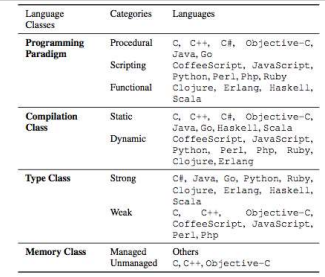
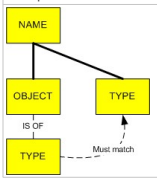
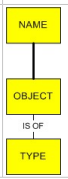
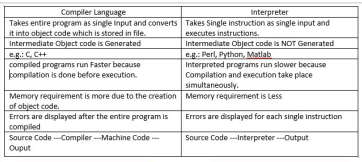
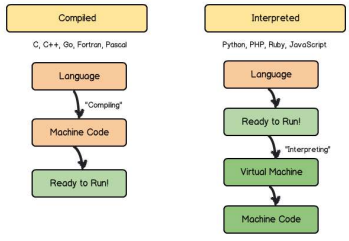
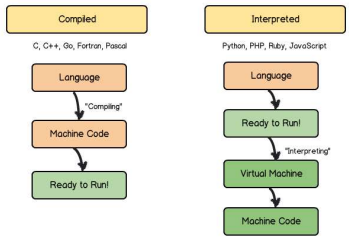
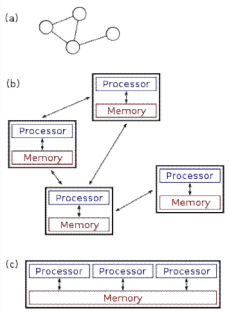
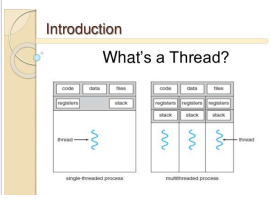
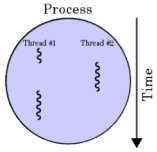
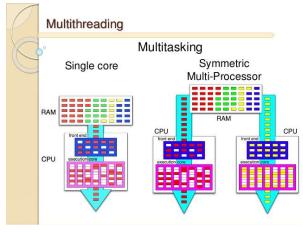
Interview Questions from Papa:

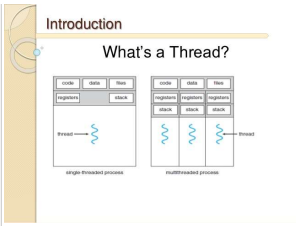
Saturday, March 30, 2019 6:10 PM

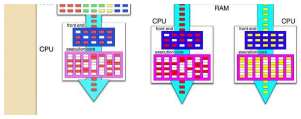
#	Question	Answer:	Extra:

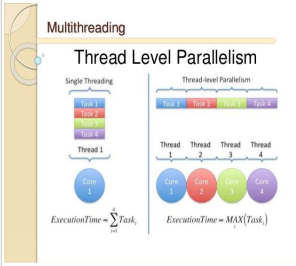
1.	What is the difference between static and dynamic programming languages.	Compiled vs. Interpreted
		"When source code is translated"
		
		Source Code: Original code (usually typed by a human into a computer)
		
		Translation: Converting source code into something a computer can read (i.e. machine code)
		
		Run-Time: Period when program is executing commands (after compilation, if compiled)
		
		Compiled: Code translated before run-time
		
		Interpreted: Code translated on the fly, during execution
		
		Typing
		"When types are checked"
		
		"3" + 5 will raise a type error in strongly typed languages, such as Python and Go, because they don't allow for "type coercion": the ability for a value to change type implicitly in certain contexts (e.g. merging two types using +). Weakly typed languages, such as JavaScript, won't throw a type error (result: '35').
		
		Static: Types checked before run-time
		Dynamic: Types checked on the fly, during execution
		
		The definitions of "Static & Compiled" and "Dynamic & Interpreted" are quite similar...but remember it's "when types are checked" vs. "when source code is translated".
		
		Type-checking has nothing to do with the language being compiled or interpreted! You need to separate these terms conceptually.
		
		Python Example
		Dynamic, Interpreted
		
		def foo(a):
		  if a > 0:
		  print 'Hi'
		  else:
		  print "3" + 5
		foo(2)
		
		Because Python is both interpreted and dynamically typed, it only translates and type-checks code it's executing on. The else block never executes, so "3" + 5 is never even looked at!
		
		What if it was statically typed?
		
		A type error would be thrown before the code is even run. It still performs type-checking before run-time even though it is interpreted.
		
		What if it was compiled?
		
		The else block would be translated/looked at before run-time, but because it's dynamically typed it wouldn't throw an error! Dynamically typed languages don't check types until execution, and that line never executes.
		
		Go Example
		Static, Compiled
		
		package main
		import ("fmt"
		)
		func foo(a int) {
		 if (a > 0) {
		  fmt.Println("Hi")
		 } else {
		  fmt.Println("3" + 5)
		 }
		}
		func main() {
		 foo(2)
		}
		The types are checked before running (static) and the type error is immediately caught! The types would still be checked before run-time if it was interpreted, having the same result. If it was dynamic, it wouldn't throw any errors even though the code would be looked at during compilation.
		
		Performance
		A compiled language will have better performance at run-time if it's statically typed because the knowledge of types allows for machine code optimization.
		
		Statically typed languages have better performance at run-time intrinsically due to not needing to check types dynamically while executing (it checks before running).
		
		Similarly, compiled languages are faster at run time as the code has already been translated instead of needing to "interpret"/translate it on the fly.
		
		Note that both compiled and statically typed languages will have a delay before running for translation and type-checking, respectively.
		
		More Differences
		Static typing catches errors early, instead of finding them during execution (especially useful for long programs). It's more "strict" in that it won't allow for type errors anywhere in your program and often prevents variables from changing types, which further defends against unintended errors.
		
		num = 2
		num = '3' // ERROR
		Dynamic typing is more flexible (which some appreciate) but allows for variables to change types (sometimes creating unexpected errors).
			
			
			| Static Allocation
				Dynamic Allocation

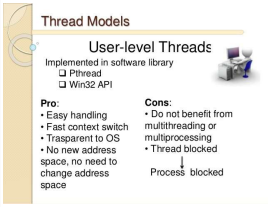
---	---
Performed at static or compile time	Performed at dynamic or run time
Assigned to stack	Assigned to heap
Size must be know at compile time	Size may be unknown at compile time
First in last out	No particular order of assignment
It is best if required size of memory known in advance	It is best if we don't have idea about how much memory require	 In a **statically typed language**, every variable name is bound both • to a type (at compile time, by means of a data declaration) • to an object. The binding to an object is optional — if a name is not bound to an object, the name is said to be *null*. Once a variable name has been bound to a type (that is, declared) it can be bound (via an assignment statement) only to objects of that type; it cannot ever be bound to an object of a different type. An attempt to bind the name to an object of the wrong type will raise a type exception. In a **dynamically typed language**, every variable name is (unless it is null) bound only to an object. Names are bound to objects at execution time by means of assignment statements, and it is possible to bind a name to objects of different types during the execution of the program.  
2.	What is the run-time	Period when program is executing commands (after compilation, if compiled) In computer science, run time, runtime or execution time is the time during which a program is running (executing), in contrast to other program lifecycle phases such as compile time, link time and load time. A run-time error is detected after or during the execution (running state) of a program, whereas a compile-time error is detected by the compiler before the program is ever executed. Type checking, register allocation, code generation, and code optimization are typically done at compile time, but may be done at run time depending on the particular language and compiler. Many other runtime errors exist and are handled differently by different languages, such as Division by zero errors, domain errors, array subscript out of bounds errors, Arithmetic underflow errors, several types of underflow and overflow errors, and many other runtime errors generally considered as software bugs which may or may not be caught and handled by any particular computer language.	When a program is to be executed, a loader first performs the necessary memory setup and links the program with any dynamically linked libraries it needs, and then the execution begins starting from the program's entry point. In some cases, a language or implementation will have these tasks done by the language runtime instead, though this is unusual in mainstream languages on common consumer operating systems. Some program debugging can only be performed (or is more efficient or accurate when performed) at runtime. Logic errors and array bounds checking are examples. For this reason, some programming bugs are not discovered until the program is tested in a production environment with real data, despite sophisticated compile-time checking and pre-release testing. In this case, the end user may encounter a runtime error message.
3.	What is the difference between compiled and interpreted languages	In a compiled implementation, the original program is translated into native machine instructions, which are executed directly by the hardware. In an interpreted implementation, the original program is translated into something else. Another program, called "the interpreter", then examines "something else" and performs whatever actions are called for. Depending on the language and its implementation, there are a variety of forms of "something else". From more popular to less popular, "something else" might be Binary instructions for a virtual machine, often called bytecode, as is done in Lua, Python, Ruby, Smalltalk, and many other systems (the approach was popularized in the 1970s by the UCSD P-system and UCSD Pascal) A tree-like representation of the original program, such as an abstract-syntax tree, as is done for many prototype or educational interpreters One thing that complicates the issue is that it is possible to translate (compile) bytecode into native machine instructions. Thus, a successful interpreted implementation might eventually acquire a compiler. If the compiler runs dynamically, (behind the scenes), it is often called a just-in-time compiler or JIT compiler. JITs have been developed for Java, JavaScript, Lua, and I daresay many other languages. At that point you can have a hybrid implementation in which some code is interpreted and some code is compiled.	 

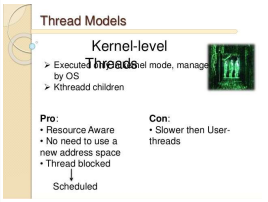
		Interpreters One thing that complicates the issue is that it is possible to translate (compile) bytecode into native machine instructions. Thus, a successful interpreted implementation might eventually acquire a compiler. If the compiler runs dynamically, (behind the scenes), it is often called a just-in-time compiler or JIT compiler. JITs have been developed for Java, JavaScript, Lua, and I daresay many other languages. At that point you can have a hybrid implementation in which some code is interpreted and some code is compiled.	
4.	What is a distributed computing.	Distributed computing is a field of computer science that studies distributed systems. A distributed system is a system whose components are located on different networked computers, which communicate and coordinate their actions by passing messages to one another. The components interact with one another in order to achieve a common goal. Three significant characteristics of distributed systems are: concurrency of components, lack of a global clock, and independent failure of components. Examples of distributed systems vary from SOA-based systems to massively multiplayer online games to peer-to-peer applications. A computer program that runs within a distributed system is called a distributed program (and distributed programming is the process of writing such programs). There are many different types of implementations for the message passing mechanism, including pure HTTP, RPC-like connectors and message queues. Distributed computing also refers to the use of distributed systems to solve computational problems. In distributed computing, a problem is divided into many tasks, each of which is solved by one or more computers, which communicate with each other via message passing. Distributed systems are groups of networked computers, which have the same goal for their work. The terms "concurrent computing", "parallel computing", and "distributed computing" have a lot of overlap, and no clear distinction exists between them. The same system may be characterized both as "parallel" and "distributed"; the processors in a typical distributed system run concurrently in parallel. Parallel computing may be seen as a particular tightly coupled form of distributed computing, and distributed computing may be seen as a loosely coupled form of parallel computing. Nevertheless, it is possible to roughly classify concurrent systems as "parallel" or "distributed" using the following criteria: In parallel computing, all processors may have access to a shared memory to exchange information between processors. In distributed computing, each processor has its own private memory (distributed memory). Information is exchanged by passing messages between the processors. As a rule of thumb, high-performance parallel computation in a shared-memory multiprocessor uses parallel algorithms while the coordination of a large-scale distributed system uses distributed algorithms.	The figure below illustrates the difference between distributed and parallel systems. Figure (a) is a schematic view of a typical distributed system; the system is represented as a network topology in which each node is a computer and each line connecting the nodes is a communication link.  Figure (b) shows the same distributed system in more detail: each computer has its own local memory, and information can be exchanged only by passing messages from one node to another by using the available communication links. Figure (c) shows a parallel system in which each processor has a direct access to a shared memory. Various hardware and software architectures are used for distributed computing. At a lower level, it is necessary to interconnect multiple CPUs with some sort of network, regardless of whether that network is printed onto a circuit board or made up of loosely coupled devices and cables. At a higher level, it is necessary to interconnect processes running on those CPUs with some sort of communication system.[26] Distributed programming typically falls into one of several basic architectures: client–server, three-tier, n-tier, or peer-to-peer; or categories: loose coupling, or tight coupling.[27] Client–server: architectures where smart clients contact the server for data then format and display it to the users. Input at the client is committed back to the server when it represents a permanent change. Three-tier: architectures that move the client intelligence to a middle tier so that stateless clients can be used. This simplifies application deployment. Most web applications are three-tier. n-tier: architectures that refer typically to web applications which further forward their requests to other enterprise services. This type of application is the one most responsible for the success of application servers. Peer-to-peer: architectures where there are no special machines that provide a service or manage the network resources.[28]:227 Instead all responsibilities are uniformly divided among all machines, known as peers. Peers can serve both as clients and as servers.[29] Another basic aspect of distributed computing architecture is the method of communicating and coordinating work among concurrent processes. Through various message passing protocols, processes may communicate directly with one another, typically in a master/slave relationship. Alternatively, a "database-centric" architecture can enable distributed computing to be done without any form of direct inter-process communication, by utilizing a shared database.[30] Applications Reasons for using distributed systems and distributed computing may include: The very nature of an application may require the use of a communication network that connects several computers: for example, data produced in one physical location and required in another location. There are many cases in which the use of a single computer would be possible in principle, but the use of a distributed system is beneficial for practical reasons. For example, it may be more cost-efficient to obtain the desired level of performance by using a cluster of several low-end computers, in comparison with a single high-end computer. A distributed system can provide more reliability than a non-distributed system, as there is no single point of failure. Moreover, a distributed system may be easier to expand and manage than a monolithic uniprocessor system.[31]
5.	What is multi-threading? The multithreading paradigm has become more popular as efforts to further exploit instruction-level parallelism have stalled since the late 1990s. This allowed the concept of throughput computing to re-emerge from the more specialized field of transaction processing. Even though it is very difficult to further speed up a single thread or single program, most computer systems are actually multitasking among multiple threads or programs. Thus, techniques that improve the throughput of all tasks result in overall performance gains. From the software standpoint, hardware support for multithreading is more visible to software, requiring more changes to both application programs and operating systems than multiprocessing. Hardware techniques used to support multithreading often parallel the software techniques used for computer multitasking. Thread scheduling is also a major problem in multithreading.	In computer science, a thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system. The implementation of threads and processes differs between operating systems, but in most cases a thread is a component of a process. Multiple threads can exist within one process, executing concurrently and sharing resources such as memory, while different processes do not share these resources. In particular, the threads of a process share its executable code and the values of its dynamically allocated variables and non-thread-local global variables at any given time. Systems with a single processor generally implement multithreading by **time slicing:** the central processing unit (CPU) switches between different software threads. This context switching generally happens very often and rapidly enough that users perceive the threads or tasks as running in parallel. On a multiprocessor or multi-core system, multiple threads can execute in parallel, with every processor or core executing a separate thread simultaneously; on a processor or core with hardware threads, separate software threads can also be executed concurrently by separate hardware threads. Threads differ from traditional multitasking operating system processes in that: **processes are typically independent**, while **threads exist as subsets of a process** processes carry considerably more state information than threads, whereas multiple threads within a process share process state as well as memory and other resources processes have separate address spaces, whereas threads share their address space processes interact only through system-provided inter-process communication mechanisms context switching between threads in the same process is typically faster than context switching between processes. In computer architecture, multithreading is the ability of a central processing unit (CPU) (or a single core in a multi-core processor) to execute multiple processes or threads concurrently, supported by the operating system. This approach differs from multiprocessing. *In a multithreaded application, the processes and threads share the resources of a single or multiple cores, which include the computing units, the CPU caches, and the translation lookaside buffer (TLB).* *Where multiprocessing systems include multiple complete processing units in one or more cores, multithreading aims to increase utilization of a single core by using thread-level parallelism, as well as instruction-level parallelism.* As the two techniques are complementary, they are sometimes combined in systems with multiple multithreading CPUs and with CPUs with multiple multithreading cores. 	 Two major techniques for throughput computing are multithreading and multiprocessing. Advantages If a thread gets a lot of cache misses, the other threads can continue taking advantage of the unused computing resources, which may lead to faster overall execution, as these resources would have been idle if only a single thread were executed. Also, if a thread cannot use all the computing resources of the CPU (because instructions depend on each other's result), running another thread may prevent those resources from becoming idle. Disadvantages Multiple threads can interfere with each other when sharing hardware resources such as caches or translation lookaside buffers (TLBs). As a result, execution times of a single thread are not improved and can be degraded, even when only one thread is executing, due to lower frequencies or additional pipeline stages that are necessary to accommodate thread-switching hardware. Overall efficiency varies; Intel claims up to 30% improvement with its Hyper-Threading Technology,[1] while a synthetic program just performing a loop of non-optimized dependent floating-point operations actually gains a 100% speed improvement when run in parallel. On the other hand, hand-tuned assembly language programs using MMX or AltiVec extensions and performing data prefetches (as a good video encoder might) do not suffer from cache misses or idle computing resources. Such programs therefore do not benefit from hardware multithreading and can indeed see degraded performance due to contention for shared resources. 











6. What is concurrency; How is concurrency different from parallelism?

Concurrent and parallel are effectively the same principle as you correctly surmise, both are related to tasks being executed simultaneously although I would say that parallel tasks should be truly multitasking, executed "at the same time" whereas concurrent could mean that the tasks are sharing the execution thread while still appearing to be executing in parallel.

Asynchronous methods aren't directly related to the previous two concepts, asynchrony is used to present the impression of concurrent or parallel tasking but effectively an asynchronous method call is normally used for a process that needs to do work away from the current application and we don't want to wait and block our application awaiting the response.

For example, getting data from a database could take time but we don't want to block our UI waiting for the data. The async call takes a call-back reference and returns execution back to your code as soon as the request has been placed with the remote system. Your UI can continue to respond to the user while the remote system does whatever processing is required, once it returns the data to your call-back method then that method can update the UI (or handoff that update) as appropriate.

From the User perspective, it appears like multitasking but it may not be.

Analogy:

Problem: You have to build two parallel brick walls. You have to carry the pile of bricks with you as you move along building the two walls.

Naive Solution: Build one wall. Go back to the start point. Build the other wall. It is simple to see how this isn't efficient. You'll be carrying the bricks along with you till the end, and then move them back to the start point.

Concurrency: Lay down one column of bricks of one wall, and then move to the other wall. When two corresponding columns of the two walls are built, move to the next column. This is a lot more efficient, since by the time you reach the end, you will be done, and the effort of carrying the pile of bricks from one end back to the start, will not be required.

Parallelism: Hire another brick layer who works alongside you on the second wall, while you are working on the first wall. This is obviously the best model, since it essentially reduces time to lay the bricks by half. However, it does require an extra brick layer.

