

10.

Module, Crates und Cargo

Begriffe

- **Crate:**
 - Einheit, die der Compiler auf einmal verarbeitet („*compilation unit*“)
 - Besteht aus einem Modulbaum
 - Kann Binary- oder Library-Crate sein
 - Oft eine pro Projekt
- **Modul:**
 - Namensbereich für Funktionen, Typen, ...
 - Modulbaum kann über mehrere Dateien aufgeteilt werden
 - Bisher in allen Beispielen u. Aufgaben: ein Modul

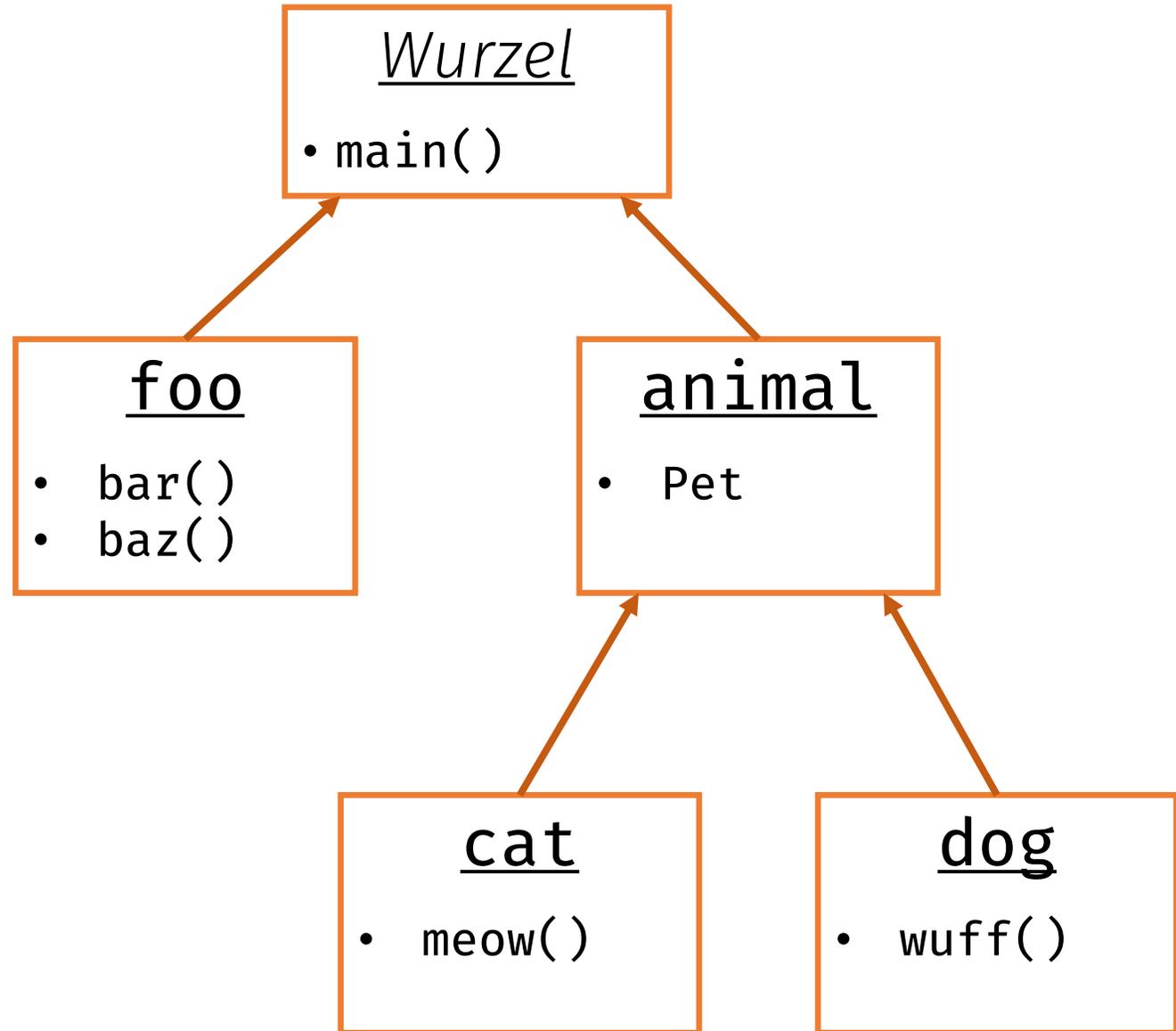
Module anlegen

```
fn main() {}

mod foo {
  fn bar() {}
  fn baz() {}
}

mod animal {
  struct Pet {}

  mod cat { fn meow() {} }
  mod dog { fn wuff() {} }
}
```



