



Dr Dinu Dragan



PARALELNE I DISTRIBUIRANE ARHITEKTURE I JEZICI (ČAS 5)

ŠTA RADIMO DANAS?



Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici

O NASTAV

- Strukture
- Enumeracija

STRUKTURE



- Strukture se u Rust programskom jeziku grade pomoću **struct** naredbe
- Služe za definisanje „novih“ tipova
- Mogu da imaju asocirane funkcije za rad sa pojedinačnim elementima
- Liče na strukture u C/C++ jeziku
- Podržane su tri vrste struktura u Rust jeziku:
 - named-field, struktura sa imenovanim poljima,
 - tuple-like, struktura sa anonimnim poljima,
 - unit-like, struktura koja nema polja.



NAMED-FIELD STRUKTURE

Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici

- Vrsta Rust Struktura koja najviše liči na ono što očekujemo od strukture:

```
/// A rectangle of eight-bit grayscale pixels.  
struct GrayscaleMap {  
    pixels: Vec<u8>,  
    size: (usize, usize)  
}
```

- Konvencija u Rust-u je da svi tipovi, uključujući strukture, imaju imena koja počinju velikim slovom svake reči, kao što je GrayscaleMap (kamilja notacija)
- Polja i metode su malim slovima, sa rečima odvojenim donjom crtom (notacija u obliku zmije)



NAMED-FIELD STRUKTURE

Dragan de Dinn - Paralelne i distribuirane arhitekture i jezici

- Sama instanca strukture se kasnije pravi na sledeći način:

```
let width = 1024;  
let height = 576;  
let image = GrayscaleMap {  
    pixels: vec![0; width * height],  
    size: (width, height)  
};
```

- Ovo se naziva strukt-izrazom (**structure expression**)
- Mora stojati naziv strukture a unutar vitičastih zagrada lista svih polja (njihovih naziva) i vrednosti koje se dodeljuju tim poljima
- Polja se mogu popuniti i u skraćenom obliku:

```
fn new_map(size: (usize, usize), pixels: Vec<u8>) -> GrayscaleMap {  
    assert_eq!(pixels.len(), size.0 * size.1);  
    GrayscaleMap { pixels, size }  
}
```



NAMED-FIELD STRUKTURE

Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici

- **GrayscaleMap { pixels, size }** je skraćeno od **GrayscaleMap { pixels: pixels, size: size }**
- Pojedinačnim elementima strukture se pristupa preko operatora .

```
assert_eq!(image.size, (1024, 576));  
assert_eq!(image.pixels.len(), 1024 * 576);
```

- Kao i sva ostala polja, strukture su privatne po definiciji, tj. vidljive samo unutar modula
- Ako se želi napraviti javna struktura vidljiva van modula, to se mora eksplicitno naglasiti

```
/// A rectangle of eight-bit grayscale pixels.  
pub struct GrayscaleMap {  
    pub pixels: Vec<u8>,  
    pub size: (usize, usize)  
}
```



NAMED-FIELD STRUKTURE

Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici

- Struktura može biti javna, a da neka od njenih polja ostanu privatna

```
/// A rectangle of eight-bit grayscale pixels  
pub struct GrayscaleMap {  
    pixels: Vec<u8>,  
    size: (usize, usize)  
}
```

- Svako polje za koje se želi da bude javno, mora se eksplicitno proglasiti za javnim
- Samo se javne strukture sa svim javnim poljima mogu kreirati van modula u kojem su definisane
- Zato nije moguće napraviti **Vec** ili **String** (koji su struktura po svojoj prirodi) upotrebom strukt-izraza, već samo upotrebom javne asocirane metode **New**



NAMED-FIELD STRUKTURE

Dragan de Dinn - Paralelne i distribuirane arhitekture i jezici

- Kada se pravi nova instanca strukture, može se koristiti već postojeća instanca iste strukture kako bi se popunili polja nove strukture
- U strukt-izrazu, prvo se navode polja čije se vrednosti definišu, zatim se navode **.. *EXPR*** gde je ***EXPR*** druga instanca iste strukture
- Sva nedefinisana polja nove instance strukture će imati istu vrednost kao i polja u ***EXPR***

```
// In this game, brooms are monsters. You'll see.
```

```
struct Broom {  
    name: String,  
    height: u32,  
    health: u32,  
    position: (f32, f32, f32),  
    intent: BroomIntent  
}
```

```
/// Two possible alternatives for what a `Broom` could be working on.
```

```
#[derive(Copy, Clone)]  
enum BroomIntent { FetchWater, DumpWater }
```



NAMED-FIELD STRUKTURE

Dragan de Dinn - Paralelne i distribuirane arhitekture i jezici

```
// Receive the input Broom by value, taking ownership.
fn chop(b: Broom) -> (Broom, Broom) {
    // Initialize `broom1` mostly from `b`, changing only `height`. Since
    // `String` is not `Copy`, `broom1` takes ownership of `b`'s name.
    let mut broom1 = Broom { height: b.height / 2, .. b };

    // Initialize `broom2` mostly from `broom1`. Since `String` is not
    // `Copy`, we must clone `name` explicitly.
    let mut broom2 = Broom { name: broom1.name.clone(), .. broom1 };

    // Give each fragment a distinct name.
    broom1.name.push_str(" I");
    broom2.name.push_str(" II");

    (broom1, broom2)
}
```



NAMED-FIELD STRUKTURE

Dragan de Dinn - Paralelne i distribuirane arhitekture i jezici

```
let hokey = Broom {  
  name: "Hokey".to_string(),  
  height: 60,  
  health: 100,  
  position: (100.0, 200.0, 0.0),  
  intent: BroomIntent::FetchWater  
};
```

```
let (hokey1, hokey2) = chop(hokey);  
assert_eq!(hokey1.name, "Hokey I");  
assert_eq!(hokey1.height, 30);  
assert_eq!(hokey1.health, 100);
```

```
assert_eq!(hokey2.name, "Hokey II");  
assert_eq!(hokey1.height, 30);  
assert_eq!(hokey2.health, 100);
```



TUPLE-LIKE STRUKTURE

Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici

- Nazivaju se tako zato što liče na n-torke

```
struct Bounds(usize, usize);
```

- Definicija instance ove strukture je slična definiciji n-torke, jedino što se ispred dodaje **struct** rezervisana reč

```
let image_bounds = Bounds(1024, 768);
```

- Poljima instance ove vrste strukture se pristupa kao i poljima n-torke preko operatora `.` i indeksa polja kojem se pristupa

```
assert_eq!(image_bounds.0 * image_bounds.1, 786432);
```

- Kao i kod name-field struktura, i ovde struktura može da se proglasi javnom, kao i njena polja

```
pub struct Bounds(pub usize, pub usize);
```



UNIT-LIKE STRUKTURE

Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici

- Definišu strukturu koja nema elemenata

```
struct Onesuch;
```

- Instanca ove strukture ne zauzima mesto u memoriji, tačnije ima samo jednu moguću vrednost

```
let o = Onesuch;
```

- Korisne su kada se radi sa osobinama (**traits**)
- Dok je izraz poput **3..5** skraćenica za vrednost strukture **Range { start: 3, end: 5 }**, izraz **..** , opseg koji izostavlja obe krajnje tačke, je skraćenica za instancu unit-like strukture **RangeFull**