Parallelism vs. Concurrency: Both are models of multi-programming.

Concurrency is essentially when two tasks are being performed at the same time. This might mean that one is 'paused' for a short duration, while the other is being worked on. Importantly, a different task is begun before an ongoing task is completed. This makes it a multi-programming model.

Parallelism requires that at least two processes/tasks are actively being performed at a particular moment in time. As illustrated by the metaphor above, this means that you require at least two 'processors' or 'workers'.

The number of tasks actively being completed/performed at any instance of time, is what differentiates the two models.

End of Analogy

Concurrency means multiple tasks which start, run, and complete in overlapping time periods, in no specific order. Parallelism is when multiple tasks OR several part of a unique task literally run at the same time, e.g. on a multi-core processor.

Differences between concurrency vs. parallelism

Now let's list down remarkable differences between concurrency and parallelism.

Concurrency is when two tasks can start, run, and complete in overlapping time periods. Parallelism is when tasks literally run at the same time, eg. on a multi-core processor.

Concurrency is the composition of independently executing processes, while parallelism is the simultaneous execution of (possibly related) computations.

Concurrency is about dealing with lots of things at once. Parallelism is about doing lots of things at once.

An application can be concurrent – but not parallel, which means that it processes more than one task at the same time, but no two tasks are executing at same time instant.

An application can be parallel – but not concurrent, which means that it processes multiple sub-tasks of a task in multi-core CPU at same time.

An application can be neither parallel – nor concurrent, which means that it processes all tasks one at a time, sequentially.

An application can be both parallel – and concurrent, which means that it processes multiple tasks concurrently in multi-core CPU at same time.

Concurrency

Concurrency is essentially applicable when we talk about minimum two tasks or more. When an application is capable of executing two tasks virtually at same time, we call it concurrent application. Though here tasks run looks like simultaneously, but essentially they MAY not. They take advantage of CPU time-slicing feature of operating system where each task run part of its task and then go to waiting state. When first task is in waiting state, CPU is assigned to second task to complete it's part of task.

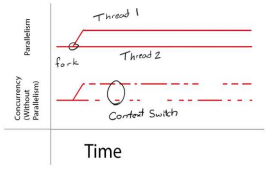
Operating system based on priority of tasks, thus, assigns CPU and other computing resources e.g. memory; turn by turn to all tasks and give them chance to complete. To end user, it seems that all tasks are running in parallel. This is called concurrency.

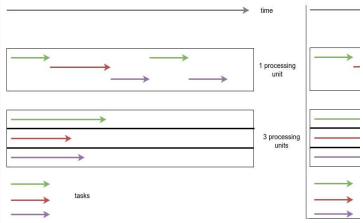
Parallelism

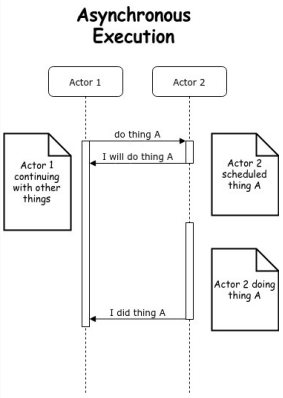
Parallelism does not require two tasks to exist. It literally physically run parts of tasks OR multiple tasks, at the same time using multi-core infrastructure of CPU, by assigning one core to each task or sub-task.

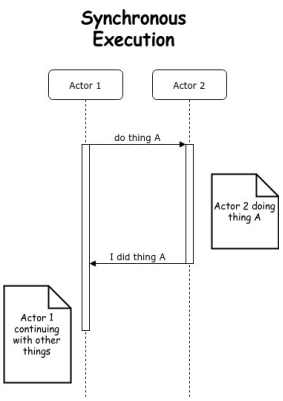
Parallelism requires hardware with multiple processing units, essentially. In single core CPU, you may get concurrency but NOT parallelism.

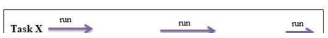
Asynchronous methods

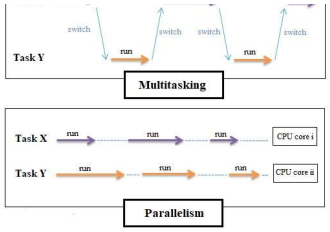


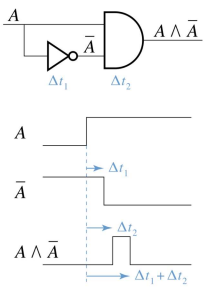


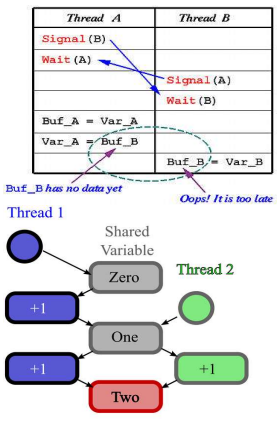






		This is not related to Concurrency and parallelism, asynchrony is used to present the impression of concurrent or parallel tasking but effectively an asynchronous method call is normally used for a process that needs to do work away from the current application and we don't want to wait and block our application awaiting the response. Asynchrony is a separate concept (even though related in some contexts). It refers to the fact that one event might be happening at a different time (not in synchrony) to another event. The below diagrams illustrate what's the difference between a synchronous and an asynchronous execution, where the actors can correspond to different threads, processes or even servers.	

7.	What is a race condition?
	
	Race conditions arise in software when an application depends on the sequence or timing of processes or threads for it to operate properly. As with electronics, there are critical race conditions that result in invalid execution and bugs. Critical race conditions often happen when the processes or threads depend on some shared state. Operations upon shared states are critical sections that must be mutually exclusive. Failure to obey this rule opens up the possibility of corrupting the shared state.
	
	The memory model defined in the C11 and C++11 standards uses the term "data race" for a race condition caused by potentially concurrent operations on a shared memory location, of which at least one is a write. A C or C++ program containing a data race has undefined behavior.[3][4]
	
	Race conditions have a reputation of being difficult to reproduce and debug, since the end result is nondeterministic and depends on the relative timing between interfering threads. Problems occurring in production systems can therefore disappear when running in debug mode, when additional logging is added, or when attaching a debugger, often referred to as a "Heisenbug". It is therefore better to avoid race conditions by careful software design rather than attempting to fix them afterwards.
		Hardware:
		
		A race condition is a behavior which occurs in software applications or electronic systems, such as logic systems, where the output is dependent on the timing or sequence of other uncontrollable events. Race conditions also occur in software which supports multithreading, use a distributed environment or are interdependent on shared resources. Race conditions often lead to bugs, as these events happen in a manner that the system or programmer never intended for. It can often result in a device crash, error notification or shutdown of the application.
		
		A race condition is also known as a race hazard.
		A race condition is often classified as either a critical race condition or non-critical race condition. A critical race condition occurs when the sequence in which internal variables change determines the final state of the machine. A non-critical race condition occurs when the sequence in which internal variables' changes do not have any impact on the final state of the machine. Race conditions are notorious for being difficult to troubleshoot, as reproduction depends on the relative timing between the different elements. Sometimes, especially with software applications, the problem disappears while running in debug mode thanks to an additional logger or debugger.
		
		One of the best ways to avoid a race condition in software and hardware applications is the use of mutual exclusion, which assures that only one process can handle the shared resource at a time, while other processes need to wait. In many cases, Race conditions can be avoided in computing environments with help of serialization of memory or storage access. Another technique that is recommended, especially in software applications, is to analyze and avoid the race condition in the software design itself. There are certain software tools available which help in the detection of race conditions for software.
		
		A race condition occurs when two or more threads can access shared data and they try to change it at the same time. Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are "racing" to access/change the data.
		
		Problems often occur when one thread does a "check-then-act" (e.g. "check" if the value is X, then "act" to do something that depends on the value being X) and another thread does something to the value in between the "check" and the "act". E.g:
		
		`if (x == 5) // The "Check"`
		`{`
		`  y = x * 2; // The "Act"`
		
		`  // If another thread changed x in between "if (x == 5)" and "y = x * 2" above,`
		`  // y will not be equal to 10.`
		`}`
		The point being, y could be 10, or it could be anything, depending on whether another thread changed x in between the check and act. You have no real way of knowing.
		
		In order to prevent race conditions from occurring, you would typically put a lock around the shared data to ensure only one thread can access the data at a time. This would mean something like this:
		
		`// Obtain lock for x`
		`if (x == 5)`
		`{`
		`  y = x * 2; // Now, nothing can change x until the lock is released.`
		`  // Therefore y = 10`
		`}`
		`// release lock for x`
			A race condition or race hazard is the behavior of an electronics, software, or other system where the system's substantive behavior is dependent on the sequence or timing of other uncontrollable events. It becomes a bug when one or more of the possible behaviors is undesirable.
			
			The term race condition was already in use by 1954, for example in David A. Huffman's doctoral thesis "The synthesis of sequential switching circuits". [1]
			
			Race conditions can occur especially in logic circuits, multithreaded or distributed software programs.
			
			 
			Race condition in a logic circuit. Here, Δt_1 and Δt_2 represent the propagation delays of the logic elements. When the input value A changes from low to high, the circuit outputs a short spike of duration ($\Delta t_1 + \Delta t_2$) − Δt_2 = Δt_1.
			
			Example [edit]
			As a simple example, let us assume that two threads want to increment the value of a global integer variable by one. Ideally, the following sequence of operations would take place:
			
			| Thread 1
				Thread 2		Integer value

			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2 | In the case shown above, the final value is 2, as expected. However, if the two threads run simultaneously without locking or synchronization, the outcome of the operation could be wrong. The alternative sequence of operations below demonstrates this scenario: | Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1 | In this case, the final value is 1 instead of the expected result of 2. This occurs because here the increment operations are not mutually exclusive. Mutually exclusive operations are those that cannot be interrupted while accessing some resource such as a memory location. Consider the following execution sequence. A thread in group A executes Signal(B), indicating the intention of a message exchange, followed by Wait(A) for a response from the other party. Because this thread blocks, the CPU is given to another thread. Suppose this is a thread in B. It executes a Signal(A) to show its intention followed by a Wait(B). The Signal() executed by B releases thread A and causes it to move its message from Var_A to the global variable Buf_A followed by retrieving B's message from global variable Buf_B. However, the value in Buf_B is not yet set by thread B and can be a garbage value or the value set by the previous message exchange. Therefore, thread A may show its good intention; however, it does not get the right message. Why is this a race condition? If everything runs fine, both A and B will get correct messages. Now, because two different orders of execution yield two different results, we have a race condition. 
8.	What is a deadlock (Dining philosophers problem)? Five silent philosophers sit at a round table with bowls of	The dining philosophers are often used to illustrate various problems that can occur when many synchronized threads are competing for limited resources. The story goes like this: Five philosophers are sitting at a round table. In front of each philosopher is a bowl of rice. Between each pair of philosophers is one chopstick. Before an individual philosopher can take a bite of rice he must have two chopsticks--one	

spaghetti. Forks are placed between each pair of adjacent philosophers.

Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when they have both left and right forks. Each fork can be held by only one philosopher and so a philosopher can use the fork only if it is not being used by another philosopher. After an individual philosopher finishes eating, they need to put down both forks so that the forks become available to others. A philosopher can take the fork on their right or the one on their left as they become available, but cannot start eating before getting both forks.

Eating is not limited by the remaining amounts of spaghetti or stomach space; an infinite supply and an infinite demand are assumed.

The problem is how to design a discipline of behavior (a concurrent algorithm) such that no philosopher will starve; i.e., each can forever continue to alternate between eating and thinking, assuming that no philosopher can know when others may want to eat or think.

taken from the left, and one taken from the right. The philosophers must find some way to share chopsticks such that they all get to eat.

The following applet does a rough animation using an image of Duke for the dining philosophers. This particular algorithm works as follows: Duke always reaches for the chopstick on his right first. If the chopstick is there, Duke takes it and raises his right hand. Next, Duke tries for the left chopstick. If the chopstick is available, Duke picks it up and raises his other hand. Now that Duke has both chopsticks, he takes a bite of rice and says "Mmm!" He then puts both chopsticks down, allowing either of his two neighbors to get the chopsticks. Duke then starts all over again by trying for the right chopstick. Between each attempt to grab a chopstick, each Duke pauses for a random period of time.

Solutions:

Dining philosoph...

Other authors, including Dijkstra, have posed simpler solutions to the dining philosopher problem than that proposed by Tannenbaum (depending on one's notion of "simplicity," of course). One such solution is to restrict the number of philosophers allowed access to the table. If there are N chopsticks but only N-1 philosophers allowed to compete for them, at least one will succeed, even if they follow a rigid sequential protocol to acquire their chopsticks.

This solution is implemented with an integer semaphore, initialized to N-1. Both this and Tannenbaum's solutions avoid deadlock a situation in which all of the philosophers have grabbed one chopstick and are deterministically waiting for the other, so that there is no hope of recovery. However, they may still permit starvation, a scenario in which at least one hungry philosopher never gets to eat.

Starvation occurs when the asynchronous semantics may allow an individual to eat repeatedly, thus keeping another from getting a chopstick. The starving philosopher runs, perhaps, but doesn't make progress. The observation of this fact leads to some further refinement of what fairness means. Under some notions of fairness the solutions given above can be said to be correct.

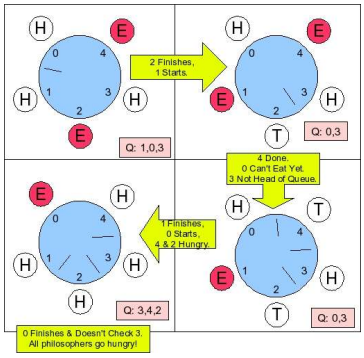
The slider controls the amount of time that each philosopher will wait before attempting to pick up a chopstick. When the slider is set to 0, the philosophers don't wait--they just grab--and the applet ends up in deadlock: all the philosophers are frozen with their right hand in the air. Why? Because each philosopher immediately has one chopstick and is waiting on a condition that cannot be satisfied--they are all waiting for the left chopstick, which is held by the philosopher to their left.

When you move the slider so that the waiting period is longer, the applet may proceed for a while without deadlocking. However, deadlock is always possible with this particular implementation of the dining philosophers problem because it is possible for all five philosophers to be holding their right chopsticks. Rather than rely on luck to prevent deadlock, you must either prevent it or detect it.

For most Java programmers, the best choice is to prevent deadlock rather than to try and detect it. Deadlock detection is complicated and beyond the scope of this tutorial. The simplest approach to to preventing deadlock is to impose ordering on the condition variables. In the dining philosopher applet, there is no ordering imposed on the condition variables because the philosophers and the chopsticks are arranged in a circle. All chopsticks are equal.

However, we can change the rules in the applet by numbering the chopsticks 1 through 5 and insisting that the philosophers pick up the chopstick with the lower number first. The philosopher who is sitting between chopsticks 1 and 2 and the philosopher who is sitting between chopsticks 1 and 5 must now reach for the same chopstick first (chopstick 1) rather than picking up the one on the right. Whoever gets chopstick 1 first is now free to take another one. Whoever doesn't get chopstick 1 must now wait for the first philosopher to release it. Deadlock is not possible.

Book Solution:
The algorithm is simple: only pick up the chopsticks when both are available. To implement this (dphil_5.c), we add an extra array to our system -- one that says whether the chopsticks are free. We have the same semaphores, but now we assign them to philosophers rather than to chopsticks. When a philosopher is hungry, he/she checks this array, and if both chopsticks are free, the philosopher sets them both as used, and then goes through the activity of picking them up. If the chopsticks aren't free, the philosopher sets his/her state to blocked and waits on his/her semaphore.



9. What is the difference between pass-by-value and pass-by-reference.

When a parameter is passed by reference, the caller and the callee use the same variable for the parameter. If the callee modifies the parameter variable, the effect is visible to the caller's variable.

When a parameter is passed by value, the caller and callee have two independent variables with the same value. If the callee modifies the parameter variable, the effect is not visible to the caller.

"Variable" here means the caller's (local or global) variable itself -- i.e. if I pass a local variable by reference and assign to it, I'll change the caller's variable itself, not e.g. whatever it is pointing to if it's a pointer.

This is now considered bad practice (as an implicit dependency). As such, virtually all newer languages are exclusively, or almost exclusively pass-by-value. Pass-by-reference is now chiefly used in the form of "output/inout arguments" in languages where a function cannot return more than one value.
The meaning of "reference" in "pass by reference". The difference with the general "reference" term is is that this "reference" is temporary and implicit. What the callee basically gets is a "variable" that is somehow "the same" as the original one. How specifically this effect is achieved is irrelevant (e.g. the language may also expose some implementation details -- addresses, pointers, dereferencing -- this is all irrelevant; if the net effect is this, it's pass-by-reference).

https://www.mathwarehouse.com/programming/passing-by-value-vs-by-reference-visual-explanation.php

What does it mean to "pass a variable"?
The term "passing a variable" is used when a function is called with a variable you defined previously. Let's look at the following example:

```
int myAge = 14;
calculateBirthYear(myAge);
```

The variable myAge is "passed" to the function calculateBirthYear. The function can then use that variable, for example:

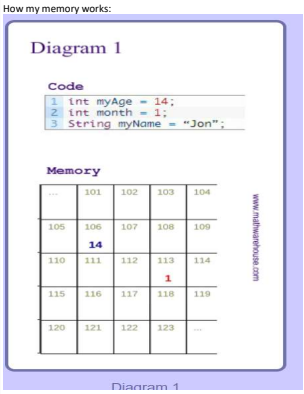
```
function calculateBirthYear(int yourAge) {
return CURRENT_YEAR - yourAge;
}
```

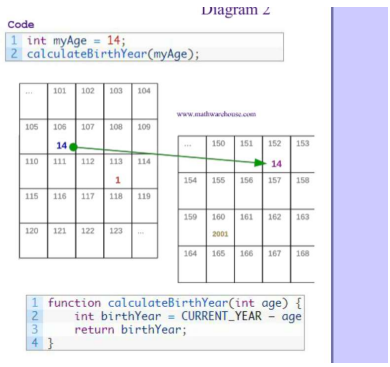
There are two ways that variable myAge can be passed to the function. The terms "pass by value" and "pass by reference" are used to describe the two ways that variables can be passed on.

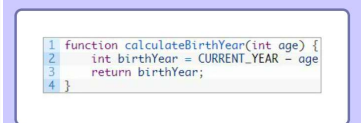
Cliff notes version: : pass by value means the actual value is passed on. Pass by reference means that a number (called a memory address) is passed on, this address defines where the value is stored. Read below for diagrams and examples illustrating this.

