# Introduction

What is the difference between a class and an object or variable. To truly understand this there are two parts of a computer program that need to be addressed first. The compiler or interrupter and how a program is stored in a computer. This paper intends to cover compilers and memory before addressing the difference between class and object.

# Compilers and Interpreter

Before a program can be run, it must first be compiled and turned into machine code. An Interpreter is basically a compiler that runs at the same time as the program and compiles the program as it runs. One of the jobs of a compiler is to determine which parts of the source code are executable statements, which one define data types, and which ones are variables.

The primary difference between a compiler and an interpreter is the output. The compiler output is a file[[1]](#footnote-2) that can be executed by itself over and over again. An interpreter output is a program that executes within the interpreter and only runs once. Each time a script is to be executed, it must be run through the interpreter again and again.

Compiled programs run faster than interpreted programs, up to a 1000 times faster. So why not use just compiled programs written in languages like Java, C++, and FORTRAN. Scripted programs have the advantage that it is faster to change a script and see if it runs than a compiled language. Also, with small programs or scripts, the difference of execution time may be too short to notice the difference.

# Memory

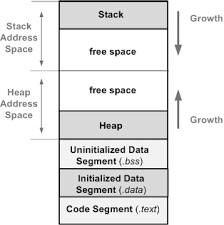
An executable program, one that has already been compiled, is stored in memory. Normally, it is stored in a disk file in a shorten form. The shorten form will only include the static parts of the program, the machine code, static data such as strings and constants, along with data that the program will use during execution that can be determined and set by the compiler.

Each unit of memory has an address used to locate it. In most computers, a unit of memory is a byte which is itself comprised of 8 bits[[2]](#footnote-3). There are some computers that use a word as its unit of memory. A word is comprised of a number of bytes. In the old days, the size of a word could vary from computer to computer. The size of a word is usually based on the number of data lines the computer data bus had for transporting data. Older PCs had a 16 bit data bus and its word size was 2 bytes. Today, the standards are 32 bit and 64 bit data buses where the word size can range from 4 bytes to 8 bytes.

There have been other computers that use unique word sizes. The Harris H-Series had a word size of 24 bits or 3 bytes[[3]](#footnote-4). A CDC series computer had a word size of 60 bits or 10 6 bit bytes[[4]](#footnote-5). The CDC computers were used before 32 bit data buses were common.

## Program Layout

When a compiled program is about to be executed, it is first copied from disk into memory. More memory will be assigned to the program than was needed to store it in a disk file. That extra memory is used for dynamic objects and call stacks.

Figure 1: Executable Virtual Memory

The compiled program, also called the executable, references memory starting at zero. When it is placed in memory, its first memory address will be a random value that is not zero. To get around this problem, an executable will be assigned virtual memory. Virtual memory is stored in real memory with the addresses in virtual memory starting at zero. The program address its virtual memory space and the operating system will translate virtual memory addresses into real memory addresses.

Looking at Figure 1, to the right, the first set of addresses store the machine code of the program. This is referred to as the Code Segment. The next two sections contain data, both initialized and uninitialized. The order of these two data segments can be reversed or even mixed. These segments contain the data that does not change size: constants, strings, predefined global variables.

The rest of the virtual memory is used to store dynamic data and call stacks. This is data that is created while the program is running. This segment is divided into three parts: the Heap at one end, the Stack at the other end, and free or unused memory in the middle.

The Stack is used to store variables and objects used that are not created using the **new** keyword. It also stores the call stack whenever execution jumps to a function or method. The call stack contains the return value from the method, the parameters passed to the method, and other information needed by the computer to jump to the correct machine code and return when the method is done. All variables declared in the scope of the meyhod are also pushed onto the stack. When execution leaves a scope, all the variables declared in that scope are popped off the Stack. When the function exits, the return value is placed in the call stack. All remaining variables used by the function are popped off the Stack. The caller of the function can then access the return value before it too can be popped off the stack.

The Heap is used to used to store dynamically created variables and objects. In Java, any object created with the **new** keyword is a dynamic variable. Memory from the Heap is allocated starting with the beginning of the Heap. If there are no free or unused chunk of memory within the Heap that is large enough to contain the variable, the Heap will grow and allocate memory that was from the free space.

The Stack is a memory where all the memory is used. In other words, there are no chunks of unused memory in the Stack. The Stack grows, increases in size, as needed and it shrinks, decreases in size, as data is no longer used. The Heap, on the other hand, cannot grow and shrink like the Stack because dynamic variables can still exist when the function that created them returns. As dynamic variables are released, the memory used to store them is marked as available. This results in holes in the Heap where dynamic memory has been released and no new dynamic variables have been created that will fit in the available space.

