Your performance goal

Your performance goal

- Ok, so your are attending this course - **that's great**!😍

Your performance goal

- Ok, so your are attending this course - **that's great**!😍
- you will need to know **how** and **where** apply this knowledge

Your performance goal

- Ok, so your are attending this course - **that's great**!😍
- you will need to know **how** and **where** apply this knowledge
- and this is the goal of this soft-skill-ish module 😇

Your performance goal

There are three main perspectives:

Your performance goal

There are three main perspectives:

- Application developer perspective

Your performance goal

There are three main perspectives:

- Application developer perspective
- Library author perspective

Your performance goal

There are three main perspectives:

- Application developer perspective
- Library author perspective
- Your company perspective

Your performance goal

Application developer perspective:

Your performance goal

Application developer perspective:

- *"you don't have to care about until it becomes a problem"* 👍

Your performance goal

Application developer perspective:

- *"you don't have to care about until it becomes a problem"* 👍
- you know your app, its context, requirements, measure it!

Your performance goal

Application developer perspective:

- *"you don't have to care about until it becomes a problem"* 👍
- you know your app, its context, requirements, measure it!
- the goal: not to write everything with pointers and **Span** - BUT...

Your performance goal

Application developer perspective:

- *"you don't have to care about until it becomes a problem"* 👍
- you know your app, its context, requirements, measure it!
- the goal: not to write everything with pointers and `Span` - BUT...
- keep in mind performance, good and bad practices, be aware of abstraction leaks. And have "entry points" how to fix, trace, debug, **measure**...

Your performance goal

Application developer perspective:

- *"you don't have to care about until it becomes a problem"* 👍
- you know your app, its context, requirements, measure it!
- the goal: not to write everything with pointers and `Span` - BUT...
- keep in mind performance, good and bad practices, be aware of abstraction leaks. And have "entry points" how to fix, trace, debug, **measure**...
- be an expert!

Your performance goal

Library author perspective:

Your performance goal

Library author perspective:

- you don't know the context, it just should be "fast" :)

Your performance goal

Library author perspective:

- you don't know the context, it just should be "fast" :)
- the goal: to write everything with pointers and **Span** ;) (yes, joke!)

Your performance goal

Library author perspective:

- you don't know the context, it just should be "fast" :)
- the goal: to write everything with pointers and `Span` ;) (yes, joke!)
- but again, you need to **measure**...

Your performance goal

Company perspective:

- *"Over a 4 week period, we analysed mobile site data from 37 retail, travel, luxury and lead generation brands across Europe and the US. Results showed that a mere **0.1s change** in load time can influence every step of the user journey, ultimately increasing conversion rates. Conversions grew **by 8% for retail sites** and **by 10% for Travel sites** on average."* - Milliseconds Make Millions report from Deloitte

Your performance goal

Company perspective:

- *"Over a 4 week period, we analysed mobile site data from 37 retail, travel, luxury and lead generation brands across Europe and the US. Results showed that a mere **0.1s change** in load time can influence every step of the user journey, ultimately increasing conversion rates. Conversions grew **by 8% for retail sites** and **by 10% for Travel sites** on average."* - Milliseconds Make Millions report from Deloitte
- *"Infrastructure will take it"*:

Your performance goal

Company perspective:

- *"Over a 4 week period, we analysed mobile site data from 37 retail, travel, luxury and lead generation brands across Europe and the US. Results showed that a mere **0.1s change** in load time can influence every step of the user journey, ultimately increasing conversion rates. Conversions grew **by 8% for retail sites** and **by 10% for Travel sites** on average."* - Milliseconds Make Millions report from Deloitte
- *"Infrastructure will take it"*:
  - agree, sometimes it is just cheaper to put more $$$ on Azure instead of paying a devoloper

Your performance goal

Company perspective:

- *"Over a 4 week period, we analysed mobile site data from 37 retail, travel, luxury and lead generation brands across Europe and the US. Results showed that a mere **0.1s change** in load time can influence every step of the user journey, ultimately increasing conversion rates. Conversions grew **by 8% for retail sites** and **by 10% for Travel sites** on average."* - Milliseconds Make Millions report from Deloitte
- *"Infrastructure will take it"*:
  - agree, sometimes it is just cheaper to put more $$$ on Azure instead of paying a devoloper
  - especially, if you are a startup