How memory works:

A basic knowledge of how memory works and how your variables are stored in memory will help to better understand this topic. Since an actual physical memory is hard to draw and different memory types look different (a hard disc vs. RAM for example), it is useful to have an abstract and simple way to imagine how memory looks like.
For most uses (especially for programming beginners), it is enough to understand the concepts. However, once you get experienced in programming, it is very useful to know more about different memory types and memory regions (harddrives, RAM, heap, stack, etc.).
Diagram 1
Memory can be thought of as blocks, blocks which are next to each other. Each block has a number (the memory address). If you define a variable in your code, the value of the variable will be stored in one of the blocks of memory. Your operating system will automatically decide where the best storage place is.
The gray numbers like 101 and 102 or 106 that are on top of each block show the address of the block in memory, the colored numbers at the bottom show values stored in those memory blocks.
Taking the previous examples again, the variables myAge and month are defined in your code, and they will be stored in memory as shown in diagram 1. The value of myAge is stored at the address 106 and the value of month is stored at the address 113.
There are two main ways that variables can be sent or passed to a function.
Pass by value
Passing by value means that the value of the function parameter is copied (green arrow in Diagram 2) into another location of your memory, and when accessing or modifying the variable within your function, only the copy is accessed/modified and the original value is left untouched. Passing by value is how values are frequently passed in many programming languages like Java.
The value in the memory is copied to another location to be used within the function
Diagram 2
As soon as your software starts processing the calculateBirthYear function, the value myAge is copied to somewhere else in your computers memory. To make this more clear, the variable within the function is named age in this example.
Diagram 3
Anything that you might do to age will not affect the value of myAge.
So, for instance if you changed that function to be birthday(int age) as show below in Diagram 4 and in that function you increase the parameter age, you are in no way changing the original value myAge. As you can see from the picture showing blocks of memory, myAge is stored in memory block 106, but the parameter age is stored in location 152. So when you run line 9 (age = age +1), you are not having any affect on myAge.







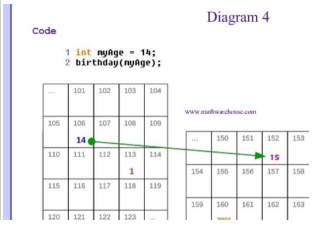


Diagram 4

Code

```
int myAge = 14;
birthday(myAge);
```

```
function birthday(int age){
    age = age + 1
    }
```

Pass by Reference

Passing by reference means that the memory address of the variable (a pointer to the memory location) is passed to the function. This is unlike passing by value, where the value of a variable is passed on. In the examples, the memory address of myAge is 106. When passing myAge to the function increaseAgeByRef, the variable used within the function (age in this example) still points to the same memory address as the original variable myAge (Hint: the & symbol in front of the function parameter is used in many programming languages to get the reference/pointer of a variable).

```
//pass by reference example
function birthday(int & age){
    age = age + 1 ;
    }

int myAge = 14;
birthday(myAge);
```

Diagram 7

When calling the function, the value of myAge is changed directly through its reference.

```
function calculateBirthYear(int age) {
    int birthYear = CURRENT_YEAR - age
    return birthYear;
}
```

```
function calculateBirthYear(int age) {
    int birthYear = CURRENT_YEAR - age
    return birthYear;
}
```

Diagram 3

When calling the function, the value of myAge is changed directly through its reference.

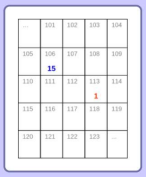


Diagram 8

Memory after altering the variable through its reference.

The value of myAge is now 15.

10. What is generics?

Generic programming is a style of computer programming in which algorithms are written in terms of types to-be-specified-later that are then instantiated when needed for specific types provided as parameters. This approach, pioneered by ML in 1973,[1][2] permits writing common functions or types that differ only in the set of types on which they operate when used, thus reducing duplication. Such software entities are known as generics in Python, Ada, C#, Delphi, Eiffel, F#, Java, Rust, Swift, TypeScript and Visual Basic .NET.

In object-oriented languages

When creating container classes in statically typed languages, it is inconvenient to write specific implementations for each datatype contained, especially if the code for each datatype is virtually identical. For example, in C++, this duplication of code can be circumvented by defining a class template:

```
template<typename T>
class List
{
  /* class contents */
};

List<Animal> list_of_animals;
List<Car> list_of_cars;
```

Above, T is a placeholder for whatever type is specified when the list is created. These "containers-of-type-T", commonly called templates, allow a class to be reused with different datatypes as long as certain contracts such as subtypes and signature are kept. This genericity mechanism should not be confused with inclusion polymorphism, which is the algorithmic usage of exchangeable sub-classes: for instance, a list of objects of type Moving_Object containing objects of type Animal and Car. Templates can also be used for type-independent functions as in the Swap example below:

```
template<typename T>
void Swap(T & a, T & b) //"&" passes parameters by reference
{
  T temp = b;
  b = a;
  a = temp;
}

string hello = "World!", world = "Hello, ";
Swap( world, hello );
cout << hello << world << endl; //Output is "Hello, World!"
```

The C++ template construct used above is widely cited[citation needed] as the genericity construct that popularized the notion among programmers and language designers and supports many generic programming idioms. The D programming language also offers fully generic-capable templates based on the C++ precedent but with a simplified syntax. The Java programming language has provided genericity facilities syntactically based on C++'s since the introduction of J2SE 5.0.

C# 2.0, Oxygene 1.5 (also known as Chrome) and Visual Basic .NET 2005 have constructs that take advantage of the support for generics present in the Microsoft .NET Framework since version 2.0.

Generics in Ada

Ada has had generics since it was first designed in 1977–1980. The standard library uses generics to provide many services. Ada 2005 adds a comprehensive generic container library to the standard library, which was inspired by C++'s standard template library.

A generic unit is a package or a subprogram that takes one or more generic formal parameters.

A generic formal parameter is a value, a variable, a constant, a type, a subprogram, or even an instance of another, designated, generic unit. For generic formal types, the syntax distinguishes between discrete, floating-point, fixed-point, access (pointer) types, etc. Some formal parameters can have default values.

To instantiate a generic unit, the programmer passes actual parameters for each formal. The generic instance then behaves just like any other unit. It is possible to instantiate generic units at run-time, for example inside a loop.

To truly understand generics, first we need to understand three main concepts:

class vs type
subclass vs subtype
variance: covariance, contravariance and invariance

Class vs type

You might not have thought about classes and types as distinct concepts.

In Java and Kotlin all classes have at least one type that is the same as the class, for example: an Integer is a class and at the same time a type.

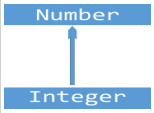
On the other hand, in Kotlin we also have nullable types, consider String?` we can't really say that String? is a class, because notionally, it is still a String.

Another example in Java and Kotlin is List. List is a class, but List<String> is not a class, it's a type.

	Class	Type
String	Yes	Yes
String?	No	Yes
List	Yes	Yes
List<String>	No	Yes

Subclass vs subtype

For a class to be a subclass of another class, it needs to inherit from it. For example, Integer inherits from Number, so Integer is a subclass of Number.



Variance

First let's define a few Kotlin classes:

Advantages and limitations[edit]

The language syntax allows precise specification of constraints on generic formal parameters. For example, it is possible to specify that a generic formal type will only accept a modular type as the actual. It is also possible to express constraints *between* generic formal parameters; for example:

```
generic
   type Index_Type is (<>); -- must be a discrete type
   type Element_Type is private; -- can be any nonlimited type
   type Array_Type is array (Index_Type range <>) of Element_Type;
```

In this example, Array_Type is constrained by both Index_Type and Element_Type. When instantiating the unit, the programmer must pass an actual array type that satisfies these constraints.

The disadvantage of this fine-grained control is a complicated syntax, but, because all generic formal parameters are completely defined in the specification, the compiler can instantiate generics without looking at the body of the generic.

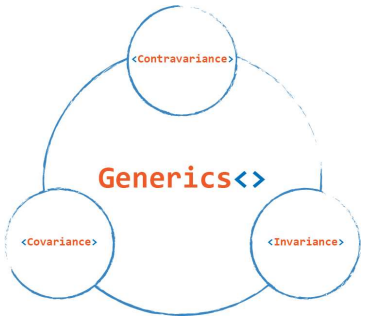
Unlike C++, Ada does not allow specialised generic instances, and requires that all generics be instantiated explicitly. These rules have several consequences:

- the compiler can implement *shared generics*: the object code for a generic unit can be shared between all instances (unless the programmer requests inlining of subprograms, of course). As further consequences:
- there is no possibility of code bloat (code bloat is common in C++ and requires special care, as explained below).
- it is possible to instantiate generics at run-time, as well as at compile time, since no new object code is required for a new instance.
- actual objects corresponding to a generic formal object are always considered to be non-static inside the generic; see Generic formal objects in the Wikibook for details and consequences.
- all instances of a generic being exactly the same, it is easier to review and understand programs written by others; there are no "special cases" to take into account.
- all instantiations being explicit, there are no hidden instantiations that might make it difficult to understand the program.
- Ada does not permit "template metaprogramming", because it does not allow specialisations.

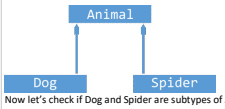
Templates in C++[edit]

Main article: Template (C++)

C++ uses templates to enable generic programming techniques. The C++ Standard Library includes the Standard Template Library or STL that provides a framework of templates for common data structures and algorithms. Templates in C++ may also be used for template metaprogramming, which is a way of pre-evaluating some of the code at compile-time rather than run-time. Using template specialization, C++ Templates are considered Turing complete.



abstract class Animal(val size: Int)
class Dog(val cuteness: Int): Animal(100)
class Spider(val terrorFactor: Int): Animal(1)



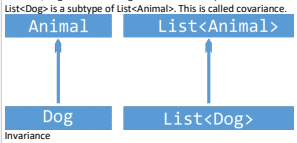
Now let's check if Dog and Spider are subtypes of Animal:

val dog: Dog = Dog(10)
val spider: Spider = Spider(9000)
var animal: Animal = dog
animal = spider
Covariance
Let's use some more complex types by wrapping our types into generic Lists.

One important thing to remember, this list is an immutable List from Kotlin: you are not able to modify its contents after you create it. You will find out why this matters shortly.

val dogList: List<Dog> = listOf(Dog(10), Dog(20))
val animalList: List<Animal> = dogList
Variance tells us about the relationship between List<Dog> and List<Animal> where Dog is a subtype of Animal.

In Kotlin, dogList can be assigned to Animal list (val animalList: List<Animal> = dogList) so the type relation is preserved and List<Dog> is a subtype of List<Animal>. This is called covariance.



Invariance
In Java, even though Dog is a subtype of Animal, you cannot do the following:

List<Dog> dogList= new ArrayList<>();
List<Animal> animalList = dogList; // Compiler error
This is because generics in Java ignore type vs subtype relation between its components. In the case when List<Dog> cannot be assigned to List<Animal> nor vice versa, this is called invariance. There is no subtype to supertype relationship here.

Contravariance
Perhaps we want to compare our animals, that's why we can create an interface Compare<T>, with a method compare(T item1, T item2) and that method can say which item is first and which item is second.

Whenever we compare dogs we look how cute are the dogs, here is code for comparing dogs:

```
val dogCompare: Compare<Dog> = object: Compare<Dog> {
  override fun compare(first: Dog, second: Dog): Int {
    return first.cuteness - second.cuteness
  }
}
```

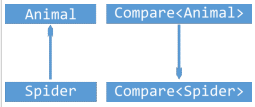
What will happen if we would try to assign this compare mechanism to animal comparator:

val animalCompare: Compare<Animal> = dogCompare // Compiler error
There is a really good reason why this does not work. If it worked, we would be able to pass spiders to animalCompare, but this would be an error because dogCompare can only compare dogs and not spiders.

On the other hand, if we would have a way to compare all the animals, that mechanism ought to work for dogs and spiders:

```
val animalCompare: Compare<Animal> = object: Compare<Animal> {
  override fun compare(first: Animal, second: Animal): Int {
    return first.size - second.size
  }
}
```

val spiderCompare: Compare<Spider> = animalCompare // Works nicely!
We can see that Spider is a subtype of Animal, but Compare<Animal> is a subtype of Compare<Spider> — the type relation is reversed. It's also known as contravariance.



11. What is the meaning of Object-Oriented-Programming (OOP)?

Object-oriented programming (OOP) refers to a type of computer programming (software design) in which programmers define not only the data type of a data structure, but also the types of operations (functions) that can be applied to the data structure.

In this way, the data structure becomes an object that includes both data and functions. In addition, programmers can create relationships between one object and another. For example, objects can inherit characteristics from other objects.

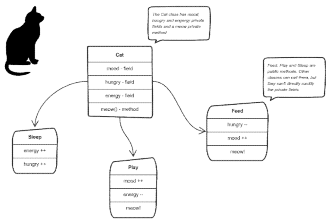
The four principles of object-oriented programming are encapsulation, abstraction, inheritance, and polymorphism.

Encapsulation
Say we have a program. It has a few logically different objects which communicate with each other — according to the rules defined in the program.

Encapsulation is achieved when each object keeps its state private, inside a class. Other objects don't have direct access to this state. Instead, they can only call a list of public functions — called methods.

So, the object manages its own state via methods — and no other class can touch it unless explicitly allowed. If you want to communicate with the object, you should use the methods provided. But (by default), you can't change the state.

Let's say we're building a tiny Sims game. There are people and there is a cat. They communicate with each other. We want to apply encapsulation, so we encapsulate all "cat" logic into a Cat class. It may look like =>



Here the "state" of the cat is the private variables mood, hungry and energy. It also has a private method meow(). It can call it whenever it wants, the other classes can't tell the cat when to meow.

What they can do is defined in the public methods sleep(), play() and feed(). Each of them modifies the internal state somehow and may invoke meow(). Thus, the binding between the private state and public methods is made.
This is encapsulation.

Abstraction:
Abstraction can be thought of as a natural extension of encapsulation.

In object-oriented design, programs are often extremely large. And separate objects communicate with each other a lot. So maintaining a large codebase like this for years — with changes along the way — is difficult.

Abstraction is a concept aiming to ease this problem.

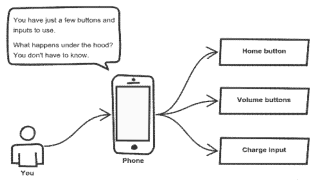
Applying abstraction means that each object should only expose a high-level mechanism for using it.

This mechanism should hide internal implementation details. It should only reveal operations relevant for the other objects.

Think — a coffee machine. It does a lot of stuff and makes quirky noises under the hood. But all you have to do is put in coffee and press a button.

Preferably, this mechanism should be easy to use and should rarely change over time. Think of it as a small set of public methods which any other class can call without "knowing" how they work.

Another real-life example of abstraction?
Think about how you use your phone:



You interact with your phone by using only a few buttons. What's going on under the hood? You don't have to know — implementation details are hidden. You only need to know a short set of actions.

Implementation changes — for example, a software update — rarely affect the abstraction you use.

Inheritance
OK, we saw how encapsulation and abstraction can help us develop and maintain a big codebase.

But do you know what is another common problem in OOP design?

Objects are often very similar. They share common logic. But they're not entirely the same. Ugh...

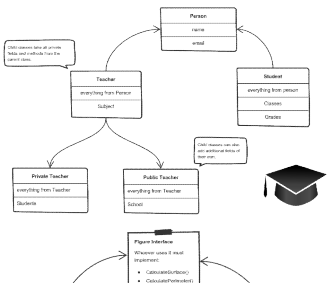
So how do we reuse the common logic and extract the unique logic into a separate class? One way to achieve this is inheritance.

It means that you create a (child) class by deriving from another (parent) class. This way, we form a hierarchy.

The child class reuses all fields and methods of the parent class (common part) and can implement its own (unique part).

If our program needs to manage public and private teachers, but also other types of people like students, we can implement this class hierarchy.

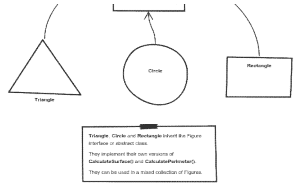
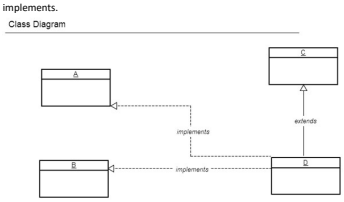
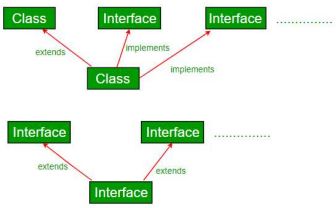
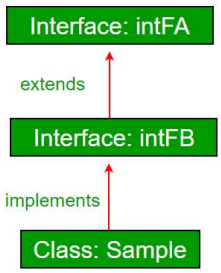
This way, each class adds only what is necessary for it while reusing common logic with the parent classes.



Polymorphism
We're down to the most complex word! Polymorphism means "many shapes" in Greek.

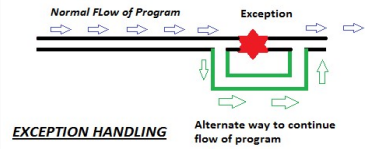
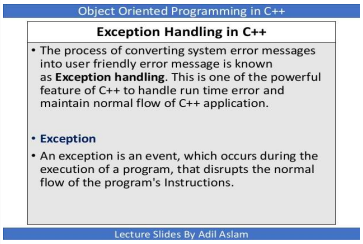
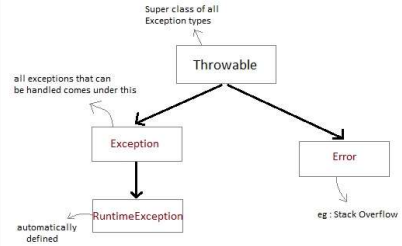
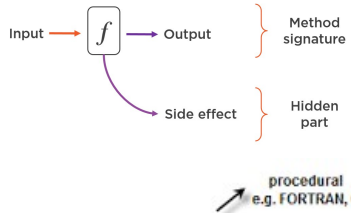
So we already know the power of inheritance and happily use it. But there comes this problem.

