inovex

Specifying behaviour with Protocols, Typeclasses and Traits

...who wears it better?



About me

Kolja Maier

- Data & ML Engineering @ inovex
- Python & Scala Enthusiast

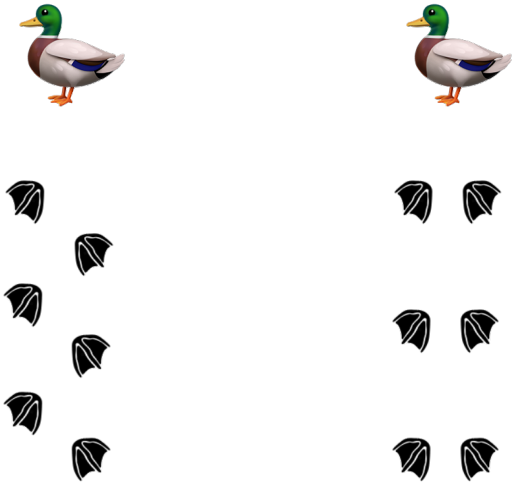
Behaviour

The What & The How

- **What**
  - **Specifying** behaviour

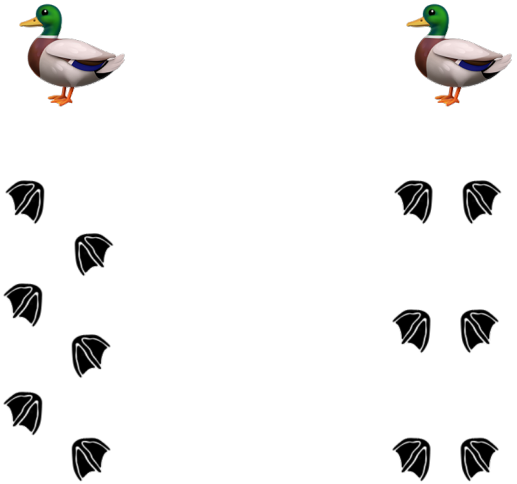
The What & The How

- **What**
  - **Specifying** behaviour
- **How**
  - **Implementing** behaviour



The What & The How

- **What**
  - **Specifying** behaviour
- **How**
  - **Implementing** behaviour
  - **Scope:** typing.Protocols, Traits, Typeclasses



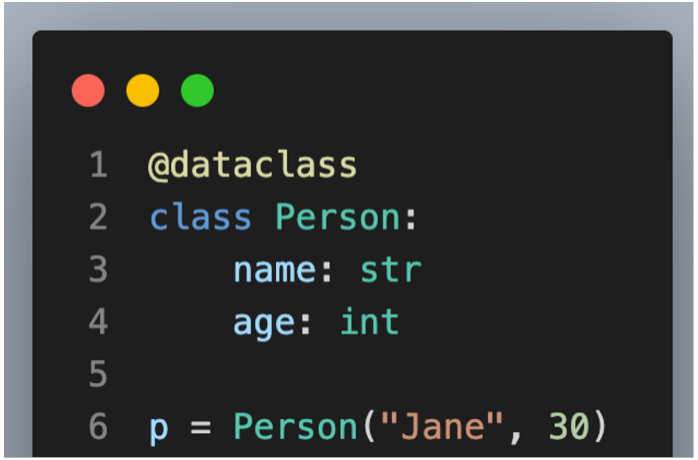
inovex

Duck typing 🦆

```python
def double(x):
    return x * 2

# numeric 🦆
double(42)
double(13.37)
double(1j+2)
```

If it **walks** like a duck and it **quacks** like a duck, then it must be a duck

Duck typing 🦆



```python
@dataclass
class Person:
    name: str
    age: int

p = Person("Jane", 30)
```

Duck typing 🦆

```python
@dataclass
class Person:
    name: str
    age: int

p = Person("Jane", 30)

def double(x):
    return x * 2

double(p)
```

Duck typing 🦆

```python
@dataclass
class Person:
    name: str
    age: int

p = Person("Jane", 30)

def double(x):
    return x * 2

double(p)
```

😎 Very flexible

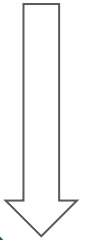
🤔 No static type checks

Repeatable behaviour

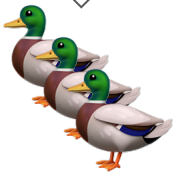
1

.repeat

1,1,...