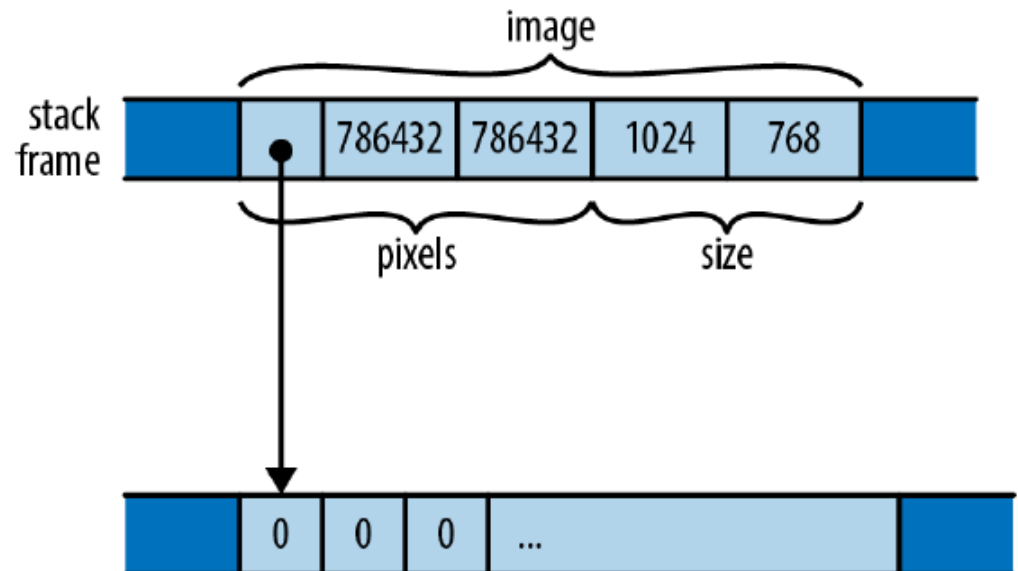
STRUKTURE U MEMORIJI

Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici

- Struktura zauzima kontinualnu memoriju sastavljenu od svih njenih elemenata (tj. njihovih delova koji idu na stek)
- Na primer, struktura:

```
/// A rectangle of eight-bit grayscale pixels.  
struct GrayscaleMap {  
    pixels: Vec<u8>,  
    size: (usize, usize)  
}
```

- se na sledeći način smešta u memoriju:
- Rust ne garantuje da će se elementi rasporediti u istom redosledu u kojem su navedeni u definiciji strukture (ovo je samo jedan mogući raspored)





METODE NAD STRUKTURAMA

Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici

- Rust omogućuje da se nad strukturom definišu metode
- One se ne definišu unutar strukture (ili objekta) već u zasebnom **impl** bloku
- Impl blok predstavlja kolekciju definicija funkcija koje postaju metode na strukturom koja ima isto ime kao i impl blok
- Primer:

```
/// A first-in, first-out queue of characters.  
pub struct Queue {  
    older: Vec<char>,    // older elements, eldest last.  
    younger: Vec<char> // younger elements, youngest last.  
}  
  
impl Queue {  
    /// Push a character onto the back of a queue.  
    pub fn push(&mut self, c: char) {
```


METODE NAD STRUKTURAMA



Dragan de Dinn - Paralelne i distribuirane arhitekture i jezici

```
        self.younger.push(c);
    }

    /// Pop a character off the front of a queue. Return `Some(c)` if there
    /// was a character to pop, or `None` if the queue was empty.
    pub fn pop(&mut self) -> Option<char> {
        if self.older.is_empty() {
            if self.younger.is_empty() {
                return None;
            }

            // Bring the elements in younger over to older, and put them in
            // the promised order.
            use std::mem::swap;
            swap(&mut self.older, &mut self.younger);
            self.older.reverse();
        }

        // Now older is guaranteed to have something. Vec's pop method
        // already returns an Option, so we're set.
        self.older.pop()
    }
}
```




METODE NAD STRUKTURAMA

Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici

- Funkcije iz impl bloka su zapravo asocirane funkcije, pošto su asocirane sa tačno određenim tipom, za razliku od slobodnih funkcija
- Prvi argument asocirane funkcije je uvek referenca na samu instancu strukture koja poziva metodu, **self**
- Pošto je self očigledan, Rust dozvoljava da se ne napiše naziv strukture, već samo da se napiše self u skraćenom obliku
 - **self** je kraćeno za **self: Queue**
 - **&self** je kraćeno za **self: &Queue**
 - **&mut self** je kraćeno za **self: &mut Queue**
- Moguće je koristiti i dužu verziju, ali ne mora
- U Rustu upotreba **self** je obavezna da bi se označilo da se promenljiva odnosi na vrednost nad kojom je funkcija pozvana



METODE NAD STRUKTURAMA

Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici

- Kako u primeru i **pop** i **push** modifikuju red, potrebno je proslediti **self** kao mutabilnu referencu, ali to nije potrebno učiniti i prilikom njihovog pozivanja, obična sintaksa je dovoljno, jer Rust prepoznaje situaciju
- Zato se koristi `q.push(...)` a ne `(&mut q).push(...)`

```
let mut q = Queue { older: Vec::new(), younger: Vec::new() };
```

```
q.push('0');  
q.push('1');  
assert_eq!(q.pop(), Some('0'));
```

```
q.push('∞');  
assert_eq!(q.pop(), Some('1'));  
assert_eq!(q.pop(), Some('∞'));  
assert_eq!(q.pop(), None);
```



METODE NAD STRUKTURAMA

Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici

- Ako ne treba da se modifikuje vrednost, može se koristiti deljena referenca

```
impl Queue {  
    pub fn is_empty(&self) -> bool {  
        self.older.is_empty() && self.younger.is_empty()  
    }  
}
```

- Rust opet sam zaključuje kakvo je pozajmljivanje potrebno, ne mora se eksplicitno navoditi

```
assert!(q.is_empty());  
q.push('○');  
assert!(!q.is_empty());
```



METODE NAD STRUKTURAMA

Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici

- Takođe, moguće je napraviti i pomeranje nad **self** vrednošću

```
impl Queue {  
    pub fn split(self) -> (Vec<char>, Vec<char>) {  
        (self.older, self.younger)  
    }  
}
```

- Rust opet sam zaključuje kakvo je ponašanje u pitanju

```
let mut q = Queue { older: Vec::new(), younger: Vec::new() };  
  
q.push('P');  
q.push('D');  
assert_eq!(q.pop(), Some('P'));  
q.push('X');  
  
let (older, younger) = q.split();  
// q is now uninitialized.  
assert_eq!(older, vec!['D']);  
assert_eq!(younger, vec!['X']);
```



METODE NAD STRUKTURAMA

Dragan de Dinn - Paralelne i distribuirane arhitekture i jezici

- Self argument metode može biti i **Box<Self>**, **Rc<Self>**, ili **Arc<Self>**

```
let mut bq = Box::new(Queue::new());
```

```
// `Queue::push` expects a `&mut Queue`, but `bq` is a `Box<Queue>`.  
// This is fine: Rust borrows a `&mut Queue` from the `Box` for the  
// duration of the call.  
bq.push('■');
```

- Prilikom poziva metode prebacuje se vlasništvo nad pokazivačem na samu metodu
- Metoda koja očekuje referencu na self radi odlično kada se pozive na bilo kojem od ovih pokazivača/referenci, kao što se vidi iz primera
- Za pozive metoda i pristup poljima, Rust automatski pozajmljuje referencu iz tipova pokazivača kao što su **Box**, **Rc** i **Arc**, tako da su **&self** i **&mut self** skoro uvek prava stvar u potpisu metode, zajedno sa povremenom upotrebom **self**