Symbole ansprechen: Pfade

```
fn main() { ... }
```

```
mod foo {  
  fn bar() {}  
  fn baz() {}  
}
```

```
mod animal {  
  struct Pet {}
```

```
  mod cat { fn meow() {} }
```

```
  mod dog { fn wuff() {} }
```

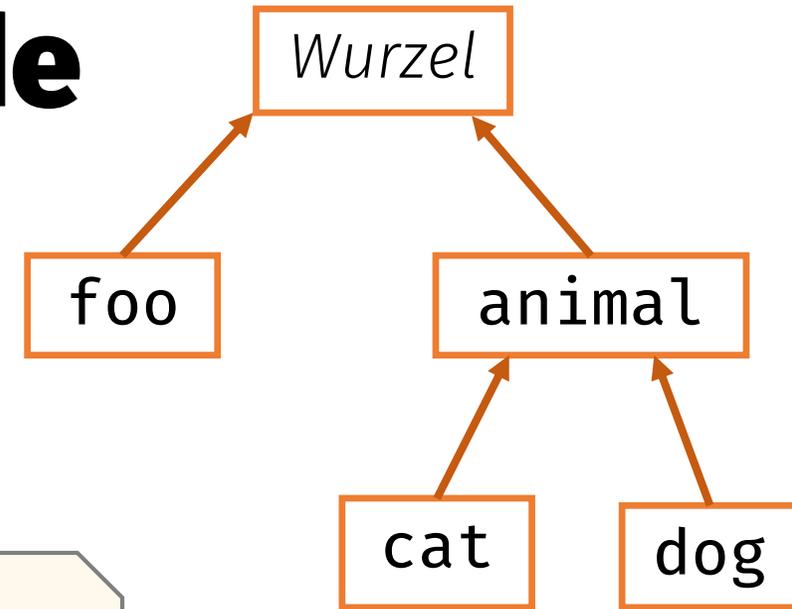
```
}
```

in main()

```
// call „baz()“  
foo::baz();
```

```
// call „wuff()“  
animal::dog::wuff();
```

```
// use type „Pet“  
let x = animal::Pet {};
```



- Pfadelemente mit „::“ getrennt
- Jedes Symbol hat vollständigen Pfad
- Unix-Dateisystem:
 - „::“ statt „/“

Symbole ansprechen: Pfade

```
fn main() {}
```

```
mod foo {  
  fn bar() {}  
  fn baz() {}  
}
```

```
mod animal {  
  struct Pet {}
```

```
  mod cat { fn meow() { ... } }
```

```
  mod dog { fn wuff() {} }
```

```
}
```

in meow()

```
// error! :-0
```

```
foo::baz();
```

```
// correct
```

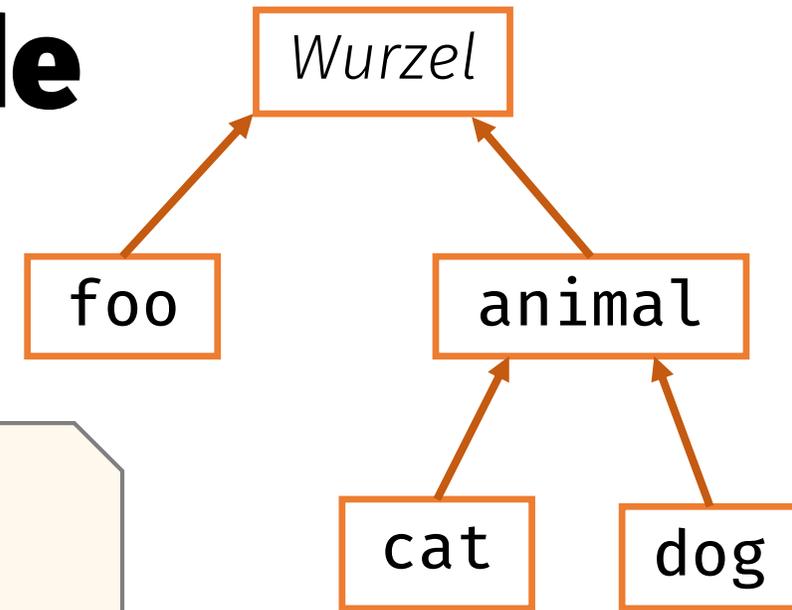
```
:::foo::baz();
```

```
// call „wuff()“
```

```
super::dog::wuff();
```

```
// use type „Pet“
```

```
let x = super::Pet {};
```



- Pfade sind relativ!
- Unix-Dateisystem:
 - „::“ statt „/“
 - „super“ statt „..“
 - „self“ statt „.“
- „super“ und „self“ nur am Anfang