.repeat



Enter typing.Protocol: **Static** Duck typing

```python
T = TypeVar('T')
class Repeatable(Protocol):
    def repeat(self: T, repeat_count: int) -> List[T]: ...
```

inovex

Credits: Last years EuroPython [1] & [2] L. Ramalho

Enter typing.Protocol: **Static** Duck typing

```python
T = TypeVar('T')
class Repeatable(Protocol):
    def repeat(self: T, repeat_count: int) -> List[T]: ...



def double(x: RT) -> RT:
    return x.repeat(2)

```

inovex

Credits: Last years EuroPython 1 & 2 L. Ramalho

inovex

Enter typing.Protocol: **Static** Duck typing

```python
T = TypeVar('T')
class Repeatable(Protocol):
    def repeat(self: T, repeat_count: int) -> List[T]: ...

# Every type that (at least) implements `Repeatable`
RT = TypeVar('RT', bound=Repeatable)
def double(x: RT) -> RT:
    return x.repeat(2)


@dataclass
class Person:
    def repeat(self, repeat_count: int):
        return [self] * repeat_count
    name: str
    age: int
```

inovex

inovex

Protocols, protocols everywhere

- 😎 No class hierarchies
- 😎 Static type checks
- 😎 Flexible way of assembling concepts
- 🤔 Clutter in classes & TypeVars

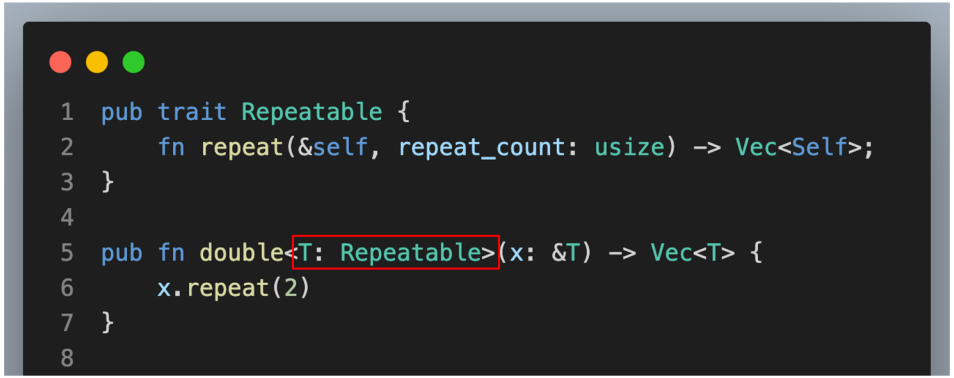
How do other langs solve this?

