Condensed Java Basics

Simon Symeonidis

Version	Author	Date	Comment
1	Simon	19/3/2015	Released to terrorize possible readers

Contents

1 Location of notes

All my notes are free to be read by the willing. Now, the quality of my writing is another question all together.

The repository of my notes exists here:

https://github.com/psyomn/

These particular notes, may be accessed here:

https://github.com/psyomn/architecture-notes/tree/master/languages/java

Changes and suggestions are welcome. You will be credited for possible contributions.

2 Introduction

The purpose of this document is to highlight the basics, and introduce core examples to the reader. With this document you should be able to remind yourself core features of the language, that in turn may be used to create simple object oriented applications.

We will go through the following: structure of a simple program. Variable declarations, and simple operations. Control structures. Classes. Visibility in classes. Inheritance, and use of polymorphism.

2.1 Structure of a Simple Program

A program is a set of operations that need to be executed at some order. Ultimately you must supply the computer with two things: what needs to be executed, and where to start. Many programming languages have the *main* construct to account for this. Java has this convention too. You need to remember 2 rules for your Java programs, which might be different if you're much more used to writing C/C++ programs:

1. The file must have the same name as the class you will be writing. For example if you are writing `class Person { ... }`, then your file should be named `Person.java`.
2. The file must only contain one class definition. You can however define inner classes within the main class of the file.

2.2 Hello World

The basic structure of a main program in *Java* is the following:

```
public class Main {
  public static void main(String[] args) {
    System.out.println("Hello world!");
  }
}
```

In Java every file contains one *root* class. That class may contain inner classes as well. But no file will ever contain more than one class.

Due to the above rule, we need to enclose our *main* call inside another class. This raises the question - can there be, in a project, more than one file, containing a class with the *main* routine?

And the answer is yes. However, when compiling and running, you would have to specify which *main* to use. We will omit this information, as it is beyond beginner scope (but good to know).

2.3 Variables

Performing operations is nice, but adding the possibility of manipulating data is even better. There are two things you need to keep in mind: there exists primitives, and classes.

Primitives would include data types such as *int*, *float*, *char*, *bool*, *double*.

```
int x = 2;
int y = 3;
int z = x + y;

double one = 2.3;
double two = one + 4.21;

boolean isOk = false;

char a = 'c';
```

Classes would include types such as *String*, *StringBuilder*, *Person*, *Calculator*, etc. The general convention is that classes tend to be capitalized.

And there exists primitive arrays. Primitive arrays means that you have a sequence of memory locations reserved for data. You can cram inside either a type of primitive, or class.

Classes, and primitive arrays need to be created in the following way, with the keyword *new*.

```
int[] a = new int[12]; /* An array of ints */
String str = new String();
String[] args = new String[12];
```

Please notice the creation of *String* and *String[]*. The former creates the object, while the second reserves *array space* for 12 *String* objects. You would have to create each string independently if you wanted to use the array of objects. So with that said this is how you would set the values of the array.

```
String[] stringArr = new String[10];

for (int x = 0; x < stringArr.length; ++x) {
  stringArr[x] = new String("The string");
}
```

Please don't confuse these primitive arrays with the Java implementation of the basic data structures such as *ArrayList* and *LinkedList*. We will see these in the next section.

2.4 Classes vs Objects

A good thing to do at this point is to differentiate between what classes and objects are. Classes are the definition. Objects are the result.

Another analogy would be the following: imagine spheres of different sizes and color. They all belong to a same family. But each one of them, with the same questions, will yield different results. For example, any two elements `A` and `B` are spheres. However they may be of different color. A better example would be to classify them with respect to behavior: all of them will respect the formula, that gives us the area of a sphere. The only information required, is the size of their radius: in other words given some differing information of a radius `r`, we may apply the same function for any `r`, and get the required behavior.

So, in other words, a class may contain crucial behavioral definitions, whereas objects contain the differing information that the behavior requires. More abstract, classes are specifications, whereas objects are instantiations of those specifications, possibly given differing data.

