COMP 2140 - Data Structures

Shahin Kamali

Topic 3 - Algorithm Analysis

University of Manitoba

Based on notes by S. Durocher.
Abstract Data Types

**Definition**

An abstract data type (ADT) consists of:

1. a collection of data items
2. a set of allowable operations on the data items

An ADT describes:

- **what** type of data are stored but **not how** they are stored (no detail about data structure), and
- **what** operations can be performed on the data but **not how** these operations are implemented (no detail about the algorithm).
ADT examples

Dictionary ADT:

- **Data:** a set of data items each having a unique key.
- **Operations:** `insert(key)`, `delete(key)`, `search(key)`

Priority Queue ADT

- **Data:** a set of data items each having a priority.
- **Operations:**
  - `insert(priority)`: insert a new item with a given priority
  - `isEmpty()`: is there any thing in the queue?
  - `extract_max()`: return the item with the highest priority and delete from the queue.
Object Oriented Design Principle

An ADT describes the functionality of a particular module within a larger project.

Encapsulation and Modularity

- A programmer only needs to understand the implementation details of the module on which he/she is working.

- For the remaining modules, he/she only needs to know the type of data input and output by a module and the result of running the relevant methods in the module.

The information about a module provided to an outside user is called the interface for the module.
Implementation of ADTs

Example

An abstract data type consists of:

1. a collection of data items, and
2. a set of allowable operations on the data items.

In Java, a class provides a natural way to implement an ADT.

Example

A Java class consists of:

1. a set of instance variables (a.k.a., data members or attributes), and
2. a set of methods (that may initialize, modify, or return information about the instance variables).
A First Class

Suppose you need to implement a database of songs. You must first specify what attributes of each song should be maintained. **Song ADT**

- **Data:**
  - title
  - singer
  - duration
  - number of times the song has been streamed (listened)

- **Operations:**
  - listen to the song
  - ask how many times the song is streamed
Defining the Class Song

The filename corresponds to the name of the class: Song.java
(By convention, class names begin with a capital letter.)

class Song {
    private String title;
    private String artist;  // instance variables
    private int timesListened;
    private int duration;

    public Song(String newTitle, String newArtist, int length) {
        title = newTitle;
        artist = newArtist;  // constructor
        duration = length;
        timesListened = 0;
    }

    public int getTimesListened() {
        return timesListened;  // accessor method
    }

    public void listen() {  // action method
        timesListened++;
    }
}

COMP 2140 - Data Structures
Using the Class Song

Song a, b;
a = new Song ("Help", "Beatles", 226);
b = new Song ("Opera Singer", "Cake", 312);
Song c = new Song ("Symphony #9", "Beethoven", 3749);
a.listen();
int d = a.getTimesListened();
Using the Class Song

Now you can specify what attributes of the database should be maintained and what operations you want to be able to perform on the database.

PlayList ADT

- **Data:**
  - list of songs
  - number of songs in the list
  - max number of songs that can be stored

- **Operations:**
  - getMostListenedSong (return the song listened most often)
  - addSong (add a new song to the list) getListSize (how many songs are in the list?)
  - mergeList (add another list to this one)
class Playlist {
    private Song[] data;
    private int numElements, maxSize;

    public Playlist () {
        maxSize = 5;
        data = new Song[maxSize];
        numElements = 0;
    }

    public Playlist (int size) {
        maxSize = size;
        data = new Song[maxSize];
        numElements = 0;
    }

    public Song getMostListened () {
        Song max = null;
        for (int i = 0 ; i < numElements ; i++)
            if (max == null || data[i].getTimesListened () > max.getTimesListened ())
                max = data[i];
        return max;
    }

    public int getListSize () { return numElements; }

    public void addSong (Song newSong) {
        if (numElements < maxSize) data[numElements++] = newSong;
    }

    public void mergeList (PlayList secondList) { ... }

    private void sortList () { ... }
}
Defining class Playlist

Song a = new Song ("Help", "Beatles", 226);
Song b = new Song ("Opera Singer", "Cake", 312);
Song c = new Song ("Symphony #9", "Beethoven", 3749);
Playlist stephsList = new Playlist(4);
Playlist andreasList = new Playlist();
stephsList.addSong(a);
stephsList.addSong(b);
andreasList.addSong(b);
andreasList.addSong(c);
a.listen();
b.listen();
a.listen();
Song d = stephsList.getMostListened();
Song e = andreasList.getMostListened();
Private vs. Public

- An outside user shouldn’t be burdened with implementation details. For example, when calling myPlayList.addSong(mySong); a user shouldn’t need to worry about the fact that mySong is stored in an array called data whose next available cell is indexed by numElements and that at most maxSize songs can be stored.

- Allowing outside users to directly access internal data can lead to errors. For example, say data is made public and is initialized to size 100. If an outside user attempts to access myPlayList.data[150] then an array indexing error will occur.