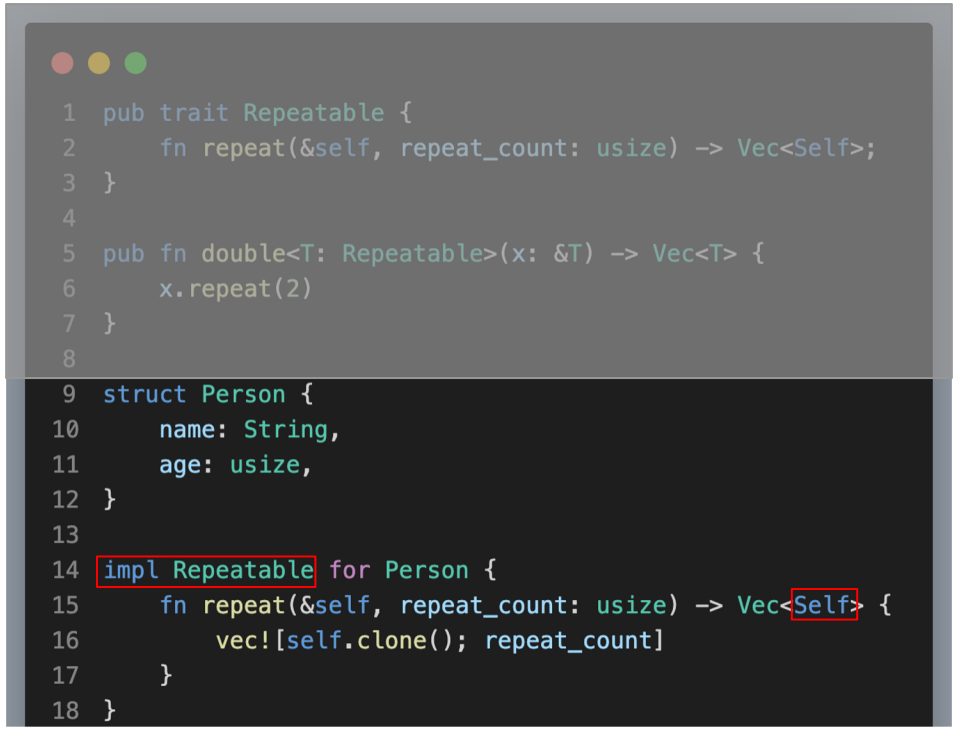
- "Use Python Protocol to define **implicit interfaces**"
- "A protocol is a special case of ABC which works **implicitly**"

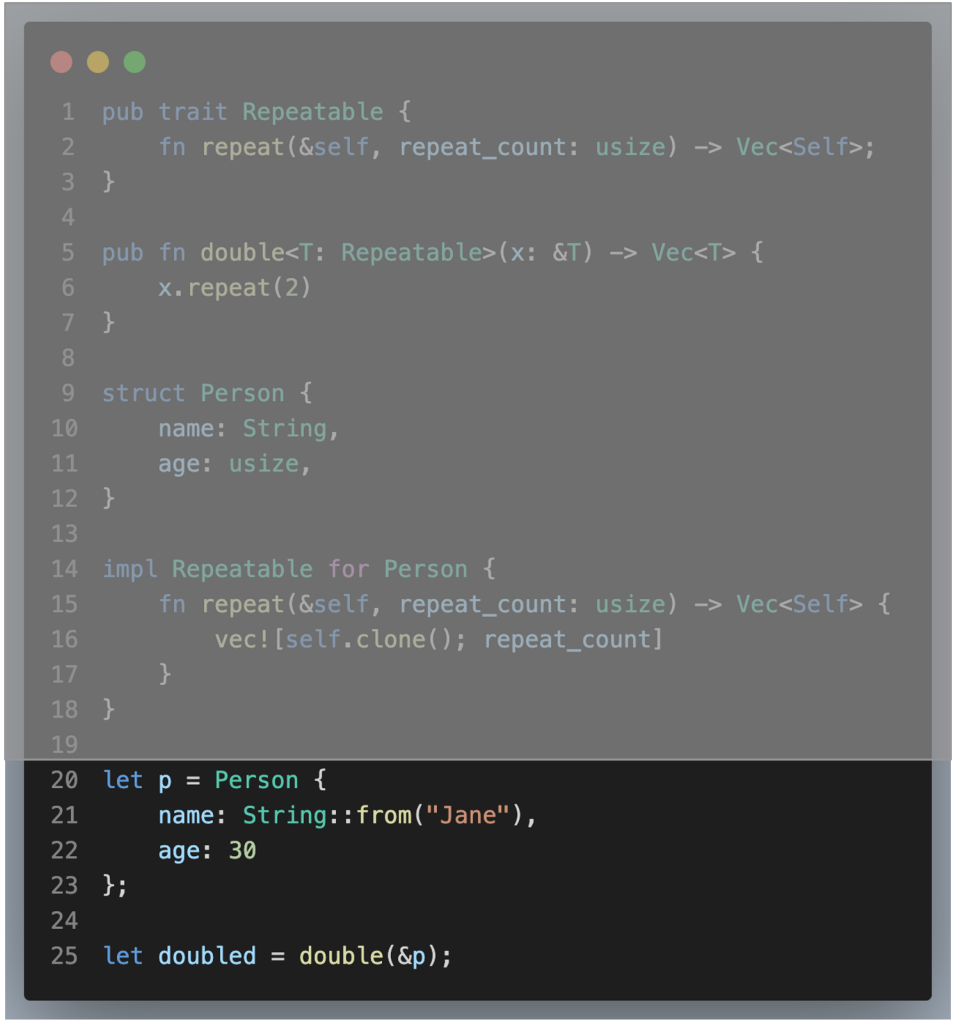
…implicits? I've heard that before!