The user has control over both classes and objects. Here is a simple class:

```
class Sphere {
  public Sphere(float iRadius, String iColor) {
    mRadius = iRadius;
    mColor  = iColor;
  }
  public String color() { return mColor; }
  public float area() { return 4 * Math.PI * Math.power(mRadius, 3); }
  private String mColor;
  private float mRadius;
}
```

You can see what was previously described in code form. We have set behavior for the Sphere we modeled. Now it will behave according to what `radius` we supply to the object, instantiated from the class. To instantiate an object, we may do the following:

```
public class Main {
  public static void main(String[] args) {
    Sphere s = new Sphere(3.2, "red");
    Sphere r = new Sphere(4.8, "blue");
    System.out.println(s.area());
    System.out.println(r.area());
  }
}
```

If we were to visualize this in the memory space, we could see these two objects as the following:

```
+--------+   +--------+
| Sphere |   | Sphere |
+--------+   +--------+
| 3.2    |   | 4.8    |
| "red"  |   | "blue" |
+--------+   +--------+
```

And using the declared method `area`, we would be able to 'send a message' to these objects in memory. Both of them know about the method `area`, and what actions to take (see the implementation of `area()` in the code listing). What these objects need to do is 'plug in' the variables they hold to their respective `instance variable` declarations. So when calculating the area of the red sphere, we would want the radius of `3.1` to be plugged in the formula. However, when calling area for the "blue" sphere, we wish to 'plug in' `4.8` as the value of the radius.

Essentially you should think of objects as entities, that know about what their structure is like (what data fields they have, and what behavior), and which behave differently whenever given different data. There can be many 'entities' expressing this behavior at any given time.

Now let us look at the more practical side of things. It is assumed that the reader is familiar with procedural programming. We will write the following in `C` code to demonstrate our point. Recall that in C we do not have classes, but structures which are strictly used to hold data. The functionality comes from providing many free functions with knowledge on how to manipulate the data. So the equivalent to the above example with the spheres would be something like the following:

```
typedef struct {
  float    radius;
  uint32_t color_code;
```

```
} sphere_t;

void
sphere_set_radius(sphere_t* _s, float _r) {
  _s->radius = _r;
}

float
sphere_calculate_area(sphere_t* _s) {
  return _s->radius * _s->radius * 3.142 * 4;
}
```

You can notice that on the above, both functions would require a reference to the structure we are dealing with. That particular structure is omitted from the type signatures in most object oriented languages, and a keyword called `this` is provided, if the particular behavior of a class needs to refer to that particular object in the runtime.

2.5 Visibility

In `C` programs we don't quite have the notion of visibility. We have something similar in the respect of static declarations in objects (by objects I'm refering to the object code generated by the compiler, usually having the `.o` suffix), but that is not what we are talking about.

Take the C structure in the following fragment:

```
typedef struct {
  int age;
  float salary;
} person_t;
```

If we had the following function, accessing members of the struct would be legal:

```
void my_func(person_t* _person) {
  /* Given that _ms is non-null */
  _person->myint = 1;
  _person->fval  = 2.32f;
}
```

2.5.1 Private Visibility

In object oriented languages we are able to restrict this visibility. The motivation is to provide an agnostic way of handling data 'behind the scenes', whilst providing a *standard interface* to the application programmer, using this unit. In Java, one is required to use the

`private` keyword for restricting this visibility. If this restriction is not required, and some feature (either a method, or a variable) is to be called directly, then we can use the `public` keyword. The above could be translated to the following Java code, if we wished to restrict the visibility to the two variables:

```
class Person {
  private Integer age;
  private Float salary;
}
```

If an application developer tried using the above class, and calling the attributes directly, there would be a compilation error:

```
Person p = new Person();
System.out.println(p.age); /* Does not compile */
```

2.5.2 Public Visibility

As we previously saw, we can declare different parts of a class as public. For example the following class has a few of it's attributes as public:

```
class Person {
  public String name;
  public String surname;
}
```

This means that we can directly access the data from some other part in the program in such a fashion:

```
public class Main {
  public static void main(String[] args) {
    Person p = new Person();
    p.name = "jonny";
    p.surname = "overload";

    System.out.println(p.name);
    System.out.println(p.surname);
  }
}
```

One might notice and pose the following question: *if I can simply access the data that I require using public visibility, why not use public everywhere?*

You could technically use public everywhere. But you would sacrifice one mechanism that you might not want to otherwise: you treat objects as black boxes which you do not care how things are implemented on the other side - but you know what the output is.

