Arrays

Collections of data are managed using **data structures**. The most basic data structure is an **array**: an area of memory where the program can store a collection of data of the same type and operators that allow access to the items in the collection.

An array is declared with:

```
    type[] identifier;
```

Where `type` is either a primitive or a class. This creates a **reference** variable, i.e. a variable that references a memory location, but cannot be used directly to access memory.

Memory for an array is allocated with:

```
    type[] identifier = new type[size];
```

The operating system will allocate the memory necessary to store all of the items and the entries are initialized with appropriate values (e.g., 0 for integers).

**new** is used to allocate memory for arrays and objects.
Array entries are accessed by following the variable with an index enclosed in square brackets. In Java arrays are zero based, that is the index for the first entry is 0 and the index for the last entry is size – 1.

The array entry can be used to access the data and to store a value in the entry.

```java
int[] nums = new int[5];
nums[0] = 3; // Store 3 in 1st entry
nums[4] = 2; // Store 2 in last entry
nums[5] = 1; // Illegal - out of bounds
int entry = nums[3]; // Store 4th entry in variable
```

Arrays can be allocated and initialized in a single statement with a list of static values separated by commas enclosed in curly brackets:

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

This will create an integer array with 5 elements and initialize them to the values in the list.
The size of an array cannot be modified after the memory has been allocated. The size of an array can be obtained using `array.length`:

```java
int[] mylist = new int[12];
int len = myList.length; // 12 stored in len
```

Java provides a convenience construct, called a `for-each loop`, to loop over the entries in an array:

```java
// Assign value of each array entry to item
for ( int item : myList )
    System.out.println(item);

for ( int index = 0; index < myList.length; ++index )
    System.out.println(myList[index]);
```
When passing an individual array entry to a method, that entry is passed by value. If the value is modified in the method, it is not modified in the caller:

```java
void changeValueValue(int value) {
    value *= 10;
}

void caller() {
    int[] list = { 1, 2, 3};
    changeValueValue(list[1]);
    System.out.println(list[1]); // Will print 2
}
```

When passing the whole array to a method, the array is passed by reference. If values of the array is modified in the method, the value is changed in the caller:

```java
void changeValue(int[] value) {
    value[1] *= 10;
}

void caller() {
    int[] list = { 1, 2, 3};
    changeValue(list);
    System.out.println(list[1]); // Will print 20
}
```
Multidimensional Arrays

One dimensional arrays are used to represent *vectors*

\[
\text{int[]} \ \text{intVec} == [n_0 \ldots n_{i-1}]
\]

Two dimensional arrays are used to represent *matrices*

\[
\text{int[][]} \ \text{intMat} ==
\begin{bmatrix}
  \ldots & \cdots & \cdots \\
  [n_{0,0} & \ldots & n_{0,i-1}] \\
  \ldots & \cdots & \cdots \\
  [n_{j-1,0} & \ldots & n_{j-1,i-1}]
\end{bmatrix}
\]

Three dimensional arrays can be used to represent an array of matrices. Java allows the creation of *n*-dimensional arrays.
Declaration of 2-dimensional arrays:

```java
int[][] intMat;
```

Static allocation of memory:

```java
int[][] intMat = new int[4][5];
```

Dynamic allocation of memory:

```java
int[][] intMat = new int[4][];
for (int row = 0; row < 4; ++row )
    intMat[row] = new int[5];
```

Initialization (regular):

```java
int[][] intMat = {{4, 5, 2, 3, 7},
                 {12, 1, 4, 5, 7},
                 {5, 2, 3, 11, 10},
                 {32, 4, 5, 8, 1}};
```

Initialization (ragged):

```java
int[][] intMat = {{4, 5, 2, 3, 7},
                 {12, 1, 4},
                 {5, 2, 3, 11, 10, 12},
                 {32, 4}};
```
The **length** operator

Returns the size of the current dimension:

```java
int[] intVec = new int[6];
len = intVec.length; // returns 6

int[][] intMat = {{1, 2, 3, 4, 5},
                 {1, 2, 3, 4},
                 {1, 2, 3, 4, 5, 6}};
len = intMat.length; // returns 3
len = intMat[0].length; // returns 5
len = intMat[1].length; // returns 4
len = intMat[2].length; // returns 6
```