|  |
| --- |
| **FURTHER EDUCATION AND TRAINING CERTIFICATE: INFORMATION TECHNOLOGY: SYSTEMS DEVELOPMENT**  **ID 78965 LEVEL 4 – CREDITS 165** |
| **LEARNER GUIDE**  **SAQA: 14909**  **DESCRIBE THE DIFFERENCE BETWEEN PROGRAMMING IN OBJECT ORIENTATED AND PROCEDURAL LANGUAGES** |

**Learner Information:**

|  |  |
| --- | --- |
| **Details** | **Please Complete this Section** |
| Name & Surname: |  |
| Organisation: |  |
| Unit/Dept: |  |
| Facilitator Name: |  |
| Date Started: |  |
| Date of Completion: |  |

**Copyright**

All rights reserved. The copyright of this document, its previous editions and any annexures thereto, is protected and expressly reserved. No part of this document may be reproduced, stored in a retrievable system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise without the prior permission.

# Key to Icons

The following icons may be used in this Learner Guide to indicate specific functions:

|  |  |
| --- | --- |
| **http://www.duluth.lib.mn.us/Images/BookStack.gif**  **Books** | **This icon means that other books are available for further information on a particular topic/subject.** |
| http://www.rpsrelocation.com/_borders/checklist.jpg  **References** | **This icon refers to any examples, handouts, checklists, etc…** |
| http://www.school-portal.co.uk/GroupDownloadAttachment.asp?GroupId=21353&AttachmentID=1300079**Important** | **This icon represents important information related to a specific topic or section of the guide.** |
| **http://cloud.graphicleftovers.com/11976/item34004/Cartoon-exercise-book.jpgActivities** | **This icon helps you to be prepared for the learning to follow or assist you to demonstrate understanding of module content. Shows transference of knowledge and skill.** |
| http://3.bp.blogspot.com/_0EodaYtqevU/TMun5XOj03I/AAAAAAAAAIU/lzrnWelQjgc/s1600/group-discussion.jpg**Exercises** | **This icon represents any exercise to be completed on a specific topic at home by you or in a group.** |
| **http://edtech.kennesaw.edu/intech/images/rubric.gif**  **Tasks/Projects** | **An important aspect of the assessment process is proof of competence. This can be achieved by observation or a portfolio of evidence should be submitted in this regard.** |
| **http://tell.fll.purdue.edu/JapanProj/FLClipart/Adjectives/busy.gifWorkplace Activities** | **An important aspect of learning is through workplace experience. Activities with this icon can only be completed once a learner is in the workplace** |
| http://blog.mindjet.com/wp-content/uploads/2010/01/helpful_tips_image.jpg**Tips** | **This icon indicates practical tips you can adopt in the future.** |
| http://school.discoveryeducation.com/clipart/images/read.gif**Notes** | **This icon represents important notes you must remember as part of the learning process.** |

# Learner Guide Introduction

|  |  |
| --- | --- |
| **About the Learner Guide…** | This Learner Guide provides a comprehensive overview of the **Describe the difference between programming in Object Orientated and Procedural Languages,**and forms part of a series of Learner Guides that have been developed for **FURTHER EDUCATION AND TRAINING CERTIFICATE: INFORMATION TECHNOLOGY: SYSTEMS DEVELOPMENT  ID 78965 LEVEL 4 – CREDITS 165** The series of Learner Guides are conceptualized in modular’s format and developed **FURTHER EDUCATION AND TRAINING CERTIFICATE: INFORMATION TECHNOLOGY: SYSTEMS DEVELOPMENT ID 78965 LEVEL 4 – CREDITS 165**They are designed to improve the skills and knowledge of learners, and thus enabling them to effectively and efficiently complete specific tasks. Learners are required to attend training workshops as a group or as specified by their organization. These workshops are presented in modules, and conducted by a qualified facilitator. |
| **Purpose** | The purpose of this Unit Standard is to **Describe the difference between programming in Object Orientated and Procedural Languages** |
| **Outcomes** | **Describe the difference between programming in Object Orientated and Procedural Languages** |
| **Assessment Criteria** | The only way to establish whether a learner is competent and has accomplished the specific outcomes is through an assessment process. Assessment involves collecting and interpreting evidence about the learner’s ability to perform a task. This guide may include assessments in the form of activities, assignments, tasks or projects, as well as workplace practical tasks. Learners are required to perform tasks on the job to collect enough and appropriate evidence for their portfolio of evidence, proof signed by their supervisor that the tasks were performed successfully. |
| **To qualify** | To qualify and receive credits towards the learning programme, a registered assessor will conduct an evaluation and assessment of the learner’s portfolio of evidence and competency |
| **Range of Learning** | This describes the situation and circumstance in which competence must be demonstrated and the parameters in which learners operate |
| **Responsibility** | The responsibility of learning rest with the learner, so:   * Be proactive and ask questions, * Seek assistance and help from your facilitators, if required. |

