Dr Dinu Dragan

PARALELNE I DISTRIBUIRANE ARHITEKTURE I JEZICI (ČAS 6)

ŠTA RADIMO DANAS?

- Rukovanje greškama
- Sanduci (Crates)
- Moduli (Modules)
- Polimorfizam u Rustu
- Traits
- Generics

RUKOVANJE GREŠKAMA

RUKOVANJE GREŠKAMA

- Postoje dva mehanizma za rukovanje greškama u Rustu:
  - Panic – je za vrstu grešaka koje ne bi trebale nikada se desi u normalnom režimu rada programa (**errors that should never happen**)
  - Results – je za obične greške koje mogu nastati u normalnom režimu rada programa usled neke kombinacije spoljašnjih faktora (loše ulazne vrednosti, prekid u radu računarske mreže, nedostatak prava pristupa, i sl.)
- U situacijama kada ne rukujete greškom, kompajler će vas na to upozoriti

PANIC

- Panic nastaje onda kada se stvari tako pokvare da su sigurno nastale usled baga u samom programu
  - Indeksiranje van opsega
  - Deljenje sa nulom
  - Pogrešno rukovanje greškom
  - Neuspeh assertion akcije
- Panic može da generiše i vaš kod primenom **Panic!** makroa koji se poziva iz kod ako se naiđe na neku neočekivanu situaciju
  - Moguće je formatirati i odgovarajuću poruku o grešci u pozivu samog makroa
- Ova vrsta greške uvek nastaje greškom programera i postoje dva načina za rukovanje greškom
  - Unwinding (standardni pristu)
  - Napuštanje procesa (abort the process)

UWINDING

- **Undwing the stak** je proces u kojem dolazi do čišćenja steka kada dođe od **Panic** greške
- Ovaj kod:

```
fn pirate_share(total: u64, crew_size: usize) -> u64 {
    let half = total / 2;
    half / crew_size as u64
}
```

- Će da izazove sledeću Panic grešku u slučaju da dođe do deljenja sa nulom:

```
thread 'main' panicked at 'attempt to divide by zero', pirates.rs:3780
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

- Ako se envijroment varijabla RUST_BACKTRACE postavi na 1, Rust će izlistati sadržaj steka

UWINDING

- Paralelno sa porukom o grešci i eventualnim izlistavanjem sadržaja steka, dolazi i do brisanja steka
- Sve privremene vrednosti, lokalne promenljive i argumenti prosleđeni funkciji (a koji se nalaze na steku) se brišu u obrnutom redosledu njihovog smeštanja na stek
- To podrazumeva i čišćenje za datim vrednostima, oslobađanje memorije, zatvaranje fajlova i sl., čak se poziva i **.drop()** metoda
- U primeru sa gusarima, nema šta da se briše
- Onog trenutka kada se obriše za pozvanom funkcijom, pristupa se delu stake funkcije koja je pozvala funkciju koja je izazvala paniku, pa se i njen sadržaj steka briše, i tako redom dok se ne vrati na vrh steka
- Na kraju nit prestaje da se izvršava
- Ako je panika izazvana u glavnoj niti izvršavanja, čitav proces prekida sa radom

UWINDING

- Čitav Panic je vrlo dobro definisan i izveden i nema ničega ni nalik panici, jedino što sve staje
- Panic mehanizam je siguran i ne krši ni jedno od Rust sigurnosnih pravila
- Neće ostati visećih pokazivača ili neinicijalizovane memorije
- Ideja je da Rust uhvati problem pre nego što se on desi i napravi ozbiljniju štetu
- Panic je na nivou niti izvršavanja
- Rust ima i mehanizam u kojem može da se uhvati čišćenje steka nakon napada panike, **std::panic::catch_unwind()**
- U određenim slučajevima tu se onda može pozvati kod za rukovanjem situacije i spašavanjem, što omogućuje da nastavi sa izvršavanjem niti

PREKID IZVRŠAVANJA (ABORTING)

- Do čišćenja steka ne dolazi u dva slučajeva
  - Ako **.drop()** metoda izazove novi napad panike za vreme čišćenje steka od aktivnog napada panike (znači, panika u panici) – ovo se smatra fatalnim i sve prestaje
  - Rustovo ponašenje se može podesiti – ako se kompajlira sa -C panic=abort, prvi napad panike u programu će dovesti do prekida njegovog izvršavanja
- Nema previše priče o mehanizmu panike, jer Rust kod nema obavezu da rukuje napadima panike (**no obligation to handle panic**)

RESULT

- Rust ne poseduje mehanizma izuzetaka (**exception**)
- Umesto toga, funkcije čiji poziv iz nekog razloga može rezultovati greškom (iz nekog razloga funkcija fejluje), imaju kao povratnu vrednost tip koji ukazuje na moguću grešku

```
fn get_weather(location: LatLng) -> Result<WeatherReport, io::Error>
```

- Rezultat ukazuje na moguću grešku
- Ako je poziv bio uspešan i normalno je realizovan, onda je povratna vrednost **Ok(weather)** gde je **weather** nova instanca tipa **WeatherReport**
- Ako je poziv bio neuspešan i rezultovao greškom, onda je povratna vrednost **Err(error_value)** gde je **error_value** nova instanca tipa **io::Error** koja sadrži objašnjenje o vrsti nastale greške
- Rust očekuje da napišemo neki kod za rukovanjem greškom kad god pozovemo funkcijom

HVATANJE GREŠAKA

- Sledeći pristup je kompletan pristup u rukovanju greškama:

```
match get_weather(hometown) {
    Ok(report) => {
        display_weather(hometown, &report);
    }
    Err(err) => {
        println!("error querying the weather: {}", err);
        schedule_weather_retry();
    }
}
```

- Ovo je Rustov mehanizam koji odgovara try/catch mehanizmu u drugim jezicima
- Ovako izgleda kada vaš kod rukuje greškom, a ne kada se želi proslediti dalje pozivaocu funkcije

HVATANJE GREŠAKA

- Rukovanje greškom pomoću **match** selekcija može da bude predetaljno i nepotrebno
- Sam **Result<T, E>** je prilično kompleksan i sadrži razne metode za rad sa greškama i rezultatima
- Neke od tih metoda (najčešće korišćenih) su:
  - **result.is_ok(), result.is_err()** – vraćaju bool vrednost
  - **result.ok()** – vraća rezultat kao **Option<T>**, tako da ako je funkcija bila uspešna, rezultat će biti **Some(success_value)**, u suprotno, vraća **None**, ignorišući grešku
  - **result.err()** – isto kao prethodno, samo vraća rezultat kao **Option<E>**

HVATANJE GREŠAKA

- Neke od tih metoda (najčešće korišćenih) su:
  - **result.unwrap_or(fallback)** – vraća rezultat, ako je funkcija bila uspešna, ili neku predefinisanu vrednost (fallback), ignorišući grešku

