PROCEDURE ABSTRACTION IN COMPUTER HIGH-LEVEL LANGUAGES: JAVA PROGRAMMING LANGUAGE

MANIRU MALAMI UMAR
Usmanu Danfodiyo University Model Secondary School, Sokoto. Nigeria
Email: manirutambuwal@gmail.com
GSM: +2347061802705, +2347059598080

SADIQ ALIYU AHMAD
Federal University Dutse, Jigawa State. Nigeria
Email: sadiqaliyu@gmail.com
GSM: +2347037852728, +2347059597999

ZIYA'U BELLO
Fawahir Tech, Sokoto. Nigeria
Email: ziyaulhaqbello@gmail.com
GSM: +2347032239961

ABSTRACT
This paper titled Procedure Abstraction in Computer High-Level Languages has discussed on the Issues of Data Abstraction and Procedural Abstraction. It first introduced and explained the concept of Procedure as an abstraction of a single memory-less action which is invoked with parameters and its effect depends upon the parameter values. Abstraction was also discussed which is process by which concepts are derived from the usage and classification of literal ("real" or "concrete") concepts, first principles, or other methods. Forms of Abstraction which are ***Data Abstraction*** *which refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details. Benefits of Data Abstraction which include: Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object.* ***Procedure Abstraction*** *which is as a particular mechanism for separating use from implementation. Benefits of Procedure Abstraction which include allows us to think about the general framework & postpone details for later and Gives us building blocks we can reuse in other situations. Also Examples of procedure Abstractions were also discussed in this paper. Finally Summary and Conclusions were drawn from the discussion.*

1.1. INTRODUCTION

Procedure Abstraction is a very wide topic in Computing. Before we proceed to the topic, let us first understand what ***a procedure*** is and also what ***Abstraction means***.

Procedure as defined by William (1996) is an abstraction of a single "memory less" action (i.e. an action with no internal state). It may be invoked with parameters, and its effect depends only upon the parameter values. (Example, a procedure to calculate the square root of a real value).

Bahrami (1999) defined ***Abstraction*** as a process by which concepts are derived from the usage and classification of literal ("real" or "concrete") concepts, first principles, or other methods. "An abstraction" is the product of this process—a concept that acts as a super-categorical noun for all subordinate concepts, and connects any related concepts as a *group, field*, or *category*. ***Abstractions*** may be formed by reducing the information content of a concept or an observable phenomenon, typically to retain only information which is relevant for a particular purpose. For example, abstracting a leather soccer ball to the more general idea of a ball retains only the information on general ball attributes and behavior, eliminating the other characteristics of that particular ball.

Bahrami (1999) also classified forms of Abstraction as:

- ***Data Abstraction and***
- ***Procedural Abstraction***

1.2 DATA ABSTRACTION:

Bahrami (1989) defined Data abstraction refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details. Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.

Data abstraction enforces a clear separation between the *abstract* properties of a data type and the *concrete* details of its implementation. The abstract properties are those that are visible to client code that makes use of the data type the *interface* to the data type—while the concrete implementation is kept entirely private, and indeed can change, for example to incorporate efficiency improvements over time. The idea is that such changes are not supposed to have any impact on client code, since they involve no difference in the abstract behaviour.

For example, one could define an abstract data type called *lookup table* which uniquely associates *keys* with *values*, and in which values may be retrieved by specifying their corresponding keys. Such a lookup table may be implemented in various ways: as a hash table, a binary search tree, or even a simple linear list of (key:value) pairs. As far as client code is concerned, the abstract properties of the type are the same in each case.

1.3. BENEFITS OF DATA ABSTRACTION

Goss (2001) stated that, Data abstraction provides two important advantages:

- Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object.
- The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code.

1.4 PROCEDURE ABSTRACTION:

Williams (1996) defined Procedural abstraction as a particular mechanism for separating use from implementation. It is tied to the idea that each particular method performs a well-specified function.

