10.

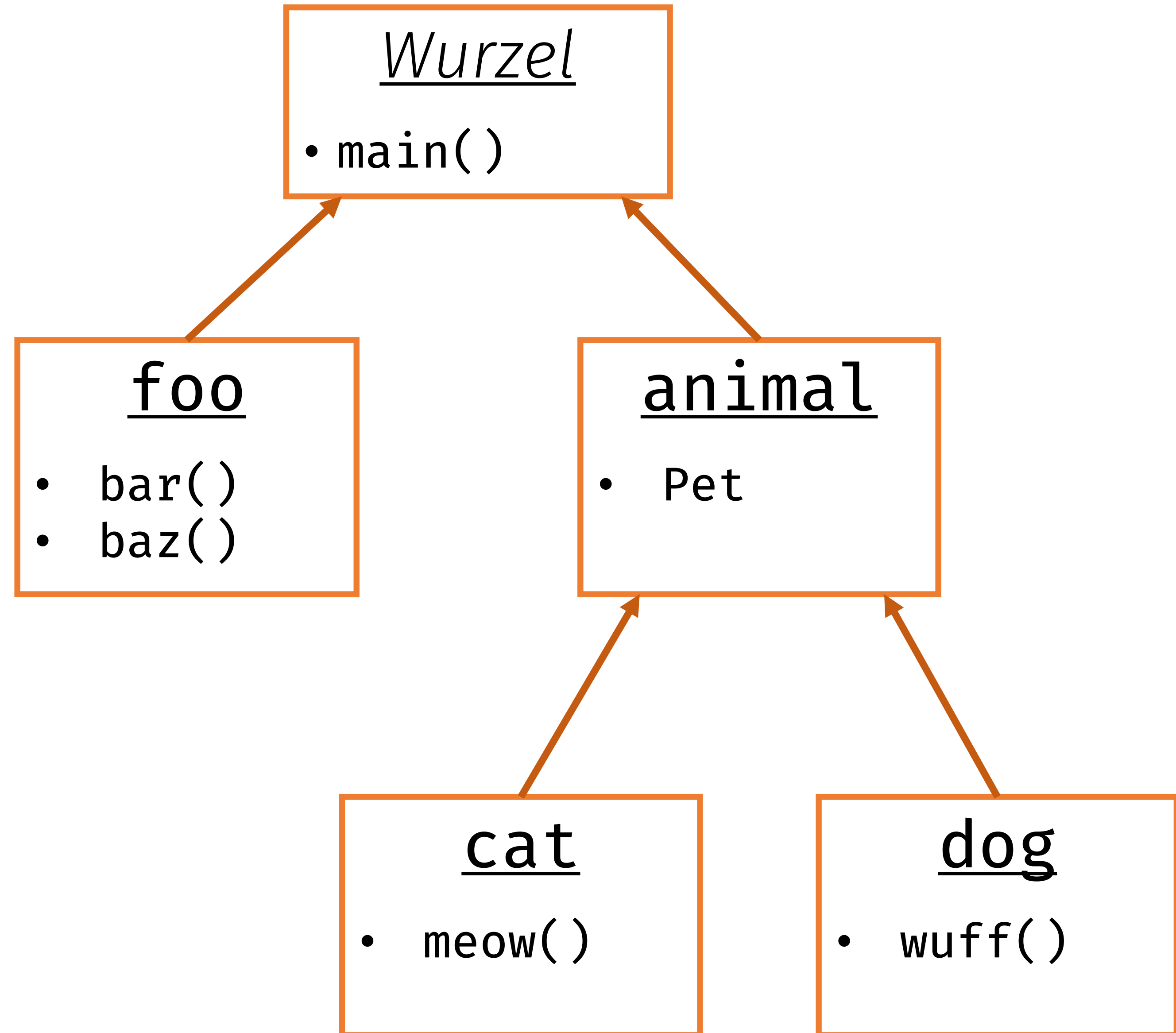
Module, Crates und Cargo

Begriffe

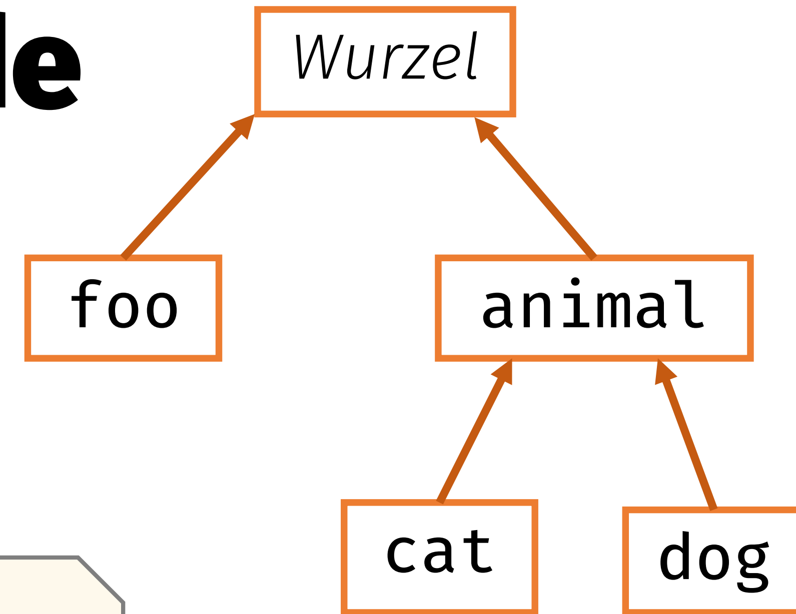
- *Crate*:
  - Einheit, die der Compiler auf einmal verarbeitet („*compilation unit*")
  - Besteht aus einem Modulbaum
  - Kann Binary- oder Library-Crate sein
  - Oft eine pro Projekt
- *Modul*:
  - Namensbereich für Funktionen, Typen, …
  - Modulbaum kann über mehrere Dateien aufgeteilt werden
  - 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




