Computer Programming and Software Development Story

A computer program consists of a set of instructions directing the computer hardware to perform some desired operations. A set of interrelated programs is called software. Hardware consists of physical components for performing computations, and software consists of instructions stored within a computer guiding it through a sequence of computations for some desired task. With different software the same computer hardware can perform different tasks. It means that a suitably general computing machine can emulate any specific ones. A computer program is nothing more than a means of turning a general-purpose computer into a special-purpose one.

In the first half of the last century, Alan Turing proposed a theoretical mechanical programming engine, known as the Turing Machine. This machine had an infinitely long tape, an internal register storing its state, and a table of actions. At each step, it would read the symbol from the current location on the tape and consult the table to find what it should do for that symbol given the current state of the engine. It would then perform some or all of the following actions:

- Write a new symbol.
- Change the state in the internal register.
- Move the tape left or right.

The first computers were highly specialized machines. Due to the source of their funding, they were focused heavily on running a set of simple algorithms that were used for code breaking. Whenever the algorithm (or, in many cases, the input) changed, the computers needed to be rewired.

It was a little while later that stored program computers emerged, such as the Manchester Baby. Like the Turing Machine, these computers stored the algorithms they were to compute in the same way they stored data. These early machines were programmed in pure machine code. The operations that the computer would perform were represented by short binary sequences, and programmers would enter them either by flipping switches, making holes in punch cards or tapes, or pressing buttons.

Instead of binary sequences, most systems enabled programmers to enter short sequences as a single octal or hexadecimal digit, but this still wasn't ideal.

The binary system, whether coded in octal or hexadecimal digits, wasn't very human-friendly, so the idea of a symbolic assembler arose. Rather than entering the binary codes directly, programmers would enter mnemonics that represented them. While an add operation might be 01101011, the programmer would enter ADD, which was much easier to remember.
These assembly language sequences had a simple one-to-one mapping with machine code instructions, so a simple program comprising a lookup table was all that was required to turn them into real code.

One of the biggest innovations introduced by symbolic assemblers was that of symbolic branch destinations. Most programs involve large numbers of conditional statements: do one thing if a value is in a certain range; otherwise, do something else.

At the machine-code level, they are translated into jumps, either relative or absolute, which move the place from which the next instruction is read, either to a specific location or to a certain offset from the current one.

A machine code programmer had to calculate these offsets and enter them in the program as fixed numbers. If the programmer wanted to add another instruction somewhere, all jumps that ended after this new instruction (or backward relative jumps from after it to before) needed to be updated.

With a symbolic assembler, jumps could be given symbolic names, and the assembler would convert these names into real addresses when it ran. If you added a new instruction somewhere, you still needed to run the assembler again, but it would take care of the jump updates for you. This made programs a lot more flexible. It also made them slightly more efficient.

The only kind of flow control available in a simple assembly language is the jump. As Edsger Dijkstra famously wrote in his paper "GOTO Considered Harmful," this is not a very good way of structuring programs. The proposed solution to this was to introduce subroutines (also often called functions or procedures) and a stack for flow control. A subroutine is a block of code that you would jump to, and then have control returned to the place of the call afterward. The first way of implementing them was to write the return address just before the first instruction in the subroutine. This is very easy to do, but there is a problem: there is no way for a subroutine to call itself. The solution was to use a stack, which is a simple data structure in which values are retrieved in the opposite order to their insertion. When you were about to jump to a subroutine, you would push the return address onto a stack and then jump to the start of the subroutine. At the end of the subroutine, it would get the top value from the stack and jump to it. This allowed a procedure to call itself as many times as you have stack space to store addresses. Some architectures, such as x86, include instructions for performing these operations.

This idea was extended to enable other values to be stored on the call stack. If a subroutine needed some private space, it would move the location of the top of the stack up a bit, and use the gap. This turned out to be a spectacularly bad idea because it meant that small bugs in a program could make it easy for specially crafted input to overwrite the return address. A better solution would have been to use a separate stack, but this has not proved to be popular. The stack is also used for passing parameters to and from functions.

The archetypal stack-based language is Forth, which provides separate stacks for data and flow control. In Forth, the language itself is stack-based, resembling reverse polish notation. Subroutines in Forth are known as *words* because this is how they are represented in the code. The interpreter simply scans a line of input, pushing the values onto the stack or executing a subroutine when a word is encountered.

The ability to add words to Forth makes it well suited to meta-programming because new words (subroutines) are indistinguishable from the core language to a programmer, giving rise to an extensible programming language.

A related construct that gained some popularity at the same time was the co-routine. While a subroutine is good for two pieces of code where one is subordinate to the other, co-routines work better when there is not such a clear division between the two. Co-routines are often used to simulate parallelism. While most modern languages don't have support for co-routines directly, they are common at the framework level for GUI programming, in which events are tied to particular actions. These event handlers are run to completion by some form of run loop, which is often hidden from the developer, giving the appearance of events happening independently.

A macro assembler allowed symbols to be defined representing sequences of instructions. Things such as a function prologue and epilogue—the sequence of instructions needed at the start and end of a subroutine—could be replaced by single symbols, reducing the amount of typing required. Programmers began accumulating collections of macros for common activities, and their code eventually began to resemble high-level descriptions of the algorithm. These sets of macros gradually evolved into complete languages. One thing that macros could not do is free the programmer from the need to manually allocate registers.

Most modern architectures can only perform computations on values stored in registers, of which there are typically between 8, 32, or 64 bits long. A programmer must load values from memory into registers, perform the calculations, and then store them back out to memory. Another drawback of assembly language programming was that code was still tied to a specific instruction set, even with a heavy use of macros. This made moving from one architecture to another a significant investment.

The first high-level languages, such as FORTRAN, solved both of these problems. The compiler would handle register allocation, and the only thing that would need to be rewritten for a new architecture was the compiler. The compiler itself could be written in a high-level language, so only the code generation part needed to be rewritten, and the rest just needed to be recompiled on the new version.

One debate as old as programming languages has to do with when a type should be assigned to a value. Most modern CPUs treat memory as a huge blob of un-typed bytes, while registers are either integer or floating-point values (a few further differentiate between pointers and integers). This means that it's up to the program to determine the type of a value before loading it into a register. If a value is a 16-bit integer, you need to use a 16-bit load operation with an integer register as a destination operand, for example. There are two general ways in which this decision can be made. The simplest to implement is the typed-variable mechanism (also called strongly typed), in which each symbolic name for a value has an associated type. With typed-variable languages, the compiler can determine the type at compile time. This is very popular with compiler writers because it makes their job a lot easier. Within this family, there are explicitly and implicitly typed languages. The former, such as Java, requires the programmer to assign a type to each variable. The latter, such as Haskell, determines the type from what is done with the value. For example, a variable used to store the result of a function that returns an integer will be assumed to be an integer.

At the other extreme are languages such as Smalltalk and Lisp, which use the typed-value approach (also called loosely, weakly, or dynamically typed languages). They keep track of types at runtime, which can make programs considerably simpler because it’s possible to write very generic code. Typed-variable languages typically do this with a template mechanism or through poly-typing. The downside of this flexibility is runtime overhead. Before you can do any work on a value, you have to determine its type.

Procedures are a way of tidying up program structure, but they are also very similar, conceptually, to mathematical functions. The difference between functions and procedures relates to state.

A mathematical function has no concept of global state, while a procedure does (and can reference global program state as well as its own).

If you add restrictions to procedures—forcing them to behave as functions—you can then use mathematical reasoning to prove that aspects of your program are correct. This is the foundation for languages such as Ocaml and Haskell. Haskell, for example, is a pure functional language—everything is a function. Haskell has no global state, so the result of a function depends solely on its input, as with a mathematical function. This has a couple of interesting side effects from the perspective of efficiency. The first is that execution order becomes less important. If you have already computed the arguments to a function, you can compute the result whenever you want. If you have computed the arguments to two functions, you can run them both in parallel because no global state means no side effects. You also get the idea of lazy evaluation. If you don’t use the return value of a function, you don’t need to bother computing it. This has been used to produce some simple programs that run on very large data sets and only bother loading the parts that are accessed. The program is written as though it performs a computation on the whole dataset, but only results that are used are executed, so only data needed to compute them is loaded.

In the late ’80s, Japan embarked on a program to build a next-generation programming language—a fifth-generation language, or 5GL for short.