Learning Unit1

**UNIT STANDARD NUMBER :** 149909

**Describe the difference between programming in Object Orientated and Procedural Languages**

**LEVEL ON THE NQF :** 4

**CREDITS :** 4

**FIELD :** Physical, Mathematical, Computer and Life Sciences

**SUB FIELD :**  Construction Information Technology and Computer Sciences

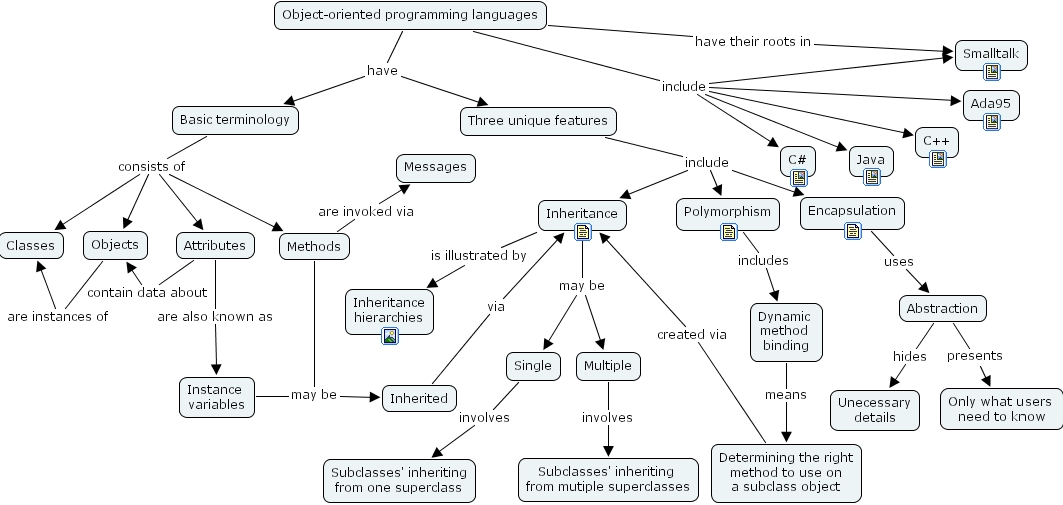
|  |  |
| --- | --- |
| **PURPOSE:** | This unit standard is intended:  to provide a conceptual knowledge of the areas covered  for those entering the workplace in the area of systems development  as additional knowledge for those wanting to understand the areas covered  People credited with this unit standard are able to:  describe basic object oriented terminology  describe the fundamental differences between procedural and object oriented programming.  The performance of all elements is to a standard that allows for further learning in this area. |
| **LEARNING ASSUMED TO BE IN PLACE:** | |
| The credit value of this unit is based on a person having the prior knowledge and skills to:  be able to apply the principles of Procedural Computer Programming | |

|  |
| --- |
| **SESSION 1.**  **Describe basic object oriented terminology.** |
| **Learning Outcomes** |
| * 1.The description explains the basic principles of a class. * 2. The description explains the basic principles of an object. * 3. The description explains the basic principles of information hiding and encapsulation * 4. The Description explains the basic principles of inheritance. * 5. The Description explains the principles of polymorphisme. |

