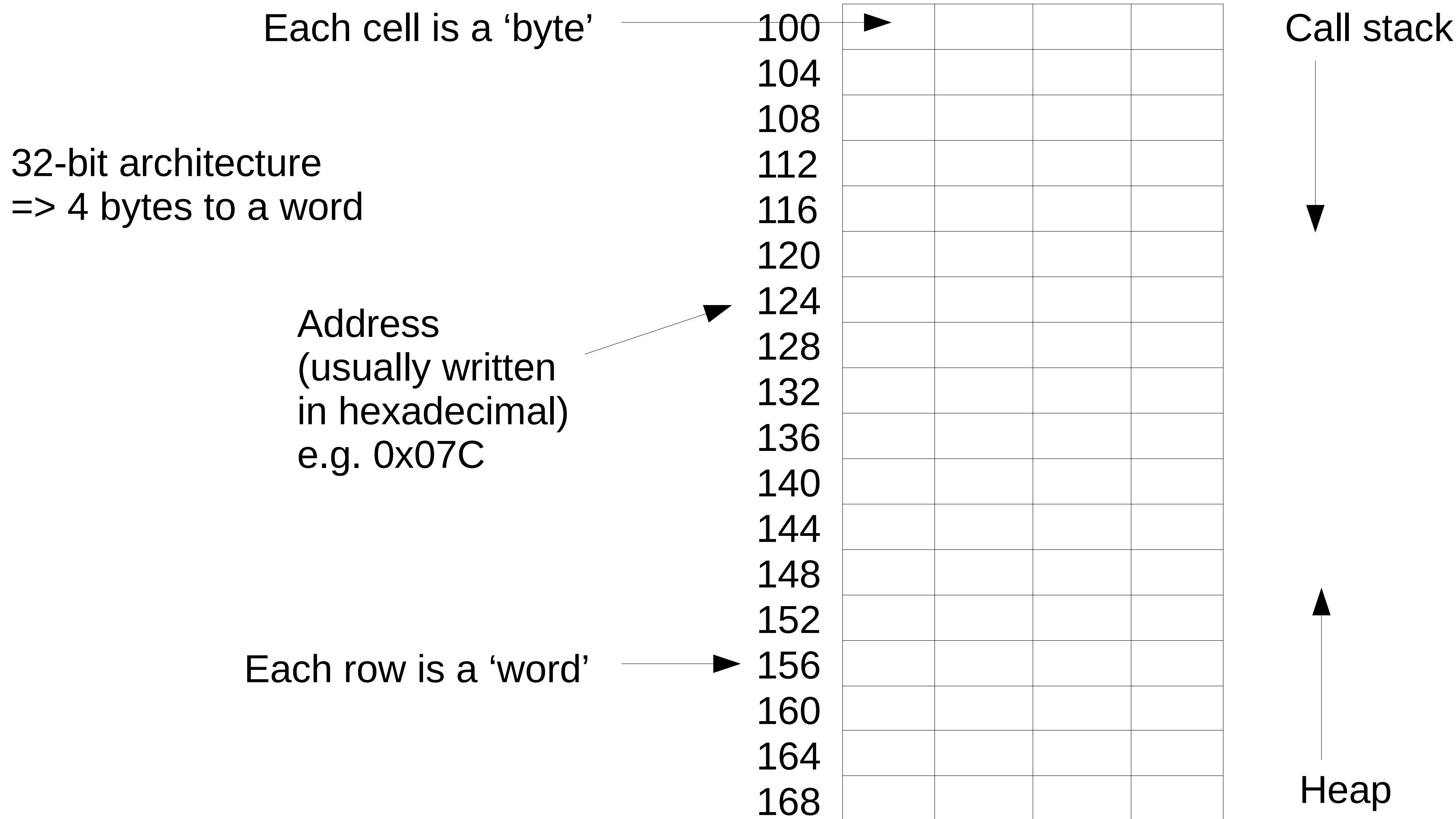
Object Oriented Programming

Additional Handout

Call stacks, heaps and pointers

Andrew Rice
Nov 2019


Each cell is a ‘byte’
Call stack
32-bit architecture
=> 4 bytes to a word
Address
(usually written
in hexadecimal)
e.g. 0x07C
Each row is a ‘word’
Heap
100
104
108
112
116
120
124
128
132
136
140
144
148
152
156
160
164
168

```
    1   void f(int x) {
    2       char c = 'a';
    3       long l = 1234;
    4       int i = 10;
    5   }
    6
>>  7   f(4);
```

100				
104				
108				
112				
116				
120				
124				
128				
132				
136				
140				
144				
148				
152				
156				
160				
164				
168				

 This example is in C/C++

```
>>  1   void f(int x) {
    2       char c = 'a';
    3       long l = 1234;
    4       int i = 10;
    5   }
    6
    7   f(4);
```

Address					Variable
100	4	0	0	0	x
104					c
108					l
112					
116					i
120					
124					
128					
132					
136					
140					
144					
148					
152					
156					
160					
164					
168					

```
   1  void f(int x) {
>> 2      char c = 'a';
   3      long l = 1234;
   4      int i = 10;
   5  }
   6