2.5.3 Why Private Visibility

To the users of some component in your software, expressed as a class, you want to publicize methods which are relevant to the problem that user of the class wants to solve. For instance, take the example of our previous example of the `Sphere`. Our user would only care about the particular function that returns the *color* of the `Sphere`, or the *area*. In the inside of the class, we might split into different functions different procedures we need to do until we reach our final answer. For example the `getColor()` method might call other functions to finally return the color to the user, and these functions could be of no particular interest to the user.

For a more concrete example, let us consider the case that `getColor()` should return the color label as a string, but all the letters should be in capitals. Inside the Sphere class we would have a function called `capitalize`, which would capitalize strings. The user would not care about such a thing, but the functionality in the class, would require such an operation. Using `private` visibility we are able to hide this information such that the user of the `Sphere` class would not know of such functionality.

A more realistic example would be if the `getArea()` functionality required more complex calculations which could be split up into smaller units, allowing for different optimizations to be made. The user of the class is interested in knowing about the `getArea()` operation, regardless of the implementation.

2.5.4 Protected Visibility

Protected visibility is something we will cover once we cover an advanced topic later on. For now it is sufficient to know that the visibilities which exist are:

- Public visibility
- Private visibility
- Protected visibility

2.6 Creating a Small Application

When you want to create a small application, you essentially want to break down different functionalities in different classes. In other words, if you choose a good granularity on how much you break these functionalities down, the easier your life will be on the development side of things.

Let's look into creating a small application, where we implement an address book. The address book will hold names, surnames, and emails. The user will be prompted to enter different information for different situations. We would like to:

- Have a menu to choose whether to:

- Create an entry of a person.
- Lookup the information of a person.

We should worry about two things: separating the structures which may hold this information, to the user interface that interacts with these structures. Essentially you want to separate these two things to two or more classes.

Let's first begin with our data classes, in other words our conceptual domain. We are interested in the three data fields: name, surname, and email.

Writing this down we first get the following:

```
class Person {
  private String name;
  private String surname;
  private String email;
}
```

We notice that the above attributes are private. How is the user of the class supposed to set these values now? We use getters and setters. The getters return the values of the attributes, and the setters overwrites or sets them. These operations are defined with public visibility instead. In code we can simply show this as such:

```
class Person {
  private String name;
  private String surname;
  private String email;

  public String getName() {
    return name;
  }

  public String getSurname() {
    return surname;
  }

  public String getEmail() {
    return email;
  }

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

  public void setSurname(String surname) {
```

```
    this.surname = surname;
  }

  public void setEmail(String email) {
    this.email = email;
  }
}
```

We pretty much have a structure to hold the information we want for a person. Now we want an `AddressBook`, which will store this information. Essentially, we want to have a method to add people, and remove people. Let's call these methods `addPerson()` and `removePerson()` respectively. We will have an extra method that looks up a person by name, and if found, it will print all that information of that person. Else, there will be an error message printed. We call such method `findPerson()`.

To store each person, we will need a data structure. The data structure we will choose for this particular example will be an `ArrayList`. An `ArrayList` may expand or shrink depending on the information given. To add objects to the data structure, we call `add()`. To remove objects from the list we call `remove()`. There is one final thing that we need to do: tell the structure that we will be storing `Person` objects. The reason to this, we will see later in the `Templates` section. To tell the structure `ArrayList` we are storing `Person` objects, we show this by adding it between angled brackets.

```
public class AddressBook {

    public void addPerson(Person person) {
        people.add(person);
    }

    public void findPersonByName(String name) {
        for (Person p : people) {
            if (p.getName().equalsIgnoreCase(name)) {
                System.out.println("Found:");
                System.out.println("Name:    " + p.getName());
                System.out.println("Surname: " + p.getSurname());
                System.out.println("Email:   " + p.getEmail());
                return; /* Stop searching in list */
            }
        }
        System.out.println("Nothing found!");
    }

    private ArrayList<Person> people = new ArrayList<Person>();
}
```

With this code, we're able to store all of our wanted information somewhere, add, and remove as needed, and search. Finally we want a final class that will handle the input and output interactions with the user. Essentially this is the interface of our application. Since we're not providing anything graphical, this is a command line interface. We will name this class `AddressBookController`.