- By differentiating between public and private instance variables and methods, a programmer can ensure that such errors don’t occur.

- Access to private instance variables can be controlled through public methods.

- A private instance variable or method can only be accessed by code within the class. (See our previous examples.)
Constructor

- Within a class, a constructor is a special method called when a new instance of that class is allocated.
- The constructor method has the same name as the class.
- Unlike normal methods, a constructor does not specify any return type (it returns the same type as its Class).
- A constructor can have arguments.
- A class can have multiple constructors (or none), so long as no two constructors have the same signature (sequence of argument types).
- A constructor is typically used to initialize instance variables.
Accessor and Action Methods

An **accessor method** simply returns information about an instance of the class. The data stored in a class instance are not modified. For example: `getTimesListended` in class `Song` and `getMostListened` and `getListSize` in class `PlayList`.

An **action (or mutator) method** modifies data stored in a class instance. For example: `listen` in class `Song` and `addSong` and `mergeList` in class `PlayList`.
A .java file may define multiple classes.

At most one class in a .java file can have a main method.

The main method is the code that is executed when that class is interpreted.
Static Members

- Normally, each object has its own copy of all the members of the class.

- Sometimes, you may want to let all instances of the class share a data member.

- The static modifier in front of an instance variable or a method means that all instances of objects of that class have access to the same member, regardless of how many (could be zero) instances exist.
Suppose you need to write a class X whose role is very similar to an existing class Y. You could:

- Copy the code into a second class and add new code. Disadvantages:
  - Time consuming
  - Redundant
  - Any future modification to class X requires making the corresponding modification to class Y.

- Inherit the code from Y, add new methods and instance variables, and overwrite methods.
Example: Class SportsTeam

Example

SportsTeam

- **instance variables**: `teamName, homeTown, playerList, wins, losses, points`
- **methods**: `addPlayer, addWin, addLoss, getName`

Suppose you want to extend the class SportsTeam to make it specific to certain sports.
Extending Class SportsTeam

**Example**

**BaseballTeam**

- *instance variables*: same as SportsTeam + homeRuns
- *methods*: same as SportsTeam

**Example**

**HockeyTeam**

- *instance variables*: same as SportsTeam + ties
- *methods*: same as SportsTeam + addTie, and a different addWin
Extending Class SportsTeam

Example

ProHockeyTeam

- instance variables: same as HockeyTeam + salaries
- methods: same as HockeyTeam + sellTo
Class Inheritance Diagram

The classes define a hierarchical relationship.

- SportsTeam is the **base class**.
- SportsTeam is a **superclass** of HockeyTeam.
- HockeyTeam is a superclass of ProHockeyTeam.
- ProHockeyTeam is a **subclass** of HockeyTeam.
- ProHockeyTeam **inherits** from HockeyTeam, or also
  ProHockeyTeam **extends** HockeyTeam.
public class SportsTeam {

    protected String teamName, homeTown;
    protected String[] playerList;
    protected int wins, losses, numPlayers, points, maxPlayers;
    private static final int DEFAULT_SIZE = 10;

    public SportsTeam(String name, String town, int size) {
        teamName = name;
        homeTown = town;
        maxPlayers = size;
        numPlayers = 0;
        playerList = new String[maxPlayers];
        wins = 0; losses = 0; points = 0;
    }

    public SportsTeam(String name, String town) {
        this(name, town, DEFAULT_SIZE);
    }

    public String getName() { return teamName; }

    public boolean addPlayer(String name) {
        boolean arrayNotFull = (numPlayers < maxPlayers);
        if (arrayNotFull) playerList[numPlayers++] = name;
        return arrayNotFull;
    }

    public void addWin() { wins++; points++; }

    public void addLoss() { losses++; }
public class BaseballTeam extends SportsTeam {
    protected int homeRuns;

    public BaseballTeam(String name, String town) {
        super(name, town);
        homeRuns = 0;  // call constructor for the parent class SportsTeam
    }
}

- super executes the constructor of the superclass.
- If super is not included in a constructor, then the constructor of the superclass is automatically called without any arguments; this is equivalent to calling super().
- In our example, there is no constructor SportsTeam(), so we need to explicitly call super with the appropriate arguments.
HockeyTeam: Overriding Methods

```java
public class HockeyTeam extends SportsTeam {
    protected int ties;

    public HockeyTeam(String name, String town) {
        super(name, town);
        ties = 0;
    }

    public void addWin() { wins++; points += 2; }
    public void addTie() { ties++; points++; }
}
```