```rust
fn main() { … }

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

mod animal {
    struct Pet {}

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

in main()

```rust
// call „baz()“
foo::baz();

// call „wuff()“
animal::dog::wuff();

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

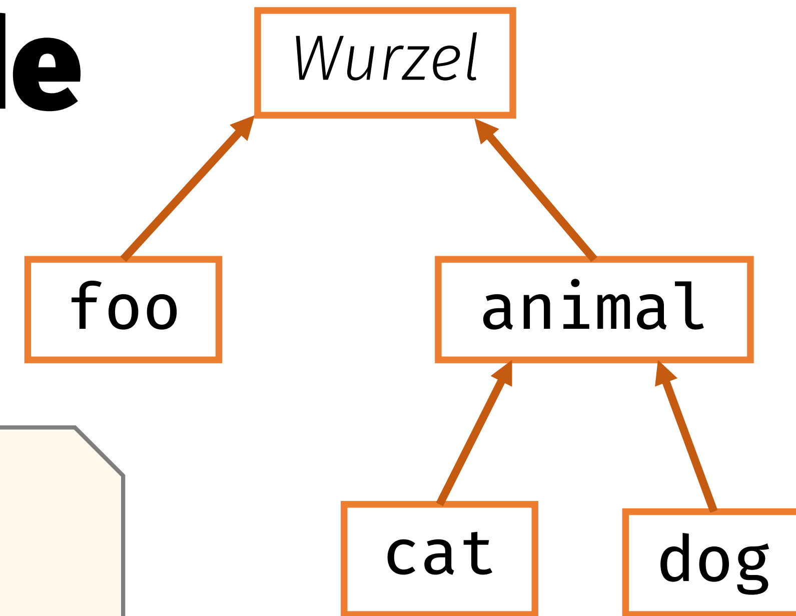
- Pfadelemente mit „::“ getrennt
- Jedes Symbol hat vollständigen Pfad
- Unix-Dateisystem:
  - „::“ statt „/“