```
// A fairly safe prediction for Southern California.
const THE_USUAL: WeatherReport = WeatherReport::Sunny(72);

// Get a real weather report, if possible.
// If not, fall back on the usual.
let report = get_weather(los_angeles).unwrap_or(THE_USUAL);
display_weather(los_angeles, &report);
```

  problem je ako nema predefinisane vrednosti za datu situaciju.

HVATANJE GREŠAKA

- Neke od tih metoda (najčešće korišćenih) su:
  - **result.unwrap_or_else(fallback_fn)** – isto kao prethodno, jedino ako je funkcija bila neuspešna, ne prosleđuje se predefinisana vrednost, već se poziva funkcija koja rukuje tom situacijom ili **closure** koji će vratiti predefinisanu vrednost

```
let report =
    get_weather(hometown)
    .unwrap_or_else(|_err| vague_prediction(hometown));
```

ovo je za situacije kada se ne isplati odrediti predefinisanu vrednost unapred.

HVATANJE GREŠAKA

- Neke od tih metoda (najčešće korišćenih) su:
  - **result.unwrap()** – vraća rezultat ako je funkcija uspela, ako ne dobija napad panike
  - **result.expect(message)** – isto kao i prethodno, jedino što u slučaju panike se prosleđuje i poruka **message**
  - **result.as_ref()** – konvertuje **Result<T, E>** u **Result<&T, &E>**
  - **result.as_mut()** – isto kao i prethodno jedino što se pozajmljuje mutabilna vrednost, rezultat je **Result<&mut T, &mut E>**

SINONIMI ZA RESULT TIP

- Često se umesto specifične definicije rezultata koristi neki od njegovih sinonima koji ne uključuju grešku
- To se koristi u situacijama kada se zna unapred koja je greška u pitanju, npr. sve metode u nekom modulu i sl.
- Primer aliasa:

```
pub type Result<T> = result::Result<T, Error>;
```

- Primer kasnije upotrebe aliasa:

```
fn remove_file(path: &Path) -> Result<()>
```

ŠTAMPANJE GREŠAKA

- Poruka o grešci se može ispisivati na ekran pomoću **println!** makroa

```
// result of `println!("error: {}", err);`
error: failed to lookup address information: No address associated with
hostname

// result of `println!("error: {:?}", err);`
error: Error { repr: Custom(Custom { kind: Other, error: StringError(
"failed to lookup address information: No address associated with
hostname") }) }
```

- Sa {:?} format specifikatorom dobija se debug verzija poruke o grešci
- Metode vezane za ispisivanje grešaka
  - **err.to_string()** – poruka o grešci pretvorena u **String**
  - **err.source()** – tipa **Option** ako postoji informacija o izvoru greške, u suprotnom **None**

ŠTAMPANJE GREŠAKA

- **println!** makro neće ispisati sve podatke iz poruke o grešci, tj. neće ispisati poruku iz **.source()** metode
- Ako se želi kontrolisati to što će se ispisivati, onda se koristi **writeln!** makro

```rust
use std::error::Error;
use std::io::{Write, stderr};

/// Dump an error message to `stderr`.
///
/// If another error happens while building the error message or
/// writing to `stderr`, it is ignored.
fn print_error(mut err: &dyn Error) {
    let _ = writeln!(stderr(), "error: {}", err);
    while let Some(source) = err.source() {
        let _ = writeln!(stderr(), "caused by: {}", source);
        err = source;
    }
}
```

PROPAGACIJE GREŠKE

- U funkcijama u kojima ne želimo da rukujemo greškom, možemo grešku da prosledimo dalje, pozivaocu
- Uslov za propagaciju je da i sama funkcija vraća tip Result
- Za propagaciju se koristi operator ?

```
let weather = get_weather(hometown)?;
```

- Ovaj operator se može koristiti nad bilo kojim izrazom koji rezultuje tipom Result
- Povratna vrednost operatora **?** zavisi od povratne vrednosti funkcije
  - Ako je poziv funkcije bio uspešan, u promenljivu se upisuje povratna vrednost
  - Ako je poziv funkcije bio neuspešan, odmah se prekida izvršavanje na tom nivou i rezultat se propagira dalje pozivaocu

PROPAGACIJE GREŠKE

- Alternativa za operator **?** bi bila sledeća:

```
let weather = match get_weather(hometown) {
    Ok(success_value) => success_value,
    Err(err) => return Err(err)
};
```

- Operator **?** olakšava život, naročito kad ima jako puno koda koji proveravaju grešku

```
use std::fs;
use std::io;
use std::path::Path;

fn move_all(src: &Path, dst: &Path) -> io::Result<()> {
    for entry_result in src.read_dir()? {  // opening dir could fail
        let entry = entry_result?;         // reading dir could fail
        let dst_file = dst.join(entry.file_name());
        fs::rename(entry.path(), dst_file)?;  // renaming could fail
    }
    Ok(())  // phew!
}
```

PROPAGACIJE GREŠKE

- Operator **?** se može koristiti na sličan način i sa tipom **Option**
- U funkcijama koji vraćaju **Option**, može se iskoristiti da ranije odmota neku promenljivu tipa **Option** i da izađe iz funkcije

```
let weather = get_weather(hometown).ok()?;
```

- Ovaj primer će dovesti do ranijeg izlaska iz funkcije, ako je poziv **get_weather** funkcije rezultovao **None** vrednošću

VIŠE GREŠAKA ODJEDNOM

- Nekada može da nastane više tipova grešaka u jednom pozivu

```rust
use std::io::{self, BufRead};

/// Read integers from a text file.
/// The file should have one number on each line.
fn read_numbers(file: &mut dyn BufRead) -> Result<Vec<i64>, io::Error> {
    let mut numbers = vec![];
    for line_result in file.lines() {
        let line = line_result?;          // reading lines can fail
        numbers.push(line.parse()?);      // parsing integers can fail
    }
    Ok(numbers)
}
```

- Čitanje int broja iz fajla

VIŠE GREŠAKA ODJEDNOM

- Rust kompajleru se to ne sviđa

