Week 9: References, the Heap and the Stack
Last Week: Learning Outcomes

From last week you should be able to:

- write programs that are split into methods
- write programs that pass information to methods using parameters
- explain what is meant by a method and parameters
Methods

A program from last week

```java
static String print_array (int[] myarray) {
    String array_string = "";
    int length = myarray.length;
    for (int i=0; i<length; i++) {
        array_string = array_string +
                       myarray[i] + " ";
    }
    return array_string;
}
```

And then elsewhere in the program..a call:

```java
output out = new output();
....
out.writeln(print_array(myarray));
....
out.close();
```
This Week: Learning Outcomes

By the end of the week you should be able to:

- compare and contrast the way integers and arrays are stored
- explain how information is stored in memory in the heap or stack
- explain how array information is passed to methods
- explain and compare and contrast pass by reference and pass by value
Arrays

Strange things can appear to happen when you use arrays...

In order to work out what’s happening we need to understand how Java handles arrays and other similar datastructures.
Exercise 1

What does the following program fragment print?

```java
int a = 1;
int b = 2;

b = a;

out.writeln(a);
out.writeln(b);

a = 5;

out.writeln(a);
out.writeln(b);
```
What does assignment do?

For integers an assignment

\[ b = a; \]

makes a copy of what was in one variable and puts it in the other.

It takes the contents of \( a \) and puts it in \( b \).

\( a \) and \( b \) are still different.
Exercise 2

What does the following program fragment print?

```java
int [] a = {1};
int [] b = {2};

b = a;

out.writeln(a[0]);
out.writeln(b[0]);

a[0] = 5;

out.writeln(a[0]);
out.writeln(b[0]);
```
What does array assignment do?

After an array assignment: \( b = a \);

\( a \) and \( b \) now contain the same things. But also...

Change an entry of \( a \) ... \( b \) changes.

Change an entry of \( b \) ... \( a \) changes.

So it looks as if \( a \) and \( b \) here refer to the same bits of storage - the same boxes.

... but if \( a \) and \( b \) were integers not arrays they would refer to different bits of storage.
Storage for arrays

So it looks as if there’s something different about how arrays are stored.

What’s the story?

I will start with a simplified view then give a bit more detail...
We have talked of variables as being like boxes. Different variables are different boxes. Assignment just puts copies of values in the appropriate box.

An array is more like a pair of distinct boxes. The array variable does not actually hold the array. Instead it holds information (known as a Reference) of where to find the array data.

```c
int i = 4;
int [] a = {1,2,3,4};
```

![Diagram showing the relationship between variables and array elements]
Using this picture of how arrays are stored we can explain the way assignment works:

Assignments to the array variable, \( a \), change what is in the box \( a \). They make the arrow point somewhere else.

Assignments to array \textit{elements} of a follow the arrow to where the actual data is stored and change that.
Exercise

Trace the execution of the following code as a series of box and arrow pictures.

```c
int a[] = {1,2};
int b[] = {3,4};
a = b;
a[0] = 5;
```
int a [] = {1,2};

int b [] = {3,4};

a = b;

a[0] = 5;
Exercise

Trace the execution of the following code as a series of box and arrow pictures.

```plaintext
int a[] = {1,2};
int b[] = {3,4};
int t[];

t = a;
a = b;
b = t;
```
Exercise

Understanding how arrays are implemented also explains an otherwise weird result when they are tested.

What does the following print to the screen?

```java
int a[] = {1,2};
int b[] = {1,2};

if (a==b) out.writeln( "a and b are equal");
else out.writeln( "a and b are NOT equal");

a = b;

if (a==b) out.writeln( "a and b are equal");
else out.writeln( "a and b are NOT equal");
```
Equality of arrays

Initially the two arrays are not “equal” but then become “equal” after the assignment.

So it looks as if \( a == b \) checks if \( a \) and \( b \) refer to the same bits of storage, not if the contents are the same.

\( a == b \) tests the contents of the array variables (the references). If both \( a \) and \( b \) are pointing to the same place they are equal otherwise they are not.
Java Storage

Why are arrays stored like this?

Java’s storage is divided into two parts.

They are called the *stack* and the *heap*.

They contain different sorts of things.
The Stack

The stack holds local variables and method parameters.

When the program enters a block with a local variable declaration, storage for that variable is created on the stack. When the program leaves the block that storage is freed.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>pi</td>
<td>3.14159</td>
</tr>
<tr>
<td>ch</td>
<td>'c'</td>
</tr>
<tr>
<td>b</td>
<td>true</td>
</tr>
<tr>
<td>n</td>
<td>21</td>
</tr>
</tbody>
</table>
The Heap

The Heap holds arrays and “objects”.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s</th>
<th>&quot;hello&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>d[]</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>c[]</td>
<td>‘a’ ‘b’ ‘c’</td>
</tr>
<tr>
<td>b[]</td>
<td>1 2 3 4</td>
</tr>
</tbody>
</table>
The Heap

The Heap holds arrays and “objects” (e.g. Strings).