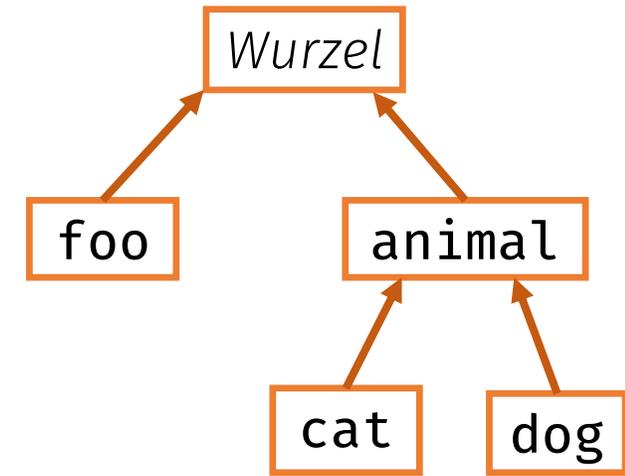
Abkürzen mit „use“

```
fn main() {  
    // call multiple times  
    animal::cat::meow();  
    animal::cat::meow();  
}
```

```
use animal::cat::meow;
```

```
fn main() {  
    // call multiple times  
    meow();  
    meow();  
}
```

```
fn main() {  
    // works, too  
    use animal::cat::meow;  
  
    // call multiple times  
    meow();  
    meow();  
}
```



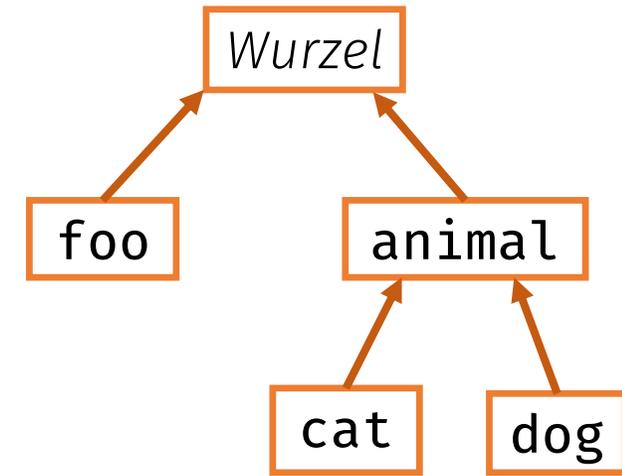
Abkürzen mit „use“

```
fn main() {  
    foo::bar();  
    foo::baz();  
}
```

```
use foo::{bar, baz};
```

```
fn main() {  
    bar();  
    baz();  
}
```

```
// This would work, too,  
// but should be avoided!  
use foo::*;
```



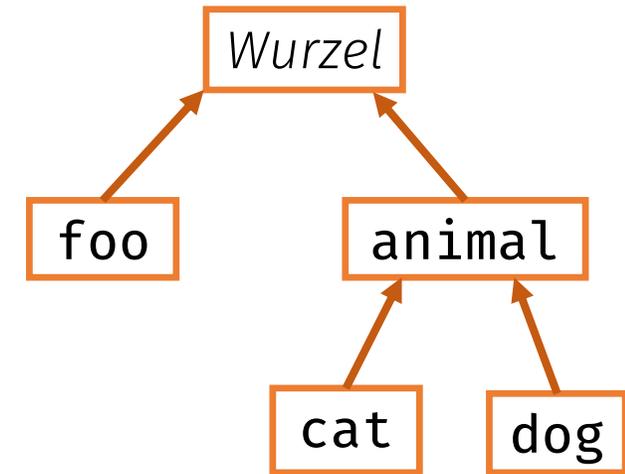
- Syntaxen:
 - Einfach: `use foo::bar::baz;`
 - Mehrere: `use foo::bar::{baz, bum};`
 - Alle pub: `use foo::bar::*;`
 - Sollte meist vermieden werden
- Liste oder Wildcard nur am Ende
- Kann in oder außerhalb von Funktion stehen

Abkürzen mit „use“

```
use animal::cat::meow;  
  
meow();
```

```
use animal::cat;  
  
cat::meow();
```

```
use animal;  
  
animal::cat::meow();
```



- Pfad kann auch *teilweise* ge-**use**-t werden
- Letzter Teil des **use**-Pfad es kann direkt angesprochen werden
- „**self**“ in `{ }` list:

```
use animal::cat::{self, meow};  
// is equivalent to:  
use animal::cat;  
use animal::cat::meow;
```

Zusammenfassung Pfade

Mit use

- Immer *absolut* (ausgehend vom Wurzel-Modul)
- Relativer Pfad mit „**self**“ und „**super**“ am Anfang

Bei Benutzung

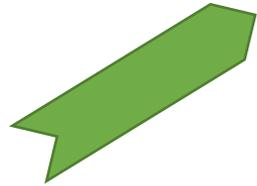
- Immer *relativ* zum aktuellen Modul
- Absoluter Pfad mit „**::**“ am Anfang

- „**use**“ verkürzt nur Namen von existierenden Symbolen!
- Mögliche Symbole: Funktionen, Typen, Module, ...