```
error: `?` couldn't convert the error to `std::io::Error`

  numbers.push(line.parse()?);     // parsing integers can fail
                              ^
              the trait `std::convert::From<std::num::ParseIntError>`
              is not implemented for `std::io::Error`

note: the question mark operation (`?`) implicitly performs a conve
on the error value using the `From` trait
```

- Problem je što u jednoj liniji koda, prilikom parsiranja fajla za celim brojem mogu nastati dve greške
  - Tip **line_result** je **Result<String,std::io::Error>**
  - Tip **line.parse()** je **Result<i64, std::num::ParseIntError>**
- Rezultat fukcije je tipa **Result<Vec<i64>, io::Error>**, pa Rust proba da konvertuje **std::num:: ParseIntError** u **io::Error** što nije implementirano

VIŠE GREŠAKA ODJEDNOM

- Rešenje je uvođenje generičkih grešaka:
  Box<dyn std::error::Error + Send + Sync + 'static>
- Prvi deo predstavlja bilo koju grešku, a drugi deo je čini sigurnu za razmenu između niti

```
type GenericError = Box<dyn std::error::Error + Send + Sync + 'static>;
type GenericResult<T> = Result<T, GenericError>;
```

- Ovde se uvode sinonimi
- Naravno, sada mora da se promeni i povratna vrednost funkcije, ali operator **?** radi automatsku konverziju bilo koje greške u ovu generičku
- Eksplicitna konverzija iz bilo koje greške u GenericError se vrši preko funkcije **GenericError::from()**

```
let io_error = io::Error::new(          // make our own io::Error
    io::ErrorKind::Other, "timed out");
return Err(GenericError::from(io_error));  // manually convert to GenericError
```

VIŠE GREŠAKA ODJEDNOM

- Kako to izgleda kada neko poziva funkcije koje propagiraju više grešaka ka pozivaocu i pozivalac želi da rukuje sa tačno određenom greškom

```
loop {
    match compile_project() {
        Ok(()) => return Ok(()),
        Err(err) => {
            if let Some(mse) = err.downcast_ref::<MissingSemicolonError>() {
                insert_semicolon_in_source_code(mse.file(), mse.line())?;
                continue;  // try again!
            }
            return Err(err);
        }
    }
}
```

- Generička metoda **error.downcast_ref::<ErrorType>()** služi za pozajmljivanje reference na grešku, ako je to ona greška koja se traži

MAIN I GREŠKE

- Kako god, propagacija greške negde mora da završi, obično u **main**
- **main** ne može dalje da propagira, jer mu povratna vrednost nije Result

```
fn main() {
    calculate_tides()?;  // error: can't pass the buck any further
}
```

- Najlakše je rukovati greškom sa **.expect()** metodom

```
fn main() {
    calculate_tides().expect("error");  // the buck stops here
}
```

- U ovom konkretnom slučaju ako se naiđe na grešku **.expect()** metoda kreće da paniči, ispisuje poruku o grešci i izlazi se iz poruka sa kodom greške različite od nule

MAIN I GREŠKE

- Poruka posle **.expect()** metode je od prilike ovakva:

```
$ tidecalc --planet mercury
thread 'main' panicked at 'error: "moon not found"', src/main.rs:2:23
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

- Alternativa je da se promeni tip povratne vrednosti **main** funkcije tako da vraća **Result** tip

```
fn main() -> Result<(), TideCalcError> {
    let tides = calculate_tides()?;
    print_tides(tides);
    Ok(())
}
```

- Ovo radi za svaku grešku koja može da se odštampa sa **{:?}** formatom, koji važi za sve standardne greške, iz **std::io::Error**

```
$ tidecalc --planet mercury
Error: TideCalcError { error_type: NoMoon, message: "moon not found" }
```

MAIN I GREŠKE

- Pristup sa time da **main** prosledi grešku je nezgodan kad treba ispisivati malo komplikovanije poruke o grešci
- Kada je poruka kompleksnija, isplati se da **main** ispiše poruku o grešci

```
fn main() {
    if let Err(err) = calculate_tides() {
        print_error(&err);
        std::process::exit(1);
    }
}
```

- Tada se dobije sledeći ispis

```
$ tidecalc --planet mercury
error: moon not found
```

SOPSTVENE (CUSTOM) GREŠKE

- Sopstvene greške se najjednostavnije prave na sledeći način:

```rust
// json/src/error.rs

#[derive(Debug, Clone)]
pub struct JsonError {
    pub message: String,
    pub line: usize,
    pub column: usize,
}
```

- U pitanju je greška za sopstvenu implementaciju JSON parsera, gde će se greška zvati **json::error::JsonError**, a aktivira se

```rust
return Err(JsonError {
    message: "expected ']' at end of array".to_string(),
    line: current_line,
    column: current_column
});
```

SOPSTVENE (CUSTOM) GREŠKE

- Da bi sopstvena greška radila na očekivani način, neophodno je implementirati i metodu za štampanje poruke o grešci kroz **impl** mehanizam

```
use std::fmt;

// Errors should be printable.
impl fmt::Display for JsonError {
    fn fmt(&self, f: &mut fmt::Formatter) -> Result<(), fmt::Error> {
        write!(f, "{} ({}:{})", self.message, self.line, self.column)
    }
}

// Errors should implement the std::error::Error trait,
// but the default definitions for the Error methods are fine.
impl std::error::Error for JsonError { }
```

- Pošto je ovo vrlo uobičajeno, postoji nekoliko paketa u Cargo-u za to, jedan od njih je **thiserror**

SOPSTVENE (CUSTOM) GREŠKE

- **thiserror** radi na sledeći način:

```
use thiserror::Error;
#[derive(Error, Debug)]
#[error("{message:} ({line:}, {column})")]
pub struct JsonError {
    message: String,
    line: usize,
    column: usize,
}
```

- **#[derive(Error)]** direktiva kaže **thiserror** paketu da generiše sav kod pokazan prethodno

PREDNOSTI ARHITEKTURE SA RESULT

- Rust zahteva od programera da donese neku vrstu odluke i zapiše je u kodu, u svakoj tački gde bi moglo da dođe do greške
  - Ovo je dobro jer je u suprotnom lako zanemariti pogrešno rukovanje greškama
- Najčešća odluka je da se dozvoli propagacija grešaka, a to se piše jednim znakom, ?
  - Dakle, otklanjanje grešaka ne zatrpava vaš kod na način na koji se to radi u drugim programskim jezicima (C i Go), ali je mogućnost pojavljivanja greške uvek vidljivo: možete pogledati deo koda i na prvi pogled videti sva mesta na kojima se šire greške
- Pošto je mogućnost greške iskazana kroz tip povratne vrednosti funkcije, jasno je koje funkcije mogu da dovedu do greške, a koje ne