We provide a command line interface by using `Scanner`. Scanner provides different methods to receive input. We specify what we expect by using the right operation. For example if we want an integer, we would call `nextInteger()`, if we wanted a double floating point number, we would call `nextDouble()`. Since we're scanning in Strings, we use `nextLine()`. This will make sure that anything typed in, until the user hits the return key, will be scanned in. On the other hand, if a simple `next()` was used, then it would parse in just the first word which would be delimited by a space.

The code below shows what our controller would look like.

```
public class AddressBookController {
    public void run() {
        String cmd = "default";
        Scanner scan = new Scanner(System.in);

        while (!cmd.equalsIgnoreCase("end")) {
            System.out.print("> ");
            cmd = scan.nextLine();
            cmd = cmd.trim();

            if (cmd.equalsIgnoreCase("insert")) {
                Person p = new Person();
                System.out.print("Person name: ");
                p.setName(scan.nextLine());

                System.out.print("Person surname: ");
                p.setSurname(scan.nextLine());

                System.out.print("Person email: ");
                p.setEmail(scan.nextLine());

                addressBook.addPerson(p);
            }
            else if (cmd.equalsIgnoreCase("find")) {
                System.out.print("Name of person: ");
                String name = scan.nextLine();
                name = name.trim();
                addressBook.findPersonByName(name);
            }
            else if (cmd.equalsIgnoreCase("help")) {
```

```
                System.out.println(
                  "insert - insert a person's details into the address book\n" +
                  "find   - find a person by name\n" +
                  "end    - quit this application\n");
            }
            else if (cmd.equalsIgnoreCase("end")) {
                continue;
            }
            else {
                System.out.println("No such command");
            }
        }
    }

    private AddressBook addressBook = new AddressBook();
}
```

Finally we just need an entry point for our application:

```
public class Main {
    public static void main(String[] args) {
        AddressBookController abc = new AddressBookController();
        abc.run();
    }
}
```

If we run it and try some input, we get the following:

```
> help
insert - insert a person's details into the address book
find   - find a person by name
end    - quit this application

> insert
Person name: jon
Person surname: jonson
Person email: jon@son.com
> find
Name of person: jon
Found:
Name:    jon
Surname: jonson
Email:   jon@son.com
> find
```

```
Name of person: potato
Nothing found!
> end
```

3 Advanced Topics

In this section, we will talk about 'tricks' we can do with Object Oriented programming. The 'tricks' we will use is to achieve grouping certain objects together for common behavior, or extending an object in order to add more functionality or specialize the object. We will go through various examples and expand on these topics.

3.1 Interfaces

We will begin, by showing how we can use interfaces of objects, we can change the implementation in compile time as well as runtime to some extent, and still have the application behave normally - without crashing.

For a more practical example think of a random number generator. The random number generator could use a very common technique to generate random numbers. Some techniques might purely be algorithmic. Other approaches may include outside resources such as hard disks, mouse input, microphone input, and anything that can aid you with providing 'real' random numbers.

In the context of OOP, you'd want a functionality that returns a random number without caring too much what kind of things had to happen in the background in order to get these numbers. So you'd want a method called `getRandom()`, which returns a random number. Let's say that this random number is of type `integer`.

We can define the following specification of operations:

```
public interface IRandomGenerator {
  int getRandom();
}
```

Notice that there are no visibility setters. When we define interfaces, everything is public - you are defining *an interface* after all. You might notice that the name `IRandomGenerator` is prefixed with an `I`. This is a common convention, to hint to the programmer that the type in question is in fact an interface. Try to prefer this convention.

Now, any class, who's responsibility is to generate a random number, can implement this interface. By implementing this interface, the user of any class that implements the `RandomGenerator` interface will have the guarantee that the `getRandom()` method exists, and will be callable.

