

# 9.

---

## Error Handling

---

Literatur zu diesem Kapitel:

<http://joeduffyblog.com/2016/02/07/the-error-model>

# Erstmal: Contracts

```
impl Point {  
    /// Normalizes the vector.  
    ///  
    /// Input vector must not be the null vector.  
    /// The resulting vector has the length 1.  
    fn normalize(&mut self) { ... }  
}
```

## Function Contracts:

- **Preconditions:** Müssen wahr sein, damit Funktion ordentlich arbeiten kann
- **Postcondition:** Sind nach der Funktion wahr

## Andere Beispiele:

- Binäre Suche
- `sqrt(f64) -> f64`

## Keine Preconditions:

- `print(&str)`

Contracts sind *nicht* Rust-spezifisch!

# Arten von Fehlern

null Dereferenzierung

Konfigurationsdatei  
hat ungültiges  
Format

Array Out Of  
Bounds Indizierung

Datei nicht  
gefunden

Contract  
Violation

Server  
antwortet nicht

Out Of Memory

Teilen durch 0

Key in HashMap  
nicht gefunden

Nutzer hat Buchstaben als  
Telefonnummer eingegeben

# Arten von Fehlern

null Dereferenzierung

Konfigurationsdatei  
hat ungültiges  
Format

Array Out Of  
Bounds Indizierung

Datei nicht  
gefunden

Contract  
Violation

Server  
antwortet nicht

Out Of Memory

Teilen durch 0

Key in HashMap  
nicht gefunden

Nutzer hat Buchstaben als  
Telefonnummer eingegeben

# Arten von Fehlern

null Dereferenzierung

Konfigurationsdatei  
hat ungültiges  
Format

Array Index Out  
Of Bounds

Datei nicht  
gefunden

Contract  
Violation

Server  
antwortet nicht

Out Of Memory

Teilen durch 0

Key in HashMap  
nicht gefunden

Nutzer hat Buchstaben als  
Telefonnummer eingegeben

# Arten von Fehlern

## Recoverable Errors

Datei nicht gefunden  
Ungültige Nutzereingabe  
Key in HashMap nicht vorhanden  
Server antwortet nicht  
Ungültiges Format

- Erwartet
- **Grund:** Ungültiger Zustand der Welt
- Können behandelt werden

`null` Dereferenzierung  
Teilen durch 0  
Array Index Out of Bounds  
Out of Memory  
(Contract Violation)

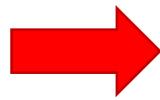
## Bugs

- Unerwartet
- **Grund:** Ungültiges Programm/Bug
- Können i.d.R. nicht behandelt werden!

# Bugs

- *Unerwartet* und meist *nicht behandelbar*!
- Status des Programms unberechenbar
  - Programm kann offensichtlich nicht mit der Situation umgehen
- **Lösung:** „Abandonment“ (Abbruch)
  - Alles abbrechen, was schon „infiziert“ sein könnte (Prozess, Thread)
- In Rust: **panic!()** → Bricht Thread ab

```
panic!();
```



```
thread 'main' panicked at 'explicit panic', panic.rs:2  
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

# Panic

```
// works like `println!()` : can print  
// additional information  
panic!("given number is negative: '{}'", n);
```

```
let arr = [1, 2];  
arr[2];
```



```
$ rustc panic.rs  
warning: this expression will panic at run-time  
--> panic.rs:3:5  
|  
3 |     arr[2];  
|     ^^^^^^ index out of bounds: the len is 2 but the index is 2  
  
$ ./panic  
thread 'main' panicked at 'index out of bounds: the len is 2 but the index is 2', panic.rs:3  
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

# Panic mit Backtrace

```
let arr = [1, 2];  
arr[2];
```



```
$ RUST_BACKTRACE=1 ./panic  
thread 'main' panicked at 'index out of bounds: the len is 2 but the index is 2', panic.rs:3  
stack backtrace:  
 1:      0x7fbce0c09aef - std::sys::backtrace::tracing::imp::write::h6f1d53a70916b90d  
 2:      0x7fbce0c0c59d - std::panicking::default_hook::{{closure}}::h137e876f7d3b5850  
 3:      0x7fbce0c0bafa - std::panicking::default_hook::h0ac3811ec7cee78c  
 4:      0x7fbce0c0c048 - std::panicking::rust_panic_with_hook::hc303199e04562edf  
 5:      0x7fbce0c0bee2 - std::panicking::begin_panic::h6ed03353807cf54d  
 6:      0x7fbce0c0be20 - std::panicking::begin_panic_fmt::hc321cece241bb2f5  
 7:      0x7fbce0c0bda1 - rust_begin_unwind  
 8:      0x7fbce0c4100f - core::panicking::panic_fmt::h27224b181f9f037f  
 9:      0x7fbce0c40fb3 - core::panicking::panic_bounds_check::h19e9bbc59320a57e  
10:     0x7fbce0c0572c - panic::main::hb3666e6eeba4db2a  
11:     0x7fbce0c14066 - __rust_maybe_catch_panic  
12:     0x7fbce0c0b371 - std::rt::lang_start::h538f8960e7644c80  
13:     0x7fbce0c05759 - main  
14:     0x7fbcdfe01f44 - __libc_start_main  
15:     0x7fbce0c055f8 - <unknown>  
16:           0x0 - <unknown>
```