This idea, that each conceptual unit of behavior should be wrapped up in a procedure, is called **procedural abstraction**. In thinking about how to design your object behaviors, you should consider which chunks of behavior -- whether externally visible or for internal use only -- make sense as separate pieces of behavior.

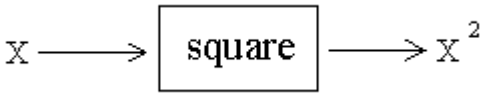
Procedural abstraction provides mechanisms for abstracting well defined procedures or operations as entities. The implementation of the procedure requires a number of steps to be performed. Procedural abstractions are used extensively by requirements analysts, as well as designers and programmers. Procedural abstractions are normally characterized in a programming language as "function/sub-function" or "procedure" abstraction

1.4.1 PROGRAM EXAMPLE FOR PROCEDURE ABSTRACTION

Let's continue our discussion on computing the area of a circle. We saw that using a named constant PI and a radius variable r helped, since we don't have to continually write out the value of Pi.

However, we still had to keep writing the formula over and over. For example, we have to say r * r every time we want to square the radius, and then we have to remember to multiply by Pi.

Let's first address the problem of squaring a number. Instead of having to write the expression r * r every time, it would be nice to have a **black box**, or a procedure, that would take in a numerical argument and return the square of that argument.



If we had this, we could write

square(837)

In Java, we can create a procedure called square as follows:

```
double square(double x) {
  return (x * x);
}
```

The "double" at the beginning is the type of return value from the procedure. This is followed by the name of the procedure, and then a list of **formal parameters** separated by commas. In this case, there is only one formal parameter, called x and its type is double.
When the procedure is **invoked**, the actual values passed in from the procedure call are bound to the formal parameters. The part in curly braces ({}) is the **procedure body**. The procedure body contains some number of statements that are executed in order. When one of the formal parameters is used inside the body, it's value is the value that was passed in from the calling procedure. The expression in the return statement is evaluated at the end of the execution of

the procedure and the resulting value is returned to the calling procedure as the value of the procedure call.
Now let's invoke the procedure. Suppose we execute the line
double area3 = square(3);

The value 3 is called the **actual parameter** to which the **formal parameter** x will be bound. Java creates a little **binding table** | x | 3 | for use inside the procedure and then evaluates the procedure body using that binding table. This little binding table is called an ***environment***.

So, the actual parameter 3 is passed into the procedure square and bound to the variable x. Then the body of the procedure is executed. The return value 9 is computed, returned as the value of the expression "square(3)", and subsequently assigned to the variable area3.

Any expressions can be used in the actual parameter list when making a procedure call, provided that the types match the types of the corresponding formal parameter list. For example,

```
double x, y;

double r = 3;

x = square(2+1);    // 2 + 1 is evaluated, 3 is passed in

y = square(r);      // r is evaluated, 3 is passed in

i = square("3");    // ERROR: type String doesn't match double parameter
```

Example 1: areaOfCircle

Having a square procedure is fine, but what we ***really*** want is a procedure called areaOfCircle that compute the area of a circle, given the radius. Then we could write statements like
double area5 = areaOfCircle(5);

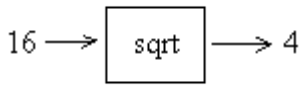
So, how do we write the procedure? Assuming that we have already defined the constant PI, we can write:

```
double areaOfCircle(double radius) {
  return (PI * square(radius));
}
```

Notice that we are using the square procedure to compute the square of the radius.
Let's do some more examples of procedures.

Example 2: hypotenuse

Suppose we are given a procedure sqrt that takes one parameter (a double) and returns its square root:



We don't know what sqrt is doing inside, but we can use it anyway because we understand its specification. This is a nice advantage of procedural abstraction.

Suppose we want a procedure **hyp** that finds the length of the hypotenuse of a right triangle, given the lengths of the other two sides.

