CS102

Introduction to data structures, algorithms, and object-oriented programming

DAY 6
A data structure in which the items are arranged as a numbered sequence, so that each individual item can be referred to by its position number.

All the items in an array must be of the same type, and the numbering always starts at zero. An array is a list of variables, each accessible by the array name and position number of the variable.

An array is, technically, an object, so the process of creating one requires an instantiation with the keyword new.

The size of an array is fixed. It must be given when instantiating the array.
An array can be of any type and must first be declared:

```java
String[] name; // declaration of String array
int[] age;    // declaration of int array
boolean[] leftHanded; // declaration of boolean array
```

Then the array must be instantiated:

```java
name = new String[10]; // 10 memory locations for Strings
// each has value null to start
age = new int[5];      // 5 memory locations for int values
leftHanded = new boolean[100]; // 100 boolean locations
```

After instantiation, the specified number of locations will be created in memory and reserved for the given type.
To put many values into an array at the same time, you can use the array name and give a list of elements in braces:

```java
String[] names = { "Penny", "Joe", "Alfred", "Will", "Nancy"};
```

If you use this technique to create an array, you must put the entire statement on one line.

This array contains 5 elements, numbered from 0 to 4. To access one of these positions, use the variable names

- `names[0].length()` => 5
- `names[1].charAt(2)` => 'e'
- `names[2].indexOf('d')` => 5
- `toUpperCase(names[3])` => "WILL"
- `names[4]` => "Nancy"
To put values into the array, you use the array name and position number to store a value at that position:

```java
name[5] = "Penny";
```

The length of an array is stored with the array as a field name accessible as, for example:

```java
name.length  // notice these are
age.length   // not method calls
```

Having access to the length of every array allows them to be easily used within a for loop to go through each element:

```java
// this for loop prints all elements in age array
for (i = 0; i < age.length; i++) {
    System.out.println( age[i] );
} // end for
```
The statement
int[] A = new int[5];
creates an array that
holds 5 elements of
type int. A is the
name for the whole
array.

A contains 5
elements, which are
referred to as A[0],
A[1], A[2], A[3], and
A[4]. Each is a
variable of type int.

The array also
contains A.length,
whose value cannot
be changed.
1-dimensional arrays

Declaration and instantiation example:

    int[] ages = new int[4];

The line above would create a matrix with 4 values, initially all set to 0. The code below would give each array position a value read from the user.

Scanner scn = new java.util.Scanner(System.in);

for (int i = 0; i < ages.length; i++) {
    ages[i] = scn.nextInt();
}
2-dimensional arrays

Declaration and instantiation example:

```java
int[][] matrix = new int[10][5];
```

The line above would create a matrix with 10 rows and 5 columns, initially all set to 0. The code below would give each array position a value.

```java
int[][] matrix = new int[10][5];
for (int row = 0; row < matrix.length; row++) {
    for (int col = 0; col < matrix[0].length; col++) {
        matrix[row][col] = row * col;
    }
}
```
2-dimensional arrays

Printing example:

The nested for loops below will print the contents of the matrix array created on the last slide.

```java
for (int row = 0; row < matrix.length; row++) {
    for (int col = 0; col < matrix[0].length; col++) {
        System.out.printf("%7d", matrix[row][col]);
    }
    System.out.println();
}
```
Arrays and memory

Each element of an array is stored in a consecutive memory location, giving fast access to any element just by adding an offset to a given starting point.

Arrays are called *random access* data structures because values can be looked up quickly. The downside to arrays is that they have a fixed size and a limited content type.
Java memory

Java divides main memory into two main regions:

- Stack and Heap

The Java run-time stack holds all the local variables and parameters of a method. When a method calls another method, a new frame is added to the top of the stack, along with the address to return to when the method ends.

Primitive values are written directly into the stack, indexed by the name given when they are declared.

Reference values (objects) and arrays are also given named positions on the Stack, but in this case, the contents of the positions are memory locations on the Heap.
Java Memory Organization

• Static Memory:
  – Stores class (static) variables.

• Dynamic (Heap) Memory:
  – Stores objects.

• Run Time Stack:
  – Stores local variables defined in method bodies.
    Stores method calls in separate frames

• Program Counter:
  – Keeps track of statement being executed.
Run Time Stack

• Each element of the stack is a “Method Frame”.

• A frame keeps track of the value of:
  – a method’s parameters.
  – local variables defined in method.

• Stack is LIFO data structure (last-in-first-out)
  – When a method is called: Push stack frame onto top.
  – When returning from method: Pop stack frame off top.
Passing Parameters to Procedures

- **Pass by Value:**
  - Used for passing Java’s primitive types, e.g., boolean, char, short, int, long, float, double.
  - The data is copied into the memory location named when value is declared.
  - Changes to parameters inside a method have no effect on the arguments passed in to the method.

- **Pass by Reference:**
  - Used for Java’s reference types, e.g., arrays, and instances of classes (objects).
  - A reference (memory location) to the data is stored in the method frame for an object type.
  - Changes to components of the formal parameters (e.g., array elements and instance variables) affect both the parameters and the arguments.
Communicating with a Method

• Getting information into a method:
  – Explicit input parameters.
  – Class and instance variables of the object on which the method is invoked.

• Getting information out of a method:
  – Explicit return value.
  – Changes to "state" of object: class and instance variables:
    • Of input parameters to the method.
    • Of the object on which the method is invoked.
Debugging Logical Errors

The point of testing is to find bugs -- semantic errors that show up as incorrect behavior rather than as compilation errors.

Most programming environments come with a debugger, which is a program that can help you find bugs by giving the value of different variables at a particular line in the code.

A more traditional approach to debugging is to insert debugging statements into your program. These are print statements that print out information about the state of the program.
import junit.framework.TestCase;
import static org.junit.Assert.*;
import org.junit.Test;

public class TryTTT extends TestCase {

    public void testCheckForNoWin() {
        char[][] ch = {{'o','x','x'},{'x','o','o'},{'o','x','x'}};
        boolean actual = TicTacTow.checkForWin('x',ch);
        boolean expected = false;
        //Using assertEquals to check-expect
        assertEquals(expected, actual);
    }
}
random numbers

The random method is a static member of the Math class. The call `Math.random()` produces a double between 0.0 and 1.0, inclusive. To use the `Math.random()` function to get an `int` between 1 and 10, for example, you would use the following call:

```
int rNum = (int)(Math.random() * 10) + 1
```

The `(int)` operator truncates the double returned by `Math.random()` to produce an `int`. This type of operator is called a “cast”. The `+ 1` is there to avoid producing 0.
Type Conversion

• Changing a datum from one type into another.

• Explicit Conversion: Programmer uses a `cast` operation to perform the type conversion.

• Implicit Conversion: Compiler automatically inserts code to perform the type conversion.
Implicit Conversion from Narrow Types to Wider Types

As a convenience, rather than requiring the programmer to indicate a type conversion explicitly, the Java programming language performs an implicit conversion from the type of the expression to a type acceptable to its surrounding context.