Symbole ansprechen: Pfade

```
fn main() {}

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

mod animal {
    struct Pet {}

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

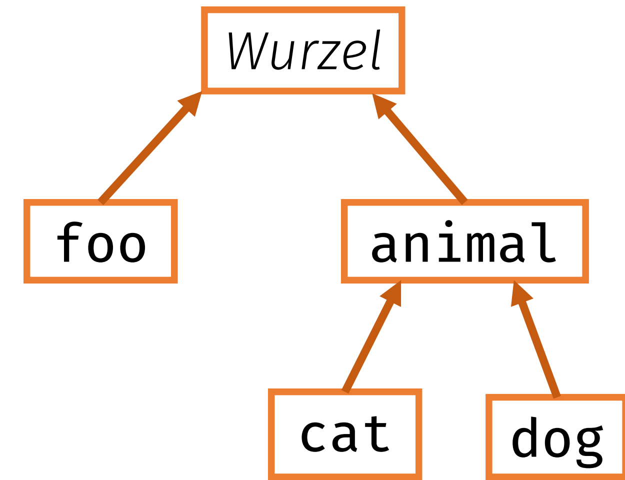
in meow()

```
// error! :-O
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“




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

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

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

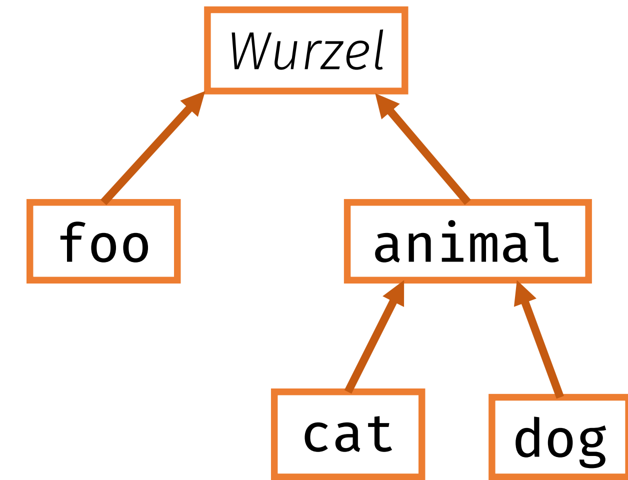
```rust
fn main() {
    // works, too
    use animal::cat::meow;

    // call multiple times
    meow();
    meow();
}
```

Abkürzen mit „use“

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

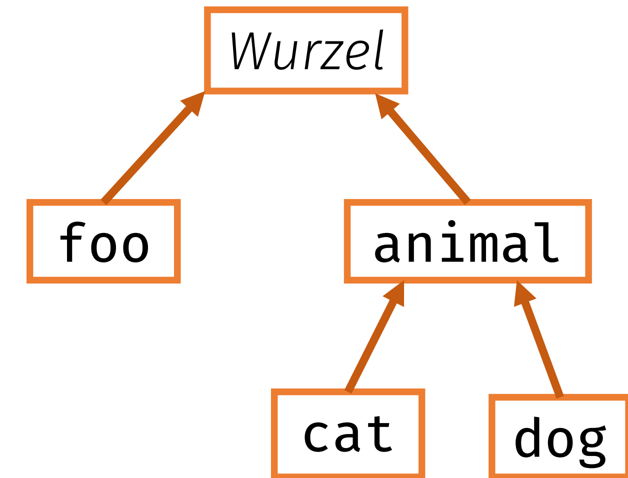
```rust
use foo::{bar, baz};

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

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



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

Abkürzen mit „use“




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

meow();
```

```rust
use animal::cat;

cat::meow();
```

```rust
use animal;

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

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

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

Zusammenfassung Pfade

Mit `use`	Bei Benutzung
• Immer ***absolut*** (ausgehend vom Wurzel-Modul) • Relativer Pfad mit „`self`" und „`super`" am Anfang	• 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

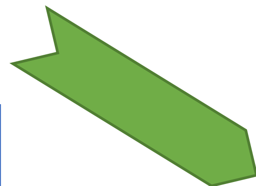
main.rs
```rust
fn main() {}

