Interfaces & the Collections Framework

Lecture 11 of TDA 540 (Objektorienterad Programmering)

Carlo A. Furia Alex Gerdes

Chalmers University of Technology – Gothenburg University

Fall 2017

Pop quiz!

1. Go to `kahoot.it`
2. Enter PIN shown on projector screen
3. Pick a nickname and go!

Interfaces

Just like we can declare custom classes, we can declare custom interfaces:

```
public interface BankAccountInterface {
   int balance();
   void withdraw(int amount);
   void deposit(int amount);
}
```

- interfaces can only contain method declarations and constants (`static final` attributes)
- interface members are implicitly public (no need to use `public`)
- interface members cannot have implementations
- interfaces cannot be instantiated (they have no implementations)

An interface is like a list of operations that clients can use and that classes can implement together. An `interface` is Java's means to declare public interfaces of classes.

Interfaces and classes

Just like a class can inherit from another class, a class can implement an interface

```
class BankAccount implements BankAccountInterface {

  private int balance;

  int balance() { return balance; }
  // ... other implementations ...
```

- a class can implement one or more interfaces
- a class should provide implementations for all methods of the interfaces it implements; we do not use `@Override` because the class's is the first implementation (an interface has no implementations)
- a class can also introduce other members (private or public) without restrictions

Interfaces, inheritance, and types

Every interface `I` also corresponds to a type (operations on sets of values).

An interface also can inherit from one or more interfaces, by providing additional public methods (or constants).

```
interface BankAccountWithInterest extends BankAccountInterface {
   // add percent% interest to balance
   void payInterest(int percent);
}
```

Interface types and class types are related by inheritance:

- If `C` is a class that implements an interface `I`, we call the type of `C` a subtype of the type of `I`.
- if `J` is an interface that extends another interface `I`, we call the type of `J` a subtype of the type of `I`.

Abstract classes

Classes and interfaces are two opposite endpoints on a spectrum of abstraction:

(CONCRETE) CLASS	INTERFACE
complete implementation	no implementation
must have constructor	cannot have constructors
can be instantiated	cannot be instantiated
all visibilities	only `public` visibility
completely concrete	completely abstract

Abstract classes

Classes and interfaces are two opposite endpoints on a spectrum of abstraction:

(CONCRETE) CLASS	ABSTRACT CLASS	INTERFACE
complete implementation	partial implementation	no implementation
must have constructor	may have constructor	cannot have constructors
can be instantiated	cannot be instantiated	cannot be instantiated
all visibilities	all visibilities	only `public` visibility
completely concrete	partially abstract	completely abstract

Abstract classes

Methods and classes can be declared **`abstract`**:

- an **`abstract`** method lacks an implementation
- a class with at least one abstract method is an **`abstract class`**
- a class can be declared **`abstract`** even if it is full implemented
- an **`interface`** is like a completely abstract class (no implementations)
- an **`abstract class`** cannot be instantiated (and hence constructors cannot be abstract)

```
abstract class PartialBankAccount {  // partial implementation

  abstract int balance();

  abstract void withdraw(int amount);

  void deposit(int amount) { withdraw(-amount); }
}
```

The Collections framework

Java's Collections framework includes very carefully designed implementations of lists, as well as other data structures of common usage.

Even though they are more powerful and better optimized than our `List`, they follow some of the same design principles:

- Public interfaces are separated from implementations
- There are different implementations of the same `List interface`
- Implementations of the same interface can be used uniformly by clients without knowing implementation details
- Collections are generic: they can be used to store elements of an arbitrary reference type

The `List` interface

```
// generic interface of lists, for any reference type E
interface List<E>  {

  void add(int index, E element); // add ‘element’ at ‘index’

  E get(int index); // element at position ‘index’

  E remove(int index); // remove element at position ‘index’

  int size(); // number of elements in the list

  // ... several more methods are available ...
}
```

Implementations of the `List` interface

The `Collections` framework includes two main implementations of `List`: `ArrayList` and `LinkedList`.