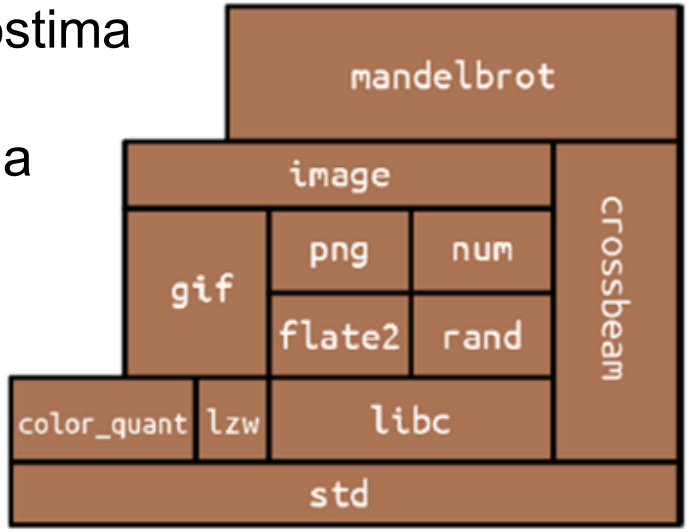
PREDNOSTI ARHITEKTURE SA RESULT

- Rust proverava da li se koriste tip **Result** kao povratna vrednost, tako da se ne može desiti da se slučajno dozvoli da greška prođe bez obrade – tiho (česta greška u C).
- Pošto je **Result** tip podataka kao i svaki drugi, lako je sačuvati rezultate o uspehu i grešci u istoj kolekciji
  - Ovo olakšava modelovanje delimičnog uspeha
  - Na primer, ako se piše program koji učitava milione zapisa iz tekstualne datoteke i potreban je način da se nosi sa tim da će najverovatniji ishod za većinu slučajeva biti uspeh, ali da samo neki neće uspeti, to se može predstaviti u memoriji pomoću vektora **Result** vrednosti
- Mana je da se projektovanju i rukovanju grešaka u Rustu mora posvetiti više pažnje nego u bilo kom drugom jeziku

CREATES I MODULE

SANDUCI (CRATES)

- Rust program se sastoji od sanduka (**Crates**
- Svaki sanduk je potpuna, kohezivna jedinica: sav izvorni kod za jednu biblioteku ili izvršnu datoteku, plus svi povezani testovi, primeri, alati, konfiguracija i drugo
- Mogu se koristiti razne biblioteke nezavisnih proizvođača (third party libreries) i one se sve distribuiraju kao sanduci
- Kad koristite **Cargo** sa određenim zavisnostima (depandencies), sam će skinuti i instalirati sanduke potrebne za izvršavanje programa
- **cargo** build naredba sa **--verbose** oznakom će izlistati veze
- Postoji niz podešavanja i akcija za rukovanje sanducima koju su pokazani na vežbama



MODULI (MODULES)

- Moduli se koriste za organizovanje kode u projektu
- Oni su zapravo imenski prostori u Rustu koji se koriste za organizovanje i čuvanje funkcija, tipova, konstanti i svega ostalog što čini neki Rust program ili biblioteku
- Primer modula je dat na sledećem slajdu
- Modul predstavlja kolekciju elemenata, u primeru su to struktura i 2 metode
- Ključna reč **pub** označava da je nešto javno i da se tome može pristupiti izvan modula
- Kombinacija **pub(crate)** znači da je element vidljiv van modula, ali samo unutar datog sanduka, nije vidljiv iz drugih sanduka i nije deo eksternog interfejsa
- Sve ostalo se tretira kao privatno i nije vidljivo, ni dostupno van samog modula ili modulima potoncima

MODULI (MODULES)


```
mod spores {
    use cells::{Cell, Gene};

    /// A cell made by an adult fern. It disperses on the wind as part of
    /// the fern life cycle. A spore grows into a prothallus -- a whole
    /// separate organism, up to 5mm across -- which produces the zygote
    /// that grows into a new fern. (Plant sex is complicated.)
    pub struct Spore {
        ...
    }

    /// Simulate the production of a spore by meiosis.
    pub fn produce_spore(factory: &mut Sporangium) -> Spore {
        ...
    }

    /// Extract the genes in a particular spore.
    pub(crate) fn genes(spore: &Spore) -> Vec<Gene> {
        ...
    }

    /// Mix genes to prepare for meiosis (part of interphase).
    fn recombine(parent: &mut Cell) {
        ...
    }

    ...
}
```

MODULI (MODULES)

- Kada se neki element modula napravi **pub** to je isto kao da je on izvezen (**exprted**)
- Moduli mogu biti ugnježdeni
- Da bi nešto bilo **pub** u ugnježdenom modulu, svi moduli na putanji moraju biti **pub**
- **pub(super)** će napraviti modul vidljiv samo nadređenom modulu
- **pub(in <path>)** će napraviti modul vidljivim u specifičnom roditeljskom modulu i svim njenim potomcima

```
mod plant_structures {
    pub mod roots {
        ...
    }
    pub mod stems {
        ...
    }
    pub mod leaves {
        ...
    }
}
```

MODULI (MODULES)

- Moduli mogu biti implementirani u zasebnim fajlovima, tada se mora koristiti sintaksa koja će označiti da je implementacija modula u fajlu

```
mod spores;
```

- Označava da je implementacija u fajlu **spores.rs**
- Onda u implementaciji modula ne treba pisati eskplicitno **mod spores**

```
// spores.rs

/// A cell made by an adult fern...
pub struct Spore {
```

MODULI (MODULES)

- Modul može da ima i svoj direktorijum, Rust gleda da li je naziv modula fajl ili direktorijum i onda unutar njega traži ugnježdene module
- U **main** funkcije se onda nalazi:

```
pub mod plant_structures;
```

- Dok se u **plant_structures.rs** nalazi:

```
// in plant_structures/mod.rs
pub mod roots;
pub mod stems;
pub mod leaves;
```

- Po tom principu to može dalje da ide…

```
fern_sim/
├── Cargo.toml
└── src/
    ├── main.rs
    ├── spores.rs
    └── plant_structures/
        ├── mod.rs
        ├── leaves.rs
        ├── roots.rs
        └── stems.rs
```

```
└── plant_structures/
    ├── mod.rs
    ├── leaves.rs
    ├── roots.rs
    ├── stems/
    │   ├── phloem.rs
    │   └── xylem.rs
    └── stems.rs
```