The idea of any programming language is to allow the programmer to provide a high-level description and have the compiler turn it into a program. The 4GL concept is based on the idea that any sufficiently well-specified problem is a solution to that problem. Instead of providing a set of algorithms to solve the problem, the developer would produce a detailed description, and the compiler would turn it into a program. The language that emerged from this project was a variant of Prolog, although initial work in Prolog had begun in the early ’70s. While an interesting language in its own right, and very useful in certain AI-related activities, it has failed as a general-purpose language.

In 1969, Alan Kay and others at Xerox PARC proposed a mechanism for decomposing a programming problem into smaller chunks that could be solved independently. These small programs would then run on simulated small computers, which communicated via an abstract message-passing interface. The simple computers were termed "objects" in the system, and the combination of dynamic typing and message passing was dubbed "Object Oriented Programming" by Alan Kay, as embodied by the Smalltalk language. Smalltalk itself was based loosely on Simula, as part of a bet that it was possible to create a language embodying the message-passing idea from Simula in a single page of code.

Just as in Haskell, in which everything is a function, in Smalltalk everything is an object. At the simplest level, things like integers are objects. You can send messages to integers for simple things such as addition, but you also get more complex ones used for flow control. Since blocks of code are also objects, they can be passed as arguments with messages, so conditional execution is performed by sending a message to (for example) an integer, containing another integer and a block of code, which is executed if the two integers have the same value. Smalltalk used a method of refinement known as inheritance. Objects were defined by sending messages to a special kind of factory object, called a class. Classes (themselves instances of meta-classes) created instances of objects according to a recipe.

When you wanted to create a new kind of object, you would copy an existing class, send messages to it by adding new methods and instance variables to the objects it would create, and then use this template to create new objects. This seems a long way away from the original machine code

A language similar to Smalltalk, Objective-C adds Smalltalk-like extensions to C, which is about as low-level as a high-level language can be—little more than a cross-platform assembler. C provides two kinds of compound data types: arrays and structures. Arrays are strings of variables of a single type (in C, a text string is an array of characters that are 8-bit integers). Structures are heterogeneous collections with a fixed layout. When C is compiled, elements in an array are found by multiplying the size of an individual element with the index, and elements in a structure are located by adding a fixed offset to the address of the start of the structure. These simple compound types are used to implement Objective-C. The first implementations of the language comprised two components: a pre-processor that would emit C code and a runtime library that handled the dynamic features. In more modern implementations the pre-processor is omitted, and the code is generated directly without a C intermediate form.

Classes in Objective-C are represented by a simple structure that contains a list of instance variable to offset mappings and a list of selectors to function mappings. A selector is simply an abstract representation of the method name.

Given an object, it is possible to get the class by inspecting the first element in the structure and then look up the methods. How are methods implemented? As plain C functions with a specific signature. The first argument of a method is a pointer to the object on which the method is being called, the second is the selector, and the remainder are the arguments passed to the method.

Objective-C makes some compromises for ease of implementation. Unlike Smalltalk, it also supports "primitive types" that are not objects. In Objective-C, a 16-bit integer is an intrinsic type that does not respond to messages and needs to be manipulated directly with C integer expressions. In Smalltalk, the lowest two bits of a pointer are usually used to indicate the type. If they have a specific value, the value is treated as an integer (after right-shifting by two) and has some special handling. This adds a small amount of overhead since it’s necessary to check the type of a value before sending it a message. It also means that integers can be compared for equivalence in exactly the same way as objects in the virtual machine (by pointer comparison), without needing a special case.

Smalltalk introduced a simple method of creating objects via factories known as classes. Self, first introduced in 1986, went in a slightly different direction by using prototypes. In Self, instead off defining the object and then instantiating it, you would clone and object and then add methods and instance variables to it. This is also possible in Smalltalk, but the language isn't really designed for it. This style, known as prototype-based, object-oriented programming is now one of the most widespread programming paradigms due to the fact that it was chosen by Netscape in the mid-1990s as the model for its JavaScript scripting language. Since then, the use of JavaScript, later standardized as ECMAScript, has found its way into a huge number of web pages and web applications.

Prototype-based programming is generally considered more flexible than class-based programming, and is often more understandable. When a specific behaviour is needed in only a small number of instances of an object, it is much easier to only add it to this small number in prototype-based languages. Class-based languages typically require more code to be written to create a specialized version of a class, which encourages bloat in a smaller number or a large spread of classes that are hard to read. The focus on inheritance in class-based languages makes it much harder to add different, orthogonal, sets of behaviour to a group of objects (multiple inheritance helps here, but comes with its own problems). Prototype-based languages, with their dynamically changing types, are much harder to compile to heavily optimized code. This is becoming less important, as the wide use of JavaScript has shown, but is still a concern in some areas.

The developers of the Lisaac language attempted to address this by introducing static typing to a prototyped language, which allows a huge body of research into optimizing languages like C to be applied.

Smalltalk implements message passing as an indirect function call. From a certain perspective, it can be seen that any function call is in fact a special case of a message-sending operation—specifically a synchronous message-passing operation. The caller sends a message to the one who called and then waits for it return. A simple extension to this to support parallelism. The concept of futures allows parallelism to be simply added to a lot of existing algorithms. (This was touched on briefly in the functional programming discussion.) The concept behind a future is that you should not block execution of the caller when the function is called, but instead when the return value is needed.

Consider the Quicksort algorithm, which partitions a set of data and then runs recursively on both parts. A simple functional implementation of this would perform the pivot, run recursively on one sub-array, run again on the other, and then return. The two recursive calls, however, do not interfere with each other (this is trivial to prove in a functional language, and fairly easy in other languages). A clever compiler could run both recursive calls in separate threads and wait for the return.

Sufficiently clever compilers are quite hard to come by, but some languages make it easy to implement this model in the library.

In Objective-C, for example, it is possible to write a generic piece of code that spawns an object in a new thread, and executes messages sent to it asynchronously, returning a proxy object that blocks whenever a message is sent to it.

Languages such as Erlang go a step further and expose asynchronous message passing directly to the developer. This is not conceptually harder than synchronous messaging, but can be difficult to grasp for people who have a lot of experience with synchronous programming. In Erlang, the processes and messages are integral parts of the language. Creating a new process is a very cheap operation, as is sending and receiving a message. While Erlang code tends to be slower on a single processor than other languages, the fact that it is very easy to write code that scales to tens or hundreds of processors makes up for it in a number of areas. Languages such as Erlang are still in their infancy, but asynchronous programming is likely to grow in the next few years as the number of processors in the average computer increases.

Synchronous programming tends to cause performance problems on parallel systems due to the overhead of locking, while an asynchronous system can be implemented using lockless data structures for communication.

The one requirement for a good parallel programming language, which is missing from most of the languages discussed here, is that it must distinguish between aliased and mutable data.
If data is allowed to be both aliased (that is, multiple threads or processes have references to it) and mutable, there are a large number of optimizations that are impossible, and locking is required for safe access.

If the language (or, at least, the library) can enforce this restriction, parallel programming becomes a lot easier. Erlang does this in a very simple way; all data is immutable with the exception of a dictionary associated with a process (which is mutable, but rarely used).
This is the sort of solution a compiler writer would think of; it makes implementation easy at the cost of some ease of use. Erlang inherited this single-assignment form from a family of languages known as dataflow programming languages. They view programs as a directed graph through which data flows.

This model fits well with parallel programming in a lot of cases because each filter in the graph can execute concurrently. This model is common in visualization, and simple versions are found in most media programming frameworks.

Web programming has introduced a lot of people to programming models that were previously consigned to niches, and server-side scripting languages have brought in more.
While the ’90s were dominated by C-with-syntactic-sugar languages, this is slowly changing as more people discover more flexible programming styles. Hopefully this trend will continue, making the next 10 years an even more fun time to be a programmer.

Program Scripts

A scripting language allows one to write compact useful programs (where source really can be modified by a person without much effort. Scripting languages such as : REXX, TCL, Perl, Python, PHP and Javascript belong to a class called very high level languages (VHL).

The key advantage of a scripting language is common for any VHL language: it is the compactness of the code, the compactness that gives a possibility to write the same applications as a fraction (sometimes 1/10) lines of code in comparison with traditional compiled (C, Pascal) or semi-compiled (Java) strongly typed languages. A typical scripting language allow to shrink the number of lines of code for a typical application several times means, and that means lower development costs, lower the amount of bugs and potentially better architecture.

