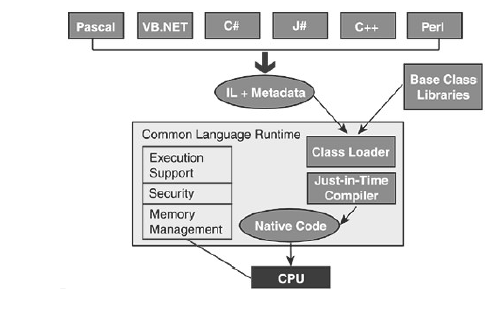
# Language execution basics

.NET is a software framework.  
.NET framework provides CLR (Common Language Runtime) and FCL (Framework Class Library).  
CLR: Virtual Machine. Manages memory, OS and hardware independence and language independence.   
FCL includes BCL (Base Class Library).  
BCL will be replaced by CoreFx.



C# code 🡺 IL (Intermediate Language) + Metadata & BCL 🡺 CLR (convert to native code) 🡺 CPU

All .NET-aware compilers generate IL instructions and metadata.  
The Just-in-Time (JIT) compiler converts the IL to a native code when application runs.

.NET Core is a framework which was built to use only one library to rule them all, where you don’t have to worry about platform incompatibilities.

C# is strongly typed (opposite of weakly typed: determine type at runtime).  
Every variable should be declared with a type.  
Assigning one type to another requires conversion.  
Primitive types: int, long, …  
Creating a class means defining a new type.

## Var keyword

Var keyword is only usable for local variables.

## Dynamic keyword

A dynamic variable can have any type and can change during runtime.  
Loosing compile-time checking.  
Performance suffering.

🡪 Try to avoid it

## Reference types

Classes are reference types.  
Variables point to objects allocated on heap; they don’t hold the objects in the stack.

## Value types

Variable holds the value.  
Fast to instantiate.  
Values types are immutable: int y = 17; int x = y; 🡪 x gets a copy of the value of y assigned.

## Stack and Heap

Heap:

* Dynamically allocate objects
* Reference types and Value types
* Garbage collection
* Serveral zones

Stack:

* Controls method execution
* Parameter binding:  
  Reference types: copy ref to stack  
  Value types: copy value of stack

## Struct

Struct definitions create value types.  
Should represent a single small value.  
Are passed by val, so copied onto the stack.  
Structs are immutable.

Normally always a class, unless performance considerations.

## Enum

Creates a value type.  
Is a set of named constants.  
Underlying data type is int by default.

You could change data type but it should be enumerable 🡪 int, long, .. but no string.

# Events and delegates

## Events

Events are notifications.  
Central role in .NET framework.  
Provide a ways to trigger notifications from end users or from objects.

Pattern known as PubSub.  
Publisher raises the event; one or more subscribers process the event.

Allows a class to send notifications to other classes or objects.

Important pattern 🡪 C# adds specialized language constructs and execution mechanisms.

This pattern promotes SoC and isolation:  
- Button doesn’t need to know who receives its events.  
- There can be multiple listeners who don’t know each other and can do different things.

Events are implemented in C# by a special construct called a delegate.

public event WorkPerformedHandler WorkPerformed;

Events are raised by calling the vent like a method.

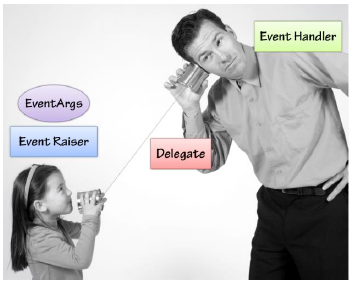
WorkPerformed(8, WorkType.Lector);

Or by accessing the event’s delegate and invoking it directly.

WorkPerformedHandler del = WorkPerformed as WorkPerformedHandler;

WorkPerformed(8, WorkType.Lector);

## Delegate



A delegate class is a specialized class often called a “Function Pointer”.

A delegate is a pipeline.

A delegate is a type that references methods.

public delegate void WorkPerformedHandler(int hours, WorkType workType);

static void WorkPerformed1(int hours, WorkType workType){

...

}

Invoking a Delegate instance

var del1 = new WorkPerformedHandler(WorkPerformed1);

del1(5, WorkType.Golf);

Invocation List

var del2 = new WorkPerformedHandler(WorkPerformed2);

del1+=del2;

A delegate returns a type.   
The last type is return from the Invocation List.

## Event handler

Event handler is responsible for receiving and processing data from a delegate.

