Dart

# How does dart VM work?

Raahavajith · Follow

Published in YavarTechWorks

6 min read · Aug 31, 2022

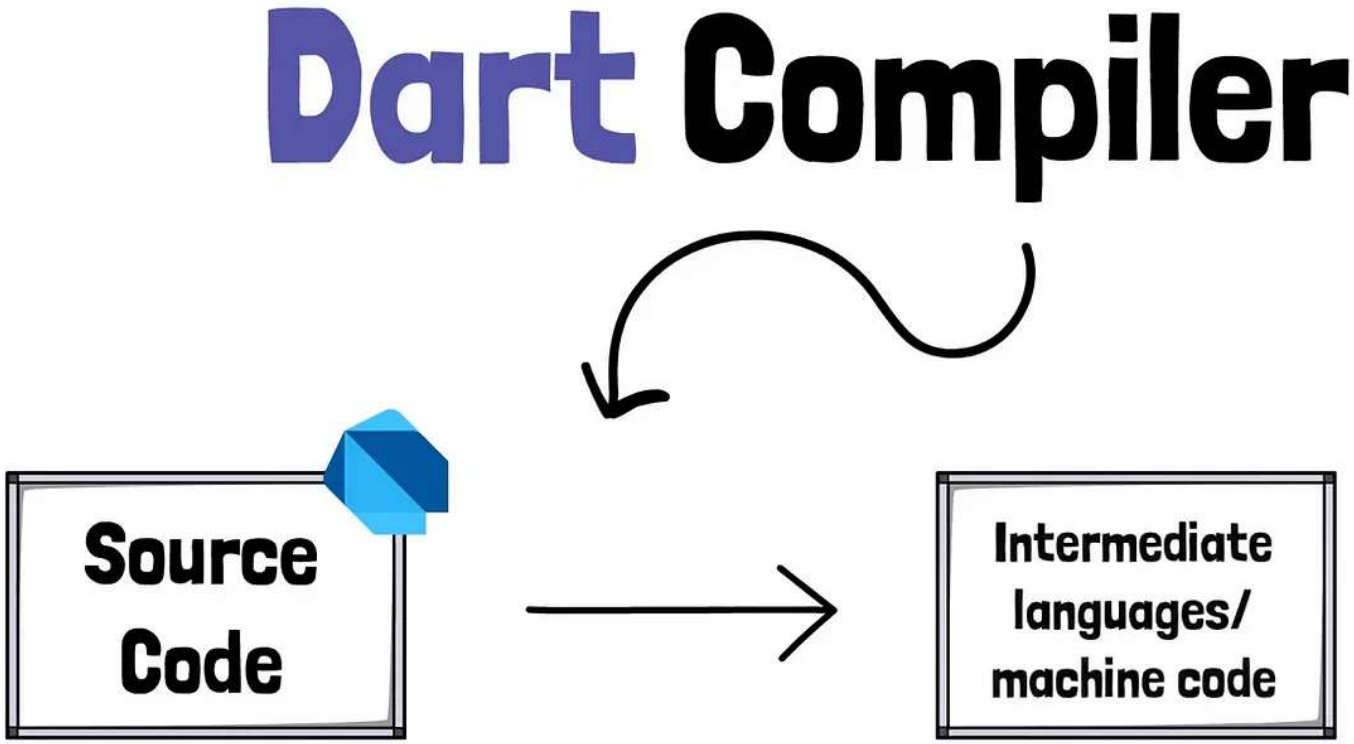


Hello Guys, today we are going to see how the virtual machine works in the amazing multi-platform and client-optimized language DART.

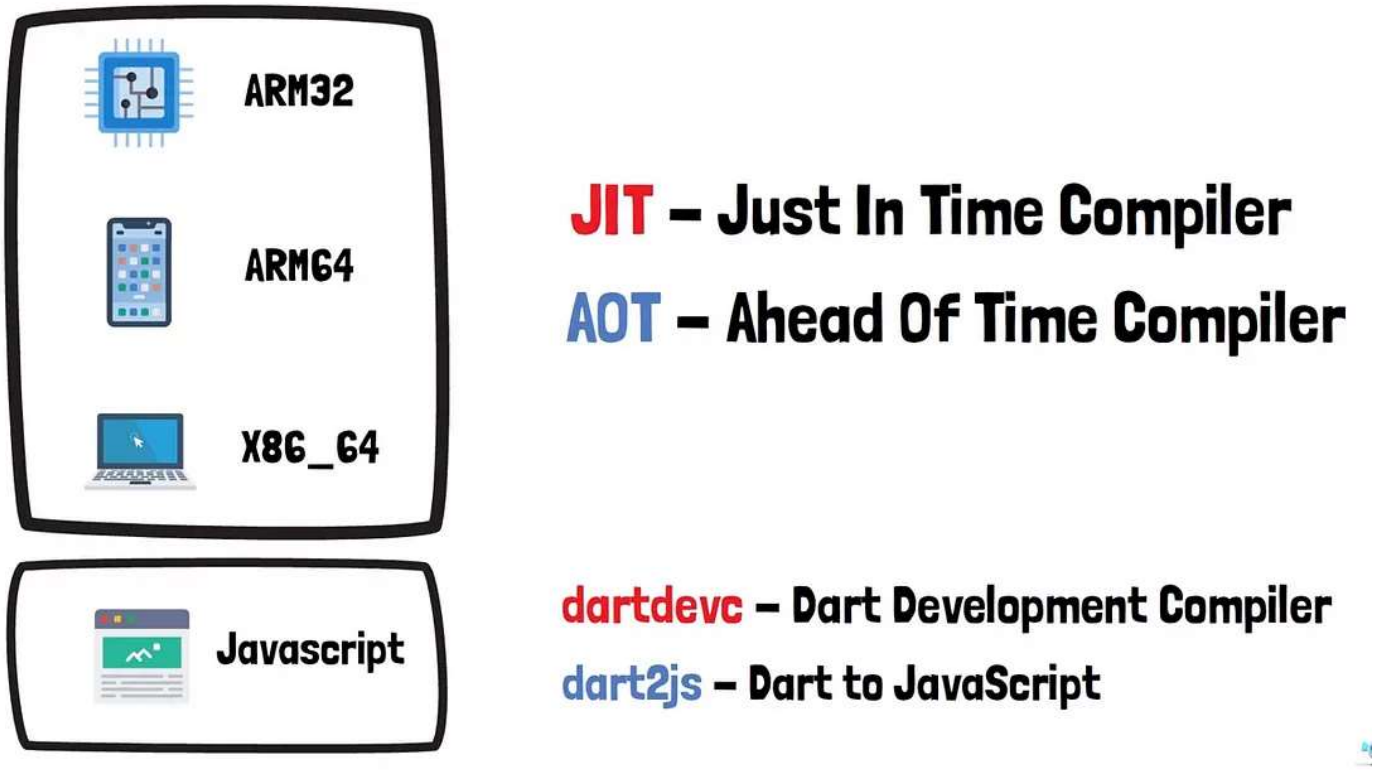
Come on let's go we gonna see,

**Dart Code Run**

Dart Compiler is a tool that converts the source code we wrote in dart language to other intermediate languages or machine code that can run on a specific platform in a dart virtual machine. Like in the below figure




Dart Uses different compilers for different specific jobs. For example, the CPU architecture listed below is usually found on desktops and phones. Dart uses JIT(Just in Time Compiler) and AOT(Ahead Of Time Compiler) for native. On the web obviously, it uses a javascript compiler since it needs to translate the code into javascript language.