METODE NAD STRUKTURAMA

Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici

- Pristup opisan prethodno nije zadovoljavajuć ako se želi napraviti povezana lista čvorova
- Nešto ovog tipa:

```
use std::rc::Rc;

struct Node {
    tag: String,
    children: Vec<Rc<Node>>
}

impl Node {
    fn new(tag: &str) -> Node {
        Node {
            tag: tag.to_string(),
            children: vec![],
        }
    }
}
```



- Svaki čvor ima tag koji određuje kakva je vrsta čvora i vektor podređenih čvorova
- Rc pokazivač se koristi kako bi moglo da se olakša deljenje čvorova i kako bi bilo više fleksibilnosti u njihovom životnom veku
- Šta ako želimo da imamo metodu koja će čvor dodati u listu nekog drugog čvora?
- Možemo napraviti sledeću metodu

```
impl Node {  
    fn append_to(self: Rc<Self>, parent: &mut Node) {  
        parent.children.push(self);  
    }  
}
```

- Pozivalac sada ima Rc<Node> pri ruci i može da poziva append_to direktno prosleđujući Rc po vrednosti



- Ipak, sledeći kod:

```
let shared_node = Rc::new(Node::new("first"));
shared_node.append_to(&mut parent);
```

- će izgubiti referencu, jer je prebačena na metodu i neće doći do povećanja brojača, ali ako se pozive **append_to** na sledeći način:

```
shared_node.clone().append_to(&mut parent);
```

- U ovom slučaju, povećaće se brojač, ali sada i **shared_note** i **parent**-ov vektor podređenih čvorova pokazuju na isti čvor
- Ako kojim slučajem pozivalac nema referencu, već direktno poseduje čvor, onda mora prvo napraviti **Rc** pokazivač

```
let owned = Node::new("owned directly");
Rc::new(owned).append_to(&mut parent);
```




FUNKCIJE ASOCIRANE SA STRUKTUROM

Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici

- Funkcije unutar **impl** bloka ne moraju da primaju referencu na samu instancu strukture
- U tom slučaju u pitanju su funkcije asocirane tipu (**type-associated functions**)
- Često se koriste za pravljenje konstruktora

```
impl Queue {  
    pub fn new() -> Queue {  
        Queue { older: Vec::new(), younger: Vec::new() }  
    }  
}
```

- Ove funkcije se pozivaju tako što se navede **naziv strukture :: naziv funkcije** (da bi se znalo da su asocirane čitavoj strukturi, a ne samoj instanci strukture)

```
let mut q = Queue::new();
```

```
q.push('*');
```



FUNKCIJE ASOCIRANE SA STRUKTUROM

Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici

- Konvencija u Rustu je da se koristi **new** za konstruktore (**Vec::new**, **Box::new**, **HashMap::new**), ali to nije niti naredba, niti rezervisana reč, pa se funkcije koje konstruišu nešto mogu nazivati i drugačije (**Vec::with_capacity**)
- Moguće je imati više **impl** blokova neke strukture, ali oni svi moraju biti u istom sanduku
- Takođe, moguće je dodati i sopstvene metode drugim tipovima (koje je neko drugi definisao)



KONSTANTE ASOCIRANE SA STRUKTUROM

Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici

- U Rustu je moguće definisati vrednosti povezane sa tipom (strukturom) kao celinom, a ne sa pojedinačnim instancama strukture (nešto poput static u Javi)
- U Rustu se to naziva asocirane konstante (**associated consts**)
- Ovo su konstantne vrednosti i često se koriste da označe neke predefinisane ili uobičajene vrednosti za dati tip, npr.

```
pub struct Vector2 {  
    x: f32,  
    y: f32,  
}  
  
impl Vector2 {  
    const ZERO: Vector2 = Vector2 { x: 0.0, y: 0.0 };  
    const UNIT: Vector2 = Vector2 { x: 1.0, y: 0.0 };  
}
```



KONSTANTE ASOCIRANE SA STRUKTUROM

Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici

- Vrednosti iz primera su asocirane direktno sa Vector2 strukturom i mogu se referisati direktno preko samog tipa
- Pristupa im se preko :: operatora, tako što se navede **naziv strukture :: naziv asocirane konstante**

```
let scaled = Vector2::UNIT.scaled_by(2.0);
```

- Ove konstante ne moraju biti istog tipa kao i struktura
- Mogu se koristiti kao bi se strukturi dodao ID ili naziv

```
impl Vector2 {  
    const NAME: &'static str = "Vector2";  
    const ID: u32 = 18;  
}
```



- Generičke strukture služe za pravljenje templejta koji može da primi različite tipove, npr.

```
pub struct Queue<T> {  
    older: Vec<T>,  
    younger: Vec<T>  
}
```

- <T>** u **Queue<T>** se može pročitati kao “**for any element type T...**”
- Konkretni primer se čita kao: “**For any type T, a Queue<T> is two fields of type Vec<T>**”
- I sam vektor je po svojoj prirodi generička struktura
- Impl blok se deklarise tako što se koristi **impl<T>** kako bi se označilo da je u pitanju implementacija generičkih funkcija za generičku strukturu
- impl<T> Queue<T>** se čita kao: “**for any type T, here are some associated functions available on Queue<T>**”



- Primer `impl<T>` bloka:

```
impl<T> Queue<T> {  
    pub fn new() -> Queue<T> {  
        Queue { older: Vec::new(), younger: Vec::new() }  
    }  
  
    pub fn push(&mut self, t: T) {  
        self.younger.push(t);  
    }  
  
    pub fn is_empty(&self) -> bool {  
        self.older.is_empty() && self.younger.is_empty()  
    }  
  
    ...  
}
```

GENERIČKE STRUKTURE



Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici

- Sa druge strane, Rust omogućava da uz generički impl blok implementirate i blok za specifičnu verziju generičke strukture

```
impl Queue<f64> {  
    fn sum(&self) -> f64 {  
        ...  
    }  
}
```

- **Queue<f64>** se čita kao “**Here are some associated functions specifically for Queue<f64>**”
- **Queue<f64>** dobija metodu **sum** koju jedino on ima, dok ostale verzije generičke strukture to nemaju
- **Rust** sadrži i **Self** parametar koji ukazuje na sam tip (bez obzira šta smo napravili, generički ili ne)

```
pub fn new() -> Self {  
    Queue { older: Vec::new(), younger: Vec::new() }  
}
```



TRAITS I STRUKTURE

Dragan de Dinn - Paralelne i distribuirane arhitekture i jezici

- Ako se želi da struktura ima neku od predefinisanih osobina (traits) to se može obezbediti (pod uslovom da sva polja implementiraju date osobine)
- Osobine se dodaju upotrebom Rust direktiva

```
#[derive(Copy, Clone, Debug, PartialEq)]  
struct Point {  
    x: f64,  
    y: f64  
}
```

- U ovom konkretnom slučaju, pored već viđenih Copy i Clone, dodaje se i Debug (omogućuje `println!("{:?}", point);`) i PartialEq (omogućuje upotrebu operatora `==` i `!=`)

ENUMERACIJA



- Enumeracija u Rustu liči na enumeraciju iz C++ ili C#, ali pored definisanja skupa imenovanih konstanti, ono nudi niz dodatnih osobina
- Enum u Rustu može da pored imenovanih konstanti sadrži i podatke, i to podatke različitih tipova
- Na primer, **Result<String, io::Error>** je **enum** sa dve imenovane konstante:
 - **Ok** čija vrednost sadrži **String**
 - **Err** čija vrednost sadrži **io::Error**
- Ovo najviše liči na C union tip, ali su naravno, Rust enum tipovi sigurni
- Gde god postoji lista unapred definisanih mogućih vrednosti, **enum** je rešenje, jedino što da bi se koristili sigurno u Rust stilu, mora da se koristi poklapanje obrasca (**pattern matching**)



