Programming Fundamentals

Mikael Svahnberg*

2023-08-21

Contents

*Mikael.Svahnberg@bth.se

1 Introduction

- Barnes & Kölling Chapter 1, Objects and Classes
- Barnes & Kölling Chapter 2, Understanding Class Definitions
- Objects, Classes, Methods, and Attributes
- Built in data types
- Fundamental Input and Output
- Conditional Execution: `if`
- Design Philosophy: *Program objects represent real world entities*
- Design Principle: *Low Coupling*

2 Objects and Classes

- Object
  - Representation of *a single entity*
  - Representation of *a single real world entity*
  - Representation of *a single real world entity with more than one data attribute*
  - Representation of *a single real world entity with behaviour and one or more data attributes*
- Class
  - Description of all objects of a particular kind
  - Description of attributes and methods common to one or more objects
  - Convenient module of related functionality (e.g. `java.lang.Math`)

3 Real World Entity

- A red car
- A shopping basket
- A translator
- Two french hens
- A partridge in a pear tree

```
Car c = new Car("XYZ012", Colour.RED);
Basket b = new ShoppingBasket();
Translator t = new LanguageTranslatorFactory().createTranslator("German", "English");

ChickenCoop cc = new ChickenCoop();
cc.add(new FrenchHen());
cc.add(new FrenchHen());

Tree theTree = new PearTree();
Partridge p = new Partridge();
theTree.add(p);
```

Note:

- We use *Classes* to tell the compiler what *type of object* we wish to create
- Each object may exist in the real world (e.g. the red car) *and* may have a representation in our program.
  - Sometimes, we create program objects that do not have a real world representation. This is also ok.
- By convention, class names are `Capitalised`, and objects are not.

4 Methods

- Objects have *Methods* (functions) that can be invoked *on that object*
- Like a mini-program consisting of the data in one object and all the methods.
- Methods have a `return value`, a `name`, and zero or more `parameters`.

```
public class DemoClass {

  public int demoMethod(int oneParameter, String otherParameter) {
    System.out.println(otherParameter);
    return 10+oneParameter;
  }

  public static void main(String [] args) {
```

```
    DemoClass dc = new DemoClass();

    int returnValue = dc.demoMethod(10, "Hello?");
    System.out.println(returnValue);
  }
}
```

4.1 Methods are called on Objects (for now)

```
DemoClass dd = new DemoClass();
dd.demoMethod(10, "This should work");

DemoClass.demoMethod(10, "This will not work");
```

5 One Class, many Objects

```
public class Car {
  private String myPlate;
  private String myColour;

  Car(String theLicensePlate, String theColour) {
    myPlate = theLicensePlate;
    myColour = theColour;
  }

  public String toString() {
    return "a " + myColour + " car with license plate " + myPlate;
  }
}

Car c1 = new Car("aaa111", "red");
Car c2 = new Car("bbb222", "yellow");
System.out.println(c1);
System.out.println(c2);
```

5.1 Objects have a state

- The *state* of an object is the collective value of all attributes
- E.g. a car with `myPlate=="aaa111"` and `myColour=="Red"`
- The state may change if a method changes the value of any attribute.
- Above, `c1` and `c2` have different states.

(The state of an application is the collective value of all attributes in all objects.) (The state of an application *may* mean that it has a specific behaviour and only a specific set of operations are available.)

6 Return values

- Methods can return a value
  - a built in data type
  - an object
  - `void` (nothing is returned)

```
public void noReturn() {
  System.out.println("no return");
}

public int someReturn() {
  System.out.println("an integer is returned");
  return 1;
}

public Car returnsAnObject() {
  return new Car("ccc333", "Green");
}
```

7 Built in Data Types

```
byte smallNumber = 127;
short largerNumber = 32767;
int normalNumber = 100000;
float smallDecimal = 0.123456f;
double largeDecimal = 0.12456789;

boolean trueOrFalse = false;
char singleCharacter = 'A';
String someText = "Longer, but not too long Text";
```

- Aside from arithemtic operations (`+-*/%=`), these usually do not have any other methods.
- Note that `String` is in fact a class.
- Java also have classes to represent the built in data types as objecs, with many useful convenience methods.
  - e.g. `Integer.parseInt("123")` (Note how this calls a method on the *class* without creating an object. We'll come back to this later)

8 The Details of Defining a Class

Classes have:

- Name, e.g. *Ticket*

- “Accessibility statement” in the package where it is declared: `public` or `private`

  Public Available anywhere

  Private Only available from inside the package

- Any number of *Fields*, or Attributes using the built in data types or object references.