inovex

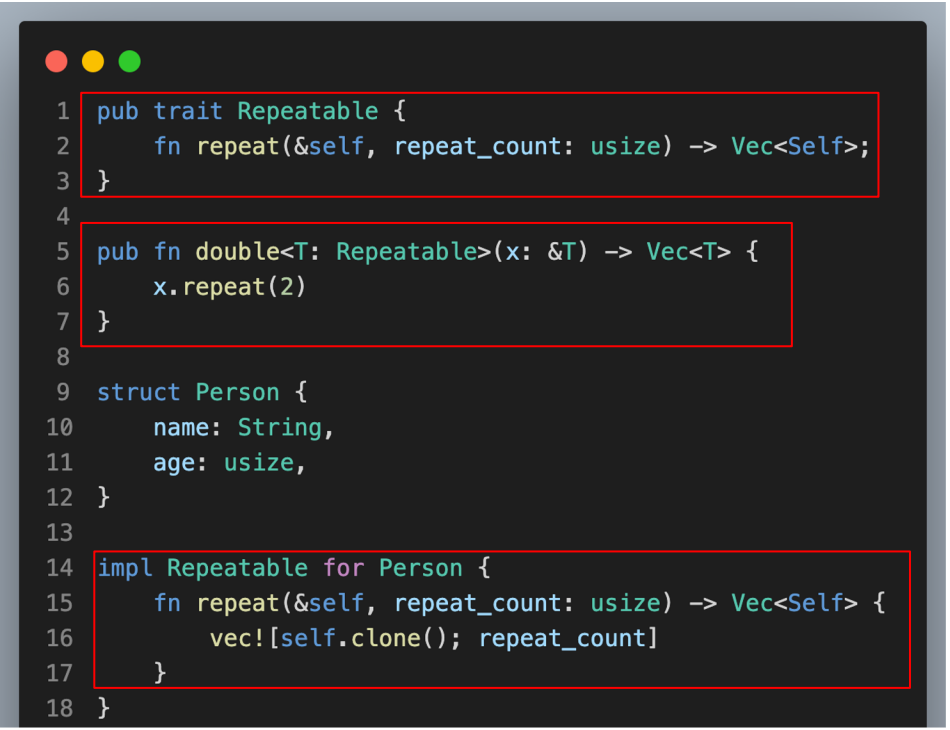
Rust


pub trait Repeatable {
fn repeat(&self, repeat_count: usize) -> Vec<Self>;
}
pub fn double<T: Repeatable>(x: &T) -> Vec<T> {
x.repeat(2)
}