Your performance goal

Company perspective:

- *"Over a 4 week period, we analysed mobile site data from 37 retail, travel, luxury and lead generation brands across Europe and the US. Results showed that a mere **0.1s change** in load time can influence every step of the user journey, ultimately increasing conversion rates. Conversions grew **by 8% for retail sites** and **by 10% for Travel sites** on average."* - Milliseconds Make Millions report from Deloitte
- *"Infrastructure will take it"*:
  - agree, sometimes it is just cheaper to put more $$$ on Azure instead of paying a devoloper
  - especially, if you are a startup
  - but... performance is a feature

Your performance goal

Company perspective:

- *"Over a 4 week period, we analysed mobile site data from 37 retail, travel, luxury and lead generation brands across Europe and the US. Results showed that a mere **0.1s change** in load time can influence every step of the user journey, ultimately increasing conversion rates. Conversions grew **by 8% for retail sites** and **by 10% for Travel sites** on average."* - Milliseconds Make Millions report from Deloitte
- *"Infrastructure will take it"*:
  - agree, sometimes it is just cheaper to put more $$$ on Azure instead of paying a devoloper
  - especially, if you are a startup
  - but... performance is a feature
  - you don't know if you are overpaying if you don't know how to **measure**

"Don't guess, measure"

- measuring only for measuring is as bad as optimizing only for optimizing, you need both!

"Don't guess, measure"

- measuring only for measuring is as bad as optimizing only for optimizing, you need both!
- optimize only what's significant

"Don't guess, measure"

- measuring only for measuring is as bad as optimizing only for optimizing, you need both!
- optimize only what's significant
- measure to know what's significant

"Don't guess, measure"

- measuring only for measuring is as bad as optimizing only for optimizing, you need both!
- optimize only what's significant
- measure to know what's significant
- and measure to track progress/regression

"Don't guess, measure"

- measuring only for measuring is as bad as optimizing only for optimizing, you need both!
- optimize only what's significant
- measure to know what's significant
- and measure to track progress/regression
- measure what matters, **NOT** what it happened you know something about

"Don't guess, measure"

- measuring only for measuring is as bad as optimizing only for optimizing, you need both!
- optimize only what's significant
- measure to know what's significant
- and measure to track progress/regression
- measure what matters, **NOT** what it happened you know something about

  (*"I've heard something about generations, I'll measure them!"* - I've seen it many times...)

"Don't guess, measure"

- measuring only for measuring is as bad as optimizing only for optimizing, you need both!
- optimize only what's significant
- measure to know what's significant
- and measure to track progress/regression
- measure what matters, **NOT** what it happened you know something about
  (*"I've heard something about generations, I'll measure them!"* - I've seen it many times...)
- *Measure-first* approach:

"Don't guess, measure"

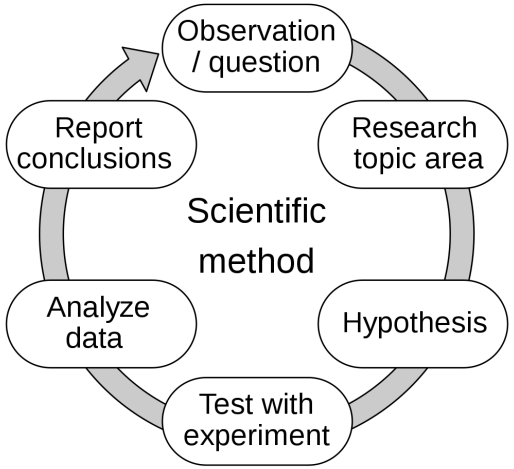
- measuring only for measuring is as bad as optimizing only for optimizing, you need both!
- optimize only what's significant
- measure to know what's significant
- and measure to track progress/regression
- measure what matters, **NOT** what it happened you know something about
  (*"I've heard something about generations, I'll measure them!"* - I've seen it many times...)
- *Measure-first* approach:
  - Measure to identify the area of possible problems - you need to understand the fundamentals to know where and how to look at

