Handling array size limitations

- Issue: array size is fixed after construction
  - Don't always know what size to allocate at start
- Solutions (besides `class ArrayList` – coming soon)
  - Allocate "way more than enough"
    - *Absolutely limits* the size of the problem – not a good idea
  - Create new, larger array, and copy values

```
if (dataSize >= data.length) {
    int[] newData = new int[2 * data.length];
    ... // here: deep copy up to (data.length – 1)
    data = newData; // copy reference (discard old array)
}
```

Arrays of object references

- Arrays of objects require 3 steps to use:

```
Rectangle[] boxes; // 1. declare array of references
boxes = new Rectangle[3]; // 2. instantiate array
for (int i=0; i<boxes.length; i++)
    // 3. instantiate each object in the array:
    boxes[i] = new Rectangle(5+i, 5+i, 5, 5);
```

- Two ways to copy (like any object that has references to other objects):
  - Shallow copy – just copies array references
  - Deep copy – makes new copies of all objects

Arrays of arrays

- e.g., `int a[][] = new int[10][4];`
  - Like a "table" with 10 rows and 4 columns
  - `a.length` is 10
  - Each `a[i].length` is 4, for all `i`
  - Component array sizes can vary
    - `a[2] = new int[6];` // now 3rd row has 6
- Typically use *nested for loops* to process
  - See TicTacToe.java (p. 307)

`java.util.ArrayList`

`ArrayList<T> a = new ArrayList<T>();`

- `T` is an object type – may not be primitive
- A generic class (since Java 5) – so "type safe"
- Use methods to `add`, `insert`, `remove`, `set`, `get` …
  - Cannot use = or `[]` notation like arrays
- Use "wrapper" classes for primitive data types
  - `Btye, Short, Integer, Long, Float, Double, Character, Boolean`
  - Autoboxing and auto-unboxing simplifies it though

```
ArrayList<Double> list = new ArrayList<Double>();
list.add( 0.74 ); // actually adds new Double(17.64)
double d = list.get(0);
    // actually executes list.get(0).doubleValue();
```

How to use `ArrayList`s

- Declare/create ArrayList (no need to size it):
  `ArrayList a = new ArrayList();`
  - Or – with Java 5 – can specify the type
    `ArrayList<T> a = new ArrayList<T>();`
    // where `T` is an object type – not a primitive data type
- Add objects to end, or set and get specific objects

```
ArrayList<Rectangle> a = new ArrayList<Rectangle>();
a.add(new Rectangle(5,5,5,5));
Rectangle r = a.get(0); // gets first
a.set(0, new Rectangle(0,0,10,10)); // replaces first
```

- Simple insert and remove too

```
a.insert(i, new Rectangle(1,1,1,1)); // inserts in position i
a.remove(i); // removes element in position i
```

Sample Quiz

1. (10 points) Let `x[]` be an array of `double` that is already initialized. Create an `ArrayList<Double>` object, and copy each `x` value to this list in *reverse order* (add the last element first, …, and the first element last) .
2. (10 points) Let `y[][]` be an array of `double` arrays that is already initialized. Translate the following nested enhanced for loops to nested while loops:

```
for (double[] row : y)
    for (double value : row)
        System.out.println(value);
```

1st Quiz – 20 homework points

1. (8 points) Let `x[]` be a `double` array that is already initialized. Translate the following enhanced `for` loop to a `while` loop:

```
for (double d : x)
    System.out.println(d);
```

2. (12 points) Let `y[][]` be an array of `double` arrays that is already initialized. Declare and create an `ArrayList<Double>` named `list`, and add copies of every value in `y[][]` to `list` (the order does not matter).

More `java.util` collections

- `List` – actually an interface
  - Defines a set of common methods like `add`, `size`, `iterator`
    - Shared by `ArrayList`, `LinkedList`, and others
  - Note: `Collections` methods to manipulate `List` objects:

  `Collections.shuffle(list);` // randomly shuffles the list
  `Collections.sort(list);` // assuming items are `Comparable`
- `Stack` – a last-in first-out (LIFO) data structure

```
Stack<String> s = new Stack<String>();
s.push("dog"); ...  // push objects onto top of stack
while (!s.isEmpty())
    ... s.pop();   // removes/returns top object
```

- Also trees, sets, hash tables, … – *more about this in CS 20*

Using methods – "invoking"

- Can look like a direct translation of an algorithm

```
getData();
process();
showResults();
```

- Then `process()` might use another method

```
result = calculate(x, y);
```

  where `calculate` returns a value based on `x` and `y`.
- And so on …
  - Translates top-down program design to method calls

Invoking methods (in formal terms)

- `methodName(`*list of arguments*`);`
  - Transfers control to the method named; may "pass" data via the list of arguments
  - After the method completes (or aborts) its work, control returns to the calling statement
  - Some methods also return some results
- Actual syntax: `objectReference.name(…)`
  - Or `ClassName.name()` if method is declared `static`
  - In same class, `this.` is implied