Normally receives two parameters: Sender and EventArgs.  
EventArgs responsible for encapsulating event data.

# Lambdas

## Anonym event lambda

SubmitButton.Click += (s, e) => {

var b = s as Button;

MessageBox.Show($"You clicked on {b.Name}");

}

## Assigning a Lambda to a Delegate

public delegate int AddDelegate(int a, int b);

AddDelegate ad = (a, b) => a + b;

int result = ad(1, 1); // Result = 2

## Emty parameter

delegate bool LogDelegate();

LogDelegate ld = () => {

UpdateDb();

WriteToLog();

return true;

};

bool status = ld();

## Generics

Action<T>: Accepts a single parameter and returns no value.  
Func<T,TResult>: Accepts a single parameter and return a value of type TResult.  
You can have multiple parameters T (and multiple different types T)  
Action<int, string>: Accepts two parameter, the first of type int and the second of type string; returns no value.  
Func<int,string,int>: Accepts two parameter, the first of type int and the second of type string; returns a int.

You can assign a lambda expression to Actions, Funcs and Delegates.

ADelegate addDel = (x, y) => x + y; // using custom delegates

Action<int, int> addAction = (x, y) => Console.WriteLine(x + y); // Using Actions

Func<int, int, int> addFunc = (x, y) => x + y; // Using Func

## LINQ

Uses extension methods that takes delegates as argument to filter data.   
Delegates are expressed most elegantly with lambda’s.

## Extension methods

Add some members to a type which you don’t want to change.

public static class Extensions{

public static Grades minPassing = Grades.D;

public static bool Passing(this Grades grade){

return grade >= minPassing;

}

}

By using the *this* keyword, you extend the method towards the object instance. It make appear like a normal method.

g1.Passing();

## Events and Delegates in action

When you want to update GUI elements, you need to use another thread.

public delegate void StartProgressDelegate(int val);

public delegate void ShowProgressDelegate(int val);

public MainWindow()

{

InitializeComponent();

}

private void StartButton\_Click(object sender, RoutedEventArgs e)

{

StartProgressDelegate startDel = new StartProgressDelegate(StartProcess);

startDel.BeginInvoke(100, null, null);

MessageBox.Show("Called after async process started.");

}

private void StartProcess(int max)

{

ShowProgres(0);

for (int i = 0; i <= max; i++)

{

Thread.Sleep(100);

ShowProgres(i);

}

MessageBox.Show("It's over !");

}

private void ShowProgres(int val)

{

if (!Dispatcher.CheckAccess())

{

ShowProgressDelegate showProgDel = new ShowProgressDelegate(ShowProgres);

Dispatcher.BeginInvoke(showProgDel, val);

}

else

{

lblOutput.Text = Convert.ToString(val);

pbStatus.Value = val;

}

}

## Background worker

To execute a time-consuming opration in the background, create a BackgroundWorker and listen for events that report the progress of your operation and signal when your operation is finished.

# Parallel and async programming

A thread creates a separate virtual CPU inside your program.  
A thread is a resource.

Multithreading is difficult:

* Synchronization between threads
  + Deadlock: threads wait indefinitely for each other
  + Race conditions: when sharing a variable, threads could possibly introduce errors
* Thread pooling (resource sharing)

Parrallel vs Async  
Async programming 🡪 Responsiveness: hire latency of potentially long-running or blocking operation by executing in background.  
Parallel programming 🡪 Performance: reduce time of CPU-bound computations by dividing workload & executing simultaneously.

## Task

Is a unit of work and an object that denotes an ongoing operation.

Two types of tasks:

* Code tasks: parallel operations
* Façade tasks: asynchronous operations

Creating a task:

Task t = new Task((){

…

});

T.start();

When the task started, the program continue to run, it uses a queue.  
There is at that moment two thread: the Main Thread and the Thread of the Task.

If you want to do some stuff after the task is over (so the code will only be run after that the task is complete):

T.ContinueWith(code);

You can also create and immediately run it with the Task.Factory:

Task T = Task.Factory.StartNew(code);

This is more efficient.

Nowadays, more common to use Task.Run:

Task.Run(code);

TPL: Task Parallel Library  
TPL provides several methods to synchronize tasks:

* Wait for a task to finish: wait() method
* Wait for one or more tasks to finish: retrieve the specific task in the array or WaitAll(tasks) method
* Retrieve a result from a task: get value via .Result property (calls Wait() implicitly)
* Compose tasks

# Garbage collection

GC means you don’t need to manually free memory.  
GC has performance trade-offs.

