INFO1113 Object-Oriented Programming

Week 2B: Contiguous memory
Arrays and Strings

Slides developed by Tyson Thomas

Copyright Warning

Topics

Arrays and initialisation

An array is a ***contiguous block of memory*** containing multiple values of the same type.

When an array is initialised the array will be allocated and will return the address of where the array is stored.

We are also able to solve the problem of having many variables.

```
variable01 = 32;
variable02 = 98;
variable03 = 34;
...
variable98 = 23;
```

We can access the 98th element if we have an array of size 100.

Array Initialisation

Within the java language we are able to intiailise an array in a few different ways. Most commonly and what is typically used is allocate and specify size.

```
int[] numbers = new int[16];
```

Array Initialisation

Within the java language we are able to intiailise an array in a few different ways. Most commonly and what is typically used is allocate and specify size.

```java
int[] numbers = new int[16];
```

However we can always initialise using **static initialisation.**

```java
int[] numbers = {1, 2, 3, 4};
```

This translate to an array of length 4, containing the elements **1, 2, 3 and 4.**
Similarly, we can initialise an array like so.

```java
int[] numbers = new int[] {1, 2, 3, 4};
```

Array Initialisation

Within the java language we are able to intiailise an array in a few different ways. Most commonly and what is typically used is allocate and specify size.

```
int[] numbers = new int[16];
```

However we can always initialise using **static initialisation.**

```
int[] numbers = {1, 2, 3, 4};
```

This translate to an array of length 4, containing the elements **1, 2, 3 and 4.**
Similarly, we can initialise an array like so.

```
int[] numbers = new int[] {1, 2, 3, 4};
```

More commonly used when passing an array with values known at compile time. This is because the compiler knows the type being passed and what values it should contain.

Refer to Chapter 7.1, pages 525-541, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Array Layout

So let's consider the layout.

Since we know it returns the starting point of the array. How does it navigate the array?

Let's say we have the current array:

```java
int[] numbers = new int[16];
```

And contains the following numbers:

2	8	3	4	90	12	45	32	43	76	1	-9	2	44	65	78

What are the addresses of each number?

Refer to Chapter 7.1, pages 525-541, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Array Layout

So let's consider the layout.

Since we know it returns the starting point of the array. How does it navigate the array?

Let's say we have the current array:

```java
int[] numbers = new int[16];
```

And contains the following numbers:

2	8	3	4	90	12	45	32	43	76	1	-9	2	44	65	78

0x1000

Let's assume the array we have received is allocated at 0x1000 (address 4096), since an int is 4 bytes, what is the address of element at **index 1**

What are the addresses of each number?

Refer to Chapter 7.1, pages 525-541, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

So let's consider the layout.

Since we know it returns the starting point of the array. How does it navigate the array?

Let's say we have the current array:

```java
int[] numbers = new int[16];
```

And contains the following numbers:

2	8	3	4	90	12	45	32	43	76	1	-9	2	44	65	78

0x1000 0x1004

Since an int is 4 bytes and we are accessing index 1, we will move 4 bytes from starting address.
0x1000 + 4 = 0x1004. When reading the value here, we are dereferencing

What are the addresses of each number?

Refer to Chapter 7.1, pages 525-541, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Array Layout

So let's consider the layout.

Since we know it returns the starting point of the array. How does it navigate the array?

Let's say we have the current array:

```java
int[] numbers = new int[16];
```

And contains the following numbers:

2	8	3	4	90	12	45	32	43	76	1	-9	2	44	65	78

0x1000 0x1004 0x100C

0x1008

Since an int is 4 bytes and we are accessing index 1, we will move 4 bytes from starting address.
0x1000 + 4 = 0x1004. When reading the value here, we are dereferencing

What are the addresses of each number?

Refer to Chapter 7.1, pages 525-541, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Array Layout

So let's consider the layout.

Since we know it returns the starting point of the array. How does it navigate the array?

Let's say we have the current array:

```
int[] numbers = new int[16];
```

This statement carries a lot of weight with it.

And contains the following numbers:

2	8	3	4	90	12	45	32	43	76	1	-9	2	44	65	78

0x1000 0x1004 0x100C

0x1008

What are the addresses of each number?

Refer to Chapter 7.1, pages 525-541, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Array Layout

So let's consider the layout.

Since we know it returns the starting point of the array. How does it navigate the array?

Let's say we have the current array:

```java
int[] numbers = new int[16];
```

Knowing the **type** and the **size** and using the **new** operator, the compiler derive the idea that we want to **dynamically allocate 16 integers.**

And contains the following numbers:

2	8	3	4	90	12	45	32	43	76	1	-9	2	44	65	78

0x1000 0x1004 0x100C

0x1008

What are the addresses of each number?

Refer to Chapter 7.1, pages 525-541, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Reference and primitive type arrays

We have seen primitive type arrays but what about **reference** type arrays?