MODULI (MODULES)

- Do elemenata modula se dolazi upotrebom operatora **::**
- Može se definisati čitava putanja prilikom pristupa nekom od elemenata iz modula

```
if s1 > s2 {
    std::mem::swap(&mut s1, &mut s2);
}
```

- Ili je moguće importovati čitav modul sa **use** naredbom

```
use std::mem;

if s1 > s2 {
    mem::swap(&mut s1, &mut s2);
}
```

- Moguće je importovati i više modula odjednom ili sve podmodule iz nekog modula pretka

```
use std::collections::{HashMap, HashSet};  // import both

use std::fs::{self, File}; // import both `std::fs` and `std::fs::File`.

use std::io::prelude::*;  // import everything
```

POLIMORFIZAM U RUSTU

POLIMORFIZAM

- Šta je to?
- Kako to radi u Rustu? Da li smo to negde već videli?
- Rustu podržava polimorfizam kroz:
  - Osobine (**Traits**)
  - Generičnost (**Generics**)
- Deo Rustovog ponašanja vuče iz inspirisanosti Haskelom
- **Traits** su Rustov mehanizam za implementaciju interfejsa ili apstraktnih klasa
- **Generics** sa druge strane su inspirasene C++ templejtima

TRAITS

ŠTA JE TO TRAIT

- Predstavlja osobinu koju bilo koji tip može ili ne mora da ima
- U neku ruku predstavlja mogućnost, nešto što bi tip mogao da uradi
- Neke osobine u std biblioteci
  - Tip koji implementira **std::io::Write** može da piše bajtove u datoteku
  - Tip koji implementira **std::iter::Iterator** može da proizvodi sekvence vrednosti
  - Tip koji implementira **std::clone::Clone** može da klonira sebe u memoriji
  - Tip koji implementira **std::fmt::Debug** može da se ispisuje upotrebom **println!()** sa **{:?}** specifikacijom formata
  - I tako dalje …
- Postoji jedno pravilo vezano za traits a to je da sam trait mora da bude u dosegu, u suprotnom su sve njegove metode skrivene

KAKO TRAIT RADI?

- Postoji jedno pravilo vezano za traits a to je da sam trait mora da bude u dosegu, u suprotnom su sve njegove metode skrivene

```
let mut buf: Vec<u8> = vec![];
buf.write_all(b"hello")?;  // error: no method named `write_all`
```

- U ovom slučaju to znači da mora da se pozove odgovarajući modul

```
use std::io::Write;

let mut buf: Vec<u8> = vec![];
buf.write_all(b"hello")?;  // ok
```

- Moguće je dodavati nove metode bilo kom tipu, pa i standardnom, a moguće je dodavati i nove metode u sanduke, pa se eksplicitnim pozivima odgovarajućih modula izbegavaju konflikti u imenovanju
- Najsigurnije je koristiti sanduke i pakovati kod u zasebne module kako bi se konflikti izbegli
- Ne važi za **Clone** i **Iterator**, jer su deo Rustovog **prelude** mehanizma

KAKO TRAIT RADI?

- Čitav mehanizam radi brzo za razliku od virtualnih metodau C++ ili C#
- Zašto? Kako rade ovi mehanizmi u C++ i C# ?
- Kako izgleda trait? Evo primera iz standardne biblioteke

```
trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;
    fn write_all(&mut self, buf: &[u8]) -> Result<()> { ... }
    ...
}
```

- Ovo bi se dalje koristilo na sledeći način:

```
use std::io::Write;

fn say_hello(out: &mut dyn Write) -> std::io::Result<()> {
    out.write_all(b"hello world\n")?;
    out.flush()
}
```

TRAITS OBJEKTI

- Jedan način da se koriste **traits** je primenom **traits objekata**
- Za definisanje **traits objekata** se koristi sintaksa **dyn Ime_Trait** ili **&mut dyn Ime_Trait** (ovo je zapravo jedini standardni dinamički poziv virtuelne metode u Rustu)
- Rust neće dozvoliti promenljivu tipa **traits objekata**

```
use std::io::Write;

let mut buf: Vec<u8> = vec![];
let writer: dyn Write = buf;  // error: `Write` does not have a constant size
```

- Zašto?

TRAITS OBJEKTI

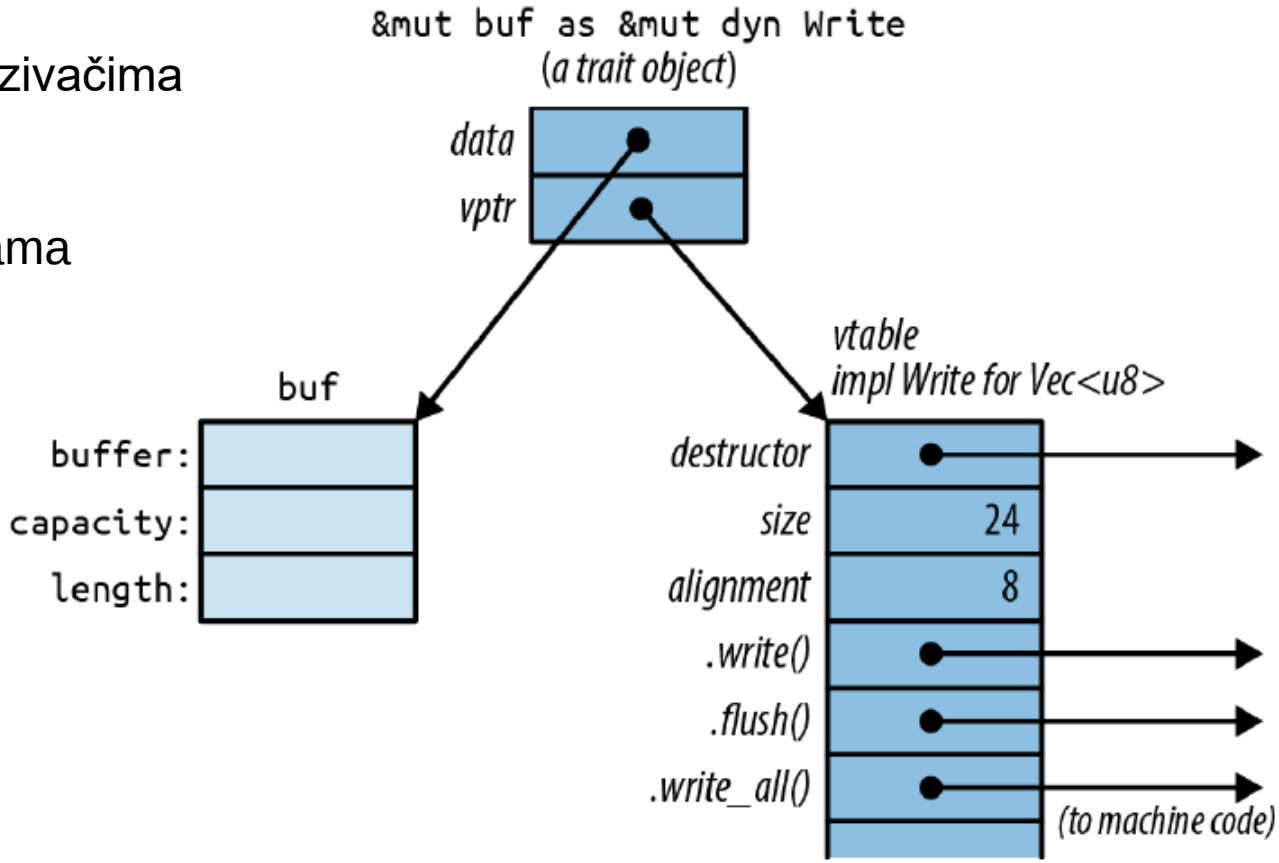
- Mora da se koristi sintaksa Rusta koja će eksplicitno reći da je ovo referenca

```
let mut buf: Vec<u8> = vec![];
let writer: &mut dyn Write = &mut buf;  // ok
```