DEKLARACIJA ENUMERACIJE

Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici

- Deklaracija najjednostavnijeg oblika enumeracije ima sledeći izgled:

```
enum Ordering {  
    Less,  
    Equal,  
    Greater,  
}
```

- Ovo deklariše tip **Ordering** sa tri vrednosti ili varijanti:

- **Ordering::Less,**
- **Ordering::Equal** i
- **Ordering::Greater**

- Ovo je deo standardne biblioteke tako da može da se importuje

```
use std::cmp::Ordering;  
  
fn compare(n: i32, m: i32) -> Ordering {  
    if n < m {  
        Ordering::Less  
    } else if n > m {  
        Ordering::Greater  
    } else {  
        Ordering::Equal  
    }  
}
```



DEKLARACIJA ENUMERACIJE

Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici

- Moguće je importovati enumeraciju sa svim svojim konstruktorima, nakon čega se varijante mogu pisati bez imena enumeracije

```
use std::cmp::Ordering::{self, *};    // `*` to import all children

fn compare(n: i32, m: i32) -> Ordering {
    if n < m {
        Less
    } else if n > m {
        Greater
    } else {
        Equal
    }
}
```

- Iako to na prvi pogled olakšava programiranje, praksa je da se ovo ne čini, kako bi se eksplicitnom upotrebom `::` operatora naglasilo da je u pitanju enumeracija



DEKLARACIJA ENUMERACIJE

Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici

- Ako se žele importovati konstruktori svih varijanti enumeracije definisanih u datom modulu, može se koristiti importovanje sa **self** promenljivom

```
enum Pet {  
    Orca,  
    Giraffe,  
    ...  
}  
  
use self::Pet::*;
```

- Kod prostih enumeracija, Rust tretira varijante kao konstante brojeve, počev od 0, pa ih dalje numeriše
- Ako se želi neka određena kombinacija brojeva, to se isto može uraditi
- #[repr]**

```
enum HttpStatus {  
    Ok = 200,  
    NotModified = 304,  
    NotFound = 404,  
    ...  
}
```



DEKLARACIJA ENUMERACIJE

Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici

- Prost enum se može kastovati u int

```
assert_eq!(HttpStatus::Ok as i32, 200);
```

- Obrnuto nije podržano. Šta mislite zašto?
- Ako se želi kastovati iz int u enum, mora se napisati sopstveni kod koji to radi

```
fn http_status_from_u32(n: u32) -> Option<HttpStatus> {  
    match n {  
        200 => Some(HttpStatus::Ok),  
        304 => Some(HttpStatus::NotModified),  
        404 => Some(HttpStatus::NotFound),  
        ...  
        _ => None,  
    }  
}
```



DEKLARACIJA ENUMERACIJE

Dragan de Dinn - Paralelne i distribuirane arhitekture i jezici

- Kao i kod struktura, enumeraciji se mogu pridružiti osobine (traits) koje imaju i elementi od kojih se sastoji (pored onih iz strukture, dodaće i == operator)

```
#[derive(Copy, Clone, Debug, PartialEq, Eq)]  
enum TimeUnit {  
    Seconds, Minutes, Hours, Days, Months, Years,  
}
```

- Enumeracije mogu da imaju i svoje metode



DEKLARACIJA ENUMERACIJE

Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici

- Enumeracije mogu da imaju i svoje metode

```
impl TimeUnit {  
    /// Return the plural noun for this time unit.  
    fn plural(self) -> &'static str {  
        match self {  
            TimeUnit::Seconds => "seconds",  
            TimeUnit::Minutes => "minutes",  
            TimeUnit::Hours => "hours",  
            TimeUnit::Days => "days",  
            TimeUnit::Months => "months",  
            TimeUnit::Years => "years",  
        }  
    }  
  
    /// Return the singular noun for this time unit.  
    fn singular(self) -> &'static str {  
        self.plural().trim_end_matches('s')  
    }  
}
```


ENUMERACIJE SA PODACIMA



Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici

- Varijante u enumeraciji mogu da imaju i svoje vrednosti

```
/// A timestamp that has been deliberately rounded off, so our program  
/// says "6 months ago" instead of "February 9, 2016, at 9:49 AM".  
#[derive(Copy, Clone, Debug, PartialEq)]  
enum RoughTime {  
    InThePast(TimeUnit, u32),  
    JustNow,  
    InTheFuture(TimeUnit, u32),  
}
```

- Dve od tri varijante u ovom primeru imaju svoje vrednosti, n-torku od dva elementa – ovakva enum varijanta se zove **tuple variant**

```
let four_score_and_seven_years_ago =  
    RoughTime::InThePast(TimeUnit::Years, 4 * 20 + 7);  
  
let three_hours_from_now =  
    RoughTime::InTheFuture(TimeUnit::Hours, 3);
```



ENUMERACIJE SA PODACIMA

Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici

- Enum varijante mogu da imaju i strukture kao svoje vrednosti, to su **struct variants**

```
enum Shape {  
    Sphere { center: Point3d, radius: f32 },  
    Cuboid { corner1: Point3d, corner2: Point3d },  
}  
  
let unit_sphere = Shape::Sphere {  
    center: ORIGIN,  
    radius: 1.0,  
};
```

- U osnovi, enum varijante predstavljaju jednu od tri vrste struktura
- Varijanta koja nema vrednost je zapravo unit-like struktura, dok je varijanta sa n-torkom, tuple-like struktura
- Sve tri vrste struktura se mogu javiti kao vrednosti enum varijanti



ENUMERACIJE SA PODACIMA

Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici

- Sve tri vrste struktura se mogu javiti kao vrednosti enum varijanti

```
enum RelationshipStatus {  
    Single,  
    InARelationship,  
    ItsComplicated    ItsExtremelyComplicated {  
        car: DifferentialEquation,  
        cdr: EarlyModernistPoem,  
    },  
}
```

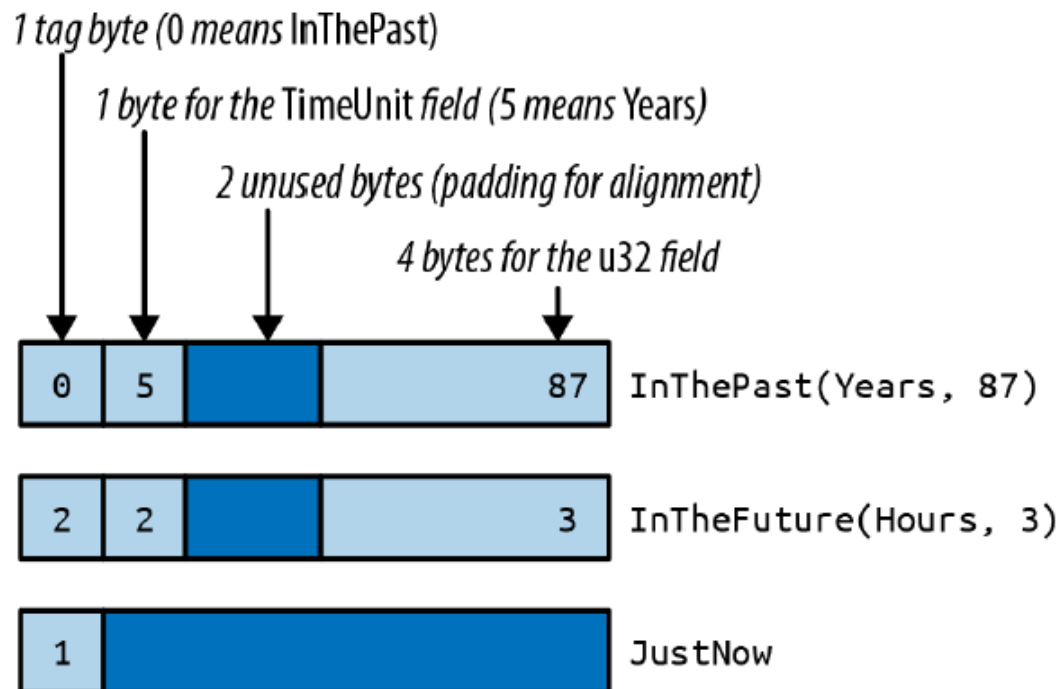
- Svi konstruktori i sva polja u enumu dele vidljivost samog enuma