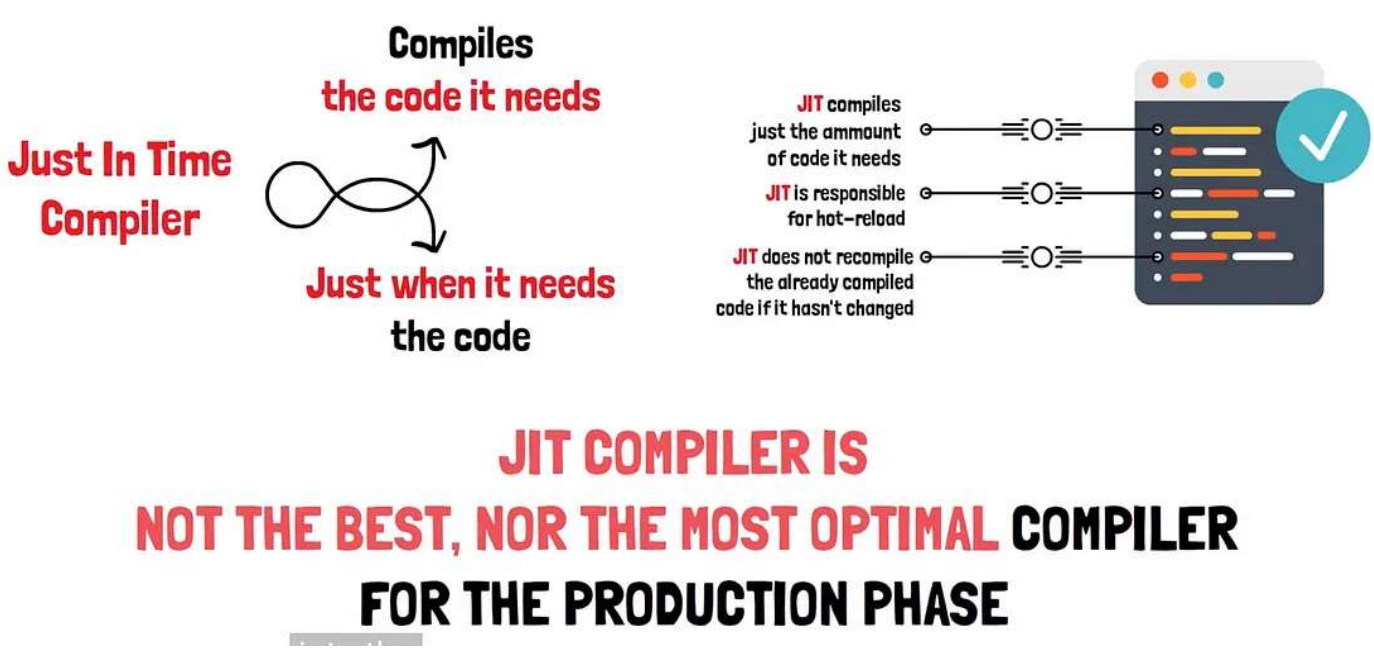


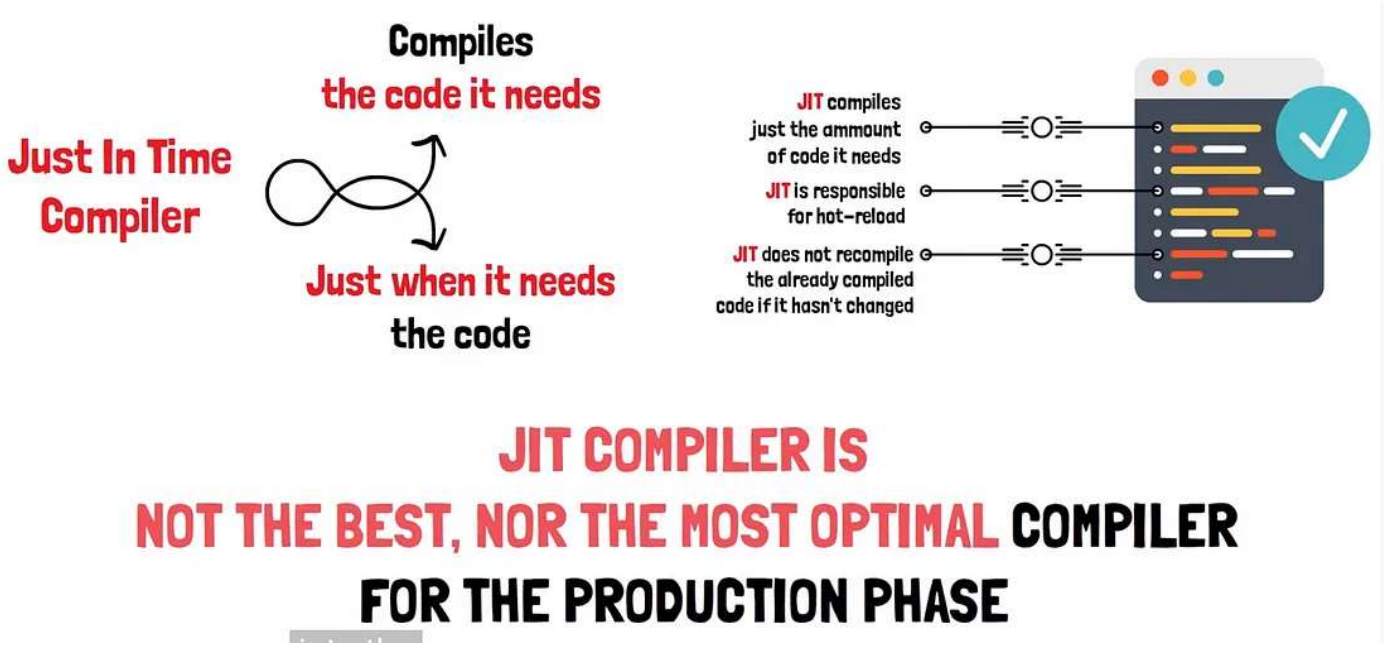

Now we gonna see the two stages of the development process,

- Development Phase and
- Production Phase.

**Development Phase - Just Intime Compiler (JIT)**


- JIT just complies just the amount of code it needs.
- JIT also comes with incremental recompilation so that it only recompiles the modified part when needed.
- JIT compiler is the main star that enables the hot reload in dart during development.






**Production Phase (AOT - Ahead Of Time)**

- The ahead-of-time compiler compiles the entire source code into the machine code supported natively by the platform. It does before the platform runs the program.
- When you decided to promote your code from development state to production state you need to use this compiler to do its specialized jobs.
- To benefit from the best and most optimized version of your code has drawbacks through compiling the same code from scratch over and over and it's not the best solution for developing an app into the development stage.

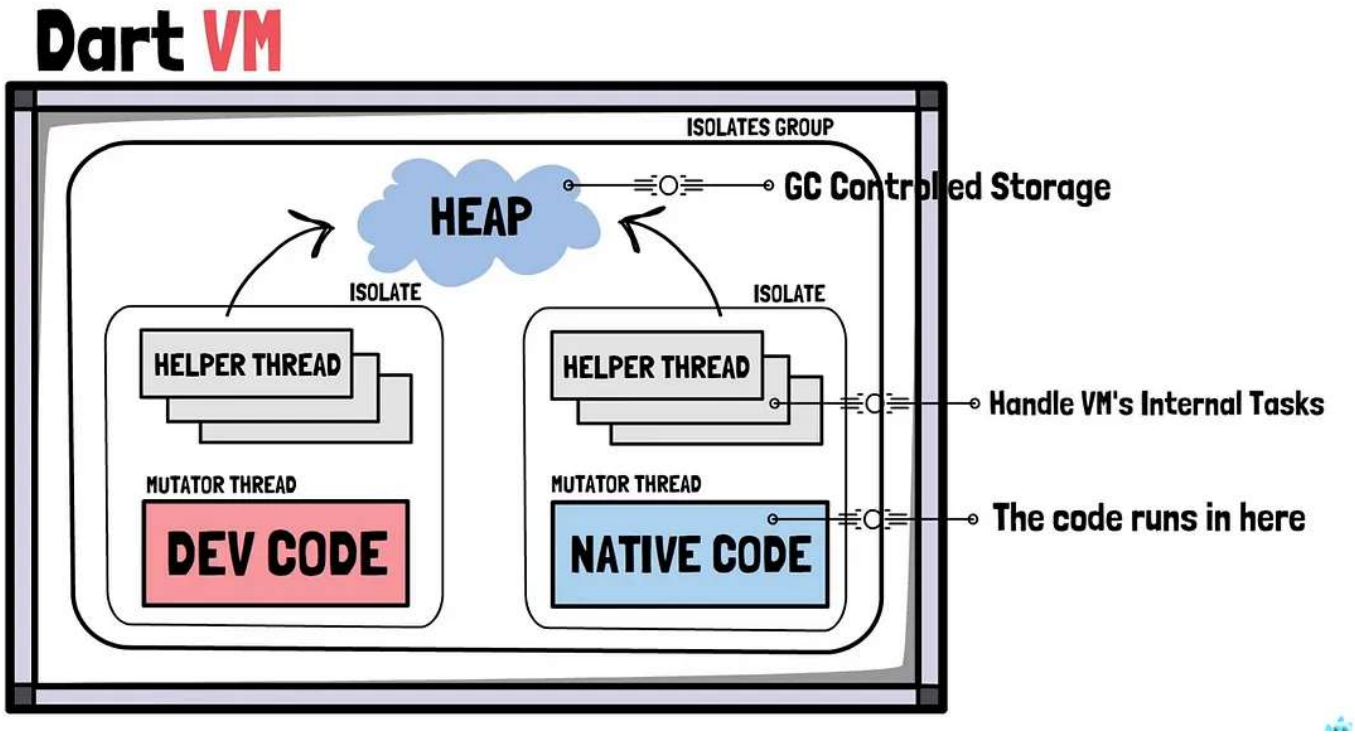
**Dart VM**

Dart VM provides an execution environment for the dart programming language.

