# ArrayLists

COMP1400 Week 8

# Working with arrays

There are a number of common actions we want to do with arrays:

- adding and deleting
- copying
- looking for a particular element
- counting the elements

# Arrays

Arrays are intrinsically fixed-length storage.

Adding or deleting elements from arrays is cumbersome:

- create a new larger/smaller array
- copy the unaffected elements
- add/remove the affected element

# Abstraction

Rather than rewriting methods that perform these actions every time we use an array, we should abstract them into a new object.

What we want is a new class that implements a variable-length list.

The public interface should provide methods to implement the actions listed above.

# ArrayList

The ArrayList class in the Java Class Library provides this capability.

http://docs.oracle.com/javase/1.5.0/docs/api/java/util/ArrayList.html

# Encapsulation

The ArrayList class abstracts and encapsulates all the methods for implementing a variable-length list using fixed-length arrays.

```
ArrayList<String>
    private String[] myList;

    public void add(String s);
    public String get(int pos);
```

# Generic classes

Each ArrayList holds a specific type (or class) of object.

This class is given as a type parameter in angle brackets <> after the name.

The type of the methods **`get`**, **`set`** and **`add`** depend on the given type.

Parameterised types are called generic classes.

# Type parameters

When we create a variable that holds an ArrayList we also need to provide the type of objects it contains:

```
ArrayList<String> listOfStrings;
ArrayList<Integer> listOfInts;
ArrayList<Book> listOfBooks;
```

These are called the type parameters.

# Type parameters

Wh

ate a variable that holds an Arr also need to provide the type of objects it contains:

```
ArrayList<String> listOfStrings;

ArrayList<Integer> listOfInts;

ArrayList<Book> listOfBooks;
```

These are called the type parameters.

# Type parameters

Wh[illegible] ate [illegible] that holds an Arr[illegible] also [illegible] rovide the type of objects it contains:

collection class

item class

```
ArrayList<String> listOfStrings;

ArrayList<Integer> listOfInts;

ArrayList<Book> listOfBooks;
```

These are called the type parameters.

# Type parameters

Wh… ate … that holds an Arr… also … rovide the type of objects it contains:

collection class

item class

```
ArrayList<String> listOfStrings;

ArrayList<Integer> listOfInts;

ArrayList<Book> listOfBooks;
```

These are called the type parameters.

angle brackets

# List of Strings

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

classRoll.add("Sim Mautner");

String who = classRoll.get(0);

// who is now “Sim Mautner”
```

# List of ints

This does not work:

```
ArrayList<int> scores =
        new ArrayList<int>();

scores.add(1);

int s = scores.get(0);
```

The type parameter must be a class
**not** a primitive type.

# List of ints

Instead we use:

```
ArrayList<Integer> scores =
    new ArrayList<Integer>();

scores.add(1);

int s = scores.get(0);
```

The Integer class is an object-wrapper for the primitive type int.

# Wrapper Classes

Java implements wrapper classes to convert primitive types into objects automatically.

```
Integer intObject = 1;
Double doubleObject = 2.0;
Boolean boolObject = true;
Character charObject = ‘x’;
```

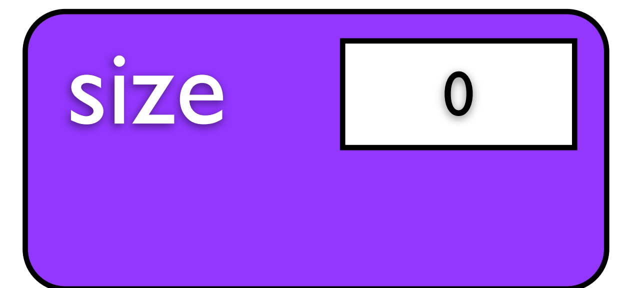
# Importing

Since ArrayLists are part of the JCL, we must first import them before use:

```
import java.util.ArrayList;
```

# Construction

To construct an ArrayList we will usually use the simple no-parameter constructor:




```
// create an empty list

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

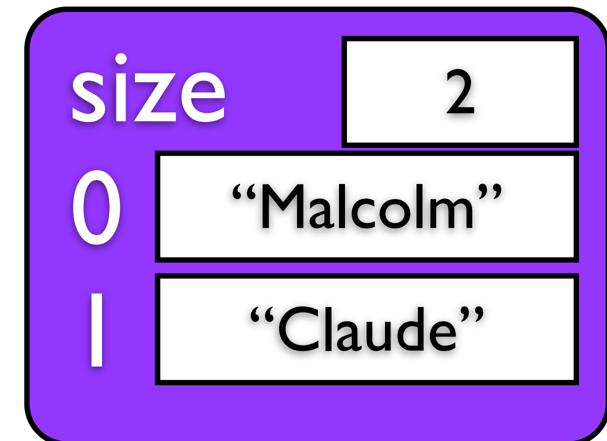
There are two other constructors but we won't be using them much.