Hilft manchmal bei  
der Fehlersuche!

# Panic und Unwinding

- Räumt vor beenden den Stack auf („Unwinding“)
  - Klettert Stack hoch (alle Funktionsaufrufe)
  - Dropt alle lokalen Objekte (≈ Destruktor, später mehr)
    - Gibt Speicher frei
    - Schließt Netzwerk-Sockets
    - Beendet Transaktionen
    - ...
- Kostet recht viel Zeit (nicht so wichtig ...)
- Kann deaktiviert werden (**panic=abort**)

# Wo treten panics auf?

- Indizierung von Arrays, **Vec**, ...
- Over-/Underflows
- • Als Platzhalter: **unimplemented!()**
- • **unreachable!()**
- • Asserts (auf contract violations prüfen)
- Deadlocks
- ...

```
impl Pokemon {  
    /// [...] level <= 100! [...]  
    fn with_level(lvl: u8) -> Self {  
        assert!(lvl <= 100);  
        // ...  
    }  
}
```

Contracts sind kein Sprachfeature...

# Recoverable Errors

- Andere Sprachen?
  - Java? → Checked Exceptions
  - C? → Error Codes als Rückgabewert, oft mit „Out Parametern“
- In Rust: Keine Exceptions, alles über Rückgabewerte
- **Result<T, E>** und **Option<T>**
- Sicherer als Error Codes
  - Fehler kann nicht ignoriert werden
  - Richtiges Ergebnis muss erst ausgepackt werden

C, nicht Rust

```
int read_file(void* buf) { ... }
```

```
char buf[100];  
read_file(&buf); // oops, ignored error
```

```
let mut file = File::open("a.txt");  
// error: `file` has the type  
// `Result<File, _>`  
file.write_all(&[0, 1, 2]);
```

# Beispiel

```
let mut file = File::open("a.txt");  
// error: `file` has the type  
// `Result<File, String>`  
file.write_all(&[0, 1, 2]);
```

```
let res = File::open("a.txt"); // : Result<File, String>  
match res {  
    Ok(file) => {  
        file.write_all(&[0, 1, 2]);  
    }  
    Err(e) => {  
        println!("not able to open file: {}", e);  
    }  
}
```

```
// this is *not* the real std impl!  
impl File {  
    fn open(name: &str)  
        -> Result<Self, String>  
    { ... }  
  
    fn write_all(&mut self, _: &[u8]) { ... }  
}
```

```
enum Result<T, E> {  
    Ok(T),  
    Err(E),  
}
```

# Was tun im Fehlerfall?

(Meist) nicht gut:

- Ignorieren
- Fehler printen
- panic

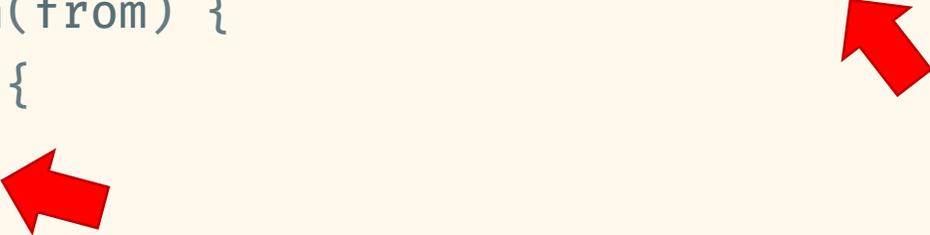
```
fn copy_file(from: &str, to: &str) {  
    match File::open(from) {  
        Ok(file) => {  
            // rest of the code is two levels indented :/  
        }  
        Err(e) => {  
            //  
            //  
            //  
            //  
            //  
            //  
        }  
    }  
}
```

→



# Besser: Fehler nach oben reichen

```
fn copy_file(from: &str, to: &str) -> Result<(), String> {  
    match File::open(from) {  
        Ok(file) => {  
            Ok(())  
        }  
        Err(e) => {  
            // not our problem ~\_(ツ)_/~  
            Err(e)  
        }  
    }  
}
```



Nur die obersten Funktionen  
kommunizieren Fehler zum  
Nutzer!