"Don't guess, measure"

- measuring only for measuring is as bad as optimizing only for optimizing, you need both!
- optimize only what's significant
- measure to know what's significant
- and measure to track progress/regression
- measure what matters, **NOT** what it happened you know something about
  (*"I've heard something about generations, I'll measure them!"* - I've seen it many times...)
- *Measure-first* approach:
  - Measure to identify the area of possible problems - you need to understand the fundamentals to know where and how to look at
  - Measure to understand effects of improvements - ...or regression

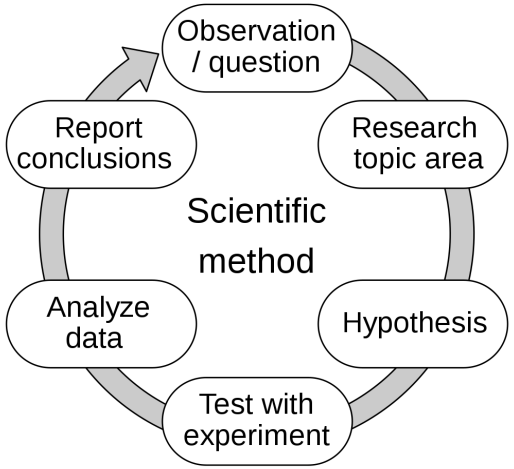
"Don't guess, measure"

Treat it as **the scientific method**:



"Don't guess, measure"

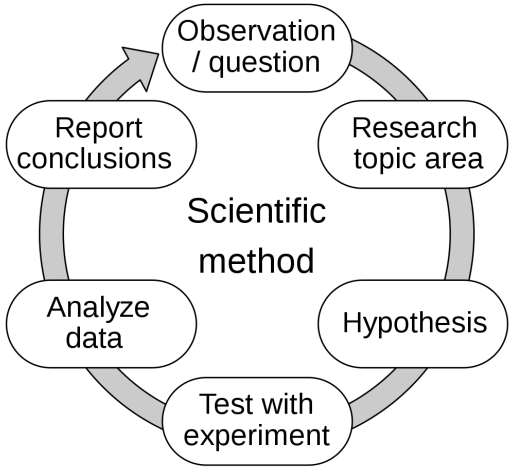
Treat it as **the scientific method**:



A performance analysis/optimization can be like detective work

"Don't guess, measure"

Treat it as **the scientific method**:

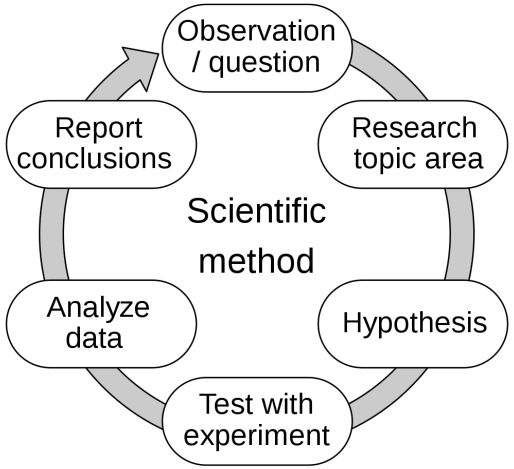


A performance analysis/optimization can be like detective work

- a lot of elements may impact your performance - hardware, operating system, runtime/JIT/GC, threading, library, your code...

