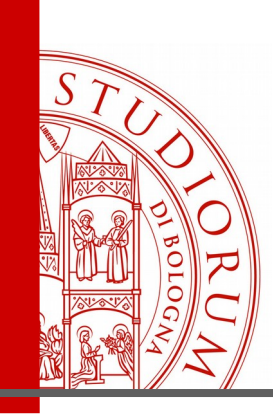
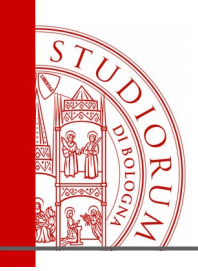


09: Type-Classes, Traits, Mix-ins, ...



Classes are for inheritance

- in OCaml
  - oggetti permettono il dynamic dispatch e il polimormismo ad-hoc (in collaborazione con gli object types che assumono il ruolo di interfacce)
  - le classi invece hanno a che fare con l'ereditarietà che è un meccanismo di riuso del codice
  - l'ereditarietà è problematica
    - confusione con il tipaggio (inheritance is NOT subtyping)
    - mondo aperto: che codice verrà eseguito? come garantire invarianti?

More against inheritance

- inheritance richiede la catalogazione in rigide gerarchie arborescenti che descrivono cosa un oggetto È e non cosa FA
- spesso eccezioni alla struttura arborescente (i pinguini sono uccelli non volanti che nuotano, le biciclette elettriche sono biciclette che inquinano, …)
- le interfacce ci dicono cosa un oggetto FA e sono più rilevanti delle classi (dal punto di vista del tipaggio)

More against inheritance

- cambiare l'interfaccia di una classe rompe tutto il codice che usa classi derivate
- una sottoclasse deve implementare tutto il behaviour della superclasse anche se non necessario; idem per l'istanziazione di campi
- quando una sottoclasse rompe gli invarianti della super-classe fissare il codice può essere molto costoso (re-factoring della gerarchia)

More against inheritance

```
class Stack<T> {          // correct implementation of a stack
        …
        push(T x) { … }
        T pop() { … }
         push_many(T[] a) { for x in a { push(a) } }
}

class StackWithSize<T> extends Stack {  // a stack with explicit size
         int size = 0;

         @Override
         push(T x) { super.push(x); size++ }

         @Override
         T pop() { super.pop(); size--; }
}
```

More against inheritance

```
class Stack<T> {  // the stack implementation is changed to an equivalent one
        …
        push(T x) { … }
        T pop() { … }
         push_many(T[] a) { ... } // direct implementation, doesn't call push
}

class StackWithSize<T> extends Stack {  // and now this code is wrong!
         int size = 0;

         @Override
         push(T x) { super.push(x); size++ }

         @Override
         T pop() { super.pop(); size--; }
}
```

Inheritance vs Composition

- vecchio dibattito, spesso risolto suggerendo di favorire la composition rispetto all'inheritance
- composition: un oggetto di una classe B ha un campo che è un oggetto di una classe A
- B può esporre selettivamente i metodi di A ove necessario, ma non può ridefinirne il codice o rompere gli invarianti
- non richiede gerarchie e non ha problemi di sub-typing

Inheritance vs Composition

Q: If you could do Java over again, what would you change?

A (Gosling): I'd leave out classes!

Dozzine di aneddoti analoghi fin dagli anni 80 da parte dei proponenti il paradigma ad oggetti

Class-less languages

- diversi linguaggi recenti portano la conseguenza all'estremo:

NO INHERITANCE ==> NO CLASSES

NO CLASSES ==> NO OBJECTS

OBJECT-LESS METHODS!

