

Intro til Rust




Agenda

- Intro på ca. 30 min
- Workshop

Installer rustup!

- Versjonsmanager av verktøy til språket
- <https://rustup.rs/>
- Følg og fullfør installasjonen (f.eks. på Windows kreves MSVC++-verktøy, XCode tools (LLVM) på macOS)

Hva er Rust?

- Programmeringsspråk, håper du visste det...
- Lansert versjon 1.0 i 2015, påbegynt allerede i 2006, men var svært annerledes
- Statisk typesystem, a la C#/Kotlin/Java/C/ osv., men ikke Python/JS osv.
- Kompilerer til én executable/binary, krever ikke VM/runtime som Java/Kotlin/C#/Go
- Ingen garbage collection 

Egenskaper

- Ytelse (Performance)
 - Ingen garbage collection
 - Ingen runtime
- Pålitelighet (Reliability)
 - Sterkt typesystem
 - Minnesikkerhet og trådsikkerhet
- Produktivitet (Productivity)
 - God dokumentasjon
 - Gode feilmeldinger og tooling

Hva brukes det til?

Build it in Rust

In 2018, the Rust community decided to improve the programming experience for a few distinct domains (see [the 2018 roadmap](#)). For these, you can find many high-quality crates and some awesome guides on how to get started.



Command Line

Whip up a CLI tool quickly with Rust's robust ecosystem. Rust helps you maintain your app with confidence and distribute it with ease.



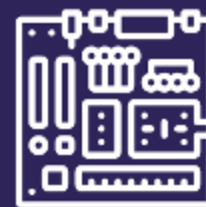
WebAssembly

Use Rust to supercharge your JavaScript, one module at a time. Publish to npm, bundle with webpack, and you're off to the races.



Networking

Predictable performance. Tiny resource footprint. Rock-solid reliability. Rust is great for network services.