There is a potential problem with using this Stack/Heap model. Each program is allocated a fixed amount of memory to use for the Stack and Heap. If this memory runs out, in other words, the Stack meets the Heap, the program will crash. The actual size allocated for the Stack and Heap can be changed though not when the program is executing. Sometimes, the Stack/Heap space can be defined as a command line parameter when the program is run. Sometimes, it can be specified when compiling the program, and at other times, with changes to the operating system configuration. Regardless of how Stack/Heap space can be changed, if your program does not have enough, it will crash and lose all the data that is still in memory.

# Classes

Java is know as a Typed Language. This means that data types are defined, some by the language and others by the programmer that are know as User Types.

## Primitive Types

In Java, everything is derived from the class Object. There are some types that are not really objects because they are considered primitive. Examples would be byte, short, int, long, char, boolean, float, and double. There are primitive classes used to contain primitive variables. Some examples of them are PrimitiveType, NumericType, IntegerType, and FloatingPointType.

## Non-Primitive Types

Beyond the primitives, there are lots of other types defined by Java and its extensive library. A prime example would String. A String holds an array of char and may encapsulate the length of the array or the number of characters actually in the array.

## Class Types

In Java source code, the programmer can define their own types. These are called user-defined data type or UDT and also just User Type. All user defined data types are classes. Not all classes are defined using the **class** keyword. The **enum** keyword is used to build special kinds of classes that define a finite set of named values. We will ignore enumeration types for another paper since all that applies to classes applies to enumerations as well.

A class defines a data structure that contains zero or more data members and zero or more methods. A class definition in the code is instructions to the compiler on how to build and use the data structure. When the compiler encounters a variable of a class, the compiler generates code that does the following:

1. Allocate memory to hold the object either from the Stack or the Heap.
   1. In the case of constants and global variables, the memory will come from the initialized or uninitialized data segment.
2. Call the constructors for each data member to initialize them[[5]](#footnote-6).
3. Calls the class constructor to complete the initialization[[6]](#footnote-7).
4. Returns a reference that is the virtual address of the object.

Once the object has been created and initialize, execution returns to the next statement in the code.

The information to perform these 4 steps are stored in code and in the initialized data segment by the compiler. In the data segment, information such as what type of each data member is as well as a list of static data members and a list of methods and static methods. The program has access to this information at run time and can use the information for various tasks.

The class also defines the methods of the class. Like the information stored about data members, the compiler will store information about each method contained in the class definition. At a minimum, this information will include the signature of the method. The signature consists of the method name, the return type, a list of the data types of each parameter, and the memory location where the method begins execution. Therefore, two methods in a class can have the same name so long as they have different signatures.

A class requires one or more constructors. Constructors are methods called when an instance of a class is created. If no constructors are explicitly defined in the class, a default constructor that takes zero parameters will be generated by the compiler. Hence, every class has at least one method even when it defines none.

A class does not consume any Stack or Heap space. Defining a class does consume some memory. Each method in the class uses memory in the code segment. Static data members will be created in the initialized data segment. Static data members are allocated space in the data segment because only one instance of the static data member exists regardless of how many instances of a class are created.

## Objects and Variables

In Java, an object and a variable are often used as interchangeable terms. For this paper, we will be using the term Object to mean any data, both simple and structured. The term variable will be discussed in a later section in how a variable and an object are not exactly the same thing.

An object is created based on a type: a primitive type or a class. Objects of primitive types, such as int and double, may use shortcuts to bypass come of the more complex concepts of a data structure. These shortcuts are hidden in the code by the compiler so we can think of primitive types the same way we think about classes.

A class by itself cannot be used by the program[[7]](#footnote-8). In order for the data structure defined by the class to be used, an object or instance of the class must first be created. This process is know as creating an instance of a class. This can be done in several ways in Java. It can be done as a local variable within a scope. It can be created as a dynamic object using the **new** keyword.

An instance of a class requires memory in either the Stack or the Heap[[8]](#footnote-9). The big difference between a class and an instance of that class, a.k.a. an object of that class, is that the class is the blueprint used to create an object and the object is what contains data. Another way to put it is that an object has state which refers to its data. Two objects of the same class can be in different states if they are not identical copies of each other, have different data.

In other words, if your program wants to use a class, it must first create an instance or object of that class. That object is the memory that contains a set of data [members] and can manipulate that data using the class methods.

### Static Data, Methods, and Classes

Normally, a class cannot not be used to access anything. There is an exception that is in accessing it’s static data members and static methods. Only one instance or object of each static data member exists and it can be accessed using the format **className.dataMemberName**. The access privilege assigned to the static data member (public, protected, private) determines who can access the data members directly.

Likewise, static methods of a class can be called using the format **className.methodName()**. The difference between a static method and a non-static method in a class is that the non-static or regular methods are passed a hidden parameter, **this**, which is used to access all non-static data members in the object. Remember that each instance of a class or object contain their own set of those data members also known as state.

