Data Structures

Arthur Hoskey, Ph.D.
Farmingdale State College
Computer Systems Department

Today's Lecture

- Generics
- Java Collections and Interfaces

What would a class look like that could hold one piece of integer data?

Class to Store an Integer

```
class Data
{
   private Integer x;
   public Integer getX()
   {
      return x;
   }

   public void setX(Integer newx)
   {
      x = newx;
   }
}
```

Data Class

Stores one piece of integer data. (Integer is wrapper class for int)

Class to Store an Integer

Class to Store an String

- We have a class that can hold one piece of integer data.
- What if we wanted a class that could store one piece of string data?
- Could we use the class we just wrote?

```
class Data
{
   private String x;
   public String getX()
   {
      return x;
   }

   public void setX(String newx)
   {
      x = newx;
   }
}
```

Data Class

Stores one piece of string data.

Class to Store an String

Class to Store an Boolean

- If we wanted to create a class that could hold boolean data, we would have to rewrite the Data class, yet again.
- For example…

```
class Data
{
   private Boolean x;
   public Boolean getX()
   {
      return x;
   }

   public void setX(Boolean newx)
   {
      x = newx;
   }
}
```

Data Class

Stores one piece of boolean data.

Class to Store an Boolean

- Rewriting the class every time we want to store a different type of data in it is inefficient and error prone.
- There is a better way to do this.
- Use generics instead.

Generics

Generics

- Generics allow you to write a class once that can be used with different data types.
- For example…

Class to Store Any Data Type

T will stand for a data type. T will be replace with a type when we declare an instance of Data.

```
class Data<T>
{
   private T x;
   public T getX()
   {
         return x;
   }

   void setX(T newx)
   {
         x = newx;
   }
}
```

Data Class

This version uses generics.

```
public staic void main(String[] args)
{
   Data<Integer> d = new Data<Integer>();

   d.setX(10);

   System.out.println(d.getX());

   Data<String> d2 = new Data<String>();
   d2.setX("Yanks");

   System.out.println(d2.getX());
}
```

main

Uses the generic version of Data.

Generics

- Now on to Java collections and interfaces…

Java Collections and Interfaces

ArrayList

- Java list collection that uses an array-based implementation.
- This collection stores its objects just like a normal array.
- Put values in and get values out of the collection using an index.
- **ArrayList resizes itself.**

ArrayList

- Java collection that is a wrapper around an array.

Store strings in the ArrayList

Create an instance of an ArrayList

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

a.add("Rose");
a.add("Mateo");
a.add("Jane");

for (String s: a)
{
    System.out.println(s);
}
```

Add data to the ArrayList

Print all the data in the ArrayList

ArrayList

LinkedList

- Java list collection that uses a doubly-linked implementation.
- It implements both the List and Deque interfaces.

```
LinkedList<String> list = new LinkedList<String>();

list.add("Rose");
list.add("Mateo");
list.add("Jane");

for (String s: list)
{
   System.out.println(s);
}
```

LinkedList

interface Collection<E>

- Base interface for some other collection interfaces (not all).
- E stands for the data type of the items in the collection.

- Contains methods for the following categories of operations:
  - Adding
  - Clearing
  - Contains
  - Removing
  - Iterator
- Classes that implement this interface should have two constructors:
  - Default constructor
  - Constructor that takes a Collection as a parameter.

interface Collection<E>

interface List<E>

- Inherits from Collection interface.

- An ordered collection.
- Can contain duplicates.
- Can manipulate elements in the list according to their indices.
- Implemented by the following classes:
  ArrayList, LinkedList, Vector

Note: ArrayList and Vector are basically both resizable arrays. They differ with respect to thread synchronization and a few other things.

interface List<E>

Create a List

- Create an empty list using **ArrayList** (ArrayList implements the List interface):

```
List<String> myList = new ArrayList<>();
```

Instance type is inferred from the variable data type

or

```
List<String> myList = new ArrayList<String>();
```

- Create a list from data using Arrays.asList (you can pass as many parameters as you want):

```
List<String> myList = Arrays.asList("a", "b", "c");
```

- Create a list from an existing array:

```
String[] myArray = {"a", "b", "c"};
List<String> myList = Arrays.asList(myArray);
```

Note: Arrays is a prewritten class in the JDK that contains static helper methods for dealing with arrays.

Create a List

- List example. ArrayList implements the List interface.

Declare interface reference

```
List<String> langList;
langList = new ArrayList<String>();
```

Create instance of ArrayList

```
langList.add("Java");
langList.add("C++");
langList.add("Python");
```

Use List interface reference to add items to the collection

```
for (String s : langList) {
   System.out.println(s);
}
```

List Example

Cannot Use Primitive Types in a Collection

- Can only use reference types as the data type in a collection.