Modul in Datei auslagern

```
main.rs
fn main() {}

mod foo {
    fn bar() {}
}
```

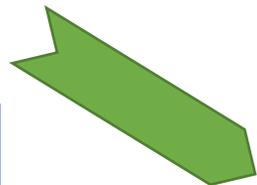


```
main.rs
fn main() {}

mod foo;
```

```
foo.rs
fn bar() {}
```

_____ oder _____



```
main.rs
fn main() {}

mod foo;
```

```
foo/mod.rs
fn bar() {}
```

mod foo {} Syntax fast nie benutzt! Fast immer eine Datei pro Modul!

- `mod <name>;`
- Compiler sucht:
 - `<name>.rs` oder
 - `<name>/mod.rs`

Von Modulbaum zu Dateien

1. Wurzel:

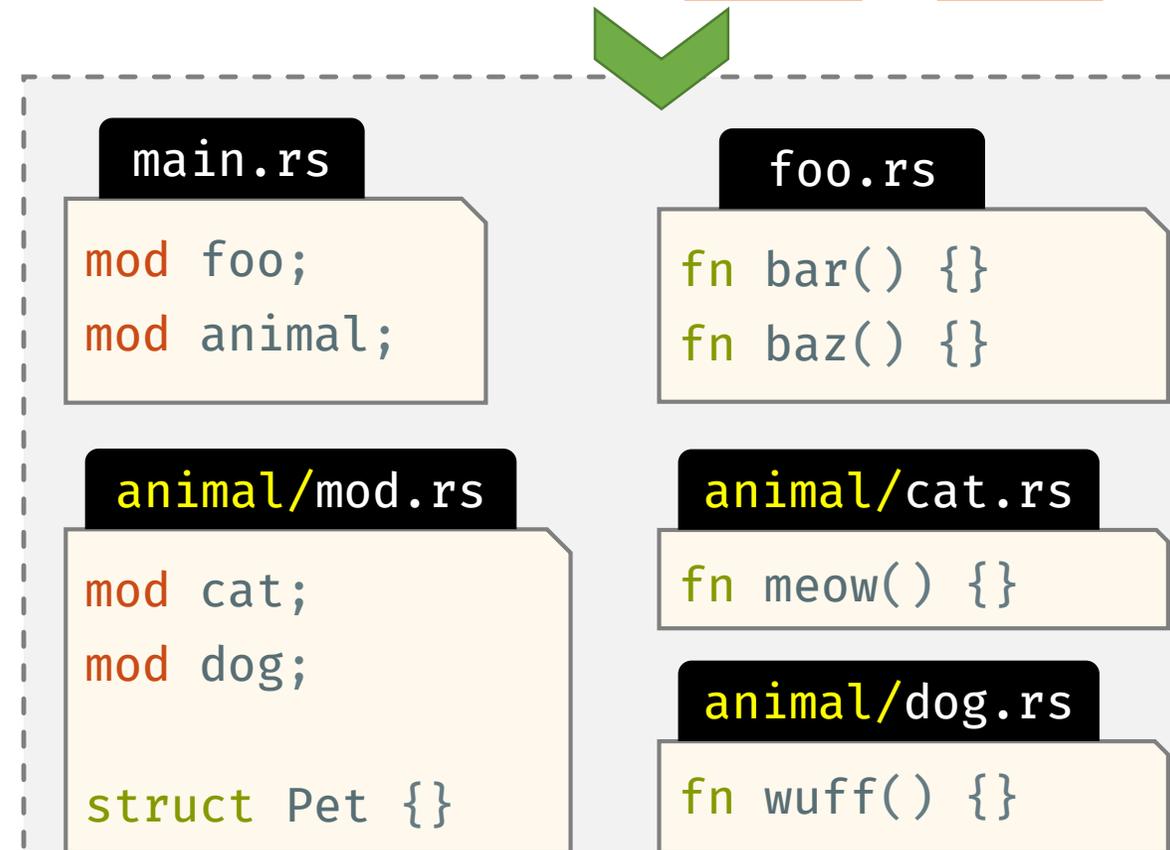
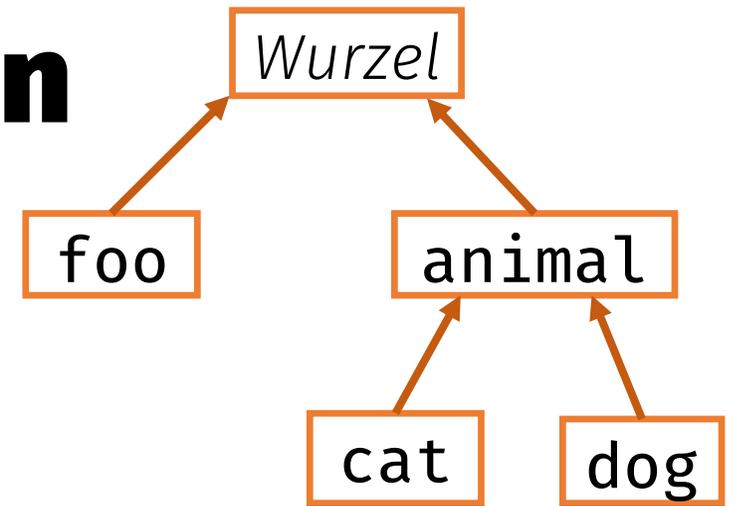
- `<crate_name>.rs` oder
- Oft benutzt: `main.rs` (binary) oder `lib.rs` (library)