The CLR GC is an **almost-concurrent**, **parallel**, **compacting**, **mark-and-sweep**, **generational**, **tracing** GC.

Objects that are completely unreachable can be reclaimed and its memory can be use for future allocations.

Mark and sweep are the two fundamental actions performed by the GC.

## Mark

Identify all live objects.

In the mark phase, the GC identifies all live objects in the heap; it is done recursively. The GC starts from a group of objects that are certainly live and follows references between the objects until all the referenced objects are explored. The GC keeps track of objects it has already visited to avoid getting into an infinite loop if there is a reference cycle between objects.

## Sweep

Reclaim dead objects.

In the sweep phase, the GC reclaims all unused memory that’s occupied by dead objects.

## Compact

Shift live objects together.  
Objects that can still be used must be kept alive.

The sweep phase often ends with a compaction phase. In this phase, the GC shifts live objects together so that they are consecutive in memory. The free space is then also consecutive and this can make allocations cheaper.

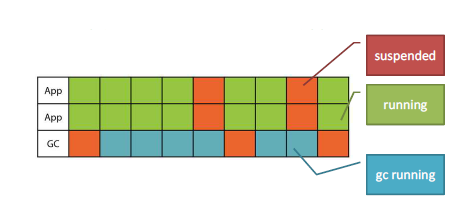
## Roots

References to objects who aren’t referenced by other objects are called roots.  
There are multiple kinds of roots in a .NET program.

## Workstation GC

There are multiple GC flavors.  
Workstation GC is “kind of” suitable for client apps.  
Default for almost all .NET applications.  
GC runs on a single thread 🡪 doesn’t use all CPU cores

### Concurrent Workstation GC

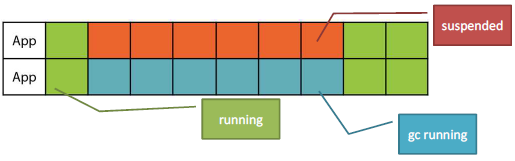


Default

In concurrent Workstation GC, the CLR creates a special GC thread as soon as your application start running. This thread monitors the GC heap and performs garbage collections occasionally. **It doesn’t wait until all memory’s exhausted**. It tries to schedule GCs in a way that takes lots of memory and reclaims it without badly affecting application performance.

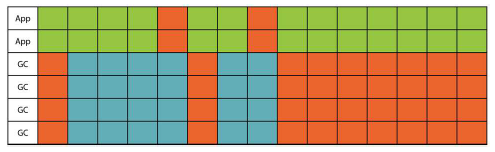
The key here is that the GC thread runs concurrently (at the same time) with the application threads most of the time. Most parts of the GC process can be done in the background while the other threads are running. Occasionally, the GC thread does have to suspend all the applications threads, but these suspensions are usually very short.

### Non-concurrent Workstation GC



In non-concurrent workstation GC, there is no special GC thread. One of the application threads is picked out to do the GC when it performs an allocation that would trigger a GC. That application thread doesn’t do anything else during GC but neither do the other threads. Non-concurrent GC has to suspend all the application threads for the duration of the GC.

### Server GC

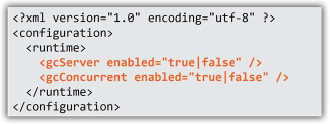


The main difference here is that there are multiple GC threads. The CLR creates a GC thread for each logical processor. When a GC is needed, these threads are all awakened at once and start churning away. Server GC also has a separate heap area for each logical processor.

Until CLR 4.5 was server GC non-concurrent but since then it became concurrent so it is now a reasonable default for many high-memory apps.

### Switching GC flavors

In app.config:



Ignored if invalid

You can’t switch flavors at runtime.

In .NET Core: edit .csporj file to include following properties:

<PropertyGroup>

<ServerGarbageCollection>true</ServerGarbageCollection>

<ConcurrentGarbageCollection>true</ConcurrentGarbageCollection>

</PropertyGroup>

## Generational GC

A full GC is expensive and inefficient.

Divide the heap into regions and perform small collection often.

**New objects die fast, old objects stay alive.**



Gen 0 is where objects are born.  
Gen 1 is the nursery.  
Gen 2 is for tenured old objects.

Garbage Collector will try to perform lots of gen 0 collection and touch gen 2 only infrequently.

Generations sizes depends on numerous runtime parameters: 32 or 64bit systems, CPU cache size, even amount of physical memory.