Recalling that the length of the hypotenuse is the squre root of the sum of the squares of the lengths of the sides, we can write:

```
double hypotenuse(double sideA, double sideB) {
  return sqrt(square(sideA) + square(sideB));
}
```

And we can call the procedure as follows:

```
double hyp = hypotenuse(3,4);
```

Let's think about the evaluation step by step, using the substitution model:

```
hypotenuse(3,4)
sqrt(square(3) + square(4))
sqrt((3 * 3) + (4 * 4))
sqrt(9 + 16)
sqrt(25)
5
```

Built-in Mathematical Procedures

To get all of this to work, we need a procedure for square root. We could write one, but Java provides a lot of standard mathematical functions as methods of a static class called Math. (A method is a procedure defined as part of a class.)

We'll say more about classes later, but for now you can think of the math class as a collection of procedures (methods) and constants, accessed by Math.*name-of-method* or Math.*name-of-constant*. Examples include:

```
Math.sqrt(double x)    // returns the square root of x

Math.pow(x,y)          // returns x raised to the power y

Math.PI                // an approximation of pi
```

See the Math class documentation for a complete list of the available methods and constants. Another useful class is the System class that provides an output stream for printing textual output from your program to the screen, as in the following example.

Example 3: Complete Program

Let's write a complete program using the procedures we have created. The program calculates hypotenuses of some right triangles.

```
public class Triangles {
  public static void main(String args[]) {
    test(3,4);
    test(9,12);
  }
  public static double square(double x) {
    return (x * x);
  }
  public static double hypotenuse(double sideA, double sideB) {
    return Math.sqrt(square(sideA) + square(sideB));
  }
  public static void test(double a, double b) {
    System.out.print("A right triangle with sides " + a + " and " + b);
    System.out.println(" has hypotenuse " + hypotenuse(a,b));
  }
}
```

Some notes:

- The first line says that we are defining a class called Triangles that is publicly available (accessible externally).
- The second line sets up a method called main that will be the entry point of the program.
- The modifier **static** means that the method is part of the class, as opposed to a method of an object instance of the class. We'll see more about this later, but for now, just think of a class as a type of object. We may create many objects of a given class and call methods on those objects to ask the objects to do things. However, sometimes we want the class itself to be able to do things. These are called **static** methods.)
- The modifier **void** means that the method does not return a result.
- The System.out.print method prints its argument to the screen. The System.out.println method is the same, except that it also goes to the next line after printing.

So, the output of the program would be:
A right triangle with sides 3 and 4 has hypotenuse 5.
A right triangle with sides 9 and 12 has hypotenuse 15.

1.5 BENEFITS OF PROCEDURE ABSTRACTION

Some of the benefits and advantages of Procedure abstraction according to Williams (1996) are:

- Hides details.
- Allows us to think about the general framework & postpone details for later.
- Gives us building blocks we can reuse in other situations.
- Allows us use local names.
- Allows us to easily replace implementations by better ones.

1.6 CONCLUSION

From the above discussions, one will see that ***procedure abstraction*** is very important especially in the area of computer programming, requirement analysis and system designers. Procedure abstraction helps computer programmers i.e. those who are learning or wants to be experts in programming to see how procedures or methods are invoked in a particular program. With Procedure abstraction, one will see and know how exactly procedures or methods are invoked in a program.

1.7 REFERENCES

Ali Bahrami (1999), "Object Oriented Systems Development", *Mc Graw-Hill. [online] Available http://www.onlineschool.com/ProcedureAbstraction.pdf Accessed (March, 2014).*

Gerhard Goss (2001): "Compiler Construction Lecture Notes. Department of Computer Science University of Karlsruhe, Germany. *[online] Available http://www.ipd.info.uni-karlsuhe.de . Accessed (March, 2014).*

Williams M. White, "Compiler Construction Lecture Notes. Department of Electrical Engineering, University of Colorado, USA. *[online] Available http://www.colorado.edu/notes/ProcedureAbstraction.pdf Accessed (March, 2014)*