2. Pro internem Knoten:

- Ordner „neben“ Vater
- `<module_name>/mod.rs`

3. Pro Blattknoten:

- `<module_name>.rs` im selben Ordner wie Vater
- („`<module_name>/mod.rs`“ auch erlaubt)*



Beispiel

main.rs

```
mod api;  
mod util;  
mod db;
```

api/mod.rs

```
mod users;  
mod posts;
```

util.rs

...

db/mod.rs

```
mod orm;  
mod sql;
```

api/users.rs

...

api/posts/mod.rs

```
mod search;  
mod text;
```

api/posts/search.rs

...

api/posts/text.rs

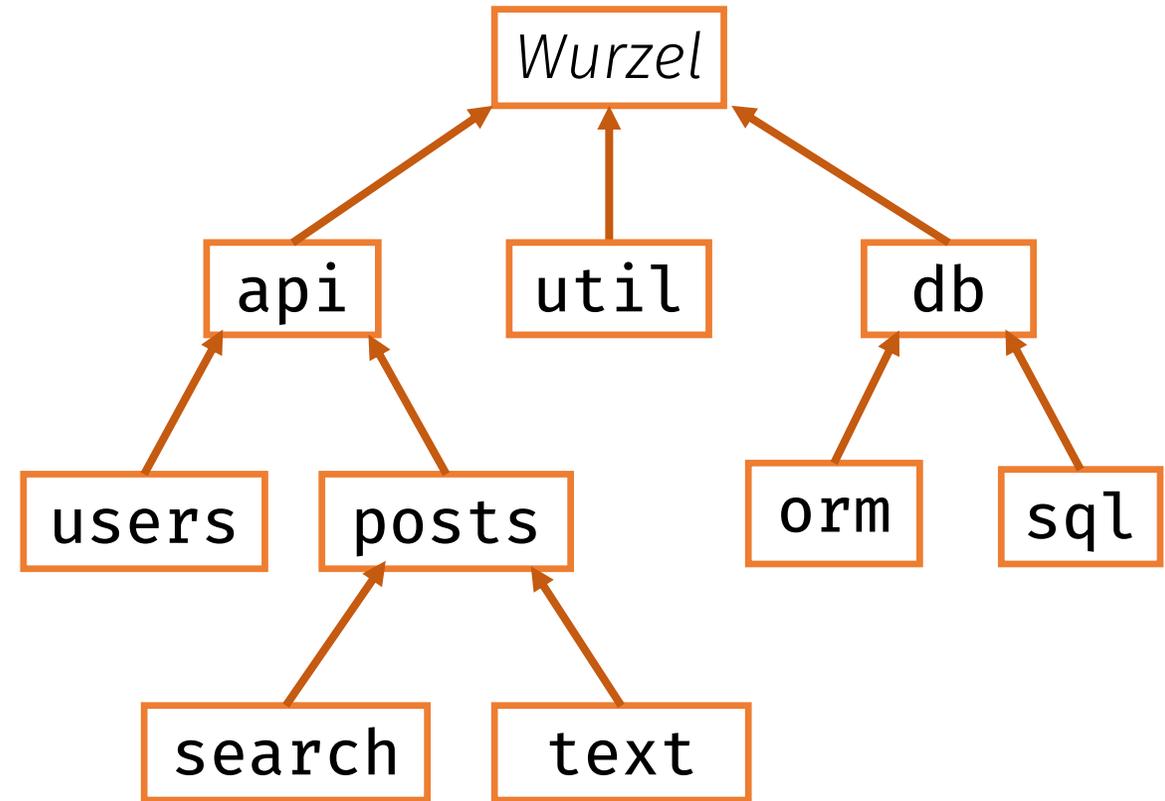
...

db/orm.rs

...

db/sql.rs

...