These types of arrays have a *slightly* difference semantic meaning behind them.

Using the **String** type as an example.

```
String[] numbers = new String[4];
```

Since it is a **reference** type it is initialising **4 references.** A reference in this instance is a **memory address**.

This infers that **reference type** arrays do **not contain** a string but a reference to a string.

Refer to Chapter 7.1, pages 525-541, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Reference and primitive type arrays

General rules with array initialisation

- Primitive integer types (byte, short, int, long) are initialised to **0 by default.**
- Default value of elements of a boolean array are **false.**
- Floating point numbers such as **float** and **double** are initialised to **0.0f** and **0.0d** respectively.
- Elements of a char array is initialised to **\u0000** (**null** character)
- Any **Reference type** is initialised to **null.**

Refer to Oracle Java Language Specification, (https://docs.oracle.com/javase/specs/jls/se8/html/jls-4.html#jls-4.12.5)

Demonstration:
Array manipulation

Multi-dimensional arrays

We are not limited just creating single dimensional arrays. We are able to create **multi-dimensional** arrays.

```
int[][] array = new int[3][3];
```

There are two types, one adheres to a **matrix-like structure** and the other is commonly referred to as a **jagged array.**

```
int[][] array = new int[3][];
```

Arrays are also **reference** types. When initialised, the variable array will contain **3 null elements.** We are able to specify lengths on each elements.

```
array[0] = new int[5];
array[1] = new int[10];
```

Refer to Oracle Java Language Specification, (https://docs.oracle.com/javase/specs/jls/se8/html/jls-4.html#jls-4.12.5)

Refer to Chapter 7.1, pages 525-541, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

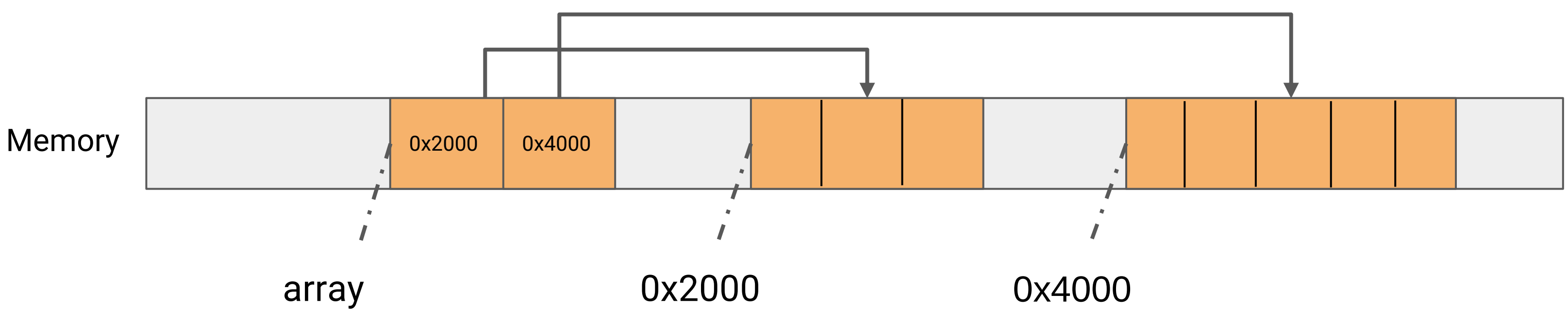
Multi-dimensional arrays

With the following code:

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

It will initialise the 2 elements within the array to **`null`**. This allows us to set each element to a separate array.

```java
array[0] = new int[3];
array[1] = new int[5];
```



Values 0x2000 and 0x4000 are used for demonstration purposes and are not the actual addresses.

Traversals

So how do we traverse a multidimensional array? Like so:

```java
int[][] array = new int[5][5];
// set elements
for(int i = 0; i < array.length; i++) {
    for(int j = 0; j < array[i].length; j++) {
        System.out.print(array[i][j]);
    }
    System.out.println();
}
```

So how do we traverse a multidimensional array? Like so:

```
int[][] array = new int[5][5];
// set elements
for(int i = 0; i < array.length; i++) {
    for(int j = 0; j < array[i].length; j++) {
        System.out.print(array[i][j]);
    }
    System.out.println();
}
```

We have some allocation of an array. That we will use

Traversals

So how do we traverse a multidimensional array? Like so:

```
int[][] array = new int[5][5];
// set elements
for(int i = 0; i < array.length; i++) {
    for(int j = 0; j < array[i].length; j++) {
        System.out.print(array[i][j]);
    }
    System.out.println();
}
```

We have some allocation of an array. That we will use

Some loop structure, in this case, a **for** loop

So how do we traverse a multidimensional array? Like so:

```
int[][] array = new int[5][5];
// set elements
for(int i = 0; i < array.length; i++) {
    for(int j = 0; j < array[i].length; j++) {
        System.out.print(array[i][j]);
    }
    System.out.println();
}
```

We have some allocation of an array. That we will use

Some loop structure, in this case, a **for** loop

Define the condition for the **first dimension**.