   7  f(4);
```

Address					Variable
100	4	0	0	0	x
104	97	.	.	.	c
108					l
112					
116					i
120					
124					
128					
132					
136					
140					
144					
148					
152					
156					
160					
164					
168					

```
    1  void f(int x) {
    2      char c = 'a';
>>  3      long l = 1234;
    4      int i = 10;
    5  }
    6
    7  f(4);
```

1234 is bigger than one byte

1234 & 0xFF = 210
1234 >> 8 = 4

Address					
100	4	0	0	0	x
104	97	.	.	.	c
108	210	4	0	0	l
112	0	0	0	0	
116					i
120					
124					
128					
132					
136					
140					
144					
148					
152					
156					
160					
164					
168					

```
   1   void f(int x) {
   2       char c = 'a';
   3       long l = 1234;
>> 4       int i = 10;
   5   }
   6
   7   f(4);
```

Address					Variable
100	4	0	0	0	x
104	97	.	.	.	c
108	210	4	0	0	l
112	0	0	0	0	
116	10	0	0	0	i
120					
124					
128					
132					
136					
140					
144					
148					
152					
156					
160					
164					
168					

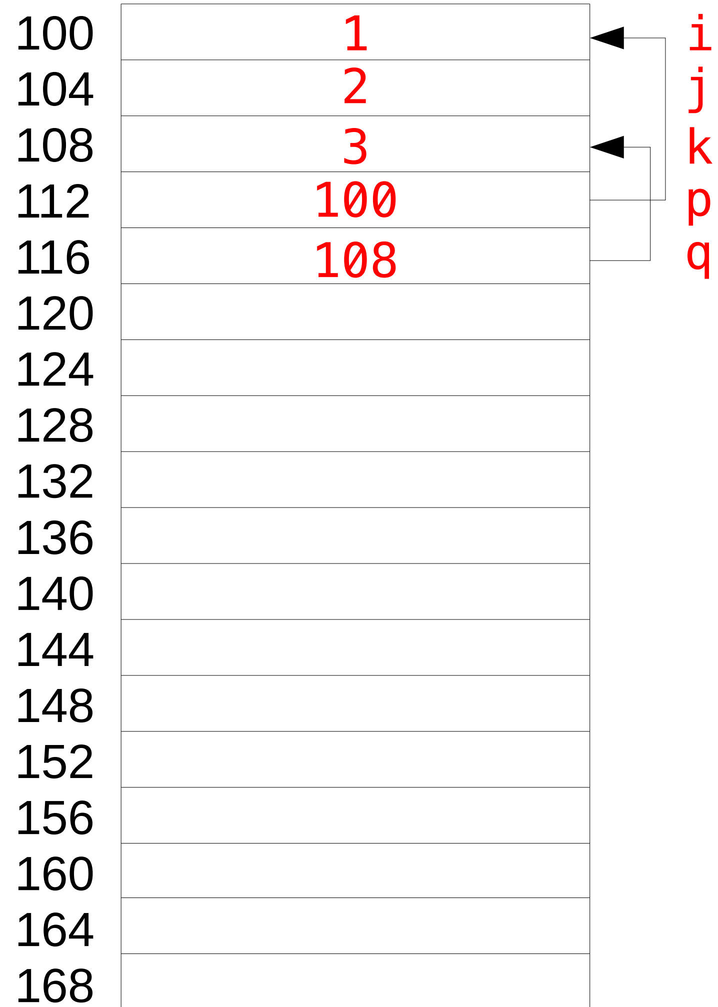
```
   1  void f(int x) {
   2      char c = 'a';
   3      long l = 1234;
>> 4      int i = 10;
   5  }
   6
   7  f(4);
```