Zusammenfassung/Tipps

- *Erst* Modulbaum erstellen/entwerfen
 - Dateien durch Moduldeklaration einbinden
 - Jedes Modul wird *nur einmal* deklariert
 - Nur eine **mod**-Zeile für jedes Modul im ganzen Projekt!
 - **Keine** Zyklen!
- *Dann* Symbole mit Pfad ansprechen
 - Oft sinnvoll: Lange Pfade mit **use** verkürzen
 - Mehrere **use**-Zeilen pro Symbol sinnvoll
 - Zyklische „Referenzen“ **ok**
- Kompilieren?
 - Nur Wurzel an Compiler geben

```
$ rustc crate_root.rs
```

Sichtbarkeit

- Mit **pub** Modifier: public, also von überall benutzbar ([Beispiel](#))
 - Modifier für: `pub {fn, struct, enum, mod, use, type, extern crate}`
 - Aber auch: (Tuple-)Structfelder
- Sonst: „*module internal*“
 - Nur aktuelles Modul und Kinder können Symbol nutzen
 - Reexport mit **pub use** möglich (Pfad kann geändert werden):

main.rs

```
mod animal;  
fn main() {  
    animal::meow();  
}
```

animal/mod.rs

```
mod cat;  
  
pub use cat::meow;
```

animal/cat.rs

```
pub fn meow() {}
```

Jedes Element des Pfades muss zugänglich sein!