ENUMERACIJE I MEMORIJA

Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici

- Svaki enum se smešta u memoriju kao jedna celina
- Sastoji se iz dva dela: oznake (tag) i dovoljno memorije da primi sva polja najveće enum varijante
- Oznaka se koristi za Rustovu internu upotrebu. Šta mislite čemu služi?





RICH-DATA I ENUMERACIJA

Dragan de Dinn - Paralelne i distribuirane arhitekture i jezici

- Osnova za izgradnju strukture podataka koja liči na stablo gde element stabla može biti koja vrednost iz enumeracije
- Npr. za radnom JSON format može se napraviti sledeće:

```
use std::collections::HashMap;

enum Json {
    Null,
    Boolean(bool),
    Number(f64),
    String(String),
    Array(Vec<Json>),
    Object(Box<HashMap<String, Json>>),
}
```

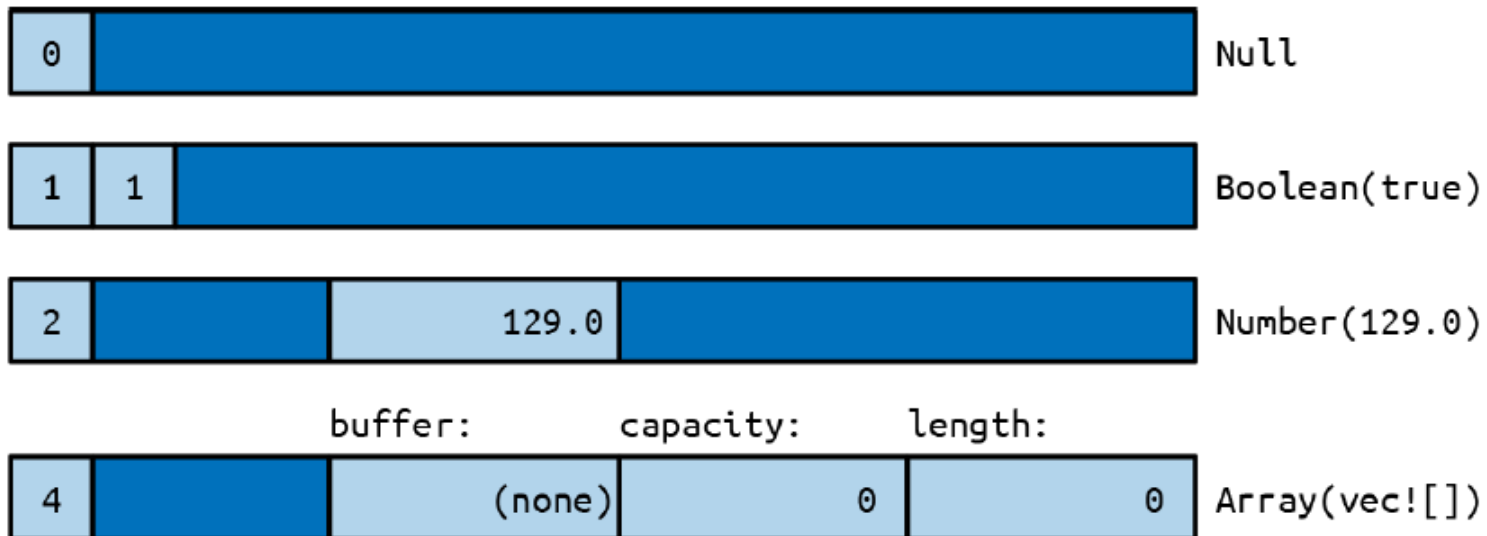
- U memoriji sada bilo koji JSON dokument može da se predstavi kao vrednost ovog novog Rust tipa (nešto što se koristi u implementaciji `serde_json`)



RICH-DATA I ENUMERACIJA

Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici

- Vrlo jednostavno se napravilo nešto što može biti vrlo korisno
- U memoriji to ima sledeći izgled:



GENERIČKA ENUMERACIJA



Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici

- Enumeracija može biti i generička

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

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

- Generičke enumeracije rade isto kao i generičke strukture
- Prednost Rusta je što može da ukloni tag polje oko **Option<T>** kada je tip **T** referenca, **Box**, ili neki drugi pametni pokazivač
- Kako ni jedan od ovih tipova ne može biti null, Rust može da **Option<Box<i32>>** predstavi u memoriji kao jednu mašinsku reč: **0** za **None** i vrednost različita od 0 **Some** pokazivač
- Ipak se očekuje u kodu da se proveriti da li je **Option Some** ili **None**

GENERIČKA ENUMERACIJA



Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici

- Generički tipovi se mogu lako napraviti, npr. u par koraka do binarnog stabla koji može da primi bilo koji broj elemenata generičkog tipa

```
// An ordered collection of `T`s.
```

```
enum BinaryTree<T> {  
    Empty,  
    NonEmpty(Box<TreeNode<T>>),  
}
```

```
// A part of a BinaryTree.
```

```
struct TreeNode<T> {  
    element: T,  
    left: BinaryTree<T>,  
    right: BinaryTree<T>,  
}
```