"Don't guess, measure"

Treat it as **the scientific method**:

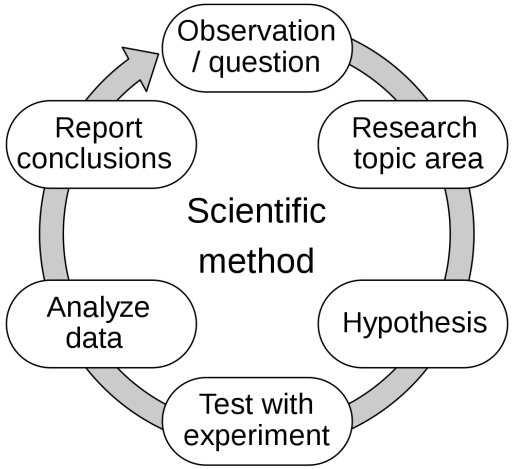


A performance analysis/optimization can be like detective work

- a lot of elements may impact your performance - hardware, operating system, runtime/JIT/GC, threading, library, your code...
- you need to **formulate hypothesis** - *"we can reduce cloud hosting costs by 10% by changing GC mode"*, *"we can improve tail latency by 25% by reducing allocations"*, ...

"Don't guess, measure"

Treat it as **the scientific method**:

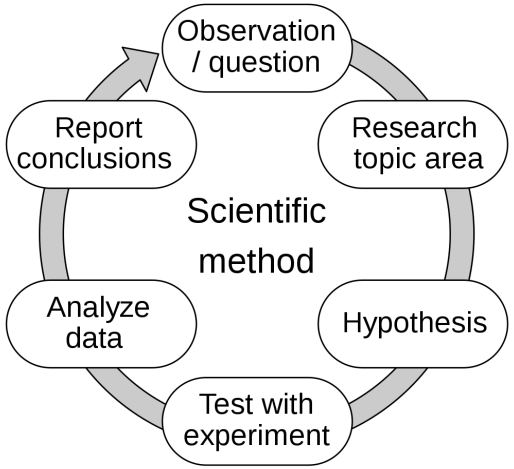


A performance analysis/optimization can be like detective work

- a lot of elements may impact your performance - hardware, operating system, runtime/JIT/GC, threading, library, your code...
- you need to **formulate hypothesis** - *"we can reduce cloud hosting costs by 10% by changing GC mode"*, *"we can improve tail latency by 25% by reducing allocations"*, ...
- test with experiment and analyze data - you need to measure

"Don't guess, measure"

Treat it as **the scientific method**:



A performance analysis/optimization can be like detective work

- a lot of elements may impact your performance - hardware, operating system, runtime/JIT/GC, threading, library, your code...
- you need to **formulate hypothesis** - *"we can reduce cloud hosting costs by 10% by changing GC mode"*, *"we can improve tail latency by 25% by reducing allocations"*, ...
- test with experiment and analyze data - you need to measure
- make conclusions, reevalute ideas, reformulate hypothesis, ...

Your performance goal

Some popular goals and metrics:

Your performance goal

Some popular goals and metrics:

- **throughput** - to make your application more efficient
  - GC aspects: *% Pause time in GC*, CPU usage, ...

Your performance goal

Some popular goals and metrics:

- **throughput** - to make your application more efficient
  - GC aspects: *% Pause time in GC*, CPU usage, ...
- **tail latency** - improve
  - GC aspects: individual, long *GC pauses*

Your performance goal

Some popular goals and metrics:

- **throughput** - to make your application more efficient
  - GC aspects: *% Pause time in GC*, CPU usage, ...
- **tail latency** - improve
  - GC aspects: individual, long *GC pauses*
- **memory footprint** - it may significantly influence the cost, e.g. *Azure Functions*-like environment
  - GC aspects: *GC heap size* & *process memory usage*

Your performance goal

Some popular goals and metrics:

- **throughput** - to make your application more efficient
  - GC aspects: *% Pause time in GC*, CPU usage, ...
- **tail latency** - improve
  - GC aspects: individual, long *GC pauses*
- **memory footprint** - it may significantly influence the cost, e.g. *Azure Functions*-like environment
  - GC aspects: *GC heap size* & *process memory usage*
- **CPU usage** - which may directly translate to the cost
  - GC aspects: *% Time in GC*, *% CPU time in GC*, ...