use EnumVariants

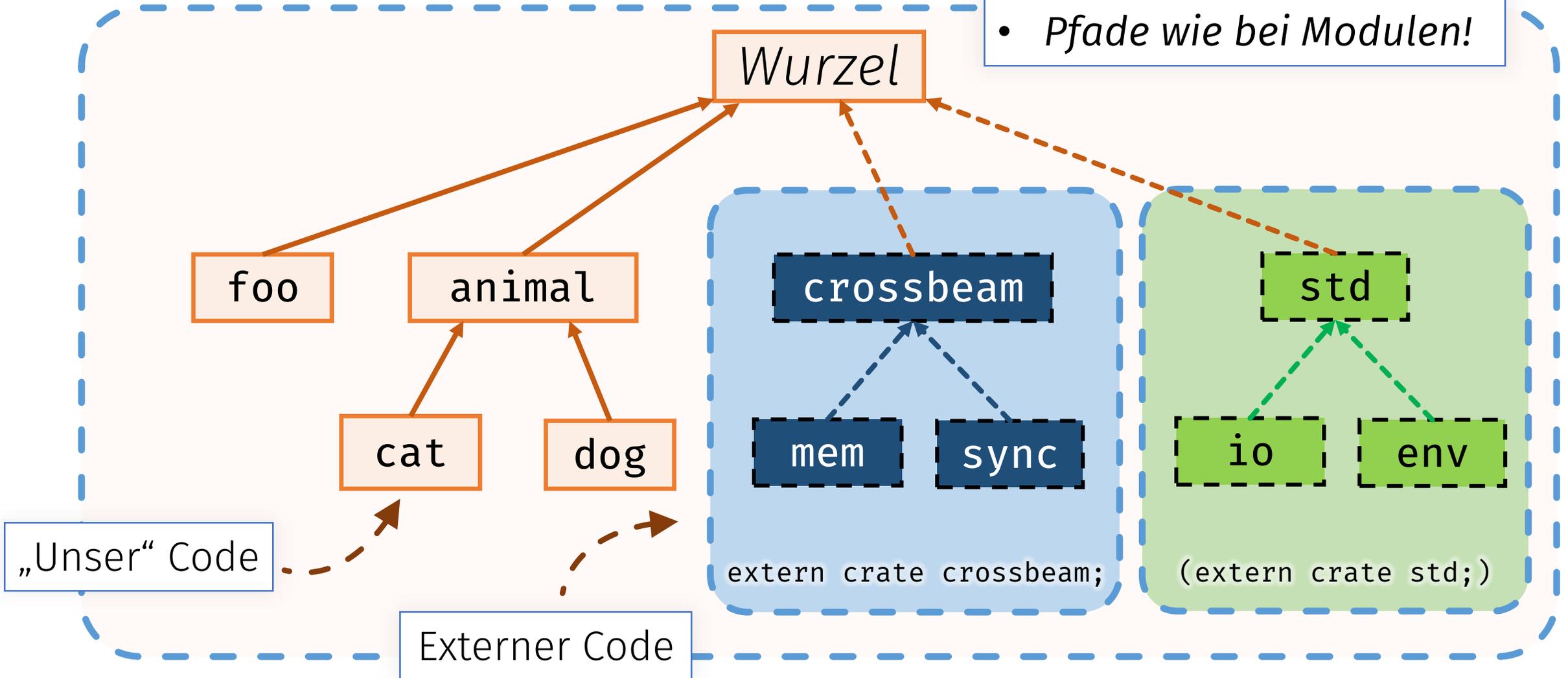
```
use Enum::*;
```

- Holt Variants in globalen Namespace
 - Können direkt angesprochen werden
- Eingeschränkt einsetzen!
 - Müllt Namensraum voll
 - Kollisionen wahrscheinlich
 - Sinnvoll in kleinen Bereichen (z.B. eine Funktion)
 - Sinnvoll für *sehr* oft benutzte Variants
 - **Some, None, Ok, Err**

```
enum Token {  
    Plus,  
    Minus,  
    Star,  
    ...  
}  
  
fn foo(t: Token) {  
    use Token::*;  
  
    match t {  
        Plus => ...,  
        Minus => ...,  
        Star => ...,  
        ...  
    }  
}
```

Extern Crates

- Externe Crates in Modulbaum eingehängt
- *Pfade wie bei Modulen!*



Extern Crates

```
extern crate foo;
```

- Hängt Modulbaum der Crate **foo** in aktuellen Modulbaum ein
- Vergleichbar mit „**mod foo;**“
 - Sucht aber nicht nach „**foo/mod.rs**“ oder „**foo.rs**“, sondern:
 - Compiler erwartet Pfad zu **.rlib** Datei per **--extern** Flag

foo.rs

```
pub fn hello() {  
    println!("hi");  
}
```

bar.rs

```
extern crate foo;  
  
fn main() {  
    foo::hello();  
}
```

```
$ rustc --crate-type lib foo.rs  
$ ls  
foo.rs  bar.rs  libfoo.rlib  
$ rustc --extern foo=libfoo.rlib bar.rs  
$ ./bar  
hi
```

Die Standardbibliothek

- Heißt „**std**“
- Wird immer automatisch eingebunden
- Hat *Prelude* ([Documentation](#))
 - Häufige Symbole, die automatisch ge-**use**-t werden

```
// The compiler secretly adds this  
// line to every1 crate-root  
extern crate std;
```

```
// The compiler secretly adds this  
// line to every1 module  
use std::prelude::v1::*;
```

¹ Man kann die Standardbibliothek deaktivieren

Cargo

- Buildtool & Package-Manager (offiziell)
- Cargo-Projekt
 - **Cargo.toml** → Konfiguration und Dependencies
 - **src/** → Der eigentliche Code
- Befehle:
 - **cargo new foo** (neues Library Projekt)
 - **cargo new --bin foo** (neues Binary Projekt)
 - **cargo build** (alles kompilieren) ←
 - **cargo run** (alles kompilieren und ausführen) ←

```
$ cargo new --bin foo
$ cd foo
$ ls
Cargo.toml      src/
$ ls src/
main.rs
```

Alle erzeugten Dateien
in **target/**
(und **Cargo.lock**)

(komplette Doku: <http://doc.crates.io/>)