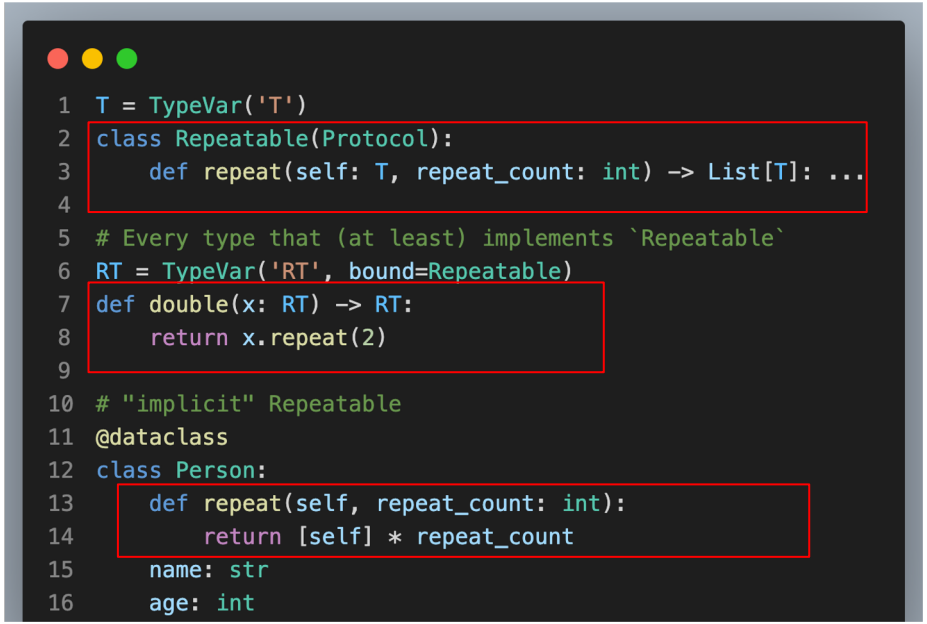

pub trait Repeatable {
fn repeat(&self, repeat_count: usize) -> Vec<Self>;
}
pub fn double<T: Repeatable>(x: &T) -> Vec<T> {
x.repeat(2)
}
struct Person {
name: String,
age: usize,
}
impl Repeatable for Person {
fn repeat(&self, repeat_count: usize) -> Vec<Self> {
vec![self.clone(); repeat_count]
}
}



```rust
pub trait Repeatable {
    fn repeat(&self, repeat_count: usize) -> Vec<Self>;
}

pub fn double<T: Repeatable>(x: &T) -> Vec<T> {
    x.repeat(2)
}

struct Person {
    name: String,
    age: usize,
}

impl Repeatable for Person {
    fn repeat(&self, repeat_count: usize) -> Vec<Self> {
        vec![self.clone(); repeat_count]
    }
}

let p = Person {
    name: String::from("Jane"),
    age: 30
};

let doubled = double(&p);
```

