ADT’s, Collections/Generics and Iterators

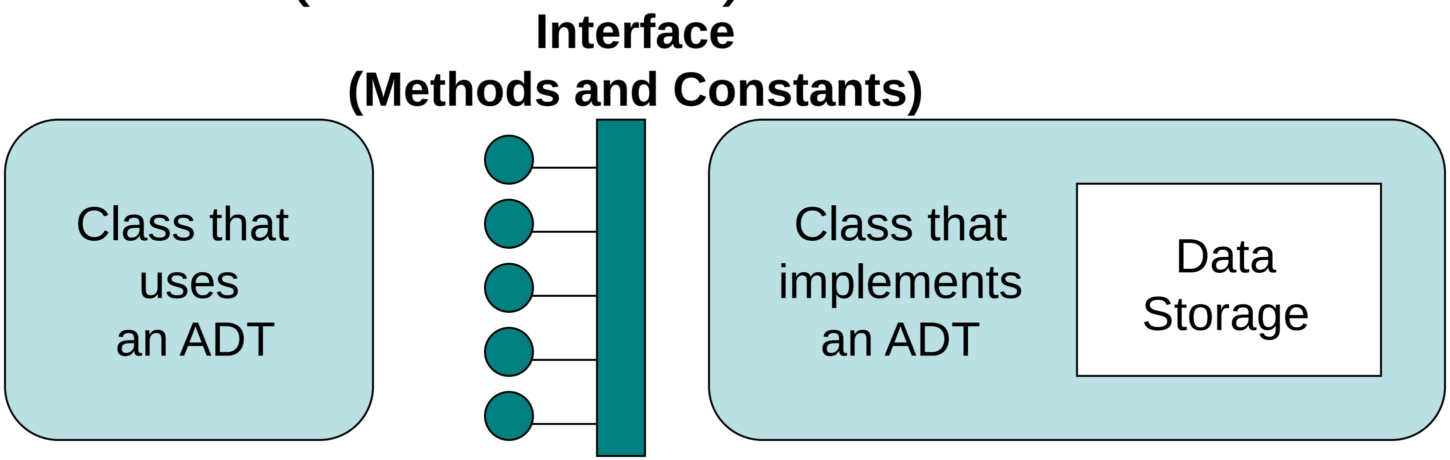
- Abstract Data Types (ADT’s)
- Collections / Stack Example
- Generics / Parameterized Classes
- Iterators
- Reading: L&C: 3.1-3.5, 7.1-7.2

Abstract Data Types (ADT's)

- A data type is a set of values and operations that can be performed on those values
- The Java primitive data types (e.g. int) have values and operations defined in Java itself
- An Abstract Data Type (ADT) is a data type that has values and operations that are not defined in the language itself
- In Java, an ADT is implemented using a class or an interface

Abstract Data Types (ADT's)

- An Abstract Data Type is a programming construct used to implement a data structure
  - It is a class with methods for organizing and accessing the data that the ADT encapsulates
  - The data storage strategy should be hidden by the API (the methods) of the ADT



Abstract Data Types (ADT's)

- The library code for Arrays.sort is designed to sort an array of Comparable objects:

```
public static void sort (Comparable [ ] data)
```

- The Comparable interface defines an ADT
  - There are no objects of Comparable "class"
  - There are objects of classes that implement the Comparable interface (e.g. the Polynomial class in our Project 1) with a `compareTo()` method
- Arrays.sort only uses methods defined in the Comparable interface, i.e. `compareTo()`

Collections

- The Java Collections classes are ADT’s that can be used to create container objects to hold and manage access to a collection of other objects
- In Java, these classes can be used similarly to Python lists, tuples, and/or dictionaries
- However, Java Collections are defined in classes (not in the language itself) so the programmer defines the most appropriate methods for adding and accessing the data objects they contain
- The Collections classes are parameterized to allow identification of the type of their contents

Parameterized Classes

- To allow the compiler to type check the contents of a collection class, the class definition is parameterized, e.g.

```
public class ArrayList<T> ... // fill in T
```

- Note: The letter used inside <> is a dummy and can be <T> like C++ or <E> like Oracle
- I prefer to use <T> based on C++ popularity and that is also what our textbook uses
- The type T must be a reference type - not primitive