As for the number of lines (or more correctly lexical tokens) to express a particular algorithm Java looks like C. Java designers tried to create "C++ done right." but "done right" in not enough. It is essentially C++ with garbage collection and without features like pointers and memory allocation. Automatic memory allocation and garbage collection is now standard feature of most modern languages so nothing new here. While Java managed to displace Cobol, it did it not without some help of pure luck and huge amount of money spent by Sun and IBM to promote the language and to create the necessary infrastructure.

Of course, there is no free lunch and sometimes you pay the price for using VHL instead of plain vanilla high level languages, but with the current much higher speed CPUs and large amounts random access memory (RAM) on desktops (and even laptops), it is not an unreasonable price for probably 80-95% of the most applications. Small critical part can always be rewritten in lower level language.

With simple scripting languages, users struggle less with the language and can devote more time to the task itself. Perl and other more complex scripting languages have a steeper learning curve but can serve as a more suitable tool for professionals. More compact and cleaner code that is achievable by using scripting languages often helps to achieve higher quality in applications development.

A language should adhere to principle of "least surprise" and do not break with previous languages (and first of all C/C++ family as the dominant family of languages) unless it is justified by some gains in power or transparency.

Simplicity and transparency of connecting to high level language (C or C++) is important, particularly in large projects. Components can be produced in other language, the language that has lower level and which is more flexible in operation on the level of detail required for some of them. And the larger the project is, the more components are specialized enough to benefit from implementation in the second language. Here TCL, a scripting language created by John Ousterhout, is really good but Python and ruby should be considered too. Both provide clean interface to C++ and C respectively. The advantage of Microsoft .NET framework is that it permits using the same runtime engine for multiple languages. The same advantage can be achieved using scripting langue that is compiled into JVM.

Quality and availability of "connectors" that permit using OS API (both built-in in the language and external libraries)) might make 80% of the usability of the language in a large and complex programming projects. In this (limited) sense libraries are more important then the language itself. And it takes a lot of time (or money or both) for a language to get quality libraries.

Paradoxically debugging for scripting language can be more complex then in mainstream languages like C/C++ for which the level of language is lower and the tools are definitely more mature, feature rich and often commercially supported. Debugging tools available even for popular scripting languages such as Perl, PHP and Python are still rather crude. This has a real impact when working on non-trivial programs. Paradoxically for complex application development the quality of the debugger is often as important as the quality of the language implementation. It's is actually an important part of the quality of the language implementation. It is not accidental that Donald Knuth, who probably is one of the greatest computer scientists of all times, preferred to work with the language that is best integrated with the OS and has the best debugger. For a viable scripting language, the debugger should be part of the language design and the key part of the implementation not an afterthought. Scripting language designers are still slow to realize this shift of the paradigm. In this area significant progress is needed. IDE environments like Active State Komodo can help too (I can attest that they manage to eliminate problems that haunted earlier, versions and version 3.1 and later are usable for Perl).

While each of the scripting language has innovative features in the design, the strong points that helped wide adoption of the language, in certain areas, each of existing scripting languages has problems that need to be recognized and rectified.

Typically, scripting languages are typeless. While this is definitely a more reasonable compromise than type safely straitjacket of Java it can create some additional problem which can easily be rectified by high quality cross reference tool, name space diagrams, pretty printers etc. Actually exactly because of weak typing, high quality cross-reference tools should be considered as a part of any decent scripting language implementation, not an add-on tool. Scripting languages are evolving and are becoming more and more competitive with Java and C++ for developing mainstream enterprise applications.

It's still unclear which scripting language will prevail in a long run, therefore right now one should probably diversify and experiment with several of them. Moreover, if they are all using .NET or JVM, then different languages can be optimal for different parts of the project. But still any large project should have the "principal" language, the language that you feel best match the majority of the project's needs. It's just impossible to learn several scripting languages to an equal degree. I currently consider Perl to be my primary scripting language, but there is no JVM based implementation of Perl and that affects scalability. I also use Python for tasks that benefit from co-routines. Python also has a distinct advantage of having a JVM-based implementation (Jython). Still Python puts more restrictions than Perl and in this sense is a little bit lower level language. Python's innovating "indentation reveals real block structure" solution partly compensates for that as it produces more vertically compact programs. Moreover, you can choose your style of braces and prettyprinting as it is easy (and probably necessary) to imitate C-style curvy braces using comments and a pretty printer. In this sense the Python is the most modern language, the language where the editor in IDE should contain pretty printer by default.

TCL and REXX are probably the most underappreciated scripting languages in existence. Both have almost zero learning curve. REXX syntax was by-and-large borrowed from PL/1 and is more readable, while TCP has really minimalistic syntax. REXX was the first language that served as both macro language and regular scripting language. It implements a very innovative approach: any command that is not recognized by REXX interpreter is considered an application command (if it used as a macro language for an application, for example an editor) or shell command. TCL can be used for programming-in-large and C for programming in the small. Few people understand that TCL+C is underappreciated and a unique development technology. Generally, combining any scripting language with clean interface to C or C++ (or any other suitable high-level language) and C/C++ is a very powerful software development paradigm.

Combination of JVM based scripting language and Java is also promising development method that can compensate several weaknesses of Java as a system implementation language. Several classic scripting languages now have Java-based implementation (Jython is one such example) and there are also active developments on new scripting languages explicitly designed to be macro languages for Java.

In dual-language (.NET, scripting language + C, or scripting language +Java) implementations each language offers support that is useful in non trivial ways.

Perl and Python can be considered as attempts to provide a "compromise" language that is usable for both *programming in the large* and *programming in the small*. Here Python has an important advantage: unlike Perl, Python has more or less usable interface to C++, so it can be used for dual language programming, although such cases are still infrequent (Python philosophy is generally that same as Perl). Despite difficulties with the managing huge and very complex interpreters both Perl and Python have a very strong following and nothing succeed like success. It's probably wise to use both languages when appropriate. None is a silver bullet that solves all the software-engineering problems.

One test of whether someone is a good programmer is to ask him about the shortcomings of the tools he uses. Watch if he talks only about language constructs. He/she probably is a mediocre programmer. Programming language environment (language + IDE + debugger + libraries) is as important or more important then the language itself. Someone who does not understand that flaws and limitation of their favourite language can be compensated by the environment, who cannot view the language as a part of a larger development environment, is either unable to think analytically and thus cannot be a good programmer, or is blindly partisan (i.e. a zealot) like many participants of Perl vs. Python debate; but please note that even the worst participant of Perl vs. Python debate is usually heads above participants of Linux vs. Windows advocacy wars...

There is a general trend toward more expressive, "very high level" solutions, the trend that drove Perl into prominence to continue. It is this trend that launched LAMP (Linux-Apache-MySQL-Perl/Python/PHP) tool set into prominence. Here neither Linux not MySQL play a significant role. For this reason, LAMP should probably more correctly called WDS (Web server-database-scripting language). Solaris, FreeBSD or even Windows can be used instead of Linux with the same tool set. The same is true for MySQL, which is just one database out of several possibilities.

With the maturity of the WEB that was the major driving force behind the scripting languages, days of great surprises and surprise winners (as for example PHP victory over Perl in WEB site scripting) are over. Despite being open source efforts, the development of scripting languages now became a cruel, unforgiving area ruled by the merciless dynamics of the marketplace. Of course, there are other valuable scripting languages like REXX, Icon, Scheme and Ruby and they also deserve study and might be successfully used for certain projects.

The "big five" likely to stay are briefly described below.

JavaScript: main attraction is a very clean syntax and the ability to be used both as macro language and as regular shell (actually JavaScript is an underutilized as shell in Windows environment despite the fact that it is available for many years via WSH). It also has superior object model. JavaScript is the only mainstream language that uses a prototype-based OO implementation. With typical class-based OO languages like C++ and Java objects come in two general types. *Classes* are templates that define set of variables and methods for the object, and *instances* are populated classes -- "usable" objects with memory allocated and values of variables filled. They cannot be extended dynamically at run time. In prototype-based OO the structure of the object is dynamic at run time and new objects are mainly constructed via c*loning* by copying the variables and methods of an existing object (its prototype). And the new object can be modified dynamically (extended) without affecting its parent. Reverse is not generally true.