Your performance goal

Some popular goals and metrics:

- **throughput** - to make your application more efficient
  - GC aspects: *% Pause time in GC*, CPU usage, ...
- **tail latency** - improve
  - GC aspects: individual, long *GC pauses*
- **memory footprint** - it may significantly influence the cost, e.g. *Azure Functions*-like environment
  - GC aspects: *GC heap size* & *process memory usage*
- **CPU usage** - which may directly translate to the cost
  - GC aspects: *% Time in GC*, *% CPU time in GC*, ...
- **application performance** - page load times, responsivenes, ...
  - GC aspects: it depends, all GC-related 😇

Pragmatic approach to .NET memory management

Pragmatic approach to .NET memory management

1. Level 1: use **`struct`** here, don't use LINQ there, set **`Capacity`** of the **`List`**
    - clean code for performance
    - some knowledge of abstractions, some basic sanity checks

Pragmatic approach to .NET memory management

1. Level 1: use **`struct`** here, don't use LINQ there, set **`Capacity`** of the **`List`**
    - clean code for performance
    - some knowledge of abstractions, some basic sanity checks
2. Level 2: know how to measure and diagnose to check if something bad happens
    - good toolbox
    - some knowledge of abstractions

Pragmatic approach to .NET memory management

1. Level 1: use **`struct`** here, don't use LINQ there, set **`Capacity`** of the **`List`**
    - clean code for performance
    - some knowledge of abstractions, some basic sanity checks
2. Level 2: know how to measure and diagnose to check if something bad happens
    - good toolbox
    - some knowledge of abstractions
3. Level 3: optimize and be really memory-aware
    - advanced C# tips
    - some internals knowledge

Pragmatic approach to .NET memory management

1. Level 1: use **`struct`** here, don't use LINQ there, set **`Capacity`** of the **`List`**
    - clean code for performance
    - some knowledge of abstractions, some basic sanity checks
2. Level 2: know how to measure and diagnose to check if something bad happens
    - good toolbox
    - some knowledge of abstractions
3. Level 3: optimize and be really memory-aware
    - advanced C# tips
    - some internals knowledge
4. I'm personally not paralized by the high-perf code too, I start from working prototype

Pragmatic approach to .NET memory management

1. Level 1: use **`struct`** here, don't use LINQ there, set **`Capacity`** of the **`List`**
    - clean code for performance
    - some knowledge of abstractions, some basic sanity checks
2. Level 2: know how to measure and diagnose to check if something bad happens
    - good toolbox
    - some knowledge of abstractions
3. Level 3: optimize and be really memory-aware
    - advanced C# tips
    - some internals knowledge
4. I'm personally not paralized by the high-perf code too, I start from working prototype
5. Then I measure and optimize, if needed

Pragmatic approach to .NET memory management

1. Level 1: use **`struct`** here, don't use LINQ there, set **`Capacity`** of the **`List`**
    - clean code for performance
    - some knowledge of abstractions, some basic sanity checks
2. Level 2: know how to measure and diagnose to check if something bad happens
    - good toolbox
    - some knowledge of abstractions
3. Level 3: optimize and be really memory-aware
    - advanced C# tips
    - some internals knowledge
4. I'm personally not paralized by the high-perf code too, I start from working prototype
5. Then I measure and optimize, if needed
6. But having those sanity checks allows me to avoid stupid mistakes
    - reading the whole table to filter it in memory

Pragmatic approach to .NET memory management

1. Level 1: use **`struct`** here, don't use LINQ there, set **`Capacity`** of the **`List`**
    - clean code for performance
    - some knowledge of abstractions, some basic sanity checks
2. Level 2: know how to measure and diagnose to check if something bad happens
    - good toolbox
    - some knowledge of abstractions
3. Level 3: optimize and be really memory-aware
    - advanced C# tips
    - some internals knowledge
4. I'm personally not paralized by the high-perf code too, I start from working prototype
5. Then I measure and optimize, if needed
6. But having those sanity checks allows me to avoid stupid mistakes
    - reading the whole table to filter it in memory
7. we wrote clean code to maintain in better, let's wrote performant code to host it better - we really, really believe in Software Craftmanship (*"Not only working software, but also **well-crafted software**"*)

