CMPINF0401 Recitation

TUESDAYS 11:00-12:50

MICHAEL BARTLETT

Overview

- ArrayLists
- Recursion
- OOP

ArrayLists

- All operations on ArrayLists are specified as method calls
  - Add to the end of an ArrayList by using arrayListName.add(object)
  - Remove from an ArrayList:
    - arrayListName.remove(object)
    - arrayListName.remove(index)
  - Get the number of elements by using arrayListName.size()
  - Get a value from an index using arrayListName.get(index)
  - Get the index of an object by using arrayListName.indexOf(object)
  - Set a value from an index using arrayListName.set(index, value)

ArrayLists

- ArrayLists are initialized using the following syntax:
  - ArrayList<Type> varName = new ArrayList<Type>();
- Let's break this down:
  - ArrayList<Type>: Need to make sure your Type is Capitalized.
    - i.e., ArrayList<int> is not valid → Needs to be ArrayList<Integer>
    - This is because you can make an ArrayList of any Generic Type. Generics haven't been covered yet, but you'll see this later. As a result, by saying ArrayList<Integer> java converts an int to it's "generic" counterpart. **All values are stored as objects.**
  - new ArrayList<Type>();
    - Because everything with ArrayLists is done using method calls, the () calls the method to create a new ArrayList of the Type you're passing. **An ArrayList is an object.**

ArrayLists vs. Arrays

- ArrayLists are built off off arrays. Unlike arrays, you do not have to worry about size constraints since they automatically take care of sizing.
  - As a result, in terms of runtime, it's better to use an ArrayList for a dynamic set of data
  - All operations are done using method calls in an ArrayList

Recursion: an overview

- Recursion is the process of making a method call itself
- This provides a way to break larger problems down into simple subproblems.
- When writing a recursive method, you must ensure that you have:
  - A base case, otherwise known as a halting case, that is attainable
  - A recursive case in which the method calls itself

Recursion Assignment4 Example

- Assignment4.java

Why recursion?

- Despite it taking up more memory than iterative methods (i.e., ones that contain loops), recursion can reduce the amount of time that it takes to do certain problems
  - A recursive function calls itself, so the memory for a called function is **allocated on top of the memory allocated for calling the function**. When the base case is reached, the function returns its value to the function that it was called from, and its memory is de-allocated.
- At its core, recursion is essentially dividing one large problem into several smaller subproblems until they are manageable.

OOP: What's an object?

- Technically speaking, an object is a bundle of state and behavior:
  - State: The data contained in the object (the object's fields)
  - Behavior: The actions supported by the object (its methods)

OOP: What's a Class?

- Every object has a class
  - A class defines the object's methods and fields
- A class defines both type and implementation
  - Type → Where the object can be used (What datatypes is the constructor going to accept?)
  - Implementation → How the object does things (its methods)

OOP: Class vs. Objects Example

- Example of a class:
  - Fruit
  - Car
- Example of corresponding objects:
  - Apple, Banana, Mango
  - Volvo, Audi, Toyota
- So, a class is a template for objects, and an object is an instance of a class

OOP: Class vs. Objects Example

- When we create individual objects, they inherit all the variables and methods from the class
  - We've seen this with Strings:
    - String name = "Michael"
      - name is a String Object where String is a class
    - name = name.toUpperCase();
      - Now, name == "MICHAEL"
      - name was able to use the predefined method for the String object

OOP: Interfaces vs Classes

- Interfaces can be used to define the methods that the class must contain
  - For example, an interface might define a Car like this:
    - interface Car { public String getColor();}
  - The class for a type of Car could look like this:
    - class Volkswagen implements Car

```
{
    String color = "blue"; // Normally done with an init method
    public String getColor()
    {
            return(this.color);
    }
}
```

How do we make objects?

- We saw the previous String example, but how do we define our own and use them?

```
public class Object {
    int x = 5;
    public static void main(String[] args) {
        Object myObj1 = new Object(); // Object 1
        Object myObj2 = new Object(); // Object 2
        System.out.println(myObj1.x); // Prints 5
        System.out.println(myObj2.x); // Prints 5
        myObj2.x = 7;                 // We can change values
        System.out.println(myObj2.x); // Prints 7
    }
}
```

Classes can have methods

- As shown before, we know that Classes can have methods. There are two types, static and public.
- Static can be accessed without creating an object of the class, public needs an instance of the class to be created.

```
public class Object {
    // Static method
    static void myStaticMethod() {
      System.out.println("Static methods can be called without creating objects");
    }

    // Public method
    public void myPublicMethod() {
      System.out.println("Public methods must be called by creating objects");
    }

    // Main method
    public static void main(String[] args) {
      myStaticMethod(); // Call the static method
      // myPublicMethod(); This would compile an error

      Object myObj = new Object(); // Create an object of Main
      myObj.myPublicMethod(); // Call the public method on the object
    }
  }
```

Constructors

- A constructor is a special method that is used to initialize objects.
- The constructor is called when an object of a class is created.
- It can be used to set initial values for object attributes