Let's write our first implementation of of such an interface. For our naive implementation, we're simply going to return the number `42`. We achieve this with the following code snippet:

```
public class NaiveRandomGenerator implements IRandomGenerator {
    @Override
    public int getRandom() {
        return 42;
    }
}
```

If we want something slightly more sophisticated, then we use `Java`'s random library implementation in our alternate implementation of the interface. We can do this by importing the `java.util.Random` package using the import keyword. We create the class `BorrowedRandom` to achieve this alternate implementation:

```
import java.util.Random;

public class BorrowedRandom implements IRandomGenerator {
    @Override
    public int getRandom() {
        Random rand = new Random();
        return rand.nextInt() % 100;
    }
}
```

Finally let's test this out via a main method:

```
public class Main {
    public static void main(String[] args) {
        IRandomGenerator rgen;

        rgen = new NaiveRandomGenerator();
        System.out.println(rgen.getRandom());

        rgen = new BorrowedRandom();
        System.out.println(rgen.getRandom());
    }
}
```

Notice: we are declaring an interface type in the beginning called `rgen`. It would be impossible to invoke any operations on this object as this is an interface - there is no implementation. The 'magic' happens later when we assign a new object on the right hand side. We see the first occurrence in this line:

```
rgen = new NaiveRandomGenerator();
```

This is possible because `NaiveRandomGenerator` implements the interface we are casting the object to. As does `BorrowedRandom`, and the reason you see the second line, as a legal statement:

```
rgen = new BorrowedRandom();
```

Both will eventually execute `getRandom()`. However, during runtime, depending on what we cast, we get the appropriate `getRandom()`. So, for the first line, we will return `42` due to the naive implementation. On the second line, we will always get random numbers, generated by `Java`'s random library. A simple run gives us these values in the output prompt:

```
42
-68
```

This means that you can have a dependency on some component in your software, that could be changed at any time, which would not require you to go to all the source to add these changes, since it respects that common interface. On top of that, we could change the behavior during runtime, by alternating between objects, which implement this common interface.

Would it be possible however, to add some sort of implementation details and behavior to interfaces, but retain this 'dynamic switching' between objects? The answer is yes. You achieve this through polymorphism, though the rules are slightly different in this case.

3.2 Polymorphism

Polymorphism follows very similar steps to that of interfaces, only that instead of interfaces, we are extending a class. In abstract terms, we have some `class A`, that exhibits features `{k, l, m, n}`. If we were to extend `class A` via `class B`, and `class B` exhibited features `{o, p}`, then the features that `class B` would have in total would be `{k, l, m, n, o, p}`. This is called 'inheritance' and happens when we extend one class via another. Let's take a look at the following code snippet:

```
public class Animal {
    public float getBodyTemperature() {
        return 37.5f;
    }
    public void talk() {
        System.out.println("I am an awesome animal");
    }
}
```

Now, to extend this class, we need to use the keyword `extends`. We will do this with a class called `Cat`. This is how you inherit from a parent class.

```
public class Cat extends Animal {
}
```

In the above class we're not really adding anything. This is purely for demonstration purposes. We can observe then, that if we do the following, the program would compile, and nothing out of the ordinary happens, since `Cat` has all the features of `Animal`.

```
public class Main {
    public static void main(String[] args) {
        Animal a = new Animal();
        Cat c = new Cat();

        System.out.println(a.getBodyTemperature());
        System.out.println(c.getBodyTemperature());
        a.talk();
        c.talk();
    }
}
```

3.2.1 Adding New Functionality

Of course, we could add more functionalities to the `Cat` class. This would mean that apart from all the features obtained from the parent class `Animal`, `Cat` would have extra functionalities as well.

```
public class Cat extends Animal {
    public void scratchFurniture() {
        System.out.println("HA HA HA HUMAN");
    }
}
```

Again the following would be legal:

```
public class Main {
    public static void main(String[] args) {
        Animal a = new Animal();
        Cat c = new Cat();

        System.out.println(a.getBodyTemperature());
        System.out.println(c.getBodyTemperature());

        a.talk();
        c.talk();
```

```
            c.scratchFurniture();
        }
    }
```

However, the following line would not be legal, since `Animal` does not implement the method `scratchFurniture()`. Only the class `Cat` in this context can scratch furniture:

```
Animal a = new Animal();
a.scratchFurniture(); /* Does not compile */
```