Pragmatic approach to .NET memory management

1. Level 1: use **`struct`** here, don't use LINQ there, set **`Capacity`** of the **`List`**
    - clean code for performance
    - some knowledge of abstractions, some basic sanity checks
2. Level 2: know how to measure and diagnose to check if something bad happens
    - good toolbox
    - some knowledge of abstractions
3. Level 3: optimize and be really memory-aware
    - advanced C# tips
    - some internals knowledge
4. I'm personally not paralized by the high-perf code too, I start from working prototype
5. Then I measure and optimize, if needed
6. But having those sanity checks allows me to avoid stupid mistakes
    - reading the whole table to filter it in memory
7. we wrote clean code to maintain in better, let's wrote performant code to host it better - we really, really believe in Software Craftmanship (*"Not only working software, but also **well-crafted software**"*)

    BTW, this applies to everything we use - architecture, **`async`**/**`await`**, ORM, .NET runtime itself, ...

"premature optimization is the root of all evil"

Structured Programming with go to Statements

DONALD E. KNUTH

Stanford University, Stanford, California 94305

A consideration of several different examples sheds new light on the problem of creating reliable, well-structured programs that behave efficiently. This study focuses largely on two issues: (a) improved syntax for iterations and error exits, making it

Computing Surveys, Vol 6, No 4, December 1974

premature optimization is the root of all evil.

The improvement in speed from Example 2 to Example 2a is only about 12%, and many people would pronounce that insignificant. The conventional wisdom shared by many of today's software engineers calls for ignoring efficiency in the small

premature optimization is the root of all evil.

The improvement in speed from Example 2 to Example 2a is only about 12%, and many people would pronounce that insignificant. The conventional wisdom shared by many of today's software engineers calls for ignoring efficiency in the small; but I believe this is simply an overreaction to the abuses they see being ***practiced by pennywise-and-pound-foolish programmers, who can't debug or maintain their "optimized" programs****.*

premature optimization is the root of all evil.

The improvement in speed from Example 2 to Example 2a is only about 12%, and many people would pronounce that insignificant. The conventional wisdom shared by many of today's software engineers calls for ignoring efficiency in the small; but I believe this is simply an overreaction to the abuses they see being ***practiced by pennywise-and-pound-foolish programmers, who can't debug or maintain their "optimized" programs****. In established engineering disciplines a 12% improvement, easily obtained, is never considered marginal.*

premature optimization is the root of all evil.

The improvement in speed from Example 2 to Example 2a is only about 12%, and many people would pronounce that insignificant. The conventional wisdom shared by many of today's software engineers calls for ignoring efficiency in the small; but I believe this is simply an overreaction to the abuses they see being ***practiced by pennywise-and-pound-foolish programmers, who can't debug or maintain their "optimized" programs****. In established engineering disciplines a 12% improvement, easily obtained, is never considered marginal. (...) Of course I wouldn't bother making such optimizations on a oneshot job, but when it's a question of preparing quality programs, I don't want to restrict myself to tools that deny me such efficiencies.*

premature optimization is the root of all evil.

The improvement in speed from Example 2 to Example 2a is only about 12%, and many people would pronounce that insignificant. The conventional wisdom shared by many of today's software engineers calls for ignoring efficiency in the small; but I believe this is simply an overreaction to the abuses they see being ***practiced by pennywise-and-pound-foolish programmers, who can't debug or maintain their "optimized" programs****. In established engineering disciplines a 12% improvement, easily obtained, is never considered marginal. (...) Of course I wouldn't bother making such optimizations on a oneshot job, but when it's a question of preparing quality programs, I don't want to restrict myself to tools that deny me such efficiencies. There is no doubt that the grail of efficiency leads to abuse. Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered.*

premature optimization is the root of all evil.