Address	Value	Variable
100	4	x
104	‘a’	c
108	1234	l
112		
116	10	i
120		
124		
128		
132		
136		
140		
144		
148		
152		
156		
160		
164		
168		

```
1   void f() {
2         int i = 1;
3         int j = 2;
4         int k = 3;
5         int* p = &i;
6         int* q = &k;
7   }
```

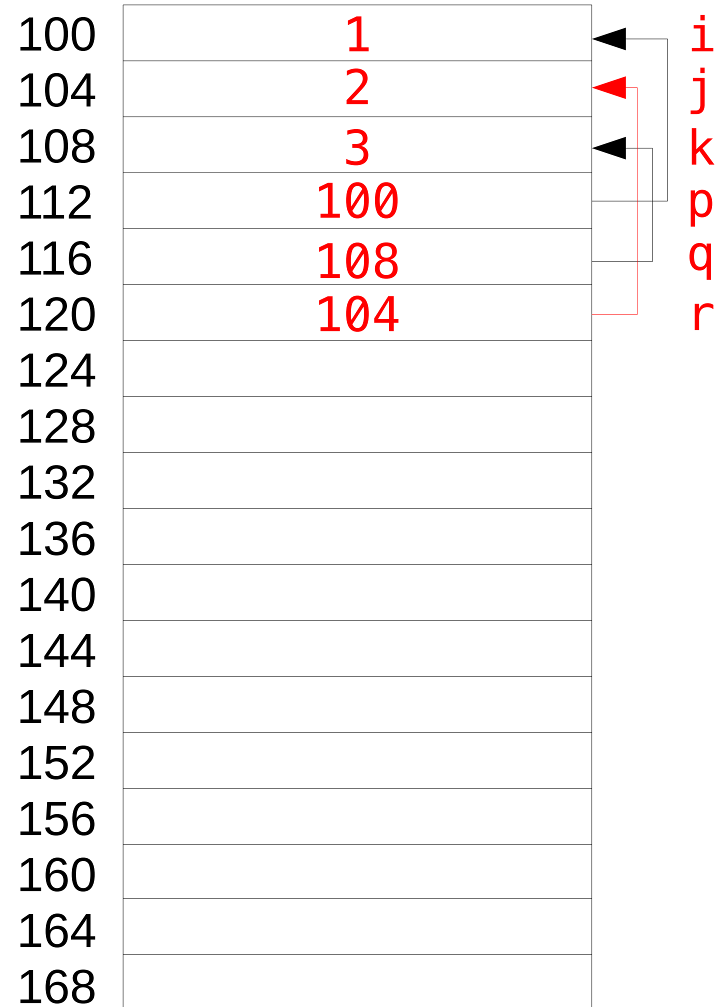
* on a LHS means
‘its a pointer’

& on a RHS means
‘take the address of’