Perl: is an interesting attempt to create "the next generation shell". It probably is the easiest to learn and use for Unix system administrators who are already familiar with Unix shell programming environment. Despite convoluted semantic, absence of co-routines, inability to use pipes for connecting loops and subroutines, Perl is a great language with many interesting and non-trivial ideas (for example very flexible set of control statements, powerful open statement where one can use pipes, etc).

PHP: Pioneered higher level of integration with web server and database. PHP was the first scripting language which successfully addressed the need for higher level integrated tool set for Web site developers on Unix. It was designed explicitly as an integral part of the Webserver/database/scripting language troika often called LAMP. This proved to be a very powerful toolset, suitable for solving wide range of tasks. That's why PHP successfully deposed already entrenched Perl in this particular area. This seamless integration with MySQL database and Apache WEB-server really makes PHP an important pioneer in the scripting language world. Although initially it was an open source server-side HTML-embedded scripting language, it is evolving beyond its HTML roots into more advanced services like remote procedure calls.

Python: is probably the best of the breed of the important class of languages that support co-routines (Icon supports co-routines too, but it is much less known). It has Google support so this is the only scripting language with a multi-million-dollar corporation behind it. Python is now compiled into Pycode. It's relatively easy (in comparison with Perl) to mix and match Pycode and regular C or C++. In this area only TCL has more transparent "dual language" programming model. Recenly, Sun hired Jruby developers so thing might change but still Python has the most of corporate support so far. Microsoft’s recent IronPython implementation shows that it get some traction outside traditional scripting community. Jython (JPython) has a unique advantage for Java and is "politically correct" scripting language for a large enterprise development. It allows to use Java explicitly as the "programming in the small" language. It has the ability to extend existing Java classes, optional static compilation (allows creation of applets, and servlets), beans, bean properties and makes the usage of Java packages much easier than any other scripting language. Still the development felled behind mainstream Python and there is no significant financial support to launch the project on a new level.

TCL: primary importance is that from the very beginning is was designed with dual-language paradigm of programming in mind. It has the best integration with C (that was the design goal). It promoted important view that we need to distinguish the use of scripting language as a glue for components (shell style usage) and its use as an application macro-language. Along with REXX TCL shares the achievement of being the first universal macro-language**.** And it gave us such a wonderful tool as Expect. TCL was the first scripting language having a simple and clean interface with C.

A programmer's real role is tool making for commonly used problem-solving tasks rather than simply using the tools available in a programming language.

Let us embark on that role.

Choice of MATLAB for Program and Software Development

Computer languages are usually designed with the solution of a certain range of problems in mind. The selection of the right language for the job at hand is of utmost importance. MATLAB is generally used if the problem to be solved is conveniently represented by matrices, solved using operations from linear matrix algebra, and presented using relatively simple two- and three-dimensional graphics. Computing the solution to a family of linear equations, and representing, manipulating, and displaying engineering data are perhaps the two best examples of problems for which MATLAB is ideally suited.

Not only is the MATLAB programming language exceptionally straightforward to use (every data object is assumed to be an array), but the MATLAB program code will be much shorter and simpler than an equivalent implementation in C or FORTRAN or Java. MATLAB is therefore an ideal language for creating prototypes of software solutions to engineering problems, and using them to validate ideas and refine project specifications. Once these issues have been worked out, the MATLAB implementation can be replaced by a C or Java implementation that enhances performance, and allows for extra functionality – for example, a fully functional graphical user interface that perhaps communicates with other software package over the Internet.

Since the early 1990s the functionality of MATLAB has been expanded with the development of toolboxes containing functions dedicated to a specific area of mathematics or engineering. Toolboxes are now provided for statistics, signal processing, image processing, neural nets, various aspects of nonlinear and model predictive control, optimization, system identification, and partial differential equation computations. MATLAB comes with an Application Program Interface that allows MATLAB programs to communicate with C and FORTRAN programs, and vice versa, and to establish client/server relationships between MATLAB and other software program.

The following is an updated hyperlinked list of available toolboxes for MATLAB.

- Statistics and Machine Learning Toolbox™
- Curve Fitting Toolbox™
- Control System Toolbox™
- Signal Processing Toolbox™
- Mapping Toolbox™
- System Identification Toolbox™
- Deep Learning Toolbox™
- DSP System Toolbox™
- Datafeed Toolbox™
- Financial Toolbox™
- Image Processing Toolbox™
- Text Analytics Toolbox™
- Predictive Maintenance Toolbox™

Program Development with M-Files

A common mode of using MATLAB is typing a command and getting the results. This mode of operation is suitable only for the specification of the small problems, perhaps using a handful of MATLAB commands or less. A much better problem-solving approach is to use a text editor to write the commands in an M-file, and then ask MATLAB to read and execute the commands listed in the M-file. This section describes two types of M-files. Script M-files correspond to a main program in programming languages such as C. Function M-files correspond to a subprogram or user written function in programming languages such as C. An M-file can reference other M-files, including referencing itself recursively.

User-Defined Code and Software Libraries

Computer programs written in MATLAB are a combination of user-defined code – that is, the computer program code we write ourselves – and collections of external functions located in software libraries or MATLAB toolboxes. Software library functions are written by computer vendors, and are automatically bundled with the compiler. Software libraries play a central role in the development of C programs because the small number of keywords and operators in C is not enough to solve real engineering and scientific problems in a practical way. What really makes C useful is its ability to communicate with collections of functions that are external to the user-written source code. This is where much of the real work in C programs takes place.

The ANSI C standard requires that certain libraries are provided with all implementations of ANSI C. For example, the standard library contains functions for I/O, manipulation of character strings, handling of run-time errors, dynamic allocation and de-allocation of memory, and functions for C program interaction with the computer's operating system. Mathematical formulae are evaluated by linking a C program to the math library. Engineering application programs may also communicate with graphical user interface, numerical analysis, and/or network communications libraries.

Generally speaking, if there is a library function that meets your needs, by all means use it. The judicious use of library functions will simplify the writing of your C programs, shorten the required development time, and enhance C program portability. Software is said to be portable if it can, with reasonable effort, be made to execute on computers other than on the one on which it was originally written.

Scripts versus Functions in MATLAB

Scripts are m-files containing MATLAB statements. MATLAB ``functions" are another type of m-file. The biggest difference between scripts and functions is that functions have input and output parameters. Script files can only operate on the variables that are hard-coded into their m-file. As you can see, functions are much more flexible. They are therefore more suitable for general purpose tasks that will be applied to different data. Scripts are useful for tasks that don't change. They are also a way to document a specific sequence of actions, say a function call with special parameter values, that may be hard to remember.

There are more subtle differences between scripts and functions. A script can be thought of as a keyboard macro: when you type the name of the script, all of the commands contained in it are executed just as if you had typed these commands into the command window. Thus, all variables created in the script are added to the workspace for the current session. Furthermore, if any of the variables in the script file have the same name as the ones in your current workspace, the values of those variables in the workspace are changed by the actions in the script. This can be used to your advantage. It can also cause unwanted side effects.

In contrast, function variables are local to the function. (The exception is that it's possible to declare and use global variables, but that requires and explicit action by the user.) The local scope of function variables gives you greater security and flexibility. The only way (besides explicitly declared global variables) to get information into and out of a function is through the variables in the parameter lists.

Scripts are useful for setting global behaviour of a MATLAB session. This includes any terminal settings for a remote serial line, or setting the parameters of the Figure window for a certain size plot.

Sometimes a script is a useful starting point in developing a MATLAB function. When I'm starting to write a new function, but I'm uncertain about the command syntax or the actual sequence of commands I want to use, I'll use the diary command to record my tests. After I get the correct command sequence, I close the diary file and open it with a text editor. This gives me a jump start on the function development because I don't have to re-enter the commands. I just delete the incorrect lines, add the function definition, and insert variables.

I've witnessed inexperienced MATLAB users who have become dependent on scripts when functions would ultimately be easier to use. This typically occurs when someone gets stuck on the function definition syntax, especially on the use of input and output parameters. Rather than figure out how to properly pass parameters to the function they repeatedly edit their script files to simulate the effect of variable arguments. I hope this hypertext reference can help you avoid that fate.

There is nothing wrong with using scripts, of course. Scripts and functions are two tools for working with MATLAB programming. Using the appropriate tool for the job will help you achieve your analysis goals more easily.

