DEPAUL UNIVERSITY
COLLEGE OF COMPUTING AND DIGITAL MEDIA

CSC 281 NOTES
Spring 2019: Amber Settle

Wednesday, May 22, 2019

Announcements

- Discuss the sixth assignment
- The seventh assignment is due at 9 pm – questions?
- The eighth assignment is due Wednesday, May 29th

Generic types

We often need to **implement classes and methods that work with multiple types**.

For example, it was awkward and frustrating that we had to define several different **printArray** methods, each of which was doing the exact same thing but on arrays of varying types.

It would have been much more convenient to declare **a single method that printed an array**, regardless of what the type is, since as long as the type has a toString method, we can print it.

A **generic class or method** is a class or method with one or more type parameters.

The type parameters can be required to be a subtype of one or more types or can be left open.

You can also use wildcards to specify that a method can accept an instantiation of a generic type with a subclass or a superclass argument.

We will look at generics in the following order:

1. Generic methods that are not part of a class
2. Built-in generic classes
3. Generic classes that we define

Generic methods: Array printing, redux

Recall that we defined more than three different 2D array printing methods, each designed for a different type.

What we want to do here is see how to **create an array printing method that works on any 2D array of any type**, so long as that type has toString defined for it (which every built-in class does).

To create a generic method you need to write a header that looks like the following:

```
visibility_mod static <T> returnType
method_name(parameters)
```

where the parameters use the type `T` as the type of at least one of the parameters (and `static` is omitted if not needed).

Example: **To create our print array method** (to be called from main) we would write:

```
public static <T> void printArray(T[][] array)
```

where T is the type we will pass to our array.

Caveat: **The type T cannot be primitive**. If we want an array of integer or double values, we need to use the wrapper types Integer and Double.

Example: Create a generic printArray method and a main that tests it and put it into the file **ArrayPrint.java**.

Built-in generics: ArrayList

Although they are (relatively) simple to use and are efficient, arrays have a significant drawback: **they cannot be resized once they are created**.

For example, `T[] anArray = new T[100];` creates an array of 100 elements of type T.

If you want to put more than 100 elements in it, you need to create a new array that's larger, copy the existing elements into the new array, and then continue with your work.

An **ArrayList** object is one that **can be resized** as you need to put more elements in it. As a result, it can grow as large as you need.

It does this by managing an array internally.

When that array becomes too small or is insufficiently utilized, another internal array is automatically created, and the elements are moved into it.

This process is invisible to the programmer.

To create an ArrayList, you write:

```
ArrayList<T> name;
```

where T is the type of the ArrayList and name is the name of the variable that will store the ArrayList.

Example: `ArrayList<String> cats;`

You then **instantiate the ArrayList by calling the constructor**:

```
name = new ArrayList<>();
```

or

```
name = new ArrayList<T>();
```

Example: `cats = new ArrayList<>();`

You then have the following **operations on ArrayList objects** (see the API for more details):

- `void add(E e)`: Add the element e of type E into the ArrayList. The ArrayList must be defined on type E.
- `void add(int index, E e)`: Add the element e of type E into the ArrayList in the specified position index.
- `boolean contains(Object o)`: Return true if the ArrayList contains the object o, and false otherwise.
- `E get(int index)`: Return the element in the ArrayList in position index.
- `int indexOf(Object o)`: Return the index of the first matching object o in the ArrayList or -1 if there is no matching object.
- `int lastIndexOf(Object o)`: Return the index of the last matching object o in the ArrayList or -1 if there is no matching object.
- `E remove(int index)`: Remove and return the item in the specified index.
- `boolean remove(Object o)`: Remove the first occurrence of the specified object if it is present. Returns true if the operation worked and false if it failed to remove anything.
- `int size()`: Return the number of elements in the ArrayList.

- `Object[] toArray()`: Return a regular array of the elements found in the ArrayList.

    Note that to use the `ArrayList` type you have to **import it from java.util**.

Example: Write a function to enter an arbitrary number of Strings from the user and save it into an `ArrayList` in the file **EnterStrings.java**.

Generic classes

A generic class is a class with one or more type parameters.