**Introduction**

**Object-oriented programming**

Object-oriented programming is a programming paradigm that uses abstraction to create models based on the real world. It uses several techniques from previously established paradigms, including modularity, polymorphism, and encapsulation. Today, many popular programming languages (such as Java, JavaScript, C#, C++, Python, PHP, Ruby and Objective-C) support object-oriented programming (OOP).Object-oriented programming may be seen as the design of software using a collection of cooperating objects, as opposed to a traditional view in which a program may be seen as a collection of functions, or simply as a list of instructions to the computer. In OOP, each object is capable of receiving messages, processing data, and sending messages to other objects. Each object can be viewed as an independent little machine with a distinct role or responsibility. Object-oriented programming is intended to promote greater flexibility and maintainability in programming, and is widely popular in large-scale software engineering. By virtue of its strong emphasis on modularity, object oriented code is intended to be simpler to develop and easier to understand later on, lending itself to more direct analysis, coding, and understanding of complex situations and procedures than less modular programming methods.[2](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript#Reference)



**Terminology**

**Class**

Defines the characteristics of the Object.

**Object**

An Instance of a Class.

**Property**

An Object characteristic, such as color.

**Method**

An Object capability, such as walk.

**Constructor**

A method called at the moment of instantiation.

**Inheritance**

A Class can inherit characteristics from another Class.

**Encapsulation**

A Class defines only the characteristics of the Object, a method defines only how the method executes.

**Abstraction**

The conjunction of complex inheritance, methods, properties of an Object must be able to simulate a reality model.

**Polymorphism**

Different Classes might define the same method or property.

* **The description explains the basic principles of an object.**

**How People Approach Object-Oriented Technology**

Object-oriented technology is both immense and far-reaching. End users of computer systems and computer-based systems notice the effects of object-oriented technology in the form of increasingly easy-to-use software applications and operating systems and in more flexible services being provided by such industries as banking, telecommunications, and cable television. For the software engineer, object-oriented technology encompasses object-oriented programming languages, object-oriented development methodologies, management of object-oriented projects, object-oriented computer hardware, and object-oriented computer aided software engineering, among others. It is not surprising, therefore, that there is some confusion regarding object-oriented terms and concepts. In this article, we will provide the reader with working definitions for object-oriented terms and concepts that are necessary for a reader to acquire a basic understanding of object-oriented technology.

Many of the terms commonly used in object-oriented technology were originally used to describe object-oriented programming (coding) concepts. Specifically, although the terms were borrowed from a non-computer-software perspective, they were first used extensively to describe concepts embodied in object-oriented programming languages, such as Smalltalk, C++, and Eiffel. However, these terms are quite useful even if one never intends to write any software at all. For example, an industrial modeler could create an object-oriented model of a plastics manufacturing facility. Molding machines, plastic parts, and even the "recipes" (proportional combinations) of the chemicals used to create the various plastics could all be described in object-oriented terms. Further, dynamic and static relationships among these items could also be described in object-oriented terms. Finally, keep in mind that there is no one ultimate set of definitions for object-oriented terms and concepts. Depending on who you are talking to, terms and definitions will vary slightly. This is normal; in different parts of the United States, the same breakfast item might be referred to as a pancake, a griddle cake, a flapjack, or a hot cake. Even in technical arenas, this variation in terminology is common. A chemist might use the terms "valance" and "oxidation state" to identify the same concept.

**Object-Oriented Terms and Concepts**

**Objects**

**Objects** are the physical and conceptual things we find in the universe around us. Hardware, software, documents, human beings, and even concepts are all examples of objects. For purposes of modeling his or her company, a chief executive officer could view employees, buildings, divisions, documents, and benefits packages as objects. An automotive engineer would see tires, doors, engines, top speed, and the current fuel level as objects. Atoms, molecules, volumes, and temperatures would all be objects a chemist might consider in creating an object-oriented simulation of a chemical reaction. Finally, a software engineer would consider stacks, queues, windows, and check boxes as objects.

