Java Arrays, Part 2

- Multiple-Dimensional Arrays
- Type Parameters
- Array Lists
- Array Algorithms

Two-Dimensional Arrays

- So far, every array we have seen has a single index
- A single index works well for many applications, but not for representing 2-Dimensional data
- Instead, we can declare that an array has multiple dimensions:

```
enum ChessPieces { Empty, Pawn, Rook, Knight, Bishop, Queen, King }
...
final int ROWS = 8;
final int COLUMNS = 8;
ChessPieces[][] chessboard = new ChessPieces [ROWS][COLUMNS];
```

Matrices

- A mathematical matrix can be represented as a 2D array:

  double[][] matrix = new double[15][17];
- You can then create a method for matrix multiplication:

```
public static double[][]
 matrixMultiply(double[][] m1, double[][] m2){
  if (m1[0].length != m2.length) {
    ...  // different sizes, cannot multiply
  }
  double[][] result =
     new double[m1.length][m2[0].length];
  result[0][0] = ...
```

- The number of rows of a matrix m is m.length. The number of columns is m[0].length

Neighboring Elements

- When a two-dimensional array is representing properties of a two-dimensional object (e.g. a picture), it is sometimes useful to be able to compute the indices of neighboring elements
- Given the origin is at 0,0 in the upper left, for the element at i, j
  - the element above it is at i-1, j
  - the element to the left is at i, j-1
  - the element below it is at i+1, j
  - the element to the right is at i, j+1
- An exercise for you: give the positions of the elements at the upper left corner, upper right corner, lower left corner, and lower right corner

Multi-Dimensional Arrays

- Java supports arrays with any number of dimensions:

```
double[][][] cube =
  new double[10][10][10];
cube[9][9][9] = 3.1415;
double[][][][] spaceAndTime = ...
```

- These work the same as two-dimensional arrays

Non-Rectangular Arrays

- A two-dimensional array in Java is really an array of arrays
- The sub-arrays may all have different sizes:

```
String[][] a = new String[5][];
// lengths will be 1, 4, 7, 3, 6
for (int i = 0; i < 5; i++) {
  a[i] = new String[(i * 3) % 7 + 1];
}
```

- Such arrays are occasionally useful
  - but are not common.

Type Parameters

- When describing a sample implementation of the Arrays.copyOf method, we used *someType* to represent the type of the array that was being copied
- This is actually useful in real programs:
  - when a type T, such as an array, stores elements of another type U, we can say that T is parametrized over U
- The type equivalent of a variable is a **type parameter**
- Arrays are built-in to Java and the type of the array element is part of the Java syntax, but when we create other **collection types** we will parametrize them

Type Parameters: Example

- ArrayList is a parametrized collection type (java.util.ArrayList)
- Type parameters are written in angle brackets. Here we declare a variable x to be an ArrayList containing strings:

```
ArrayList<String> x =
  new ArrayList<String>();
```

- in creating this new object, we need both `new` and `()`
- Java is clever enough to figure out the second type parameter, so it can be omitted:

```
ArrayList<String> x = new ArrayList<>();
```

Type Parameters: Objects Only

- A type parameters can only be an Object type, we cannot use int, double, char, boolean as type parameter
- Because of this, an Object type has been defined in Java for each of the basic types: Character, Boolean, Byte, Short, Integer, Long, Float, Double
- These object types can be used as type parameters:

```
ArrayList<Double> x = new ArrayList<>();
```

Using the Object equivalents of the basic types

- Because these object types are built-in to Java, Java can automatically convert between the basic types and their equivalent object types:

```
Boolean t = true;
if (t) { ...
```

- As you know, object variables are references to the memory where the object value is actually stored
- The process of putting a basic type into an object is called **boxing**
- Java provides auto-boxing and auto-unboxing, so programmers in general don't have to think about the distinction between, e.g. int and Integer
  - except that only Integer can be used as a type parameter!!!