Anatomy of a MATLAB function

MATLAB functions are similar to C functions or Fortran subroutines.

MATLAB programs are stored as plain text in files having names that end with the extension ``.m''. These files are called, not surprisingly, m-files. Each m-file contains exactly one MATLAB function. Thus, a collection of MATLAB functions can lead to a large number of relatively small files.

One nifty difference between MATLAB and traditional high-level languages is that MATLAB functions can be used interactively. In addition to providing the obvious support for interactive calculation, it also is a very convenient way to debug functions that are part of a bigger project. MATLAB functions have two parameter lists, one for input and one for output. This supports one of the cardinal rules of MATLAB programming: *don't change the input parameters of a function*. Like all cardinal rules, this one is broken at times. My free advice, however, is to stick to the rule. This will require you to make some slight adjustments in the way you program. In the end this shift will help you write better MATLAB code.

Creating function m-files with a plain text editor

MATLAB m-files must be plain text files, i.e. files with none of the special formatting characters included by default in files created by word-processors. Most word-processors provide the option of saving the file as plain text, (look for a ``Save As...'' option in the file menu). A word-processor is overkill for creating m-files, however, and it is usually more convenient to use a simple text editor, or a ``programmer's editor''. For most types of computers there are several text editors (often as freeware or shareware). Usually one plain text editor is included with the operating system.

When you are writing m-files you will usually want to have the text editor and MATLAB open at the same time. Since modern word-processors require lots of system RAM it may not even be possible or practical (if you are working on a stand-alone personal computer) for you to use a word-processor for m-file development. In this case a simple, text editor will be your only option.

Function Definition

The first line of a function m-file must be of the following form.

```
function [output_parameter_list] = function_name(input_parameter_list)
```

The first word must always be ``function''. Following that, the (optional) output parameters are enclosed in square brackets [ ]. If the function has no output_parameter_list the square brackets and the equal sign are also omitted. The function_name is a character string that will be used to call the function. The function_name must also be the same as the file name (without the ``.m'') in which the function is stored. In other words, the MATLAB function, ``foo'', must be stored in the file, ``foo.m''. Following the file name is the (optional) input_parameter_list.

There can exactly be one MATLAB function per m-file.

Input and Output parameters

The input_parameter_list and output_parameter_list are comma-separated lists of MATLAB variables.

Unlike other languages, the variables in the input_parameter_list should never be altered by the statements inside the function**.** Expert MATLAB programmers have ways and reasons for violating that principle, but it is good practice to consider the input variables to be constants that cannot be changed. The separation of input and output variables helps to reinforce this principle. The input and output variables can be scalars, vectors, matrices, and strings. In fact, MATLAB does not really distinguish between variables types until some calculation or operation involving the variables is performed. It is perfectly acceptable that the input to a function is a scalar during one call and a vector during another call.
To make the preceding point more concrete, consider the following statement in MATLAB terminal (screen) interface:

```
>> y = sin(x)
```

which is a call to the built-in sine function. If x is a scalar (i.e. a matrix with one row and one column) then y will be a scalar. If x is a row vector, then y will be a row vector. If x is a matrix then y is a matrix. (You should verify these statements with some simple MATLAB calculations.)

This situation-dependence of input and output variables is a very powerful and potentially very confusing feature of MATLAB.

Comment statements

MATLAB comment statements begin with the percent character, %. All characters from the % to the end of the line are treated as a comment. The % character does not need to be in column 1.

Program Development Cycle

For a novice programmer, the two most important issues are likely to be about learning the syntax of the language and becoming familiar with the step-by- step details of planning, writing, compiling, running, testing, and documenting small programs.

The following are some basic steps:
1. Create source code file(s) using an editor.
2. Type source code
3. Review source code file to correct errors
4. Prepare input data
5. Execute program with direct input or from a file
6. Receive output

An M-file can be prepared using the MATLAB editor, or commonly used text editors such as Notepad or WordPad After the list of commands has been typed into the file, it can be saved by clicking on save as in the edit menu. The file can be executed from the command window by typing the M-file name without the .m extension.

Consider the following example of program saved as a file named Newton.m:
```
% Calculate square root of a given number by Newton's method
% Read input
x=input('Please type a number for calculating square root:');
if x<=0
display('The number must be positive'); end;
% Initial guess.
xstart = x/2;
% Iteration loop to compute square root.
for i = 1:100
xnew = (xstart + x/xstart)/2;    % new estimate of square root.
display(xnew);% print xnew.
if abs(xnew - xstart)/xnew < eps
        break % on convergence.
end;
xstart = xnew; % update estimate of square root.
end
```

Running this program In MATLAB
```
>> Newton
% MATLAB response
Please type a number for calculating square root:20
xnew =  6
xnew =  4.6667
xnew =  4.4762
xnew =  4.4721
xnew =  4.4721
xnew =  4.4721
```

When MATLAB executes a program M-file for the first time, it will open the appropriate text M-file and compile the function into a low-level representation that will be stored within MATLAB. For those cases where an M-file function references other M-file functions, they will also be compiled and placed in MATLAB's memory.

A program M-file will terminate its execution when either a return statement is encountered or, as is the case in this example, an end-of-file (EOF) is reached.

The MATLAB programs should include a good number of comments telling the reader in plain English what is occurring.

For example, typing note the following command and MATLAB response.

```
>> help Newton
  Calculate square root of a given number by Newton's method
  Read input
```

Observations:

The example shown is simple. However, it contains important programming constructs, likely to be used again and again in larger programs.

The first point to note is the line with % marker. This denotes a comment. A comment highlights some aspect of the program code. The first few comment lines explain what the program is all about. MATLAB displays these lines when the help command is used, as shown above.

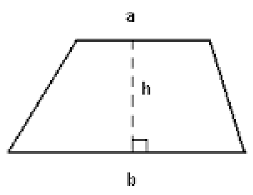
The same program can be used with different inputs if the input data is read from a file or directly from the keyboard as shown.

Two important constructs used in this program are the 'for' statement and the 'if' statement. The first construct allows for iterations or looping, allowing execution of the statements in the loop repeatedly. However, the 'break' statement allows one to jump out of the loop on a specified condition.

Perhaps the most important feature is that of extending the language. At the MATLAB command prompt ">>", we simply typed the name of the user written program as if it was a built-in operation in MATLAB itself. Thus, the program can be written and executed within the MATLAB as the natural extension of the MATLAB programming language.

More Examples of Programming and Software Development in MATLAB

Example 1: Area of a Trapezoid



Contents of traparea.m file:

```
function area = traparea(a,b,h)
%  traparea(a,b,h)   Computes the area of a trapezoid given
%  the dimensions a, b and h, where a and b
%  are the lengths of the parallel sides and %  h is the distance between these sides
%  Compute the area, but suppress printing of the result
area = 0.5*(a+b)*h;
```

MATLAB Session:

```
>> TrapArea=traparea(4,6,5)
TrapArea =
   25
```

Example 2: Cartesian to Polar Coordinates

cartpolar.m contents:

```
function [r,theta]=cartpolar(x,y)
%   cartrtpolar  Convert Cartesian coordinates to polar coordinates
%
%[r,theta] = cart2plr(x,y) computes r and theta with
%
%r = sqrt(x^2 + y^2);
%
%theta = atan2(y,x);
r = sqrt(x^2 + y^2);
theta = atan2(y,x);
```

MATLAB Session:

```
>> [a,b]=cartpolar(3,4)
a =
    5
b =
0.9273
```

Example 3: One of the fun example of writing a program often presented in a language on programming is that of calculating the roots of the quadratic equation given by the formula:

$$x = \frac{-b\pm\sqrt{b^2-4ac}}{2a}$$

Shown below is a simple rendering of this problem in the form of a solution that takes care of various possibilities, including those of degenerate case as well those of real and complex roots of a quadratic equation. Normally programming solutions are typed in text using a Notepad or Wordpad.

Here are the contents of the program saved as "roots.m" in a directory of users choice in the MATLAB path and changing to that directory by using a chdir command in MATLAB.