Parameterized Classes

- Defining a parameterized class named Generic:

```
public class Generic<T> {
    // use T in attribute declarations
    private T whatIsThis;
    // use T as a method’s parameter type
    public void doThis(T input) { … }
    // use T as a method’s return type
    public T doThat( … ) {
        return whatIsThis;
    }
}
```

Parameterized Classes

- Instantiating parameterized class Generic

```
Generic<String> g = new Generic<String>();
```

- Use methods with objects of the actual type

```
g.doThis("Hello");
String s = g.doThat( … );
```

- The compiler can verify the correctness of any parameters passed or assignments of the return values
- No casting of data types should be required (If it is, you aren't using generics correctly)

Parameterized Classes

- Use a known class - not the dummy letter T

```
Generic<T> g = new Generic<T>();  // error
```

- Unless in a generic class where T is defined

```
public class AnotherGenericClass<T>
{
  …
    Generic<T> g = new Generic<T>(); // OK
  …
}
```

Parameterized Classes

- Sometimes we want to place a constraint on the class that can be used as T
- We may need T to be a type that implements a specific interface (e.g. Comparable)

```
public class Sorter<T extends
                        Comparable <T>>
{
   // now our code can call compareTo
   // method on type T objects here
}
```

Parameterized Classes

- Don’t omit an identified <type> in new code

```
Generic g = new Generic(); // legacy code?
```

- Compiler will give incompatible type errors without an explicit cast (narrowing)

```
String s = g.doThat( … );  // error
String s = (String) g.doThat( … );  // OK
```

- Compiler will give unchecked warnings

```
g.doThis(“Hello”);          // warning
```

Parameterized Classes

- Can’t instantiate arrays of the generic data type without using a “trick”

```
T [] t = new T[10];     // compile error
T [] t = (T []) new Object[10];   // OK
```

- Can’t instantiate arrays of a parameterized class without using a slightly different “trick”

```
ArrayList<String>[] a =
    (ArrayList<String>[]) new ArrayList[10];
```

  – Just casting a `new Object[10]` compiles OK but throws an exception at run time (Ouch!)

Parameterized Classes

- When you use either of the above “tricks”, the compiler will give you an “unchecked” warning
- Normally, we would “fix” our code to get rid of all compiler warnings but here we can’t “fix” it
- Use the compiler SuppressWarnings directive
- Place this line ahead of the method header