Objects are thought of as having state. The **state** of an object is the condition of the object, or a set of circumstances describing the object. It is not uncommon to hear people talk about the "state information" associated with a particular object. For example, the state of a bank account object would include the current balance, the state of a clock object would be the current time, the state of an electric light bulb would be "on" or "off." For complex objects like a human being or an automobile, a complete description of the state might be very complex. Fortunately, when we use objects to model real world or imagined situations, we typically restrict the possible states of the objects to only those that are relevant to our models. We also think of the state of an object as something that is internal to an object. For example, if we place a message in a mailbox, the (internal) state of the mailbox object is changed, whereas the (internal) state of the message object remains unchanged. Sometimes people think of objects as being strictly static. That is, the state of an object will not change unless something outside of the object requests the object to change its state. Indeed, many objects are passive (static). A list of names does not spontaneously add new names to itself, nor would we expect it to spontaneously delete names from itself. However, it is possible for some objects to change their own state. If an object is capable of spontaneously changing its own state, we refer to it as an "**object with life**." (Objects with life are sometimes also called "active objects" or "actors.") Clocks and timers are common examples of objects with life. If we were modeling a business process, we would recognize that salespeople and customers were also objects with life.

**The description explains the basic principles of a class.**

**Classes, Metaclasses, Parameterized Classes, and Exemplars**

There are two broad categories of objects: classes and instances. Users of object-oriented technology usually think of classes as containing the information necessary to create instances, i.e., the structure and capabilities of an instance is determined by its corresponding class. There are three commonly used (and different) views on the definition for "class":

|  |  |
| --- | --- |
| * A **class** is a pattern, template, or blueprint for a category of structurally identical items. The items created using the class are called **instances**. This is often referred to as the "class as a `cookie cutter'" view. As you might guess, the instances are the "cookies." * A **class** is a thing that consists of both a pattern and a mechanism for creating items based on that pattern. This is the "class as an `instance factory'" view; **instances** are the individual items that are "manufactured" (created) using the class's creation mechanism. * A **class** is the set of all items created using a specific pattern. Said another way, the class is the set of all **instances** of that pattern. | http://www.ipipan.gda.pl/~marek/objects/TOA/oobasics/InstFac.gif |

In this article, we will use the definition of a "class an `instance factory.'"

We should note that it is possible for an instance of a class to also be a class. A **metaclass** is a class whose instances themselves are classes. This means when we use the instance creation mechanism in a metaclass, the instance created will itself be a class. The instance creation mechanism of this class can, in turn, be used to create instances -- although these instances may or may not themselves be classes.

A concept very similar to the metaclass is the parameterized class. A **parameterized class** is a template for a class wherein specific items have been identified as being required to create non-parameterized classes based on the template. In effect, a parameterized class can be viewed as a "fill in the blanks" version of a class. One cannot directly use the instance creation mechanism of a parameterized class. First, we must supply the required parameters, resulting in the creation of a non-parameterized class. Once we have a non-parameterized class, we can use its creation mechanisms to create instances. In this article, we will use the term "class" to mean metaclass, parameterized class, or a class that is neither a metaclass nor a parameterized class. We will make a distinction only when it is necessary to do so. Further, we will occasionally refer to "non-class instances." A **non-class instance** is an instance of a class, but is itself not a class. An instance of a metaclass, for example, would *not* be a non-class instance.

In this article, we will sometimes refer to "instantiation." **Instantiation** has two common meanings:

* as a *verb*, instantiation is the process of creating an instance of a class, and
* as a *noun*, an instantiation is an instance of a class.

Some people restrict the use of the term "object" to instances of classes. For these people, classes are not objects. However, when these people are confronted with the concepts of metaclasses and parameterized classes, they have a difficulty attempting to resolve the "problems" these concepts introduce. For example, is a class that is an instance of a metaclass an object -- even though it is itself a class? In this article, we will use the term "object" to refer to both classes and their instances. We will only distinguish between the two when needed.