The heap holds storage that is allocated through the use of the keyword “new”, as opposed to some kind of variable declaration.

(The use of new can sometimes be implicit.)

Things on the heap can live longer than the block in which they are created.
Array allocation

So what happens when we declare and allocate an array?

```java
int a[] = new int[2];
```

is the same as

```java
int a[];
a = new int[2];
```
Array allocation

```java
int a[];
```

```java
a = new int[2];
```

```java
int a[];
```

is the declaration of a new local variable a.

The rules say it goes on the stack.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>s</td>
</tr>
<tr>
<td>pi</td>
<td>&quot;hello&quot;</td>
</tr>
<tr>
<td>ch</td>
<td>f[]</td>
</tr>
<tr>
<td>b</td>
<td>e[]</td>
</tr>
<tr>
<td>n</td>
<td>d[]</td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pi</td>
<td>3.14159</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ch</td>
<td>'c'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>true</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f[]</td>
<td>0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e[]</td>
<td>'a''b''c'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d[]</td>
<td>1 2 3 4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Array allocation

int a[];
a = new int[2];

new int[2]
allocates a new array of length 2.

The rules say it lives on the heap.
Array allocation

But this means that the array variable is storage on the stack, and the array is storage on the heap!!!

But fortunately we haven’t finished.

That is where the references come in.
Array allocation

```java
int a[];
a = new int[2];

a = new int[2];
puts something in the variable a.

What?
```
Array allocation

The location (address) of the array we’ve just created on the heap.
Array allocation

Now we can see why we got that strange behaviour from arrays.

Let's look at assignment of, say, characters.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>-[]</td>
</tr>
<tr>
<td>pi: 3.14159</td>
<td>s: &quot;hello&quot;</td>
</tr>
<tr>
<td>ch2: 'c'</td>
<td>-[] 0 0 0 0 0</td>
</tr>
<tr>
<td>ch1: 'b'</td>
<td>-[] 'a' 'b' 'c'</td>
</tr>
<tr>
<td>n: 21</td>
<td>-[] 1 2 3 4</td>
</tr>
</tbody>
</table>
Array allocation

```java
ch1 = ch2;
```

puts the value (the contents) of `ch2` in `ch1`.

So we get

```
Stack

<table>
<thead>
<tr>
<th>a</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>pi</td>
<td>3.14159</td>
<td></td>
</tr>
<tr>
<td>ch2</td>
<td>'c'</td>
<td></td>
</tr>
<tr>
<td>ch1</td>
<td>'c'</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>21</td>
<td></td>
</tr>
</tbody>
</table>

Heap

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[-]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>&quot;hello&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[-]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>[-]</td>
<td>'a''b''c'</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[-]</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
```
Array allocation

Now, if we have two arrays:

<table>
<thead>
<tr>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>-[]</td>
</tr>
<tr>
<td>a</td>
<td>-[]</td>
</tr>
<tr>
<td>pi</td>
<td>3.14159</td>
</tr>
<tr>
<td>ch2</td>
<td>'c'</td>
</tr>
<tr>
<td>ch1</td>
<td>'c'</td>
</tr>
<tr>
<td>n</td>
<td>21</td>
</tr>
<tr>
<td>s</td>
<td>&quot;hello&quot;</td>
</tr>
<tr>
<td></td>
<td>-[]</td>
</tr>
<tr>
<td></td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>-[]</td>
</tr>
<tr>
<td></td>
<td>'a''b''c'</td>
</tr>
<tr>
<td></td>
<td>-[]</td>
</tr>
<tr>
<td></td>
<td>1 2 3 4</td>
</tr>
</tbody>
</table>
Array allocation

\[ b = a; \]

takes the contents of \( a \) (the address) and puts it in \( b \).

So both \( a \) and \( b \) contain the location of the same array.

<table>
<thead>
<tr>
<th></th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td>-[]</td>
</tr>
<tr>
<td>a</td>
<td></td>
<td>-[]</td>
</tr>
<tr>
<td>pi</td>
<td>3.14159</td>
<td></td>
</tr>
<tr>
<td>ch2</td>
<td>'c'</td>
<td>-[]</td>
</tr>
<tr>
<td>ch1</td>
<td>'c'</td>
<td>-[]</td>
</tr>
<tr>
<td>n</td>
<td>21</td>
<td>-[]</td>
</tr>
</tbody>
</table>

- [X] Stack and Heap Diagram
Array allocation

This explains everything we observed in our experiments tracing code fragments.

In Java things stored in stack variables are very simple: either values of basic type (char, int, float, boolean, double, long, ...) or the locations of things on the heap. They are short and have known size.

