CSC 335 - Collections

Dr. Jonathan Misurda
jmisurda@cs.arizona.edu

The `java.util` Framework

- This framework is used to solve a common problems with the storage and retrieval of data.
  - Maintain order.
  - Quickly access data.
  - Handle any type of data.
  - Reduce the need to rewrite code.

Framework

- A **framework** is a *design* and *implementation* artifact.
- It represents a specific *domain* (or an important aspect of one) and is:
  - Reusable
  - Built of abstract classes (or interfaces)
  - Implemented with concrete classes
- Good frameworks have:
  - Reusable implementations
  - Well-defined boundaries in how it interacts with clients
  - An implementation that is hidden from the outside.

Reusablility

- We very well could write our own collection and storage facilities, but instead of writing our own we can use the facilities provided and quickly and get on with the rest of our code.
- If we have a common storage facility then we can pass them to, or retrieve them from, code we didn't write.
- Why do it yourself if it's already done?

How Does It Work?

- The `java.util` framework is a set of abstract classes that encapsulate the functionality of a generalized storage facility.
- They have been implemented with concrete classes that can be instantiated and used according to their individual APIs.
- In English:
  - make an object.
  - use the methods provided by the object.
  - let it do the rest.

Primary Advantages

- **Reduces programming effort**
  - Provides data structures and algorithms so you don't have to write them yourself.
- **Increases performance**
  - Provides high-performance implementations of data structures and algorithms.
  - Because the various implementations of each interface are interchangeable, programs can be tuned by switching implementations.
- **Provides interoperability between unrelated APIs**
  - Establishes a common language to pass collections back and forth.

Advantages, cont.

- **Reduces the effort required to learn APIs** by requiring you to learn multiple ad hoc collection APIs.
- **Reduces the effort required to design and implement APIs** by not requiring you to produce ad hoc collections APIs.
- **Fosters software reuse** by providing a standard interface for collections and algorithms with which to manipulate them.

Java Collections

- Java's Collection Framework is a unified architecture for representing collections
- Collection framework contains
  - Interfaces (**ADTs**): specification not implementation
  - Concrete implementations as classes
  - Algorithms to search, sort, find, shuffle, ...

Abstract Data Type

- Abstract data type (ADT) is a specification of the behavior (methods) of a type
  - Specifies method names to add, remove, find
  - Specifies if elements are unique, indexed, accessible from only one location, mapped,...
  - An ADT shows no implementation
    - no structure to store elements, no implemented algorithms
- What Java construct nicely specifies ADTs?
  - Interfaces

Java Interfaces

```
/**
   This interface describes a
 collection of unique elements
*/
public interface Set<E> {
  // Add element to the Set if it is
  // not already in the Set and return
  // true. Return false if not.
  public boolean add(E element);

  // Returns true if element is in
  // the Set or false if not.
  public boolean contains(E element);

  // Remove element from this Set if
  // it is the Set. If element is not
  // in the Set, the Set is not
  // changed.
  public void remove(E element);

  // Remove true if there are no
  // elements in the Set, or false if
  // one or more elements are in
  // the Set.
  public boolean isEmpty();
}
```

Java Generics

- We want to be able to hold a collection of anything, including things of disparate data types
- We make the observation that every class we create IsAn Object
- So we could just have our collections hold Objects
- Then Set becomes:
  - add(Object o)
  - contains(Object o)
  - ...

Object-based Collections

- This is fine until we want to either:
  - Get an element from a collection:
    `Person p = collection.get(0); //Error`
    - Can't narrow Object to Person without an explicit cast:
    `Person p = (Person)collection.get(0);`
  - Use static typing to ensure that we aren't adding elements that don't belong:

```
Object[] arrOfPeople = new Object[10];
arrOfPeople[0] = new Person();
arrOfPeople[1] = new Student();
arrOfPeople[2] = “Bob”; //No error, String IsAn Object
```

What About Primitive Types

- All classes extend Object, but the primitive types (byte, int, short, long, char, boolean, float, double) do not
- We have wrapper Objects that hold these to use in this case (Integer, Byte, Short, etc.)
- The Java compiler will help us construct these wrappers automatically in doing what is called **auto-boxing**

Auto-boxing

- We can do:

```
Object b = 3; // No error
```

- Because the compiler rewrites our code to auto-box 3 into the appropriate java.lang.Integer class:

```
Object b = new Integer(3);
```

- In some circumstances, the compiler can even “unbox” something

Templates

- We can insert type information to get the compiler to verify that our collection is holding the proper things and also avoid having the tedious casts when we extract them
- The way we do this is by using a **template**
- A templated class (or method) allows the use of a type as a parameter to the class (or method) in the same sense as we can parameterize it by data

Using Templates

```
ArrayList<Person> arrOfPeople = new ArrayList<Person>();

arrOfPeople.add(new Person());
arrOfPeople.add(new Student());
arrOfPeople.add(“Bob”); //Error
```

- This is the older Java template style, where we have to double the templated type in both the reference declaration and the construction of the class, newer Java lets us use the diamond operator to type less:

```
ArrayList<Person> arrOfPeople = new ArrayList<>();
```

Type Erasure

- The inclusion of generics in Java happened in Java 1.5 (Java 5), well after Java already existed
- Java “stole” the template syntax from C++ which uses the same concept, but implements it differently:
  - In C++, if you create two instances of a templated class with different types, it creates two different objects, each with the actual type replacing the generic type:

```
std::vector<int> // contains an actual int[] array
std::vector<float> // contains an actual float[] array
```

  - In Java both `ArrayList<Integer>` and `ArrayList<Float>` contain an `Object[]` array.