The code within the VM is running within the same isolate and it is also called an isolated dart universe with its own memory known as heap its own thread of control called the mutator thread and its own helper thread.

The heap is a garbage collector that manages the memory storage for all the objects allocated by the code running. This isolates the garbage collector's try to reuse the memory which is allocated by the program but no longer referenced.

Each isolate has a single mutator thread that executes the dart code but benefits from multiple helper threads which handle VM's internal tasks.




Components of Dart:

- The Runtime System
- Development experience components debugging and hot reload
- JIT and AOT compilation pipelines

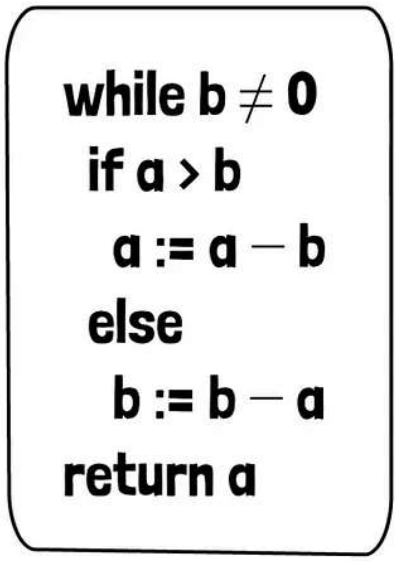
Dart VM can execute dart apps in 2 ways from source by using JIT and AOT compiler and from snapshots JIT, AOT, or kernel snapshots.

The source code had to be directed to the dart VM in order to be run but the dart VM doesn't have the ability to run the raw dart code.

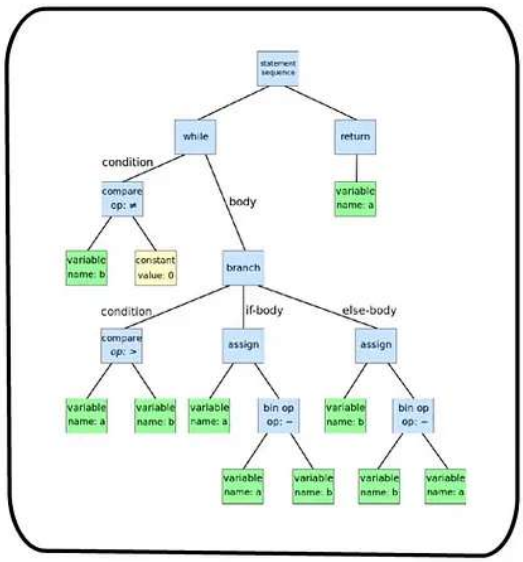
It expects some kernel binary also called .dill files. Which contains serialized abstract syntax tree known as kernel AST. The dart kernel is a small high-level intermediary language derived from dart therefore the kernel AST is actually based on this intermediary language.

The process of translating the dart source code to dart kernel AST is handled by a dart package called Common Front End or CFE.

**For example kernel AST tree which is translated from dart source code,**



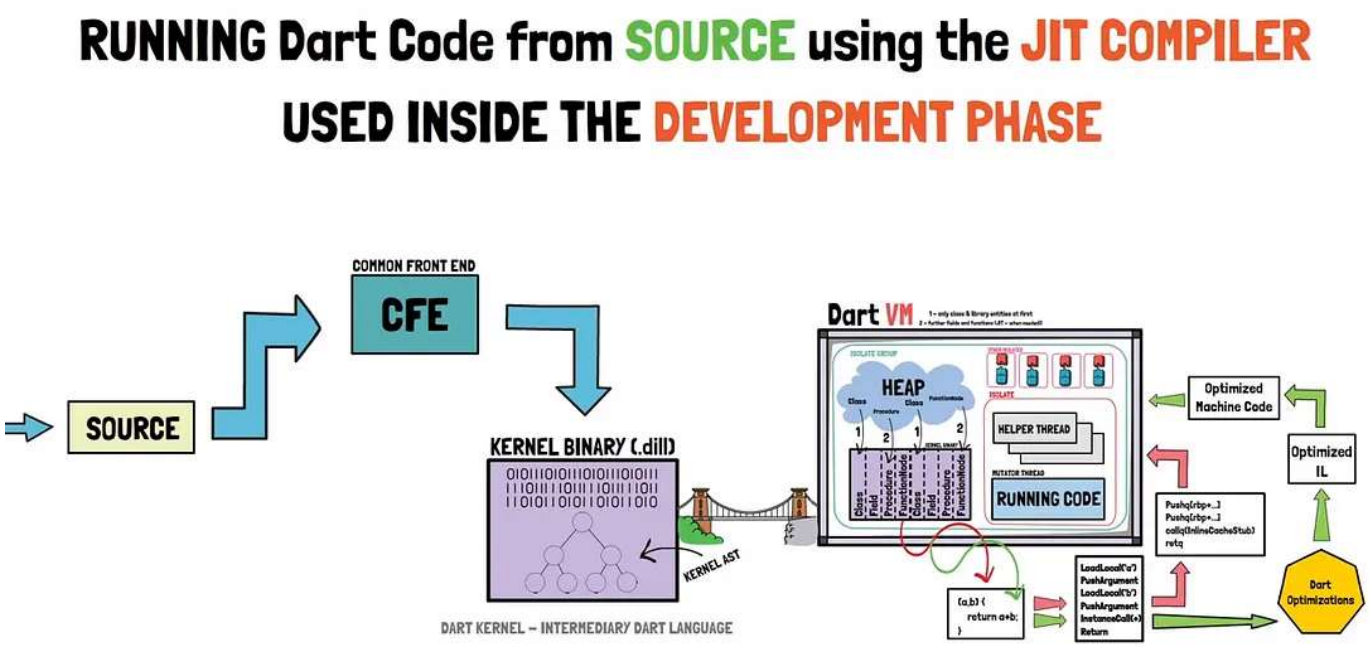

Euclidean Algorithm



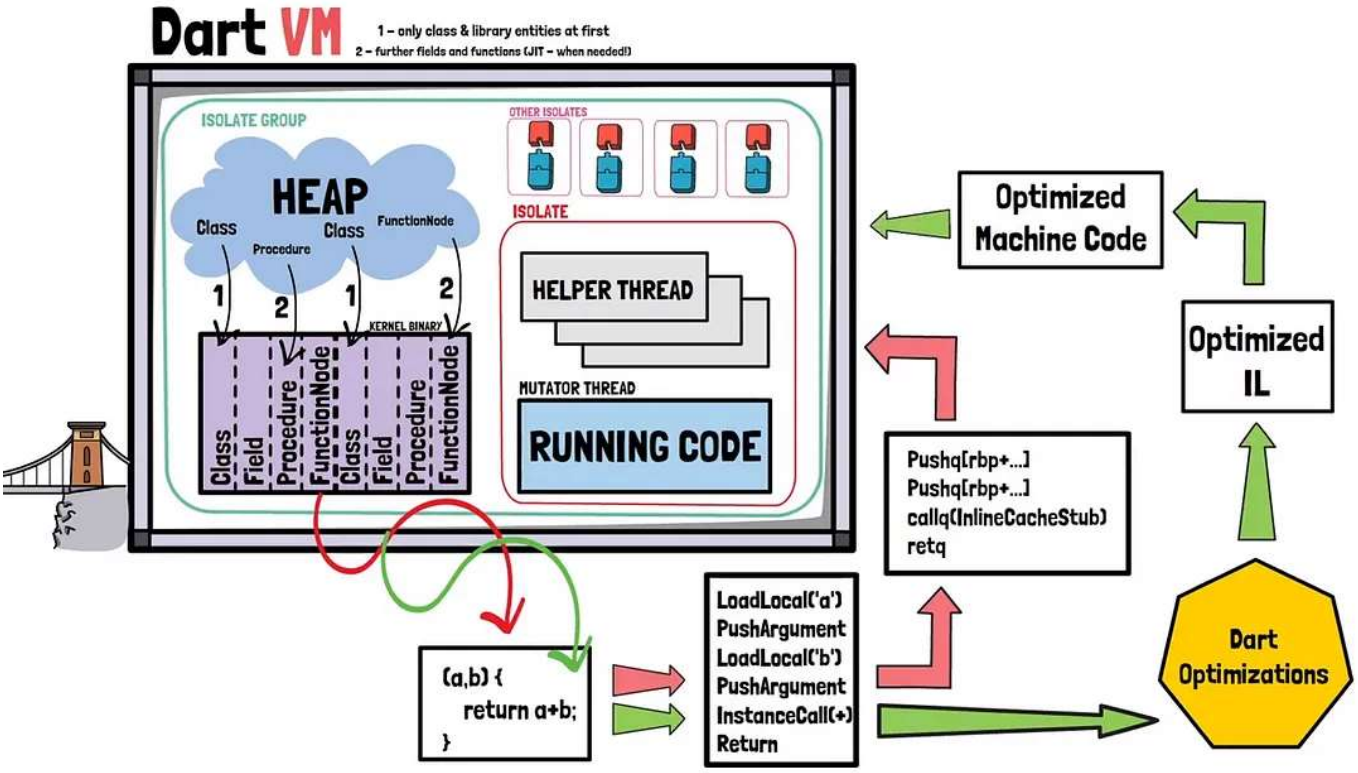

Abstract Syntax Tree

The kernel AST is the file that's being center-right to the VM in the next step the kernel binary is loaded into the dart VM. It is being parsed to create objects representing various program entities like classes and libraries however this is done lazily which means first and foremost it parses basic information about these entities each entity keeps a pointer back to the kernel binary so that later on it can be accessed if needed hence the name of JIT- Just In Time which is similar to just when it needed the