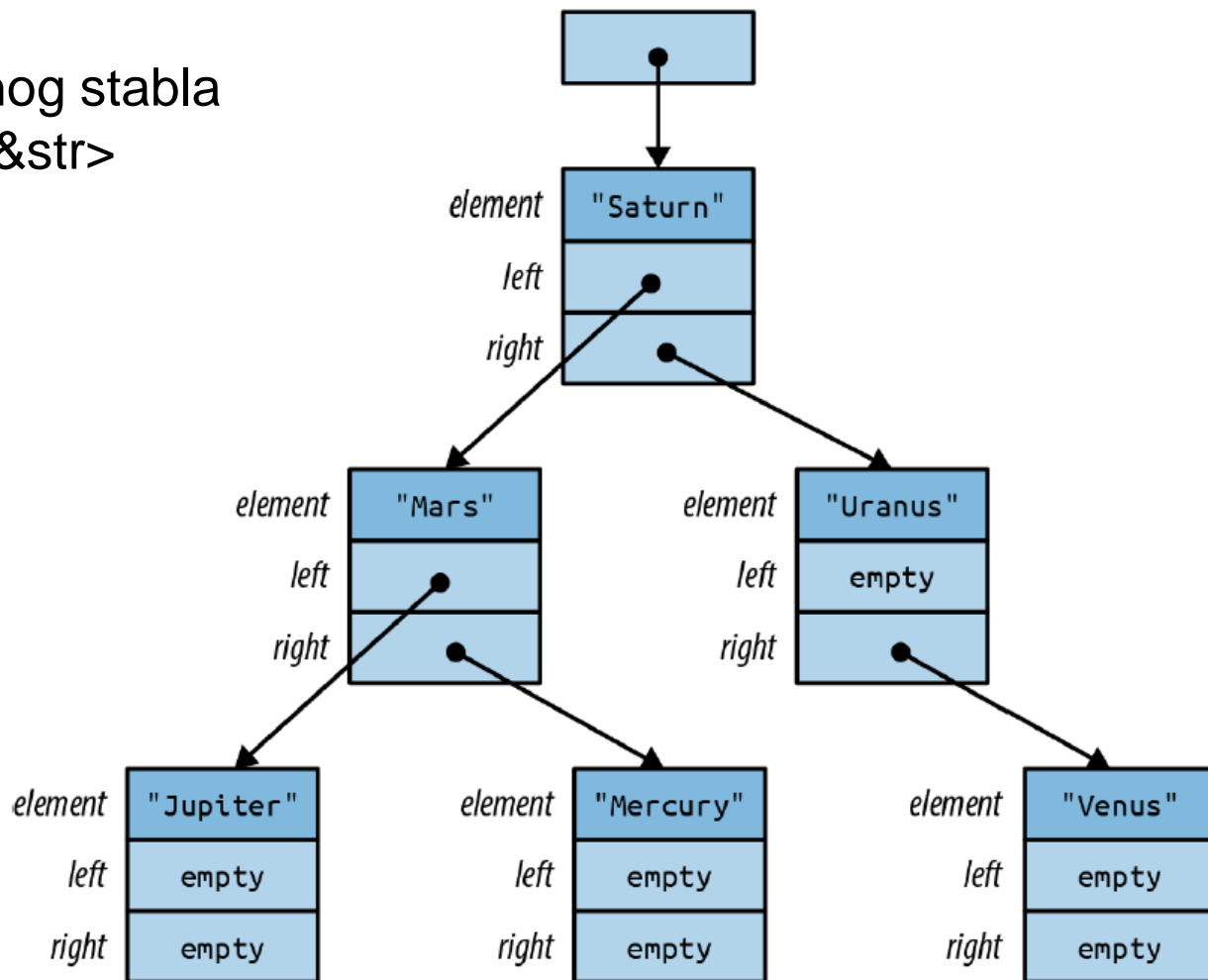
- Svaka vrednost **BinaryTree** je ili **Empty** ili **NonEmpty**
- Ako je **NonEmpty**, onda ima pokazivač na **TreeNode** koji se nalazi na heapu

GENERIČKA ENUMERACIJA



Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici

- Svaki **TreeNode** sadrži jedan element koji je tipa **T** i dva pokazivača na **BinaryTree**
- Struktura binarnog stabla za `BinaryTree<str>` bi izgledala:



GENERIČKA ENUMERACIJA



Dragan de Dinn - Paralelne i distribuirane arhitekture i jezici

- Izgradnja binarnog stabla je jednostavna:

```
use self::BinaryTree::*;  
let jupiter_tree = NonEmpty(Box::new(TreeNode {  
    element: "Jupiter",  
    left: Empty,  
    right: Empty,  
}));
```

- Više stabla se može spojiti u jedno:

```
let mars_tree = NonEmpty(Box::new(TreeNode {  
    element: "Mars",  
    left: jupiter_tree,  
    right: mercury_tree,  
}));
```

- Voditi računa da ova dodele prebacuje vlasništvo nad **jupiter_node** i **mercury_node** na njihov nadređeni čvor, **mars_tree**

GENERIČKA ENUMERACIJA

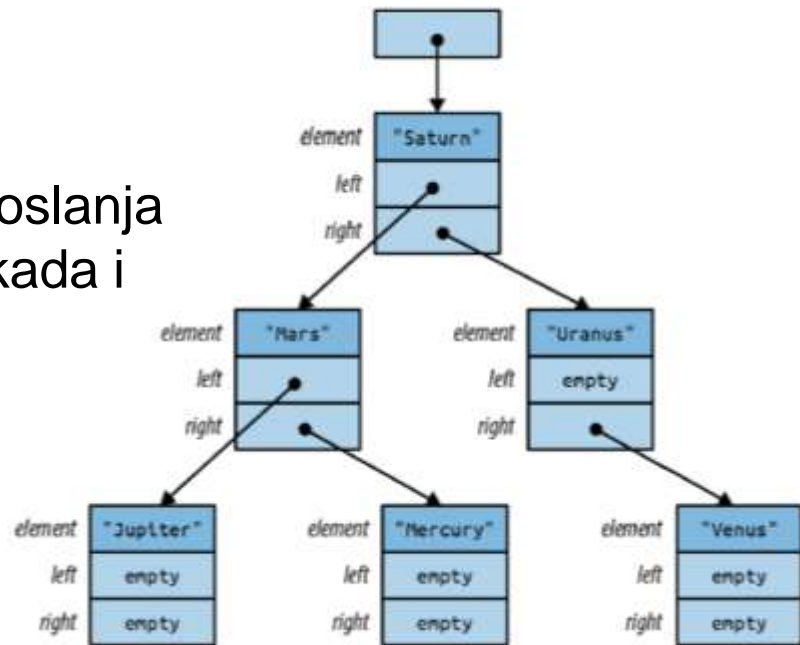


Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici

- Izgradnja stabla nastavlja dalje, a sam korenski čvor se ne razlikuje od ostalih čvorova

```
let tree = NonEmpty(Box::new(TreeNode {  
    element: "Saturn",  
    left: mars_tree,  
    right: uranus_tree,  
}));
```

- Pravljenje ovakvih struktura u Rustu se oslanja **Box** i potrebno je malo navikavanje na kada i gde se ono koristi
- Nekada je najbolji pristup nacrtati sliku strukture i videti gde vam treba pokazivač
- Svaki pravougaonik na slici je ili struktura, ili enumeracija ili n-torka, svaka strelica je ili Box ili neki drugi pametni pokazivač





PATERNI I ENUMERACIJA

Dragan de Dinn - Paralelne i distribuirane arhitekture i jezici

- Mana enumeracije je da se vrednosti ne može pristupiti direktno, već mora preko paterna

- Ovo neće da može:

```
let r = shape.radius; // error: no field `radius` on type `Shape`
```

- Mora da se uvek uhvati ceo patern
- Npr. za enumeraciju:

```
enum RoughTime {  
    InThePast(TimeUnit, u32),  
    JustNow,  
    InTheFuture(TimeUnit, u32),  
}
```

- Da bi se ispisala neka vrednost instance **RoughTime** mora se napisati patern za ispis vrednosti
- Potreba za paternom ne treba da čudi. Zašto?

