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 28, 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;
```

```java
   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:

```java
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`.