Polymorphism and Arrays

- With polymorphism, we can have (for example) the interfaces `Animal` and `Comparable` and classes that implement them called `ComparableCat` and `ComparableDog`.

- We can then create an array of `Animal` references or an array of `Comparable` references, either of which could contain both `ComparableCat` and `ComparableDog` objects.

- However, if we want to invoke any method that is not defined within the interface on an object, we have to explicitly cast to a specific type like `ComparableCat` or `ComparableDog` before doing so.
Multi-use Data Structures

• What if we wanted to create a more complex data structure that could contain any type of object?
• We could have it contain `Object` references, but then we would need to cast things every time we wanted to use them.
• We would also potentially need to write a great deal more code for error-checking and/or error-handling and would have less compiler-level checking possible.

Generics

• In C++ you can have a template data structure for which you explicitly say what type of value it can hold when you declare the structure.
• In Java, a similar feature was added in Java version 5.0 which is called Generics.
**ArrayList<**Type**>**

- A useful "collection" data structure provided by Java is an array-based, resizable list.

- It has similarities to the `StringBuffer` class in how it can have a structure behind the scenes that has a greater capacity than its utilized size.
  - Unlike with `StringBuffer`, we can not access the current capacity information.
  - We do have a method `ensureCapacity()` that can be used before a large number of additions that will grow the internal structure to at least that size in a single operation.

---

**Declaring and Filling an ArrayList**

```java
ArrayList<Integer> arrName;
arrName = new ArrayList<Integer>();

arrName.add(11);
arrName.add(20);
arrName.add(2010);

//This line would NOT compile.
arrName.add("hi");
```
Copying an ArrayList

ArrayList<Integer> newArr;
newArr = new ArrayList<Integer>(oldArr);

Iterable<Type>

• Among other things, the ArrayList<Type> class is a Collection that implements the Iterable<Type> generic interface.

• We can iterate through each of the individual elements of an ArrayList<Type> object using the syntax of a "for each" loop:
  for (Typename iteratedVal : collection)
  {
     //process the iteratedVal object
  }

Iterable<Type>

• Among other things, the ArrayList<Type> class is a Collection that implements the Iterable<Type> generic interface.

• We can iterate through each of the individual elements of an ArrayList<Type> object using the syntax of a "for each" loop:
  for (Typename iteratedVal : collection)
  {
     //process the iteratedVal object
  }
Iterating through an ArrayList

```java
for (Integer i : arrName) {
    System.out.print(i + " ");
}
System.out.println();
```

NOTE: You cannot alter a list while iterating through it. If you want to do something like that, you would need to create a duplicate of the list and iterate through that one while altering the other.

One way to delete all Even Numbers

```java
ArrayList<Integer> a2;
a2 = new ArrayList<Integer>(a1);
for (Integer i : a1) {
    if (i%2 == 0) {
        a2.remove(i);
    }
}
```
The idea of a stack

• Some data structures have very limited and strict access rules, though specific libraries can add non-standard access methods.

• We have discussed the idea of a stack previously when discussing memory.

• The standard ways to access a general use stack are via push() and pop() or peek().
  – The idea is to push a value onto the top of a stack and to pop a value off the top with no way to access anything not at the top. You can peek at the value on the top also.

Stack<Type>

• There is a Collection provided by Java called Stack.
• This Stack class is also generic class.
• It implements the push, pop, peek access methods as well as others which are part of the Collection interface, such as a search method contains, and a method to get an Iterator for the structure called iterator.
Our own stack implementation?

- What if we wanted to write our own `Stack` class which only had public methods that are explicitly part of the idea of a stack?

- We could hold the values in an `ArrayList` and could try to mimic some of the things we saw in `StringBuffer` and even try to "help out" the Java garbage collection algorithm.

Consider the following code:

```java
public static void main(String [] args) {
    Integer[] values = new Integer[10];
    int top = -1;
    for (int i=0; i<5; i++) {
        top++;
        values[top] = new Integer(i);
    }
    System.out.println(values[top]);
    top--;
    System.out.println(values[top]);
}
```

At the end of this code, is the Integer which contains the value 4 ready for garbage collection?