Classes and Objects

Pieter Joubert

October 5, 2023

Classes and Objects

Table of Contents

Section 1

Classes and Objects

Classes created to run applications

- In object oriented programming we often create class that are used to run applications.
- One common example of this is the "main" class we created in every Java program.
- Giving this class the same name as file is in lets the Java compiler know this is the "main" class in the program and should be executed first.

Everything is an object

- In object oriented programming everything is a object.
- Every object is a member of a class.
- This means that all objects have an *is-a* relationship with the class it is instantiated from.
- For example: *Car* is a *Vehicle*.
- In this way Classes provide *reusability* - objects get their attributes from their classes.

Section 2

Creating a Class

Defining a class

- A class definition consists of:
  - An optional access specifier.
  - The keyword *class*
  - Any legal identifier (the class name)

```
01 |  public class Vehicle {}
02 |
03 |  class Weapon {}
04 |
05 |  public class Student2 {}
```

- A public class is accessible by all objects and can be extended as a basis for another class.

Instance variables

- Any Class contains variables or data fields.
- These are variables that store information about that class.
- For example a *Vehicle* might have a *topSpeed* or *numWheels* variable.
- When we instantiate a new object, each object get its own copy of all the fields in the class.

```
01 |  public class Weapon {
02 |      String name;
03 |      int damage;
04 |      double attackSpeed;
```

- No other classes can access these variables.
- This is known as information hiding.

Section 3

Instance Methods

Parts of a Class

- Classes contain fields (as we have seen) and methods.
- The different types of methods that a class can have are:
  - Mutator methods: set or change field values (known as *setter* methods)
  - Accessor methods: retrieve values (known as *getter* methods)
  - Utility methods: perform related actions without changing values.

```
01 |     public void setDamage(int dam) {
02 |         damage = dam;
03 |     }
04 |
05 |     public int getDamage() {
06 |         return damage;
07 |     }
08 |
09 |     public double calcDPS () {
10 |         return damage / attackSpeed;
11 |     }
```

Nonstatic vs. nonstatic methods

- Nonstatic methods are used with object instantiations.
- Called instance methods.
- You can use both a *static* or nonstatic methods in class.

Section 4

Declaring and Using Objects

Instatiating a new Object

- To create an object that is an instance of a class:
  - Supply a type and identifier.
  - Allocate computer memory for the object using the *new* keyword.
  - This creates a reference to an object.

```
01 |        public static void main (String args[]) {
02 |             Weapon gun = new Weapon();
03 |             gun.setDamage(100);
04 |        }
```

Data hiding

- Data hiding (or encapsulation) is a Object Oriented Programming design pillar.
- Data fields should be private and inaccessible from a client application.
- Fields should only be accessed through *getter* and *setter* methods.
- Clients should not be able to alter assigned values or set values directly using calculations.

Section 5

Classes as Data Types

Abstract Data Type

- This is a class we create. (In the same way the *String* class was created)
- Implementation is hidden and accessed only through public methods.
- This is a programmer-defined type and is not built into the language.
- Defining a Data Type we need to consider:
  - What shall we call it?
  - What are its attributes?
  - What methods are needed?

Section 6

Constructors

Creating a Constructor

- Constructs an object.
- Java automatically creates a default constructor with no arguments for us. We use it by calling the *new* keyword.
- A constructor must have the same name as the class it constructs.
- Cannot have a return type.
- Normally declared as *public*
- Generally we create constructors that can accept parameters.
- These parameters define the starting values for our data fields.
- If you define a constructor this overloads the default.
- You can define multiple constructors so long as their method signature is different.

Constructor Example

```
01 |       public Weapon(String nam, int dam, double speed)
       {
02 |           name = nam;
03 |           damage = dam;
04 |           attackSpeed = speed;
05 |       }
```

Section 7

The *this* reference

Repetion of variable names

- Very often we have numerous variables that refer to the same thing in a class.
- For example we have the *name* data field, but we also have the *nam* vriable that we accept as a parameter in the Constructor.
- We need to make sure the variable identifier is different to prevent errors.
- This can lead to ugly code, like in our previous example with *name* and *nam*.
- We can use *this* reference to indicate we are referring to the data field in a class.

Example of *this* reference

```
01 |       public Weapon(String name, int damage, double
       attackSpeed) {
02 |           this.name = name;
03 |           this.damage = damage;
04 |           this.attackSpeed = attackSpeed;
05 |       }
```

Section 8

Using Static Fields

Static fields

- Nonstatic fields are copied for each instance of the class.
- If you create 50 instances you get 50 different variables.
- Static fields are shared between all instances.
- If you create 50 instances you get one variable shared by all the instances.

Section 9

Using existing classes

Java Terminology

- We do not need to re-invent the wheel when writing our Java code. Many of the problems we need to solve have already been solved.
- These solutions are available in classes other programmers have already written.
- These classes are bundled into packages.
- The *java.lang* class is implicitly imported into every program you write. It contains fundamental classes that are very useful.

The *math* class

- The *java.lang.Math* class contains various constants and methods to preform mathematical functions.
- As it is a part of *java.lang* we do not need to import it.
- We can access these functions and methods by using the *Math* class, e.g.*Math.PI* or *Math.Pow*()

Importing classes

- To use a package that is not a part of *java.lang* we use the *import* statement
- We generally do this at the top of any class in which we want to use this package.
- For example the *LocalDate* package has methods related to working with the current date and time.

```
01 |  import java.time.*;
02 |  public class localdate {
03 |
04 |      public static void main(String args[]) {
05 |          LocalDate today = LocalDate.now();
06 |          System.out.println("Today is: " + today);
07 |      }
08 |
09 |  }
```

Section 10

Composition and Nested Classes

Composition

- Composition is when one class is a data field of another class.
- This object needs to have its values set up like any other class.
- In this case there is a *has-a* relation ship between the two classes.

```
01 |  public class Inventory {
02 |      Weapon leftHand;
03 |      Weapon rightHand;
04 |  }
```

Nested Classes

- A class containing another class.
- *static* member classes have access to all static methods of the top-level class.
- Non static methods of the inner class require an instance and have access to all data and methods of the top-level class.
- Local classes are local to a block.

Section 11

Lecture summary

Lecture summary

- Classes and Objects
- Creating a class
- Instance Methods
- Declaring and Using Objects
- Classes as Data Types
- Constructors
- The *this* reference
- Using Static Fields
- Using existing classes
- Composition and Nested Classes

Questions

Thank you! Questions?