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

- mod ⟨name⟩;
- Compiler sucht:
  - ⟨name⟩.rs *oder*
  - ⟨name⟩/mod.rs

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

mod foo;
```

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

oder

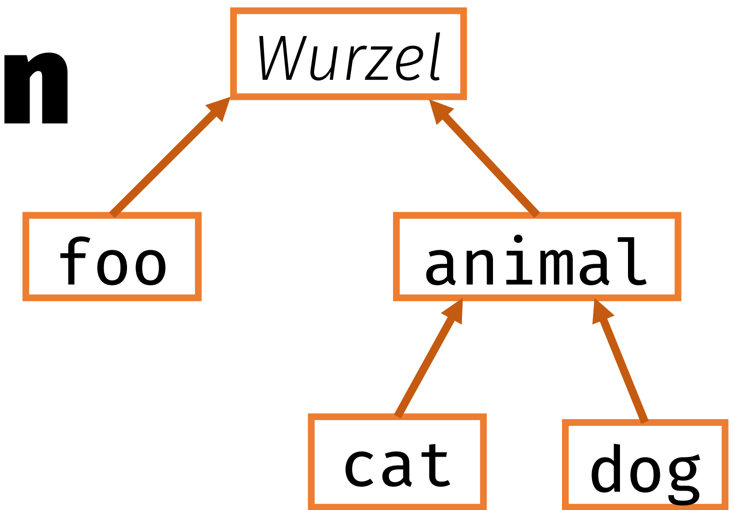


main.rs
```rust
fn main() {}