# Adding items

A newly constructed list is empty.

We can add items using the **add** method.

```
classRoll.add("Malcolm");
classRoll.add("Claude");
```




The type of objects we add must match the type parameter.

```
classRoll.add(47); // ERROR
```

# Adding duplicates

We can add something to a list more than once.

```
classRoll.add("Malcolm");
classRoll.add("Malcolm");
```

| size | 4 |
| --- | --- |
| 0 | "Malcolm" |
| 1 | "Claude" |
| 2 | "Malcolm" |
| 3 | "Malcolm" |

# Size

We can use the **size** method to access the number of objects on the list:

```
int nStudents = classRoll.size();
// nStudents == 2
classRoll.add("Sim");
nStudents = classRoll.size();
// nStudents == 3
```

| size | 2 |
|---|---|
| 0 | "Malcolm" |
| 1 | "Claude" |

# Size

We can use the **size** method to access the number of objects on the list:

```
int nStudents = classRoll.size();
// nStudents == 2
classRoll.add("Sim");
nStudents = classRoll.size();
// nStudents == 3
```

| size | 3 |
| --- | --- |
| 0 | "Malcolm" |
| 1 | "Claude" |
| 2 | "Sim" |

# Get

We can access elements on the list by their index using the **get** method.

```
String me = classRoll.get(0);
// me == "Malcolm"
String him = classRoll.get(1);
// him == "Claude"
```

# Indexing

Note that the indices always start from 0 and end at size-1.

```
classRoll.size();
// returns 3
classRoll.get(0);
// returns "Malcolm"
classRoll.get(2);
// returns "Sim"
classRoll.get(3); // ERROR
```

| size | 3 |
|---|---|
| 0 | “Malcolm” |
| 1 | “Claude” |
| 2 | “Sim” |

# Remove

We can use the **`remove`** method to remove the element at a given index.

The other elements get renumbered.

```
classRoll.remove(1);

classRoll.size();
// returns 2

classRoll.get(1); // "Sim"
```

| | size | 3 |
|---|---|---|
| 0 | "Malcolm" | |
| 1 | "Claude" | |
| 2 | "Sim" | |

# Remove

We can use the **`remove`** method to remove the element at a given index.

The other elements get renumbered.

```
classRoll.remove(1);

classRoll.size();
// returns 2

classRoll.get(1); // "Sim"
```

| size | 2 |
|---|---|
| 0 | "Malcolm" |
| 1 | "Sim" |

# Copying

There is a special constructor for creating a copy of a list:

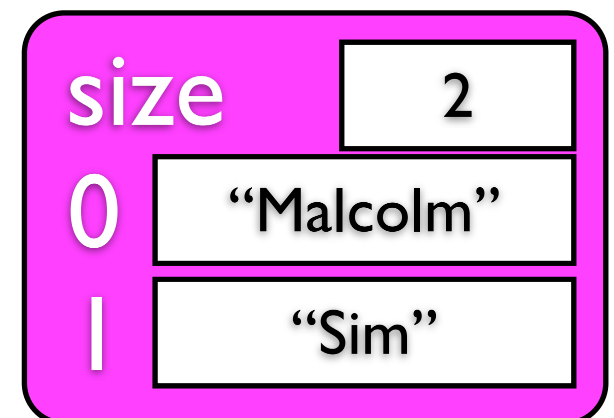
```
ArrayList<String> copy =
   new ArrayList<String>(
              classRoll);

copy.add("Troy");
```

# Copying

There is a special constructor for creating a copy of a list:

```
ArrayList<String> copy =
   new ArrayList<String>(
            classRoll);
```

```
copy.add("Troy");
```