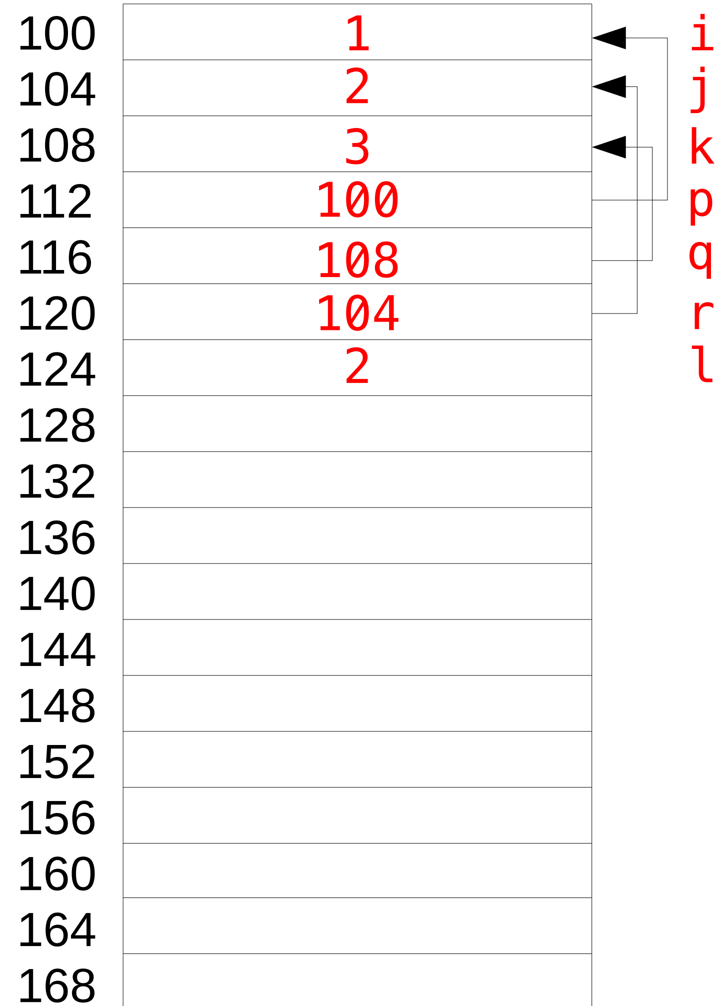
```
1   void f() {
2         int i = 1;
3         int j = 2;
4         int k = 3;
5         int* p = &i;
6         int* q = &k;
7         int* r = p + 1;
8   }
```

We can do arithmetic on pointers (based on the datatype size)



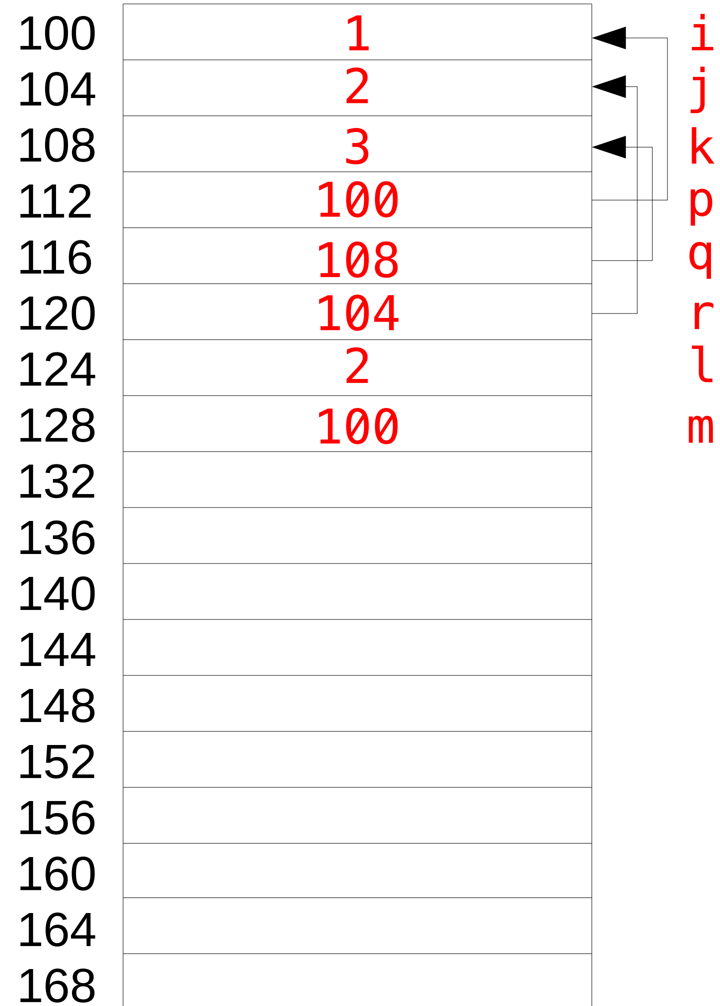
```
1  void f() {
2        int i = 1;
3        int j = 2;
4        int k = 3;
5        int* p = &i;
6        int* q = &k;
7        int* r = p + 1;
8        int l = *r;
   }
```