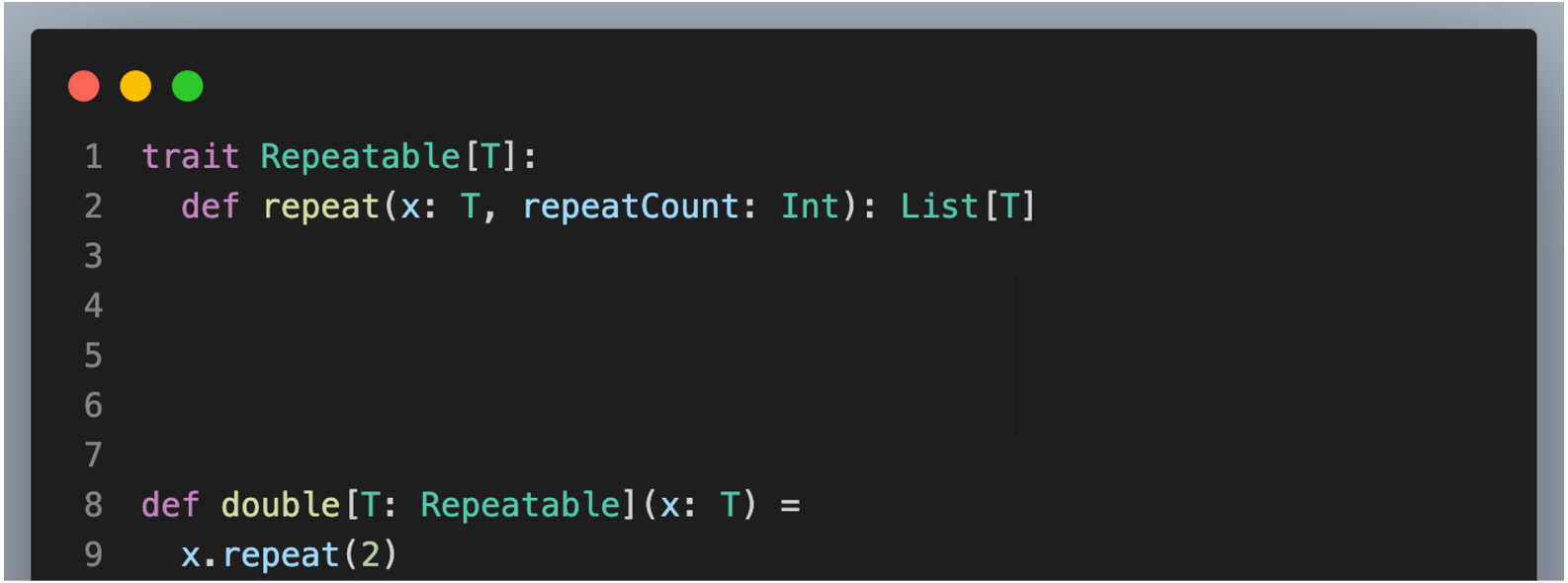


```rust
pub trait Repeatable {
    fn repeat(&self, repeat_count: usize) -> Vec<Self>;
}

pub fn double<T: Repeatable>(x: &T) -> Vec<T> {
    x.repeat(2)
}

struct Person {
    name: String,
    age: usize,
}

impl Repeatable for Person {
    fn repeat(&self, repeat_count: usize) -> Vec<Self> {
        vec![self.clone(); repeat_count]
    }
}
```



```python
T = TypeVar('T')
class Repeatable(Protocol):
    def repeat(self: T, repeat_count: int) -> List[T]: ...

# Every type that (at least) implements `Repeatable`
RT = TypeVar('RT', bound=Repeatable)
def double(x: RT) -> RT:
    return x.repeat(2)

# "implicit" Repeatable
@dataclass
class Person:
    def repeat(self, repeat_count: int):
        return [self] * repeat_count
    name: str
    age: int
```

Scala 3


trait Repeatable[T]:
def repeat(x: T, repeatCount: Int): List[T]
def double[T: Repeatable](x: T) =
x.repeat(2)