## Achtung:

Man muss Rückgabety  
verändern!

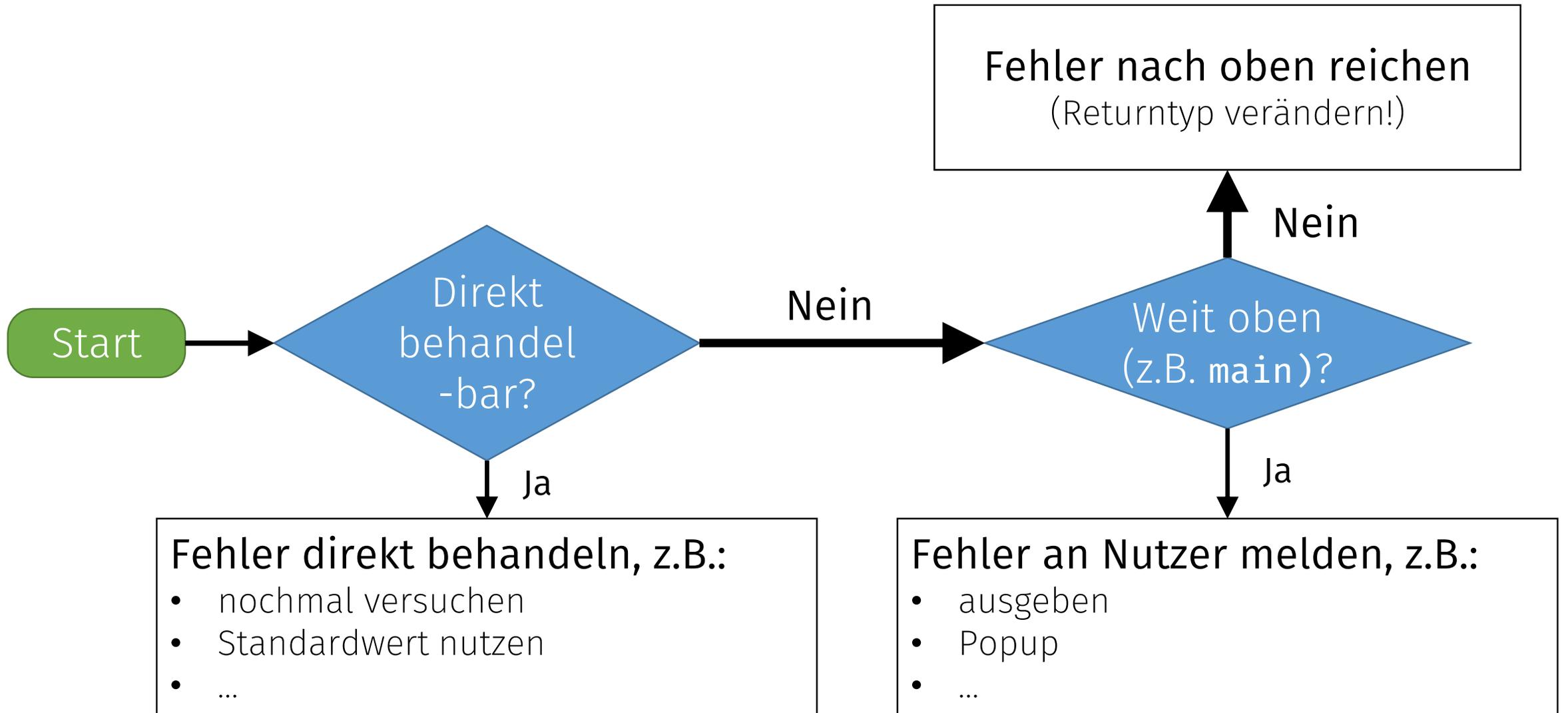
# Oder: Fehler direkt behandeln

```
/// Reads a valid `usize` integer from the terminal/user.  
fn read_usize() -> usize {  
    loop {  
        match read_string().parse() {  
            Ok(res) => return res,  
            Err(_) => println!("Please try again!"),  
        }  
    }  
}
```