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
% roots.m -- Coefficients are read in from keyboard, answers printed on screen
%
% Note:
% The algorithm or procedure used here does not take into account
% possible loss of accuracy when two floating point numbers of almost equal size are subtracted.
%
% Print Welcome Message
display('Welcome to the Quadratic Equation Solver');
% Prompt User for Coefficients of Quadratic Equation
display('Please enter coefficients for the equation ax^2 + bx + c');
a=input('Enter coefficient a: ');
b=input('Enter coefficient b: ');
c=input('Enter coefficient c: ');
% Print Quadratic Equation to Screen
% Compute Roots of simplified equations : A equals zero
rootfound=0;
if a==0 & b==0,
display('Cannot solve for roots.  Degenerate case');
rootfound=1;
end;
if a==0 & b~=0 & rootfound==0,
root1=-c/b;
display('Degenerate root');
root1
rootfound=1;
end;
% Compute Roots of Quadratic Equation : a not equal to zero
if rootfound==0
discriminant=b*b-4*a*c;
% Compute discriminant of quadratic equation.
if discriminant>=0, % Case for two real roots
root1=-b/2*a - sqrt(discriminant)/(2*a);
root1
root2=-b/(2*a) + sqrt(discriminant)/(2*a);
root2
else      % Case for complex roots
display('Two complex roots');
display('Root 1 real part');
root1r=-b/(2*a)
display('Root 1 complex part')
root1c=sqrt(-discriminant)/(2*a)
display('Root 2 real part');
root2r=-b/(2*a)
display('Root 2 complex part')
root2c=-sqrt(-discriminant)/(2*a)
end;
end;
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```

The actual interactions in MATLAB are:

```
>> roots
Welcome to the Quadratic Equation Solver
Please enter coefficients for the equation ax^2 + bx + c
Enter coefficient a: 1
Enter coefficient b: 1
Enter coefficient c: -12
root1 =
    -4
root2 =
     3

>> roots
Welcome to the Quadratic Equation Solver
Please enter coefficients for the equation ax^2 + bx + c
Enter coefficient a: 0
Enter coefficient b: 0
Enter coefficient c: -12
Cannot solve for roots.  Degenerate case

>> roots
Welcome to the Quadratic Equation Solver
Please enter coefficients for the equation ax^2 + bx + c
Enter coefficient a: 4
Enter coefficient b: 2
Enter coefficient c: 4
Two complex roots
Root 1 real part
root1r =
   -0.2500
Root 1 complex part
root1c =
    0.9682
Root 2 real part
```

```
root2r =
   -0.2500
Root 2 complex part
root2c =
   -0.9682

>> roots
Welcome to the Quadratic Equation Solver
Please enter coefficients for the equation ax^2 + bx + c
Enter coefficient a: 0
Enter coefficient b: 4
Enter coefficient c: 4
Degenerate root
root1 =
    -1

>> help roots
  roots.m -- Coefficients are read in from keyboard, answers printed on screen
  Note:
The algorithm or procedure used here does not take into account possible loss of accuracy when
two floating point numbers of almost equal size are subtracted.
Print Welcome Message
```

Example 4: A program for solutions of simultaneous in 1 or more variables.

Program Code:

```
%simult.m, Simultaneous Equations Solver
a=input('Please type data in the form of a square matrix of coefficients:\n');
b=input('Please type the corresponding column vector of constants:\n');
d=det(a);
if d~=0,
display('Determinant Not Zero, Solution Exists');
s=a\b;
display('Column vector s is the solution showing values of variables:');
s
else
display('Determinant Zero. Solution Does not exist');
end;
```

MATLAB Session

```
%Equation with two variables of the form ax+by=c
>> simult
Please type data in the form of a square matrix of coefficients:
[2 -2;7 -8]
Please type the corresponding column vector of constants:
[3; -2]
Determinant Not Zero, Solution Exists
```

```
Column vector s is the solution showing values of variables:
s =
   14.0000
   12.5000
```

%Equation with three variables of the form $ax_1+bx_2+cx_3=z$

```
>> simult
Please type data in the form of a square matrix of coefficients:
[1 1 1; 0 2 5; 2 5 -1]
Please type the corresponding column vector of constants:
[6; -4; 27]
Determinant Not Zero, Solution Exists
Column vector s is the solution showing values of variables:
s =
     5
     3
    -2
```

%Equation with five variables of the form $ax_1+bx_2+cx_3+dx_4+ex_5=z$

```
>> simult
Please type data in the form of a square matrix of coefficients:
[0 2 -1 -2 0;-2 2 -3 -4 -8;0 0 1 3 5;1 0 1 2 6;0 2 -1 -1 3]
Please type the corresponding column vector of constants:
[-1; 0; 1; 0; -1]
Determinant Not Zero, Solution Exists
Column vector s is the solution showing values of variables:
s =

    1.0000
    0.5000
   -1.0000
    1.5000
   -0.5000
```

Object-Oriented Programming in MATLAB

The program examples introduced so far have made use of procedural programming. In procedural programming, design of solution focuses on the steps to be executed for achieving a desired result. Typically, one represents data as individual variables or fields of a structure. Operations are implemented as functions that take the variables as arguments. Programs usually call a sequence of functions, each one of which is passed data, and then returns modified data. Each function performs an operation or many operations on the data.

Object-oriented program design involves Identifying the components of the system or application to be used in developing a solution, based on an analysis and identification of patterns that may be used repeatedly.

Each component type is defined as a class with desired attributes called properties and behaviour called methods. A class may also be designed to recognize events representing changes in an object instance such as modification of data, execution of a method, querying or setting a property value, or destruction of an object.

A class describes a set of objects with common characteristics. Objects are specific instances of a defined class. The values contained in an object's properties are what make an object different from other objects of the same class. The functions defined by the class (called methods) are what implement object behaviours that are common to all objects of a class.

Advantages of Object-Oriented Programming

1. ***Code Reuse and Recycling***: Objects created for Object-oriented Programs can easily be reused in other programs.
2. ***Encapsulation:*** Once an Object is created, knowledge of its implementation is not necessary for its use. In older programs, coders needed understand the details of a piece of code before using it (in this or another program). Objects have the ability to hide certain parts of themselves from programmers. This prevents programmers from tampering with values they shouldn't. Additionally, the object controls how one interacts with it, preventing other kinds of errors.
3. ***Design Benefits***: Large programs are very difficult to write. Object-oriented Programs force designers to go through an extensive planning phase, which makes for better designs with less flaws. In addition, once a program reaches a certain size, Object Oriented Programs are actually *easier* to program than non-object-oriented ones.
4. ***Software Maintenance:*** Programs are not disposable. Legacy code must be dealt with on a daily basis, either to be improved upon (for a new version of an exist piece of software) or made to work with newer computers and software. An object-oriented Program is much easier to modify and maintain than a non-object-oriented Program. So, although a lot of work is spent before the program is written, less work is needed to maintain it over time.

Some disadvantage of Object-Oriented Programming

1. ***Size***: Object Oriented programs are much larger than other programs. In the early days of computing, space on hard drives, floppy drives and in memory was at a premium. Today we do not have these restrictions.
2. ***Effort***: Object Oriented programs require a lot of work to create. Specifically, a great deal of planning goes into an object-oriented program well before a single piece of code is ever written. Initially, this early effort was felt by many to be a waste of time. In addition, because the programs were larger (see above) coders spent more time actually writing the program.
3. ***Speed***: Object Oriented programs are slower than other programs, partially because of their size. Other aspects of Object-oriented Programs also demand more system resources, thus slowing the program down.
4. In recent years, however, improvements in computer performance have made restrictions about size and speed inconsequential. The question of human effort still exists, however; many novice programmers do not like Object-oriented Programming because of the great deal of work required to produce minimal results.

When Should One Create Object-Oriented Programs?

You can implement simple programming tasks as simple functions. However, as the magnitude and complexity of your tasks increase, functions become more complex and difficult to manage. As functions become too large, you can break them into smaller functions and pass data from one to function to another. However, as the number of functions becomes large, designing, and managing the data passed to functions becomes difficult and error prone.

Understand a Problem in Terms of Its Objects

Thinking in terms of objects is simpler and more natural for some problems. Think of the nouns in your problem statement as the objects to define and the verbs as the operations to perform. Consider the design of classes to represent money lending institutions (banks, mortgage companies, individual money lenders, and so on). It is difficult to represent the various types of lenders as procedures. However, you can represent each one as an object that performs certain actions and contains certain data. The process of designing the objects involves identifying the characteristics of a lender that are important to your application.