PATERNI I ENUMERACIJA



Dragan de Dinn - Paralelne i distribuirane arhitekture i jezici

- Kako polja u enumeraciji pored drugačijih varijanti mogu imati i različite dodatne vrednost, postoji mogućnost da se svaka varijanta razlikuje, što može dovesti do potencijalnih grešaka
- Cena toga je da se instanci enumeracije mora pristupati preko paterna

```
1 fn rough_time_to_english(rt: RoughTime) -> String {
2     match rt {
3         RoughTime::InThePast(units, count) =>
4             format!("{}", {} ago", count, units.plural()),
5         RoughTime::JustNow =>
6             format!("just now"),
7         RoughTime::InTheFuture(units, count) =>
8             format!("{}", {} from now", count, units.plural()),
9     }
10 }
```



PATERNI I ENUMERACIJA

Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici

- Jako je važno razumeti kako radi pattern matching
- Pretpostaviti da rt ima sledeću vrednost:
RoughTime::InTheFuture(TimeUnit::Months, 1)
- Matching prvo radi poklapanje po nazivu varijante enumeracije
- Prvo proba poklapa sa **InThePast**

value: RoughTime::InTheFuture(TimeUnit::Months, 1)



pattern: RoughTime::InThePast(units, count)

- Kako tu nema poklapanja, ide dalje i proba poklapanje sa **InTheFuture**

value: RoughTime::InTheFuture(TimeUnit::Months, 1)



pattern: RoughTime::InTheFuture(



units, count)



PATERNI I ENUMERACIJA

Dragan de Dinn - Paralelne i distribuirane arhitekture i jezici

- I tu je ostvario poklapanje
- Kada patern sadrži proste identifikatore (a ne vrednosti), onda ti identifikatori postaju lokalne promenljive nakon paterna i mogu dalje da se koriste
- Šta god da se nalazi u vrednosti varijante enumeracije, kopira se ili premešta u te promenljive
- Ako se umesto identifikatora, koristi vrednost, onda poklapanje proverava da li postoji poklapanje i za tu vrednost

```
RoughTime::InTheFuture(unit, 1) =>  
    format!("a {} from now", unit.singular()),
```

- Ovde sada druga vrednost mora da bude 1 da bi došlo do poklapanja



PATERNI I ENUMERACIJA

Dragan de Dinn - Paralelne i distribuirane arhitekture i jezici

- Rust ima vrlo moćan mehanizam za poklapanje paterna

Pattern type	Example	Notes
Literal	<code>100</code> <code>"name"</code>	Matches an exact value; the name of a <code>const</code> is also allowed
Range	<code>0 ..= 100</code> <code>'a' ..= 'k'</code>	Matches any value in range, including the end value
Wildcard	<code>_</code>	Matches any value and ignores it
Variable	<code>name</code> <code>mut count</code>	Like <code>_</code> but moves or copies the value into a new local variable
ref variable	<code>ref field</code> <code>ref mut field</code>	Borrows a reference to the matched value instead of moving or copying it
Binding with subpattern	<code>val @ 0 ..= 99</code> <code>ref circle @ Shape::Circle { .. }</code>	Matches the pattern to the right of <code>@</code> , using the variable name to the left



PATERNI I ENUMERACIJA

Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici

Pattern type	Example	Notes
Enum pattern	<code>Some(value)</code> <code>None</code> <code>Pet::Orca</code>	
Tuple pattern	<code>(key, value)</code> <code>(r, g, b)</code>	
Array pattern	<code>[a, b, c, d, e, f, g]</code> <code>[heading, carom, correction]</code>	
Slice pattern	<code>[first, second]</code> <code>[first, _, third]</code> <code>[first, .., nth]</code> <code>[]</code>	
Struct pattern	<code>Color(r, g, b)</code> <code>Point { x, y }</code> <code>Card { suit: Clubs, rank: n }</code> <code>Account { id, name, .. }</code>	



PATERNI I ENUMERACIJA

Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici

Pattern type	Example	Notes
Reference	<code>&value</code> <code>&(k, v)</code>	Matches only reference values
Multiple patterns	<code>'a' 'A'</code>	In refutable patterns only (<code>match</code> , <code>if let</code> , <code>while let</code>)
Guard expression	<code>x if x * x <= r2</code>	In <code>match</code> only (not valid in <code>let</code> , etc.)

- Deo ovih mehanizama ćemo preći u nastavku



- Kao što smo već rekli **match** se može koristiti isto kao i **switch**

```
match meadow.count_rabbits() {  
  0 => {} // nothing to say  
  1 => println!("A rabbit is nosing around in the clover."),  
  n => println!("There are {} rabbits hopping about in the meadow", n),  
}
```

- Šta ovo radi? Šta radi **n**?
- I drugi leterali mogu da se koriste kao paterni, boolean, string, char, ...

```
let calendar = match settings.get_string("calendar") {  
  "gregorian" => Calendar::Gregorian,  
  "chinese" => Calendar::Chinese,  
  "ethiopian" => Calendar::Ethiopian,  
  other => return parse_error("calendar", other),  
};
```

- Šta ovo radi? Šta radi **other**?



LITERALI I PATERNI

Dragan de Dinn - Paralelne i distribuirane arhitekture i jezici

- Generalno, za hvatanje svih preostalih vrednosti može da se koristi i znak **_** (**wildcard pattern**)

```
let caption = match photo.tagged_pet() {  
  Pet::Tyrannosaur => "RRRAAAAHHHHHHH",  
  Pet::Samoyed => "*dog thoughts*",  
  _ => "I'm cute, love me", // generic caption, works for any pet  
};
```

- Poklapanje mora da bude sveobuhvatno i najmanju ruku mora da ima makar granu sa znakom **_**
- Redosled navođenja stavki u match je bitan!



N-TORKE I PATERNI

Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici

- Za poklapanje sa n-torkom koristi se tuple matching
- Koristi se kad god želite da proverite poklapanje više podataka u jednom match-u

```
fn describe_point(x: i32, y: i32) -> &'static str {  
    use std::cmp::Ordering::*;  
    match (x.cmp(&0), y.cmp(&0)) {  
        (Equal, Equal) => "at the origin",  
        (_, Equal) => "on the x axis",  
        (Equal, _) => "on the y axis",  
        (Greater, Greater) => "in the first quadrant",  
        (Less, Greater) => "in the second quadrant",  
        _ => "somewhere else",  
    }  
}
```



- Za poklapanje sa strukturom koristi se struct matching koje koristi vitičastu zagradu
- Ovde koristi i posmatra poklapanje za svako od polja

```
match balloon.location {  
    Point { x: 0, y: height } =>  
        println!("straight up {} meters", height),  
    Point { x: x, y: y } =>  
        println!("at ({}m, {}m)", x, y),  
}
```
- U ovom primeru, ako je došlo do poklapanja u prvoj grani, vrednost **balloon.location.y** će se smestiti u promenljivu **height**
- Kako radi poklapanje u ovom slučaju?
- Pretpostaviti da je **balloon.location** tačka **Point { x: 30, y: 40 }**
- Rust kreće od prve grane i redom pokušava poklapanja dok do njih ne dođe



STRUKTURE I PATERNI

Dragan de Dinn - Paralelne i distribuirane arhitekture i jezici

- Rust kreće od prve grane i redom pokušava poklapanja dok do njih ne dođe
- Prva grana ne daje poklapanje

`value: Point { x: 30, y: 40 }`



`pattern: Point { x: 0, y: height }`

- Ali druga zato daje:

`value: Point { x: 30, y: 40 }`



`pattern: Point { x: x, y: y }`

- Zašto?
- Umesto **Point { x: x, y: y }** može i samo **Point {x, y}**



STRUKTURE I PATERNI

Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici

- Pošto polja u strukturi može da bude puno, čak i sa onako skraćenim zapisom može da bude previše, naročito ako nas ne zanimaju sva polja već samo par

```
match get_account(id) {  
  ...  
  Some(Account {  
    name, language, // <--- the 2 things we care about  
    id: _, status: _, address: _, birthday: _, eye_color: _,  
    pet: _, security_question: _, hashed_innermost_secret: _,  
    is_adamantium_preferred_customer: _, }) =>  
    language.show_custom_greeting(name),  
}
```