*Hier: durch wiederholtes  
einlesen sicherstellen, dass  
Ergebnis gültig ist*

**Nicht immer möglich!**

# Guide: Wie Fehler behandeln?



# Einrückung vermeiden

```
fn copy_file(from: &str, to: &str) -> Result<(), String> {  
    match File::open(from) {  
        Ok(file) => {  
            // rest of the code is two levels indented :/  
            Ok()  
        }  
        Err(e) => {  
            // not our problem ~\_(\ツ)\_/~  
            Err(e)  
        }  
    }  
}
```



# Einrückung vermeiden

```
fn copy_file(from: &str, to: &str) -> Result<(), String> {  
    let from_file = match File::open(from) {  
        Ok(file) => file,  
        Err(e) => return Err(e),  
    };  
  
    let to_file = match File::open(to) {  
        Ok(file) => file,  
        Err(e) => return Err(e),  
    };  
  
    // the rest of the code here now :)  
    Ok(())  
}
```

# Welcher Error Typ?

- In den Beispielen: **String**
  - Oft nicht gut!
  - Unterscheidung zwischen „*file not found*“ und „*no permission*“?
- Besser:
  - Enum mit Fehlerfällen
  - Methoden für Ausgabe für Menschen
- Wenn Error Typ () → **Option<T>!**

```
enum ParseIntError {
    Empty,
    InvalidChar(char),
    Overflow,
}

impl ParseIntError {
    fn description(&self) -> String {
        match *self { ... }
    }
}

fn parse_int(s: &str)
    -> Result<i32, ParseIntError>
{ ... }
```

# Ein bisschen Typentheorie 2

Warum geht das eigentlich?

```
let file = match File::open(...) {  
  Ok(file) => file,  
  Err(e) => return Err(e),  
};
```

Beide Match-Arme müssen doch den gleichen Typ zurückgeben!

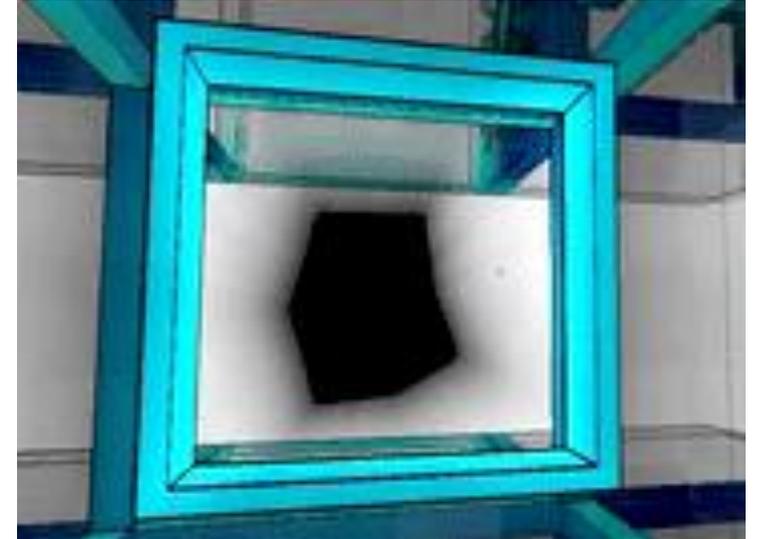
# Ein bisschen Typentheorie 2

- Mächtigkeit der Menge und benötigte Bits:
  - `u8` → 256 Elemente, 8 Bits
  - `bool` → 2 Elemente, 1 Bit
  - `(bool, bool)` → 4 Elemente, 2 Bit
  - `(bool, bool, bool)` → 8 Elemente, 3 Bit
  - `()` → 1 Element, 0 Bit
- **void** hat ein Element, braucht  $\log_2(1) = 0$  Bits
- Typ mit 0 Elementen?
  - Nicht Struct
    - Wie Tupel Produkttyp
    - Neutrales Element bzgl. Multiplikation ist 1

```
struct EmptyPlease {}  
  
let not_empty_sorry = EmptyPlease {};
```