Constructors Example

```
public class Car {
    int modelYear;
    String modelName;
    int odometer;

    public Car(int modelYear, String modelName) { // Constructor has the same name as the class name
      this.modelYear = modelYear; // conventionally, name the variables the same and then use "this" to specify the variable you're setting.
      this.modelName = modelName;
      this.odometer = 0;
    }

    public Car(int modelYear, String modelName, int odometer) {  // You can have multiple constructors, depending on the amount of args passed in different ones will run
        this.modelYear = modelYear;
        this.modelName = modelName;
        this.odometer = odometer;
    }


    public static void main(String[] args) {
      Car myCar = new Car(1969, "Mustang");
      System.out.println(myCar.modelYear + " " + myCar.modelName + " with " myCar.odometer " miles.");
    }
  }

  // Outputs 1969 Mustang with 0 miles
```

Encapsulation

- Encapsulation: make sure that “sensitive” data is hidden from users. To do this:
  - Declare class variables/attributes as private
  - Provide public get and set methods to access and update the value of a private variable.

```
public class Person {
    private String name; // private = restricted access

    // Getter
    public String getName() {
      return name;
    }

    // Setter
    public void setName(String newName) {
      this.name = newName;
    }
  }

public static void main(String[] args) {
    Person myObj = new Person();
    myObj.setName("Michael"); // Set the value of the name variable to "Michael"
    System.out.println(myObj.getName());
  }
}

// Outputs "Michael"
```

Why Encapsulation?

- Better control of class attributes and methods
- Class attributes can be read-only (if they can only be accessed through a get method), or write-only (if they can only be accessed through a set method).
- Flexible: The programmer can change one part of the code without affecting others
- Increased Security: hide data that shouldn't be available to all users, test to see if the value being set is valid (i.e., a Person's weight shouldn't be 0)

Inheritance

- An important feature of OOP, it allows for one class (child class) to inherit the fields and methods of another class (parent class).
- When defining a child class, we use the keyword *extends* to inherit from a parent class

```
// Parent Class
class Animal {
  // Animal class members
}

// Child Class
class Dog extends Animal {
  // Dog inherits traits from Animal

  // additional Dog class members
}
```

Super()

- A child class inherits its parent's fields and methods, meaning it also inherits the parent's constructor.
  - Sometimes we may want to modify the constructor, in which case we can use the super() method, which acts like the parent constructor inside the child class constructor.
- Alternatively, we can also completely override a parent class constructor by writing a new constructor for the child class.

```
// Parent class
class Animal {
  String sound;
  Animal(String snd) {
    this.sound = snd;
  }
}

// Child class
class Dog extends Animal {
  // super() method can act like the parent
constructor inside the child class constructor.
  Dog() {
    super("woof");
  }
  // alternatively, we can override the constructor
completely by defining a new constructor.
  Dog() {
    this.sound = "woof";
  }
}
```

Protected & Final

- When creating classes, sometimes we may want to control child class access to parent class members.
  - We can use the protected and final keywords to do just that.
- protected keeps a parent class member accessible to its child classes, to files within its own package, and by subclasses of this class in another package.
- Adding final before a parent class method's access modifier makes it so that any child classes cannot modify that method - it is immutable.

```
class Student {
  protected double gpa;
  // any child class of Student can access gpa

  final protected boolean isStudent() {
    return true;
  }
  // any child class of Student cannot modify
isStudent()
}
```

Polymorphism

- Java incorporates the object-oriented programming principle of *polymorphism*.
- Polymorphism allows a child class to share the information and behavior of its parent class while also incorporating its own functionality. This allows for the simplified syntax.

```java
// Parent class
class Animal {
  public void greeting() {
    System.out.println("The animal greets you.");
  }
}

// Child class
class Cat extends Animal {
  public void greeting() {
    System.out.println("The cat meows.");
  }
}

class MainClass {
  public static void main(String[] args) {
    Animal animal1 = new Animal();  // Animal
object
    Animal cat1 = new Cat();  // Cat object
    animal1.greeting(); // prints "The animal
greets you."
    cat1.greeting(); // prints "The cat meows."
  }
}
```

Method Overriding

- Wcan easily *override* parent class methods in a child class. Overriding a method is useful when we want our child class method to have the same name as a parent class method but behave a bit differently.
- In order to override a parent class method in a child class, we need to make sure that the child class method has the following in common with its parent class method:
  - Method name
  - Return type
  - Number and type of parameters

```
// Parent class
class Animal {
  public void eating() {
    System.out.println("The animal is eating.");
  }
}

// Child class
class Dog extends Animal {
  // Dog's eating method overrides Animal's eating
method

  public void eating() {
    System.out.println("The dog is eating.");
  }
}
```

Child Classes in Arrays & Arraylists

- Polymorphism allows us to put instances of different classes that share a parent class together in an array or ArrayList.
- For example, if we have an Animal parent class with child classes Cat, Dog, and Pig we can set up an array with instances of each animal and then iterate through the list of animals to perform the same action on each.

```
// Animal parent class with child classes Cat, Dog,
and Pig.
Animal cat1, dog1, pig1;

cat1 = new Cat();
dog1 = new Dog();
pig1 = new Pig();

// Set up an array with instances of each animal
Animal[] animals = {cat1, dog1, pig1};

// Iterate through the list of animals and perform
the same action with each
for (Animal animal : animals) {

  animal.sound();

}
```