- Kao i sve druge reference, i ona je deljiva ili mutabilna, pokazuje na određeno parče memorije, ima životni vek
- Ono po čemu se razlikuje je što Rust ne zna unapred kog će tipa biti ova referenca tokom kompajliranja
- Zato traits objekat nosi dodatnu informaciju koja nije vidljiva i služi Rustovom kompajleru

TRAITS OBJEKTI

- U memoriji je **traits** objekat kompleksni pokazivač (**fat pointer**) koji se sastoji iz dva dela
  - Pokazivač na deo sa konkretnim podacima instance tipa
  - Pokazivač na tabelu sa pokazivačima ka konkretnim vrednostima i implementacijama tog tipa



TRAITS OBJEKTI

- Vtable je tabela informacije vezanih za virtualne metode (slično se implementira i u C++)
- Generiše se jednom tokom kompajliranja (**compile time**) i deli se između svih objekata datog tipa
- Sve osenčeno tamnom bojom na slici je deo privatnih implementacijskih podataka Rusta i nije vidljivo programeru (tj. polja nisu direktno dostupna)
- Rust automatski konvertuje obične reference u traits objekte po potrebi, kao i Box<Tip> u traits objekat
- Ova konverzija je zapravo jedini način da nastane traits objekat
- Zapravo je konverzija vrlo jednostavna – u trenutku kada dolazi do konverzije, Rust već zna osnovni tip, tako ostaje još samo da se doda da adresa odgovarajuće vtable strukture

GENERIČKE FUNKCIJE I TIP PARAMETRI

- Primer od ranije sa funkcijom koja prima traits objekat kao parametar

```
fn say_hello(out: &mut dyn Write) -> std::io::Result<()> {
    out.write_all(b"hello world\n")?;
    out.flush()
}
```

se može modifikovati tako da postane generička funkcija

```
fn say_hello<W: Write>(out: &mut W) -> std::io::Result<()> {
    out.write_all(b"hello world\n")?;
    out.flush()
}
```

- Promena postoji samo u zaglavlju funkcije gde se dodaje fraza **<W: Write>** koja zapravo omogućuje generičnost – ovo je tip parametar (**type parameter**)
- **<W: Write>** znači da W stoji na mesto tipa koji implementira **Write** osobinu (obično se koristi jedno veliko slovo za oznaku)

GENERIČKE FUNKCIJE I TIP PARAMETRI

- Koji će tip biti reprezentovan sa W zavisi od poziva funkcije i njene upotrebe:

```
say_hello(&mut local_file)?;  // calls say_hello::<File>
say_hello(&mut bytes)?;       // calls say_hello::<Vec<u8>>
```

- Samim tim zavisi i koje će se metode pozivati
- Ovaj proces se zove monomofizacija (**monomorphization**) i kompajler to radi automatski
- Parametri se mogu eksplicitno navesti prilikom poziva, ali je to retko kad potrebno, pošto Rust to može da dedukuje sam

```
say_hello::<File>(&mut local_file)?;
```

- Slučaju da su argumenti funkcije takvi da se ne može lako dedukovati

```
// calling a generic method collect<C>() that takes no arguments
let v1 = (0 .. 1000).collect();  // error: can't infer type
let v2 = (0 .. 1000).collect::<Vec<i32>>(); // ok
```

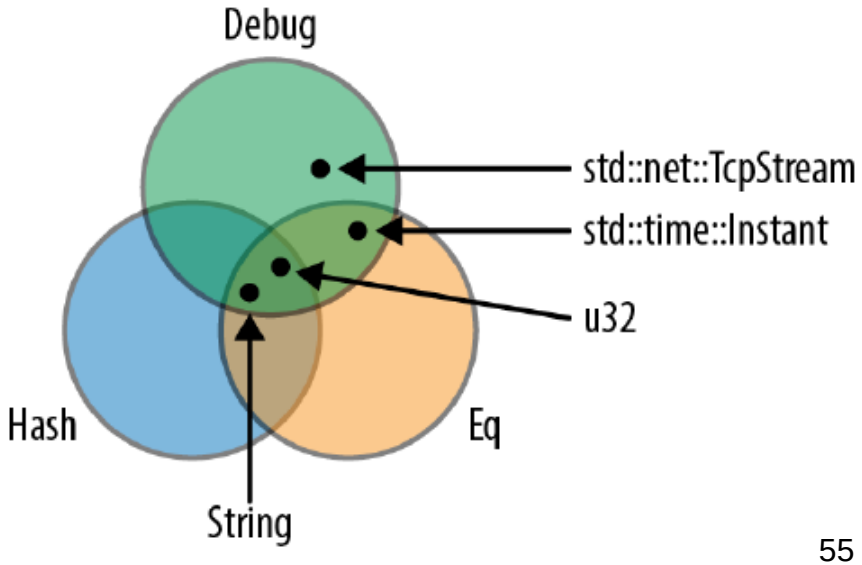
GENERIČKE FUNKCIJE I TIP PARAMETRI

- Moguće je vezati više osobina za jedan tip parametar
- To se radi tako što se sve osobine koje su potrebne (a koje tip implementira) navedu i povežu sa **+** operatorom

```
use std::hash::Hash;
use std::fmt::Debug;

fn top_ten<T: Debug + Hash + Eq>(values: &Vec<T>) { ... }
```