The improvement in speed from Example 2 to Example 2a is only about 12%, and many people would pronounce that insignificant. The conventional wisdom shared by many of today's software engineers calls for ignoring efficiency in the small; but I believe this is simply an overreaction to the abuses they see being ***practiced by pennywise-and-pound-foolish programmers, who can't debug or maintain their "optimized" programs****. In established engineering disciplines a 12% improvement, easily obtained, is never considered marginal. (...) Of course I wouldn't bother making such optimizations on a oneshot job, but when it's a question of preparing quality programs, I don't want to restrict myself to tools that deny me such efficiencies. There is no doubt that the grail of efficiency leads to abuse. Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered.* *We should forget about small efficiencies, say* ***about 97% of the time****: premature optimization is the root of all evil.*

The improvement in speed from Example 2 to Example 2a is only about 12%, and many people would pronounce that insignificant. The conventional wisdom shared by many of today's software engineers calls for ignoring efficiency in the small; but I believe this is simply an overreaction to the abuses they see being ***practiced by pennywise-and-pound-foolish programmers, who can't debug or maintain their "optimized" programs****. In established engineering disciplines a 12% improvement, easily obtained, is never considered marginal. (...) Of course I wouldn't bother making such optimizations on a oneshot job, but when it's a question of preparing quality programs, I don't want to restrict myself to tools that deny me such efficiencies. There is no doubt that the grail of efficiency leads to abuse. Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered.* *We should forget about small efficiencies, say* ***about 97% of the time****: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.*

The improvement in speed from Example 2 to Example 2a is only about 12%, and many people would pronounce that insignificant. The conventional wisdom shared by many of today's software engineers calls for ignoring efficiency in the small; but I believe this is simply an overreaction to the abuses they see being ***practiced by pennywise-and-pound-foolish programmers, who can't debug or maintain their "optimized" programs****. In established engineering disciplines a 12% improvement, easily obtained, is never considered marginal. (...) Of course I wouldn't bother making such optimizations on a oneshot job, but when it's a question of preparing quality programs, I don't want to restrict myself to tools that deny me such efficiencies. There is no doubt that the grail of efficiency leads to abuse. Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered.* *We should forget about small efficiencies, say* ***about 97% of the time****: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%. A* ***good programmer will not be lulled into complacency*** *by such reasoning, he will be wise to look carefully at the critical code;*

The improvement in speed from Example 2 to Example 2a is only about 12%, and many people would pronounce that insignificant. The conventional wisdom shared by many of today's software engineers calls for ignoring efficiency in the small; but I believe this is simply an overreaction to the abuses they see being ***practiced by pennywise-and-pound-foolish programmers, who can't debug or maintain their "optimized" programs****. In established engineering disciplines a 12% improvement, easily obtained, is never considered marginal. (...) Of course I wouldn't bother making such optimizations on a oneshot job, but when it's a question of preparing quality programs, I don't want to restrict myself to tools that deny me such efficiencies. There is no doubt that the grail of efficiency leads to abuse. Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered.* *We should forget about small efficiencies, say* ***about 97% of the time****: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%. A* ***good programmer will not be lulled into complacency*** *by such reasoning, he will be wise to look carefully at the critical code; but* ***only*** *after that code has been identified.*

So, don't be *"pennywise-and-pound-foolish programmer"*

So, don't be *"pennywise-and-pound-foolish programmer"*

- optimisation for millions of users while likely you will have 100 - well...

So, don't be *"pennywise-and-pound-foolish programmer"*

- optimisation for millions of users while likely you will have 100 - well...
- but, remember about that critical 3%

So, don't be *"pennywise-and-pound-foolish programmer"*

- optimisation for millions of users while likely you will have 100 - well...
- but, remember about that critical 3%
- a **good programmer will not be lulled into complacency** by such reasoning.

So, don't be *"pennywise-and-pound-foolish programmer"*

- optimisation for millions of users while likely you will have 100 - well...
- but, remember about that critical 3%
- a **good programmer will not be lulled into complacency** by such reasoning.
- don't call laziness a premature optimization - it just may be an experience :)

So, don't be *"pennywise-and-pound-foolish programmer"*

- optimisation for millions of users while likely you will have 100 - well...
- but, remember about that critical 3%
- a **good programmer will not be lulled into complacency** by such reasoning.
- don't call laziness a premature optimization - it just may be an experience :)
- it's all about to avoid common pitfals, have a good "programming culture" and to know how to measure if it works.

Materials

- Milliseconds Make Millions
- ... many, many articles about "web performance business impact" 😇