- Any number of *Constructors* with the same name as the class.

```
public class Ticket {
  private int aField;
  public Car anotherField;
  protected String aThirdField; // We will get back to what "protected" means.

  private String startStation;
  private String endStation;
  private String customerId;

  Ticket() {  // Default constructor, used when you create objects without any parameters.
    startStation = "Karlskrona";
    endStation = "anywhereElse";
    customerId = "";
  }

  Ticket(String theEndStation) {
    this();
    endStation = theEndStation;
  }

  public static void main(String[] args) {    } // Should not be needed, using this just t
}
```

8.1 Chaining Constructors

- The example above is usually not the best way to chain constructors.

- More often you want one “complete” constructor and chain yourself to that instead.

- Example:

```
Ticket(start, end, customer) {
  this.startStation = start;
  this.endStation = end;
  this.customerID = customer;
}

Ticket(start, end) {
  this(start, end, "DefaultCustomerID");
}
```

```
Ticket(end) {
  this("Karlskrona", end);
}

Ticket() {
  this("anywhereElse");
}
```

- You can chain to *one* other constructor.
- Has to be the *first* call.
- `this` always refers to the current object.

9 Method Parameters

- Methods may have "any" number of *parameters*

```
public int CalculateTicketCost(int discount) {
  /* Do some magic
     depending on what
     the start and end station
     is, as well as the customer id.
  */
  int basePrice = PriceCalculator.magicCalculation(startStation, endStation, customerId);
  this.myTicketPrice = basePrice*discount; // Assign the result to an attribute of the obj

  return myTicketPrice;
}

public void addNumbers(int first, int second, int third, int fourth, int fifth, int sixth,
  return first+second+third+fourth+fifth+sixth+seventh;
}
```

10 Getters and Setters

- Most attributes should be *private*
- Design Principle: *Low Coupling*
  - Private attributes means that no-one *except the object itself* can access the value
  - Conceptually, only the object itself knows that there even *is* an attribute with that name
  - ... or what type it has.
  - The object has full control over any calculations or side effects
- Accessing an attribute is done via public methods on the object
- In their most simple form, they are called *getters* and *setters*
  - Also called *accessors* and *mutators*

10.1 Example of get/set

```
private Frobnicator myFrob;

public void setFrobnicator(Frobnicator theFrobnicator) {
  myFrob = theFrobnicator;
}

public Frobnicator getFrobnicator() {
  returb myFrob;
}
```

- Yes, it is work to write this. Painful even.
- *That is precisely the point!* Objects should not expose details unless they absolutely *must*.
- A warning:
  - Returning a variable that is a built-in datatype returns a *copy*
  - Returning a variable that is an object reference returns a *copy of the reference* but points to the same object.
  - With the innocent statement `return myFrob` this object just lost control over its private data.
- . . . so, when are mutators ok? How can we make a safe accessor?

11 Parameters, Attributes, Local Variables

- Attributes are defined in the class
- Attributes have one value for each object
- e.g. `Car.myColour` ; each object of the type Car has its own value:
  - `c1.myColour == "red"`
  - `c2.myColour == "yellow"`
- Attributes can be defined with a start value.
- Attributes may be changed in methods.
- Parameters are defined as part of a method,
- Parameters have one value each time the method is called.
- The value is "given" by the calling method.
  - e.g. `theCar.calculateFuelConsumption(theCar.getCurrentDistance(), 40) // current distance in km, 40 litres`
- The value can change within the method, but this does not change the value in the caller.

- Parameters are essentially *Local Variables* whose value is defined elsewhere.
- Parameters *can not* be defined with a default value.
- Local variables are definied anywhere inside a method.
- Local variables are only usable *from that point onwards*
- Local variables can be defined with a start value.
- Local variables can change within the method.
- Local variables are, in fact, valid for a specific block {}, which we will discuss later.

```
public class FluxCapacitor {
  private static final int POWERCONSUMPTION = 2; // MJ
  private int startYear;// = Date.now();
  private int destinationYear = 1955;
  private int requiredPower = 0;

  FluxCapacitor(int theStartYear) { // This is not the most obvious constructor, or indeed
    this.startYear = theStartYear;
    this.setDestinationYear(2015);
  }

  public void setDestinationYear(int theDestinationYear) {
    this.destinationYear=theDestinationYear;

    int tripLength = startYear - destinationYear;
    requiredPower = Math.abs(tripLength * POWERCONSUMPTION);

    theDestinationYear = 1955;
    tripLength = startYear - theDestinationYear;
  }

  public String toString() {
    return "FluxCapacitor set to " + startYear + " (start) " + destinationYear + " (destin
  }

  public static void main(String [] args) {
    FluxCapacitor fc = new FluxCapacitor(1985);
    System.out.println(fc); // Special "Java Magic": any object can be cast to a String. T
  }
}
```