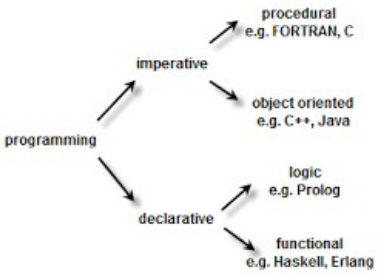
Say we have a parent class and a few child classes which inherit from it. Sometimes we want to use a collection — for example a list

		— which contains a mix of all these classes. Or we have a method implemented for the parent class — but we'd like to use it for the children, too. This can be solved by using polymorphism. Simply put, polymorphism gives a way to use a class exactly like its parent so there's no confusion with mixing types. But each child class keeps its own methods as they are. This typically happens by defining a (parent) interface to be reused. It outlines a bunch of common methods. Then, each child class implements its own version of these methods. Any time a collection (such as a list) or a method expects an instance of the parent (where common methods are outlined), the language takes care of evaluating the right implementation of the common method — regardless of which child is passed. Take a look at a sketch of geometric figures implementation. They reuse a common interface for calculating surface area and perimeter:	 Having these three figures inheriting the parent Figure Interface lets you create a list of mixed triangles, circles, and rectangles. And treat them like the same type of object. Then, if this list attempts to calculate the surface for an element, the correct method is found and executed. If the element is a triangle, triangle's CalculateSurface() is called. If it's a circle — then cirlce's CalculateSurface() is called. And so on. If you have a function which operates with a figure by using its parameter, you don't have to define it three times — once for a triangle, a circle, and a rectangle. You can define it once and accept a Figure as an argument. Whether you pass a triangle, circle or a rectangle — as long as they implement CalculateParamter(), their type doesn't matter.
12.	In OOP, what is the meaning of inheritance, composition, polymorphism.	Abstraction One of the most fundamental concept of OOPs is Abstraction. Abstraction is a process where you show only "relevant" data and "hide" unnecessary details of an object from the user. For example, when you login to your Amazon account online, you enter your user_id and password and press login, what happens when you press login, how the input data sent to amazon server, how it gets verified is all abstracted away from the you. Another example of abstraction: A car in itself is a well-defined object, which is composed of several other smaller objects like a gearing system, steering mechanism, engine, which are again have their own subsystems. But for humans car is a one single object, which can be managed by the help of its subsystems, even if their inner details are unknown. encapsulation is: Binding the data with the code that manipulates it. It keeps the data and the code safe from external interference Looking at the example of a power steering mechanism of a car. Power steering of a car is a complex system, which internally have lots of components tightly coupled together, they work synchronously to turn the car in the desired direction. It even controls the power delivered by the engine to the steering wheel. But to the external world there is only one interface is available and rest of the complexity is hidden. Moreover, the steering unit in itself is complete and independent. It does not affect the functioning of any other mechanism. Similarly, same concept of encapsulation can be applied to code. Encapsulated code should have following characteristics: Everyone knows how to access it. Can be easily used regardless of implementation details. There shouldn't any side effects of the code, to the rest of the application. The idea of encapsulation is to keep classes separated and prevent them from having tightly coupled with each other. A example of encapsulation is the class of java.util.Hashtable. User only knows that he can store data in the form of key/value pair in a Hashtable and that he can retrieve that data in the various ways. But the actual implementation like, how and where this data is actually stored, is hidden from the user. User can simply use Hashtable wherever he wants to store Key/Value pairs without bothering about its implementation.	In programming terms, an object is a self-contained component that contains properties and methods needed to make a certain type of data useful. An object's properties are what it knows and its methods are what it can do. The project management application mentioned above had a status object, a cost object, and a client object, among others. One property of the status object would be the current status of the project. The status object would have a method that could update that status. The client object's properties would include all of the important details about the client and its methods would be able to change them. The cost object would have methods necessary to calculate the project's cost based on hours worked, hourly rate, materials cost, and fees. In addition to providing the functionality of the application, methods ensure that an object's data is used appropriately by running checks for the specific type of data being used. They also allow for the actual implementation of tasks to be hidden and for particular operations to be standardized across different types of objects. You will learn more about these important capabilities in Object-oriented concepts: Encapsulation. Objects are the fundamental building blocks of applications from an object-oriented perspective. You will use many objects of many different types in any application you develop. Each different type of object comes from a specific class of that type. Classes, instances, and instantiation A class is a blueprint or template or set of instructions to build a specific type of object. Every object is built from a class. Each class should be designed and programmed to accomplish one, and only one, thing. (You'll learn more about the Single Responsibility Principle in Object-oriented programming concepts: Writing classes.) Because each class is designed to have only a single responsibility, many classes are used to build an entire application. An instance is a specific object built from a specific class. It is assigned to a reference variable that is used to access all of the instance's properties and methods. When you make a new instance the process is called instantiation and is typically done using the new keyword. Think about classes, instances, and instantiation like baking a cake. A class is like a recipe for chocolate cake. The recipe itself is not a cake. You can't eat the recipe (or at least wouldn't want to). If you correctly do what the recipe tells you to do (instantiate it) then you have an edible cake. That edible cake is an instance of the chocolate cake class. You can bake as many cakes as you would like using the same chocolate cake recipe. Likewise, you can instantiate as many instances of a class as you would like. Pretend you are baking three cakes for three friends who all have the same birthday but are different ages. You will need some way to keep track of which cake is for which friend so you can put on the correct number of candles. A simple solution is to write each friend's name on the cake. Reference variables work in a similar fashion. A reference variable provides a unique name for each instance of a class. In order to work with a particular instance, you use the reference variable it is assigned to. Just as a cookbook contains numerous recipes, ActionScript 3 includes a number of classes that you can use. A very commonly used class in ActionScript 3 is Sprite. A Sprite object can display graphics and contain children (learn about display objects in ActionScript 3 fundamentals: Display objects and the display list - available soon)
13.	What is an interface? Interfaces fulfill two goals: 1. They allow the programmer to be more abstract when referencing objects (for example, var vehicle : Vehicle, can reference any car, truck, etc... anything that is a vehicle (and not care what type it is.) This occurs at "program time". When the vehicle.start_engine() function is invoked, the correct function associated with the real object is actually used. This occurs at "run time". 2. They require the programmer to create specific functions that are expected in an implementing class when it **implements** an Interface. Again, this allows all objects in a "set" of like objects to be treated based on the "high level" type of the set, rather than on the specific type of the individual object.	**Interfaces in Object Oriented Programming Languages** An interface is a programming structure/syntax that allows the computer to enforce certain properties on an object (class). For example, say we have a car class and a scooter class and a truck class. Each of these three classes should have a start_engine() action. How the "engine is started" for each vehicle is left to each particular class, but the fact that **they must** have a start_engine action is the domain of the interface. **The syntax of an Interface** An interface has a very simple syntax that looks very much like a class definition... public interface XYZZY. Inside the {} of the interface is a list of functions that must be found in any object that purports to "follow" the interface. Interfaces are placed in their own files which have the same name as the interface (are Capitalized) and end with the familiar language extension (e.g., ".as"). The following interface would be placed in a "Vehicle.as" file.	**Interfaces in Object Oriented Programming Languages** An interface is a programming structure/syntax that allows the computer to enforce certain properties on an object (class). For example, say we have a car class and a scooter class and a truck class. Each of these three classes should have a start_engine() action. How the "engine is started" for each vehicle is left to each particular class, but the fact that **they must** have a start_engine action is the domain of the interface. **The syntax of an Interface** An interface has a very simple syntax that looks very much like a class definition... public interface XYZZY. Inside the {} of the interface is a list of functions that must be found in any object that purports to "follow" the interface. Interfaces are placed in their own files which have the same name as the interface (are Capitalized) and end with the familiar language extension (e.g., ".as"). The following interface would be placed in a "Vehicle.as" file.
14.	How is an interface different from inheritance? A class can extends another class and/ can implement one and more than one interface.	Interface is a 100% abstract class. It contains only constants and method signatures. In other words it is a reference type similar to class. An interface can't be instantiated. It can be implemented by a class or extended by another interface. The methods declared in an interface don't have method bodies. By default all the methods in an interface are public abstract. Similarly all the variables we define in an interface are essentially constants because they are implicitly public static final. So, the following definition of interface is equivalent to the above definition. An interface defines a contract which an implementing class must adhere to. The above interface DriveCar defines a set of operations that must be supported by a Car. So, a class that actually implements the interface should implement all the methods declared in the interface. Interfaces act as APIs (Application Programming Interfaces). Let us take an example of an image processing company which writes various classes to provide the image processing functionalities. So, a nice approach will be creating interfaces, declaring the methods in them and making the classes implement them. In this way the software package can be delivered to the clients and the clients can invoke the methods by looking at the method signatures declared inside the interfaces. They won't see the actual implementation of the methods. As a result the implementation part will be a secret. Later on the company may decide to re-implement the methods in another way. But the clients are concerned about the interfaces only. Interfaces provide an alternative to multiple inheritance. Java programming language does not support multiple inheritance. But interfaces provide a good solution. Any class can implement a particular interface and importantly the interfaces are not a part of class hierarchy. So, the general rule is extend one but implement many. A class can extend just one class but it can implement many interfaces. So, here we have multiple types for a class. It can be of the type of its super class and all the interfaces it implements. 	 Interface inheritance : An Interface can extend other interface. interface inheritance  Interface inheritance : An Interface can extend other interface.

15.	What is an abstract class, in OOP?
	
	An abstract class is a template definition of methods and variables of a class (category of objects) that contains one or more abstracted methods. Abstract classes are used in all object-oriented programming (OOP) languages
		A
		Abstract classes are very similar to interfaces but they have different rules that a web developer need to be aware of.
		
		The definition is quite straightforward, basically, an abstract class is a class that contains abstract methods,
		
		Abstract methods are methods that have been declared but not implemented in the code.
		
		This kind of classes cannot be constructed as an object, in fact, they need to be extended.
		
		Consider an interface as an empty shell.
		
		It has only signatures of the methods, which means the methods do not have a body.
		
		An interface does not do anything is just a pattern, a set of methods that you need to replicate.
		
		On the other hands, abstract classes are proper classes.
		Even though they look like interfaces the have some specific characteristics.
			A 'normal' class that extends an abstract one must implement all methods that are inside the parent.
			
			Something to notice is that a child class must define the abstract methods with the same or less restrictive visibility,
			
			This means that if in the abstract class there is a protected method it can be implemented only as a public or protected.
			
			Another rule is that the number and type or required arguments need to be the same as the parameters.
			
			`abstract class Building {`
			`  abstract protected function getWindowsCount();`
			`  public function __construct()`
			`  {`
			`  echo "Building cannot be constructed";`
			`  }`
			`}`

		You can specify their use by what they do by a given method name.	8
			9 class Office extends Building {
			10 public function getWindowsCount()
			11 {
			12 return 12;
			13 }
			14 }
			15
			16 $myOffice = new Office;
			17 // This command will return "Building cannot be constructed"
			18 $myOffice->getWindowsCount();
			19 // This command will return 12
			
			have defined the Office class that extends the abstract class *Building*.
			
			I have implemented *getWindowsCount()* and changed its visibility from protected to public.
			
			There are no abstract methods within the *Office* class thus, I can construct an object from it.
			
			I have added a constructor inside the abstract class so that you will be aware of the fact the parent's constructor is called when the object is instantiated.

16.	What is exception handling?	Exception handling is the process of responding to the occurrence, during computation, of exceptions – anomalous or exceptional conditions requiring special processing – often disrupting the normal flow of program execution. It is provided by specialized programming language constructs, computer hardware mechanisms like interrupts or operating system IPC facilities like signals. In general, an exception breaks the normal flow of execution and executes a pre-registered exception handler. The details of how this is done depends on whether it is a hardware or software exception and how the software exception is implemented. Some exceptions, especially hardware ones, may be handled so gracefully that execution can resume where it was interrupted. Alternative approaches to exception handling in software are error checking, which maintains normal program flow with later explicit checks for contingencies reported using special return values or some auxiliary global variable such as C's errno or floating point status flags; or input validation to preemptively filter exceptional cases. In Software: Software exception handling and the support provided by software tools differs somewhat from what is understood by exception handling in hardware, but similar concepts are involved. In programming language mechanisms for exception handling, the term exception is typically used in a specific sense to denote a data structure storing information about an exceptional condition. One mechanism to transfer control, or raise an exception, is known as a throw. The exception is said to be thrown. Execution is transferred to a "catch". From the point of view of the author of a routine, raising an exception is a useful way to signal that a routine could not execute normally - for example, when an input argument is invalid (e.g. value is outside of the domain of a function) or when a resource it relies on is unavailable (like a missing file, a hard disk error, or out-of-memory errors). In systems without exceptions, routines would need to return some special error code. However, this is sometimes complicated by the semipredicate problem, in which users of the routine need to write extra code to distinguish normal return values from erroneous ones. Programming languages differ substantially in their notion of what an exception is. Contemporary languages can roughly be divided into two groups:[7] Languages where exceptions are designed to be used as flow control structures: Ada, Java, Modula-3, ML, OCaml, Python, and Ruby fall in this category. Languages where exceptions are only used to handle abnormal, unpredictable, erroneous situations: C++,[8] C#, Common Lisp, Eiffel, and Modula-2. Kiniry also notes that "Language design only partially influences the use of exceptions, and consequently, the manner in which one handles partial and total failures during system execution. The other major influence is examples of use, typically in core libraries and code examples in technical books, magazine articles, and online discussion forums, and in an organization's code standards."[7] Contemporary applications face many design challenges when considering exception handling strategies. Particularly in modern enterprise level applications, exceptions must often cross process boundaries and machine boundaries. Part of designing a solid exception handling strategy is recognizing when a process has failed to the point where it cannot be economically handled by the software portion of the process.[9]	  
17.	What is functional programming? Functional programming is a programming paradigm — a style of building the structure and elements of computer programs — that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data	It is a declarative programming paradigm, which means programming is done with expressions or declarations[1] instead of statements. Functional code is idempotent, the output value of a function depends only on the arguments that are passed to the function, so calling a function f twice with the same value for an argument x produces the same result f(x) each time; this is in contrast to procedures depending on a local or global state, which may produce different results at different times when called with the same arguments but a different program state. Eliminating side effects, i.e., changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming. Functional programming has its origins in lambda calculus, a formal system developed in the 1930s to investigate computability, the Entscheidungsproblem, function definition, function application, and recursion. Many functional programming languages can be viewed as elaborations on the lambda calculus. Another well-known declarative programming paradigm, logic programming, is based on relations.[2] In contrast, imperative programming changes state with commands in the source code, the simplest example being assignment. Imperative programming does have subroutine functions, but these are not functions in the mathematical sense. They can have side effects that may change the value of program state. Functions without return values therefore make sense. Because of this, they lack referential transparency, i.e., the same language expression can result in different values at different times depending on the state of the executing program.[2] Functional programming languages have largely been emphasized in academia rather than in commercial software development. However, prominent programming languages that support functional programming such as Common Lisp, Scheme,[3][4][5][6] Clojure, Wolfram Language[7][8] (also known as Mathematica), Racket,[9] Erlang,[10][11][12] OCaml,[13][14] Haskell,[15][16] and F#[17][18] have been used in industrial and commercial applications by a wide variety of organizations. JavaScript, one of the world's most widely distributed languages,[19][20][21] has the properties of a dynamically typed functional language,[20] in addition to imperative and object-oriented paradigms. Functional programming is also key to some languages that have found success in specific domains, like R (statistics),[22][23] J, K and Q from Kx Systems (financial analysis), XQuery/XSLT (XML),[24][25] and Opal. Widespread domain-specific declarative languages like SQL and Lex/Yacc use some elements of functional programming, especially in eschewing mutable values.[26] Programming in a functional style can also be accomplished in languages that are not specifically designed for functional programming. For example, the imperative Perl programming language has been the subject of a book describing how to apply functional programming concepts.[27] This is also true of the PHP programming language.[28] C++11, Java 8, and C# 3.0 all added constructs to facilitate the functional style. The Julia language also offers functional programming abilities. An interesting case is that of Scala[29] – it is frequently written in a functional style, but the presence of side effects and mutable state place it in a grey area between imperative and functional languages. Purely functional data structures are often represented in a different way than their imperative counterparts.[68] For example, the array with constant access and update times is a basic component of most imperative languages, and many imperative data-structures, such as the hash table and binary heap, are based on arrays. Arrays can be replaced by maps or random access lists, which admit purely functional implementation, but have logarithmic access and update times. Purely functional data structures have persistence, a property of keeping previous versions of the data structure unmodified. In Clojure, persistent data structures are used as functional alternatives to their imperative counterparts. Persistent vectors, for example, use trees for partial updating. Calling the insert method will result in some but not all nodes being created.[69]  	The first fundamental concept we learn when we want to understand functional programming is pure functions. But what does that really mean? What makes a function pure? So how do we know if a function is pure or not? Here is a very strict definition of purity: It returns the same result if given the same arguments (it is also referred as deterministic) It does not cause any observable side effects It returns the same result if given the same arguments Imagine we want to implement a function that calculates the area of a circle. An impure function would receive radius as the parameter, and then calculate radius * radius * PI. In Clojure, the operator comes first, so radius * radius * PI becomes (* radius radius PI): Why is this an impure function? Simply because it uses a global object that was not passed as a parameter to the function. Higher-order functions are functions that can either take other functions as arguments or return them as results. In calculus, an example of a higher-order function is the differential operator {\displaystyle d/dx} d/dx, which returns the derivative of a function {\displaystyle f} f. Higher-order functions are closely related to first-class functions in that higher-order functions and first-class functions both allow functions as arguments and results of other functions. The distinction between the two is subtle: "higher-order" describes a mathematical concept of functions that operate on other functions, while "first-class" is a computer science term that describes programming language entities that have no restriction on their use (thus first-class functions can appear anywhere in the program that other first-class entities like numbers can, including as arguments to other functions and as their return values). Higher-order functions enable partial application or currying, a technique that applies a function to its arguments one at a time, with each application returning a new function that accepts the next argument. This lets a programmer succinctly express, for example, the successor function as the addition operator partially applied to the natural number one. Pure functions Pure functions (or expressions) have no side effects (memory or I/O). This means that pure functions have several useful properties, many of which can be used to optimize the code: If the result of a pure expression is not used, it can be removed without affecting other expressions. If a pure function is called with arguments that cause no side-effects, the result is constant with respect to that argument list (sometimes called referential transparency), i.e., calling the pure function again with the same arguments returns the same result. (This can enable caching optimizations such as memoization.) If there is no data dependency between two pure expressions, their order can be reversed, or they can be performed in parallel and they cannot interfere with one another (in other terms, the evaluation of any pure expression is thread-safe). If the entire language does not allow side-effects, then any evaluation strategy can be used; this gives the compiler freedom to reorder or combine the evaluation of expressions in a program (for example, using deforestation). While most compilers for imperative programming languages detect pure functions and perform common-subexpression elimination for pure function calls, they cannot always do this for pre-compiled libraries, which generally do not expose this information, thus preventing optimizations that involve those external functions. Some compilers, such as gcc, add extra keywords for a programmer to explicitly mark external functions as pure, to enable such optimizations. Fortran 95 also lets functions be designated pure.[44] C++11 added constexpr keyword with similar semantics. Recursion Main article: Recursion (computer science) Iteration (looping) in functional languages is usually accomplished via recursion. Recursive functions invoke themselves, letting an operation be repeated until it reaches the base case. Although some recursion requires maintaining a stack, tail recursion can be recognized and optimized by a compiler into the same code used to implement iteration in imperative languages. The Scheme language standard requires implementations to recognize and optimize tail recursion. Tail recursion optimization can be implemented by transforming the program into continuation passing style during compiling, among other approaches. Common patterns of recursion can be abstracted away using higher-order functions, with catamorphisms and anamorphisms (or "folds" and "unfolds") being the most obvious examples. Such recursion schemes play a role analogous to built-in control structures such as loops in imperative languages. Most general purpose functional programming languages allow unrestricted recursion and are Turing complete, which makes the halting



and "unfolds") being the most obvious examples. Such recursion schemes play a role analogous to built-in control structures such as loops in imperative languages.