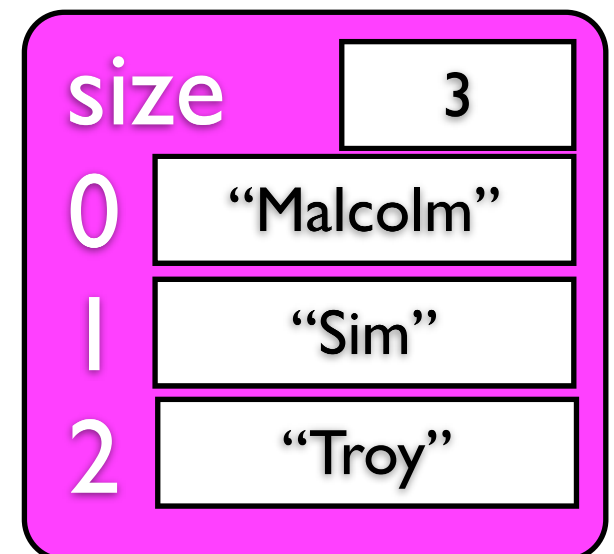


# Copying

There is a special constructor for creating a copy of a list:

```
ArrayList<String> copy =
   new ArrayList<String>(
              classRoll);
```

```
copy.add("Troy");
```



# Other methods

There are other methods you can read about in the API documentation:

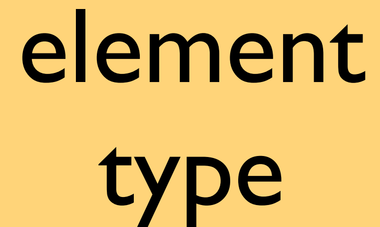
- **`clear`** - remove all elements
- **`set`** - set value of a given element
- **`insert`** - insert an element at an index
- **`contains`** - test if an element is on the list

# For-each loop

A for-each loop is used to iterate over all the elements of a collection:

```
for (String who : classRoll) {
   // action performed on
   // each element
   System.out.println(who);
}
```

# For-each loop




A f op is used to iterate over all the elements of a collection:

```
for (String who : classRoll) {
  // action performed on
  // each element
  System.out.println(who);
}
```

# For-each loop

element type

element variable

A f op is terate over all the elements of a collection:

```
for (String who : classRoll) {
  // action performed on
  // each element
  System.out.println(who);
}
```

# For-each loop

element type

element variable

list variable

A for-each loop is used to iterate over all the elements of a collection:

```
for (String who : classRoll) {
  // action performed on
  // each element
  System.out.println(who);
}
```

# Arrays vs ArrayLists

```
// declaring array variables

String[] arrayOfStrings;
int[] arrayOfInts;

// declaring list variables

ArrayList<String> listOfStrings;
ArrayList<Integer> listOfInts;
```

# Arrays vs ArrayLists

```
// constructing arrays

arrayOfStrings = new String[5]
arrayOfInts = new int[10]

// constructing lists (empty)

listOfStrings =
      new ArrayList<String>();
listOfInts =
      new ArrayList<Integer>();
```

# Arrays vs ArrayLists

```
// measuring size

int arraySize =
    arrayOfStrings.length;

int listSize =
    listOfStrings.size();
```

# Arrays vs ArrayLists

```
// getting elements

String s1 = arrayOfStrings[2];
int i1 = arrayOfInts[1];

String s2 =
     listOfStrings.get(2);
int i2 =
     listOfInts.get(1);
```

# Arrays vs ArrayLists

```
// setting elements

arrayOfStrings[2] = "Malcolm";
arrayOfInts[1] = 40;

listOfStrings.set(2, "Malcolm");
listOfInts.set(1, 40);
```

# Arrays vs ArrayLists

```
// iterating

for (int i = 0;
     i < arrayOfStrings.length;
     i++) {

    // process arrayOfStrings[i]

}

foreach (String s : listOfStrings) {

    // process s

}
```

# Arrays vs ArrayLists

```
// Copying

String[] copyArray =
     new String[arrayOfStrings.length+1];
for (int i = 0;
     i < arrayOfStrings.length;
     i++) {
   copyArray[i] = arrayOfStrings[i];
}

ArrayList<String> copyList =
    new ArrayList<String>(listOfStrings);
```

# Arrays vs ArrayLists

```
// Adding an element

String[] copyArray =
      new String[arrayOfStrings.length+1];
for (int i = 0;
      i < arrayOfStrings.length;
      i++) {
   copyArray[i] = arrayOfStrings[i];
}
copy[arrayOfStrings.length] = value;
arrayOfStrings = copyArray;

listOfStrings.add(value);
```

# Other collections

Sets (unordered, no duplicates):

http://docs.oracle.com/javase/1.5.0/docs/api/java/util/HashSet.html

Maps (look-up tables with key/value pairs):

http://docs.oracle.com/javase/1.5.0/docs/api/java/util/HashMap.html