mod foo;
```

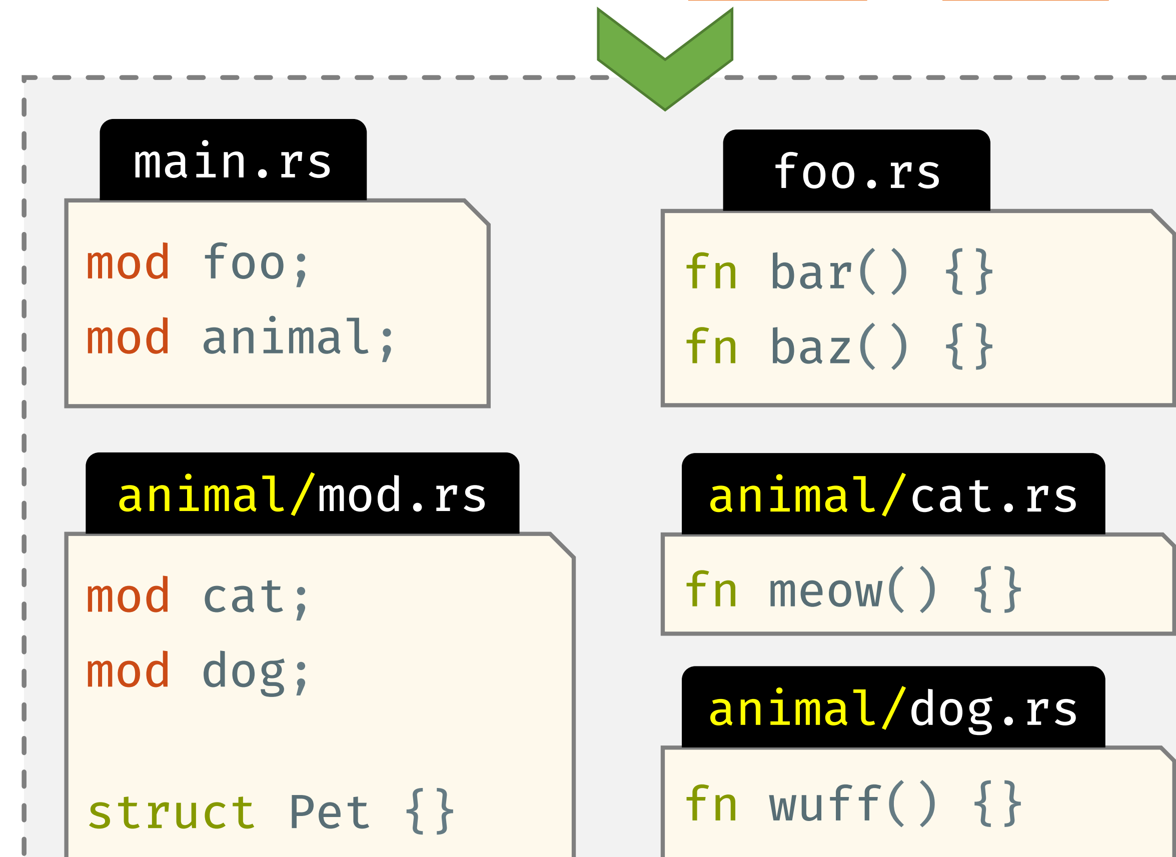
foo/mod.rs
```rust
fn bar() {}
```

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

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)*



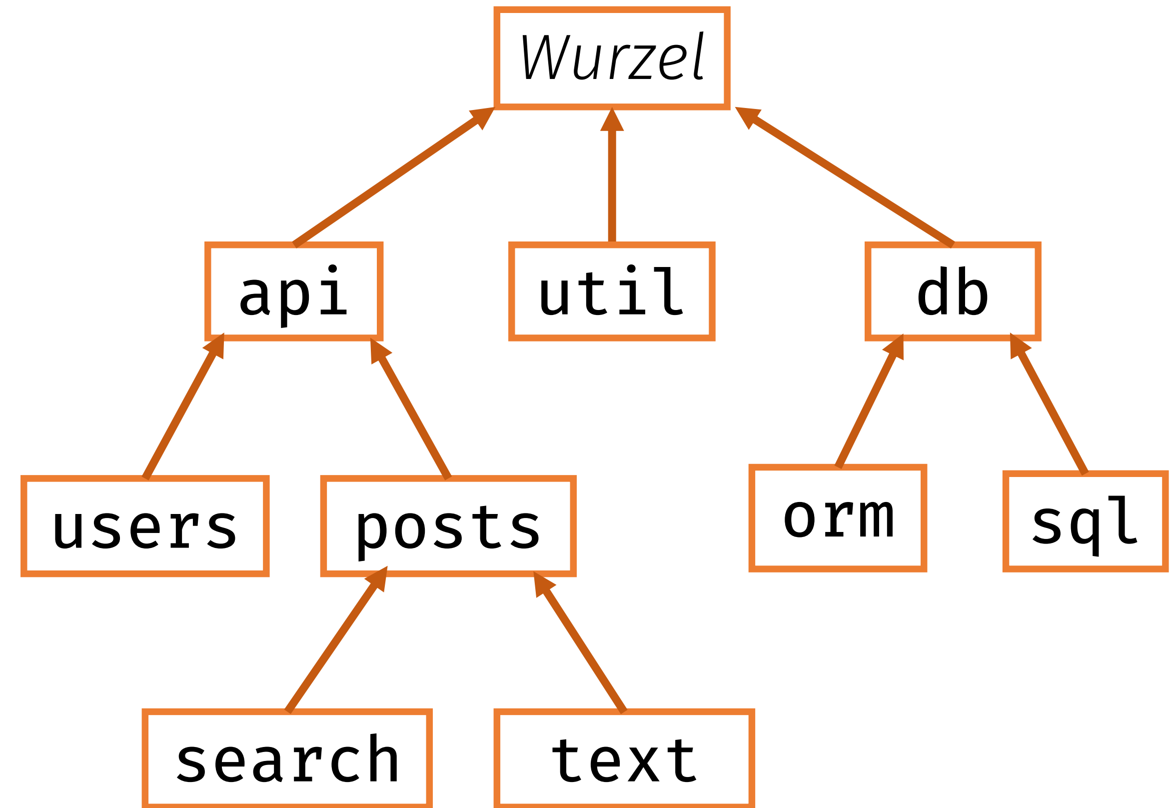
Beispiel

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!