information about the classes is fully deserialized only when the runtime needs it to keep in mind that all those entities lie inside the VM heap which is the VM allocated memory this stage the rest of the entities like fields functions procedures are read from the kernel binary however only their signatures are deserialized at this stage currently there's enough information loaded from the kernel binary or the runtime to start invoking methods, for example, this is when the run time may start to invoke the main function.




From this step first time, our function is compiled it happens sub-optimally which means the compiler goes and retrieves a function's body from the kernel binary and converts it into an intermediate, and then later on this intermediate language is lowered directly without any optimization passes right into pure machine code the main goal of this approach is to produce executable code quickly, however, the next time this function is called it will use optimized code that means instead of directly lowering that intermediate language into machine code the optimized compiler based on the information gathered from the sub-optimal run proceeds to translate the unoptimized intermediate language through a sequence of classical dart specific optimizations.



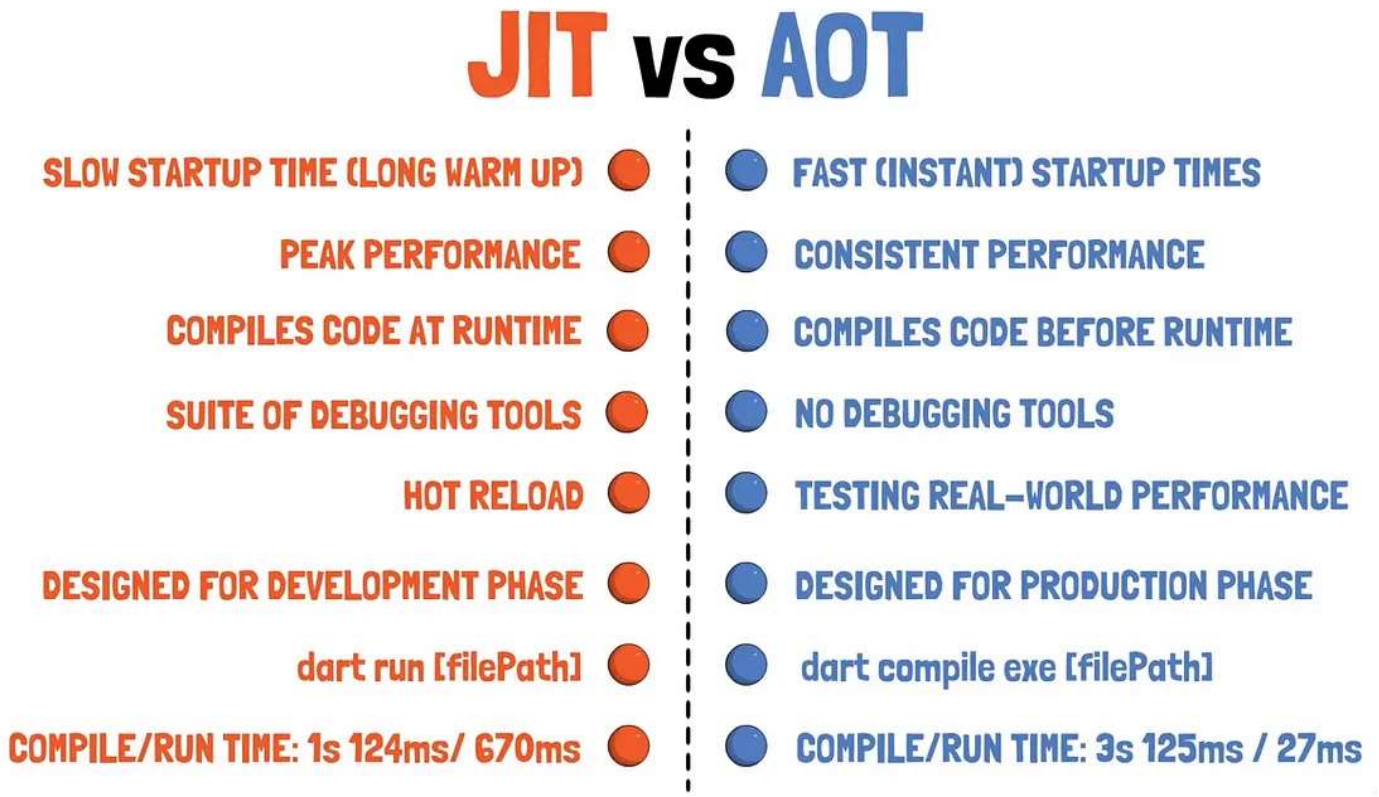

For example inlining range analysis type propagation and so on and so forth finally the optimized intermediate language is again lowered into the machine code and run by the VM and that is it this is what actually happens when you run your dart program by typing in dart on command inside the dart CLI this git approach is used inside the development phase.

We have previously talked about since during this phase a fast developer cycle is critical for iteration therefore since the JIT compiler comes with incremental recompilation enabling hot reload multiple optimization techniques and rich debugging support. It's the perfect choice for this job.


Let's move on to what happens when we run the code from a source this time by using AOT compiler.

AOT compiler was originally introduced platforms which makes JIT compilation impossible. So, it was mainly an alternative to JIT.

