Concepts of programming languages

Language XXXX

The authors

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Variable bindings

- Use ***let*** keyword to create binding
- Bindings are immutable by default
- Lhs not a name, but a ***pattern***
- Type annotations

```
fn main() {
    let y = 3;
}
let (a, b) = (1, 2);
let mut x : u8 = 10; // Make x mutable
x = 255;
```

Functions

- Use *fn* keyword
- Statements vs expressions

```
fn square(x: i32) -> i32 {
 x * x // Expression
}
fn printSomething() {
println!("Something"); // Statement
}
```

Arrays and tuples

Arrays can be created like this:

```
let mut y = [4, 5, 6];
let second = y[1];
let mut z = [1; 10]; // Ten elements initialized to 1
```

Tuples are created as follows:

```
let mut tuple = (1, 2);
let x = tuple.0;
let y = tuple.1;

println!("Value of y is {}", y);

let (x, y, z) = (1, 2, 3); // Destructured tuple
```

Control flow

- *If*, *else*, *else if*…
- Loops, like ***loop***, ***while*** and ***for***

```
let x = 2;
let y = if x == 3 { 4 } else { 5 };
loop { println!("Infinite loop"); }

for x in 1..10 {}
for (index, value) in (1..10).enumerate() { }

let a = [1, 2, 3, 4, 5];
for elem in a.iter() {
    println!("the value is: {}", elem);
}
```

Vectors

- Dynamic arrays
- Allocated on heap (as opposed to arrays)

```
let mut v = vec![1,2,3]

for i in v {} // For loop takes ownership
for i in &v {} // Reference
for i in &mut v {} // Mutable reference
```

Structs

- Comparable to classes.
- Fields cannot be mutable
- Can also have methods and associated functions

```
struct Point {
    x: i32,
    y: i32,
}

impl Point {
    fn print_xy(&self) {
        println!("x is {}, y is {}", self.x, self.y);
    }
}
let point = Point { x: 3, y: 6 };
point.print_xy(); // Shows x is 3, y is 6.
```

Match

- Comparable to switch
- Allows matching on expressions

```
let x: u32 = 3;

match x {
    1 => println!("one"),
    2 => println!("two"),
    _ => println!("three or more"),
}
```

Enums

- Define type and enumerate on its variants

```
enum Choice {
    Milk(i32),
    Tea(String),
}

let m = Choice::Milk(20);
```

Generics

- Generic structs, enums and functions.

```
fn takes_anything<T, U>(x: T, u: U) {}

struct Value<T> {
value : T
}

enum Choice<T> {
    Milk(T),
}
```

Traits

- Somewhat comparable to an interface
- Use trait bounds on generics

```
struct Square {
    side: f64,
}
trait HasArea {
    fn area(&self) -> f64;
}
impl HasArea for Square {
    fn area(&self) -> f64 {
        self.side * self.side
    }
}
fn print_area<T: HasArea>(shape: T) {
    println!("This shape has an area of {}", shape.area());
}
```

Much, much more..

- Smart pointers
- Concurrency
- Closures
- ...