Java Classes Reference

Yuvraj Lakhotia

Overview

Object-oriented programming allows us to create real-life parallelisms with the approach of *"programming through modelling."*

The main method is our *"driver class"* so we don't want to write all of our code inside it - this would make it unnecessarily clogged. The main method is used for instantiating the classes.

A **Java class** allows users to outline the *basic properties (variables)* and *behaviors (methods)* of objects, which interact in our program to produce desired outputs. A class is defined like this:

```
public class Board{
}
```

Classes are organized into **packages**, which need to be imported to allow us to use its classes in our program. Packages are used to group related classes, like a folder in a file directory.

```
import java.util.Scanner;        // import a single class from a package
import java.util.*;              // import all classes from package
```

I'll be using a board game as an example to guide you through the parts of a Java class.

Properties

As previously mentioned, properties are represented as **member variables** in the class. It is the programmer's responsibility to decide which data type would best represent a property. These can be accessed anywhere within the class (at a minimum). Coding standards place them in the top half of the class definition.

```
public class Dice{
        public int sides;             // number of sides on the dice
        public String color;          // color of the dice
        public Game game;             // game (other class) which the dice is from

        // methods go here
}
```

Behaviors

Methods, or functions, are used to model an object's behaviors. Parameters can be passed to a method which allow the function to "do something" with that data. Parameters can not be accessed outside the method, unlike class properties.

Return Types

Methods can return different types of data depending on the **return type** specified in the declaration.

Return types include:

- `void`
- `int`

- `String`
- any class/data type

```
public class Dice{
        public int sides;
        public String color;
        public Game game;

        // prints the phrase passed in the string
        public void say(String phrase){
                System.out.println(phrase);
        }

        // returns a number rolled on the dice from 1...number of sides
        public int roll(){
                Random rand = new Random()
                int number = rand.nextInt(sides) + 1;
                return number;
        }
}
```

Overloading

The **method signature** consists of the method name followed by the types of its parameters. We can **overload** methods with the same name by creating a new definition with a *different signature*. By changing the type and order of parameters, we can define another method that can behave differently from the others. Note: the return type is **not** part of the method signature.

```
public int combine(int a, int b){
        return a + b;
}
public String combine(String a, String b){
        return a + b;
}
public double combine(double a, double b){
        return a + b;
}
```

Comparing Objects

If we try to compare two objects using `==`, it will likely return `false` because it is comparing the *memory addresses* of the two objects. We can create a custom `equals()` method to check for equality between two objects by comparing fields.

```
public class Dice{
        int sides;
        String color;
        Game game;

        public boolean equals(Dice dice){
                return (this.sides == dice.sides
                && this.color == dice.color
                && this.game.equals(dice.game));
        }
```

Notice how we used `this` to differentiate between the current object and the object passed in the function. In methods with parameters named the same as object properties, we can use `this` to also differentiate between

the local variable (the parameter) and the object property. If we omit the `this`, local variables always take preference.

Access Levels

In the real world and practical applications, it's not safe or sensible to make everything `public` because that creates a huge security risk in your software. Java, like many other languages, allows us to use `public`, `protected`, and `private` **access modifiers** to **encapsulate** our data and limit visibility between classes.

- `public`
  - mainly applies to methods
  - least secure
- `protected`
  - case-by-case basis
  - programmer judgement
- `private`
  - mainly applies to properties
  - most secure
- Classes inside a package can see each other

Visibility:

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

Dot Operator

The **dot operator** (`.`) allows us to access/modify public variables and methods of a class/object.

```
public class BoardGame(){
        public static void main(String[] args){
                Dice dice = new Dice();
                dice.sides = 6;
                int count = dice.roll();
        }
}
```

Getter/Setter Methods

With the added security and encapsulation that access modifiers give us, non-public fields will become inaccessible to other classes. To "get" and "set" the values of these fields, we will use simple "getter" and "setter" methods. We can also added conditionals and other code inside these to make sure the proper data is being passed and retrieved.

A getter method just returns the field we want:

```
public int getSide(){
        return this.sides;
}
```

A setter method allows us to pass a value and set the field to it:

```
public void setSides(int sides){
        if (sides > 0 && sides <= 12)
                this.sides = sides;
}
```

Static Variables & Methods

We've already used the `Math` class, and likely noticed that we don't need to instantiate a `Math` object to access its fields and methods, we can do it by *directly referencing the class* with `Math.<field>` or `Math.<method>()`. These are known as **static variables** and **static methods.**

To declare a static variable or method, use the keyword `static` after its access modifier. Make sure to initialize them as they're declared, or repeated constructor calls could "initialize" them many times. Static methods or variables can't contain references to instance variables or methods.

Static constant fields can be declared with the keyword `final` . Naming conventions state to capitalize all letters and separate words by underscores. This is seen in mathematical constants such as `Math.PI`.

```
public class Dice{
        public static final MAX_NUM_DICE = 100;
        private static int numDice = 0;

        public static int getNumDice(){
                return Dice.numDice;
        }
}
```

Constructors

Constructors are "blueprints" called during instantiation to create an object of the class. They are *neither methods nor variables*.

The **default constructor** is internally defined for all classes, but it only gives us the space in memory with no additional steps done. If we want to assign values to some of the fields in the same line as instantiation, we can define and call additional constructors.

Remember that when defining new constructors, you *must also type out the default constructor* in order for the program to work. This default constructor can be left blank, however.

In the example below, we use the `this()` constructor, which allows us to call other constructors in the class.

```
public class Dice{
        private static int numDice = 0;
        private int sides;
        private String color;
        private Game game;

        public Dice(){
                ++Dice.numDice;
        }
        public Dice(int sides){
                this();
                this.sides = sides;
        }
        public Dice(int sides, String color){
                this(sides);
                this.color = color;
        }
```

```
        public Dice(int sides, String color, Game game){
                this(sides, color);
                this.game = game;
        }
}
```

Instantiation

To create an instance of a class, use the keyword `new` and assign its value to a variable. This stores the reference to the new object in the variable.

```
public class BoardGame{
        public static void main(String[] args){
                Game g1 = new Game();
                Dice d1 = new Dice();
                Dice d2 = new Dice(3);
                Dice d3 = new Dice(5, "black");
                Dice d4 = new Dice(12, "green", g1);
        }
}
```

Memory Allocation

The two main types of memory, *stack* (last-in-first out) and *heap* (dynamic allocation) have different uses.

Heap

The **heap** stores:

- objects
- JRE (Java Runtime Environment) classes

Stack

The **stack** stores:

- local variables
- primitive data types
- references to locations in heap

Object References

You probably noticed how printing a String reference (the variable) prints the actual value of the String instead of the memory address. This is just one of the specialties of the `String` class.

It's possible to instantiate strings with `new`, but this is memory-inefficient. This creates two different references to point to two different spaces in the heap.

```
// bad
String a = new String("test");       // different location
String b = new String("test");       // different location
```

The better way is to use the assignment operator (`=`). Java is intelligent - it recognizes identical strings and makes both references point to the same location in the heap. Altering the value of a string using `=` will create a new reference to a new memory location.

```
// good
String a = "test"                    // same location
String b = "test"                    // same location
String a = "chest"                   // new location
```

Java also has a useful builtin feature known as **garbage collection** which will free up memory spaces in the heap that no longer have references pointing to them.