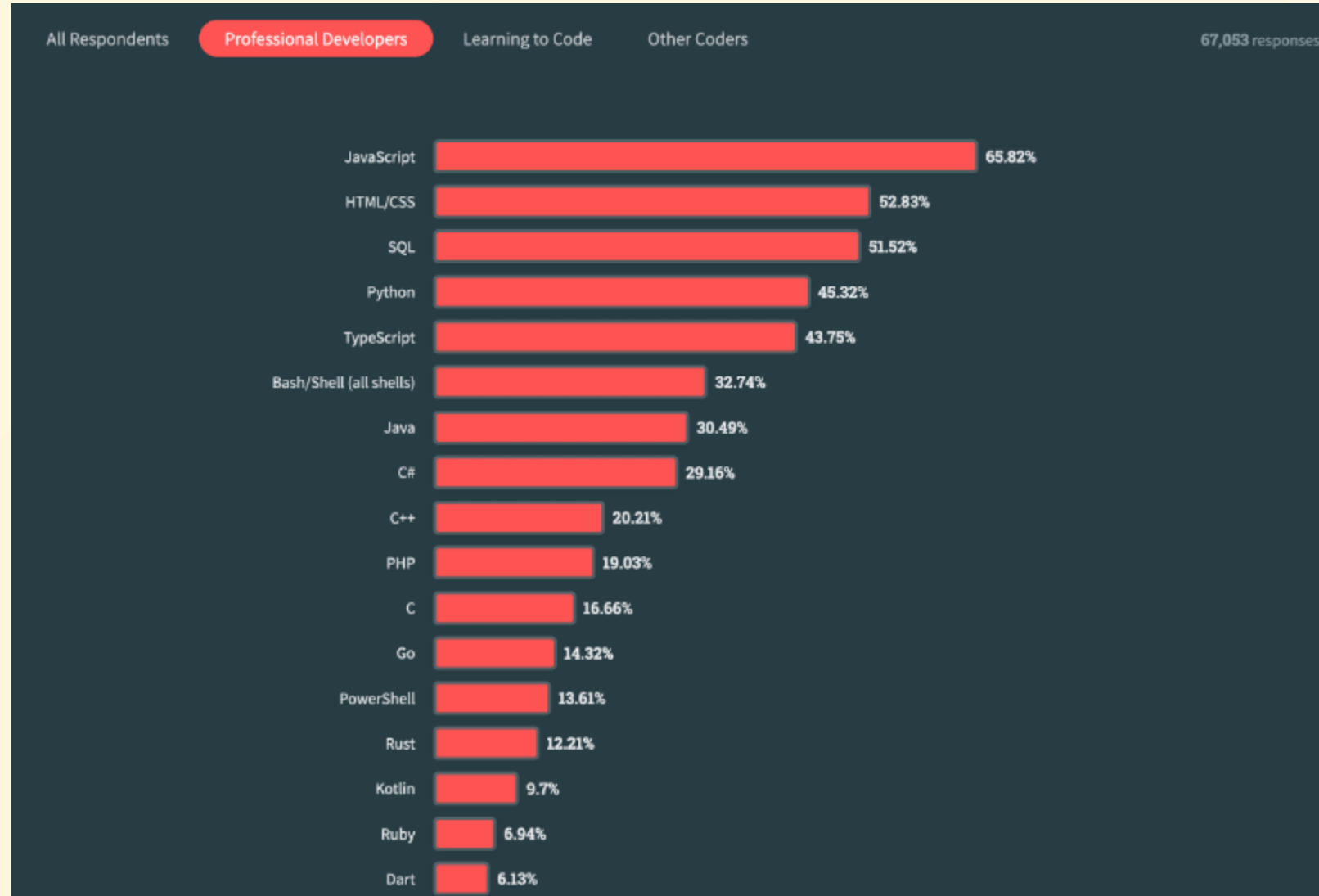


Embedded

Targeting low-resource devices? Need low-level control without giving up high-level conveniences? Rust has you covered.

Brukes det egentlig av noen?

Brukes mer enn Kotlin!



Forskjellen på Rust og andre språk

- Garbage-collecta språk (C#, F#, Haskell, Java, Kotlin, JS, Python, Dart, Swift, Go, osv.)
 - Mindre å tenke på
 - Tregere
- Manuell minnehåndtering (malloc/free) (C, C++, Zig, Assembly)
 - Mer å tenke på
 - Betydelig raskere og mindre ressurskrevende
- Rust
 - Litt mer å tenke på, men like raskt som manuelt håndtert

Ingen garbage collection?

- Garbage-collection betyr at språket/runtimeet håndterer allokering og deallokering av minne for deg
- `var x = new Thing();` i C# allokterer for deg og deallokerer etter hvert når du slutter å bruke `x`
- `var x = Thing();` i Rust gjør det samme for deg, men umiddelbart etter du ikke bruker `x`, og du slipper overhead med at det skjer i runtime, siden dette håndteres ved kompilering

Hvordan ser syntaksen ut?

Java:

```
Thing thing = new Thing();
```

C:

```
Thing* thing = create_thing();  
if (thing == NULL) return NULL;  
// Do stuff, then deallocate  
free(thing);
```

Hvordan ser syntaksen ut?

```
// On the stack  
let thing = Thing();  
// On the heap  
let thing_on_the_heap = Box::new(Thing());
```

Hvordan er typesystemet?

- Ingen `null`
- Ingen exceptions
- Immutable by default
- Eksplisitt fremfor implisitt
- Typeinferens
- Generics
- Traits
- Algebraiske datatyper
- Lifetimes, eller *hvor lenge lever en verdi*

Algebraiske datatyper?

- I ca alle andre språk vi bruker støttes kun *product types*
- En klasse kan ha felt `read` og `write` som er bools, som da tilsvarer fire tilstander
- Hva er greia når `read` er `false` og `write` er `true`?
- Da tyr man til enums, `ReadOnly`, `ReadAndWrite`, og `NoAccess`
- Hvor og hva er *sum types*?

Sum types

- Summen av alle tilstandene en type kan ha, ikke produktet
- bools har to typer, så to felt med bools blir $2*2=4$ tilstander
- Enumen tidligere har tre tilstander, `ReadOnly`, `ReadAndWrite`, og `NoAccess`
- Men sum-typer kan også bære med seg data

Endelig litt Rust!

```
enum OptionallyCarriesString {  
    Nothing,  
    Some(String)  
}
```

Tilstanden er alltid kun én av de to. Enten `Nothing` eller `Some("123456")`

```
struct Person {  
    age: u8,  
    name: String  
}
```

Må vi lage en sånn type hver gang vi ville skrevet `null` ellers?

