CMSC 132

Intro to Object Oriented Programming II

Ekesh Kumar
Prof. Nelson Padua-Perez • Fall 2019 • University of Maryland
https://www.cs.umd.edu/class/fall2019/cmsc132/

Last Revision: August 30, 2019

Contents

---

1 Tuesday, August 27, 2019

This class is CMSC132: Object Oriented Programming II. We will be covering modern software development techniques, various algorithms and data structures, and more. This course is taught in Java 11, but the techniques taught in this course carry over to several other programming languages.

Logistics

- All lectures are recorded and posted on Panopto.
- No collaboration on projects.
- No pop quizzes.
- We will be using the Eclipse IDE.
- Projects, office hours, lecture videos, and most other resources can be accessed from the class webpage: https://www.cs.umd.edu/class/fall2019/cmsc132/.
- Projects are due at 11:30 p.m. on the specified day. They can, however, be submitted up to 24 hours late with a 12% penalty.
- The grade breakdown is 26% for Projects, 16% for Quizzes, Exercises, and Lab Work, 8% for Midterm 1, 11% for Midterm 2, 11% for Midterm 3, and 28% for the Final Exam.

Abstraction and Encapsulation

There are important two techniques used in Object-Oriented Programming that we need to become familiar with: abstraction and encapsulation.

Abstraction is a technique in which we provide a very high-level model of activity or data. Small details about the model's functionality are not specified to the user. A good example . Abstraction can further be divided into two sub-categories, which are described below:

1. **Procedural abstraction** is a type of abstraction in which the user is aware of what actions are being performed, but they are not told how the action is performed. For example, suppose we would like to support a list of numbers. There are many algorithms that can do this for us. Under procedural abstraction, we would know that our end result is a sorted list of numbers, but we wouldn't know which algorithm is being used.
2. **Data abstraction** is a type of abstraction in which various data objects are known to the user, but how they are represented or implemented is not known to the user. An example of data abstraction is shown by representing a list of people. While the user would know that they have a list of people, they wouldn't know how the list is being represented (for example, we could use an array, an ArrayList, or any other data structure).

An **abstract data type** (ADT) is an entity that has values and operations. More formally, an abstract data type is an implementation of interfaces (a set of methods). Note that it is "abstract" because it does not provide any details surrounding how these various operations are implemented.

An example of an abstract data type is a queue, which supports the operation of inserting items to the end of the queue as well as the operation of retrieving items from the front of the queue. Note, again, that we are not concerned with how these operations should be performed internally.

Finally, **encapsulation** is a design technique that calls for hiding implementation details while providing an interface (a set of methods) for data access. A familiar example of encapsulation is shown through the ArrayList in Java. The ArrayList provides various methods that are accessible to us, such as the `.add()` and `.at()` methods. We aren't concerned with how they are implemented internally.

The Java Programming Language

Different programming languages provide varying levels of support for object-oriented programming. In particular, Java and C++ allow us to easily perform object-oriented programming. How does Java does this?

- Java provides us with **interfaces**, which allows us to specify a set of methods that another class must implement. This allows us to easily express an abstract data type since we can specify the operations and data that an entity must have. Interfaces follow an "is-a" relationship, meaning that a class that implements an interface is what the interface specifies. As an example, consider an animal interface. If an elephant implements the animal interface, we can perform tasks that are meant for animals with elephants (such as passing an elephant into a function that accepts animals).
- Java provides us with **classes**, which can be used as blueprints for other classes. Classes can extend other classes, which makes them **subclasses**. These subclasses inherit functions from the original class, and this also defines an "is-a" relationship.

Here are some key points that we should remember about interfaces:

- An interface cannot be instantiated. So, if we have an interface called `animal`, typing `Animal a = new Animal()` **would not compile**.
- An interface can contain many public members, including static final constants, abstract methods (which have no body), default methods (with a body), static methods, and static nested types.

Let's look at an example of an interface:

```
import java.util.*;

public interface Animal {

    public void feed(String food);
    public int getAge();
    public boolean manBestFriend();

    default void grow() {
        System.out.println("I grow");
    }
}
```

Here, we've created an interface named `Animal`, which other classes can implement. Any class that implements this class will need to also implement the functions `feed`, `getAge()` and `manBestFriend()` with the same return values and parameters specified. The `grow()` function will be associated with any class that implements this interface.

