Introduction to Java

Handout-2a
Runtime internals – stack & heap

• Stack: a run-time data structure. Used to do automatic memory management in block-structured languages
  – Lifetime of storage allocated on stack is tied to the scope in which it was allocated

• Heap: all Java objects are allocated on the heap
  – Lifetime of storage allocated on heap is independent of the scope in which it was allocated
Arrays (i)

- In Java, arrays are objects
- Java arrays are allocated dynamically and keep track of their length

Ex:
```java
int day[]; // day is variable that points to an array of integers
```

- Since day is a reference, it’s good for any size on int array
Arrays (ii)

• Indexing starts from zero
• Array indexes are checked at run-time
  – If subscript attempts to access element outside the bounds of array, the program will raise exception and cease execution
Arrays (iii)

• Arrays *are* like objects
  – The language specification says so
  – Array types are reference types, just like object types
  – Arrays are allocated with “new” operator
  – Arrays are always allocated on heap not stack
  – The parent class of all arrays is Object; you can call any of the methods of Object on an array
Arrays (iv)

• In some ways arrays are not like objects
  – Can’t make an array be the child of some class other than Object
  – Arrays have a different syntax from other object classes
  – Can’t define your own methods for arrays
Ex:

public static void main(String args[]) {
    int i=0, n, k;

    n = args.length; // number of arguments (the total number of Strings in args[])
    k = args[i].length(); // length of string at index i in args[]
}

Arrays (vi)

- Declaring an array only creates a reference
  ```
  int days[]; // days can hold a reference to
  // to any size array of int
  ```

- You must make the reference point to an
  array before you can use it
  ```
  days = new int[7];
  ```

- Once the array object has been created it
  cannot change in size
Arrays (vii)

- Initialization; same as objects
  - Fields that are primitive types are created and initialized to zero
  - Fields that are reference type are initialized to null (don’t point to anything yet)

Ex:
cherry = new int[256]; // creates 256 integers
cherry[7] = 123;

Fruit cherry = new Fruit[256]; // array of 256 references
cherry[7].grams = 4; // Run-time error. cherry[7] is a null reference
Arrays (viii)

• You can initialize an array in its declaration using an array initializer

```java
byte b[] = { 0, 1, 2, 3, 4, 2 }; // array of 6 bytes
String weekendDays[] = {“Sat”, “Sun”, };
```

• Can’t use an array initializer anywhere out side a declaration

```java
weekendDays = {“Sat”, “Sun”, }; // Error
weekendDays = new String[] {“Sat”, “Sun”, }; // Ok
```
Arrays (ix)

• The language specification says there are no *multi-dimensional* arrays in Java.

• Java only has *arrays of arrays* and it calls them arrays of arrays.

Ex:

```java
Fruit plums[][]; // array of arrays whose elements
                 // are Fruit objects
plums[i] = new Fruit[7];  // Ok
plums[i][j] = new Fruit(); // individual Fruit
```
Arrays (x)

• Bottom-level arrays do not have to be all of the same size

Ex:

```java
int myTable[][] = new int[][] {
    new int[] {0},
    new int[] {0,1},
    new int[] {0,1,2},
};
```
Arrays (xi)

- If you don’t instantiate all dimensions at once, then you have to instantiate the most significant dimensions first

Ex:
```java
int apple[][] = new int[5][];    // Ok
int apple[][] = new int[5][6];  // Ok
int apple[][] = new int[5][6];  // Ok
int apple[][] = new int[][[3];   // Error
```
Operators (i)

• The order of operand evaluation is well defined in Java
  – Expressions are evaluated left-to-right
  – The left operand is evaluated before the right operand of a binary expression; true even for the assignment operator
  – In an array reference the expression before the [] is fully evaluated before any part of the index is evaluated
Operators (ii)

• A method call for an object has the general form
  \textit{objectInstance.methodName}(arguments)
  
  – The objectInstance is fully evaluated before the
    methodName and arguments
  
  – Arguments are evaluated one by one, from left to the
    right

• In an allocation expression for an array of several
  dimensions, the dimension expressions are
  evaluated one by one from left to right
Associativity

• There are three factors that influence the ultimate value of an expression:
  – *Precedence* indicates that some operations bind more tightly than others
  – *Associativity* is the tie breaker for deciding the binding when we have several operators of equal precedence strung together
  – *Order of evaluation* tells the sequence, for each operator, in which the operands are evaluated