3.2.2 Overloading

A brief reminder of Overloading: overloading is when we have two functions that return the same type, and have the same name, but differ in the parameters they actually take in, be it type of parameters, and number of parameters. So for example, these two functions are overloaded:

```
class MyClass {
  public int myMethod() {
    return 42;
  }

  public int myMethod(int i, int j) {
    return i + j;
  }
}
```

3.2.3 Overriding

Overriding exists as a result of Polymorphic types. Recall the previous discussion about having a `class A` with features `{k, l, m, n}`, and extending it via `class B` with features `{o, p}`, resulting in all the features `{k, l, m, n, o, p}` available to `class B`. So now the question is, if we have a `class C`, with features `{a, b}`, and `class D` with similar feature `{b}`, then which `b` is the `b` that will be used? The short answer, is the `b` of `class D` will be used. It can get more complicated however.

In less abstract terms, consider a `class A` which has method `void talk()`. We extend `class A` via `class B`. In `class B` we add again, the method `void talk()`. What happens now? When we instantiate this class, which method actually runs? Below is the code represented in Java:

```
public class A {
    public void talk() { System.out.println("Hello from A"); }
```

```
}

/* ... */

public class B {
    @Override
    public void talk() { System.out.println("B B B B B"); }
}
```

If we perform the following:

```
public class Main {
    public static void main(String[] args) {
        A ab = new B();
        ab.talk();
    }
}
```

The result is going to be:

```
B B B B B
```

So, extending the parent class `A`, and in the subclass `B` adding the same method with the same method parameters and return time, we can get `rid` of the parent's behavior. Getting `rid` - Over-rid-ing. Let us examine the above in a little more detail. First you might notice this line:

```
A ab = new B();
```

This is possible because of rules similar to that of the interface - we know that `B` implements whatever the interface of `A` is. Therefore, when we call for an operation on `ab`, we know that the operation will be either found first in `B`, or eventually up in its superclass `A`. Therefore we can cast a `B` instantiated object onto a type, declared as `A`. In essence you need to observe two things with the above: the left hand side `ab` is the declared type, which is of type `A`, and the right hand side is of dynamic type `B`.

One might ask *Why would we do such a thing?* And again the answer would be very similar to that of interfaces: to interchangeably be able to switch one object, with another for the implementation. This would provide you, as stated in the interface section, two things: being able to change some implementation in the static time during compilation, and switching between objects with different implementations in the runtime. The difference however is that we can reuse implementation when extending a class - something not possible with interfaces. In classes you could extend a class, and `override` functionalities you would want different. In interfaces, you would have to re-implement everything for each class that implemented that interface.

3.2.4 Protected Visibility and Inheritance

You may recall the `protected` visibility we talked about previously. When we have a class with `private` features, we are unable to call these features from the outside, from another method. Also if we extend that class, then in the child class we would be also unable to access these features. What about a visibility that would disallow outside callers from seeing such features, whilst providing the child classes with the parent's features? This type of visibility is called `protected`.

If we had the following code, the operations would be legal:

```
public class Parent {
  protected void doThings() { /* ... */ }
}

/* ... */

public class Child extends Parent {
  public void myMethod() {
    doThings();
  }
}
```

3.3 Abstract classes

Abstract classes are something that are a mix of classes, and interfaces. That is we have some implementation, and we define some methods to be implemented when the class in question is subclassed. Therefore it behaves as a class, as well as an interface at the same time.

What we mean by this, is that we can freely implement any method in a class, and omit functionality to the methods we do not wish to implement in the base class. On a more concrete example, you may have some class that provides common functionality, depending on what another method yields. For example, imagine you are writing a small robot to scrape over html pages, over the internet. You want it to be able to detect links, and visit all the links it finds, in order to investigate all the possible paths. How you do this scraping, and what links you find, might not be important to the knowledge of the robot-scrapper - what it just wants, are list of links.

You could have two implementations: the robot that wants to look at all the links on the site, and the robot, who is only interested in crawling through the links that are hosted on the same domain (ie: first robot doesn't really care what links it finds - it just passes them to the implementation; whereas the second gets the links, filters the uninteresting ones, and then passes them to the behavior).