Array Lists

- Arrays are very convenient, and use an intuitive syntax supported by Java
- However, the length is fixed
  - if we want to change the length, we have to copy the array
- ArrayList is a collection type that is designed to be similar to arrays, but:
  - grows on demand
  - has additional methods that provide convenient functionality for programmers
- ArrayList access does not have the convenient Java syntax that arrays have, and is slightly slower, so programmers often still choose to use arrays even though ArrayLists offer more functionality
- Just as in arrays and strings, the first index in an ArrayList is 0
- Just as with arrays and strings, ArrayLists can be used as parameter types and method return types:

```
public static ArrayList<String> convert(ArrayList<Integer> a) { ...
```

Array List methods: add

- ArrayList.add(value) adds value to the end of the array list, extending the array list
- ArrayList.add(index, value) adds the value at the given index, moving out of the way all the elements with that index and higher
- so if an array list x has 1, 7, 33, 42, the call x.add(2, 25) changes x to have 1, 7, 25, 33, 42
- Whereas for the same array list x with 1, 7, 33, 42, x.add(999) changes x to have 1, 7, 33, 42, 999

Array Lists: other methods

- all examples are with x having 1, 2, 3
- ArrayList.size() returns 3, the number of elements
- ArrayList.get(index) returns the value at that index: x.get(2) returns 3
- ArrayList.set(index, value) is like the assignment of an array element: after x.set(0, 55), x has 55, 2, 3
- ArrayList.remove(index) removes the value at the given index, moving the other elements to fill the gap
  - after x.remove(1), x has 1, 3 and x.size() returns 2
- Copying array lists is accomplished by creating a new array list, giving the old one as parameter:

```
ArrayList<String> myCopy = new ArrayList<String>(oldCopy);
```

Array Lists: enhanced for

- The enhanced for loop works with ArrayLists, and in general, with all Java collection types

```
ArrayList<Double> x = new ArrayList<>();
...
for (Double e: x) {
  total += e;
}
```

Comparison of Arrays, Strings, ArrayLists

- array.length, String.length(), ArrayList.size()
- 0 is always the first index
- a[n], String.charAt(n), ArrayList.get(n)
- a[n] = value; ArrayList.set(n, value);
- variable size: arrays need an additional variable, ArrayLists do it naturally
- adding and removing elements: only in ArrayList

Array Algorithms

- We have already seen a few array algorithms
  - printing elements with separators (demonstrated in class)
- Most of these algorithms work equally well with arrays and array lists
  - in general, we will refer to arrays unless specifically talking about ArrayList
- Refer to the book (section 6.3) for a more comprehensive list; only a few presented here

Array Algorithms: Linear Search

- There are many cases when we want to look through all of an array to find something
- If you imagine the elements of the array stretched out in a line, and starting from element 0 to the last element, this is a **linear search**
- There are many forms of linear search, but imagine we just want to find a specific value:

```
public static boolean contains(int[] a, int v) {
  for (int x: a) {
    if (v == x) {
      return true;
    }
  }
  return false;
}
```

-

Array Algorithms: Inserting or Removing Elements

- If we have enough room in the array, and want to move elements out of the way so we can insert a new value, we can do so. Note that we have to move elements from the end of the array:

```
// the first inUse elements of a are in use. insert v at insertPos
// this code does not handle the case where the array needs to be resized
public static int insert(String[] a, int inUse, int insertPos, String v) {
  int copy = inUse;
  while (copy >= insertPos) {  // move higher elements out of the way
    a[copy] = a[copy - 1];
    copy--;
  }
  a[insertPos] = v;            // now insert the element
  return inUse+1;              // return the new size
}
```

- remove is the same, but we must copy elements from low to high indices
  - exercise: take a minute to write the code for remove
- the ArrayList add and remove methods do all this
  - and also resizing the array, if add needs more room

Array Algorithms: Swapping Elements

- If you want to swap two elements of an array, you need a temporary variable:

```
int[] a = ....
int x = ...
int y = ...
// now swap a[x] and a[y]
int temp = a[x];
a[x] = a[y];
a[y] = temp;
```

- The temporary variable is needed because we have to save the value of a[x] before we can store the value of a[y] into it

Array Algorithms: Sorting

- Sorting an array means ordering its elements from low to hight
- Java already has a method

```
Arrays.sort(a);
```

- We can also sort a partly-filled array:

```
Arrays.sort(a, 0, currentSize);
```

- `ArrayList.sort(null)` is also provided
  - the `null` parameter is a sentinel to request sorting according to the type's natural order
  - a different parameter may specify a different sort order

Summary

- Multi-dimensional arrays are arrays of arrays
  - the sub-arrays may have different lengths, but usually all have the same length
- collection types are parametrized on specific Object types
  - each basic type has a corresponding Object type
  - and Java handles the conversion automatically
- Array lists have all the features of arrays but also automatically extend and shrink to fit the contents
- Arrays, loops, and methods from the Java standard library let us write many interesting and useful programs!