Generics!

```
enum Option<T> {  
    None,  
    Some(T)  
}
```

Tilsvareer `null` i andre språk, men er ikke en default-verdi *a/t* kan ha

Rust primitive data types

- `bool` som er `true` eller `false`
- `i32`: heltall som er 32 bits, signed, fra `-2147483648` til `2147483647`
- `u32`: heltall som er 32 bits, unsigned, fra `0` til `4294967295`
- Flyttall: `f32`, `f64`
- Resten av heltallene: `u8`, `i8`, `u16`, `i16`, `u64`, `i64`, `u128`, `i128`
- `usize` og `isize` som er antall bits i arkitekturen på CPU (64-bit)
- Unit: `()`, den tomme typen, ligner på `void` i andre språk
- `1..100`, exclusive range med alle tall f.o.m 1 til 100, `1..=100`, inclusive range emd alle tall f.o.m. 1 t.o.m. 100

Feilhåndtering

Ingen exceptions?? 🙄 Neida

```
enum Result<T,E> {  
    Ok(T),  
    Err(E)  
}  
  
let result: Result<GoodValue, Error> = something_that_can_fail();
```

Hvordan sjekker vi disse verdiene?

- `match` er en mye bedre versjon av `switch` / `case`

```
let maybe_something: Option<i32> = get_value();
match maybe_something {
    Some(x) => do_stuff_with(x),
    None => handle_missing_value(),
}
let maybe_error: Result<i32, Error> = something_that_can_fail();
match maybe_error {
    Ok(x) => do_stuff_with_ok_value(x),
    Err(e) => handle_error_somewhat(e),
}
```

Kan gjøre mye mer

```
let maybe_restaurant = Some(Restaurant {  
  name: String::from("Hos Thea"),  
  address: Address {  
    street: String::from("Strandveien 123"),  
    postal_code: 9006,  
  },  
});  
match maybe_restaurant {  
  Some(Restaurant {  
    adress: Address {  
      street,  
      postal_code: 9000..=9299,  
    },  
    name,  
    ..  
  }) => println!("{name} ligger i {street} i Tromsø"),  
  _ => /* Do nothing */,  
}
```

Funksjoner

```
fn process_something(data: Data) -> Output {  
    let output = process_data(data, some_options_or_something);  
    // Implicitly returns the last non-semicolon terminated value  
    output  
  
    // Can also be written explicitly with  
    // return output;  
}
```

Generiske funksjoner

Når du vil abstrahere over forskjellige datatyper

```
fn do_stuff<T>(t: T) {  
    // But what can you really do with a totally generic type you know nothing about?  
}
```

Traits

- Som interfaces i andre språk. Mer lik typeclasses fra Haskell

```
trait Bark {  
    fn bark() -> String;  
}  
  
impl Bark for Dog {  
    fn bark() -> String {  
        String::from("woof")  
    }  
}  
  
impl Bark for Fox {  
    fn bark() -> String {  
        String::from("Wa-pa-pa-pa-pa-pa-pow!")  
    }  
}
```

Generics med trait bounds

Nå kan vi skrive en funksjon som printer noe ved å bruke traiten som sir om noe kan printes, `Display`

```
fn print_it_my_way<T: Display>(t: T) {  
    println!("WOW LOOK AT THIS: {t}!");  
}
```


Hello world

```
fn main() {  
    println!("Hello, world!");  
}
```

Men hva er `!` i `println!` der?

Makroer

- Makroer er kode som genererer kode
- `println!("Hello, world!")` utvides til koden som trengs for å printe noe
- Veldig nyttig for å slippe duplikasjon, kompleks kode, boilerplate
- Kan tillate nye patterns og ting som DSL (domain-specific language), men pass på å ikke misbruke
- F.eks. `dbg!(x)` der `x` er en variabel med 3 vil printe `3` med ekstrainfo slik:

```
[src/main.rs:2] x = 3
```

Mer om **dbg!**

dbg! (3) utvides til følgende så du slipper å skrive det!

```
match 3 {  
  tmp => {  
    {  
      $crate::io::_eprint(builtin #format_args("[{}:{}] {} = {:#?}", "", 0u32, "x", &tmp));  
    };  
    tmp  
  }  
}
```

Eierskap

- Alle verdier i Rust har en eier
- Når eieren ikke lenger bruker verdien vil den deallokeres (droppes)
- Eieren av en verdi kan låne ut tilgang til verdien via referanser, eller gi fra seg eierskapet
- Lånene kan enten være lesereferanse eller skrive- og lesereferanse, og det kan kun enten være den ene eller den aktive på et gitt tidspunkt
- En som låner en referanse kan dele ut nye lån, så lenge de varer kortere enn man selv låner for

Referanser

- Anta vi har en type `T`. Har vi en verdi av typen `T` tilsier det at vi er eieren
- For å låne ut en referanse til `T` skriver vi `&T`, og typen kalles også `&T`. Dette er en lesereferanse. Ingen andre kan skrive til verdien så lenge referansen lever.
- For å låne ut en skrive-(og lese)-referanse skriver vi `&mut T`, og som navnet tilsier har vi *mutable* tilgang, også mer korrekt kalt *exclusive access* Ingen andre kan lese eller skrive til verdien så lenge denne eksisterer.