* on the RHS means ‘dereference’ i.e. follow the pointer.



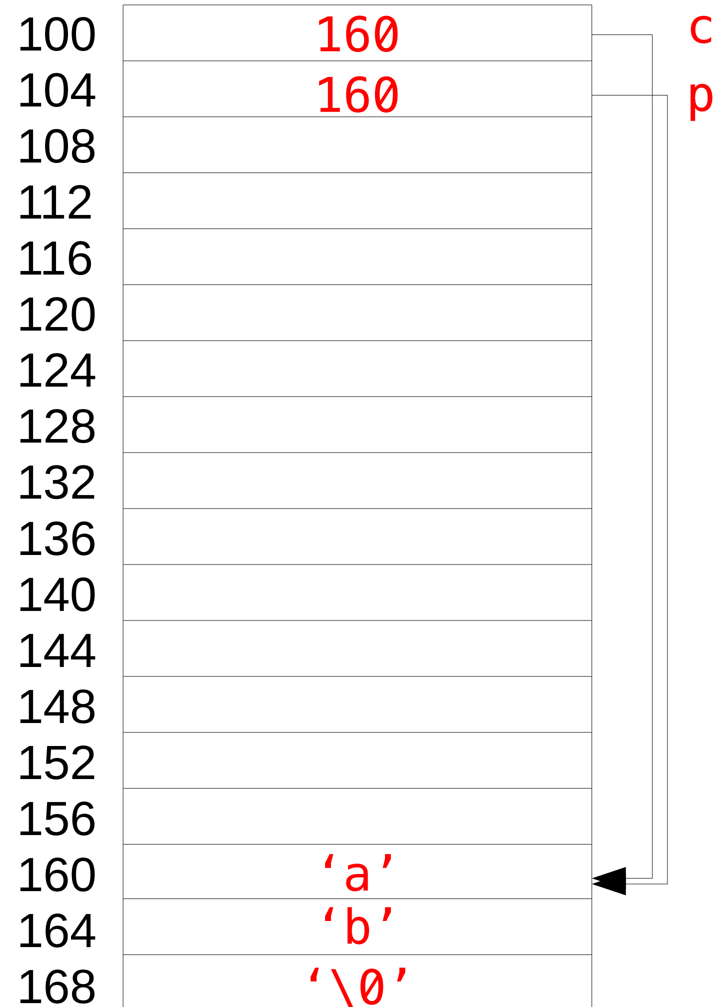
```
1   void f() {
2       int i = 1;
3       int j = 2;
4       int k = 3;
5       int* p = &i;
6       int* q = &k;
7       int* r = p + 1;
8       int l = *r;
9       int m = *(q + 1);
10  }
```

