Java: interfaces

Madhavan Mukund, S P Suresh

Programming Language Concepts

Lecture 9, 7 February 2023

Interfaces

- An interface is a purely abstract class
  - All methods are abstract
- A class implements an interface
  - Provide concrete code for each abstract function
- Classes can implement multiple interfaces
  - Abstract functions, so no contradictory inheritance
- Interfaces describe relevant aspects of a class
  - Abstract functions describe a specific “slice” of capabilities
  - Another class only needs to know about these capabilities

Interfaces express relevant capabilities

- Generic `quicksort` for any datatype that supports comparisons
- Express this capability by making the argument type `Comparable[]`
  - Only information that `quicksort` needs about the underlying type
  - All other aspects are irrelevant
- Describe the relevant functions supported by `Comparable` objects through an interface
- However, we cannot express the intended behaviour of `cmp` explicitly

```
public class SortFunctions{
  public static void quicksort(Comparable[] a){
    ...
    // Usual code for quicksort, except that
    // to compare a[i] and a[j] we use
    //  a[i].cmp(a[j])
  }
}
```

```
public interface Comparable{
  public abstract int cmp(Comparable s);
    // return -1 if this < s,
    //         0 if this == 0,
    //        +1 if this > s
}
```

Interactions with state

- Connect database query to logged in status of the user
- Use objects!
  - On log in, user receives an object that can make a query
  - Object is created from private class that can look up `railwaydb`
- How does user know the capabilities of private class `QueryObject`?
- Use an interface!
  - Interface describes the capability of the object returned on login

```
public interface QIF{
  public abstract int
    getStatus(int trainno, Date d);
}

public class RailwayBooking {
  private BookingDB railwaydb;
  public QIF login(String u, String p){
    QueryObject qobj;
    if (valid_login(u,p)) {
       qobj = new QueryObject();
       return(qobj);
    }
  }
  private class QueryObject implements QIF {
    public int getStatus(int trainno, Date d){
     ...
    }
  }
}
```

Interactions with state . . .

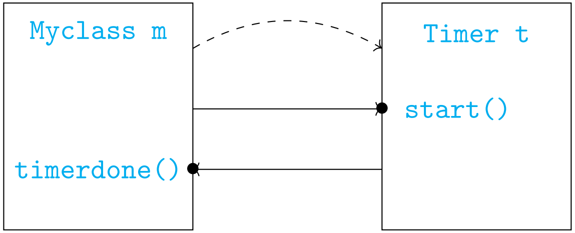
- Query object allows unlimited number of queries
- Limit the number of queries per login?
- Maintain a counter
  - Add instance variables to object returned on login
  - Query object can remember the state of the interaction

```
public class RailwayBooking {
  private BookingDB railwaydb;
  public QIF login(String u, String p){
    QueryObject qobj;
    if (valid_login(u,p)) {
       qobj = new QueryObject();
       return(qobj);
    }
  }
  private class QueryObject implements QIF {
    private int numqueries;
    private static int QLIM;

    public int getStatus(int trainno, Date d){
      if (numqueries < QLIM){
        // respond, increment numqueries
      }
    }
  }
}
```

Implementing a call-back facility

- Myclass m creates a Timer t
- Start t to run in parallel
  - Myclass m continues to run
  - Will see later how to invoke parallel execution in Java!
- Timer t notifies Myclass m when the time limit expires
  - Assume Myclass m has a function timerdone()



Implementing callbacks

- Code for `Myclass`
- `Timer t` should know whom to notify
  - `Myclass m` passes its identity when it creates `Timer t`
- Code for `Timer`
  - Interface `Runnable` indicates that `Timer` can run in parallel
- `Timer` specific to `Myclass`
- Create a generic `Timer`?

```
public class Myclass{

  public void f(){
    ..
    Timer t =
      new Timer(this);
      // this object
      // created t
    ...
    t.start(); // Start t
    ...
  }

  public void timerdone(){...}
}
```

```
public class Timer
       implements Runnable{
  // Timer can be
  // invoked in parallel

  private Myclass owner;

  public Timer(Myclass o){
    owner = o;  // My creator
  }

  public void start(){
    ...
    owner.timerdone();
    // I'm done
  }
}
```

A generic timer

- Use Java class hierarchy
- Parameter of `Timer` constructor of type `Object`
  - Compatible with all caller types
- Need to cast `owner` back to `Myclass`

```
public class Myclass{

  public void f(){
    ..
    Timer t =
      new Timer(this);
      // this object
      // created t
    ...
    t.start(); // Start t
    ...
  }

  public void timerdone(){...}
}
```

```
public class Timer
       implements Runnable{
  // Timer can be
  // invoked in parallel

  private Object owner;

  public Timer(Object o){
    owner = o;  // My creator
  }

  public void start(){
    ...
    ((Myclass) owner).timerdone();
    // I'm done
  }
}
```

Use interfaces

- Define an interface for callback

```
public interface
    Timerowner{

  public abstract
    void timerdone();
}
```

- Modify `Myclass` to implement `Timerowner`
- Modify `Timer` so that `owner` is compatible with `Timerowner`

```
public class Myclass
   implements Timerowner{

  public void f(){
    ..
    Timer t =
      new Timer(this);
      // this object
      // created t
    ...
    t.start(); // Start t
    ...
  }

  public void timerdone(){...}
}
```

```
public class Timer
       implements Runnable{
  // Timer can be
  // invoked in parallel
  private Timerowner owner;

  public Timer(Timerowner o){
    owner = o;  // My creator
  }

  public void start(){
    ...
    owner.timerdone();
    // I'm done
  }
}
```