**Black Boxes and Interfaces**

Objects are "black boxes." Specifically, the underlying implementations of objects are hidden from those that use the object. In object-oriented systems, it is only the producer (creator, designer, or builder) of an object that knows the details about the internal construction of that object. The consumers (users) of an object are denied knowledge of the inner workings of the object, and must deal with an object via one of its three distinct interfaces:

* the "public" interface. This is the interface that is open (visible) to everybody.
* the "inheritance" interface. This is the interface that is accessible only by direct specializations of the object. (We will discuss inheritance and specialization later in this chapter.) In class-based object-oriented systems, only classes can provide an inheritance interface.
* the "parameter" interface. In the case of parameterized classes, the parameter interface defines the parameters that must be supplied to create an instance of the parameterized class.

Another way of saying that an item is in the public interface of an object is to say that the object "exports" that item. Similarly, when an object requires information from outside of itself (e.g., as with the parameters in a parameterized class), we can say that the object needs to "import" that information.

* **The description explains the basic principles of information hiding and encapsulation**

**Encapsulation vs. Information Hiding**

How is encapsulation related to information hiding? You can think of it as two ways of refer- ring to the same idea. Information hiding is the goal, and encapsulation is the technique you use to accomplish that goal. **Encapsulation** can be defined as the hiding of internal data representation and imple- mentation details in an object. The only way to access the data within an encapsulated object is to use defined operations. By using encapsulation, you are enforcing information hiding. Many object-oriented languages use keywords to specify that methods and attributes should be hidden. In Java, for instance, adding the private key word to a method will ensure that only code within the object can execute it. There is no such keyword in JavaScript; we will instead use the concept of the closure to create methods and attributes that can only be accessed from within the object. It is more complicated (and confusing) than just using keywords, but the same end result can be achieved.

Encapsulation seems to be a combination of one or more of:

* Grouping of relating things together
* GateKeeper (state or data protection)

Information Hiding, on the other hand, is

* Hiding details of implementation

**Overview** .The term ***encapsulation* i**s often used interchangeably with information hiding. Not all agree on the distinctions between the two though; one may think of information hiding as being the principle and encapsulation being the technique. A software module hides information by encapsulating the information into a module or other construct which presents an interface. A common use of information hiding is to hide the physical storage layout for data so that if it is changed, the change is restricted to a small subset of the total program. For example, if a three-dimensional point (*x*,*y*,*z*) is represented in a program with three floating point scalar variables and later, the representation is changed to a single array variable of size three, a module designed with information hiding in mind would protect the remainder of the program from such a change. In object-oriented programming, information hiding (by way of nesting of types) reduces software development risk by shifting the code's dependency on an uncertain implementation (design decision) onto a well-defined interface. Clients of the interface perform operations purely through it so if the implementation changes, the clients do not have to change.

**The Description explains the basic principles of inheritance.**

**Specialization and Inheritance**

Aggregation is not the only way in which two objects can be related. One object can be a specialization of another object. **Specialization**is either:

* the process of defining a new object based on a (typically) more narrow definition of an existing object, or
* an object that is directly related to, and more narrowly defined than, another object.

Specialization is usually associated with classes. It is usually only in the so-called "classless" object-oriented systems that we think of specialization for objects other than classes. Depending on their technical background, there are a number of different ways in which people express specialization. For example, those who are familiar with an object-oriented programming language called Smalltalk refer to specializations as "subclasses" and to the corresponding **generalizations** of these specializations as "superclasses." Those with a background in the C++ programming language use the term "derived class" for specialization and "base class" for corresponding generalizations. It is common to say that everything that is true for a generalization is also true for its corresponding specialization. We can, for example, define "checking accounts" and "savings accounts" as specializations of "bank accounts." Another way of saying this is that a checking account is a kind of bank account, and a savings account is a kind of bank account. Still another way of expressing this idea is to say that everything that was true for the bank account is also true for the savings account and the checking account. In an object-oriented context, we speak of specializations as "inheriting" characteristics from their corresponding generalizations. **Inheritance** can be defined as the process whereby one object acquires (gets, receives) characteristics from one or more other objects. Some object-oriented systems permit only **single inheritance,** a situation in which a specialization may only acquire characteristics from a single generalization. Many object-oriented systems, however, allow for **multiple inheritances**, a situation in which a specialization may acquire characteristics from two or more corresponding generalizations.