Things stored on the heap can be much more complicated (arrays, Strings, objects).

This is a somewhat simplified account: things on the heap can contain the locations of other things on the heap. For example, if you have an array of arrays, or an array of Strings.
String equality again

Strings are stored on the heap. Their equality is determined by that. Some storage optimisation happens so that you get equalities you shouldn’t expect.

See experiment_5.java
Strings

By the way, ...

You can’t change Strings in Java.

That is you can’t change the bit on the heap, in the way that you can change a cell in an array.

Operations that look as if they’re changing Strings are actually creating new Strings (and assigning their locations to String variables).
Methods

What does this have to do with methods?

Quite a lot.

Let’s start with procedures.
Array parameters

Consider the two following methods:

```java
static void array_inc(int[] array, int index) {
    array[index]++;  
}
```

```java
static void int_inc (int x) {
    x++;  
    x++;  
}
```

What do they do?
Array parameters

Remember the rules about a call being equivalent to substituting the method body in its place...The rules say that a call to \texttt{int\_inc}:

\begin{verbatim}
int n=1;
int\_inc(n);
\end{verbatim}

is equivalent to:

\begin{verbatim}
int n=1;
{ int x=n;
  x++;
}
\end{verbatim}

This does not change \texttt{x}.
int n=1;
int_inc(n);

int n=1;
{
    int x=n;
    x++;
    x++;
}
int n=1;
int inc(n);

int n=1;
{ int x=n;
    x++;
}

<table>
<thead>
<tr>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
</tr>
</tbody>
</table>
int n=1;
int_inc(n);

int n=1;
{ int x=n;
   x++;
}

Stack

<table>
<thead>
<tr>
<th>x</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>1</td>
</tr>
</tbody>
</table>

Heap
int n=1;
int inc(n);

int n=1;
{ int x=n;
  x++;
}

Stack

Heap

| n | 1 |
Array parameters

On the other hand consider a call to `array_inc`:

```java
int a[] = new int[1];
a[0] = 1;
array_inc(a, 0);
```

is equivalent to:

```java
int a = new int[1];
a[0] = 1;
{
    int array[] = a;
    int index = 0;
    array[index]++;
}
```

This does change `a[0]`. 

int a[] = new int[1];
a[0] = 1;
array_inc(a, 0);

int a = new int[1];
a[0]=1; { int array[] = a;
   int index = 0;
   array[index]++;
}

Stack

Heap

a

[]
int a[] = new int[1];
a[0] = 1;
array_inc(a, 0);

int a = new int[1];
a[0] = 1;
{
    int array[] = a;
    int index = 0;
    array[index]++;
}

Stack

Heap

a

[] 1
int a[] = new int[1];
a[0] = 1;
array_inc(a, 0);

int a = new int[1];
a[0] = 1;
{
    int array[] = a;
    int index = 0;
    array[index]++;
}
int a[] = new int[1];
a[0] = 1;
array_inc(a, 0);

int a = new int[1];
a[0] = 1;
{
    int array[] = a;
    int index = 0;
    array[index]++;
}

Stack

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
</tr>
</tbody>
</table>

Heap

[-[]] 1
int a[] = new int[1];
a[0] = 1;
array_inc(a, 0);

int a = new int[1];
a[0] = 1;
{ int array[] = a;
  int index = 0;
  array[index]++;
}

Stack

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
</tr>
</tbody>
</table>

Heap

[2]
int a[] = new int[1];
a[0] = 1;
array_inc(a, 0);

int a = new int[1];
a[0] = 1;
{ int array[] = a;
  int index = 0;
  array[index]++;}

Stack

Heap

a

-[]  2
Array parameters

Similarly:

```c
static void int_swap(int x, int y)
{
    int tmp = x;
    x = y;
    y = tmp;
}
```
does nothing.

Whereas

```c
static void array_swap(int a[], int x, int y)
{
    int tmp = array[x];
    array[x] = array[y];
    array[y] = tmp;
}
```
does swap the elements of the array.

This means you can write procedures that sort an array.
Arrays and functions

Finally you can of course return arrays from functions.

```java
static int[] create(int length, int init)
{
    int a[] = new int[length];
    for (int i=0; i<length; i++)
    {
        a[i] = init;
    }
    return a;
}
...
int myarray[] = create(5,1);
...
```

Note that this creates the array, and initialises it.

You can run through the storage model for yourselves!
This Week: Learning Outcomes

By the end of the week you should be able to:

- compare and contrast the way integers and arrays are stored
- explain how information is stored in memory in the heap or stack
- explain how array information is passed to methods
- explain and compare and contrast pass by reference and pass by value

Reading Computing Without Computers Chapter 7

Brinch Hansen:
pp 143-166: Chapter 9 Methods