Now let's look at another class that implements the `Animal` interface:

```
package animalExample;

public class Dog implements Animal {
  private String name;
```

```
   private int age;

   public Dog(String name, int age) {
      this.name = name;
      this.age = age;

   }

   public void feed(String food) {
      System.out.println("Feeding " + name + " with " + food);
   }

   public int getAge() {
      return age;
   }

   public boolean manBestFriend() {
      return true;
   }

   public void bark() {
      System.out.println(name + " is barking");
   }
}
```

Here are the key things that we should note:

- The `Dog` class implements the `Animal` class. Thus, it is **required** to implement all of the methods specified by `Animal` (if we were to comment out even one function, the code wouldn't compile).
- The `Dog` class is allowed to add its own methods, like `bark()`. Likewise, the `Dog` class is allowed to create its own variables, like `name` and `age`.

Here's another class that implements the `Animal` interface:

```
package animalExample;

public class Piranha implements Animal {
   private int age;

   public Piranha(int age) {
      this.age = age;
   }

   public void feed(String food) {
      System.out.println("Piranha is eating " + food);
   }

   public int getAge() {
      return age;
   }

   public boolean manBestFriend() {
      return false;
   }

   public void grow() {
```

```
    System.out.println("I grow like a fish");
  }
}
```

Here are some more things that we should take careful note of:

- More than one class can implement an interface. In this example, both `Dog` and `Piranha` implement `Animal`.
- We can rewrite methods that are provided to us. More specifically, `Piranha` implements `Animal`, but it rewrites the `grow()` function that was originally provided to it.

Recall that interfaces define an "is-a" relationship. Since `Piranha` and `Dog` implement `Animal`, we can now treat them as an `Animal`. A good way to verify the "is-a" relationship is the `instanceof` keyword. In Java, `instanceof` allows us to check whether one object is an instance of a specified type (class, subclass, or interface).

To demonstrate, suppose we declare a piranha variable `p`. We could write a conditional such as `if (p instanceof Animal) { /* Code to be executed */}`, in order to make sure that `p` is an instance of `Animal`. Why would we ever need to do this? This is particularly useful when we're casting. If we attempt to cast an object to a subclass of which it is not an instance of, Java throws a **ClassCastException**. We can prevent this exception by putting the relevant code in a conditional checking the instance of what we're casting.

2 Wednesday, August 30, 2019

Collections

The **Java Collections Framework** is a set of methods and classes that provide support for **collections**, which are objects that group multiple elements into one unit. There is a collection interface that defines a set of methods that certain classes must have. It turns out that ArrayList is in the collections framework, and it implements the set of required methods (such as `.add()`, `.clear()`, `.contains()`, and more).

Consider the following code segment:

```
package miscellaneous;

import java.util.Collection;
import java.util.ArrayList;
import java.util.LinkedList;

public class CollectionExample {

   public static Collection<Integer> defineElements(boolean arrayFlag, int numberOfValues, int
       maxValue) {
      Collection<Integer> elements;

      if (arrayFlag) {
         elements = new ArrayList<Integer>();
      } else {
         elements = new LinkedList<Integer>();
      }

      for (int j = 0; j < numberOfValues; j++) {
         elements.add((int) (Math.random() * maxValue + 1));
      }

      return elements;
   }

   public static void displayValues(Collection<Integer> data) {
      if (data.isEmpty()) {
         System.out.println("Empty Collection");
      } else {
         for (Integer value : data)
            System.out.print(value + " ");
      }
   }

   public static void main(String[] args) {
      Collection<Integer> firstCollection = defineElements(true, 5, 10);

      System.out.println("First Collection:");
      displayValues(firstCollection);

      System.out.println("\nSecond Collection:");
      Collection<Integer> secondCollection = defineElements(false, 5, 10);
      displayValues(secondCollection);

      /* Example of methods defined by the Collection interface */
      Collection<Integer> union = new ArrayList<Integer>();
```

```
      /* Combining elements */
      union.addAll(firstCollection);
      union.addAll(secondCollection);
      System.out.println("\nUnion");
      displayValues(union);

      /* Checking if elements from one collection are present in antoher */
      if (union.containsAll(firstCollection))
         System.out.println("\nIncludes all elements of first collection");

      /* Removing elements in union present in second collection */
      union.removeAll(secondCollection);
      System.out.println("After removing");
      displayValues(union);

      /* Clearing the collection */
      union.clear();
      System.out.println();
      displayValues(union);
   }
}
```