|  |  |
| --- | --- |
| http://www.ipipan.gda.pl/~marek/objects/TOA/oobasics/inheritance.gif | Our previous discussion of the bank account, checking account, and savings account was an example of single inheritance. A telescope and a television set are both specializations of "device that enables one to see things far away." A television set is also a kind of "electronic device." You might say that a television set acquires characteristics from two different generalizations, "device that enables one to see things far away" and "electronic device." Therefore, a television set is a product of multiple inheritance. |

**Inheritance**

One important characteristic of object-oriented languages is inheritance. *Inheritance* refers to the capability of defining a new class of objects that inherits from a parent class. New data elements and methods can be added to the new class, but the data elements and methods of the parent class are available for objects in the new class without rewriting their declarations.

For example, Java uses the following syntax for inheritance:

public class B extends A {

*declarations for new members*

}

Objects in class B will have all members that are defined for objects in class A. In addition, they have the new members defined in the declaration of class B. The extends keyword signals that class B inherits from class A. We also say that B is a *subclass* of A and that A is the *parent class* of B.

In some languages, Java for example, the programmer has some control over which members are inherited. In Java, a member is defined with a keyword indicating its level of accessibility. The keyword private indicates that the member is *not* inherited by subclasses. This capability is not often used.

[an error occurred while processing this directive]

**The Description explains the principles of polymorphism**

**Polymorphism and Overloading**

*Polymorphism* refers to the capability of having methods with the same names and parameter types exhibit different behavior depending on the receiver. In other words, you can send the same message to two different objects and they can respond in different ways. More generally, the capability of using names to mean different things in different contexts is called *overloading*. This also includes allowing two methods to have the same name but different parameters types, with different behavior depending on the parameter types. Note that a language could support some kinds of overloading without supporting polymorphism. In that case, most people in the object-oriented community would not consider it to be an object-oriented language. Polymorphism and overloading can lead to confusion if used excessively. However, the capability of using words or names to mean different things in different contexts is an important part of the power of natural languages. People begin developing the skills for using it in early childhood.

**Members**

Objects can have their own data, including variables and constants, and their own methods. The variables, constants, and methods associated with an object are collectively refered to as its *members* or *features*.

**Classes**

Many object-oriented languages use an important construction called a class. A *class* is a category of objects, classified according to the members that they have. Like objects, classes can also be implemented in classical languages, using separate compilation and structs for encapsulation. The object-oriented language Java uses the following syntax for class definitions:

public class A {

*declarations for members*

}

Each object in the class will have all members defined in the declarations.

**Class Members and Instance Members**

In many object-oriented languages, classes are objects in their own right (to a greater or lesser extent, depending on the language). Their primary function is as factories for objects in the category. A class can also hold data variable and constants that are shared by all of its objects and can handle methods that deal with an entire class rather than an individual object. These members are called *class members* or, in some languages (C++ and Java, for example), *static members*. The members that are associated with objects are called *instance members*.

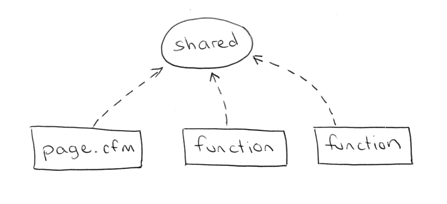
|  |
| --- |
| **SESSION 2.**  **Describe the fundamental differences between procedural and object oriented programming.** |
| **Learning Outcomes** |
| * 1. The description explains the use of functions and variables in structure programming, using simple examples. * 2. The description compares encapsulation of data and functions in objects versus procedural programming * 3. The Description identifies possible classes for simple examples. |

**Procedural vs Object Oriented Programming**

ColdFusion started its life as a procedural language and only in more recent times gained object oriented features. As a result of this history there is a substantial number of procedural ColdFusion systems in existence today. Considering this, it's worth taking a brief look at what it means to write procedural code and then see how this differs from an object oriented approach. To make this comparison we need to first consider the problem that both approaches help us to solve. When programming any system you are essentially dealing with data and the code that changes that data. These two fundamental aspects of programming are handled quite differently in procedural systems compared with object oriented systems, and these differences require different strategies in how we think about writing code.