Most general purpose functional programming languages allow unrestricted recursion and are Turing complete, which makes the halting problem undecidable, can cause unsoundness of equational reasoning, and generally requires the introduction of inconsistency into the logic expressed by the language's type system. Some special purpose languages such as Coq allow only well-founded recursion and are strongly normalizing (nonterminating computations can be expressed only with infinite streams of values called codata). As a consequence, these languages fail to be Turing complete and expressing certain functions in them is impossible, but they can still express a wide class of interesting computations while avoiding the problems introduced by unrestricted recursion. Functional programming limited to well-founded recursion with a few other constraints is called total functional programming.[45]

18.

What is the pointer?

A pointer is essentially a variable that holds the address of another variable or function. Pointers may be used to perform what is known in the assembly language world as indirect addressing. When you use pointers, you are able to access the variable that they point to, and you are able to change the pointer so that it points to a different variable. This simple mechanism has hundreds of uses and is one of the most powerful features of the C programming language.

In computer science, a pointer is a programming language object that stores the memory address of another value located in computer memory. A pointer references a location in memory, and obtaining the value stored at that location is known as dereferencing the pointer. As an analogy, a page number in a book's index could be considered a pointer to the corresponding page; dereferencing such a pointer would be done by flipping to the page with the given page number and reading the text found on that page. The actual format and content of a pointer variable is dependent on the underlying computer architecture.

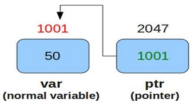
Using pointers significantly improves performance for repetitive operations like traversing iterable data structures, e.g. strings, lookup tables, control tables and tree structures. In particular, it is often much cheaper in time and space to copy and dereference pointers than it is to copy and access the data to which the pointers point.

Pointers are also used to hold the addresses of entry points for called subroutines in procedural programming and for run-time linking to dynamic link libraries (DLLs). In object-oriented programming, pointers to functions are used for binding methods, often using what are called virtual method tables.

A pointer is a simple, more concrete implementation of the more abstract reference data type. Several languages, especially low-level languages support some type of pointer, although some have more restrictions on their use than others. While "pointer" has been used to refer to references in general, it more properly applies to data structures whose interface explicitly allows the pointer to be manipulated (arithmetically via pointer arithmetic) as a memory address, as opposed to a magic cookie or capability which does not allow such.[citation needed] Because pointers allow both protected and unprotected access to memory addresses, there are risks associated with using them, particularly in the latter case. Primitive pointers are often stored in a format similar to an integer; however, attempting to dereference or "look up" such a pointer whose value is not a valid memory address will cause a program to crash. To alleviate this potential problem, as a matter of type safety, pointers are considered a separate type parameterized by the type of data they point to, even if the underlying representation is an integer. Other measures may also be taken (such as validation & bounds checking), to verify that the pointer variable contains a value that is both a valid memory address and within the numerical range that the processor is capable of addressing.

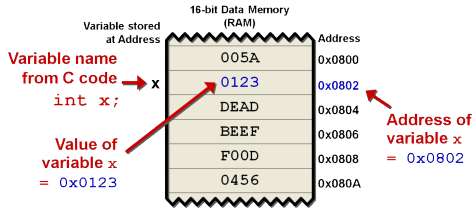
Different from other normal variables which can store values.
pointers are special variables that can hold the address of a variable. Since they store memory address of a variable, the pointers are very commonly said to "point to variables".

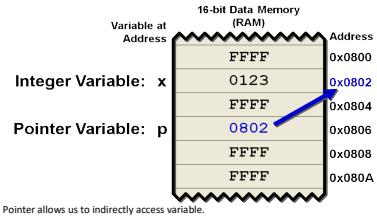
Lets try to understand the concept.



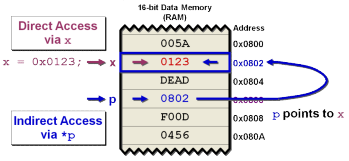
As shown in the above diagram:

- A normal variable 'var' has a memory address of 1001 and holds a value 50
- A **pointer variable** has its own address 2047 but stores 1001, which is the address of the variable 'var'





Pointer allows us to indirectly access variable.



When setting up data structures like lists, queues and trees, it is necessary to have pointers to help manage how the structure is implemented and controlled. Typical examples of pointers are start pointers, end pointers, and stack pointers. These pointers can either be absolute (the actual physical address or a virtual address in virtual memory) or relative (an offset from an absolute start address ("base") that typically uses fewer bits than a full address, but will usually require one additional arithmetic operation to resolve).

Relative addresses are a form of manual memory segmentation, and share many of its advantages and disadvantages. A two-byte offset, containing a 16-bit, unsigned integer, can be used to provide relative addressing for up to 64 kilobytes of a data structure. This can easily be extended to 128K, 256K or 512K if the address pointed to is forced to be aligned on a half-word, word or double-word boundary (but, requiring an additional "shift left" bitwise operation—by 1, 2 or 3 bits—in order to adjust the offset by a factor of 2, 4 or 8, before its addition to the base address). Generally, though, such schemes are a lot of trouble, and for convenience to the programmer absolute addresses (and underlying that, a flat address space) is preferred.

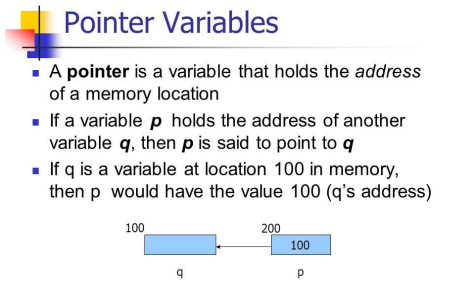
A one byte offset, such as the hexadecimal ASCII value of a character (e.g. X'29') can be used to point to an alternative integer value (or index) in an array (e.g., X'01'). In this way, characters can be very efficiently translated from 'raw data' to a usable sequential index and then to an absolute address without a lookup table.
Pointers are directly supported without restrictions in languages such as PL/I, C, C++, Pascal, FreeBASIC, and implicitly in most assembly languages. They are primarily used for constructing references, which in turn are fundamental to constructing nearly all data structures, as well as in passing data between different parts of a program.

In functional programming languages that rely heavily on lists, data references are managed abstractly by using primitive constructs like cons and the corresponding elements car and cdr, which can be thought of as specialised pointers to the first and second components of a cons-cell. This gives rise to some of the idiomatic "flavour" of functional programming. By structuring data in such cons-lists, these languages facilitate recursive means for building and processing data—for example, by recursively accessing the head and tail elements of lists of lists; e.g. "taking the car of the cdr of the cdr". By contrast, memory management based on pointer dereferencing in some approximation of an array of memory addresses facilitates treating variables as slots into which data can be assigned imperatively.

When dealing with arrays, the critical lookup operation typically involves a stage called address calculation which involves constructing a pointer to the desired data element in the array. In other data structures, such as linked lists, pointers are used as references to explicitly tie one piece of the structure to another.

Pointers are used to pass parameters by reference. This is useful if the programmer wants a function's modifications to a parameter to be visible to the function's caller. This is also useful for returning multiple values from a function.

Pointers can also be used to allocate and deallocate dynamic variables and arrays in memory. Since a variable will often become redundant after it has served its purpose, it is a waste of memory to keep it, and therefore it is good practice to deallocate it (using the original pointer reference) when it is no longer needed. Failure to do so may result in a memory leak (where available free memory gradually, or in severe cases rapidly, diminishes because of an accumulation of numerous redundant memory blocks).



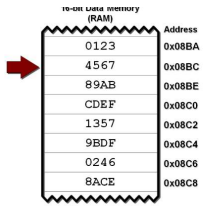
Example:
Data Buffer

```
1  //Point to RAM buffer starting address
2  char *bufPtr = &buffer;
3
4  while ((DataAvailable) && (ReceivedCharacter != '\0'))
5  {
6     //Read byte from UART and write it to RAM buffer
7     ReadUART(bufPtr);
8     //Point to next available byte in RAM buffer
9     bufPtr++;
10 }
```

RAM buffer allocated over a range of addresses
(perhaps an array)

Pseudo-code:

1. Point arrow to first address of buffer
2. Write data from UART to location pointed to by arrow
3. Move arrow to point to next address in buffer
4. Repeat until data from UART is 0, or buffer is full (arrow points to last address of buffer)



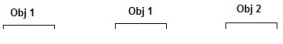
Pointers may be used in many ways. Some of the more common include being used in conjunctio with dynamic memory allocation. A pointer may point to a location in the heap where a variable i created at runtime since the variable doesn't have a name in the same way as variables w explicitly declare before compilation.

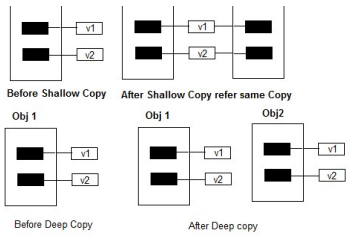
More frequently, you will see pointers used as a mechanism to pass multiple parameters to or fror a function, or as a way to pass a value *BY REFERENCE* to a function. Since C ordinarily passe variables to functions *BY VALUE*, the original variable is never changed. In some cases we want t modify the original variable as a part of the function's operation. Therefore, we can pass a pointer t the function, and it will then be able to modify the original variable (much more on this later Pointers also provide an alternate means of accessing arrays, which can be more efficien especially when dealing with strings

19.

What is the difference between "deep copy" and "shallow copy"?

Shallow copying is creating a new object and then copying the non static fields of the current object to the new object. If the field is a value type, a bit by bit copy of the field is performed. If the field is a reference type, the reference is copied but the referred



	An object copy is a process where a data object has its attributes copied to another object of the same data type. In .Net Shallow copy and deep copy are used for copying data between objects.	object is not, therefore the original object and its clone refer to the same object. A shallow copy of an object is a new object whose instance variables are identical to the old object. In .Net shallow copy is done by the object method MemberwiseClone(). The situations like , if you have an object with values and you want to create a copy of that object in another variable from same type, then you can use shallow copy, all property values which are of value types will be copied, but if you have a property which is of reference type then this instance will not be copied, instead you will have a reference to that instance only. Deep copy is creating a new object and then copying the non-static fields of the current object to the new object. If a field is a value type, a bit by bit copy of the field is performed. If a field is a reference type, a new copy of the referred object is performed. A deep copy of an object is a new object with entirely new instance variables, it does not share objects with the old. While performing Deep Copy the classes to be cloned must be flagged as [Serializable]. Deep copy is intended to copy all the elements of an object, which include directly referenced elements of value type and the indirectly referenced elements of a reference type that holds a reference to a memory location that contains data rather than containing the data itself.	
20.	What is a null reference?	You have a function that takes 3 parameters, but 2 of them are optional. If you call that function with 2 parameters, does that make that 3rd parameter null or undefined? Just a little pop quiz. The answer is undefined. What is the difference between the two values? null is an object that is a assigned where the null value is a member of every type, defined to have a neutral behavior that has no value undefined is the evaluation of a variable or value that has been declared but not yet been assigned While undefined has been in existence since the creation of coding, null is the misguided invention of British computer scientist Tony Hoare (most famous for his Quicksort algorithm) in 1964, who coined his invention of null references as his "billion dollar mistake".	**NULL POINTER** • To overcome the problem of Dangling or Wild pointer, we use NULL pointer. • A NULL value will be assigned after deleting the object or the pointer. • A null pointer is a regular pointer of any pointer type which has a special value that indicates that it is not pointing to any valid reference or memory address. This value is the result of type-casting the integer value zero to any pointer type. `int * p;` `p = 0;` // p has a null pointer value 28 Ritika sharma

21.	What is the difference between build-in types and user defined types?
	
	Built in Data types are those that a programming language natively supports e.g. int, float, char, double.
	User Defined Data types are those that are structured by users i.e. programmers for the sake of convinience e.g. array, you don't have any data type in C where you can store a list of integer elements, other examples include queue, stack.
		n computer programming, data type is a classification that specifies to compiler or interpreter which type of data user is intending to use.
		There are two types of data types –
		
		Primitive/Fundamental data type : Each variable in C/C++ has an associated data type. Each data type requires different amounts of memory and has some specific operations which can be performed over it.
		Example of fundamental data types –
		
		Derived data type : These data types are defined by user itself. Like, defining a class in C++ or a structure. These include Arrays, Structures, Class, Union, Enumeration, Pointers etc.
		Examples of derived data type :
		Pointer
		
		FUNDAMENTAL DATA TYPES / DERIVED DATA TYPES
		Fundamental data type is also called primitive data type. These are the basic data types. / Derived data type is the aggregation of fundamental data type.
		character, integer, float, and void are fundamental data types. / Pointers, arrays, structures and unions are derived data types.
		Character is used for characters. It can be classified as char, Signed char, Unsigned char. / Pointers are used for storing address of variables.
		Integer is used for integers(not having decimal digits). It can be classified as signed and unsigned. Further classified as int, short int and long int. / Array is used to contain similar type of data.
		float is used for decimal numbers. These are classified as float, double and long double. / structure is used to group items of possibly different types into a single type.
		void is used where there is no return value required. / It is like structure but all members in union share the same memory location
			C++ Java C#
			`// C program to illustrate`
			`// primitive data types`
			`#include <stdio.h>`
			`// main method starts`
			`int main()`
			`{`
			`// Integer value`
			`int a = 2;`
			`// Float value`
			`float b = 2.0;`
			`// Double value`
			`double c = 2.0003;`
			`// Character`
			`char d = 'D';`
			`printf("Integer value is = %d", a);`
			`printf("\nFloat value is = %f", b);`
			`printf("\nDouble value is = %lf", c);`
			`printf("\nChar value is = %c", d);`
			`return 0;`
			`}`
			
			Output:
			`Integer value is = 2`
			`Float value is = 2.000000`
			`Double value is = 2.000300`
			`Char value is = D`
			
			`// C program to illustrate pointer`
			`// as derived data type`
			`#include <stdio.h>`
			`// main method starts from here`
			`int main()`
			`{`
			`// integer variable`
			`int variable = 10;`
			`// Pointer for storing address`
			`int* pointr;`
			`// Assigning address of variable to pointer`
			`pointr = &variable;`
			`printf("Value of variable = %d", variable);`
			`// printf("\nValue at pointer = %d", pointr);`
			`printf("\nValue at *pointer = %d", *pointr);`
			`return 0;`
			`}`
			
			`Value of variable = 10`
			`Value at *pointer = 10`

- Array :

C Java C#

```c
// C program to illustrate array
// derived data type
#include <stdio.h>
// main method starts from here
int main()
{
    // array of size 5
    int a[5] = { 1, 2, 3, 4, 5 };

    // indexing variable
    int i;
    for (i = 0; i < 5; i++)
        printf("%d  ", a[i]);
    return 0;
}
```

Output:

```
1  2  3  4  5
```

- Structures –

```c
// C program to illustrate structure
// derived data type
#include <stdio.h>
// structure
struct structure_example {
    int integer;
    float decimal;
    char character[20];
};
// main method starts from here
void main()
{
    struct structure_example s = { 15, 98.9, "geeksforgeeks" };
    printf("Structure data - \n integer = %d \n decimal =
    %f \n name = %s", s.integer, s.decimal, s.character);
}
```

Output:

```
Structure data -
 integer = 15
 decimal = 98.900002
 name = geeksforgeeks
```

22. What is immutability.?

In object-oriented and functional programming, an immutable object (unchangeable[1] object) is an object whose state cannot be modified after it is created.[2] This is in contrast to a mutable object (changeable object), which can be modified after it is created. In some cases, an object is considered immutable even if some internally used attributes change, but the object's state appears unchanging from an external point of view. For example, an object that uses memoization to cache the results of expensive computations could still be considered an immutable object.

Strings and other concrete objects are typically expressed as immutable objects to improve readability and runtime efficiency in object-oriented programming. Immutable objects are also useful because they are inherently thread-safe.[2] Other benefits are that they are simpler to understand and reason about and offer higher security than mutable objects

Class	Description	Immutable?
bool	Boolean value	✓
int	integer (arbitrary magnitude)	✓
float	floating-point number	✓
list	mutable sequence of objects	
tuple	immutable sequence of objects	✓
str	character string	✓
set	unordered set of distinct objects	
frozenset	immutable form of set class	✓
dict	associative mapping (aka dictionary)	

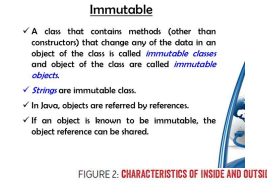


FIGURE 2: CHARACTERISTICS OF INSIDE AND OUTSIDE [IMMUTABLE] DATA

	INSIDE DATA	OUTSIDE DATA
Changeable	Yes!	No! Immutable
Granularity	Relational field	Document, file, or message
Representation	Typically relational	Typically semi-structured
Schema	Prescriptive	Descriptive
Identity	No identity: Data by values	Identity: URL, Msg#, Doc-ID...
Versioning	No versioning: data by value	Versions may augment identity

Immutable variables
In imperative programming, values held in program variables whose content never changes are known as constants to differentiate them from variables that could be altered during execution. Examples include conversion factors from meters to feet, or the value of pi to several decimal places.

Read-only fields may be calculated when the program runs (unlike constants, which are known beforehand), but never change after they are initialized.

Weak vs strong immutability
Sometimes, one talks of certain fields of an object being immutable. This means that there is no way to change those parts of the object state, even though other parts of the object may be changeable (weakly immutable). If all fields are immutable, then the object is immutable. If the whole object cannot be extended by another class, the object is called strongly immutable.[3] This might, for example, help to explicitly enforce certain invariants about certain data in the object staying the same through the lifetime of the object. In some languages, this is done with a keyword (e.g. const in C++, final in Java) that designates the field as immutable. Some languages reverse it: in OCaml, fields of an object or record are by default immutable, and must be explicitly marked with mutable to be so.

References to objects
In most object-oriented languages, objects can be referred to using references. Some examples of such languages are Java, C++, C#, VB.NET, and many scripting languages, such as Perl, Python, and Ruby. In this case, it matters whether the state of an object can vary when objects are shared via references.

Copying objects
If an object is known to be immutable, it can be copied simply by making a copy of a reference to it instead of copying the entire object. Because a reference (typically only the size of a pointer) is usually much smaller than the object itself, this results in memory savings and a potential boost in execution speed.