The MATLAB language defined objects are used for programming solutions in MATLAB. Consider the following example of a file called Ex1Class.m save in the current directory:

```
% Ex1Class.m represents a class dealing with numeric data.
classdef Ex1Class
  properties
    Value
  end
  methods
% Class constructor method
    function obj = Ex1Class(val)
      if nargin == 1
        if isnumeric(val)
          obj.Value = val;
        else
          error('Value must be numeric')
        end
      end
    end
% Class roundOff method
    function r = roundOff(obj)
      r = round([obj.Value],2);
    end
    function r = multiplyBy(obj,n)
      r = [obj.Value] * n;
    end
% Class plus method for adding two inout values of Ex1Class instances
```

```
      function r = plus(o1,o2)
         r = [o1.Value] + [o2.Value];
      end
   end
end
```

Use of the defined class in a MATLAB Session:

```
>> a=Ex1Class
a =
  Ex1Class with properties:
    Value: []

>> a(1)=Ex1Class(5.678)
a =
  Ex1Class with properties:
    Value: 5.6780

>> a(2)=Ex1Class(9.123)
a =
  1x2 Ex1Class array with properties:
    Value

>> a(3)=Ex1Class(2.456)
a =
  1x3 Ex1Class array with properties:
    Value

>> sum=plus(a(1),a(3))
sum =
    8.1340
>> prod=multiplyBy(a,5)
prod =
   43.5000

>> roundOff(a)
ans =
    5.6800    9.1200    2.4600

>> a(1)
ans =
  Ex1Class with properties:
    Value: 5.6780
% Note that the roundOff displayed rounded values without affecting the stored value

% The following are some built in functions provided MATLAB for probing a class
```

```
>> help Ex1Class %shows the beginning comments in the class definition
 Ex1Class.m represents a class dealing with numeric data.

>> class Ex1Class % class of object
ans =
char

>> isobject(a) % Is input is MATLAB object?
ans =
   1
>> methods Ex1Class % Provide a list of all methods defined in the class
Methods for class Ex1Class:
Ex1Class    multiplyBy  plus        roundOff

>> properties Ex1Class %Provide the names of class properties
Properties for class Ex1Class:
   Value
```

Notes on some object-oriented programming conventions:

Class names should be nouns in UpperCamelCase, with the first letter of every word capitalised. Use whole words — avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).

Methods should be verbs in lowerCamelCase or a multi-word name that begins with a verb in lowercase; that is, with the first letter lowercase and the first letters of subsequent words in uppercase.

Local variables, instance variables, and class variables are also written in lowerCamelCase. Variable names should not start with underscore (_) or dollar sign ($) characters, even though both are allowed.

Variable names should be short yet meaningful. The choice of a variable name should be mnemonic — that is, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except for temporary "throwaway" variables. Common names for temporary variables are i, j, k, m, and n for integers; c, d, and e for characters.

Constants should be written in uppercase characters separated by underscores. Constant names may also contain digits if appropriate, but not as the first character.

Systematic Approach to Design and Implementation of Classes: An Example

This example discusses how to approach the design and implementation of a class. The objective of this class is to represent a familiar concept (a bank account). However, one can apply the same approach to most class designs.

To design a class that represents a bank account, first determine the elements of data and the operations that form your abstraction of a bank account.

For example, a bank account has:
• An account number
• An account balance
• A status (open, closed, etc.)

Also needed are certain operations on a bank account:
• Create an object for each bank account
• Deposit money
• Withdraw money
• Generate a statement
• Save and load the BankAccount object

If the balance is too low there is an attempt to withdraw money, the bank account broadcasts a notice. When this event occurs, the bank account broadcasts a notice to other entities that are designed to listen for these notices. In this example, a simplified version of an account manager program performs this task.

An account manager program may determine the status of all bank accounts. This program monitors the account balance and assigns one of three values:
• open — Account balance is a positive value
• overdrawn — Account balance is overdrawn, but by $200 or less.
• closed — Account balance is overdrawn by more than $200.

These features define the requirements of the BankAccount and AccountManager classes. Included only is functionality required to meet specific objectives. Support may be provided by special types of accounts as subclass of BankAccount and adding more specific features to the subclasses. AccountManager class may be required to support new account types.

Specifying Class Components
Classes store data in properties, implement operations with methods, and support notifications with events and listeners. Here is how the BankAccount and AccountManager classes define these components.

Class Data

The class defines the properties to store the account number, account balance, and the account status:
• AccountNumber — A property to store the number identifying the specific account. MATLAB assigns a value to this property when one creates an instance of the class. Only BankAccount class methods can set this property. The SetAccess attribute is private.

• AccountBalance — A property to store the current balance of the account. The class operation of depositing and withdrawing money assigns values to this property.

Only BankAccount class methods can set this property. The SetAccess attribute is private.
• AccountStatus — The BankAccount class defines a default value for this property.

The AccountManager class methods change this value whenever the value of the ccountBalance falls below 0. The Access attribute specifies that only the AccountManager and BankAccount classes have access to this property.
• AccountListener — Storage for the InsufficentFunds event listener. Saving a BankAccount object does not save this property because you must recreate the listener when loading the object.

Class Operations
The methods implement the operations defined in the class formulation:
• BankAccount — Accepts an account number and an initial balance to create an object that represents an account.
• deposit — Updates the AccountBalance property when a deposit transaction occurs
• withdraw — Updates the AccountBalance property when a withdrawal transaction occurs
• getStatement — Displays information about the account
• loadobj — Recreates the account manager listener when you load the object from a MAT-file.

Class Events
The account manager program changes the status of bank accounts that have negative balances. To implement this action, the BankAccount class triggers an event when a withdrawal results in a negative balance. Therefore, the triggering of the InsufficientsFunds event occurs from within the withdraw method. To define an event, specify a name within an events block. Trigger the event by a call to the notify handle class method. Because InsufficientsFunds is not a predefined event, one can name it with any string and trigger it with any action.

BankAccount Class Implementation

It is important to ensure that there is only one set of data associated with any object of a BankAccount class. It would not be appropriate to have independent copies of the object that could have, for example, different values for the account balance. Therefore, there is a need to implement the BankAccount class as a handle class. All copies of a given handle object refer to the same data.

BankAccount Class Synopsis

Bank Account Class	**Comments**
classdef BankAccount < handle	Handle class because there should be only one copy of any instance of BankAccount

properties (SetAccess = private)
    AccountNumber
    AccountBalance
end
	AccountStatus propert access by Account Manager class methods. AccountNumber and AccountBalance properties have private set access

properties (Transient)	AccountListener property is transient so the

AccountListener end	listener handle is not saved.
events InsufficientFunds end	Class defines event called InsufficentFunds. withdraw method triggers event when account balance becomes negative.

Methods
function BA = BankAccount(AccountNumber,InitialBalance)
BA.AccountNumber = AccountNumber;
BA.AccountBalance = InitialBalance;
BA.AccountListener = AccountManager.addAccount(BA);
End
 	
function deposit(BA,amt)
BA.AccountBalance = BA.AccountBalance + amt;
if BA.AccountBalance > 0
BA.AccountStatus = 'open';
end
end
	Constructor initializes property values with input arguments.
	
	AccountManager.addAccount is static method of AccountManager class. Creates listener for InsufficientFunds event and stores listener handle in
	AccountListener property.
	
	Deposit adjusts value of AccountBalance property
	
	If AccountStatus is closed and subsequent deposit brings AccountBalance into positive range, then AccountStatus is reset to open.

function withdraw(BA,amt)
if (strcmp(BA.AccountStatus,'closed')&& BA.AccountBalance < 0)
disp(['Account ',num2str(BA.AccountNumber),...
' has been closed.'])
return
end
newbal = BA.AccountBalance - amt;
BA.AccountBalance = newbal;
if newbal < 0
notify(BA,'InsufficientFunds')
end
end
	Updates AccountBalance property. If value of account balance is negative as result of the withdrawal, notify triggers InsufficentFunds event.

function getStatement(BA)	Display selected information about the

`disp('-------------------------')`
`disp(['Account: ',num2str(BA.AccountNumber)])`
`ab = sprintf('%0.2f',BA.AccountBalance);`
`disp(['CurrentBalance: ',ab])`
`disp(['Account Status: ',BA.AccountStatus])`
`disp('-------------------------')`
`end`
`end`
`methods (Static)`
	account.
	