There is also the case where a class can be defined as static. A static class can only contain static data members and static methods. This is a classic technique for defining global variables in a Java program. No instances can be created of a static class.

## Variables

This section is one of nitpicking definitions. For most purposes, a variable and an object can be treated as the same thing. It is only when peeking under the hood do the two terms have different meanings. Everything revolves around the use of memory. Every unit of memory has an address that defines its location in the array of memory. An address is needed to locate and access any object. In C++ and C, the address is stored in a type called a Pointer. In C++ and C, the program has direct access to the address in the pointer and can manipulate it. For instance, take an array of Type T, where Type is any class, the address to the first element of the array is provided to the program. To figure out the address of the Nth element in the array, where N starts at 0 for the first element, use the following formula:

**address = N\*sizeof(Type)**[[9]](#footnote-10)

which is equivalent to **T[N]**

In Java, pointers and addresses are consider too dangerous to use and are therefor hidden from the programmer. Pointers and address are hidden but are still used.

The name of an object is the variable that references it. For instance, take Type foobar. If Type is not a primitive type, foobar will actually contain the address of the object of Type. If foobar is an array of Type, foobar contains the address or pointer to the first element of the array.

### Passing Parameters

There are two ways in which parameters can be passed to a method or function. One is by value and the other is by reference. In the first case, a completely new object is created on the Stack. This involves calling the class constructor and making a copy of all the data members. When passing by reference, only the address of the object is pushed onto the Stack.

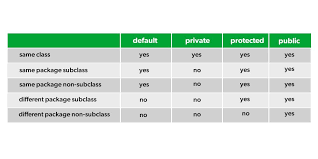
Passing by reference is faster than passing by value, especially if the class constructor is complicated. Passing by value insures that anything done to the parameter in the method does not change the original object in the calling method. There are trade offs to be considered when deciding to pass an parameter by value or reference.

In Java, all parameters are passed by value. The value passed is whatever is stored in the variable. If that is a primitive data type, then the value of the data type is duplicated. If the variable is to a complex object, such as an array, the variable contains the reference and the reference is duplicated.

## Access Modifiers

In every class definition, the class itself, all of its data members, and all of its methods must be assigned an access level[[10]](#footnote-11) by using an access modifier. If no access level is specified, a default level will be assigned. There are three access modifiers used in Java: public, protected, and private.

Figure 2 shows the three different access modifiers and how they affect access to data and members within an object. Basically, the access levels are:

Figure 2: Java Access Modifiers

* Accessed directly by any code within the class itself.
* Access directly by any subclass[[11]](#footnote-12).
* Access directly by any other class in the same package.
* Access by any other class regardless of package.

# Conclusion

The terms Class and Object are often used interchangeable even though they are two distinct things. The Class is the blueprint of the type while an object is an instance of the type. A Class uses no Stack or Heap space. An Object uses either Stack or Heap space.

Another way to look at is that a Class defines an Object and an Object is an individual container of data and methods that is accessed by the executing program.

1. The actual output of a compiler is often a binary file or an object file that contains the compiled code. This file must be run through a program called a Linker that resolves calls to functions and libraries. We are ignoring the linker for now and assuming that the output from the compiler has already been linked. [↑](#footnote-ref-2)
2. Some older computers like the CDC computers used different size bytes than what is standard today. [↑](#footnote-ref-3)
3. The H-Series computer used addresses for each word in memory. In order to address an individual byte, the top 2 bits in the address was used to address the byte. This computer could have a NULL address (address 0) while one of the upper 2 bits could be non-zero. Therefore, NULL and zero were not interchangeable. [↑](#footnote-ref-4)
4. The CDC computer was designed for high accuracy floating point math, hence the large word size. The disadvantage with the 6 bit bytes was that its strings could only hold upper case letters. [↑](#footnote-ref-5)
5. If a data member is declared with no value, the constructor for that data member is not called and it is either set to NULL or left undefined. [↑](#footnote-ref-6)
6. If the class is a child class, the constructor of the parent class is called as the first statement in the child class constructor even though it is not explicitly stated. [↑](#footnote-ref-7)
7. There are exceptions to this in the case of static classes, static data members and static methods which will be discussed in a later section. [↑](#footnote-ref-8)
8. There are exceptions to this. An instance of a class that is static or a global variable can exist in the data segment of the program’s virtual memory. [↑](#footnote-ref-9)
9. The **sizeof**(type or object) returns the number of bytes an instance of that type uses. [↑](#footnote-ref-10)
10. The same is true for classes defined within another class as well. [↑](#footnote-ref-11)
11. Note. There is another distinction for access if the subclass is defined within the same package or in a different package. [↑](#footnote-ref-12)