But the real ArrayList can hold *any type* of object

- You just have to specify type inside <angle brackets>
- Examples:

```java
List<String> dwarves = new ArrayList<>();
List<Integer> my_fave_numbers = new ArrayList<>();
```

Let's talk generics. A simple example:

- A class Pair<E> that manages a pair of the same objects
Aside: Generic Types (Cont.)

Generic type E specified here. E could be called anything.

```java
public class Pair<E> {
    private E first;
    private E second;

    public Pair(E first, E second) {
        this.first = first;
        this.second = second;
    }

    public E getFirst() {
        return this.first;
    }

    public E getSecond() {
        return this.second;
    }
}
```

(Let's update our ArrayList to support generics!)
Analysis of ArrayList: add(..)

- Interested in analyzing the **average case** of commonly used methods
  - **boolean add**(E new_item)
    - In most cases, it adds to tail of the list: $O(1)$
    - In rare cases, we may need to **reallocate** space and copy old elements over:
      - Seems like it should be $O(n)$?
      - Actually... we argue reallocation and copy is still only $O(1)$ !!!!
        » **Called amortized analysis**

- Excerpt from book (K&W pg. 74):

```
Doubling an array of size n allows us to add n more items before we need to do another [reallocation]. Therefore, we can add n new items after we have copied over n existing items. This averages out to 1 copy per add. Therefore, reallocation is effectively an $O(1)$ operation.
```
Analysis of ArrayList: add(..)

- Effect of Reallocation:
  - Array size doubles each time it reallocates called! (Exponential growth)
  - After 20 calls, the array can already hold over 1M elements!
    - Therefore, chances are, you're not having to reallocate a whole lot either!

![Array List Storage Growth Rate Graph]

- [Graph showing exponential growth of array size with reallocation calls.]
Okay, what about the 2-argument version of `add()`?

- `void add(int index, E new_item)`
  - May need to reallocate and copy again: amortized $O(1)$ again, but...
  - May need to shift items right:
    - Worst case, shift all $n$ items: $O(n)$
    - Average case, shift $n/2$ items: $O(n)$
    - (Best Case is just adding item to the tail - no shifts): $O(1)$
Interested in analyzing the *average case* of commonly used methods

- \( E \text{ get()} \) is \( O(1) \)
- \( E \text{ set()} \) is \( O(1) \)

These are due to arrays being used as the backing

- Remember, array accesses \( a[i] \) are constant time
- Nothing else being done in setter and getter methods that take a significant amount of steps
Analysis of ArrayList

- Interested in analyzing commonly used methods
  - `remove(int index)`
    - May need to shift `n` items left: $O(n)$
  - `indexOf(E item)`
    - Performs linear search to find item: $O(n)$
  - `toString()`
    - Need to traverse the entire array in order to construct String: $O(n)$
## Analysis of ArrayList Methods

<table>
<thead>
<tr>
<th>Signature</th>
<th>Average/Worst-Case Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>public E get(int index)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>public E set(int index, E item)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>public int size()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>public boolean add(E item)</td>
<td>$O(1)$ amortized</td>
</tr>
<tr>
<td>public boolean add(int index, E item)</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>public int indexOf(E item)</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>public E remove(int index)</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>public String toString()</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

Can we do better than linear time?
Analysis

- **ArrayList Pros:**
  - Grows and shrinks without users having to do it themselves
  - Fast when items are usually **added** to the tail of the list
  - Fast when items are usually **removed** from the tail of the list
  - Fast getters and setters
    - O(1)-time to use `get(i)` to grab *ith* item
    - O(1)-time to use `set(i)` to replace an item

- **ArrayList Cons:**
  - Slow when items are **added** to head position
  - Slow when items are **removed** from head position
  - *(What can we do about that?)*