12 Deeper into the Difference between built in Data Types and Objects

- Computer Memory is used in two ways:

Heap allocate a piece of memory at a random place with `new`

Stack One continuous piece of memory that shrinks and grows based on current needs.

```
 int x; // allocate four bytes on the top of the stack. When x is used, these four bytes a
 Car c3; // allocate 64 bits on the top of the stack

 new Car(); // allocate size of all attributes in Car on the heap
            // since we don't do anything with this, it will go straight
            // to the garbage collector.

Car c4 = new Car(); // allocate 64 bits on the top of the stack AND
                    // the size of Car on the heap.
                    // Put the address of the Car object in the 64 bits referred to by c4.

c3 = c4; // Copy the address of c4 into c3 (the specific 64 bits on the stack referred to
         // UNLESS the class Car has a copy constructor. Which we'll get to eventually.

int y = x; // allocate four bytes on the top of the stack AND
           // copy the contents of the four bytes referred to by x into these.
y = y +1;  // Since y is a copy, this does not change the value of x
c3.setSpeed(70) // c3 and c4 refer to the same object, so c4.getSpeed() will also return 7

public int someMethod(int aParameter) { // Nothing really happens here, BUT when the metho
                                        // Allocate four bytes on the stack AND
                                        // copy the parameter value from wherever someMet

  int localVar; // Allocate four bytes on the stack
  Car c3 = new Car(); // Allocate four bytes on the stack for the variable c3
                      // AND allocate the size of a Car on the heap, as before.
                      // Within this method, c3 refers to these bytes, and it is
                      // tricky to access the other c3 that was defined outside this block

  c4 = new Car(); // Allocate the size of a Car on the heap AND
                  // overwrite the previous reference that c4 held.
                  // The old car goes to the garbage collector.
  return localVar;

} // End of this method block
  // Back up the stack with 32 bits for c3 and four bytes for localVar
  // The Car formely referred to by c3 is now "free" and goes to the garbage collector.
  // Push the value previously stored in localVar onto the stack, where it will be
  // retrieved by wherever someMethod() was called.
```

13 Fundamental Input and Output

- `System` class, available anywhere.
- In particular, `System.out.print()` and `System.out.println()`

- `System.out` is a `PrintStream` which normally refers to standard output (the console)
- Please look at the Java documentation to see what a `Printstream` can do:
  - https://docs.oracle.com/en/java/javase/20/docs/api/index.html
  - https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/io/PrintStream.html
- `System.in` is an `InputStream`, but this is tricker to use straight off.
- Better to use a `Console` , which you can get by calling `System.console()`.

```
import java.io.*;
Console con = System.console();

System.out.print("Enter your name: ");
String name = con.readLine();
System.out.println("Hello " + name);

// Or, shorter
name = con.readLine("Please enter your name again: ");
System.out.println("Hello again, " + name);

// We can complicate things
String lastname = con.readLine("What is your lastname, o %s? ", name);
con.printf("Greetings, %s %s!\n", name, lastname);
```

14 Conditional Execution: if

```
// Basic form:
if ( /* some true or false test */ ) {
  // Code to run if true
}

// With an 'else':
if ( /* some true or false test */ ) {
  // Code to run if test is true
} else {
  // Code to run if test is false
}

// Daisy-chaining
if ( /* some true or false test */ ) {
  // Code to run if test is true
} else if ( /* some other true or false test */) {
  // Code to run if the first test is false AND the second test is true
} else {
```

```
  // Code to run if the first test is false AND the second test is false
}

public class Car {
  private String owner;

  public boolean isAvailable() {
    if ("" == this.owner) {
      return true;
    } else {
      return false;
    }
  }

  public boolean isAbandoned() { // Same as above, but just evaluate the test.
    return ("" == this.owner);   // We can do this since we did not have any other code th
  }                              // we wanted to execute. Until we do, then we need to ref
}
```