- Ovo može da se skрати tako što upotrebom operatora .. kažemo da nas ostala polja ne interesuju

```
Some(Account { name, language, .. }) =>  
  language.show_custom_greeting(name),
```




NIZOVI, SLICE I PATERNI

Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici

- Za poklapanje nizova koriste se match arrays
- Koriste se za filtriranje nekih specijalnih kombinacija elemenata niza, naročito kada vrednosti na različitim pozicijama imaju različito značenje
- Kao u primeru transformacije HSL vrednosti u RGB vrednost

```
fn hsl_to_rgb(hsl: [u8; 3]) -> [u8; 3] {  
    match hsl {  
        [_, _, 0] => [0, 0, 0],  
        [_, _, 255] => [255, 255, 255],  
        ...  
    }  
}
```

- Slice patterns se koriste za poklapanje isečaka, ali se ovde pored poklapanja vrednosti gleda i poklapanje broja elemenata u isečku jer i to može da varira



NIZOVI, SLICE I PATERNI

Dragan de Dinn - Paralelne i distribuirane arhitekture i jezici

- Primer slice paterna:

```
fn greet_people(names: &[&str]) {  
    match names {  
        [] => { println!("Hello, nobody.") },  
        [a] => { println!("Hello, {}.\"", a) },  
        [a, b] => { println!("Hello, {} and {}.\"", a, b) },  
        [a, .., b] => { println!("Hello, everyone from {} to {}.\"", a, b) }  
    }  
}
```



REFERENCE I PATERNI

Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici

- Postoje dve vrste reference paterna, prvi koji proverava poklapanje referenci na delove vrednosti koja se poklapa i drugi koji proverava poklapanje reference na čitavu vrednost
- Provera poklapanja vrednosti koje se ne mogu kopirati dovodi do njihovog pomeranja, što uglavnom dovodi do invalidnog koda

```
match account {  
    Account { name, language, .. } => {  
        ui.greet(&name, &language);  
        ui.show_settings(&account); // error: borrow of moved value: `account`  
    }  
}
```

- Šta se ovde dogodilo?



REFERENCE I PATERNI

Dragan de Dinn - Paralelne i distribuirane arhitekture i jezici

- Da bi se to izbeglo potrebno je koristiti poklapanje koje pozajmljuje vrednosti umesto da ih pomera
- Za to se koristi **ref** ključna reč

```
match account {  
    Account { ref name, ref language, .. } => {  
        ui.greet(name, language);  
        ui.show_settings(&account); // ok  
    }  
}
```

- Sada lokalne promenljive **name** i **language** ne preuzimaju vrednosti iz strukture, već ih pozajmljuju, tj. pozajmljuju reference za njih
- Kako je **account** samo pozajmljen, nije konzumiran, ok je koristiti ga dalje u kodu



REFERENCE I PATERNI

Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici

- Za promenljivo pozajmljivanje se koristi **ref mut**

```
match line_result {  
    Err(ref err) => log_error(err), // `err` is &Error (shared ref)  
    Ok(ref mut line) => {           // `line` is &mut String (mut ref)  
        trim_comments(line);       // modify the String in place  
        handle(line);  
    }  
}
```

- Patern **Ok(ref mut line)** mećuje bilo koji uspešan rezultat i pozajmljuje mutabilnu referencu na vrednost smeštenu u njemu



REFERENCE I PATERNI

Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici

- Dok **ref** poklapanje pozajmljuje referencu na delove vrednosti čije se poklapanje sa paternom posmatra, **&** patern gleda poklapanje čitave reference i koristi se kad sa druge strane stiže referenca na nešto

```
match sphere.center() {  
    &Point3d { x, y, z } => ...  
}
```

- Poklapanje u ovom slučaju radi tako da se vraća adresa neke **Point3d**

```
value:  &Point3d { x: 0.0, y: 0.0, z: 0.0}  
        ↓      ↓      ↓  
        ✓      ✓      ✓  
pattern: &Point3d { x, y, z }
```

- U svakom slučaju, i dalje mora da se vodi računa o svemu što Rust radi
- Ne može se tek tako pristupiti delu referenciranog objekta



REFERENCE I PATERNI

Dragan de Dinn - Paralelne i distribuirane arhitekture i jezici

- Ovo ne radi:

```
match friend.borrow_car() {  
    Some(&Car { engine, .. }) => // error: can't move out of borrow  
    ...  
    None => {}  
}
```

- Ali ovo radi:

```
Some(&Car { ref engine, .. }) => // ok, engine is a reference
```



PATERNI I DODATNI USLOVI

Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici

- Nekada je potrebno da poklapanje zadovolji dodatni uslov

```
fn check_move(current_hex: Hex, click: Point) -> game::Result<Hex> {  
    match point_to_hex(click) {  
        None =>  
            Err("That's not a game space."),  
        Some(current_hex) => // try to match if user clicked the current_hex  
                               // (it doesn't work: see explanation below)  
            Err("You are already there! You must click somewhere else."),  
        Some(other_hex) =>  
            Ok(other_hex)  
    }  
}
```

- Zašto ovo ne može?



- Alternativa preko if selekcije:

```
match point_to_hex(click) {  
  None => Err("That's not a game space."),  
  Some(hex) => {  
    if hex == current_hex {  
      Err("You are already there! You must click somewhere else")  
    } else {  
      Ok(hex)  
    }  
  }  
}
```

- Alternativa je tzv. **match guard**:

```
match point_to_hex(click) {  
  None => Err("That's not a game space."),  
  Some(hex) if hex == current_hex =>  
    Err("You are already there! You must click somewhere else"),  
  Some(hex) => Ok(hex)  
}
```



PATERNI I DODATNI USLOVI

Dragan de Dinu - Paralelne i distribuirane arhitekture i jezici

- Poklapanje više paterna može se kombinovati pomoću operatora |

```
let at_end = match chars.peek() {  
    Some(&'\\r') | Some(&'\\n') | None => true,  
    _ => false,  
};
```

- Operator | više gledajte kao znak u regularnom izrazu nego kao ILI
- U primeru će **at_end** biti **true** ako je bilo koji od paterna ispravan
- Za proveru poklapanja sa celom opsegom vrednosti koristi se izraz **..=**

```
match next_char {  
    '0'..'9' => self.read_number(),  
    'a'..'z' | 'A'..'Z' => self.read_word(),  
    ' ' | '\\t' | '\\n' => self.skip_whitespace(),  
    _ => self.handle_punctuation(),  
}
```



DODAVANJE ČVORA U BINARNO STABLO

Dragan de Dinu – Paralelne i distribuirane arhitekture i jezici

- Poklapanje paterna može se iskoristiti za izgradnju metode za dodavanje čvora u binarno stablo (primer sa 90. slajda)

```
1  impl<T: Ord> BinaryTree<T> {
2      fn add(&mut self, value: T) {
3          match *self {
4              BinaryTree::Empty => {
5                  *self = BinaryTree::NonEmpty(Box::new(TreeNode {
6                      element: value,
7                      left: BinaryTree::Empty,
8                      right: BinaryTree::Empty,
9                  })))
10             }
11             BinaryTree::NonEmpty(ref mut node) => {
12                 if value <= node.element {
13                     node.left.add(value);
14                 } else {
15                     node.right.add(value);
16                 }
17             }
18         }
19     }
20 }
```



DODAVANJE ČVORA U BINARNO STABLO

Dragan de Dinn - Paralelne i distribuirane arhitekture i jezici

- Nova metoda se može koristiti na sledeći način:

```
let mut tree = BinaryTree::Empty;  
tree.add("Mercury");  
tree.add("Venus");  
...
```