When a GC is complete in gen 0, the surviving objects are promoted to gen 1. The objects of gen 1 are still considered young so the GC still assumes it’s useful to run frequent collections in gen 1. Objects surviving a gen 1 collection are promoted to gen 2. The GC tries its best to never touch objects in gen 2 but once in a while, there might be a gen 2 collection. Gen 2 collections can take very long because it doesn’t have a size limit. So, ideally, there should be very few gen 2 collections.

## The Large Object Heap (LOH)

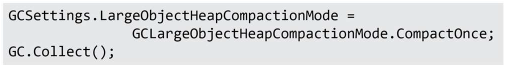
Large objects are stored in a separate heap region, which is called the LOH.  
Opposing to the SOH (Small Objects Heap).  
Objects are considered large when they are larger than 85 000 bytes. + Special exception for arrays of doubles: an array of more than 1000 doubles is also considered a large object.

The GC doesn’t compact the LOH, this may cause fragmentation. The idea is that compaction requires copying objects around, and copying large objects is pretty expensive.

LOH is considered part of gen 2.

### Explicit LOH Compaction

.NET 4.5.1 introduces LOH compaction



Try to avoid it !

## Foreground and background GC

Since CLR 4.0, if the application tries to allocate during GC, the application thread launches a foreground GC. (In CLR 2.0, application thread was waiting for full GC to complete).

## Resource cleanup

GC only takes care of memory, not all reclaimable ressources (sockets, file handles, db transaction, etc).  
For example, when a database transaction dies, it has to tell the database to either commit or abort the transaction. And it also possibly has to close an open network connection to the database server. Both of these cleanup operations must be explicitly performed. They're not a part of normal memory management, which is what the GC is good at.

## Finalization

Garbage Collector can invoke the finalizer method that has been written by the programmer. This method is the class’s opportunity to release any non-memory resources.

When create an object that has a finalizer, we put it in the finalization queue. The finalization queue should be a GC root so we don’t accidentally reclaim memory for an object that hasn’t been finalized yet.  
Then we need a way to know that an object is eligible for finalization. We start making the heap from all the roots except the finalization queue, and then we will look at the finalization queue. Any objects that weren’t marked yet are now eligible for finalization because they’re only referenced by the finalization queue.  
Now suppose we have an object that has to be finalized. We are now in the middle of a GC. Let’s postpone finalization and schedule it on a different thread, the finalizer thread. To put work on the finalizer thread, we need another queue of objects that are ready for finalization; CLR calls it the F-reachable queue. Objects on this queue are live only because they are waiting for finalization. Finally, the finalizer thread picks objects from the F-reachable queue, runs their finalizers, and objects with finalizers that have run can be removed from the F-reachable queue and can be collected by the Garbage Collector. So the next GC cycle is going to pick them up.

### Performance problems

The first problem with finalization is that it extends object lifetime. An object will survive at least one GC if it has a finalizer. Objects that used to die in gen 0, might die in gen 1. Objects that used to slip into gen 1 will now slip into gen 2. Now take many of these objects, and you’ve got a performance problem because there will be many gen 2 GCs.

Second problem with finalization is that it is object to an unbounded queue problem. The F-reachable queue can fill at a faster rate than the finalizer thread can drain it. So there can be a memory leak.

Third problem is that finalization happens on a separate thread, which can introduce nontrivial race conditions. The crux of the matter is that it’s possible for a finalizer of an object to run while the method of that very object is still executing.

## Dispose pattern

In the Dispose pattern, you call the cleanup method Dispose, and you make the class implement the IDisposable interface.

Sometimes, it might also make sense to combine Dispose pattern with a finalizer. The finalizer serves as a fallback in case the user forgot to call Dispose.

## Resurrection and Object Pooling

You can bring object back to life from inside a finalizer. You can create a new root reference to the current object. The GC won’t be able to reclaim it in that case. This technique is called resurrection, and is often used to create object pools.

Object pool: Instead of recreating objects every time you need them, your objects sit in a cache and you can take them from the cache when you need them and return them back to the cache, or to the pool, when you’re done with them. This can save the initialization costs for some expensive objects.

Database frameworks usually use this approach for pulling database connections instead of creating a new connection for every operation. A way to return the object to the pool is by using a finalizer that will do this automatically. The finalizer can put the object back in the pool even if the user forgot to.

GC.ReRegisterForFinalize method: Puts the object back in the finalization queue.

# Logging

Niet gezien.