```
List<int> myList = new ArrayList<>();
```

CANNOT use a primitive type as the data type. You will see a compile error similar to the following: "Unexpected type, required reference, found int".

```
// Use the wrapper type instead of a primitive type
List<Integer> myList = new ArrayList<>();
myList.add(10);
```

Will auto box the int data

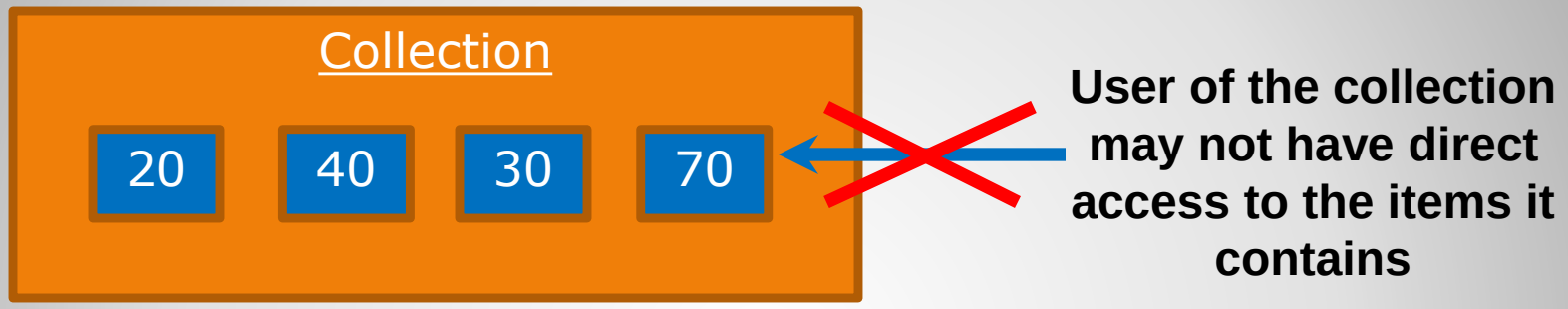
- You can write your own List collection.
- If you write a new class that contains a collection of some sort you could have your class implement the Java List interface.

List Interface and Custom Collections

General Description of Iterators

- **Iterators** – In general, objects which allow a program to traverse through a collection.
- Internals of a collection may be hidden (private) so there needs to be a way to access them all.
- Iterators are used to "visit" each element of a collection.

- Here is a collection with data (could be an array):

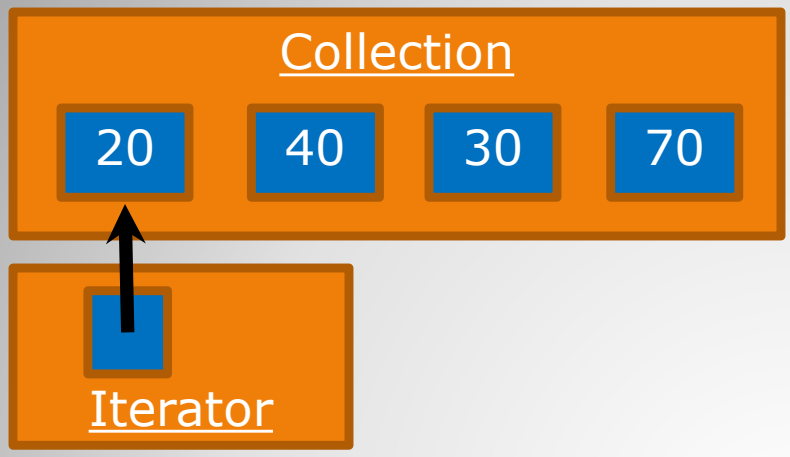


- If we want to print all items in this collection, we would not be able to in this case.

General Description of Iterators

General Description of Iterators

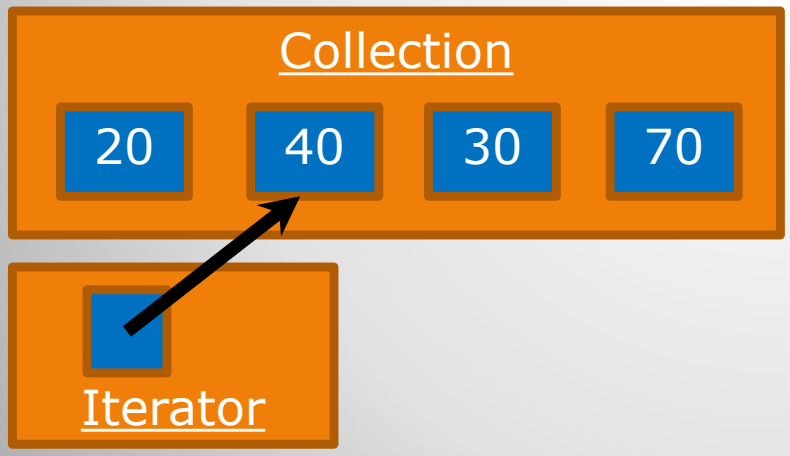
- Iterators are helper classes that have access to the items of the collection.
- An iterator "points at" one item of the class.
- In general, you can do the following with an iterator:
  - Get data from the current item.
  - Go to the next item in the collection.
  - Some iterators allow you to traverse the collection in reverse.
  - Some iterators allow you to remove items from the collection.
- For example…



This iterator points at the first item of the collection.

You can get the data (20) at that item if you want but not any other item's data.

If we told the iterator to go to the next item then it would look like the following….



Iterator now points at the second item.

You can get the data in the second item (40) but not the other items.

General Description of Iterators

interface Iterator<E>

- The Iterator interface allows you to traverse a collection from beginning to the end (not in reverse).

- Some methods:
  - **next** – Returns the data at the current item and moves the iterator to the next item.
  - **hasNext** – Returns true if there is another item in the collection after the current item and false otherwise.

interface Iterator<E>

interface Iterator<E>

- The Iterator interface allows you to traverse a collection from beginning to the end (not in reverse).