**Comparison between JIT and AOT**



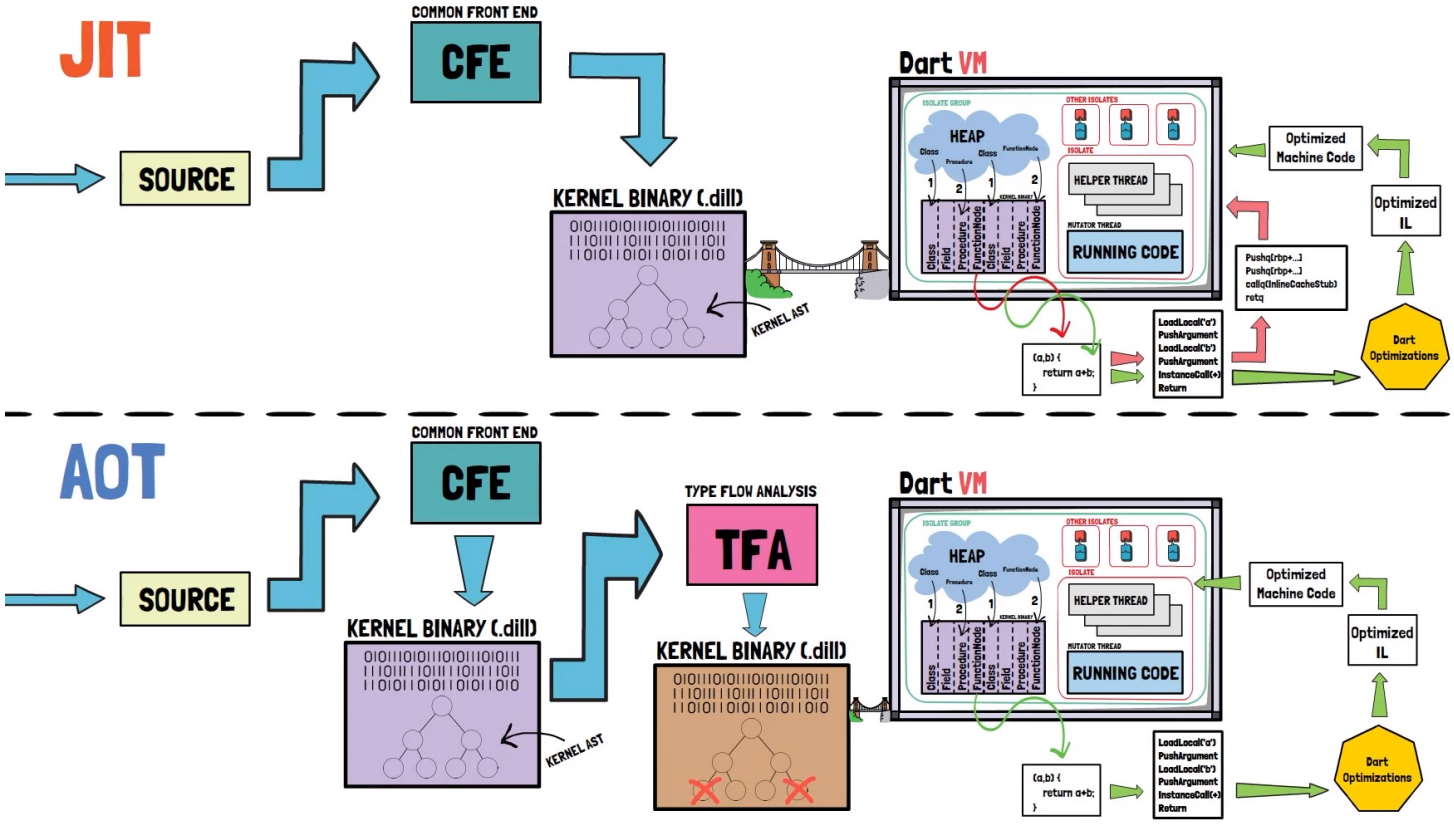

Dart Dartlang Flutter


Follow

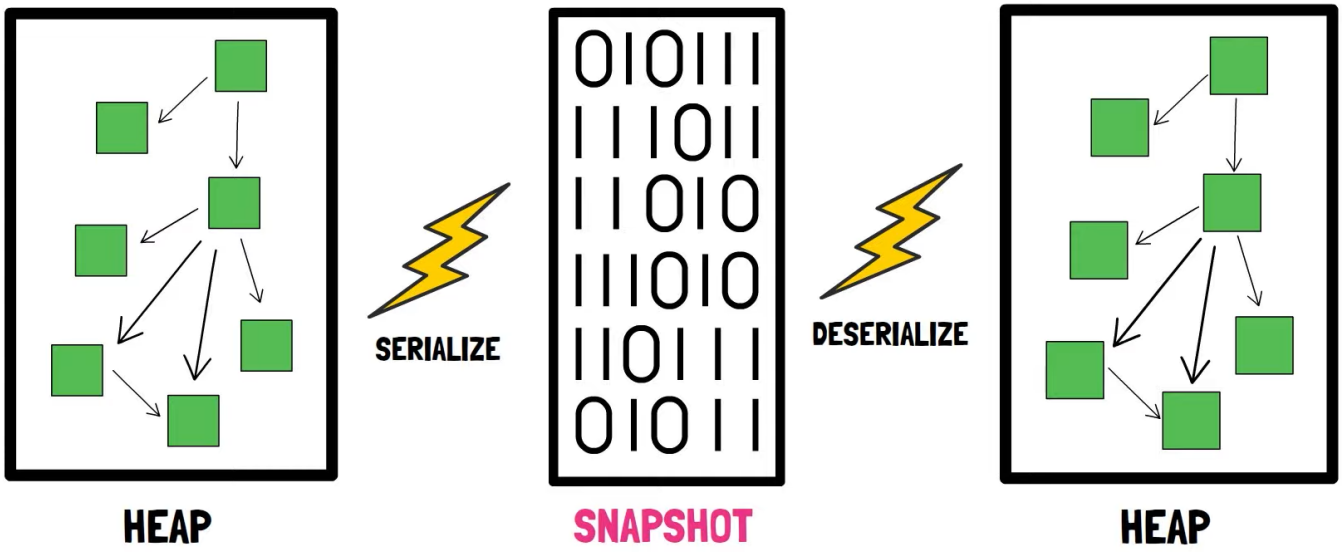
## Written by Raahavajith

10 Followers · Writer for YavarTechWorks

More from Raahavajith and YavarTechWorks



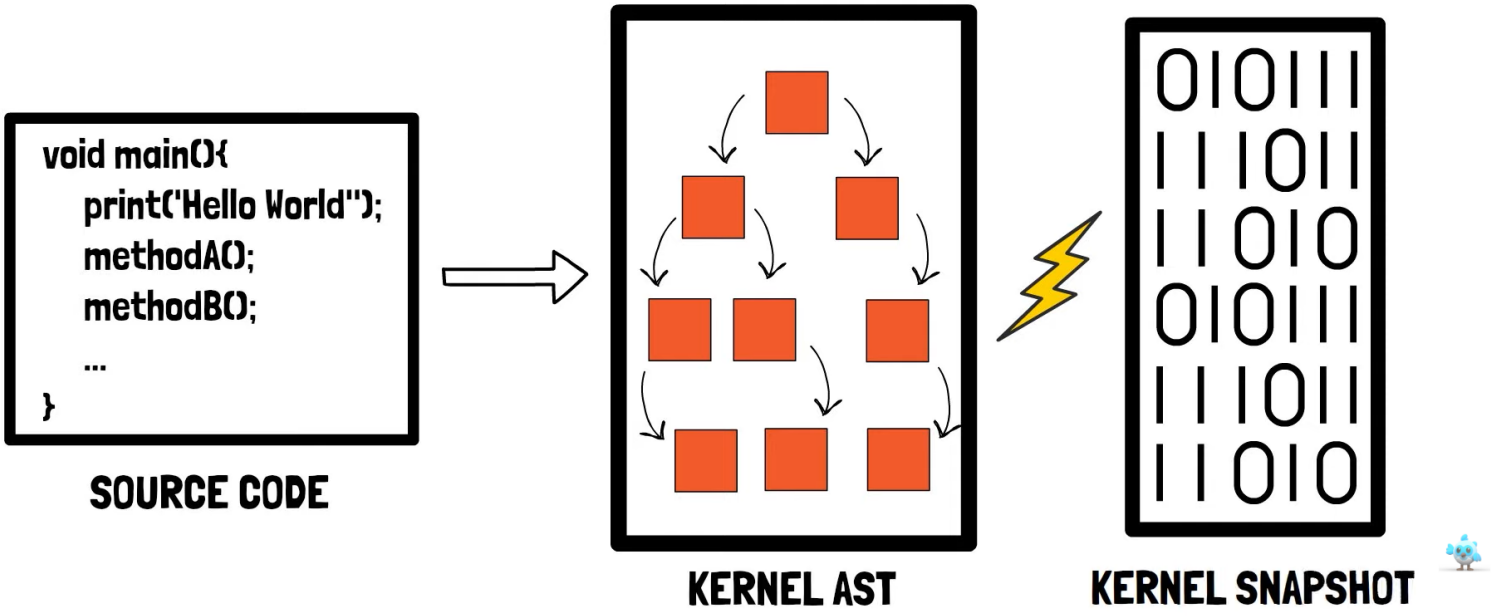

# WHAT IS A DART SNAPSHOT?

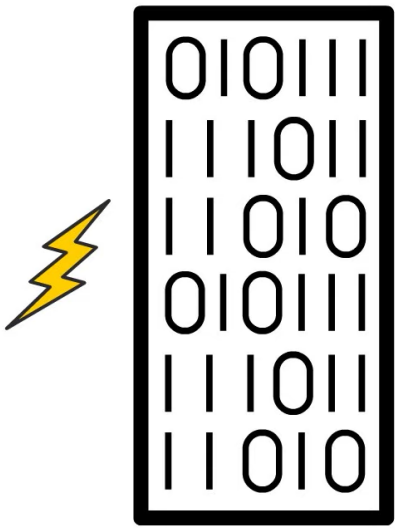


# RUNNING FROM SNAPSHOTS

- JIT-SNAPSHOT
- AOT-SNAPSHOT
- KERNEL-SNAPSHOT

## KERNEL-SNAPSHOT

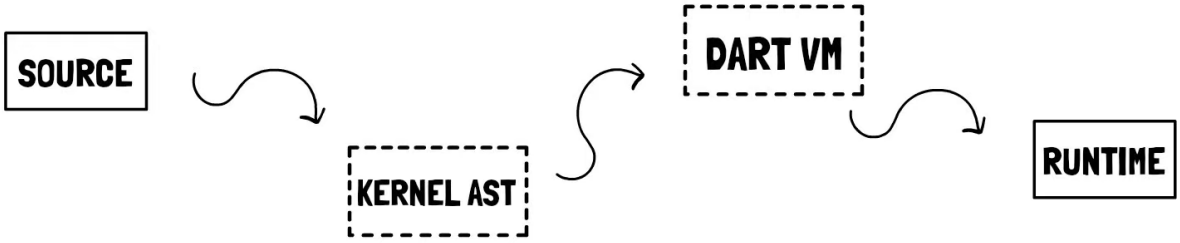




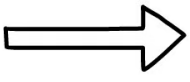
KERNEL SNAPSHOT

- NO PARSED CLASSES, FUNCTIONS
- NO ARCHITECTURE SPECIFIC CODE
- PORTABLE AROUND ALL ARCHITECTURES
- DART VM WILL NEED TO COMPILE IT FROM SCRATCH