Type Erasure

- Java doesn’t create different class instances based upon the type, instead it uses a concept of **type erasure**
  - Type erasure means that the parameterized type only exists at compile time, but is lost at runtime

```
class MyClass<T> {
    private final T o;

    public MyClass() {
        this.o = new T();
    }
}
```

Type Erasure

- Java doesn't create different class instances based upon the type, instead it uses a concept of **type erasure**
  - Type erasure means that the parameterized type only exists at compile time, but is lost at runtime

```
class MyClass<T> {
    private final T o;

    public MyClass() {
        this.o = new T();
    }
}
```

```
javac MyClass.java
MyClass.java:5: error: unexpected
type
        this.o = new T();
                     ^
  required: class
  found:    type parameter T
  where T is a type-variable:
    T extends Object declared in
class MyClass
1 error
```

Type Erasure

- What did the compiler do?
  - In Java, generics are implemented by finding a type that can represent all things that T is allowed to be
- If T is **unbounded**, T is replaced by the compiler with Object
- **Bounded** types allow you to limit what a template accepts using the inheritance hierarchy
  - `<E extends Person>` says to only allow types that are a subclass of Person
  - In this case, the type isn't erased to Object, but rather to Person

java.util.Collection interface

```
public interface Collection<E> extends Iterable<E>
    • boolean add(E e)
    • boolean addAll(Collection<? extends E> c)
    • void clear()
    • boolean contains(Object o)
    • boolean containsAll(Collection<?> c)
    • boolean equals(Object o)
    • int hashCode()
    • boolean isEmpty()
    • Iterator<E> iterator()
    • boolean remove(Object o)
    • boolean removeAll(Collection<?> c)
    • int size()
    • Object[] toArray()
    • <T> T[] toArray(T[] a)
    • …
```

Wildcard

- The question mark (?) here means any type
- `<? extends E>` means any type that extends E

java.util.List ADT

```
public interface List<E> extends
Collection<E> {
  int size();
  boolean isEmpty();
  boolean contains(Object o);
  Iterator<E> iterator();
  Object[] toArray();
  boolean add(E e);
  boolean remove(Object o);
  void clear();
  E get(int index);
  E set(int index, E element);
  void add(int index, E element);
  E remove(int index);
  int indexOf(Object o);
  int lastIndexOf(Object o);
}
```

A List ADT written as a Java interface

- interface List<E> defines a collection with a first element, a last, and distinct predecessors and successors, can insert anywhere
- Duplicate elements are allowed

```
List<String> list = new ArrayList<>();
list.add("Abc");
list.add(0, "Def");
assertEquals("Def", list.get(0));
assertEquals("Abc", list.get(1));
```

Iterators

- Iterators provide a general way to visit all elements in a collection

```
List<String> list = new ArrayList<>();
list.add("1-FiRsT");
list.add("2-SeCoND");
list.add("3-ThIrD");
list.add("4-foURtH");
Iterator<String> itr = list.iterator();
while (itr.hasNext()) {
  System.out.println(itr.next().toLowerCase());
  System.out.println(itr.next().toUpperCase()); // Bad
}
```

- We don't want to call next() twice without asking if the iterator hasNext() in between – capture itr.next() in a variable if we want the object twice per iteration

Java's Enhanced for Loop

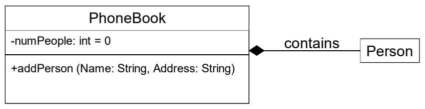
- General form:

```
    for (Type element : collection) {
        element is the next thing visited each iteration
    }


for (String str : list) {
  System.out.println(str.toLowerCase());
  System.out.println(str.toUpperCase());
}
```

Iterator Pattern

- Recurring Problem:
  - You have an aggregate data type that you want to access sequentially
  - How to not reveal the specific representation used to contain these data, while iterating over that data?
- Example: PhoneBook
  - How can you access the individual entries without exposing the PhoneBook's underlying representation?

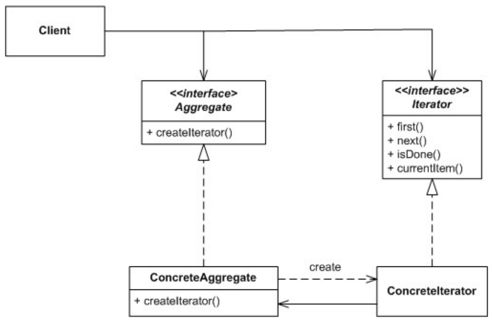


Iterator Pattern

- Solution:
  - Provide a separate object, an *iterator*, that can retain the current position, and go to the next element
  - The iterator knows about the underlying data structure, but doesn't reveal this information
- Also known as:
  - Cursor

P-28

UML for the Iterator Pattern



Participants

- Aggregate
  - This is an interface that defines a method for creating an Iterator object.
- ConcreteAggregate
  - Implements the createIterator( ) method to return an instance of the ConcreteIterator.
- Iterator
  - This is an interface for accessing and traversing elements of the aggregate.
- ConcreteIterator
  - This concrete class implements the Iterator interface.
  - It uses private fields to retain the current position within ConcreteAggregate.

Explanation

- The ConcreteIterator is often a private inner class. All the caller knows is that it is an Iterator.
- We can use polymorphism to allow multiple ConcreteAggregator classes, each with their associated ConcreteIterator classes.
- Iterator.next() returns the type aggregated by Aggregator

Java's Iterator Interface

- The Iterator<E> interface (Iterator) must be implemented by the iterator object's class (the ConcreteIterator).
- It has only four methods.
  - hasNext(): boolean
  - next(): <E>
  - remove()
    - removes the last element returned by the iterator (optional operation).
  - forEach()