The reference copying technique is much more difficult to use for mutable objects, because if any user of a mutable object reference changes it, all other users of that reference see the change. If this is not the intended effect, it can be difficult to notify the other users to have them respond correctly. In these situations, defensive copying of the entire object rather than the reference is usually an easy but costly solution. The observer pattern is an alternative technique for handling changes to mutable objects.

Copy-on-write
A technique that blends the advantages of mutable and immutable objects, and is supported directly in almost all modern hardware, is copy-on-write (COW). Using this technique, when a user asks the system to copy an object, it instead merely creates a new reference that still points to the same object. As soon as a user attempts to modify the object through a particular reference, the system makes a real copy, applies the modification to that, and sets the reference to refer to the new copy. The other users are unaffected, because they still refer to the original object. Therefore, under COW, all users appear to have a mutable version of their objects, although in the case that users do not modify their objects, the space-saving and speed advantages of immutable objects are preserved. Copy-on-write is popular in virtual memory systems because it allows them to save memory space while still correctly handling anything an application program might do.

23. What is atomicity?

In computer programming, an operation done by a computer is considered atomic if it is guaranteed to be isolated from other operations that may be happening at the same time. Put another way, atomic operations are indivisible.

Atomic operations are critically important when dealing with shared resources.

Imagine two people, Sam and Sally, are trying to buy an item online using the same bank account with a $100 balance:

1. Sam clicks "buy now" on a $100 Lego set.
2. Banking software checks to see that at least $100 is in its account database, and confirms that that there is enough to make the purchase.
3. Sally clicks "buy now" on a $50 video game.
4. Banking software checks to see that at least $50 is in its account database and confirms that that there is enough to buy the game.
5. The bank withdraws $100 to pay for Sam's Lego set. (Balance is now $0.)
6. The bank withdraws $50 to pay for Sally's video game. (Balance is now $-50.)

The account is now overdrawn because the banking transaction (checking for and withdrawing funds) is not atomic. To correct this problem, the banking software would need some form of mutual exclusion to ensure that the multi-step process of checking funds and withdrawing funds is indivisible.

One approach is to use a *lock*, which we set before accessing the shared resource and then release when we are finished. Other threads must wait until the lock is released to use the shared resource. The code within the block is called a *critical section*.

```
acquire lock
  /* Entering critical section */
  if $100 is availabile
    withdraw $100
  /* Leaving critical section */
release lock
```

If the bank software used the above algorithm, the sequence of events between Sam and Sally would be:

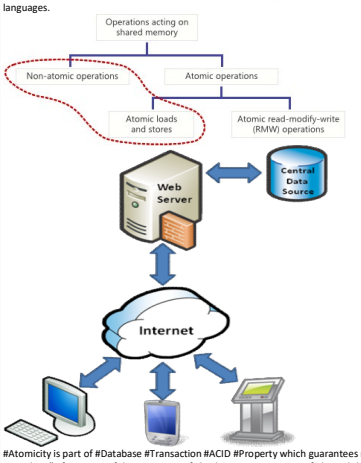
7. Sam clicks "buy now" on a $100 Lego set.
8. Sam's request acquires the lock, checks to see that at least $100 is in its account database, and confirms that that there is enough to make the purchase.
9. Sally clicks "buy now" on a $50 video game.
10. The banking software waits on the lock—it's currently in use by Sam's request.
11. The bank withdraws $100 to pay for Sam's Lego set (balance is now $0), and releases the lock.

Objects
In Object-Oriented programming, a class is a type of thing, and an object is a specific example of that thing. For example, for the class Car, you might have objects like John's Mazda3 or Stephanie's Civic.

A class works like a template for creating objects. Each class has properties that define it. The class Car might have properties like:

Brand
Model
Year Manufactured
Color
Number of Doors
The object John's Mazda3 is a Mazda, a model 3, was manufactured in 2008, is gray, and has five doors. The object Stephanie's Civic is a Honda, a Civic, was manufactured in 2003, is beige, and has four doors.

Both John's Mazda3 and Stephanie's Civic are objects of the type Car. Objects are a fundamental programming concept in most modern languages.



#Atomicity is part of #Database #Transaction #ACID #Property which guarantees reliability. Atomicity requires that each transaction is "all or nothing": if one part of the transaction fails, the entire transaction fails, and the database state is left unchanged.

12. Sally's request acquires the lock, checks to see if at least $50 is in the account database, and sees that there is not enough.
13. Sally is unable to buy the video game.

Atomicity is a essential feature of many computer systems, multi-threading, databases, parallel processing, memory management, etc.

or nothing : If one part of the transaction fails, the entire transaction fails, and the database state is left unchanged

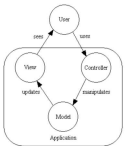
24. What are the design patterns (as they apply to programming)?

Explain the producer-consumer pattern?
What is the client-server model?

What is the meaning of MVC - Model-View-Controller model

Design patterns are used to represent some of the best practices adapted by experienced object-oriented software developers. A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints and examples.

Figure 1 - The basic MVC relationship



Design Patterns | Set 1 (Introduction)
Design pattern is a general reusable solution or template to a commonly occurring problem in software design. The patterns typically show relationships and interactions between classes or objects. The idea is to speed up the development process by providing tested, proven development paradigm.

Goal:
• Understand the problem and matching it with some pattern.
• Reusage of old interface or making the present design reusable for the future usage.

Example:
For example, in many real world situations we want to create only one instance of a class. For example, there can be only one active president of country at a time regardless of personal identity. This pattern is called Singleton pattern. Other software examples could be a single DB connection shared by multiple objects as creating a separate DB connection for every object may be costly. Similarly, there can be a single configuration manager or error manager in an application that handles all problems instead of creating multiple managers.

Types of Design Patterns
There are mainly three types of design patterns:
1. Creational
These design patterns are all about class instantiation or object creation. These patterns can be further categorized into Class-creational patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.

Creational design patterns are Factory Method, Abstract Factory, Builder, Singleton, Object Pool and Prototype.

2. Structural
These design patterns are about organizing different classes and objects to form larger structures and provide new functionality.

Structural design patterns are Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Private Class Data and Proxy.

3. Behavioral
Behavioral patterns are about identifying common communication patterns between objects and realize these patterns.

Behavioral patterns are Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Null Object, Observer, State, Strategy, Template method, Visitor

Design Patterns | Set 2 (Factory Method)
Factory method is a creational design pattern, i.e., related to object creation. In Factory pattern, we create object without exposing the creation logic to client and the client use the same common interface to create new type of object.
The idea is to use a static member-function (static factory method) which creates & returns instances, hiding the details of class modules from user.

A factory pattern is one of the core design principles to create an object, allowing clients to create objects of a library(explained below) in a way such that it doesn't have tight coupling with the class hierarchy of the library.

What is meant when we talk about library and clients?
A library is something which is provided by some third party which exposes some public APIs and clients make calls to those public APIs to complete its task. A very simple example can be different kinds of Views provided by Android OS.

The client-server model is a distributed communication framework of network processes among service requestors, clients and service providers. The client-server connection is established through a network or the Internet.

The client-server model is a core network computing concept also building functionality for email exchange and Web/database access. Web technologies and protocols built around the client-server model are:
Hypertext Transfer Protocol (HTTP)
Domain Name System (DNS)
Simple Mail Transfer Protocol (SMTP)
Telnet
Clients include Web browsers, chat applications, and email software, among others. Servers include Web, database, application, chat and email, etc.

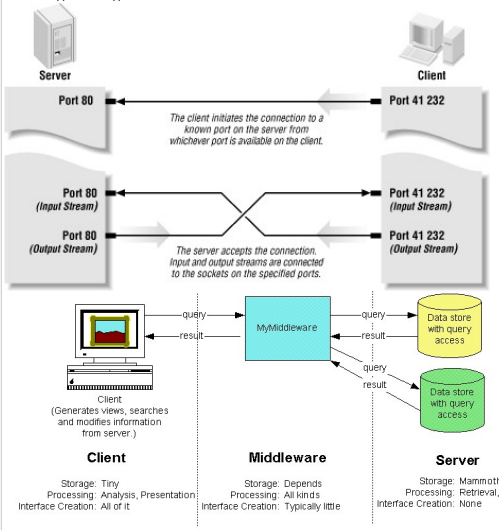
Help us Help You | 2019 Techopedia Reader Survey | Complete this Short 1 Minute Survey
Techopedia explains Client-Server Model
A server manages most processes and stores all data. A client requests specified data or processes. The server relays process output to the client. Clients sometimes handle processing, but require server data resources for completion.

The client-server model differs from a peer-to-peer (P2P) model where communicating systems are the client or server, each with equal status and responsibilities. The P2P model is decentralized networking. The client-server model is centralized networking.

One client-server model drawback is having too many client requests underrun a server and lead to improper functioning or total shutdown. Hackers often use such tactics to terminate specific organizational services through distributed denial-of-service (DDoS) attacks.
Most modern network programming is based on a client/server model. A client/server application typically stores large quantities of data on an expensive, high-powered server, while most of the program logic and the user interface is handled by client software running on relatively cheap personal computers. In most cases, a server primarily sends data, while a client primarily receives it, but it is rare for one program to send or receive exclusively. A more reliable distinction is that a client initiates a conversation, while a server waits for clients to start conversations with it. Figure 2.5 illustrates both possibilities. In some cases, the same program may be both a client and a server.

A client/server connection
Figure 2-5. A client/server connection

Some servers process and analyze the data before sending the results to the client. Such servers are often referred to as "application servers" to distinguish them from the more common file servers and database servers. A file or database server will retrieve information and send it to a client, but it won't process that information. In contrast, an application server might look at an order entry database and give the clients reports about monthly sales trends. An application server is not a server that serves files that happen to be applications.



Stands for "Model-View-Controller." MVC is an application design model comprised of three interconnected parts. They include the model (data), the view (user interface), and the controller (processes that handle input).

A classic concurrent programming design pattern is *producer-consumer*, where processes are designated as either producers or consumers. The producers are responsible for adding to some shared data structure and the consumers are responsible for removing from that structure. Only one party, either a single producer or a single consumer, can access the structure at any given time.
Here we consider an example with a shared queue, using a mutex (introduced previously) to protect the queue:

```
let f = Queue.create() and m = Mutex.create()
```

We divide the work of a producer into two parts: `produce`, which simulates the work of creating a product, and `store`, which adds the product to the shared queue. The `produce`operation increments the counter `p` that it is passed, sleeps for `d` seconds to simulate the time taken to produce, and outputs a status message:

```
let produce i p d =
incr p;
Thread.delay d;
Printf.printf "Producer %d has produced %d" i !p;
print_newline()
```

The operation `store` acquires the mutex `m`, adds to the shared queue, outputs a status message and releases the mutex:

```
let store i p =
Mutex.lock m;
Queue.add (i, !p) f;
Printf.printf "Producer %d has added its %d-th product" i !p;
print newline();
Mutex.unlock m
```

The `producer` loops `n` times. Each time through the loop, it calls `produce` and then `store`, then sleeps for a random interval of time up to 2.5 seconds. When it is done looping, it outputs a status message.

```
let producer (n, i) =
let p = ref 0
and d = Random.float 2. in
for j = 1 to n do
produce i p d;
store i p;
Thread.delay (Random.float 2.5)
done;
Printf.printf "Producer %d is exiting." i;
print_newline()
```

The `consumer` loops `n` times, acquiring the mutex `m`, then attempting to take an item from the shared queue. If it succeeds, it prints out the item. If not, it prints out that it failed to get an item. In either event, it unlocks the mutex and then waits a random interval of time up to 2.5 seconds. When it is done looping, it outputs a status message.

```
let consumer (n, i) =
for j = 1 to n do
Mutex.lock m;
(try
let ip, p = Queue.take f
in
Printf.printf "Consumer %d has taken product (%d,%d)" i ip p;
print_newline()
with Queue.Empty ->
Printf.printf "Consumer %d has returned empty-handed" i;
print_newline());
Mutex.unlock m;
Thread.delay (Random.float 2.5)
done;
Printf.printf "Consumer %d is exiting." i;
print_newline()
```

This use of mutual exclusion is very coarse grained. It would be better to be able to have a consumer wait until something is in the queue rather than returning empty handed. For this we can make use of condition variables, which were introduced previously. Now the store function `store2` signals the condition `c` to indicate that something is in the queue.

```
let c = Condition.create()

let store2 i p =
Mutex.lock m;
Queue.add (i, !p) f;
Printf.printf "Producer %d has added its %d-th product" i !p;
print newline();
Condition.signal c;
Mutex.unlock m
```

Now we split the consumer function `consumer2` into two parts, `wait` and `take`.

```
let wait i =
Mutex.lock m;
while Queue.length f = 0 do
Printf.printf "Consumer %d is waiting" i;
print_newline();
Condition.wait c m
done

let take i =
let ip, p = Queue.take f in
Printf.printf "Consumer %d has taken product (%d,%d)" i ip p;
print newline();
Mutex.unlock m

let consumer2 (n, i) =
for j = 1 to n do
wait i;
take i;
Thread.delay (Random.float 2.5)
done;
Printf.printf "Consumer %d is exiting." i;
print_newline()
```

Note that `wait` locks the mutex, so no work should be done after wait returns unless it is intended to be in the critical section. Generally, a function that has the effect of locking or unlocking a mutex should be used with caution, and should be clearly documented as doing so.

Thread Pools

A *thread pool* consists of a collection of threads, called *workers*, that are used to process work. Each worker looks for new work to be done. When it finds work to do, it does it, and when finished, it goes back to get more work. The workers play the role of consumers in the producer-consumer model that we just considered above. In fact, thread pool implementations often use a shared queue to store the work, thus building quite directly on the previous example. Before considering implementation of thread pool, let's get a better idea of what it does and where it is useful.
The basic operations on a thread pool are:

1. create a new thread pool with some specified number of workers,
2. add work to an existing thread pool, which will subsequently be performed by one of the workers, and
3. destroy an existing thread pool, shutting it down once all previously added work is complete.

Here is the signature for a basic thread pool:

```
module type SIMPLE_THREAD_POOL = sig
type pool
(* A No_workers exception is thrown if addwork is called when the
threadpool is being shut down.  The work is not added. *)
exception No_workers
(* create a thread pool with the specified number of worker threads *)
val create: int -> pool
(* add work to the pool, where work is any unit -> unit function *)
val addwork: (unit -> unit) -> pool -> unit
(* destroy a thread pool, stopping all the threads once all work
* in the pool has been completed. *)
val destroy: pool -> unit
end
```

Thread pools are particularly useful in settings where work arrives asynchronously, such as occurs with a server where many network requests may need to be handled promptly. In such settings, a thread receives an event such as a network request, adds the corresponding work to a thread pool (which will be run at some point in the future), and then quickly returns indicating to the caller that the request will be handled. Sometimes it is also useful to have a handle associated with each unit of work to which some value is sent. The simple abstraction that we presented here does not have any means of returning a result, as the functions representing work are of type `unit -> unit`.
Here we consider an implementation of the `SIMPLE_THREAD_POOL` interface in terms of a 4-tuple: a mutable counter, a mutable queue of functions that are the work remaining to be done, a mutex, and a condition variable. The mutex is used to protect the counter and the queue. The condition variable is used to signal when a worker should wake up to get new work.

```
module Tpool : SIMPLE_THREAD_POOL = struct
type pool = int ref * (unit -> unit) Queue.t * Mutex.t * Condition.t
exception No_workers
let dowork tp = ...
let create size = ...
let addwork f tp = ...
let rec done_wait tp n = ...
let destroy tp = ...
end
```

If the counter is positive, it indicates the number of worker threads that the thread pool was created with. If the counter is non-positive, it indicates that the thread pool is being destroyed, and the absolute value of the counter is the number of threads that have properly exited. This allows the destroy function to wait for the threads to finish their work and exit before returning.
Each worker thread runs the function `dowork`. This function is not exposed in the interface, so it can only be called from inside the implementation of `Tpool`. The function `dowork` loops as long as the thread pool has not finished its work. When the work is finished, it exits. A thread pool is finished when it is being destroyed and there is no work remaining to do. We use the counter in the 4-tuple, here called `nworkers`, to indicate that the thread pool is being destroyed by setting its value to something less than 1. In that case, if the queue of work is also empty, then the thread exits as the pool is finished. Otherwise, on each loop the worker waits for work to do, and then takes that work from the queue, executing it inside a try to ensure that unhandled exceptions in the work do not cause the worker to exit.

```
  let dowork (tp : pool) =
match tp with (nworkers, q, m, c) ->
Mutex.lock m;
(* When nworkers <= 0, the thread pool is being
* destroyed.  If that is true and there is also no work left to do
* then stop looping and drop through to exit processing. *)
while (!nworkers > 0) || (Queue.length q > 0) do
(* If nworkers > 0, wait for stuff in the queue. *)
while (!nworkers > 0) && (Queue.length q = 0) do
dbgprint "waiting";
Condition.wait c m
done;
(* Verify something in the queue rather than we are now being
* shut down, then get the work from the queue, unlock the
```

The MVC model or "pattern" is commonly used for developing modern user interfaces. It is provides the fundamental pieces for designing a programs for desktop or mobile, as well as web applications. It works well with object-oriented programming, since the different models, views, and controllers can be treated as objects and reused within an application.

Below is a description of each aspect of MVC:

1. Model
A model is data used by a program. This may be a database, file, or a simple object, such as an icon or a character in a video game.

2. View
A view is the means of displaying objects within an application. Examples include displaying a window or buttons or text within a window. It includes anything that the user can see.

3. Controller
A controller updates both models and views. It accepts input and performs the corresponding update. For example, a controller can update a model by changing the attributes of a character in a video game. It may modify the view by displaying the updated character in the game.

The three parts of MVC are interconnected (see diagram). The view displays the model for the user. The controller accepts user input and updates the model and view accordingly. While MVC is not required in application design, many programming languages and IDEs support the MVC architecture, making it an common choice for developers.