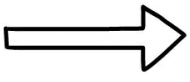
## Training Run

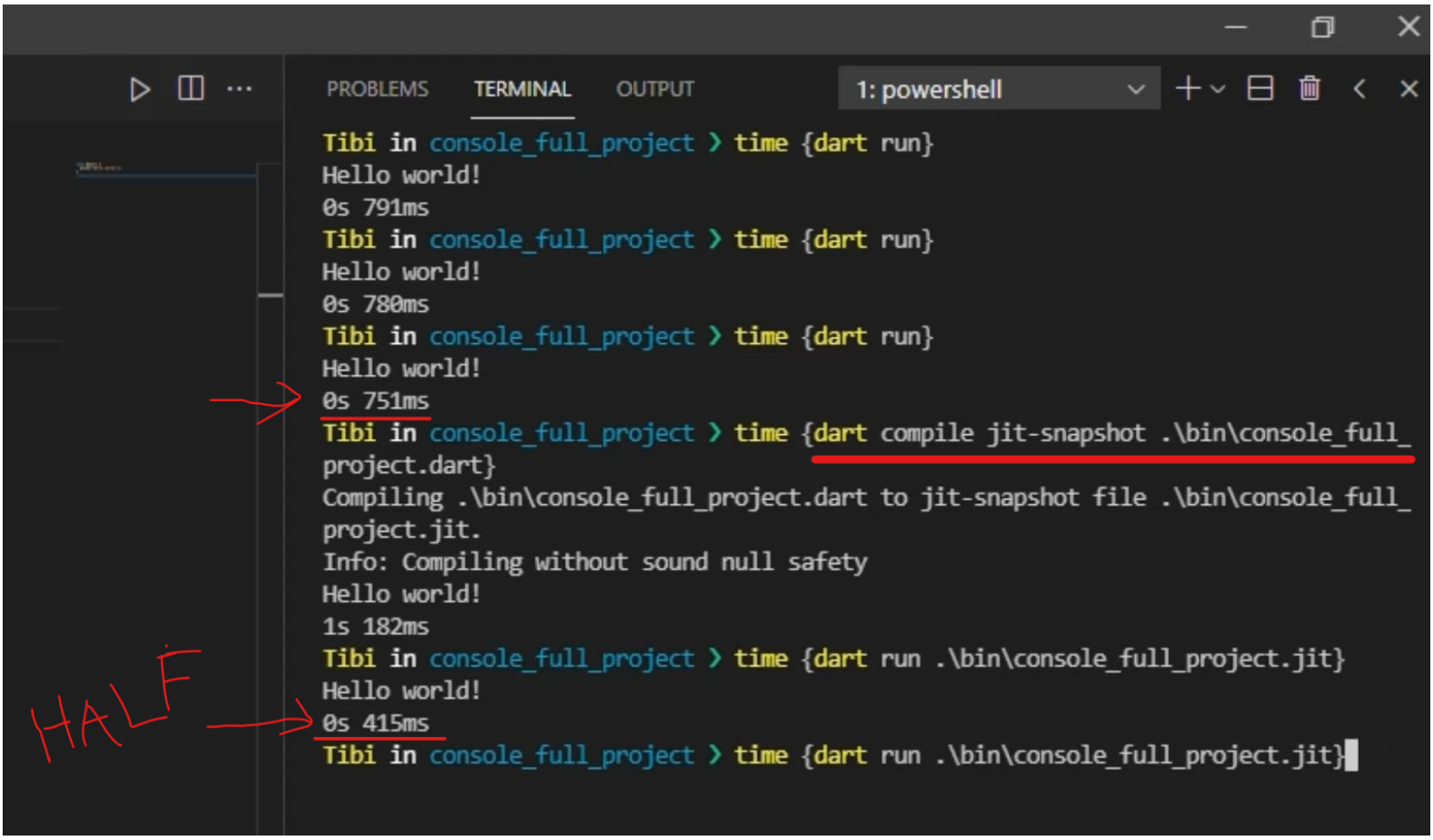



JIT-SNAPSHOT




Dart VM






## STANDARD AOT

Press Esc to exit full screen

⟹ dart compile exe

⟹ standalone platform-specific executable file WITH DART RUNTIME

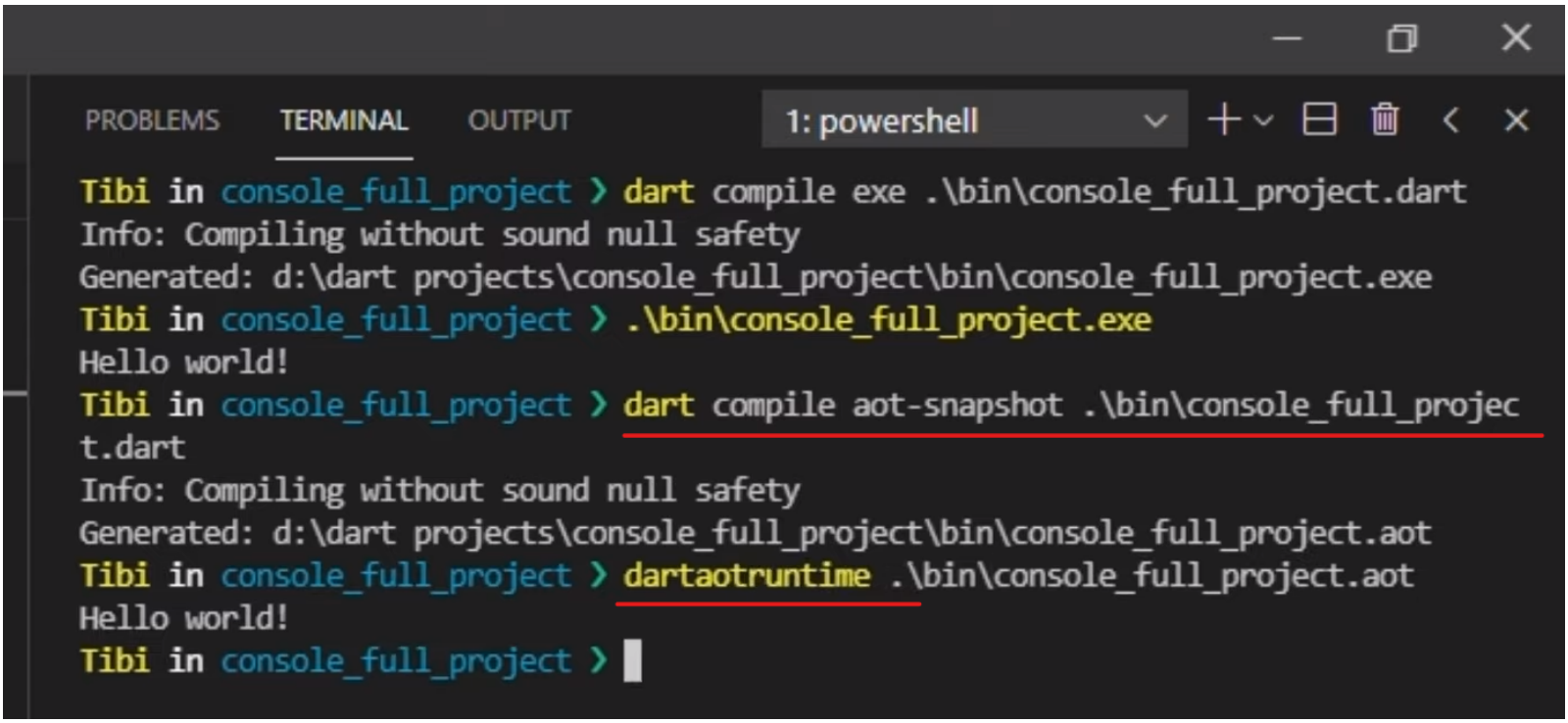
⟹ outputs an AOT-SNAPSHOT & COMBINES it with a DART VM RUNTIME

## AOT-SNAPSHOT

⟹ dart compile aot-snapshot

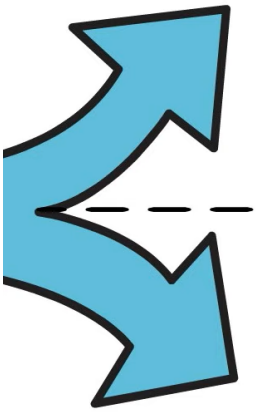
⟹ standalone platform-specific executable file WITH NO DART RUNTIME

⟹ outputs a plain AOT-SNAPSHOT, can be run with dartaotruntime




# JIT COMPILATION + JIT-SNAPSHOT

- PEAK PERFORMANCE, DEBUGGING SUPPORT, HOT RELOAD
- SLOW STARTUP, 2X FASTER WITH JIT-SNAPSHOT
- DESIGNED FOR THE DEVELOPMENT PHASE



- CONSISTENT PERFORMANCE, NO DEBUGGING SUPPORT
- INSTANT STARTUP, HIGH COMPILE TIMES
- DESIGNED FOR THE PRODUCTION PHASE

# AOT COMPILATION + AOT-SNAPSHOT

# We know from tutorial #4 ...

**DART PACKAGE** = directory containing, at minimum, a **pubspec.yaml** file

**DART ECOSYSTEM** uses **PACKAGES** to manage **SHARED SOFTWARE** (libraries & tools)

**PUB PACKAGE MANAGER**

The name should be all lowercase, with underscores to separate words

A version number is 3 numbers separated by dots (1.2.3) 1 = major, 2 = minor, 3 = patch; build ( +1, +2, +hotfix.error) / prerelease (-dev.4, -alpha.12)

The description is the sales pitch for your package. Users see it when they browse for packages.