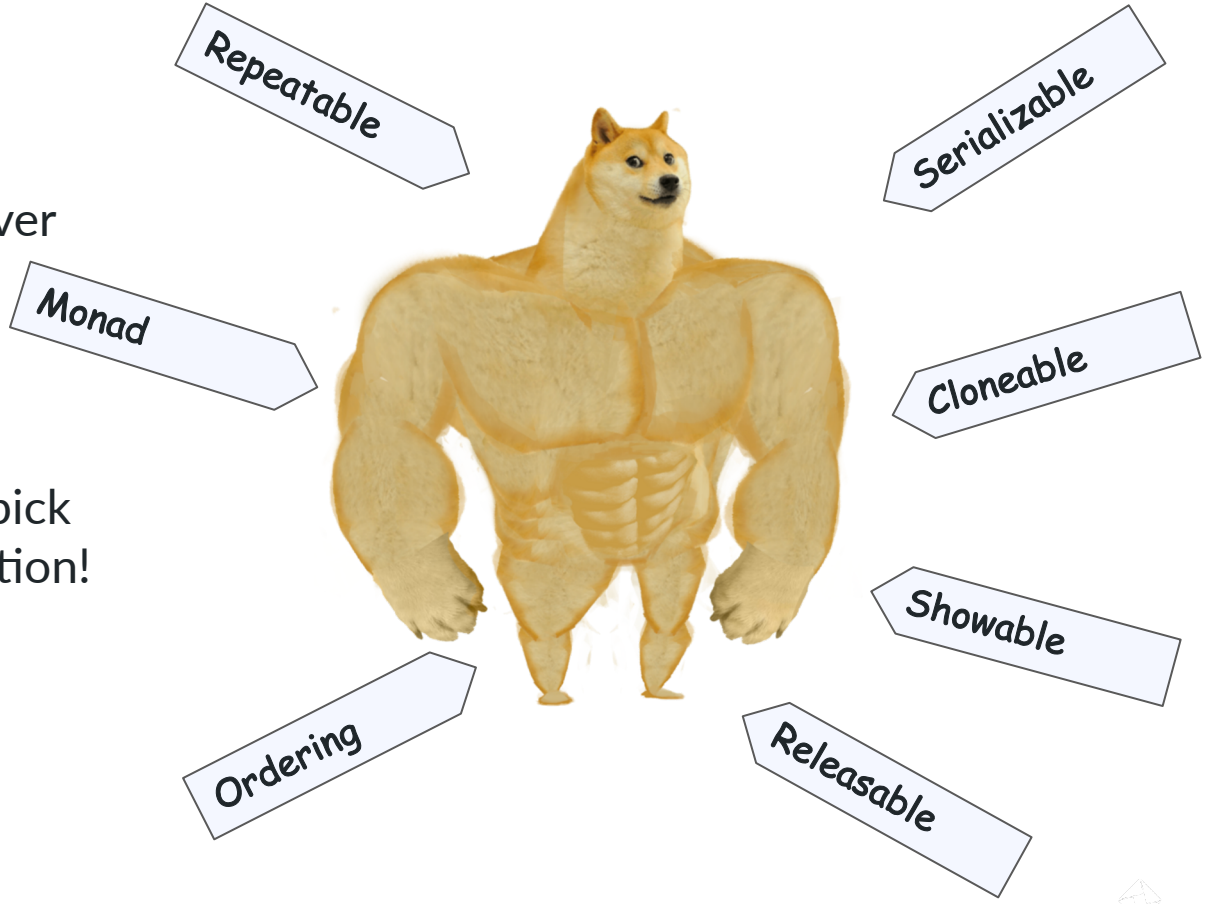
```scala
trait Repeatable[T]:
  def repeat(x: T, repeatCount: Int): List[T]

  extension (x: T)(using r: Repeatable[T])
    def repeat(repeatCount: Int): List[T] =
      r.repeat(x, repeatCount)

def double[T: Repeatable](x: T) =
  x.repeat(2)

case class Person(name: String, age: Int)

given Repeatable[Person] with
  override def repeat(x: Person, repeatCount: Int): List[Person] =
    List.fill(repeatCount)(x)
```

```scala
trait Repeatable[T]:
  def repeat(x: T, repeatCount: Int): List[T]

  extension (x: T)(using r: Repeatable[T])
    def repeat(repeatCount: Int): List[T] =
      r.repeat(x, repeatCount)

def double[T: Repeatable](x: T) =
  x.repeat(2)

case class Person(name: String, age: Int)

given Repeatable[Person] with
  override def repeat(x: Person, repeatCount: Int): List[Person] =
    List.fill(repeatCount)(x)
```

```rust
pub trait Repeatable {
    fn repeat(&self, repeat_count: usize) -> Vec<Self>;
}

pub fn double<T: Repeatable>(x: &T) -> Vec<T> {
    x.repeat(2)
}

struct Person {
    name: String,
    age: usize,
}

impl Repeatable for Person {
    fn repeat(&self, repeat_count: usize) -> Vec<Self> {
        vec![self.clone(); repeat_count]
    }
}
```

Summary

- Powerful mechanisms to **abstract & modularize** over concepts (Repeatable, Serializable,...)
- Languages are moving **towards each other**
- All roads lead to rome - pick the **most pragmatic** solution!



Thank you!

Kolja Maier

Code to this talk

Catch me at our inovex booth

inovex