Go: tipi e funzioni

- Go ha funzioni e tipi di dato user-defined del tutto analoghi al C

```
package main

import (
	"fmt"
	"math"
)

type Vertex struct {
	X, Y float64
}

func Abs(v Vertex) float64 {
	return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func main() {
	v := Vertex{3, 4}
	fmt.Println(Abs(v))
}
```

Go: metodi

- un metodo è una funzione per la quale viene dichiarato un receiver di un tipo dichiarato nel file corrente
- i metodi vengono invocati con la sintassi

  o.m(t1,..,tn)

  invece che con la tradizionale

  m(o,t1,….,tn)
- questo aiuta gli IDE per l'autocompletion, etc.

Go: metodi

```
package main

import (
	"fmt"
	"math"
)

type Vertex struct {
	X, Y float64
}

func (v Vertex) Abs() float64 {
	return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func main() {
	v := Vertex{3, 4}
	fmt.Println(v.Abs())
}
```

Go: metodi

```
package main

import (
	"fmt"
	"math"
)

type MyFloat float64

func (f MyFloat) Abs() float64 {
	if f < 0 {
		return float64(-f)
	}
	return float64(f)
}

func main() {
	f := MyFloat(-math.Sqrt2)
	fmt.Println(f.Abs())
}
```

Go: metodi

- un “metodo” Go è solo una funzione invocata con una sintassi differente
- no special typing, la funzione non è un campo di un record, etc.
- invocazione efficiente: static dispatch

Go: metodi

- il receiver può essere un pointer
- la sintassi dell'invocazione non cambia

```
func (v Vertex) Abs() float64 {
	return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func (v *Vertex) Scale(f float64) {
	v.X = v.X * f
	v.Y = v.Y * f
}

func main() {
	v := Vertex{3, 4}
	v.Scale(10)
	fmt.Println(v.Abs())
}
```

Go: interfaces

- interfaccia = insieme di method signatures
- usate per tipare gli argomenti delle funzioni polimorfe (come gli object type di Ocaml!)
- le funzioni che prendono argomenti di tipo un'interfaccia vengono invocate con dynamic dispatch
- per implementare un'interfaccia serve solo implementarne tutti i metodi

Go: interfaces

```
type Perimetrable interface {
        Perimeter() float64
}

func PrintPerimeter(o Perimetrable) {
        fmt.Println(o.Perimeter())
}

type Square struct {
        length float64
}

func (o Square) Perimeter() float64 {
        return o.length*o.length
}
```

```
type Rectangle struct {
        width float64
        height float64
}

func (o Rectangle) Perimeter() float64
{
        return o.width*o.height
}

func main() {
        var s = Square{1}
        var r = Rectangle{2,3}
        PrintPerimeter(s)
        PrintPerimeter(r)
}
```

Go: interfaces

- Under the hood:
  - un valore di tipo interfaccia è o **nil** oppure una coppia

    (v,p)

    dove
    - v è il valore
    - p è un puntatore alla tabella dei metodi per quell'interfaccia
  - in particolare (**nil**, p) è valido e permette di passare a una funzione/metodo **nil**

Go: casts

se o ha tipo un'interfaccia I, allora

- x := o.(T)
  assegna a x il valore di o (la prima proiezione di o) sse o ha tipo T (la seconda proiezione punta a T)
- x,err := o.(T)
  come il precedente, ma invece di un run-time error ritorna **nil** + il booleano err a false

Go: casts

Type switches:

```
switch v := i.(type) {
case T:
    // here v has type T
case S:
    // here v has type S
default:
    // no match; here v has the same type as i
}
```

Go: casts

- cast e type switch sono chiari sintomi dei linguaggi che non hanno un sistema di tipi sufficientemente espressivo:
  - no polimorfismo uniforme (generics, templates, etc.) ==> i container contengono valori di tipo **interface {}**
  - no tipi di dati algebrici ==> no unioni disgiunte di tipi, no valori taggati

Go: inheritance

- un’interfaccia può includere tutti i metodi di un’altra interfaccia tramite il nome della seconda

```
type ColoredPoint interface {
        Point
        GetColor() color
        SetColor(c color)
}
```