These fields should contain URLs with the appropiate websites of your package. They help users understand where your package is coming from

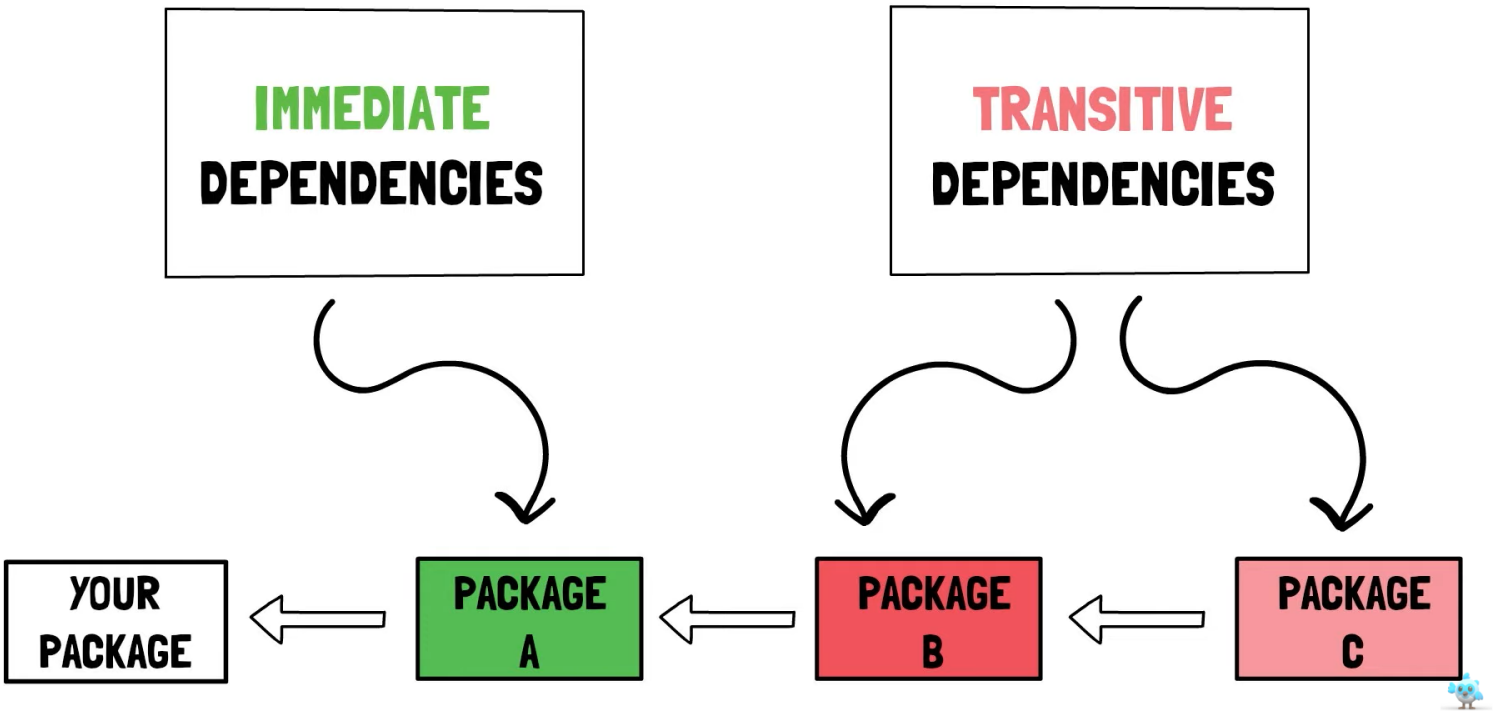
To make a script publicly available, list it under the executables field. Once the package is activated using pub global activate, typing wckd_package executes bin/main.dart.
Typing wckd executes bin/wckd.dart.

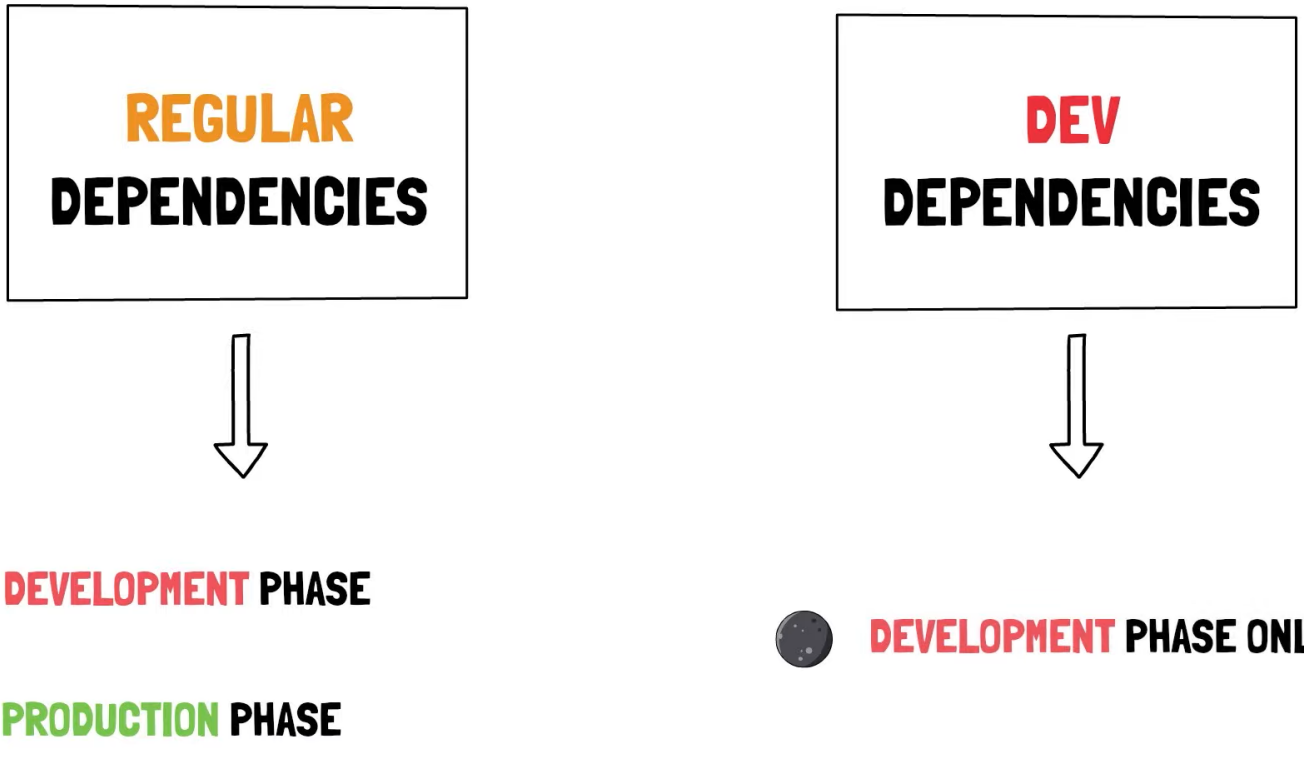
The default uses the pub.dev site. Specify none to prevent a package from being published.

The Dart platform evolves over time, and a package might only work with certain versions of the platform.A package can specify those versions using an SDK constraint.