So how do we traverse a multidimensional array? Like so:

```java
int[][] array = new int[5][5];
// set elements
for(int i = 0; i < array.length; i++) {
    for(int j = 0; j < array[i].length; j++) {
        System.out.print(array[i][j]);
    }
    System.out.println();
}
```

We have some allocation of an array. That we will use

Some loop structure, in this case, a **for** loop

Define the condition for the **first dimension**.

For the second dimension, since the element at **i** is an array itself we are able to access the **length** property.

Traversals

So how do we traverse a multidimensional array? Like so:

```
int[][] array = new int[5][5];
// set elements
for(int i = 0; i < array.length; i++) {
    for(int j = 0; j < array[i].length; j++) {
        System.out.print(array[i][j]);
    }
    System.out.println();
}
```

We can output the element at **[i][j]**.

Traversals

So how do we traverse a multidimensional array? Like so:

```
int[][] array = new int[5][5];
// set elements
for(int i = 0; i < array.length; i++) {
    for(int j = 0; j < array[i].length; j++) {
        System.out.print(array[i][j]);
    }
    System.out.println();
}
```

Let’s assume we set all the elements in **array** from 0-24.

```
> java ArrayOutput
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
20 21 22 23 24
<program end>
```

Demonstration:
Array initialisation and manipulation

String is Special!

Strings

String is a **reference** type that aggregates **characters**. However like many other programming languages, Java treats Strings as immutable.

When initialising a string type, the JVM will **allocate memory** to contain the string.

```
String cat = "Meow";
```

When assigned, the string is allocated and binded to the variable **“cat”**.

What happens with the following expression?

```
String cat = "Meow";
cat += ", says the cat!"
```

Refer to Chapter 2.2, pages 116-124, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Strings

String is a **reference** type that aggregates **characters**. However like many other programming languages, Java treats Strings as immutable.

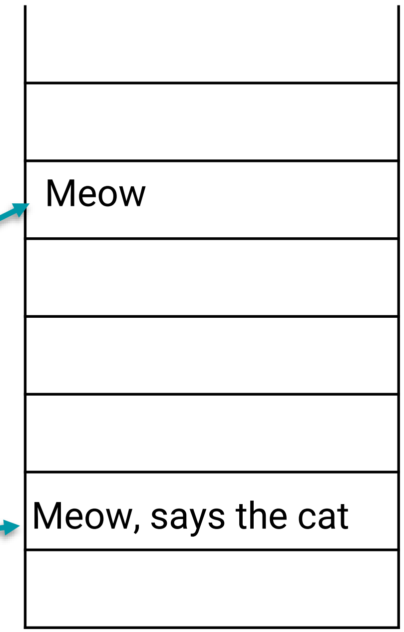
When initialising a string type, the JVM will **allocate memory** to contain the string.

A string is **immutable**. For concatenation to occur, the JVM will need to allocate a **new String** object to fit the contents of the **first string ("Meow")** and **second string**.

When assigned, the string is allocated and binded to the variable **"cat"**.

What happens with the following expression?

```java
String cat = "Meow";
cat += ", says the cat!"
```



Memory

Refer to Chapter 2.2, pages 116-124, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Strings

String is a **reference** type that aggregates **characters**. However like many other programming languages, Java treats Strings as immutable.

When initialising a string type, the JVM will **allocate memory** to contain the string.

```
String cat = "Meow";
```

Wh d and
“ca

+= with string is a **special case.** Since string concatenation/manipulation is a common operation. Java has provided an operator to easily concatenate strings.

A string is **immutable**. For concatenation to occur, the JVM will need to allocate a **new String** object to fit the contents of the **first string (“Meow”)** and **second string**.

What happens with the following expression?

```
String cat = "Meow";
cat += ", says the cat!"
```

Refer to Chapter 2.2, pages 116-124, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Strings

String is a **reference** type that aggregates **characters**. However like many other programming languages, Java treats Strings as immutable.

When initialising a string type, the JVM will **allocate memory** to contain the string.

```
String cat = "Meow";
```

When assigned, the string is allocated and **“cat”**.

The same operation will occur if we were to reassign with the **equivalent** expression.

What happens with the following expression?

```
String cat = "Meow";
cat = cat + ", says the cat!"
```

Refer to Chapter 2.2, pages 116-124, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Comparing strings

Any reference type variable holds onto the memory reference of the object.

So let’s analyse the following

```
String cat1 = "Meow";
String cat2 = "Meow";

System.out.println(cat1 == cat2);
```

Memory

We can see when run the following the output will be **true**.