Go: composition

- una struttura può avere come campo (anonimo) un'altra struttura
- tutti i campi del campo anonimo sono accessibili come campi della struttura che li importa

```
type ColoredPoint struct {
        Point
        c color
}
```

Go: more on methods

- è possibile estrarre un metodo da un'interfaccia assegnandolo a una funzione

```
type I interface {
        M(name string)
}

func (o I) {
        f := I.M
        var x T          // T implements I
        x.M("ciao")
        f(x,"ciao")      // here x is passed explicitely
}
```

Go: more on methods

- è impossibile definire un metodo il cui receiver sia di tipo interfaccia

```
func (o I) M() { }
```

invalid receiver type I (I is an interface type)

Go: more on methods

Nulla vieta di dichiarare un metodo che opera su tipi funzioni, per esempio decorandole

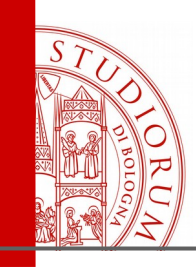
```
type F func(string) string

func (f F) upperCase(s string) string {
    return strings.ToUpper(f(s))
}

func main() {
    f := F(doubleString)

    fmt.Println(f("a"))

    fmt.Println(f.upperCase("a"))
}
```

Rust vs Go

- Rust e Go condividono lo stesso approccio, anche se con terminologia e vincoli differenti
- i metodi sono funzioni con sintassi di invocazione speciale, come in Go, ma dichiarati in appositi blocchi
- possibile dichiarare non-instance methods
- i traits di Rust sono le interfacce di Go, ma vengono implementate esplicitamente

Rust: metodi vs functions

```
struct Point {
    x: f64,
    y: f64,
}

impl Point {
    fn origin() -> Point {                          // non- instance method
        Point { x: 0.0, y: 0.0 }
    }
    fn abs(&self) → f64 {                           // instance method
         (self.x * self.x + self.y * self.y).sqrt()  // method invocation
    }
}

fn abs_origin() → f64 {                             // function declaration
       let x = Point::origin();                     // non-instance met. call
       x.abs()
}
```



Rust: traits

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

trait HasArea {                          // trait declaration
    fn area(&self) -> f64;
}

impl HasArea for Circle {                // trait implementation
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}
```

Rust: trait bounds

- i trait sono usati come bound nel polimorfismo generico

```
trait HasArea {
    fn area(&self) -> f64;
}

fn print_area<T: HasArea>(shape: T) {
    println!("This shape has an area of {}", shape.area());
}
```

Rust: trait bounds

- i trait sono usati come bound nel polimorfismo generico

```
struct Rectangle<T> {
    x: T,
    y: T,
    width: T,
    height: T,
}

impl<T: PartialEq + Debug> Rectangle<T> {
    fn is_square(&self) -> bool {
        println!("{:?}", self.width);       // this requires Debug for T
        self.width == self.height          // this requires PartialEq for T
    }
}
```

Rust: traits

- per implementare il trait T per un tipo U è necessario che almeno uno dei due sia stato definito nel file corrente

Rust: traits

- i traits in Rust possono definire implementazioni di default
- le implementazioni di default possono essere ridefinite quando si implementa il trait

```
trait Foo {
    fn is_valid(&self) -> bool;

    fn is_invalid(&self) -> bool { !self.is_valid() }
}
```

Rust: inheritance

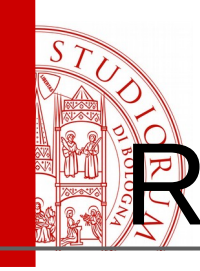
```
trait Foo {
    fn foo(&self);
}

trait FooBar : Foo {
    fn foobar(&self);
}
```

Rust: static vs dynamic dispatch

- come nel caso di Go, tutte le chiamate di metodo sono statiche a meno che non si dichiarino variabili di tipo un trait
- Rust monomorfizza tutte le chiamate, generando più copie dello stesso codice specializzate sul tipo concreto (come fa C++)
- l'inlining di funzioni monomorfizzate funziona senza problemi

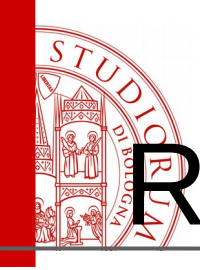
Rust: static vs dynamic dispatch