A dependency is just another package that your package depends on, in order to work, and they are specified in the pubpsec.yaml file under the dependencies field.

```yaml
name: wckd_package

version: 1.2.3

description:
  Wanna become a part of wckd community? The moment has come
  to showcase your wckd implementations, try this package today!

homepage: https://wckd-package.com/news
repository: https://github.com/TheWCKD/wckd_package
issue_tracker: https://github.com/TheWCKD/wckd_package/issues
documentation: https://wckd-package.com/docs

executables:
  wckd_package: main
  wckd:

publish_to: none

environment:
  sdk: '>=2.10.0 <3.0.0'

dependencies:
  bloc: ^8.0.0
  http: ^14.3.2

dev_dependencies:
  test: '>=1.15.0 <2.0.0'
```


IMMEDIATE
DEPENDENCIES
TRANSITIVE
DEPENDENCIES
YOUR
PACKAGE
PACKAGE
A
PACKAGE
B
PACKAGE
C


REGULAR
DEPENDENCIES
DEV
DEPENDENCIES
DEVELOPMENT PHASE
PRODUCTION PHASE
DEVELOPMENT PHASE ONLY

# SEMANTIC VERSIONING

– https://semver.org –

pubspec.yaml

```
dependencies:
  bloc: ^8.0.1
dev_dependencies:
  test: ^0.16.2
```

## GENERAL RULE:

dependency: ^1.2.3 ⟹

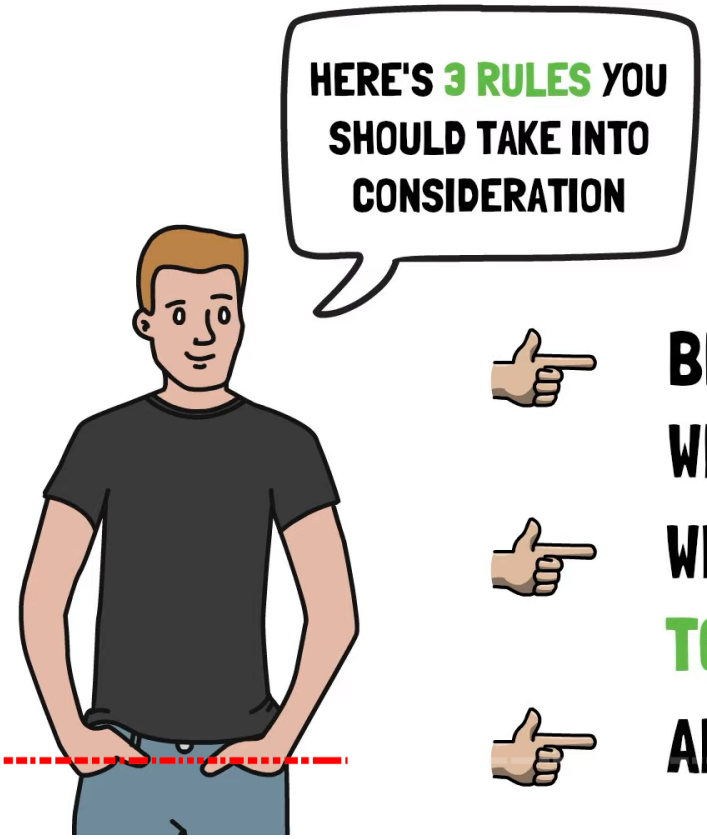
- dependency: 1.3.0
- dependency: 1.4.7
- dependency: 1.8.3
- dependency: 1.9.9
- dependency: 2.0.0


UNORGANIZED

INCONSISTENT

OVERCOMPLICATED

CODE

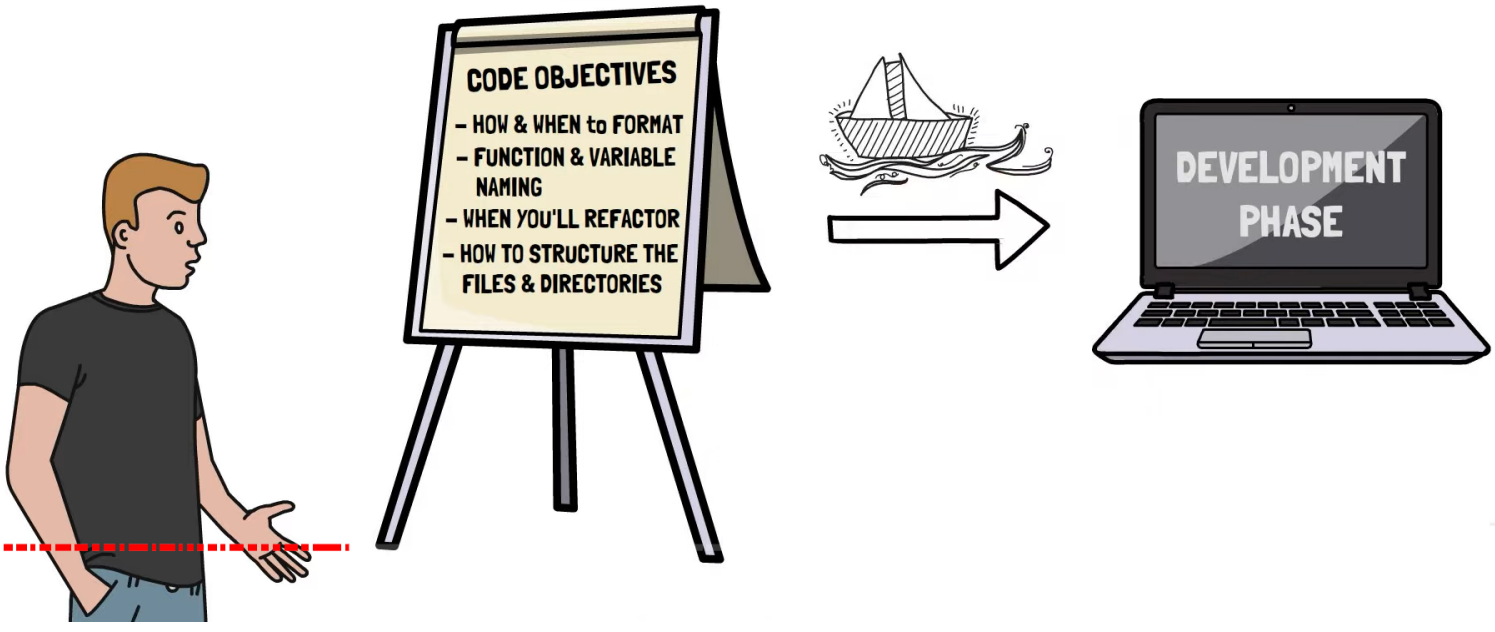



BE ORGANIZED & CONSISTENT WITH YOUR CODE.

WRITE MINIMALIST & SIMPLE TO UNDERSTAND CODE.

ALWAYS TEST YOUR CODE.

## BE ORGANIZED & CONSISTENT WITH YOUR CODE.



# WRITE MINIMALIST & SIMPLE TO UNDERSTAND CODE.

```
double radius = 2;
double pi = 3.1415;
double area = 2*pi*
pow(radius,2);
```

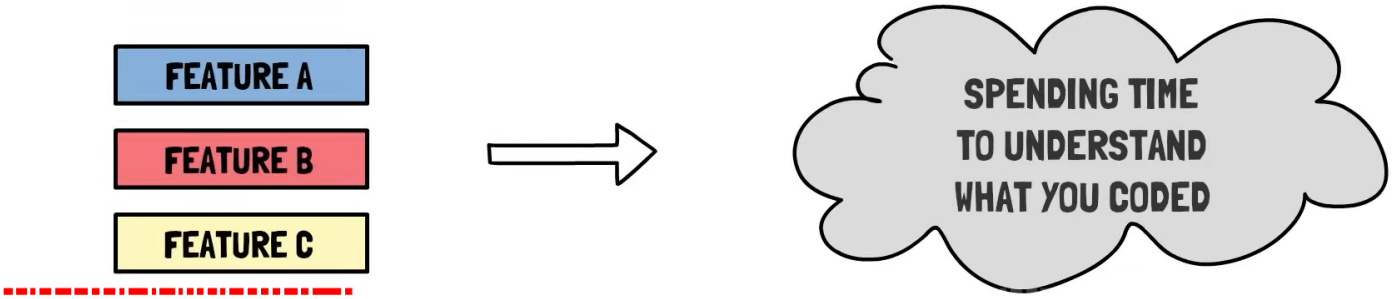
reusable, cleaner

```
import dart:math as math;

double calculateArea(double radius)
    => return 2*(math.pi)*pow(radius,2);
```

MINIMALISM = SMARTER, SIMPLER TO UNDERSTAND,
MORE PERFORMANCE EFFICIENT

MINIMALISM = SMARTER, SIMPLER TO UNDERSTAND,
MORE PERFORMANCE EFFICIENT


# DEVELOPMENT WORKFLOW MUST NOT BE RUSHED


# ALWAYS TEST YOUR CODE.

- For a SMALL APP, TESTING may be OMITTED, since it can be MANUALLY PERFORMED
- MEDIUM to BIG projects should ALWAYS use TESTS