One thing to note: just like interfaces, you can not instantiate abstract classes.

Here is a sample abstract class:

```
public abstract class AbstractScraper {

    public AbstractScraper(List<String> links) {
        mLinks = links;
    }

    public void scrape() {
        System.out.println("Doing work...");
        List<String> links = selectLinks();

        for (String link : links) {
            System.out.println("Fetching: " + link + " ...");
        }
    }

    public abstract List<String> selectLinks();
    protected List<String> mLinks;
}
```

Notice that we set `mLinks` to protected. And here is the concrete implementation:

```
public class FetchAllScraper extends AbstractScraper {
    public FetchAllScraper(List<String> links) {
        super(links);
    }
    @Override
    public List<String> selectLinks() {
        return mLinks;
    }
}
```

Since this is the scraper that is to include all links, we may just return the `mLinks` member variable, when we are using `selectLinks()`. In turn, this list will be used in `void scrape()`, reusing the rest of the parts of the `AbstractScraper` class.

On the other hand, we can make a comparison using `Regular Expressions` in Java, and omit any link that is not from the particular domain we are interested in. This way we filter out some items from the list and provide the user of this small 'contraption', the wanted links:

```
public class DomainOnlyScraper extends AbstractScraper {

    public DomainOnlyScraper(List<String> links, String interestedDomainName) {
        super(links);
```

```
            domainName = interestedDomainName;
        }

        @Override
        public List<String> selectLinks() {
            Pattern pat = Pattern.compile(".*" + domainName + ".*");
            List<String> filteredList = new ArrayList<String>();
            Matcher m;

            for (String el : mLinks) {
                m = pat.matcher(el);
                if (m.matches()) {
                    filteredList.add(el);
                }
            }

            return filteredList;
        }

        private String domainName;
    }
```

And finally, the driver for the application, which we use to demonstrate the above:

```
    public class Main {
        public static void main(String[] args) {
            AbstractScraper absFetchAll, absSelective;
            List<String> links = new ArrayList<String>();

            links.add("domain1");
            links.add("domain2");
            links.add("domain3");
            links.add("domain4");
            links.add("cooldomain");
            links.add("notsocooldomain");

            absFetchAll = new FetchAllScraper(links);
            absSelective = new DomainOnlyScraper(links, "cooldomain");

            absFetchAll.scrape();
            absSelective.scrape();
        }
    }
```

Recall since both implementations of the `AbstractScraper` respect the its interface, then we can in the runtime, cast either one of the implementations (either `FetchAllScraper`, or `DomainOnlyScraper`) to the declared type. Both run the `scrap()` method, and we can observe the output. The fetch-all implementation will yield all the links. The second one will look for links which have the string 'cooldomain' somewhere within them.

```
Doing work...
Fetching: domain1 ...
Fetching: domain2 ...
Fetching: domain3 ...
Fetching: domain4 ...
Fetching: cooldomain ...
Fetching: notsocooldomain ...
Doing work...
Fetching: cooldomain ...
Fetching: notsocooldomain ...
```

Notice how the second part only yields those two links (*cooldomain*, *notsocooldomain*).

3.3.1 Alternate nomenclature

Abstract in Java is what is called `virtual` in C++.

3.4 Templates

To explain templates, let us go over an analogy. Imagine you are cutting out a square shape from a piece of paper. You can use this paper with the square hole, and paint, to quickly draw similar shapes with the same paint. That is you have a set of features that you want constant when drawing the shape - the only difference is the color you use. Templates in essence are similar, but may a little more dynamic than your average stencil art.

With templates, you are able to use the same functionality, but with different types. Templates are named like classes, but usually have angled brackets denoting what type they're taking in. Here is an example, that might now seem familiar:

```
ArrayList<String> listOfStrings = new ArrayList<String>();
```

We know what an array list is, and possible operations it can support (such as `add`, `get` etc). All this functionality may be specified in an generic form. For example, we can say that a collection such as `ArrayList` contains elements of type `T`. And we can also remove these elements, or add them to the structure dynamically. Templates, in this respect, are much more specific than interfaces, but more generic than abstract classes. There will be some sort of implementation, but one that does not specify the types.