**Procedural programming**

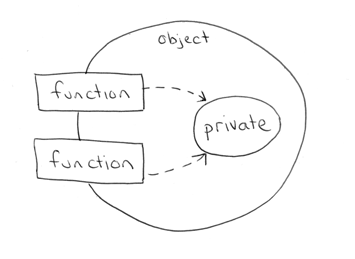
In procedural programing our code is organised into small "procedures" that use and change our data. In ColdFusion, we write our procedures as either custom tags or functions. These functions typically take some input, do something, then produce some output. Ideally your functions would behave as "black boxes" where input data goes in and output data comes out. The key idea here is that our functions have no intrinsic relationship with the data they operate on. As long as you provide the correct number and type of arguments, the function will do its work and faithfully return its output. Sometimes our functions need to access data that is not provided as a parameter, i.e., we need access data that is outside the function. Data accessed in this way is considered "global" or "shared" data.



So in a procedural system our functions use data they are "given" (as parameters) but also directly access any shared data they need.

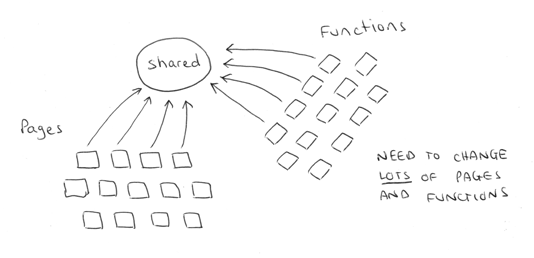
**Object oriented programming**

In object oriented programming, the data and related functions are bundled together into an "object". Ideally, the data inside an object can only be manipulated by calling the object's functions. This means that your data is locked away inside your objects and your functions provide the only means of doing something with that data. In a well designed object oriented system objects never access shared or global data, they are only permitted to use the data they have, or data they are given.

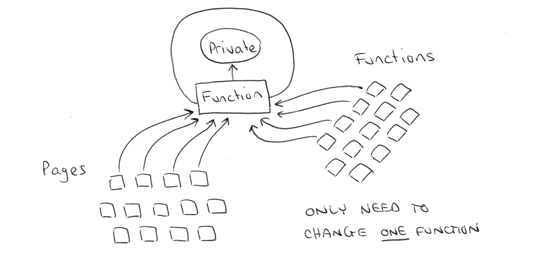


**Global and shared data**

We can see that one of the principle differences is that procedural systems make use of shared and global data, while object oriented systems lock their data privately away in objects. Let's consider a scenario where you need to change a shared variable in a procedural system. Perhaps you need to rename it, change it from a string to a numeric, change it from a struct to an array, or even remove it completely. In a procedural application you would need to find and change each place in the code where that variable is referenced. In a large system this can be a widespread and difficult change to make.



In an object oriented system we know that all variables are inside objects and that only functions within those objects can access or change those variables. When a variable needs to be changed then we only need to change the functions that access those variables. As long as we take care that the functions' input arguments and output types are not changed, then we don't need to change any other part of the system.



**The cost of OO**

Object oriented design is complicated to do well, and a substantial amount of time is likely to be required to learn it in depth. If you have been developing procedural systems for some time then object oriented concepts will require learning a different way of thinking which is always challenging and requires effort. However the time to learn is not the only cost. Once you start learning, you may start to question yourself time and time again if you are writing code "correctly". Your productivity may be affected as you try different ideas, aiming for a good object oriented solution. A further cost to consider is not specific to OO, but is specific to OO within ColdFusion. You may read many object oriented articles and books but you cannot apply their teachings blindly in your ColdFusion applications. There is a performance factor associated with creating objects in ColdFusion so applying many of the pure object oriented ideas can adversely affect your application. This then adds an additional challenge in knowing when not to apply some object oriented ideas.