Dependencies und crates.io

Cargo.toml

```
[package]
name = "foo"
version = "0.1.0"
authors = ["Peter"]

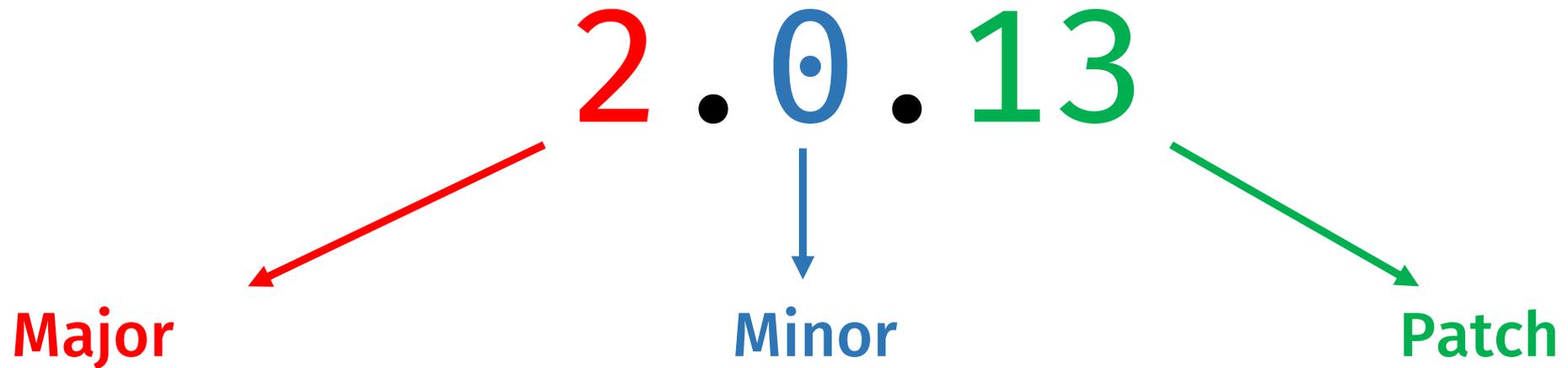
[dependencies]
time = "0.1"
```



- Offizieller Crate-Host: <https://crates.io>
- **cargo build**:
 - Lädt alle Dependencies runter
 - Kompiliert Dependencies
 - Gibt Dependencies per **--extern** Flag weiter

```
$ cargo build
  Updating registry `https://github.com/rust-lang/crates.io-index`
  Downloading ...
  Compiling libc v0.2.17
  Compiling winapi v0.2.8
  Compiling winapi-build v0.1.1
  Compiling kernel32-sys v0.2.2
  Compiling time v0.1.35
  Compiling foo v0.1.0 (file:///tmp/foo)
  Finished debug [unoptimized + debuginfo] target(s) in 34.15 secs
```

Semantic Versioning



- Erhöhen bei rückwärts-*in*kompatiblen Änderungen

- Erhöhen bei rückwärts-kompatiblen neuen Funktionen

- Erhöhen bei rückwärts-kompatiblen Bugfixes

Sonderregel: (nur bei Rust)

Wenn erste Stelle 0, dann:

- zweite Stelle „Major“
- dritte Stelle „Minor“ & „Patch“

Version in Cargo.toml

Meist einfach ohne Operator!

[Weitere Infos](#)

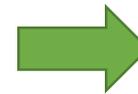
- Verschiedene Operatoren & Version **V**
 - Ohne Operator gleicht **^** Operator

- *Caret Requirements*: **V** oder kompatible, neuere Version

```
time = "1.2.3"
```

=

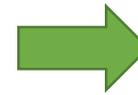
```
time = "^1.2.3"
```



```
>= 1.2.3 && < 2.0.0
```

- *Exact Requirement*: Nur genau **V**

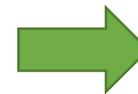
```
time = "=1.2.3"
```



```
== 1.2.3
```

- *Tilde Requirement*: **V** oder eingeschränkt neuere Version

```
time = "~1.2.3"
```



```
>= 1.2.3 && < 1.3.0
```

Weitere Cargo Befehle

- Alle Tests ausführen: `cargo test`
- Beispiel ausführen: `cargo run --example peter`
 - Beispiele sind einzelne Dateien in `examples/`
- Temporäre Dateien löschen: `cargo clean`
- Dependency Versionen aktualisieren: `cargo update`

Programmoptimierung:

- Deutlich schnellere Ausführungsgeschwindigkeit
- Kompilieren dauert länger

```
$ cargo build --release  
$ cargo run --release  
$ rustc -O foo.rs
```

Tools im Vergleich

rustc

- Compiler
- „Der Kern“

- *Benutzung*
 - „Ein-Modul-Programm“
 - Kleine Tests
 - Von anderen Tools

cargo

- Buildtool
- Package-Manager
- Nutzt **rustc**

- *Benutzung*
 - Fast alle Rust-Projekte
 - Alles mit externen Dependencies

rustup

- Compiler-Version-Manager
- Werkzeug zum Cross-Kompilieren

- *Benutzung*
 - Zur Installation
 - Cross-Kompilierung

Compiler Releases

- Alle 6 Wochen neue „stable“ Version
- Nightly „jede Nacht“ neu vom **master**-Branch
 - Bietet *unstable* Features
 - Geht öfters mal kaputt...

	stable	beta	nightly
Aktuell	1.13.0	1.14	1.15
Ab 22. Dez	1.14.0	1.15	1.16
Ab 02. Feb	1.15.0	1.16	1.17