Linear list

- A generic linear list of objects
- Internal implementation may vary
- An array implementation
- A linked list implementation

```
public class Linearlist {
  // Array implementation
  private int limit = 100;
  private Object[] data = new Object[limit];
  private int size;  // Current size

  public Linearlist(){ size = 0; }

  public void append(Object o){
    data[size] = o;
    size++;
    ...
  }
  ...
}


public class Linearlist {
  private Node head;
  private int size;
```

Iteration

- Want a loop to run through all values in a linear list
- If the list is an array with public access, we write this
- For a linked list with public access, we could write this
- We don't have public access . . .
- . . . and we don't know which implementation is in use!

```
int i;
for (i = 0; i < data.length; i++){
  ... // do something with data[i]
}
```

```
Node m;
for (m = head; m != null; m = m.next}
  ... // do something with m.data
}
```

Iterators

- Need the following abstraction

```
Start at the beginning of the list;
while (there is a next element){
  get the next element;
  do something with it
}
```

- Encapsulate this functionality in an interface called `Iterator`

```
public interface Iterator{
  public abstract boolean has_next();
  public abstract Object get_next();
}
```

Iterators

- How do we implement `Iterator` in `Linearlist`?
- Need a “pointer” to remember position of the iterator
- How do we handle nested loops?

```
for (i = 0; i < data.length; i++){
  for (j = 0; j < data.length; j++){
      ... // do something with data[i] and data[j]
  }
}
```

Iterators

- Solution: Create an `Iterator` object and export it!

```
public class Linearlist{

  private class Iter implements Iterator{
    private Node position;
    public Iter(){...}   // Constructor
    public boolean has_next(){...}
    public Object get_next(){...}
  }

 // Export a fresh iterator
  public Iterator get_iterator(){
    Iter it = new Iter();
    return(it);
  }
 }
```

- Definition of `Iter` depends on linear list

Iterators

- Now, we can traverse the list externally as follows:

```
Linearlist l = new Linearlist();
...
Object o;
Iterator i = l.get_iterator();

while (i.has_next()){
  o = i.get_next();
  ...   // do something with o
}
```

- The new Java `for` over lists implicitly constructs and uses an iterator

```
for (type x : a)
  do something with x;
}
```

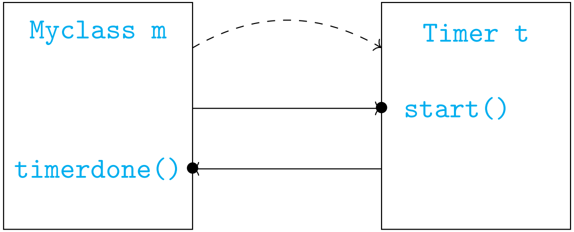
- For nested loops, acquire multiple iterators!

```
Linearlist l = new Linearlist();
...
Object oi,oj;
Iterator i,j;

i = l.get_iterator();
while (i.has_next()){
  oi = i.get_next();
  j = l.get_iterator();
  while (j.has_next()){
    oj = j.get_next();
    ... // do something with oi, oj
  }
}
...
```

Passing functions

- Recall callbacks
  - Myclass m creates a Timer t
  - t starts running in parallel
  - t notifies m when the time limit expires
- m needs to pass timerdone() to t
- Achieved this through an interface



```
public interface Timerowner{
  public abstract void timerdone();
}

public class Myclass
       extends Timerowner{
  ...
}
```

```
public class Timer implements Runnable{
  private Timerowner owner;
  ...
  public void start(){
    ...
    owner.timerdone();
  }
}
```

Passing functions

- Customize `Arrays.sort`
- `Comparator` interface provides signature for comparison function
  - `T` is type variable — structural polymorphism, Java generics, to be discussed later
- Implement `Comparator`
- Pass to `Arrays.sort`

```
public interface Comparator<T>{
  public abstract int compare(T o1, T o2);
}
```

```
public class StringCompare
  implements Comparator<String>{

  public int compare(String s1, String s2){
    return s1.length() - s2.length();
  }
}
```

```
String[] strarr = new ...;
StringCompare strcmp = new StringCompare();
Arrays.sort(strarr,strcmp);
```

Functional interfaces

- Interfaces that define a single function are called functional interfaces
  - `Comparator`, `Timerowner`
- How can we directly pass the required function?
- In Python, function names are similar to variable names
  - Define a function
  - Pass it as an argument to another function
  - `map` is a higher order function

```
public interface Comparator<T>{
  public abstract int compare(T o1, T o2);
}
```

```
public interface Timerowner{
  public abstract void timerdone();
}
```

```
def square(x):
  return(x*x)

l = list(map(square,range(100)))
```

Lambda expressions

- Lambda expressions denote anonymous functions
  - `(Parameters) -> Body`
  - Return value and type are implicit
- From λ-calculus (Alonzo Church)
  - Foundational model for computing, parallel to Alan Turing's machines
  - Basis for functional programming, will discuss in detail later
- Substitute wherever a functional interface is specified

```
(String s1, String s2) ->
   s1.length() - s2.length()
```

```
String[] strarr = new ...;
Arrays.sort(strarr,
            (String s1, String s2) ->
               s1.length() - s2.length());
```

Lambda expressions

- More complicated function body can be defined as a block
- Note that the function is anonymous only for the caller
- The function that receives the lambda expression still needs to use a functional interface for the parameter type

```
public static <T> void
   Arrays.sort(T[] a, Comparator<T> c)}
```

  - Inside `Arrays.sort()`, refer to the function as `c.compare()`, name defined in the `Comparator` interface

```
(String s1, String s2) -> {
     if s1.length() < s2.length()
       return -1;
     else  if s1.length() > s2.length()
       return 1;
     else
       return 0;
     }
```