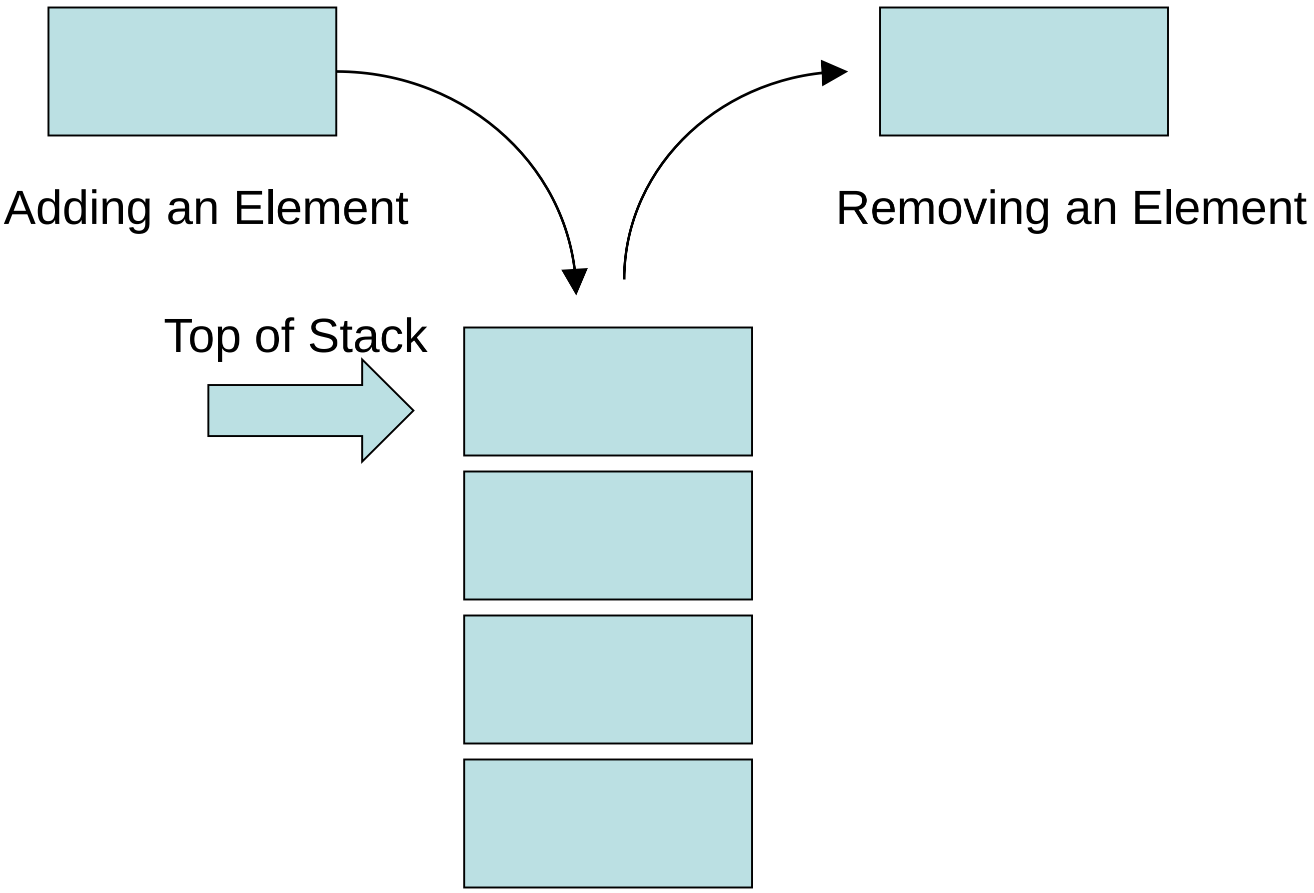
```
@SuppressWarnings("unchecked")
```

- That directive will allow a “clean” compilation

An Example Collection: Stack

- A *stack* is a linear collection where the elements are added or removed from the same end
- The access strategy is *last in, first out (LIFO)*
- The last element put on the stack is the first element removed from the stack
- Think of a stack of cafeteria trays

A Conceptual View of a Stack



An Example Collection: Stack

- A stack collection has many possible uses:
  - Reversing the order of a group of elements
    - Push all elements onto a stack and pop them
  - Evaluating Post-fix Expressions
    - Text example which we will cover later in the course
  - Back tracking in solution for a maze
    - Push previous location on a stack to save it
    - Pop previous location off the stack at a blind end
- We will discuss more uses for stacks later

Iterating over a Collection

- If we need to write code that retrieves all the elements of a collection to process them one at a time, we may use the “Iterator” design pattern from *Design Patterns*, Gamma et al.
- We call this *iterating over the collection*
- All Collection classes implement Iterable

```
Collection<T> extends Iterable<T>
```

- The Iterable interface requires one method:

```
Iterator<T> iterator();
```

Iterable Objects and Iterators

- An *Iterable* object allows you obtain an *Iterator* object to retrieve objects from it
  - `Iterator<T> iterator()` returns an Iterator object to access this Iterable group of objects
- An *Iterator* object allows you to retrieve a sequence of all T objects using two methods:
  - `boolean hasNext()` returns true if there are more objects of type T available in the group
  - `T next()` returns the next T object from the group

Iterable Objects and Iterators

- All classes in the Java Collections library implement the Collection interface and are Iterable OR you can implement Iterable in any class that you define
- If `myBookList` is an object of an <u>Iterable</u> class named List that contains `Book` objects

```
ArrayList<Book> myBookList = new ArrayList<Book>();
```

- We can retrieve all the available `Book` objects from it in either of two ways:

Iterable Objects and Iterators

- We can obtain an Iterator object from an Iterable object and use it to retrieve all the items from the Iterable object <u>indirectly</u>:

```
ArrayList<Book> bookList = new ArrayList<Book>();
// Code to add some Books to the bookList
Iterator<Book> itr = bookList.iterator();
while (itr.hasNext())
   System.out.println (itr.next());
```

- We can use the Java `for-each` loop to retrieve the contents of an Iterable object

```
for (Book myBook : bookList)
   System.out.println (myBook);
```

Iterable Objects and Iterators

- The Iterator hasNext() and next() methods do not modify the contents of the collection
- There is a third method called remove() that can be used to remove the last object retrieved by the next() method from the collection
- This is not always required for an application, but the method must be present to compile, so the code of the class that implements the iterator could throw an exception when remove is called