This code segment is another demonstration of how we can utilize the fact that some objects are instances of other types. The `defineElements` takes in various parameters and returns either a LinkedList (another type list provided in the Collections Framework) or an ArrayList depending on the parameters provided. The reason why this is allowed is because of the "is-a" relationship between ArrayLists, LinkedLists, and Collections. More specifically, both ArrayLists and LinkedLists are collections. Consequently, since the return type of this function is a Collection, we are allowed to return any object that is an instance of Collection.

On the other hand, the `displayValues` method takes in a Collection type, and we're allowed to pass in any object that is an instance of Collection, as seen in the driver.

Collections are also really useful because they have built-in functions like `.shuffle()` (which scrambles the collection) and `.sort()`, which sorts the function.

Generics

We typically write, `ArrayList<Integer> = new ArrayList<Integer>()`, when we're instantiating an ArrayList of integers. Likewise, we could write, `ArrayList<Bananas> = new ArrayList<Bananas>()` if we had created a class called `Bananas`, or `ArrayList<String> = new ArrayList<String>()` if we wanted an ArrayList of strings. This is example of **generics**: we are allowed to pass in whatever type we want our ArrayList to store.

Why are generics useful? Firstly, generics help us reduce the amount of code we need to write. Instead of writing lots of code that gets executed depending on whether the user is using an integer or a string, we can use generics in order to reduce the amount of code needed. The `.add()`, `.remove()` and other methods used by ArrayList are valid no matter what type we are storing in our ArrayList. If we didn't have generics, we'd need to have.

Secondly, generics help us move some casting errors from runtime to compile time. Consider the following code segment:

```
class A { ... }
class B { ... }
List myL = new ArrayList(); // raw type
myL.add(new A()); // Add an object of type A
```

```
...
B b = (B) myL.get(0);
```

Here, we've declared two classes, and we've created a List called `myL`. In this example, we have not used any generics (this is called a **raw type**) since we haven't specified the type that this list will store. Thus, Java will permit us to add whatever we want to this list. This can lead to issues if we don't remember what type is stored at each index (in the above example, for instance, retrieved the first element by casting it as `B` when it is actually of type `A`.). This issue won't be seen until runtime. Using generics, however, would move this error to compile time.

Here is the same code segment but with generics:

```
class A { ... }
class B { ... }
List<A> myL = new ArrayList<A>();
myL.add(new A());
A a = myL.get(0);
...
B b = (B) myL.get(0); // Compile-time error
```

Now that we've used generics, we'll find the casting error during compile time.

So far, we've only seen how collections use generics. How do we use our own generics? We can parametrize classes, interfaces, and methods using `<X>` notation. For example, we could write, `public class foo<X, Y, Z> { ... }`, to define a class called `foo` that takes in three types.

The Iterable Interface

The **Iterable Interface** defines a method called `Iterator`, which returns an iterator that allows us to traverse a container (particularly lists). ArrayLists in Java implement the iterable interface.

The following example demonstrates how we can use iterators:

```
public static void main(String[] args) {
    ArrayList<String> myList = new ArrayList<String>();
    myList.add("a");
    myList.add("b");

    Iterator<String> it = myList.iterator();

    while (it.hasNext()) {
        System.out.println(it.next());
    }
}
```

This code segment will print out everything in our list `myList`.

However, it turns out that we don't actually need to handle iterators ourselves. We can use the **enhanced for loop** (also known as a **for each loo**), which does this for us. These types of for loops handle the iterator automatically.

Here's how we can use an enhanced for loop with our previous example:

```
public static void main(String[] args) {
    ArrayList<String> myList = new ArrayList<String>();
    myList.add("a");
    myList.add("b");
```

```
    for (String s : myList) {
        System.out.println(s);
    }
}
```

The for loop in this code segment can be read as, "for each string `s` in `myList`, print `s`."