Refer to Chapter 3.2, pages 189-191 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Comparing strings

Any reference type variable holds onto the memory reference of the object.

So let's analyse the following

```
String cat1 = "Meow";
String cat2 = "Meow";

System.out.println(cat1 == cat2);
```

We can see when run the above the output will be **true**.

Hang on though… Let's try something!

Refer to Chapter 3.2, pages 189-191 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

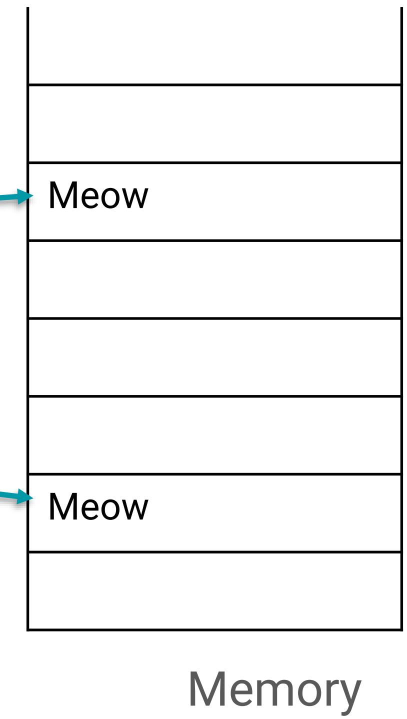
Comparing strings

Any reference type variable holds onto the memory reference of the object.

So let's analyse the following

```
String cat1 = "Meow";
String cat2 = new String("Meow");

System.out.println(cat1 == cat2);
```



We can see when run the above the output will be **false**.

Refer to Chapter 3.2, pages 189=191 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Comparing strings

Any reference type variable holds onto the memory reference of the object.

So let's analyse the following

```
String cat1 = "Meow"; -> 0x1000
String cat2 = new String("Meow"); -> 0x2000

System.out.println(cat1 == cat2);
```

We can see when run the above the output will be **false**.

Refer to Chapter 3.2, pages 189=191 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Comparing strings

Reference type does not **implicitly** have the ability to compare itself to another type (**besides reference**) without first defining a method. To compare them, we need to write a **method** to compare them.

Using **==** operator is testing the equivalence of the memory reference.

```
String cat1 = "Meow"; -> 0x1000
String cat2 = new String("Meow"); -> 0x2000

System.out.println(cat1 == cat2);
```

Since “**new String**” has returned a new allocation of a string, the **contents** will be the same but the allocation is different.

Refer to Chapter 3.2, pages 189=191 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Comparing strings

Reference type does not **implicitly** have the ability to compare itself to another type (**besides reference**) without first defining a method. To compare them, we need to write a **method** to compare them.

Using **==** operator is testing the equivalence of the memory reference.

```
String cat1 = "Meow"; -> 0x1000
String cat2 = new String("Meow"); -> 0x2000

System.out.println(cat1.equals(cat2));
```

Since “**new String**” has returned a n **contents** will be the same but the a

We use **.equals** as this method allows a string to compare its own contents with another. **We can define our own .equals method** for our **own types** to show define how equality is evaluated.

Refer to Chapter 3.2, pages 189=191 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Beware the string pool and equality

When writing our first program we have been able to define a **string literal**. However java employs an intelligent but often mistaken optimisation for strings.

```
String cat1 = "Meow"; -> 0x1000
String cat2 = "Meow"; -> 0x1000
```

If a string **literal** is specified, it will be added to a **string pool**. This explains the behaviour we have observed on slide **32.**

This allows the compiler to optimise for memory usage. The compiler will use the same allocation and provide same reference to a string variable of refers to the same **literal.**

Refer to Oracle Java Language Specification, (https://docs.oracle.com/javase/specs/jls/se8/html/jls-3.html#jls-3.10.5)

Demonstration:
String comparison and equality

Mutating string (Welcome StringBuilder)

Recreating a new string and deallocating the old one can be a costly for the java virtual machine (**jvm**).

We are able to mitigate this by using a class called **StringBuilder.** The **StringBuilder** class contains an internal array of characters and is **mutable** but does not have the same kind of affordances as **String**.

```
StringBuilder b = new StringBuilder();
```

The **StringBuilder** class allows us to assemble a string and resize the internal **char array** when the number of characters exceeds the capacity of the array.

Refer to Oracle Java SE8 API, (https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html)

Why do we have both?

Remember: Each time we use **+=** with the **String** type we are creating a new string. The **String** class is immutable and this can have great benefits for ensuring we have a read only or for simple concatenations.

However, when it comes to complicated string manipulation or excessive string manipulation, we will need to

```
StringBuilder b = new StringBuilder();
b.append("Hello");
b.append(" World");
```

Refer to Oracle Java SE8 API, (https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html)

Demonstration:
String vs StringBuilder

See you next time!