Nothing will stop you making mistakes!



```
1   int len() {
2       char[] c = new[]
3           {'a','b','\0'};
4       char* p = c;
5   }
6
7
8
9
```

In C++ you can choose whether you want to allocate on the stack or the heap



Items on the stack exist only for the duration of your function call

Items on the heap exist until they are deleted

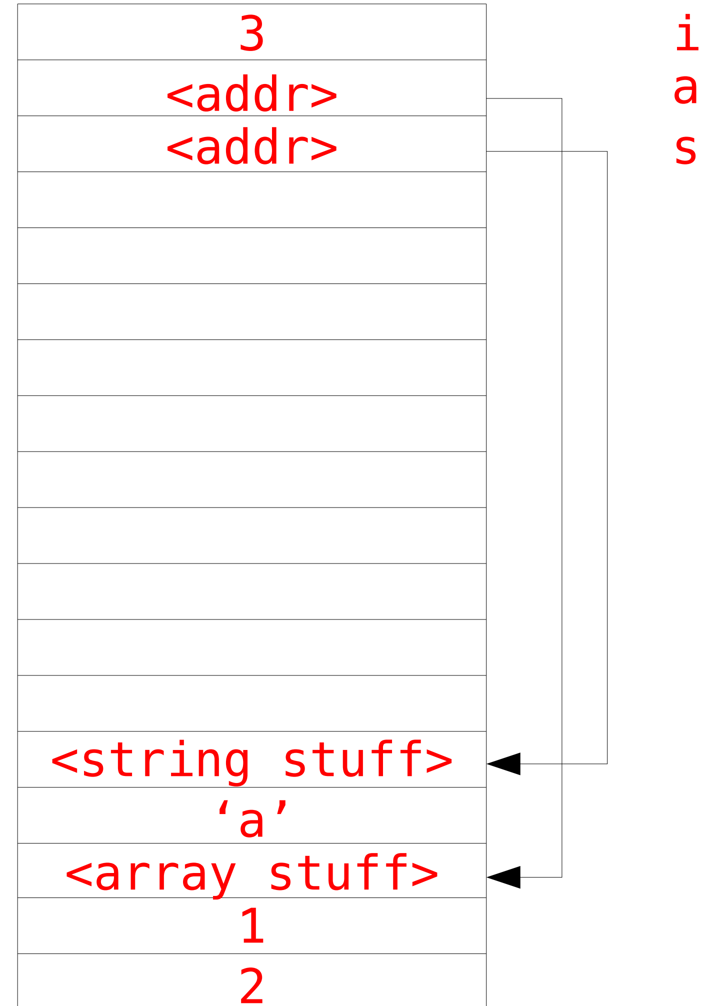
Now go back to the main notes!

In Java primitive types go on the stack

Everything else goes on the heap

```
1  static void test() {
2       int i = 3;
3       int[] a = new int[] {1,2};
4       String s = "a";
5  }
```

Java delete's for us automatically. This is called Garbage Collection



This example is in Java

```java
1  static void test() {
2      int i = 3;
3      int[] a = new int[] {1,2};
4      String s = "a";
5  }
```

‘a’ and ‘s’ are references. These are like pointers but you can’t do arithmetic on them.

When you say s.toUpperCase() you are ‘dereferencing’ s and calling the method toUpperCase on it.

References in C++ are a completely different concept!

```
   1  static void test() {
>> 2      int i = 3;
   3      int* k = &i;
   4      int& j = i;
   5  }
```

3	i

This example is in C++

```
>>
1 static void test() {
2     int i = 3;
3     int* k = &i;
4     int& j = i;
5 }
```

```
1  static void test() {
2        int i = 3;
3        int* k = &i;
4        int& j = i;
5  }
```

>>

& on the LHS means ‘reference’

Recap for Java

- Primitive types on the stack
- Everything else on the heap
- References are values on the stack that ‘point’ to somewhere on the heap
- References are like pointers but you can’t do artithmetic on them
- Java references are not much like C++ references