- `ArrayList` is similar to our example: it uses an array to store data
  - `get` is very fast, `add` and `remove` are slower
- `LinkedList` stores data in a sequence of objects, each referencing the next node
  - `add` and `remove` are fast if we call them using iterators, `get` is slower

In practice the performance is very good for both unless you deal with really huge lists. Use `ArrayList` as default choice.

Both perform automatic resizing: if the list is full, it transparently allocates more memory (provided more memory is available). Thus, we do not have to worry about the list being full.

How to use the Collections framework

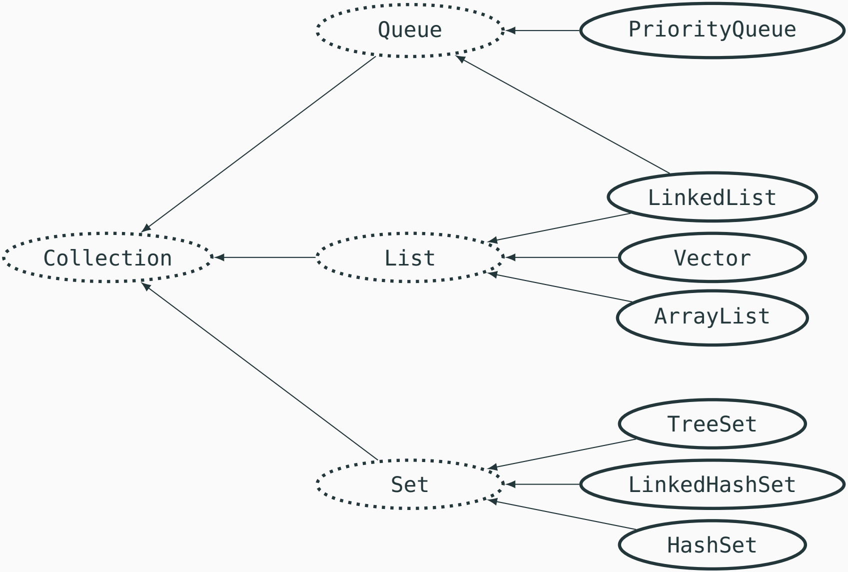
There is plenty of official documentation about the Collections framework online:

`https://docs.oracle.com/javase/8/docs/technotes/guides/collections/`

1. Select the interface that provides the operations your application needs
2. Select one implementation class of the interface that offers efficient implementation of those operations

In most cases, you do not have to worry too much about the implementation details.

Partial overview of the Collections framework



The Collections framework: some implementations

- `ArrayList`: indexed, dynamically growing
- `LinkedList`: ordered, efficient insertion and removal
- `HashSet`: unordered, rejects duplicates
- `TreeSet`: ordered, rejects duplicates
- `HashMap`: key/value associations (dictionary)
- `TreeMap`: key/value associations, sorted keys

Genericity in collections

Java offers a special syntax to define classes (and interfaces) that are generic with respect to one (or more) types. The types determined by generic classes are also called generic.

```
interface List<E>
```

- `List` is generic with respect to type parameter `E`
- For any concrete reference type `C`, `List<C>` is the interface that operates on type `C`, and `ArrayList<C>` is the implementation that operates on objects of type `C`
- Operations work for any choice of `C`; for example `get` returns elements of the chosen concrete type `C`
- While `C` can be anything, it is fixed once we declare an entity of a generic class

```
ArrayList<Integer> intList;  // intList stores Integers
ArrayList<String> strList;   // strList stores Strings
// we cannot put strings in intList, or integers in strList!
```

Polymorphism: subtypes and type compatibility

The subtype relation introduced by inheritance supports a powerful coding style using polymorphism:

- declare variables using the most general type `G`
- use the variables according to `G`'s interface
- flexibly switch between different concrete implementations of `G` (subtypes of `G`) without changing anything else in the program!

```
interface List<E> {
  E get(int index);
  void add(int index, E e);
  int size();
}
```

```
List<String> l;
l = // assign any List implementation
l.add(0, "hej");
l.add(1, " då");
if (l.size() > 0)
 System.out.println(l.get(0) + l.get(1));
```

Polymorphism: formal definitions

More rigorously, polymorphism is a type compatibility rule:

> If `S` is a subtype of `T`, an expression of type `S` can be used wherever an expression of type `T` is expected.

