"Get your data structures correct first, and the rest of the program will write itself."
- David Jones
Abstract Data Types

- Abstract Data Types (aka ADTs) are descriptions of how a data type will work without implementation details
- Example, Stack at NIST DADS
  - [http://xw2k.nist.gov/dads/HTML/stack.html](http://xw2k.nist.gov/dads/HTML/stack.html)
- Description can be a formal, mathematical description
- Java interfaces are a form of ADTs
  - some implementation details start to creep in
A Data Structure is:

- an implementation of an abstract data type and
- "An organization of information, usually in computer memory", for better algorithm efficiency.

![Diagram of a List Object with elements A C E B A and size 5]
Data Structure Concepts

- Data Structures are containers:
  - they hold other data
  - arrays are a data structure
  - ... so are lists

- Other types of data structures:
  - stack, queue, tree,
    binary search tree, hash table,
    dictionary or map, set, and on and on
  - [www.nist.gov/dads/](http://www.nist.gov/dads/)

- Different types of data structures are optimized for certain types of operations
Core Operations

- Data Structures will have 3 core operations
  - a way to add things
  - a way to remove things
  - a way to access things
- Details of these operations depend on the data structure
  - Example: List, add at the end, access by location, remove by location
- More operations added depending on what data structure is designed to do
ADTs and Data Structures in Programming Languages

- Modern programming languages usually have a library of data structures
  - Java collections framework
  - C++ standard template library
  - .Net framework (small portion of VERY large library)
  - Python lists and tuples
  - Lisp lists
Data Structures in Java

- Part of the Java Standard Library is the Collections Framework
  - In class we will create our own data structures and discuss the data structures that exist in Java

- A library of data structures

- Built on two interfaces
  - Collection
  - Iterator

- [http://java.sun.com/j2se/1.5.0/docs/guide/collections/index.html](http://java.sun.com/j2se/1.5.0/docs/guide/collections/index.html)
The Java Collection interface

- A generic collection
- Can hold any object data type
- Which type a particular collection will hold is specified when declaring an instance of a class that implements the Collection interface
- Helps guarantee *type safety* at compile time
Methods in the Collection interface

```java
public interface Collection<E>
{
    public boolean add(E o);
    public booleanaddAll(Collection<? extends E> c);
    public void clear();
    public boolean contains(Object o);
    public boolean containsAll(Collection<?> c);
    public boolean equals/Object o);
    public int hashCode();
    public boolean isEmpty();
    public Iterator<E> iterator();
    public boolean remove(Object o);
    public boolean removeAll(Collection<?> c);
    public boolean retainAll(Collection<?> c);
    public int size();
    public Object[] toArray();
    public <T> T[] toArray(T[] a);
}
```
The Java ArrayList Class

- Implements the List interface and uses an array as its *internal storage container*
- It is a list, not an array
- The array that actually stores the elements of the list is hidden, not visible outside of the ArrayList class
- All actions on ArrayList objects are via the methods
- ArrayLists are generic.
  - They can hold objects of any type!
ArrayList's (Partial) Class Diagram
Back to our Array Based List

- Started with a list of ints
- Don't want to have to write a new list class for every data type we want to store in lists
- Moved to an array of Objects to store the elements of the list

```java
// from array based list
private Object[] myCon;
```
Using Object

- In Java, all classes inherit from exactly one other class except Object which is at the top of the class hierarchy.
- Object variables can point at objects of their declared type and any descendants – polymorphism.
- Thus, if the internal storage container is of type Object it can hold anything – primitives handled by *wrapping* them in objects. int – Integer, char - Character.
Difficulties with Object

- Creating generic containers using the Object data type and polymorphism is relatively straightforward.
- Using these generic containers leads to some difficulties:
  - Casting
  - Type checking
- Code examples on the following slides.
Attendance Question 1

