



Programming and Software Development
COMP90041

Lecture 7

Arrays



- Classes allow design of a type holding any combination of values of any types
- But what if you want to store an arbitrary number of values of the same type?
- *E.g.*, the arbitrary number of command line arguments passed to the program
- These are passed as an array
- An array can be used like an arbitrary sequence of separate variables of the same type, but also as a single value



- Declare a single variable to hold a whole array
- Form: *baseType [] varName*
- Where *baseType* is the type of every array element
- E.g.: `String[] args;`
- Use the same syntax to declare:
 - ▶ local variables
 - ▶ instance or class variables
 - ▶ method parameters
- Similar syntax to declare method that returns an array:
baseType [] methodName (type name, ...)
(can be `public` or `private`, `static` or not)



- As for class types, declaring a variable does not create an array
- Simplest way to get an array is initialise to a constant
- Form: *baseType [] varName = {value, ...};*
- E.g.:

```
public static final int[] DAYS_IN_MONTH
    = {31,28,31,30,31,30,31,31,30,31,30,31};
```

- (Why make this **static final**?)
- But **{...}** is not an expression; this syntax can only be used in variable initialisations



- To create a fresh array, use special `new` syntax
- Form: `new type [size]`

```
double[] data = new double[10];
```

- Note array type never includes size, but `new` expression always does
- Size can be specified as any integer-valued expression ≥ 0 , so you can calculate the size
- You cannot specify initial values for array elements in `new` expression
 - ▶ Number elements are initialised to 0 or 0.0, `booleans` to `false`, `chars` to '`\0`', and objects to `null`
 - ▶ You must initialize object array elements yourself!



- Access array elements using *array [index]*
- *E.g.: args[2]*
- The value inside the brackets is called an array index
- Array indexes range from 0 to array size - 1
- Use the same syntax to assign to an array element
- *E.g.: data[2] = data[1] * 10;*



Array Index Out of Bounds

- Size of an array is fixed at the time it is created
- Arrays cannot be expanded (or shrunk)
- Accessing or assigning an array at an index < 0 or \geq the size of the array causes an error:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
        at Tst.main(Tst.java:3)
```



- An array index can be specified as an expression
- Very common to index an array with a variable
- *E.g.:*

```
public static final int[] daysInMonth
    = {31,28,31,30,31,30,31,31,30,31,30,31};
public static void main(String[] args) {
    int month = Integer.parseInt(args[0]);
    int day = Integer.parseInt(args[1]);
    int yearday = 0;
    for (int i = 0; i < month - 1; ++i) {
        yearday += daysInMonth[i];
    }
    System.out.println(yearday + day);
}
```



- The size of an array is *array.length*
- *E.g.:*

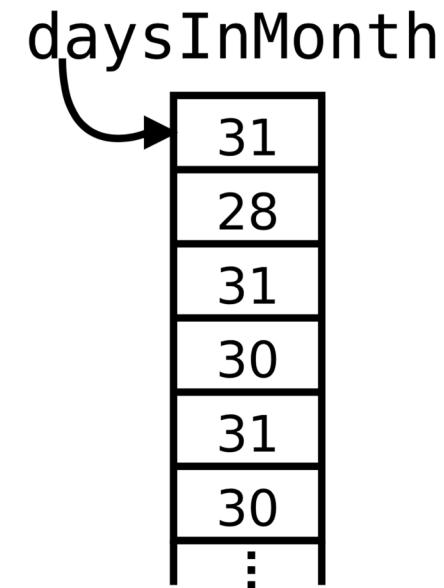
```
public class ArgCount {  
    public static void main(String[] args) {  
        System.out.printf("You entered %d arguments%n",  
            args.length);  
    }  
}
```

- Very useful for iterating over a whole array:

```
for (int i = 0 ; i < a.length ; ++i) ... a[i] ...
```



- Arrays are reference types
- Similar to objects (but not quite the same)
- Stored in memory as a sequence of elements: efficient to index
- Unlike your own classes, array types cannot be made immutable
- Beware **privacy leaks**: copy arrays if necessary





- You cannot write a copy constructor for arrays
- but you can write a **static** method to copy an array, e.g.:

```
public static String[] copyStringArray(String[] orig){  
    String[] copy = new String[orig.length];  
    for (int i = 0; i<orig.length; ++i) {  
        copy[i] = orig[i];  
    }  
    return copy;  
}
```

- But you need to write a separate copy method for each base type of array you need to copy



- Can declare array with a class as base type
- *E.g., String[] args*
- creating an array of objects does not create any objects, just an array full of nulls
- Be sure to initialise the array elements to new objects if necessary

```
Person[] people = new Person[args.length];
for (int i = 0; i < people.length ; ++i) {
    people[i] = new Person(0, args[i]);
}
```

(creates an array of **Persons** with age 0 and name taken from command line arguments)



- You cannot assign to an array's `length` “member” to resize it

```
arr.length = arr.length + 1; // compilation error!
```

- But it's easy to create a new, bigger array and copy
- To grow `arr` and add 42 at the end:

```
int[] tmp = new int[arr.length+1];
for (int i = 0; i < arr.length; ++i) tmp[i] = arr[i];
tmp[arr.length] = 42;
arr = tmp;
```



- Copying an array to add one element to the end is expensive and inconvenient
- One common trick is the partially filled array: an array where the first elements are the intended elements, and the rest are meaningless
- A variable holds the number of meaningful elements
- Make effective array bigger by just incrementing the number of meaningful elements
- But when that reaches the size of the actual array, must copy to new, bigger array



```
public class ExpandingIntArray {  
    public static final int INITIAL_SIZE = 10;  
    public static final int EXPANSION = 2;  
    private int[] data = new int[INITIAL_SIZE];  
    private int length = 0;  
    public void append(int val) {  
        if (length >= data.length) {  
            int[] tmp = new int[EXPANSION*data.length];  
            for (int i = 0; i<data.length; ++i) {  
                tmp[i] = data[i];  
            }  
            data = tmp;  
        }  
        data[length] = val;  
        length++;  
    }  
}
```

3



- As of Java 5, there is a special form of `for` loop:
the foreach loop
- Form:
`for(elementType name : array) body`
- Executes `body` for each element of `array`, with variable `name` bound to the element
- No need to worry about array indices
- E.g.:*

```
int daysInYear = 0;  
for (int monthLength : DAYS_IN_MONTH) {  
    daysInYear += monthLength;  
}
```



- Unlike some languages, Java does not directly support 2 or more dimensional arrays
- However, you can make arrays of arrays, e.g.:

```
int[][] tTable = new int[10][];
for (int i = 0; i<10; ++i) tTable[i] = new int[10];
for (int i = 0; i < tTable.length; ++i) {
    for (int j = 0; j < tTable[i].length; ++j) {
        tTable[i][j] = (i+1)*(j+1);
    }
}
for (int[] row : tTable) {
    for (int n : row) System.out.printf("%4d", n);
    System.out.println();
}
```



- An enumerated type is a type whose values are all specific constants
- New feature in Java 5
- (Simplest) form:
- `enum typeName {value1, value2, ...};`
- Place this at top level inside a class, or in a file by itself, named `typeName.java`
- Convention: values are spelled all uppercase, words separated by underscores
- *E.g.:*

```
enum DayOfWeek {SUNDAY, MONDAY, TUESDAY,  
    WEDNESDAY, THURSDAY, FRIDAY, SATURDAY  
};
```



- With `enum` declaration, `typeName` is a valid type
- Each `value` is a constant referred to as `typeName.value`
- Use `==` and `!=` to compare `enum` values for equality
- E.g.:*

```
public static boolean isWeekend(DayOfWeek day) {  
    return day == DayOfWeek.SATURDAY  
        || day == DayOfWeek.SUNDAY;  
}
```



- Java implements `enums` as classes
- They automatically have useful public methods:
 - ▶ `String toString()` returns value as a `String`
 - ▶ `static type valueOf(String)` returns enum value of string (Note `static`; error if no exact match)
 - ▶ `int ordinal()` returns 0 for the first value, 1 for the second, etc..
 - ▶ `static type [] values()` Returns an array of all the enum values for the type, in order
- `values` is very useful, as it allows you to loop over all values of an enum type



- Every class type t , including enum types, defines:
 - ▶ boolean `equals(t)` returns true iff the object is “the same” as the argument t
 - ▶ int `compareTo(t)` returns a negative number if the object is “less than” the argument t , zero if the same, and positive if “greater”
- For classes, the programmer must define these
- It’s up to you to decide what “the same”, “less” and “greater” mean for your class
- For enum types, they are defined automatically; “less than” means appearing earlier in the declaration
- Usually must use `a.equals(b)`, not `a==b`, to test objects for equality; enum types are an exception
- Not defined for arrays, only classes and enums



```
public class DayOfWeekTest {  
    enum DayOfWeek {SUNDAY, MONDAY, TUESDAY,  
        WEDNESDAY, THURSDAY, FRIDAY, SATURDAY  
    };  
  
    public static void main(String[] args) {  
        DayOfWeek d = DayOfWeek.valueOf(args[0]);  
        System.out.println(d + ": " +  
            (d.ordinal()+1) + "th day of week");  
        int next = d.ordinal() + 1;  
        next %= DayOfWeek.values().length;  
        d = DayOfWeek.values()[next];  
        System.out.println("Next day is " + d);  
    }  
}
```



- An array is a collection of any number of values of the same type (can be primitive or object type)
- `baseType []` is type of array of `baseType` elements
- `new baseType [size]` returns a fresh array of `size` elements of `baseType`
- Access/set elements using `array [index]` syntax;
`array [0]` is first element
- `array .length` is number of elements in array
- `for(type name : array)` iterates over `array`
- `enum type {value1, ...};` declares an enumerated type



THE UNIVERSITY OF

MELBOURNE