Since objects of type `S` are a specialization of objects of type `T` (a `Convertible` is a `Car`!), polymorphism still supports compiler checks and avoids type incompatibility errors.

```
interface List<E> {
  E get(int index);
  void add(int index, E e);
  int size();
}
```

```
List<String> l;
// the following operations are
// consistent with the List interface
l.add(0, "hej");
l.add(1, " då");
if (l.size() > 0)
 System.out.println(l.get(0) + l.get(1));
```

Polymorphism

Polymorphism provides a powerful abstraction mechanism for design:

- inheritance captures the relations between abstract models and implementations (e.g. `interface List` and `class ArrayList`), and among different variant implementations (e.g. `class ArrayList` and `class LinkedList`)
- code handles object uniformly at the appropriate level of abstraction, without depending on implementation choices
- decoupling between interfaces and implementations
- cohesion (consistency) on the shared types and operations
- component-based (bottom-up) construction of software

Polymorphism on client-side: example

Polymorphism provides a powerful abstraction mechanism for design

```
class CreditCard
{ BankAccountInterface account;

  void setPayments(BankAccountInterface ba)
  { account = ba; }

  List<Transaction> transactions;

  void pay(int nt) {
     Transaction tr = transactions.get(nt);
     if (tr != null) {
        account.withdraw(tr.amount());
        transactions.remove(nt);
     } }
}
```

Inheritance and collections

The subtyping relation introduced by inheritance applies to classes, not to collections of classes related by subtyping.

- `class Convertible extends Car`, thus `Convertible` is a subtype of `Car`
- `class Sedan extends Car`, thus `Sedan` is a subtype of `Car`
- the list type `List<Convertible>` (list of `Convertible`) is not a subtype of the list type `List<Car>` (list of `Car`)
- the list type `List<Sedan>` is not a subtype of `List<Car>`

Inheritance and collections

The subtyping relation introduced by inheritance applies to classes, not to collections of classes related by subtyping.

- `class Convertible extends Car`
- `class Sedan extends Car`
- the list type `List<Convertible>` (list of `Convertible`) is not a subtype of the list type `List<Car>` (list of `Car`)

```
// add a Sedan object to the end of List 'cars'
public static void addSedan(List<Car> cars)
{ cars.add(new Sedan()); }  // OK: a sedan is a car
```

If `Convertible[]` were a subtype of `Car[]`, we could write:

```
List<Car> convs = new List<Convertible>();
convs.add(new Convertible());  // add a convertible to list
addSedan(convs);  // add a sedan to list
```

A list of `Convertible` includes a `Sedan`, which is not a `Convertible`!

Live coding!

Let us design an array-based implementation of flexible lists.

Lists of anything

Very often, programs need to organize and access objects in some kind of list data structure:

- lists can store objects of any type (type generic)
- but the elements in a given list instance all have the same type (homogeneous)
- a list can store an arbitrary number of objects
- operations on a list:
  - access objects in the list at any position
  - add objects to the list at any position
  - remove objects in the list at any position

Arrays as lists

We have used class `Array` as lists.

- an array can store elements of arbitrary type (including primitive types)
- the size of an array is fixed and set upon creating it
- a default value may denote absence of element at that position

OPERATION	SYNTAX
declare list of type `T`	`T[] a = new T[10];`
number of stored objects (fixed)	`a.length`
object at position `k`	`a[k]`
add object `o` at position `k`	`a[k] = o;`
remove object `o` at position `k`	`a[k] = null;`

A different kind of list

Let us use `Array` to implement a more flexible class for lists.

- The size of a list can change dynamically
- A list is initially (when is created) empty
- Adding elements to a list increases the size of the list
- Removing elements from a list decreases the size of the list
- When we add an element in the middle of the list, the other elements shift position to make space for the new element
- When we remove an element in the middle of the list, the other elements shift position to close the gap left by the removed element
- We can still access an arbitrary element in the list by giving its index

To keep things simple, let us write the list class for elements of `Character` type only.