```
fn do_something(x: &Foo) {       // Foo is a trait
    x.method();
}

fn main() {
    let x = 5u8;
    do_something(&x as &Foo);  // cast to the trait object
                               // triggers dynamic dispatch
}
```



Rust: static vs dynamic dispatch

- un trait object è sempre un puntatore a un trait
- i puntatori hanno tutti la stessa dimensione e quindi possono essere passati allo stesso codice polimorfo (come nel caso di Ocaml)
- come per Go, un trait object in verità è sempre un puntatore all'oggetto più un puntatore alla vtable del trait per un particolare tipo concreto
- il dynamic dispatch inibisce l'inlining



Rust: static vs dynamic dispatch

- non tutti i trait possono essere usati per dichiarare trait objects
- intuitivamente: un trait può essere usato per trait objects se tutti i suoi metodi sono polimorfizzabili, ovvero se prendono in input dati di dimensione nota a compile time (p.e. puntatori) o a run-time (se il loro tipo implementa il trait Sized)

Rust: associated types

```
trait Graph {
    type N: fmt::Display;
    type E;

    fn has_edge(&self, &Self::N, &Self::N) -> bool;
    fn edges(&self, &Self::N) -> Vec<Self::E>;
}

fn distance<G: Graph>(graph: &G, start: &G::N, end: &G::N) -> u32 { ... }

impl Graph for MyGraph {
    type N = Node;
    type E = Edge;

    fn has_edge(&self, n1: &Node, n2: &Node) -> bool {
        true
    }
}
```

Traits, Mix-ins, Interfaces, ...

- essenzialmente variazioni semantiche dello stesso concetto, con differenti trade-off
- la nomenclatura non è consistente: quello che i ricercatori chiamano traits, traits with state, mix-ins, etc. varia da articolo ad articolo e in genere NON corrisponde alla terminologia di chi implementa linguaggi

Traits, Mix-ins, Interfaces, ...

- stato (campi): sì o no?
  - Traits generalmente non definiscono campi, solo metodi in funzione di altri metodi
  - Quindi traits = behaviour only
  - Mix-ins e traits with state: campi + metodi
  - Quindi mix-ins = state + behaviour
- diamond problem:
  - come gestisco l'inclusione di mix-ins che dichiarano lo stesso campo? E lo stesso metodo?
  - traits chiedono al programmatore di risolvere il conflitto facendo override del metodo in conflitto (es. Go, Java)
  - mix-ins risolvono il conflitto favorendo il primo/l'ultimo incluso (es. OCaml)

Traits, Mix-ins, Interfaces, ...

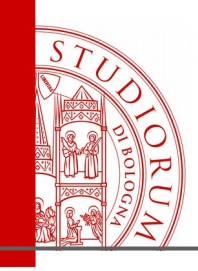
- traits vs interfaces:
  - un'interfaccia dichiara metodi (a volte campi…) senza implementarli...
  - … ma in alcuni linguaggi (es. Java >= 8) i metodi possono avere un'implementazione di default rendendoli di fatto traits
- mix-in vs abstract classes:
  - cambia la pragmatica di utilizzo, ma di fatto concetti equivalenti
  - spesso inheritance ristretta a single inheritance per le abstract classes, sempre multipla per i mix-ins e i traits

Type-classes

- il concetto più corrispondente ai traits nei linguaggi funzionali puri sono le type-classes di Haskell
- una type-class predica di una lista di tipi l'esistenza di alcune funzioni
- Es:
  - Num a vale se il tipo a implementa operazioni numeriche (somma, prodotto, etc.)
  - Eq a vale se elementi di tipo a possono essere confrontati con l'operatore ==

OCaml vs Haskell

OCaml	**Haskell**
‘a, ‘b, ‘c	a, b, c
int, bool, ‘a list, ‘a * ‘b, (‘a,’b) tree, ‘a → ‘b	Int, Bool, [a], (a,b), Tree a b, a → b
true, false, Node (2,3), (2,3), [], 2::[], [2;3;4]	True, False, Node (2,3), (2,3), [], 2:[], [2,3,4]
let f = **function** (2, x) → x | (x, n) → x * n	f 2 x = x f x n = x * n

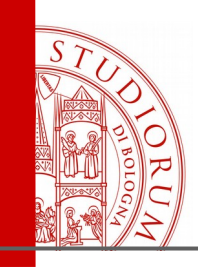


Type classes