**A Real-World Example**

Okay, that's enough theory. We're going to put both types of programming to the test with a real-world example. Let's say that you are working for a vehicle parts manufacturer that needs to update its online inventory system. Your boss tells you to program two similar but separate forms for a website, one form that processes information about cars and one that does the same for trucks.

For cars, we will need to record the following information:

* Color
* Engine Size
* Transmission Type
* Number of doors

For trucks, the information will be similar, but slightly different. We need:

* Color
* Engine Size
* Transmission Type
* Cab Size
* Towing Capacity

In procedural programming, you would write the code first to process the car form and then the code for the truck form. With object-oriented programming, you would write a base class called vehicle that would record the common characteristics what we need from both trucks and cars. In this case, the vehicle class will record:

* Color
* Engine Size
* Transmission Type

We'll make each one of those characteristics into a separate method. The color method, for example, could take the color of the vehicle as a parameter and do something with it, like storing it in a database. Next, we will create two more classes: truck and car, both of which will inherit all of the methods of the vehicle class and extend it with methods that are unique to them. The car class will have a method called number Of Doors and the truck class will have the methods cab Size and towing Capacity. Okay, so let's assume that we have a working example for both procedural and OO programming. Now, let's run through a few scenarios that we could come across in a normal working environment. You know the type of scenario because it always begins with the thought: I really wish my boss didn't send this in an email request at 4pm on a Friday afternoon.

**Scenario 1**

Suppose that we suddenly need to add a bus form, that records the following information:

* Color
* Engine Size
* Transmission Type
* Number of passengers

*Procedural:* We need to recreate the entire form, repeating the code for Color, Engine Size, and Transmission Type.

*OOP:* We simply extend the vehicle class with a bus class and add the method, number Of Passengers.

**Scenario 2**

Instead of storing color in a database like we previously did, for some strange reason our client wants the color emailed to him.

*Procedural:* We change three different forms: cars, trucks, and buses to email the color to the client rather than storing it in the database.

*OOP:* We change the color method in the vehicle class and because the car, truck, and bus classes all extend (or inherit from, to put it another way) the vehicle class, they are automatically updated.

**Scenario 3**

We want to move from a generic car to specific makes, for example: Nissan and Mazda.

*Procedural:* We create a new form for each make, repeating all of the code for generic car information and adding the code specific to each make.

*OOP:* We extend the car class with a nissan class and a mazda class and add methods for each set of unique information for that car make.

**Scenario 4**

We found a bug in the transmission type area of our form and need to fix it.

*Procedural:* We open and update each form.

*OOP:* We fix the transmission Type method in the vehicle class and the change perpetuates in every class that inherits from it.

**Wrapping It Up**

As you can see from the above scenarios, employing an OOP style has significant advantages over procedural programming, especially as your scale increases. Consider the savings we would receive from OOP in terms of repeated code, flexibility, and maintenance if we also had to add forms for boats, motorcycles, planes, go-karts, ATVs, snowmobiles, etc. Objects and methods are also far easier to test than procedural programming by using unit testing to test results. Does this mean that you should never use procedural programming? Not necessarily. If you're doing a mockup or a proof-of-concept app, you might not have the time to make everything object-oriented and so I think it might would be better to use procedural programming for a prototype, but it would be best to make the production product in an OO-manner.

This has been just a brief foray into a very large subject, but I hope that you've been able to get a better understanding of procedural vs. object-oriented programming and when and how to use each. If this tutorial has been helpful to you, please bookmark it for your reference and share it with your friends and colleagues.

We can summarize the differences as follows :

• **Procedural Programming**

– top down design

– create functions to do small tasks

– communicate by parameters and return values

• **Object Oriented Programming**

– design and represent objects

– determine relationships between objects

– determine attributes each object has

– determine behaviours each object will respond to

– create objects and send messages to them to use or manipulate their attributes