Custom class `List`: public interface

```java
public class ListInterface
{
   public ListInterface()
   { /* initialize an empty list */ }

   public int size()
   { /* number of elements in the list */ }

   public Character get(int index)
   {  /* return element at position ‘index’ */ }

   public void add(int index, Character e)
   {  /* add ‘e’ at position ‘index’ */ }

   public void remove(int index)
   {  /* remove element at position ‘index’ */ }
}
```

ListInterface as abstract

```
abstract public class ListInterface
{  // number of elements in the list
   abstract public int size();

   // return element at position ‘index’
   abstract public Character get(int index);

   // add ‘e’ at position ‘index’
   abstract public void add(int index, Character e);

   // remove element at position ‘index’
   abstract public void remove(int index); }

public class List extends ListInterface
{   // no override: first implementation
   public int size() { return size; } }
```

`ListInterface` as an `interface`

```java
public interface ListInterface
{  // number of elements in the list
   public int size();

   // return element at position 'index'
   public Character get(int index);

   // add 'e' at position 'index'
   public void add(int index, Character e);

   // remove element at position 'index'
   public void remove(int index); }

public class List implements ListInterface
{   // no override: first implementation
   public int size() { return size; } }
```

Example client of `List`

```
// create empty list
List list = new List();  // list: []
// insert in invalid position
list.add(2, 'X');  // list: []
// insert in valid position
list.add(0, 'X');  // list: [X]
list.add(1, 'Y'); // list: [X, Y]
list.add(2, 'Z'); // list: [X, Y, Z]
// remove
list.remove(1);   // list: [X, Z]
// insert back
list.add(1, 'A');  // list: [X, A, Z]
```

Custom class `List`: implementation

```java
public class List implements ListInterface
{  // maximum number of elements
   protected final int CAPACITY = 10_000;

   // non-public array to store elements
   protected Character[] elements;

   // how many elements are currently stored
   protected int size;
}
```

Custom class `List`: initialization

```
public class List implements ListInterface
{  // maximum number of elements
   protected final int CAPACITY = 10_000;

   // non-public array to store elements
   protected Character[] elements;

   // how many elements are currently stored
   protected int size;

   public List() {
     // make room for at most ‘CAPACITY’ elements
     elements = new Character[CAPACITY];
     // initially, the list is empty
     size = 0;
   }
```

Custom class `List`: `size`

```
public class List implements ListInterface
{  // maximum number of elements
   protected final int CAPACITY = 10_000;

   // non-public array to store elements
   protected Character[] elements;

   // how many elements are currently stored
   protected int size;

   public int size() { return size; }
```

Custom class List: get

```
public class List implements ListInterface
{  // maximum number of elements
   protected final int CAPACITY = 10_000;

   // non-public array to store elements
   protected Character[] elements;

   // how many elements are currently stored
   protected int size;

   public Character get(int index) {
      if (0 <= index && index < size)
         return elements[index]; // valid position: return element
      else // invalid position: return null
         return null;
   }
```

Custom class List: add

```
public class List implements ListInterface
{  protected final int CAPACITY = 10_000;
   protected Character[] elements;
   protected int size;

   public void add(int index, Character e) {
     // if ‘index’ is a valid insertion position
     if (0 <= index && index <= size) {
        // make room at position ‘index’
        // by shifting elements to the right
        for (int k = size; index < k; k--)
           elements[k] = elements[k - 1];
        elements[index] = e; // add ‘e’ at (freed) position ‘index’
        size = size + 1; // update size
     } }
```

Custom class `List`: `remove`

```
public class List extends ListInterface
{  protected final int CAPACITY = 10_000;
   protected Character[] elements;
   protected int size;

   @Override
   public void remove(int index) {
     // if ‘index’ is a valid position inside the list
     if (0 <= index && index < size) {
        // overwrite at position ‘index’
        // by shifting elements to the left
        for (int k = index; k < size - 1; k++)
           elements[k] = elements[k + 1];
        // update size
        size = size - 1;
     }
   }
```