The parameters are used immediately after the name of the class:

```
public class ClassName <param1, param2, …> {
      …
      // definition in which param1, param2, … are
      used
}
```

Note: Type parameters cannot be instantiate with primitive types.

As a first example, consider a class for storing type, value pairs (**Entry.java**) and a driver program that uses it (**TestEntry.java**).

Type bounds

Sometimes the type parameters of a generic class or method need to fulfill certain requirements.

You can **specify a type bound** to require that the type extends certain classes or implements certain interfaces.

For example, if you wanted to ensure that you had an `ArrayList` of a type that can be treated as a number you could require that it extends the `Number` class.

A type parameter can have multiple bounds, in which case the bounds are connected together using **&** in the statement at the top of the class or method.

For example, we might want to compare numbers so that would require that we use the `Comparable` interface.

See the **GenericDemo.java** file for an example of using type bounds on a method.

Collections

The **Collection<E>** interface in the **java.util** package lets us use these data structures without having to code them ourselves.

> Recall: An **interface** specifies behavior but omits any implementation. The interface specifies a type for objects, and can be implemented in various ways by Java classes.

Collection<E>: Represents a group of objects of type E, known as its elements. It abstracts the behavior common to both sets and lists.

1. **Set**: Contains unsorted elements. We can determine if an element is in the set, but there is no order relation between one element and another in the set.
2. **List**: Supports a sequence in which elements are ordered from first to last.

All of these interfaces specify operations in two main groups:

- **Accessors** that get information, but do not change the collection.
- **Mutators** that change the collection.

The Collection<E> interface

The Collection<E> interface specifies the operations common to sets and lists. It contains the following operations:

The Accessors include:

- **boolean equals(Object o)**: Returns true if the Object o and the calling Collection contain the same set of elements and false if they do not.
- **boolean isEmpty()**: Returns true if the calling Collection is empty and false if it contains at least one element.
- **Iterator<E> iterator()**: A method that allows you to move through the elements of the Collection.
- **int size()**: Returns the size of the calling Collection.

The Mutators include:

- **boolean add(E o)**: Add the object passed in as a parameter to the calling Collection. Returns true if the operation succeeded.
- **boolean remove(Object o)**: Remove the Object o from the calling Collection. Returns true if the operation succeeded.

The following table lists some interfaces and classes that implement those interfaces.

Interface	Implementing class
Set	HashSet
List	ArrayList, LinkedList

The Iterator interface

Each collection provides a way to iterate through its elements in some order. The `Iterable<T>` superinterface of Collection defines a method:

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

which **yields an iterator you can use to visit all elements**.

For example:

```
Collection<String> coll = …;
Iterator<String> iter = coll.iterator();
While (iter.hasNext()) {
      String element = iter.next();
      // process element here
}
```

If you want to use a loop that visits each element once, there is **an enhanced for loop** that allows you to access the iterator:

For example:

```
for (String str: coll) {
      // process element
}
```

This is equivalent to the loop above.

The iterator interface also has a remove method which removes the previously visited elements.

For example, this loop removes all the elements that fulfill a condition:

```
while(iter.hasNext()) {
      String element = iter.next();
      if (element fulfills the condition)
            iter.remove();
}
```

The ListIterator interface

The `ListIterator` interface is a subinterface of `Iterator` with methods for adding an element before the iterator, setting the visited elements to a different value, and for navigating backwards.

You can create a ListIterator out of a List by using the following constructor:

- **ListIterator<E> listIterator(int index)**: Returns a list iterator over the elements in this list, starting with the specified index.

If you want to move backward through a list, you need to make sure that you provide an index and that it is equal to the list of the list.

The additional methods include the following:

- **boolean hasPrevious()**: Returns true if the iterator is not at the beginning of the list and false if it is at the beginning of the list.
- **int nextIndex()**: Returns the index past the current Object or -1 if the iterator is at the end of the list.
- **E next()**: Return the next element in the List<E>.
- **E previous()**: Returns the previous element in the List<E>.
- **int previousIndex()**: Returns the index of the element appearing before the current one or -1 if the current element is the first in the list.

Example: Re-write the `ArrayList` example that enters Strings from the user to use a `ListIterator` on an `ArrayList` and place it in the file **EnterStringsRedux.java**.