# Bottom Type

```
enum Empty {}  
  
// it's impossible to create an instance  
let x: Empty = /* how?! */;
```

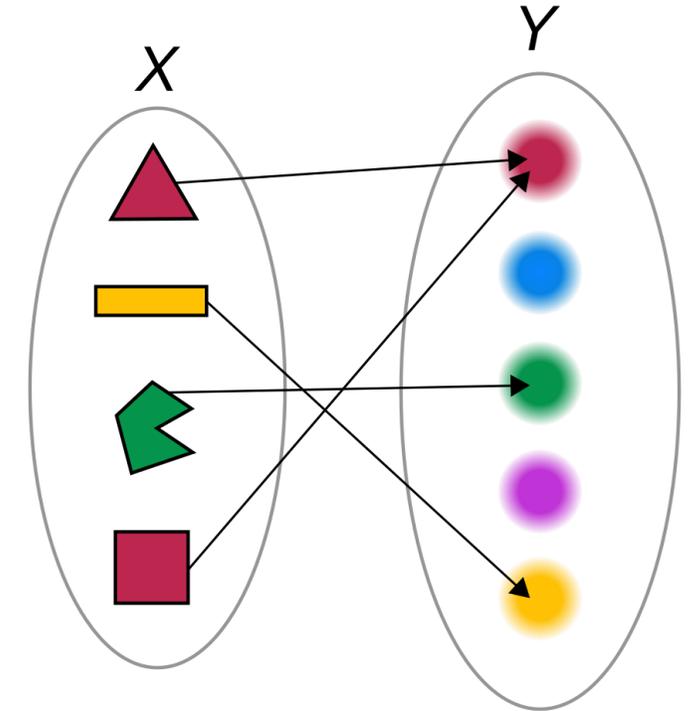


- Leerer Typ: meist „*Bottom*“ genannt (in Rust !)
- Instanz des Typen ist unmöglich!
- Rückgabebetyp von *divergierenden* Funktionen und Expressions
  - `std::process::exit(1)`
  - `panic!()`
  - Expressions: `break`, `return`, ...

```
fn exit(code: i32) -> !
```

# Konvertierung zwischen Typen

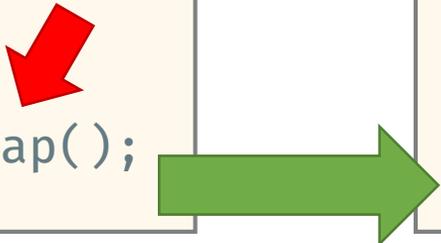
- Mapping von einer Menge auf die andere
- Konvertierung:
  - Ein „Pfeil“ für jedes Element in der Ursprungsmenge
- Konvertierung von *Bottom* trivial
  - Kein Pfeil benötigt



```
// valid Rust code!  
let x: String = panic!();
```

# Unwrapping

```
hash_map.insert("peter", 27);  
  
// I know, the key "peter" exists!  
let age = hash_map.get("peter").unwrap();
```



Besser:

```
// Use „entry API“  
let age = hash_map  
    .entry("peter")  
    .or_insert(27);
```

- `unwrap()` gibt Wert zurück, panic im Fall **None**
- Konzeptuell: Wandelt *Recoverable Error* in *Bug* um!

```
if some_opt.is_some() {  
    let value = some_opt.unwrap();  
}
```



```
if let Some(value) = some_opt {  
    // ...  
}
```

`unwrap()` kann fast immer geschickt vermieden werden!

# Besseres Unwrapping

```
// If the user doesn't specify a port,  
// we'll use 8080  
let port = config.port().unwrap_or(8080);
```

```
let age = hash_map  
    .get("peter")  
    .expect("faulty hashmap");
```

- **unwrap\_or()**: Für Fälle mit sinnvollem Default
  - *Achtung*: Nicht einfach aus Faulheit leere Instanzen nutzen!
- **expect()**: Wie **unwrap()** nur mit zusätzlichen Informationen

**unwrap()** nicht per se schlecht:

- In quick'n'dirty Code ok
- Um auszudrücken: „Dieser Error ist *nicht* recoverable!“

# Umwandlung Option/Result

```
// Throw away the error information  
let res = ...; // : Result<T, E>  
let opt = res.ok(); // : Option<T>
```

```
// Add information about the absence of a value  
let opt = ...; // : Option<T>  
let res = opt.ok_or(27); // : Result<T, i32>
```

(Eher selten notwendig)

# Das Ende?

- *Bisher*: recht viel Code für Fehlerbehandlung
- *Später*: Weitere Tricks, wie Fehlerbehandlung elegant wird
- *Aber*: Aus didaktischen Gründen jetzt noch nicht



# Tests

- `#[test]` Attribut für Unit Tests
  - Test „failed“, wenn Funktion panic't
  - Funktion wird nur mit `--test` kompiliert
- `assert!(expr)` und `assert_eq!(a, b)`
  - Panic wenn `expr` nicht `true`
  - Panic wenn `!(a == b)`
- Wenn panic erwartet: `#[should_panic]`

```
#[test]
fn small_primes() {
    assert!(is_prime(7));
}
```

```
$ rustc --test foo.rs
$ ./foo
```

```
#[test]
#[should_panic]
fn empty_vec_index() {
    let v = Vec::new();
    v[0]
}
```