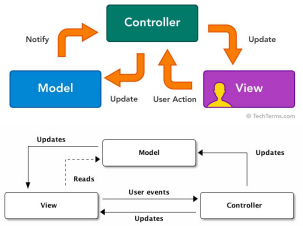
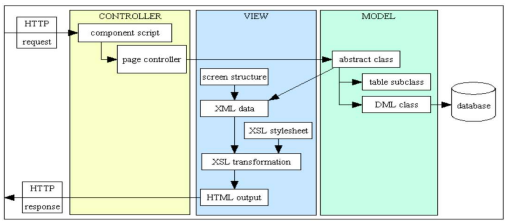


Figure 3 - My implementation of the MVC pattern (2)




```
* mutex, do the work and relock the mutex before looping
* back. *)
if Queue.length q > 0
then
let f = Queue.take q in
dbgprint "starting work";
Mutex.unlock m;
(* Don't let an exception in the work, f, kill the thread,
* just catch it and go on. *)
(try ignore (f()) with _ -> ());
Mutex.lock m
done;
(* A worker thread exits when the pool is being shut down.  It
* decrements the worker count, which should be -n when all
* threads are finished, where n was the number of threads in
* the pool (counts down from 0). *)
decr nworkers;
dbgprint "exiting";
Mutex.unlock m
```

Note the use of the mutex in the 4-tuple, called `m` here, to protect accesses to the queue and the counter. Before entering the while loop, and at the end of the while loop (thus before the next iteration of the loop), the mutex must be locked because both the counter and the queue are accessed. The nested while loop checking for work to do uses `Condition.wait` to release the mutex and sleep until it receives the condition of work being ready. Recall that this reacquires the mutex before returning. It is important that the mutex is then unlocked before calling `f`, the work to be done. This allows other workers to safely run concurrently, as `f` cannot access the queue or counter. Then the mutex is reacquired before the end of the while loop. When exiting the while loop, the mutex is already locked, so the counter is simply decremented and then the mutex is released before the worker exits.
The `create` function makes a 4-tuple and starts up the specified number of threads, each of which runs the `dowork` function. The `create` function simply acquires the mutex at the beginning and releases it at the end. There are other possible ways of writing this code in which the mutex is not held during the entire process of creating the thread pool.

```
  let create size =
if size < 1
then failwith "Tpool.create needs at least one thread"
else let tp =
(ref 0, Queue.create(), Mutex.create(), Condition.create()) in
match tp with (nworkers, _, m, _) ->
Mutex.lock m;
while !nworkers < size do
ignore (Thread.create dowork tp);
incr nworkers
done;
Mutex.unlock m;
tp
```

The `add_work` function adds the given function to the queue. To do so, it first locks the mutex and checks whether the pool is being destroyed. If the pools is being destroyed, instead of adding the work it raises the `No_workers` exception after first releasing the mutex. If the pool is not being destroyed, it adds the work to the queue, signals that there is work to be done, and unlocks the mutex.

```
  let addwork (f : unit -> unit) (tp : pool) =
match tp with (nworkers, q, m, c) ->
Mutex.lock m;
if !nworkers < 1
then (Mutex.unlock m; raise No_workers)
else
(Queue.add f q;
Condition.signal c;
Mutex.unlock m)
```

The `destroy` function acquires the mutex, then sets the number of workers to zero to indicate that the thread pool is being shut down. It then broadcasts to all the workers that there is something to be done to ensure that all the workers exit, including ones that were currently sleeping while awaiting work. Finally, it releases the mutex, then waits for all the workers to exit using the helper function `done_wait`.

```
  let destroy (tp : pool) =
match tp with (nworkers, _, m, c) ->
Mutex.lock m;
let n = !nworkers in
nworkers := 0;
Condition.broadcast c;
Mutex.unlock m;
done_wait tp n
```

From <https://www.cs.cornell.edu/courses/cs3110/2010fa/lectures/lec18.html>

25. What is the difference between "compile-time" and "run-time

- **Run-time**
  - Imperative
  - Object-oriented
  - Generative
  - (some) Functional
- **Compile-time**
  - Pure Functional

Runtime and compile time are programming terms that refer to different stages of software program development. In order to create a program, a developer first writes source code, which defines how the program will function. Small programs may only contain a few hundred lines of source code, while large programs may contain hundreds of thousands of lines of source code. The source code must be compiled into machine code in order to become and executable program. This compilation process is referred to as compile time.

A compiled program can be opened and run by a user. When an application is running, it is called runtime.

The terms "runtime" and "compile time" are often used by programmers to refer to different types of errors. A compile time error is a problem such as a syntax error or missing file reference that prevents the program from successfully compiling. The compiler produces compile time errors and usually indicates what line of the source code is causing the problem.

If a program's source code has already been compiled into an executable program, it may still have bugs that occur while the program is running. Examples include features that don't work, unexpected program behavior, or program crashes. These types of problems are called runtime errors since they occur at runtime.

Overloading and overriding2Polymorphism is one of the crucial features of OOP. It simply means 'using one name for multiple forms'. Polymorphism can be implemented using 'function overloading', 'operator overloading' and 'virtual function'. Both, 'overloading' and 'overriding' implies the concept of polymorphism. Here, 'overloading' is compile time polymorphism and 'overriding' is run time polymorphism. Studying further, if we talk about the major difference in 'overloading' and 'overriding'. In 'overloading' we redefine the overloaded functions with the same function name but, different number and type of parameters. In 'overriding' prototype of overridden function is same throughout the program but, function to be overridden is preceded by the keyword 'virtual' in the base class and is redefined by the derived class without any keyword.

- **Run-time**
  - Functions
  - Values, literals
  - Data structures
  - If/else
  - Loop
  - Assignment
  - May depend on input
- **Compile-time**
  - Metafunctions (type)
  - Const, enum,constexpr
  - Typelist (type)
  - Pattern matching
  - Recursion
  - Ref. Transparency
  - Deterministic

26. What is Scalability?
Scalability is an attribute that describes the ability of a process, network, software or organization to grow and manage increased demand. A system, business or software that is described as scalable has an advantage because it is more adaptable to the changing needs or demands of its users or clients.

Scalability is often a sign of stability and competitiveness, as it means the network, system, software or organization is ready to handle the influx of demand, increased productivity, trends, changing needs and even presence or introduction of new competitors.

Scalability is the capability of a system, network, or process to handle a growing amount of work, or its potential to be enlarged to accommodate that growth.[1] For example, a system is considered scalable if it is capable of increasing its total output under an increased load when resources (typically hardware) are added. An analogous meaning is implied when the word is used in an economic context, where a company's scalability implies that the underlying business model offers the potential for economic growth within the company.

Scalability, as a property of systems, is generally difficult to define[2] and in any particular case it is necessary to define the specific requirements for scalability on those dimensions that are deemed important. It is a highly significant issue in electronics systems, databases, routers, and networking. A system whose performance improves after adding hardware, proportionally to the capacity added, is said to be a scalable system.

An algorithm, design, networking protocol, program, or other system is said to scale if it is suitably efficient and practical when applied to large situations (e.g. a large input data set, a large number of outputs or users, or a large number of participating nodes in the case of a distributed system). If the design or system fails when a quantity increases, it does not scale. In practice, if there are a large number of things (n) that affect scaling, then resource requirements (for example, algorithmic time-complexity) must grow less than n2 as n increases. An example is a search engine, which scales not only for the number of users, but also for the number of objects it indexes. Scalability refers to the ability of a site to increase in size as demand warrants

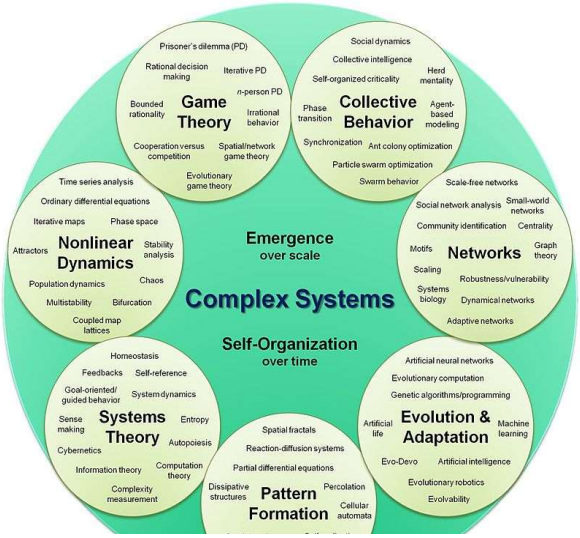
Scalability can be measured in various dimensions, such as:

Administrative scalability: The ability for an increasing number of organizations or users to easily share a single distributed system.
Functional scalability: The ability to enhance the system by adding new functionality at minimal effort.
Geographic scalability: The ability to maintain performance, usefulness, or usability regardless of expansion from concentration in a local area to a more distributed geographic pattern.
Load scalability: The ability for a distributed system to easily expand and contract its resource pool to accommodate heavier or lighter loads or number of inputs. Alternatively, the ease with which a system or component can be modified, added, or removed, to accommodate changing load.
Generation scalability: The ability of a system to scale up by using new generations of components. Thereby, heterogeneous scalability is the ability to use the components from different vendors.[4]

A routing protocol is considered scalable with respect to network size, if the size of the necessary routing table on each node grows as O(log N), where N is the number of nodes in the network.
A scalable online transaction processing system or database management system is one that can be upgraded to process more transactions by adding new processors, devices and storage, and which can be upgraded easily and transparently without shutting it down.

Methods of adding more resources for a particular application fall into two broad categories: horizontal and vertical scaling.[5]

To scale horizontally (or scale out/in) means to add more nodes to (or remove nodes from) a system, such as adding a new

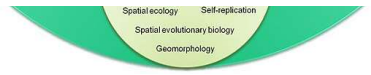


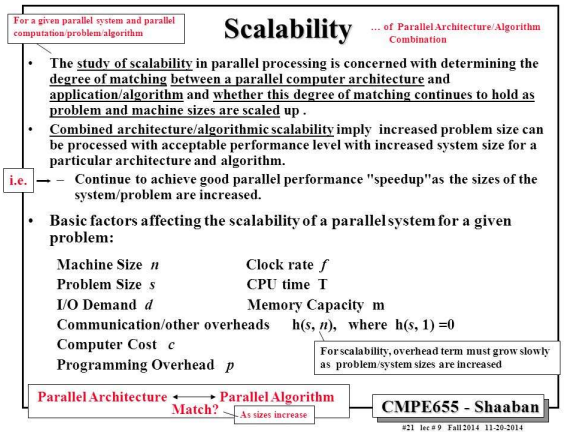
computer to a distributed software application. An example might involve scaling out from one Web server system to three. As computer prices have dropped and performance continues to increase, high-performance computing applications such as seismic analysis and biotechnology workloads have adopted low-cost "commodity" systems for tasks that once would have required supercomputers. System architects may configure hundreds of small computers in a cluster to obtain aggregate computing power that often exceeds that of computers based on a single traditional processor. The development of high-performance interconnects such as Gigabit Ethernet, InfiniBand and Myrinet further fueled this model. Such growth has led to demand for software that allows efficient management and maintenance of multiple nodes, as well as hardware such as shared data storage with much higher I/O performance. Size scalability is the maximum number of processors that a system can accommodate.[4]
To scale vertically (or scale up/down) means to add resources to (or remove resources from) a single node in a system, typically involving the addition of CPUs or memory to a single computer. Such vertical scaling of existing systems also enables them to use virtualization technology more effectively, as it provides more resources for the hosted set of operating system and application modules to share. Taking advantage of such resources can also be called "scaling up", such as expanding the number of Apache daemon processes currently running. Application scalability is the improved performance of running applications on a scaled-up version of the system.[4]
There are tradeoffs between the two models. Larger numbers of computers means increased management complexity, as well as a more complex programming model and issues such as throughput and latency between nodes; also, some applications do not lend themselves to a distributed computing model. In the past, the price difference between the two models has favored "scale up" computing for those applications that fit its paradigm, but recent advances in virtualization technology have blurred that advantage, since deploying a new virtual system over a hypervisor (where possible) is often less expensive than actually buying and installing a real one. Configuring an existing idle system has always been less expensive than buying, installing, and configuring a new one, regardless of the model.

One technique supported by most of the major database management system (DBMS) products is the partitioning of large tables, based on ranges of values in a key field. In this manner, the database can be scaled out across a cluster of separate database servers. Also, with the advent of 64-bit microprocessors, multi-core CPUs, and large SMP multiprocessors, DBMS vendors have been at the forefront of supporting multi-threaded implementations that substantially scale up transaction processing capacity.

Network-attached storage (NAS) and Storage area networks (SANs) coupled with fast local area networks and Fibre Channel technology enable still larger, more loosely coupled configurations of databases and distributed computing power. The widely supported X/Open XA standard employs a global transaction monitor to coordinate distributed transactions among semi-autonomous XA-compliant database resources. Oracle RAC uses a different model to achieve scalability, based on a "shared-everything" architecture that relies upon high-speed connections between servers.

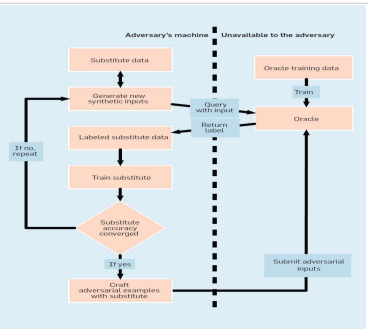
While DBMS vendors debate the relative merits of their favored designs, some companies and researchers question the inherent limitations of relational database management systems. GigaSpaces, for example, contends that an entirely different model of distributed data access and transaction processing, space-based architecture, is required to achieve the highest performance and scalability. On the other hand, Base One makes the case for extreme scalability without departing from mainstream relational database technology.[7] For specialized applications, NoSQL architectures such as Google's Bigtable can further enhance scalability. Google's horizontally distributed Spanner technology, positioned as an relational alternative to Bigtable[8], supports general-purpose database transactions and provides a more conventional SQL-based query language.[9]





27. What is Robustness?

In computer science, robustness is the ability of a computer system to cope with errors during execution[1][2] and cope with erroneous input.[2] Robustness can encompass many areas of computer science, such as robust programming, robust machine learning, and Robust Security Network. Formal techniques, such as fuzz testing, are essential to showing robustness since this type of testing involves invalid or unexpected inputs. Alternatively, fault injection can be used to test robustness. Various commercial products perform robustness testing of software analysis



28. What is a database?

What is SQL?
A pure relational database, as designed by Codd, is built on tuples grouped into relations, consistent with first-order predicate logic. Real-world relational databases have tables that contain fields, constraints, and triggers, and tables are related through foreign keys. SQL is used to declare the data to be returned, and a SQL query processor and query optimizer turn the SQL declaration into a query plan that is executed by the database engine.

SQL includes a sub-language for defining schemas, the data definition language (DDL), along with a sub-language for modifying data, the data manipulation language (DML). Both of these have roots in early Codasyl specifications. The third sub-language in SQL declares queries, through the SELECT statement and relational joins.

What is Normalization?

What is Data?
In simple words data can be facts related to any object in consideration.

For example your name, age, height, weight, etc are some data related to you.

A picture , image , file , pdf etc can also be considered data.

What is a Database?
Database is a systematic collection of data. Databases support storage and manipulation of data. Databases make data management easy. Let's discuss few examples.

An online telephone directory would definitely use database to store data pertaining to people, phone numbers, other contact details, etc.

Your electricity service provider is obviously using a database to manage billing , client related issues, to handle fault data, etc.

Let's also consider the facebook. It needs to store, manipulate and present data related to members, their friends, member activities, messages, advertisements and lot more.
What is a Database Management System (DBMS)?
Database Management System (DBMS) is a collection of programs which enables its users to access database, manipulate data, reporting / representation of data .

It also helps to control access to the database.

Database Management Systems are not a new concept and as such had been first implemented in 1960s.

Charles Bachmen's Integrated Data Store (IDS) is said to be the first DBMS in history.

With time database technologies evolved a lot while usage and expected functionalities of databases have been increased immensely.

Types of DBMS
Let's see how the DBMS family got evolved with the time. Following diagram shows the evolution of DBMS categories.

What is Database? What is SQL?

There are 4 major types of DBMS. Let's look into them in detail.

Hierarchical - this type of DBMS employs the "parent-child" relationship of storing data. This type of DBMS is rarely used nowadays. Its structure is like a tree with nodes representing records and branches representing fields. The windows registry used in Windows XP is an example of a hierarchical database. Configuration settings are stored as tree structures with nodes.
Network DBMS - this type of DBMS supports many-to many relations. This usually results in complex database structures. RDM Server is an example of a database management system that implements the network model.
Relational DBMS - this type of DBMS defines database relationships in form of tables, also known as relations. Unlike network DBMS, RDBMS does not support many to many relationships.Relational DBMS usually have pre-defined data types that they can support. This is the most popular DBMS type in the market. Examples of relational database management systems include MySQL, Oracle, and Microsoft SQL Server database.
Object Oriented Relation DBMS - this type supports storage of new data types. The data to be stored is in form of objects. The objects to be stored in the database have attributes (i.e. gender, ager) and methods that define what to do with the data. PostgreSQL is an example of an object oriented relational DBMS.
What is SQL?
Structured Query language (SQL) pronounced as "S-Q-L" or sometimes as "See-Quel"is actually the standard language for dealing with Relational Databases.

SQL programming can be effectively used to insert, search, update, delete database records.

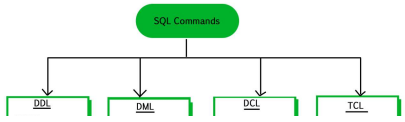
That doesn't mean SQL cannot do things beyond that.

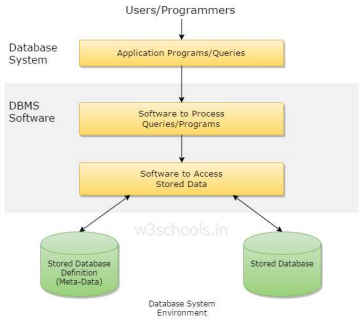
In fact it can do lot of things including, but not limited to, optimizing and maintenance of databases.

Relational databases like MySQL Database, Oracle, Ms SQL server, Sybase, etc uses SQL ! How to use sql syntaxes?

SQL syntaxes used in these databases are almost similar, except the fact that some are using few different syntaxes and even proprietary SQL syntaxes.

Today, Structured Query Language is the standard means of manipulating and querying data in relational databases, though with proprietary extensions among the products. The ease and ubiquity of SQL have even led the creators of many "NoSQL" or non-relational data stores, such as Hadoop, to adopt subsets of SQL or come up with their own SQL-like query languages.





The SELECT statement in SQL
The SELECT statement tells the query optimizer what data to return, what tables to look in, what relations to follow, and what order to impose on the returned data. The query optimizer has to figure out by itself what indexes to use to avoid brute force table scans and achieve good query performance, unless the particular database supports index hints.

Part of the art of relational database design hangs on the judicious use of indexes. If you omit an index for a frequent query, the whole database can slow down under heavy read loads. If you have too many indexes, the whole database can slow down under heavy write and update loads.

Another important art is choosing a good, unique primary key for every table. You not only have to consider the impact of the primary key on common queries, but how it will play in joins when it appears as a foreign key in another table, and how it will affect the data's locality of reference.

In the advanced case of database tables that are split up into different volumes depending on the value of the primary key, called horizontal sharding, you also have to consider how the primary key will affect the sharding. Hint: You want the table distributed evenly across volumes, which suggests that you don't want to use date stamps or consecutive integers as primary keys.

Discussions of the SELECT statement may start simple, but can quickly become confusing. Consider:

SELECT * FROM Customers;
Simple, right? It asks for all fields and all rows of the Customers table. Suppose, however, that the Customers table has a hundred million rows and a hundred fields, and one of the fields is a large text field for comments. How long will it take to pull down all that data over a 10 megabit per second network connection if each row contains an average of 1 kilobyte of data?