- Ovo ovde sad povezuje osobine iz **Debug**-a i **Hash**-a i **Eq**-a
- Neki tipovi podržavaju samo **Debug**, neki samo **Hash**, neki **Eg**, a neki njihovu kombinaciju, dok u32 i String podržavaju sva tri



GENERIČKE FUNKCIJE I TIP PARAMETRI

- Moguće tip parametar nije povezan ni sa kakvim trait-om, ali se ne može ništa posebno raditi sa njim
- Generičke funkcije mogu imati više tip parametara

```
/// Run a query on a large, partitioned data set.
/// See <http://research.google.com/archive/mapreduce.html>.
fn run_query<M: Mapper + Serialize, R: Reducer + Serialize>(
    data: &DataSet, map: M, reduce: R) -> Results
{ ... }
```

- Pošto ovo može da bude baš dugačko i rogobatno, postoji i alternativa

```
fn run_query<M, R>(data: &DataSet, map: M, reduce: R) -> Results
    where M: Mapper + Serialize,
          R: Reducer + Serialize
{ ... }
```

- Tip parametri su definisani unapred, ali se kasnije navodi za koje su osobine parametri vezani

GENERIČKE FUNKCIJE I TIP PARAMETRI

- Tip parametri se mogu vezati i sa anotacijom životnog veka, samo što se anotacija životnog veka navodi prva

```
/// Return a reference to the point in `candidates` that's
/// closest to the `target` point.
fn nearest<'t, 'c, P>(target: &'t P, candidates: &'c [P]) -> &'c P
    where P: MeasureDistance
{
    ...
}
```

GENERIČKE FUNKCIJE vs. TRAITS OBJEKTI

- Traits objekti su pravi izbor kad god je potrebno da se odjednom koristi kolekcija vrednosti mešovitih tipova, tj. kad nad vrednostima različitih tipovima koristimo istu funkcionalnost
- Traits objekti se koriste i da smanji količina kompajliranog koda, jer se kod generičkih funkcija mora kompajlirati više puta za svaki od tipova sa kojim se koristi
- Generičke funkcije imaju tri prednosti u odnosu na traits objekte
  - Brže su, jer nema trait objekata i referenciranja kroz memoriju, uvek se tačno zna za koji tip se poziva trait metoda
  - Ne podržavaju sve osobine traits objekte; osobina može sadržati i asocirane funkcije koje rade samo sa generičkim funkcijama
  - Moguće je povezati više osobina kroz tip parametra i to proslediti generičkoj funkciji, ovo nije podržano kroz traits objekte
- Zato se generičke funkcije više koriste

DEFINISANJE I IMPLEMENTACIJA OSOBINA

- Definisanje osobine se vrši tako što se ona imenuje i izlistaju se njene metode
- Npr. ako bismo pisali igru

```
/// A trait for characters, items, and scenery -
/// anything in the game world that's visible on screen.
trait Visible {
    /// Render this object on the given canvas.
    fn draw(&self, canvas: &mut Canvas);

    /// Return true if clicking at (x, y) should
    /// select this object.
    fn hit_test(&self, x: i32, y: i32) -> bool;
}
```

DEFINISANJE I IMPLEMENTACIJA OSOBINA

- Sintaksa za implementaciju je sledeća:

```
impl Visible for Broom {
    fn draw(&self, canvas: &mut Canvas) {
        for y in self.y - self.height - 1 .. self.y {
            canvas.write_at(self.x, y, '|');
        }
        canvas.write_at(self.x, self.y, 'M');
    }

    fn hit_test(&self, x: i32, y: i32) -> bool {
        self.x == x
        && self.y - self.height - 1 <= y
        && y <= self.y
    }
}
```

- Impl blok mora sadržati naziv osobine (**trait**) i tip za koji se osobina implementira
- Impl blok sadrži samo metode osobine za dati tip, ništa više

DEFINISANJE I IMPLEMENTACIJA OSOBINA

- Ako se želi dodati neka pomoćna metoda za dati tip koja asistira metodi iz osobine, mora se pisati novi impl blok za sam tip

```
impl Broom {
    /// Helper function used by Broom::draw() below.
    fn broomstick_range(&self) -> Range<i32> {
        self.y - self.height - 1 .. self.y
    }
}
```

- Ta pomoćna metoda se može koristiti iz metode same osobine

```
impl Visible for Broom {
    fn draw(&self, canvas: &mut Canvas) {
        for y in self.broomstick_range() {
            ...
        }
        ...
    }
    ...
}
```

DEFINISANJE I IMPLEMENTACIJA OSOBINA

- Implementacija nekih standardnih osobina u Rust-u sadrži i predefinisane funkcije
- Na primer, moguće je napraviti strukturu i implementirati neke od funkcija osobine, ali ne i sve; u tom slučaju Rust će pozvati podrazumevane (**default**) funkcije

```
trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;

    fn write_all(&mut self, buf: &[u8]) -> Result<()> {
        let mut bytes_written = 0;
        while bytes_written < buf.len() {
            bytes_written += self.write(&buf[bytes_written..])?;
        }
        Ok(())
    }

    ...
}
```

DEFINISANJE I IMPLEMENTACIJA OSOBINA

- Ako se pogleda implementacija **Write** osobine, vidi se da 2 metode nemaju implementaciju, dok **write_all** ima
- To znači, da je prve 2 metode potrebno implementirati, ali da se **write_all** i može i ne mora implementirati

```
/// A Writer that ignores whatever data you write to it.
pub struct Sink;

use std::io::{Write, Result};

impl Write for Sink {
    fn write(&mut self, buf: &[u8]) -> Result<usize> {
        // Claim to have successfully written the whole buffer.
        Ok(buf.len())
    }

    fn flush(&mut self) -> Result<()> {
        Ok(())
    }
}
```

DEFINISANJE I IMPLEMENTACIJA OSOBINA

- Iako nije implementirana za **Sink**, **write_all** se može koristiti

```
let mut out = Sink;
out.write_all(b"hello world\n")?;
```

- Funkcije koje nemaju predefinisanu implementaciju se moraju implementirati
- Osobine koje piše programer takođe mogu imati podrazumevane implementacije funkcija