	End of ordinary methods block.

`function obj = loadobj(s)`
`if isstruct(s)`
`accNum = s.AccountNumber;`
`initBal = s.AccountBalance;`
`obj = BankAccount(accNum,initBal);`
`else`
`obj.AccountListener = AccountManager.addAccount(s);`
`end`
`end`
`end`
`end`
	loadobj method:
	• If the load operation fails, create the object from a struct.
	• Recreates the listener using the newly created BankAccount object as the source.
	
	End of static methods block
	
	End of classdef

```
classdef BankAccount < handle
        properties (Access = ?AccountManager)
AccountStatus = 'open';
end
properties (SetAccess = private)
        AccountNumber
        AccountBalance
end
properties (Transient)
        AccountListener
end
events
        InsufficientFunds
end
methods
        function BA = BankAccount(accNum,initBal)
                BA.AccountNumber = accNum;
                BA.AccountBalance = initBal;
                BA.AccountListener = AccountManager.addAccount(BA);
```

```
end
        function deposit(BA,amt)
                BA.AccountBalance = BA.AccountBalance + amt;
                if BA.AccountBalance > 0
                BA.AccountStatus = 'open';
        end
end
        function withdraw(BA,amt)
        if (strcmp(BA.AccountStatus,'closed')&& BA.AccountBalance <= 0)
                disp(['Account ',num2str(BA.AccountNumber),' has been closed.'])
                return
        end
        newbal = BA.AccountBalance - amt;
        BA.AccountBalance = newbal;
        if newbal < 0
                notify(BA,'InsufficientFunds')
        end
end
        function getStatement(BA)
                disp('-------------------------')
                disp(['Account: ',num2str(BA.AccountNumber)])
                ab = sprintf('%0.2f',BA.AccountBalance);
                disp(['CurrentBalance: ',ab])
                disp(['Account Status: ',BA.AccountStatus])
                disp('-------------------------')
        end
end
methods (Static)
        function obj = loadobj(s)
                if isstruct(s)
                        accNum = s.AccountNumber;
                        initBal = s.AccountBalance;
                        obj = BankAccount(accNum,initBal);
                else
                        obj.AccountListener = AccountManager.addAccount(s);
                end
        end
end
end
```
....

BankAccount class requires AccountManager class as follows:

```
classdef AccountManager
   methods (Static)
       function assignStatus(BA)
           if BA.AccountBalance < 0
             if BA.AccountBalance < -200
               BA.AccountStatus = 'closed';
```

```
                else
                   BA.AccountStatus = 'overdrawn';
                 end
             end
         end
         function lh = addAccount(BA)
             lh = addlistener(BA, 'InsufficientFunds', ...
               @(src, ~)AccountManager.assignStatus(src));
         end
     end
end
```

Each class is distinct and each class has its description in a separate file.
Contents of BankAccount class in file BankAccount.m:

```
%BankAccount.m - BankAccount class.  It requires AccountManager class.
classdef BankAccount < handle
  properties (Access = ?AccountManager)
  AccountStatus = 'open';
end
  properties (SetAccess = private)
    AccountNumber
    AccountBalance
  end
  properties (Transient)
    AccountListener
  end
  events
    InsufficientFunds
  end
  methods
    function BA = BankAccount(accNum,initBal)
      BA.AccountNumber = accNum;
      BA.AccountBalance = initBal;
      BA.AccountListener = AccountManager.addAccount(BA);
    end
    function deposit(BA,amt)
      BA.AccountBalance = BA.AccountBalance + amt;
      if BA.AccountBalance > 0
        BA.AccountStatus = 'open';
      end
    end
    function withdraw(BA,amt)
      if (strcmp(BA.AccountStatus,'closed')&& BA.AccountBalance <= 0)
        disp(['Account ',num2str(BA.AccountNumber),' has been closed.'])
        return
      end
      newbal = BA.AccountBalance - amt;
```

```
        BA.AccountBalance = newbal;
        if newbal < 0
          notify(BA,'InsufficientFunds')
        end
      end
      function getStatement(BA)
        disp('-------------------------')
        disp(['Account: ',num2str(BA.AccountNumber)])
        ab = sprintf('%0.2f',BA.AccountBalance);
        disp(['CurrentBalance: ',ab])
        disp(['Account Status: ',BA.AccountStatus])
        disp('-------------------------')
      end
    end
    methods (Static)
      function obj = loadobj(s)
        if isstruct(s)
          accNum = s.AccountNumber;
          initBal = s.AccountBalance;
          obj = BankAccount(accNum,initBal);
        else
        obj.AccountListener = AccountManager.addAccount(s);
        end
      end
    end
  end
```

Contents of BankManager class in BankManager.m file:

```
% AccountManager.m - AccountManager class description
classdef AccountManager
  methods (Static)
    function assignStatus(BA)
      if BA.AccountBalance < 0
        if BA.AccountBalance < -200
          BA.AccountStatus = 'closed';
      else
        BA.AccountStatus = 'overdrawn';
      end
    end
  end
    function lh = addAccount(BA)
      lh = addlistener(BA, 'InsufficientFunds', ...
        @(src, ~)AccountManager.assignStatus(src));
    end
  end
end
```

In MATLAB Session

```
>> BA1=BankAccount(101,500)
BA1 =
  BankAccount with properties:
      AccountNumber: 101
     AccountBalance: 500
    AccountListener: [1x1 event.listener]

>> getStatement(BA1)
-------------------------
Account: 101
CurrentBalance: 500.00
Account Status: open
-------------------------
>> withdraw(BA,100)
Undefined function or variable 'BA'.

>> withdraw(BA1.100)

>> getStatement(BA1)
-------------------------
Account: 101
CurrentBalance: 400.00
Account Status: open
-------------------------
>> withdraw(BA1,500)
>> getStatement(BA1)
-------------------------
Account: 101
CurrentBalance: -100.00
Account Status: overdrawn
-------------------------
Account: 101
CurrentBalance: -200.00
Account Status: overdrawn
-------------------------
>> withdraw(BA1,100)
>> withdraw(BA1,100)
Account 101 has been closed.
```

A Brief on Advanced Software Development in MATLAB

The high-level language in MATLAB® includes features for developing and sharing code, such as error handling, object-oriented programming (OOP), and a unit testing framework. You also can integrate MATLAB applications with those written in other languages.

For example, a MEX-file is a function, created in MATLAB, that calls a C, C++, or Fortran subroutine. To call a MEX-file, use the name of the file, without the file extension. The MEX-file contains only one function or subroutine, and its name is the MEX-file name. The file must be on your MATLAB path.

Binary MEX files are subroutines produced from C/C++ or Fortran source code. They behave just like MATLAB® scripts and built-in functions. While scripts have a platform-independent extension .m, MATLAB identifies MEX files by platform-specific extensions. The following table lists the platform-specific extensions for MEX files.

The following are MEX-File extensions:

Platform	Binary MEX-File Extension
Linux® (64-bit)	mexa64
Apple Mac (64-bit)	mexmaci64
Microsoft® Windows® (32-bit)	mexw32
Windows (64-bit)	mexw64

Binary MEX file on a platform cannot be use if on a different platform. It must be recompiled using the source code on the platform that will use MEX file.

Sources:
Chisnall, David, A Brief History of Programming, Part 1 & 2
http://www.informit.com/articles/article.aspx?p=1077906
http://www.informit.com/articles/article.aspx?p=1080343

Bezroukov, NNikolai, Scripting Languages as a Step in Evolution of Very high Level Languages
http://www.softpanorama.org/People/Scripting_giants/scripting_languages_as_vhll.shtml
Robat, Cornelius, Introduction to Software History
http://www.thocp.net/software/software_reference/introduction_to_software_history.htm
https://en.wikipedia.org/wiki/Computer_programming
Austin, Mark.A., Engineering Programming in MATLAB : A Primer, University of Maryland, 2000
http://www.eng.umd.edu/~austin/wiley.d/book.pdf
http://www.mathworks.com/help/matlab/object-oriented-programming.html
http://www.mathworks.com/help/matlab/software-development.html
https://www.mathworks.com/help/pdf_doc/matlab/matlab_oop.pdf
http://www.mathworks.com/help/matlab/object-oriented-programming.html

http://www.mathworks.com/help/matlab/software-development.html