Arrays

- **Objects** in Java (but not in C/C++).

- An **array** is a container object that holds a *fixed number of values of a single type*.

- Specify length when creating.
Arrays

First index

Element (at index 8)

Array length is 10

Indices
Arrays

Example:

*int array of length 5.*

Declaration:

```
int[] array1;
```

Initialization:

```
array1 = new int[5];
```
Arrays

Length provided can be an *int variable*.

**Example:**

```java
int x = 6;
array1 = new int[x];
```
Arrays

Two ways to declare:

```java
int[] array1;     // convention
int array1[];    // discouraged!
```
Arrays

Declare and initialize with some values:

```
int[] array1 = {5, 100, 200, 45};
```

Length = no. of elements between the braces.
Arrays

- Property “length”.
- Very useful!
- Especially for traversing
Arrays

- Example: Printing out all the values of an array

```java
for (int i = 0; i < array1.length; i++) {
    System.out.println(array1[i]);
}
```
Arrays

- Another way of traversing:

  for-each loop:

    ```java
    for (int i : array1) {
        System.out.println(i);
    }
    
    For each element \textit{i} in \textit{array1} do...
    ```
Arrays

- What happens here?

```java
System.out.println(array1[array1.length]);
```
Arrays

- `ArrayIndexOutOfBoundsException` thrown
- Will later look at how to handle these exceptions.
- Be careful about boundaries!
public static void printArray(int[] array) {
    for (int i = 0; i < array.length; i++) {
        System.out.println(array[i]);
    }
}

int[] array1 = {5, 100, 200, 45};

printArray( array1 );
Passing Arguments to Methods

- Two ways:
  - Pass-by-Value:
    Used for primitive types (int/char/float/etc.)
    Changing value in the method does not change the original variable
  - Pass-by-reference:
    Used for objects (such as arrays). Changing in the method changes the original!
public void changeValue(float y) {
    y = 3.4;
}

float y = 5.4;
changeValue(y);
System.out.println(y);
Pass-by-reference

public void changeValue(float[] y) {
    y[y.length - 1] = 3.4;
}

float[] y = {5.4, 10.4, 1.2};
changeValue(y);
System.out.println(y[1]);

What is the output?
Pass-by-reference

\[ y = \{5.4, 10.4, 3.4\}; \]

After calling the method
```java
public void changeValue(float y) {
    y = 3.4;
}

float[] y = {5.4, 10.4, 1.2};
changeValue(y[0]);
System.out.println(y[0]);
```

What is the output?
Passing Arguments to Methods

- So be careful when passing objects/arrays to methods.
Copying arrays

- public static void arraycopy(Object src,
  int srcPos,
  Object dest,
  int destPos,
  int length)

- Provided in the **System** class

  System.arrayCopy(…)

  ```java
  void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)
  ```
Copying arrays

Example:

```java
char[] copyFrom = {'d', 'e', 'c', 'a', 'f', 'f', 'e', 'i', 'n', 'a', 't', 'e', 'd'};
char[] copyTo = new char[7];
System.arraycopy(copyFrom, 2, copyTo, 0, 7);
System.out.println(copyTo);
```

What is the output?
Copying arrays

Output: caffeine

```plaintext
char[] copyFrom = {'d', 'e', 'c', 'a', 'f', 'f', 'e', 'i', 'n', 'a', 't', 'e', 'd'};
```
Multi-dimensional Arrays

- Looked at 1-dimensional (1-D) arrays so far.

- Multi-dimensional arrays:
  - Array of arrays!