```
Prelude> :t ("ciao", 2, 2.4)
("ciao", 2, 2.4) :: (Num b, Fractional c) => ([Char], b, c)

Prelude> f x y z = (x == 2, y < z)
Prelude> :t f
f :: (Eq a1, Num a1, Ord a2) => a1 -> a2 -> a2 -> (Bool, Bool)

Prelude> show 3.1478 ++ show [1,2]
"3.1478[1,2]"
Prelude> f x = (show x, show 2)
Prelude> :t f
f :: Show a => a -> (String, String)
```

Type classes

```
ghci> :t read
read :: (Read a) => String -> a
ghci> read "True" || False
True
ghci> read "8.2" + 3.8
12.0
ghci> read "5" - 2
3
ghci> read "[1,2,3,4]" ++ [3]
[1,2,3,4,3]

ghci> read "4"
<interactive>:1:0:
    Ambiguous type variable `a' in the constraint:
      `Read a' arising from a use of `read' at <interactive>:1:0-7
    Probable fix: add a type signature that fixes these type variable(s)
```



Type classes

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x == y = not (x /= y)          – default implementation
    x /= y  = not (x == y)         – default implementation
    – the two implementations are weird, aren't they? :-)

data TrafficLight = Red | Yellow | Green

instance Eq TrafficLight where
    Red == Red = True
    Green == Green = True
    Yellow == Yellow = True
    _ == _ = False
```

Type classes

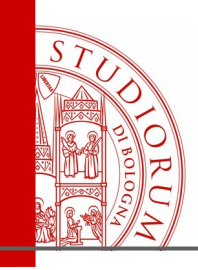
```
data Maybe m = Just m | Nothing
```

– an instance can require some type variables to be members
– of some other type class

```
instance (Eq m) => Eq (Maybe m) where
    Just x == Just y = x == y
    Nothing == Nothing = True
    _ == _ = False
```

– same for classes (a form of inheritance)

```
class (Eq a) => Ord a where
    ...
```

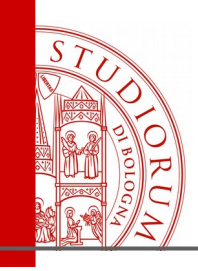


Type classes

– a type class over a type constructor

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b

instance Functor [] where
    fmap = map

instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing
```

```
ghci> fmap (*2) [1..3]
[2,4,6]
ghci> fmap (*2) (Just 200)
Just 400
```

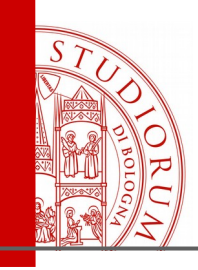


Type classes

ghci -XMultiParamTypeClasses -XFlexibleInstances -XflexibleContexts

– a type class over multiple types

```
Prelude> class Map k v m where search :: k -> m -> Maybe v
Prelude> instance (Eq k) => Map k v [(k,v)] where search = lookup
```



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Claudio Sacerdoti Coen

Dipartimento di Informatica: Scienza e Ingegneria (DISI)

claudio.sacerdoticoen@unibo.it

www.unibo.it