Eksempel

Si vi har en dagbok:

```
struct Diary {  
    notes: String,  
}  
  
let mut diary = Diary { notes: String::new() };  
  
fn lend_diary(diary: &Diary) {  
    println!("{}", diary.notes)  
}  
  
fn write_diary(diary: &mut Diary) {  
    diary.notes.push_str("Omg you wouldn't believe what happened today")  
}
```

Eksempel del 2

Vi kan fint både låne ut lesereferanser og eksklusive referanser, hvorfor?

```
lend_diary(&diary);  
lend_diary(&diary);  
lend_diary(&diary);  
write_diary(&mut diary);  
lend_diary(&diary);
```

Pga. lifetimes. Alle referansene vi lager og låner ut varer kun til funksjonene returnerer.

Eksempel del 3

Å lagre notatene på en eller annen plass ville gitt en feil når vi lager referansen. F.eks.

```
fn store_notes(diary: &Diary) {  
    // Anta vi har storage: &mut &String en eller annen plass  
    // *storage gir oss mulighet til å modifisere det &mut refererer til, nemlig &String  
    *storage = &diary.notes;  
}
```

Å kalle denne funksjonen fra der vi har dagboken ville gitt feilmelding på at referansen vi lager til dagboken kanskje lever for kort, så man må være tydelig med om **storage** sin referanse eller dagboken sin referanse skal leve lengst.

Lifetimes

- En del av typesystemet, skrives som en del av typesignaturen som en generisk type, siden det er noe kompilatoren finner ut av
- `&T` skrives eksplisitt som `&'noe T`, der `'noe` er navnet på lifetimen til referansen. Navnet har ingenting å si, så det er ofte vanlig å bruke `'a`, `'b`, osv.
- F.eks. kan vi skrive signaturen til `read_diary` eksplisitt med:

```
fn read_diary<'a>(diary: &'a Diary)
```

- Rust har lifetime elision som vil si at i 9/10 tilfeller skjønner kompilatoren hvilke lifetimes referanser bør ha og bruker de

Moduler og namespaces

```
// also equivalent to placing the contents in `diary_module.rs`  
// and writing `mod diary_module;` here instead  
mod diary_module {  
    struct Diary;  
    struct SomethingElse;  
}  
  
// we can use items within modules using ::  
fn read_diary(diary: &diary_module::Diary) {  
}  
  
// or import them using `use`  
use diary_module::Diary;  
// Or import multiple values  
use diary_module::{Diary, SomethingElse};
```

Bruker også `::` for å gå inni typer

Kan brukes til å hente ut alle ting som er *statiske*

```
enum MinType {  
    MinVariant  
}  
  
impl MinType {  
    fn min_metode() {  
    }  
}  
  
// Kan nåes med  
MinType::MinVariant;  
MinType::min_metode();
```

Løkker og sånt

Rust har også løkker på lik linje som andre språk:

- `while`-loops, a la `while condition { body }`
- `for`-løkker, a la `for i in 0..100 { body }`
- `loop`-løkker, a la `loop { forever }`

Iteratorer

Dersom du skal prosessere data er det vanlig og idiomatisk å bruke iteratorer. Altså med `.map`, `.filter`, `.fold` og en haug andre hjelpefunksjoner. Dersom du har en liste med ting ville vi kanskje skrevet noe slikt:

```
noe_som_kan_itereres_over
  .iter()
  .map(|x| x * 2)
  //      ^^^^^^^^^
  // En closure i Rust, med argumentene innenfor |pipes|
  .filter(|x| x < 25)
  .fold(0, |sum, x| sum + x) // eller bare .sum()
  // osv.
```

Cargo

Rustup installerer Cargo, et multiverktøy med alt du trenger til Rust. Cargo kan gjøre bl.a. dette

- Lage nye prosjekter: `cargo new navn`
- Hente dependencies, kompilere og kjøre koden: `cargo run`
- Legge til dependencies: `cargo add navn-på-dependency`
- Kjøre tester: `cargo test`
- Publisere koden din som en pakke: `cargo publish`
- Kjøre benchmarks: `cargo bench`
- Fikse feil i koden din: `cargo fix`

Mye mer å lære, men her er noen tips på vei

- VS Code med rust-analyzer eller IntelliJ/Clion med Rust-pluginen
- Bok fra A til Z: <https://doc.rust-lang.org/book/>
- Dokumentasjon til standardbiblioteket: <https://std.rs>