ADTs, Collection, Iterable/Iterator Interfaces

- Collections and the Java Collections API
- The Collection Interface and its Hierarchy
- The Iterable and Iterator Interfaces
- For-each Loops with Iterable Collections
- Introduction to Project 1
- Reading: L&C 3.1, 7.1, and Project 1
Collections

• A collection is a typical Abstract Data Type
• A collection is a class that contains and allows access to a group of objects
• The access “strategy” is in accordance with an abstract idea of the model for the collection
  – A Stack is a pile of objects, LIFO
  – A Queue is a waiting line of objects, FIFO
  – A Set is a group of objects (no duplicates)
  – A List is an indexed group of objects (duplicates OK)
The Java Collections API

• The classes in the Java Collections Library are named to indicate the underlying data structure and the abstract type of access

• For example, the `ArrayList` we studied in CS110 uses an underlying `array` as the data structure for storing its objects and implements its access model as a `list`

• However, from the user’s code point of view, the data structure is hidden by the API
The Collection Interface

• A class that implements the `Collection<T>` interface can contain a group of objects
• The type of objects that a Collection class contains can be specified using generics
• `ArrayList<T>` implements `Collection<T>`
• Hence, polymorphism allows us to do these:
  ```java
  ArrayList<String> a = new ArrayList<String>();
  Collection<String> c = a;  // widening
  ArrayList<String> d = (ArrayList<String>) c;
  ```
Collection Interface Methods

boolean add(T o)
addAll(Collection c)
void clear()
boolean contains(T o)
boolean containsAll(Collection c)
boolean equals(Object o)
int hashCode()
boolean isEmpty()
Collection Interface Methods

Iterator iterator()
boolean remove(T o)
boolean removeAll(Collection c)
boolean retainAll(Collection c)
int size()
Object [] toArray()
<T> T[] toArray(T[] a)
ArrayList Unique Methods

- Indexing is NOT a feature of all collections
- It is a unique feature of the ArrayList class
- Additional ArrayList methods involving indexing:
  - void add(int index, T o)
  - boolean addAll(int index, Collection c)
  - T get(int index)
  - int indexOf(T element)
  - int lastIndexOf(T element)
  - T remove(int index)
  - T set(int index, T element)
The Collection Interface Hierarchy

- Typically, the Collection interface is not implemented directly by a class.
- There is a hierarchy of interfaces that extend the Collection interface which extends the Iterable interface.
- Each subclass of the Collection interface is designed to support a specific model for access to the contents of the collection:
  - Stack, Queue, Set, List, etc.
The Collection Interface Hierarchy

<<interface>>
Iterable<T>

<<interface>>
Collection<T>

<<interface>>
List<T>

<<interface>>
Set<T>

<<interface>>
SortedSet<T>

<<interface>>
Queue<T>
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

Adding an Element
Top of Stack
Removing an Element
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*
  
  \[
  \text{Collection}<\text{T}> \text{ extends } \text{Iterable}<\text{T}>
  \]
  (which is another interface)
- The Iterable interface requires one method:
  
  \[
  \text{Iterator}<\text{T}> \text{ iterator}();
  \]
Iterable Objects and Iterators

• An *Iterable* object allows you obtain an *Iterator* object to retrieve objects from it

  \[
  \text{Iterator}\langle T \rangle \text{ iterator()} \text{ returns an Iterator object to access this Iterable group of objects}
  \]

• An *Iterator* object allows you to retrieve a sequence of \( T \) objects using two methods:

  \[
  \text{boolean hasNext()} \text{ returns true if there are more objects of type } T \text{ available in the group}
  \]

  \[
  \text{T next()} \text{ returns the next } T \text{ object from the group}
  \]
Iterable Objects and Iterators

• Classes in the Java standard class library that implement the Collection interface are Iterable OR you can implement Iterable in a class that you define (Project 1)

• If `bookList` is an object of an `Iterable` class that contains `Book` objects, we can retrieve all the available `Book` objects in either of two ways:
Iterable Objects and Loops

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

```java
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());

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

- The Java 5.0 for-each loop simplifies the access to the contents of an Iterable object

```java
for (Book myBook : bookList)
    System.out.println(myBook);
```
Introduction to Project 1

- In Project 1, you need to implement or add code to:
  - An Iterable class that contains the $N^2$ cells of a Kenken
  - A Cell class to represent each cell in the puzzle
  - An Iterator class that returns arrays of cells in the order of their rows, columns, and constraints
  - An isSuccess method that iterates over the puzzle and determines if each row, column, and constraint is valid

- The Iterator will return each cell two times:
  - In an array representing its row
  - In an array representing its column

- It will return a valid or an invalid Cell array for each constraint
UML Class Diagram for Project 1

KenkenValidator

+ main (args: String [ ] ) : void
+ KenkenValidator(name : String)
+ isSolution() : boolean

<<interface>>
Iterable<Cell[]>

+ iterator() : Iterator

Kenken

- puzzle : Cell [ ] [ ]
- constraints : Constraint [ ]
+ Kenken(file : Scanner)
+ toString() : String

<<instantiates>>

Cell

- value : int
+ Cell()
+ setValue(value : int) : void
+ getValue(): int
+ toString() : String

<<uses>>
Constraint

- value : int
- op : char
- array : Cell[ ]
+ Constraint (attributes)
+ isValid () : boolean

<<instantiates>>

KenkenIterator

{Kenken inner class}

- cursor : int
- cursorConstraint : int
+ KenkenIterator ()

<<uses>>

<<interface>>
Iterator<Cell[]>

+ hasNext() : boolean
+ next() : Object
+ remove() : void

<<uses>>
UML Sequence Diagram for Project 1

KenkenValidator

- Instantiates Kenken object
- return reference
- iterator()
- return reference
- hasNext()
- return true/false
- next()
- return Cell [ ]
- getValue()
- return int value

For-Each loop in isSolution()

Color green is used for constructors
Introduction to Project 1

• In Project 1 with an NxN array:
• hasNext() returns true 3xN times
• next() returns a one dimensional Cell [N] for N rows, N columns, and ~N constraints
• The iteration process g(N) is 3xN and is O(n)
• However, the code calling the hasNext() and next() methods processes N elements each time for overall g(N) = 3NxN or O(N^2) again
Introduction to Project 1

• For a KenkenIterator class:

• Class Header could be:

```java
public class KenkenIterator
    implements Iterator<Cell[]>
{
    private Cell[][] puzzle;
    private int cursor;
    private int cursorConstraint;
}
```

• Constructor could be:

```java
public KenkenIterator()
{
    cursor = cursorConstraint = 0;
}
```
Introduction to Project 1

• For a “rows only” KenkenIterator class:

• KenkenIterator hasNext method could be:

```java
public boolean hasNext()
{
    return cursor < puzzle.length;
}
```

• KenkenIterator next method could be:

```java
public Cell [] next()
{
    // careful: this returns an alias of the
    Cell [] value = puzzle[cursor];  // row
    cursor++;
    return value;
}
```