Perhaps you should cut down how much you send over the wire. Consider:

SELECT TOP 100 companyName, lastSaleDate, lastSaleAmount, totalSalesAmount FROM Customers
WHERE state ="Ohio" AND city ="Cleveland"
ORDER BY lastSaleDate DESCENDING;
Now you're going to pull down a lot less data. You have asked the database to give you only four fields, to only consider the companies in Cleveland, and to give you just the 100 companies with the most recent sales. To do that most efficiently at the database server, however, the Customers table needs an index on state+city for the WHERE clause and an index on lastSaleDate for the ORDER BY and TOP 100 clauses.

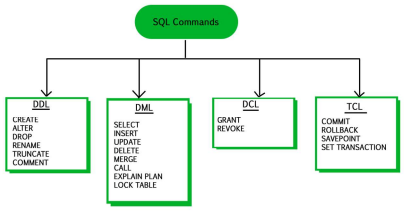
By the way, TOP 100 is valid for SQL Server and SQL Azure, but not MySQL or Oracle. In MySQL, you'd use LIMIT 100 after the WHERE clause. In Oracle, you'd use a bound on ROWNUM as part of the WHERE clause, i.e. WHERE... AND ROWNUM <=100. Unfortunately, the ANSI/ISO SQL standards (and there are nine of them to date, stretching from 1986 to 2016) only go so far, beyond which each database introduces its own proprietary clauses and features.

Relational joins in SQL
So far, I've described the SELECT syntax for single tables. Before I can explain JOIN clauses, you need to understand foreign keys and relations between tables. I'll explain this by using examples in DDL, using SQL Server syntax.

The short version of this is fairly simple. Every table that you want to use in relations should have a primary key constraint; this can either be a single field or a combination of fields defined by an expression. For example:

```
CREATE TABLE Persons (
   PersonID int NOT NULL PRIMARY KEY,
   PersonName char(80),
   ...
```

Every table that needs to relate to Persons should have a field that corresponds to the Persons primary key, and to preserve relational integrity that field should have a foreign key constraint. For example:



Domain Constraints : A domain of possible values must be associated with every attribute (for example, integer types, character types, date/time types). Declaring an attribute to be of a particular domain acts as the constraints on the values that it can take.
Referential Integrity : There are cases where we wish to ensure that a value appears in one relation for a given set of attributes also appear in a certain set of attributes in another relation i.e. Referential Integrity. For example, the department listed for each course must be one that actually exists.
Assertions : An assertion is any condition that the database must always satisfy. Domain constraints and Integrity constraints are special form of assertions.
Authorization : We may want to differentiate among the users as far as the type of access they are permitted on various data values in database. These differentiation are expressed in terms of Authorization. The most common being :
read authorization – which allows reading but not modification of data ;
insert authorization – which allow insertion of new data but not modification of existing data
update authorization – which allows modification, but not deletion.

DML (Data Manipulation Language) :
DML statements are used for managing data with in schema objects.
DML are of two types –

Procedural DMLs : require a user to specify what data are needed and how to get those data.
Declerative DMLs (also referred as Non-procedural DMLs) : require a user to specify what data are needed without specifying how to get those data.
Declarative DMLs are usually easier to learn and use than procedural DMLs. However, since a user does not have to specify how to get the data, the database system has to figure out an efficient means of accessing data.

TCL (Transaction Control Language) :
Transaction Control Language commands are used to manage transactions in the database. These are used to manage the changes made by DML-statements. It also allows statements to be grouped together into logical transactions.

DCL (Data Control Language) :
A Data Control Language is a syntax similar to a computer programming language used to control access to data stored in a database (Authorization). In particular, it is a component of Structured Query Language (SQL).

Relational databases and SQL
Why would you give up a doubling of execution speed and memory? There were two big reasons: ease of development and portability. I didn't think either one mattered much in 1980 compared to performance and memory requirements, but as computer hardware improved and became cheaper people stopped caring about execution speed and memory and worried more about the cost of development.

In other words, Moore's Law killed Codasyl databases in favor of relational databases. As it happened, the improvement in development time was significant, but SQL portability turned out to be a pipe dream.

Database normalization is the process of structuring a relational database[clarification needed] in accordance with a series of so-called normal forms in order to reduce data redundancy and improve data integrity. It was first proposed by Edgar F. Codd as an integral part of his relational model.

Normalization entails organizing the columns (attributes) and tables (relations) of a database to ensure that their dependencies are properly enforced by database integrity constraints. It is accomplished by applying some formal rules either by a process of synthesis (creating a new database design) or decomposition (improving an existing database design).

be a single field or a combination of fields defined by an expression. For example:

```
CREATE TABLE Persons (
   PersonID int NOT NULL PRIMARY KEY,
   PersonName char(80),
   ...
```

Every table that needs to relate to Persons should have a field that corresponds to the Persons primary key, and to preserve relational integrity that field should have a foreign key constraint. For example:

```
CREATE TABLE Orders (
   OrderID int NOT NULL PRIMARY KEY,
   ...
   PersonID int FOREIGN KEY REFERENCES Persons(PersonID)
);
```

There are longer versions of both statements that use the CONSTRAINT keyword, which lets you name the constraint. That's what most database design tools generate.

Primary keys are always indexed and unique (the field values cannot be duplicated). Other fields can optionally be indexed. It's often useful to create indexes for foreign key fields and for fields that appear in WHERE and ORDER BY clauses, although not always because of the potential overhead from writes and updates.

How would you write a query that returns all of the orders placed by John Doe?

```
SELECT PersonName, OrderID FROM Persons
INNER JOIN Orders ON Persons.PersonID = Orders.PersonID
WHERE PersonName ="John Doe";
```

In fact, there are four kinds of JOIN: INNER, OUTER, LEFT, and RIGHT. The INNER JOIN is the default (you can omit the word INNER), and it's the one that includes only rows that contain matching values in both tables. If you want to list persons whether or not they have orders, you'd use a LEFT JOIN, for example:

```
SELECT PersonName, OrderID FROM Persons
LEFT JOIN Orders ON Persons.PersonID = Orders.PersonID
ORDER BY PersonName;
```

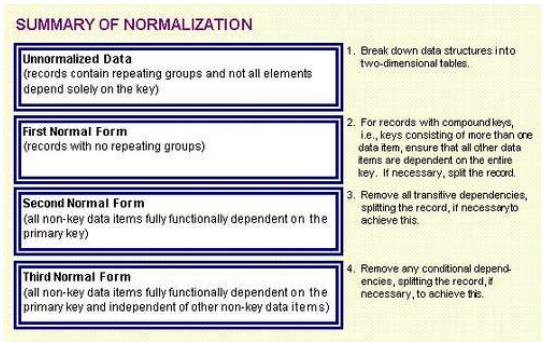
When you start doing queries that join more than two tables, use expressions, or coerce data types, the syntax can get a little hairy at first. Fortunately, there are database development tools that can generate correct SQL queries for you, often by dragging and dropping tables and fields from the schema diagram into a query diagram.

SQL stored procedures
Sometimes the declarative nature of the SELECT statement doesn't get you where you want to go. Most databases have a facility called stored procedures; unfortunately this is an area where the databases pretty much all use proprietary extensions to the ANSI/ISO SQL standards.

In SQL Server, the initial dialect for stored procedures (or stored procs) was Transact-SQL, aka T-SQL; in Oracle, it was PL-SQL. Both databases have added additional languages for stored procedures, such as C#, Java, and R. A simple T-SQL stored procedure might only be a parameterized version of a SELECT statement. Its advantages are ease of use and efficiency. Stored procedures are optimized when they are saved, not every time they are executed.

A more complicated T-SQL stored procedure might use multiple SQL statements, input and output parameters, local variables, BEGIN...END blocks, IF...THEN...ELSE conditions, cursors (row-by-row processing of a set), expressions, temporary tables, and a whole host of other procedural syntax. Obviously if the stored procedure language is C#, Java, or R, you're going to use the functions and syntax of those procedural languages. In other words, despite the fact that the motivation for SQL was to use standardized declarative queries, in the real world you see lots of database-specific procedural server programming.

That doesn't quite take us back to the bad old days of Codasyl database programming (although cursors come close), but it does walk back from the ideas that SQL statements should be standardized and that performance concerns should be left to the database query optimizer. In the end, a doubling of performance is often too much to leave on the table.



29. What is the difference between Unix and Windows

1.Unix is a Command Line User Interface and Windows is Graphic User Interface operating system.

2. Unix is command based and Windows is menu based operating system.

3. Windows is event driven whereas this feature is absent in Unix operating system.

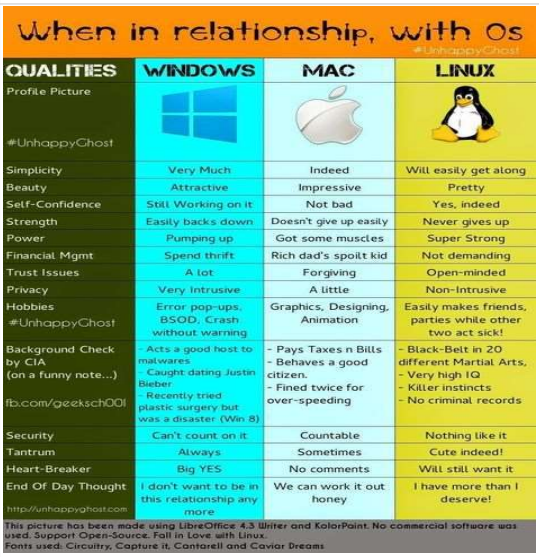
4. File system in Unix is (STD.ERR,STD.IO), and in Windows it is (FAT32,NTFS).

5. In Unix multiprocessing is possible whereas it is not possible in Windows.

6. In terms of security, Unix is more secure than Windows as we can restrict the permission of each user.

7. Windows operating system support plug and play and this feature is not available in Unix.

8. Windows is licensed operating system and Unix is free source operating system.



30. What is Linux?

Linux is also distributed under an open source license. Open source follows the following key philosophies:

The freedom to run the program, for any purpose.

The freedom to study how the program works, and change it to make it do what you wish.

The freedom to redistribute copies so you can help your neighbor.

The freedom to distribute copies of your modified versions to others.

Just like Windows XP, Windows 7, Windows 8, and Mac OS X, Linux is an operating system. An operating system is software that manages all of the hardware resources associated with your desktop or laptop. To put it simply – the operating system manages the communication between your software and your hardware. Without the operating system (often referred to as the "OS"), the software wouldn't function.

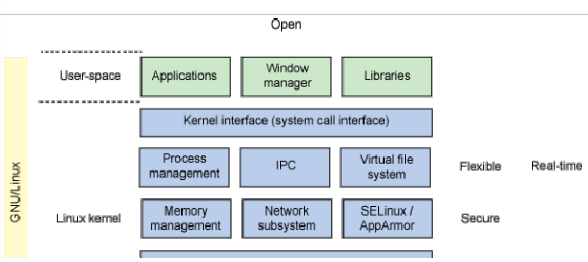
The OS is comprised of a number of pieces:

The Bootloader: The software that manages the boot process of your computer. For most users, this will simply be a splash screen that pops up and eventually goes away to boot into the operating system.

The kernel: This is the one piece of the whole that is actually called "Linux". The kernel is the core of the system and manages the CPU, memory, and peripheral devices. The kernel is the "lowest" level of the OS.

Daemons: These are background services (printing, sound, scheduling, etc) that either start up during boot, or after you log into the desktop.

The Shell: You've probably heard mention of the Linux command line. This is the shell – a command process that allows you to control the computer via commands typed into a text interface. This is what, at one time, scared people away from Linux the most

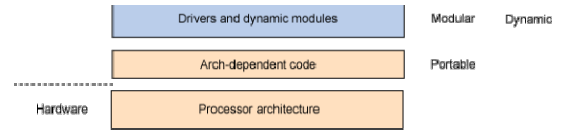


(assuming they had to learn a seemingly archaic command line structure to make Linux work). This is no longer the case. With modern desktop Linux, there is no need to ever touch the command line.

Graphical Server: This is the sub-system that displays the graphics on your monitor. It is commonly referred to as the X server or just "X".

Desktop Environment: This is the piece of the puzzle that the users actually interact with. There are many desktop environments to choose from (Unity, GNOME, Cinnamon, Enlightenment, KDE, XFCE, etc). Each desktop environment includes built-in applications (such as file managers, configuration tools, web browsers, games, etc).

Applications: Desktop environments do not offer the full array of apps. Just like Windows and Mac, Linux offers thousands upon thousands of high-quality software titles that can be easily found and installed. Most modern Linux distributions (more on this in a moment) include App Store-like tools that centralize and simplify application installation. For example: Ubuntu Linux has the Ubuntu Software Center (Figure 1) which allows you to quickly search among the thousands of apps and install them from one centralized location.



31.

Definition - What does Breakpoint mean?
A breakpoint, in the context of C#, is an intentional stop marked in the code of an application where execution pauses for debugging. This allows the programmer to inspect the internal state of the application at that point.

A breakpoint helps to speed up the debugging process in a large program by allowing the execution to continue up to a desired point before debugging begins. This is more efficient than stepping through the code on a line-by-line basis.

Conditions associated with a breakpoint represent an expression that determines whether the breakpoint is to be hit or skipped. When filters that specify process or thread are attached to the breakpoint, it is easier to debug parallel applications spread across multiple processors.

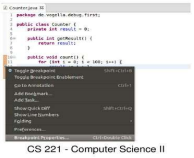
Techopedia explains Breakpoint
Whenever a breakpoint is hit, the application and the debugger are said to be in "break" mode, during which the following actions can be executed:

Inspect the values of local variables set in the current block of code in a separate local window.
Terminate the execution of a single or multiple application.
Step through the code line by line. If there is no source code underlying the execution statements, it leads to debugging in the disassembly window.
Make adjustments to the program result by viewing and modifying the values of variables.
Move the execution point so as to resume the application execution from that point.
Change the code using the "Edit and Continue" feature, and resume the execution with applied changes without having to stop and restart the debugging session.
The key features of breakpoints include:

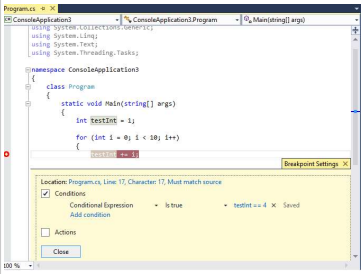
A breakpoint can be set and used while building an application using debug information.
A breakpoint can be set on the line of source code or on a function, with the ability to enable/ disable, edit and delete it.
A breakpoint can also be set at a memory address in the disassembly window and on a function using the call stack window.
Multiple breakpoints can be set on a line containing multiple executable statements.
A breakpoint can be set for all functions with the same name (both overloaded methods and functions occurring in multiple projects) in a single step.
Breakpoints are displayed in the source code and disassembly window using red symbols called glyphs in the left margin. The breakpoint tip displayed while resting the mouse on a glyph indicates information such as its associated condition, hit count (used for tracking the number of times a breakpoint is hit), filter, error condition, etc.

Conditional Breakpoints

- For each breakpoint, you can also create a conditional expression. The execution of the program only stops at the breakpoint, if the condition evaluates to true.
- Can be used for additional logging
  - the code that specifies the condition is executed every time the program execution reaches that point.



CS 221 - Computer Science II 20



Breakpoints are one of the most important debugging techniques in your developer's toolbox. You set breakpoints wherever you want to pause debugger execution. For example, you may want to see the state of code variables or look at the call stack at a certain breakpoint. If this is the first time that you've tried to debug code, you may want to read Debugging for absolute beginners before going through this article.

Set breakpoints in source code
You can set a breakpoint on any line of executable code. For example, in the following C# code, you could set a breakpoint on the variable declaration, the for loop, or any code inside the for loop. You can't set a breakpoint on the namespace or class declarations, or on the method signature.

To set a breakpoint in source code, click in the far left margin next to a line of code. You can also select the line and press F9, select Debug > Toggle Breakpoint, or right-click and select Breakpoint > Insert breakpoint. The breakpoint appears as a red dot in the left margin.

In C# code, breakpoint and current execution lines are automatically highlighted. For C++ code, you can turn on highlighting of breakpoint and current lines by selecting Tools (or Debug) > Options > Debugging > Highlight entire source line for breakpoints and current statement (C++ only).
When you debug, execution pauses at the breakpoint, before the code on that line is executed. The breakpoint symbol shows a yellow arrow.
When the debugger stops at the breakpoint, you can look at the current state of the app, including variable values and the call stack. For more information about the call stack, see How to: Use the Call Stack window.

The breakpoint is a toggle. You can click it, press F9, or use Debug > Toggle Breakpoint to delete or reinsert it.

To disable a breakpoint without deleting it, hover over or right-click it, and select Disable breakpoint. Disabled breakpoints appear as empty dots in the left margin or the Breakpoints window. To re-enable a breakpoint, hover over or right-click it, and select Enable breakpoint.

Set conditions and actions, add and edit labels, or export a breakpoint by right-clicking it and selecting the appropriate command, or hovering over it and selecting the Settings icon.

32.

33.

34.

35.

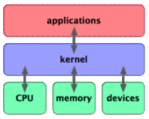
36.

37.

38.

39.

Misc



Difference between Object, Var and Dynamic type

Object	Var	Dynamic
Object was introduced with C# 1.0	Var was introduced with C# 3.0	Dynamic was introduced with C# 4.0
It can store any kind of value, because object is the base class of all type in .NET framework.	It can store any type of value but It is mandatory to initialize var types at the time of declaration.	It can store any type of the variable, similar to old VB language variable.
Compiler has little information about the type.	It is type safe i.e. Compiler has all information about the stored value, so that it doesn't cause any issue at run-time.	It is not type safe i.e. Compiler doesn't have any information about the type of variable.
Object type can be passed as method argument and method also can return object type.	Var type cannot be passed as method argument and method cannot return object type. Var type work in the scope where it defined.	Dynamic type can be passed as method argument and method also can return dynamic type.
Need to cast object variable to original type to use it and performing desired operations.	No need to cast because compiler has all information to perform operations.	Casting is not required but you need to know the properties and methods related to stored type.
Cause the problem at run time if the stored value is not getting converted to underlying data type.	Doesn't cause problem because compiler has all information about stored value.	Cause problem if the wrong properties or methods are accessed because all the information about stored value is get resolve only at run time.
Useful when we don't have more information about the data type.	Useful when we don't know actual type i.e. type is anonymous.	Useful when we need to code using reflection or dynamic languages or with the COM objects, because you need to write less code.

Kernel-level Threads

- Executed in kernel mode, managed by OS
- Kthreadd children

Pro:
- Resource Aware
- No need to use a new address space
- Thread blocked
  ↓
  Scheduled

Con:
- Slower then User-threads