- The method `addWin` has been replaced.
- Say instead you wanted to refine method `addWin`:

```java
public void addWin() {
    super.addWin();
    points++;
}
```
Example

```java
HockeyTeam habs = new HockeyTeam("Canadiens", "Montreal");

habs.addTie();
habs.addWin();
habs.addLoss();
call addWin method defined in HockeyTeam, i.e., points += 2

SportsTeam leafs = new HockeyTeam("Maple Leafs", "Toronto");
leafs.addWin();
call addWin method defined in HockeyTeam, i.e., points += 2

leafs = new BaseballTeam("Blue Jays", "Toronto");
leafs.addWin();
call addWin method defined in SportsTeam ,i.e., points += 1

boolean test = (leafs instanceof BaseballTeam); True
test = (leafs instanceof HockeyTeam); False
test = (leafs instanceof SportsTeam); True

HockeyTeam nucks = new SportsTeam("Canucks", "Vancouver"); is it correct?
```

This last statement results in an error, because SportsTeam is not an extension of HockeyTeam
A set of .java files can be grouped together into a **package**.

Begin your file with the line:

```java
package myPackageName;
```

Grouping classes into packages provides access protection and namespace management.

- You can have two classes of the same name in two different packages (the name of the package is a part of the class names).

Similarly, to define a subpackage, begin your file with the line:

```java
package myPackageName.mySubpackageName;
```
Public, Private, Protected

Access to class members can be set to one of four levels:

<table>
<thead>
<tr>
<th>modifier</th>
<th>class</th>
<th>package</th>
<th>subclass</th>
<th>world</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>protected</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>no modifier</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>private</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

This table answers the following question:
If class A is a subclass of class B and c is an instance variable or method declared in B with the modifier private/protected/no modifier/public, then can code in class A access c?
Interfaces

- Once an abstract data type has been defined and is to be implemented in a class, each ADT operation is assigned a corresponding method.

- An **interface** specifies the name and signature of each method without specifying the method’s actual code.

- The interface allows a class’ users and implementers to work independently with a common definition for referring to the methods that implement the ADT.

- In principle, an interface is a description of the class for an outside user with all hairy things removed.
Interface Example

Our sports example could have used the following interface:

```java
public interface Team {

    public String getName();
    // return team name

    public boolean addPlayer(String name);
    // add player to team

    public void addWin();
    // increment number of wins

    public void addLoss();
    // increment number of losses

}
```
## Interface Example

The SportsTeam class is now implemented as follows:

```java
public class SportsTeam implements Team {
    protected String teamName, homeTown;
    protected String [] playerList;
    protected int wins, losses, numPlayers, points, maxPlayers;
    private static final int DEFAULT_SIZE = 10;

    public SportsTeam(String name, String town, int size) {
        teamName = name;
        homeTown = town;
        maxPlayers = size;
        numPlayers = 0;
        playerList = new String [maxPlayers];
        wins = 0; losses = 0; points = 0;
    }

    public SportsTeam(String name, String town) {
        this(name, town, DEFAULT_SIZE);
    }

    public String getName() { return teamName; }

    public boolean addPlayer(String name) {
        boolean arrayNotFull = (numPlayers < maxPlayers);
        if (arrayNotFull) playerList[numPlayers++] = name;
        return arrayNotFull;
    }

    public void addWin() { wins++; points++; }

    public void addLoss() { losses++; }
}
```
Interface

- A class that implements an interface must implement all methods declared in the interface.
  - In our example, the class SportsTeam must implement the methods getName, addPlayer, addWin, and addLoss.
  - A class may implement other methods in addition to those declared in the interface.

- An interface cannot be instantiated.

- All interface members must be public.
  - It is better not to have a variable (unless it is final).
  - An interface has no constructor.

- Multiple classes can implement the same interface.
  public class Undergraduate implements Student {...}
  public class Graduate implements Student {...}

- A class can implement multiple interfaces.
  public class Database implements Dictionary, Multiset {...}
Abstract Classes

- An **abstract class** may contain:
  - empty method declarations
  - concrete method definitions
  - instance variables

- An abstract class lies somewhere between an interface and a complete class.

- An abstract class cannot be instantiated.

- An abstract class can implement an interface. In this case, methods declared in the interface are either implemented or redeclared in the abstract class.

- An abstract class $X$ is implemented by another class $Y$ that extends $X$.

- An abstract class may contain both private and public members.
Interfaces vs Abstract Classes

- Interface provides **full abstraction** while an abstract class does not (an interface is sufficient for an outside user).
- We cannot declare a member field (private member) in an interface while we can in abstract class.
- Everything is ‘public’ in an interface, while you can have access modifiers in an abstract class.
- A class may implement multiple interfaces while it inherits at most one abstract class.
- An abstract class can provide complete default code or just the details that have to be overwritten. An interface cannot implement any code;
Strategy

• In your projects, start with an interface (or more) which describes the methods to an outside user.

• Implement the interface(s) with abstract class(es). Put all common codes for the default methods in the abstract class.

• Extend the abstract class(es) with further abstract or regular classes (inheritance hierarchy).