15 Now the same for C++

15.1 Header file

- Separate file for the *class definition*, called a *header file*
- Preprocessor commands (`#ifndef`, `#define`, and `#endif`)
  - These make sure that the class Car is only defined once.
  - Only necessary for the header file, since this is the only one that will be included by others.
- Strings are not "built-in", but have to be included as a library with `#include <sting>`
- Strings are in a separate *namespace* called `std` .
- We use the *scoping operator* `::` to get to the right scope: `std::string`
- We have to tell c++ when there are no parameters, `std::string toString(void);`
- **Never forget** the semicolon after the class definition `};`

```
#ifndef CAR_H
#define CAR_H
#include <string>

class Car {
public:
  Car(std::string theLicensePlate, std::string theColour);

  std::string toString(void);
```

```
private:
  std::string myPlate;
  std::string myColour;
  std::string myOwner;

};
#endif
```

15.2 Class file

- Include the header file
- Include all libraries we may wish to use
- We may take a shortcut to avoid having to write `std::` so often, i.e. `using namespace std`.
  - Now, everything in `std` is directly available to us.
- We must scope all methods to say which class they implement, e.g. `string Car::toString(void)`
- We may use an *initialiser list* with our constructors.

```
#include <string>
#include "car.hh"

using namespace std;

Car::Car(string theLicensePlate, string theColour) : myPlate(theLicensePlate), myColour(th
}

string Car::toString(void) {
  string s = "a " + this->myColour + " car with license plate " + myPlate;
  return s;
}

bool Car::isAvailable(void) {
  if ("" == myOwner) { // if works in the same way as in Java. Be careful with operators o
    return true;
  } else {
    return false;
  }
}
```

15.3 Main function

```
#include "car.hh"
#include <iostream>

using namespace std;
```

```
int main(void) {
  Car* c = new Car("aaa111", "red");

  cout << c->toString() << endl;
}
```

15.4 Pointers or Variables

- The biggest difference between Java and C++ is *pointers*
- They allow direct reference to a memory location
- Used extensively in Object Oriented Programming in C++
- Careful! With great power comes great responsibility
  - It is easy to make mistakes and point to something which no longer exists
  - It is equally easy to forget to clear the memory of an object that is no longer being used

```
#include <iostream>

using namespace std;

int main(void) {
  int x = 10;
  int *y; // A pointer to an int
  y = &x; // y now points to the address of x;

  cout << "We start with " << x << " : " << *y << endl;
  // We have to de-reference y in order to get the value it points to.

  x = 20;
  cout << "Now we have " << x << " : " << *y << endl;
}
```

15.5 Pointers to an Object

- An object is accessed differently if the variable is a pointer or a “normal” variable.

```
Car *carPointer = new Car("bbb222", "yellow");
Car carNormal("ccc333", "green");

carPointer->toString()
carNormal.toString();

// It does not matter how the object was created
// only what type the current reference is.
```

```
Car *anotherPointer = &carNormal;
anotherPointer->toString();
```

15.6 Fundamental Input and Output

```
#include <iostream>
#include <string>
using namespace std;

int main(void) {
  cout << "You have already seen the ";
  cout << "output stream" << " operator in action" << endl;

  string name;

  cout << "Input could have been similar. What is your name? ";
  cin >> name;

  cout << "But with a glaring problem, " << name << ". What is your full name? ";
  cin >> name;
  cout << "hello " << name << endl;
  // This will not only not work, it will break the input stream so the next input
  // will seemingly not wait for any input.
  // The reason is that cin will stop reading at any blank character (space, tab, newline,

  cout << "Let's fix this. Try again with your full name: ";
  getline(cin, name);
  cout << "hello, dear " << name << endl;
}
```

16 Summary

Real World:

- full of *Objects*
  - with data and behaviour

Software Program:

- *Object* represents a real world entity
- *Class* to describe a group of objects
- *Built-in data types* vs Classes/objects
- An *attribute* describes a single datum for a single object
- A *method* operates on a single object
  - accessor and mutator methods

- Conditional: `if`
- Fundamental input/output

Software Design:

- Design Principle: *Low Coupling*
  - Private vs Public interface of a class

17 Next Lecture: Interacting Objects

- Barnes & Kölling Chapter 3, Object Interaction
- Design Principle: *High Cohesion*
- Design Principle: *Encapsulation*
- Object Oriented Analysis