- What is output by the following code?

```java
ArrayList list = new ArrayList();
String name = "Olivia";
list.add(name);
System.out.print( list.get(0).charAt(2) );
```

A. i

B. O

C. l

D. No output due to syntax error.

E. No output due to runtime error.
Assume a list class

```java
ArrayList li = new ArrayList();
li.add("Hi");
System.out.println(li.get(0).charAt(0));
// previous line has syntax error
// return type of get is Object
// Object does not have a charAt method
// compiler relies on declared type
System.out.println(
    ((String)li.get(0)).charAt(0));
// must cast to a String
```
Code Example – type checking

//pre: all elements of li are Strings
public void printFirstChar(ArrayList li) {
    String temp;
    for (int i = 0; i < li.size(); i++) {
        temp = (String) li.get(i);
        if (temp.length() > 0 )
            System.out.println(
                temp.charAt(0) );
    }
}

// what happens if pre condition not met?
Too Generic?

Does the compiler allow this?

```java
ArrayList list = new ArrayList();
list.add( "Olivia" );
list.add( new Integer(12) );
list.add( new Rectangle() );
list.add( new ArrayList() );
```

A. Yes
B. No
Is this a bug or a feature?
"Fixing" the Method

//pre: all elements of li are Strings
public void printFirstChar(ArrayList li){
    String temp;
    for(int i = 0; i < li.size(); i++){
        if( li.get(i) instanceof String ){
            temp = (String)li.get(i);
            if( temp.length() > 0 )
                System.out.println(
                                    temp.charAt(0) );
        }
    }
}
Generic Types

- Java has syntax for *parameterized data types*
- Referred to as *Generic Types* in most of the literature
- A traditional parameter *has* a data type and can store various values just like a variable
  ```java
  public void foo(int x)
  ```
- Generic Types are like parameters, but the data type for the parameter is *data type*
  - like a variable that stores a data type
Making our Array List Generic

- Data type variables declared in class header

```java
public class GenericList<E> {

    // The <E> is the declaration of a data type parameter for the class
    // - any legal identifier: Foo, AnyType, Element, DataTypeThisListStores
    // - Sun style guide recommends terse identifiers

    // The value E stores will be filled in whenever a programmer creates a new
    GenericList<String> li =
        new GenericList<String>();

```
Modifications to GenericList

- instance variable
  ```java
  private E[] myCon;
  ```

- Parameters on
  - add, insert, remove, insertAll

- Return type on
  - get

- Changes to creation of internal storage container
  ```java
  myCon = (E[]) new Object[DEFAULT_SIZE];
  ```

- Constructor header does not change
Using Generic Types

- Back to Java's ArrayList

```java
ArrayList list1 = new ArrayList();
- still allowed, a "raw" ArrayList
- works just like our first pass at GenericList
- casting, lack of type safety
```
Using Generic Types

ArrayList<String> list2 =
    new ArrayList<String>();

    for list2 E stores String
list2.add( "Isabelle" );
System.out.println(
    list2.get(0).charAt(2) ); //ok
list2.add( new Rectangle() );
// syntax error
// syntax error
Parameters and Generic Types

- **Old version**
  
  ```java
  //pre: all elements of li are Strings
  public void printFirstChar(ArrayList li){
  }
  ```

- **New version**
  
  ```java
  //pre: none
  public void printFirstChar(ArrayList<String> li){
  }
  ```

- **Elsewhere**
  
  ```java
  ArrayList<String> list3 = new ArrayList<String>();
  printFirstChar( list3 ); // ok
  ArrayList<Integer> list4 = new ArrayList<Integer>();
  printFirstChar( list4 ); // syntax error
  ```
Generic Types and Subclasses

ArrayList<ClosedShape> list5 =
    new ArrayList<ClosedShape>();
list5.add( new Rectangle() );
list5.add( new Square() );
list5.add( new Circle() );
// all okay

- list5 can store ClosedShapes and any descendants of ClosedShape