```
// Code to create langList here…
Iterator<String> iter;
iter = langList.iterator();


while (iter.hasNext()) {
   String current = iter.next();
   System.out.println(current);
}
```

Declare iterator reference

Get an iterator from a collection (assumes langList was created and populated with data)

Keep going while there is another item

Returns the current item and moves the iterator to the next item

interface Iterator<E>

interface ListIterator<E>

- Iterator used specifically for a list.
- Derived from the Iterator<E> interface.
- Allows for:
  - Traversing the list in reverse.
  - Adding new items into the list at the iterator's current location.
  - Some other functionality as well.

interface ListIterator<E>

Collections Class

- The Collections class is **different than the Collection interface discussed earlier**.
- Contains methods that can be used to manipulate and query a given collection.
- Only contains **static methods**.
- Here are a few of the methods on the Collections class…

Collections Class

Collections Class Methods

- **sort** – Sorts the elements of a list.
- **binarySearch** – Locates an object in a list.
- **reverse** – Reverses the elements of a list.
- **shuffle** – Randomly orders a List's elements.
- **min** – Returns the smallest element in a Collection.
- **max** – Returns the largest element in a Collection.

- For an exhaustive list of methods go to:

https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/Collections.html

Collections Class Methods

Collections Class Method (example for sort)

- The sort method puts the items in sorted order.

```
System.out.println("Original order");
for (String s : langList) {
   System.out.println(s);
}

// Sort the list
Collections.sort(langList);

System.out.println("Sorted order");
for (String s : langList) {
   System.out.println(s);
}
```

Call the sort method on collections (it is a static method so we call directly on the class)

Note: To sort a collection of a user-defined class that class must implement the Comparable interface.

sort

- **End of Slides**

End of Slides