DEFINISANJE I IMPLEMENTACIJA OSOBINA

- Rust dozvoljava upotrebu i implementaciju bilo koje osobine nad bilo kojim tipom sve dok su ili osobina ili tip definisani u trenutnom sanduku
- To znači da bilo kada se želi dodati nova funkcija u neki tip, to se može izvesti preko osobina

```
trait IsEmoji {
    fn is_emoji(&self) -> bool;
}

/// Implement IsEmoji for the built-in character type.
impl IsEmoji for char {
    fn is_emoji(&self) -> bool {
        ...
    }
}

assert_eq!('$'.is_emoji(), false);
```

- **is_emoji** je vidljiva samo kada je **IsEmoji** u dosegu
- Ova osobina dodaje jednu metodu char tipu (**extension trait**)

DEFINISANJE I IMPLEMENTACIJA OSOBINA

- Moguće je iskoristiti generički impl blok kako bi proširena osobina dodala čitavoj familiji tipova odjednom
- Na primer, ovo se može primeniti nad bilo kojim tipom:

```
use std::io::{self, Write};

/// Trait for values to which you can send HTML.
trait WriteHtml {
    fn write_html(&mut self, html: &HtmlDocument) -> io::Result<()>;
}
```

- Ovo se može proširiti i dodati svim Writerima:

```
/// You can write HTML to any std::io writer.
impl<W: Write> WriteHtml for W {
    fn write_html(&mut self, html: &HtmlDocument) -> io::Result<()> {
        ...
    }
}
```

- **impl<W: Write> WriteHtml** znači **“for every type W that implements Write, here’s an implementation of WriteHtml for W”**

DEFINISANJE I IMPLEMENTACIJA OSOBINA

- Self se može koristiti kao tip u osobini
- Npr. Clone osobina (u pojednostavljenoj verziji) izgleda ovako:

```
pub trait Clone {
    fn clone(&self) -> Self;
    ...
}
```

- Ovde je **Self** iskorišćen kako bi se naglasilo da je povratna vrednost funkcije istog tipa tip koji je pozvao funkciju, tj. da je tip povratne vrednosti **x.clone()** istog tipa kao i **x**
- Čak i ako se naprave dve implementacije za istu osobinu, ali za različite tipove, **Self** se tretira kao alijas za odgovarajući tip

```
pub trait Spliceable {
    fn splice(&self, other: &Self) -> Self;
}
```

DEFINISANJE I IMPLEMENTACIJA OSOBINA

- Tako da je

```
impl Spliceable for CherryTree {
    fn splice(&self, other: &Self) -> Self {
        ...
    }
}

impl Spliceable for Mammoth {
    fn splice(&self, other: &Self) -> Self {
        ...
    }
}
```

- u obe implementacije jasno kada se Self odnosi na CherryTree a kada na Mammoth
- Osobina koja se oslanja na **Self** nije kompatibilna sa trait objektom (tip trait objekta se ne može odrediti tokom kompajliranja)
- Trait objekti su zamišljeni za najjednostavnije vrste osobina

DEFINISANJE I IMPLEMENTACIJA OSOBINA

- Moguće je definisati da je neka osobine proširenje neke druge osobine (**subtrait**)

```
/// Someone in the game world, either the player or some other
/// pixie, gargoyle, squirrel, ogre, etc.
trait Creature: Visible {
    fn position(&self) -> (i32, i32);
    fn facing(&self) -> Direction;
    ...
}
```

- Ovo znači da svaki tip koji implementira **Creature** mora implementirati i **Visible**
- Nije važno kojim se redosledom implementiraju osobine, sve dok su obe implementirane
- Vodite računa da ovo ne podrazumeva nasleđivanje osobina
- Obe osobine moraju biti u dosegu da bi se mogle koristiti (nije dovoljno uvući samo jendu)

DEFINISANJE I IMPLEMENTACIJA OSOBINA

- Osobine mogu uključivati i tip asocirane funkcije korišćenjem odgovarajuće sintakse

```
trait StringSet {
    /// Return a new empty set.
    fn new() -> Self;

    /// Return a set that contains all the strings in `strings`.
    fn from_slice(strings: &[&str]) -> Self;

    /// Find out if this set contains a particular `value`.
    fn contains(&self, string: &str) -> bool;

    /// Add a string to this set.
    fn add(&mut self, string: &str);
}

// Create sets of two hypothetical types that impl StringSet:
let set1 = SortedStringSet::new();
let set2 = HashedStringSet::new();
```

DEFINISANJE I IMPLEMENTACIJA OSOBINA

- U generičkom kodu je slično kao i u negeneričkom samo što se sada poziva konstruktor nad generičkim tipom

```
/// Return the set of words in `document` that aren't in `wordlist`.
fn unknown_words<S: StringSet>(document: &[String], wordlist: &S) -> S {
    let mut unknowns = S::new();
    for word in document {
        if !wordlist.contains(word) {
            unknowns.add(word);
        }
    }
    unknowns
}
```

- Tip asocirane funkcije nisu podržane u trait objektima

POZIVI TRAIT METODA

- U većini slučajeva, da bi se pozvala neka trait funkcija, koristi njen poziv kao metoda nad vrednošću pomoću **.** Operatora
- Na primer:

```
"hello".to_string()
```

- Ovde Rust zaključuje da se radi o **to_string()** funkciji **ToString** osobine i da se poziva njena implementacija za **str** tip
- Čak četiri „igrača“ postoje u ovom pozivu: osobina, funkcija te osobine, implementacija te funkcije za odgovarajući tip i vrednost nad kojom se primenjuje funkcija
- Puni pozivi ove iste metodu su:

```
str::to_string("hello")

ToString::to_string("hello")

<str as ToString>::to_string("hello")
```

POZIVI TRAIT METODA

- U većini slučajeva dovoljno je koristiti **value.method()**
- Ovo ne radi u nekim slučajevima i tada je potrebno navesti punu putanju:

1. Kada postoje dve metode sa istim imenom

```
outlaw.draw();  // error: draw on screen or draw pistol?

Visible::draw(&outlaw);  // ok: draw on screen
HasPistol::draw(&outlaw);  // ok: corral
```

2. Kada se tip **self** argumenta ne može odrediti

```
let zero = 0;  // type unspecified; could be `i8`, `u8`, ...

zero.abs();  // error: can't call method `abs`
             // on ambiguous numeric type

i64::abs(zero);  // ok
```

POZIVI TRAIT METODA

3. Kada je sama funkcija argument u pozivu druge funkcije

```
let words: Vec<String> =
    line.split_whitespace()  // iterator produces &str values
        .map(ToString::to_string)  // ok
        .collect();
```

4. Kada se funkcija osobine poziva u makrou