- Used for multi-dimensional data
Multi-dimensional Arrays

- Examples:
  - Storing all student info in a 2-D array
  - Matrices
  - Storing 3D data such as points/vertices \((x, y, z)\)
Multi-dimensional Arrays

- Declaration (2-D arrays):

  ```
  elementType[][] variableName;
  ```

- Example:

  ```
  float[][] matrix;
  ```
Multi-dimensional Arrays

- Initialization:

```java
matrix = new float[2][3];
```

- 2 rows, 3 columns
Multi-dimensional Arrays

- Declare and initialize with some values:

```c
float[][] matrix = { {2.3, 50.1, 5.5},
{9.8, 4.0, 11.4} };
```

Creates 2 arrays with length 3;
Multi-dimensional Arrays

- Each row can have a different length

```c
float[][] matrix = { {2.3, 50.1, 5.5},
                    {9.8, 4.0}   };
```

- First row = length 3
- Second row = length 2
Multi-dimensional Arrays

```java
float[][] matrix = new float[2][];
```

- Specifies size of the array - 2

- But not the size of the array in each row – do it individually

```java
matrix[0] = new float[3];
matrix[1] = new float[1];
```
Multi-dimensional Arrays

- Traversal:

```java
for(int row = 0; row < matrix.length; row++) {
    for (int column = 0; column < matrix[row].length; column++)
    {
        System.out.println(matrix[row][column]);
    }
}
```
The **String** class

- Array of characters in many languages (C/C++ for ex.)

- But: Strings are **objects** in Java

- Strings are widely used – so should know how to use them.
The **String** class

- **Creating Strings:**

  ```java
  String string1 = new String(“this is a string”);
  String string2 = “this is a string”;
  // slight difference in the 2nd one
  ```

- **String from a character array:**

  ```java
  char[] charArray = {‘s’, ‘t’, ‘r’, ‘i’, ‘n’, ‘g’};
  String string3 = new String(charArray);
  ```
The **String** class

- Strings are **immutable** – Not changeable!
The **String** class

- Strings are *immutable* – Not changeable!

- So what happens here?:

```java
String s = "Java";
s = "Another String";
```
The **String** class

- Initially:
  - `s` -> `Java`

- Now:
  - `s` -> `Java` -> `Another String`
The **String** class

- A new object "Another String" is created
- Reference `s` now points to it
- No reference to "Java", so garbage collection cleans it up
String comparisons

- **Interned** instances of the same string

- Example:

  ```java
  String s1 = "Java";
  String s2 = new String("Java");
  String s3 = "Java";
  ```
String comparisons

In Memory:

s1 → Java

s2 → Java
String comparisons

In Memory:

s1

s3

s2

Java

Java
String comparisons

- `s1 == s2` is?
String comparisons

- `s1 == s2` is false (different reference)
String comparisons

- $s_1 == s_2$ is false (different reference)

- $s_1 == s_3$ is ?
String comparisons

- \( s1 == s2 \) is false (different reference)
- \( s1 == s3 \) is true (same reference)
String comparisons

- Don’t use “==” for comparing objects!
  - Compares references
String comparisons

- **Don’t use “==“ for comparing objects!**
  - Compares references

- `string1.equals(string2)` for content comparison
  - `s1.equals(s2)` is true
String comparisons

- Don’t use “==” for comparing objects!
  - Compares references

- `string1.equals(string2)` for content comparison
  - `s1.equals(s2)` is true

- Define `equals()` method individually for every class – will look at it later
The **String** class

- Some useful String methods:
  - `.length()` – returns the length
  - `.charAt(int index)` – returns char at that index
  - `.split(String regex)` – Splits string according to a regular expression

- Will use String throughout
public static void main(String[] args) {
    ...
}

public static void main(String[] args) {
    ...
}

String[] args = array of arguments to the program
public static void main(String[] args) {
    ...
}

String[] args = array of arguments to the program

Executing a program:

java programName arg1 arg2 arg3 ...
Convert from String

- Input are **Strings**

- Need to convert to **int, double, etc...**

- Use *Wrapper* classes to convert

- Primitives have wrapper classes:
  - **Integer** for **int**, **Double** for **double**, etc.
Convert from String

- java programName 2 6.5 “testing java”

```java
int firstArgument = Integer.parseInt(args[0]);

double secArgument = Double.parseDouble(args[1]);

String thirdArgument = args[2]; // “testing java”
```
Next lecture...

- Java subclasses and inheritance