Perché usare i tipi?

1. garantire compile-time assenza di (certi) errori run-time(Safety)
2. permette numerose **ottimizzazioni compile-time**:
   - ottimizzo la taglia di memoria allocata per una variabile;
   - memory management: alloca e automaticamente dealloca fixed-size variables su *stack* invece che su *heap*;
   - meno dynamic binding overhead risolvendo a compile-time la method selection (se so a,b:String opp. a,b:Int $a + b$ fa differenza)
   - si compila 1 volta invece di interpretare ogni volta. Più sono le *informazioni sull'esecuzione* che conosco all'inizio (i.e. tipi statici), maggiori sono le ottimizzazioni.
3. **meno tempo di debugging**, e.g. x.sortt() non compila

Già 2. e 3. fanno **enorme differenza di performance** (l'80% delle prestazioni runtime dipendono dal 20% del codice...va identificato bene)

Perché usare i tipi?

- maggiore documentazione, chiarezza, e quindi manutenibilità e riuso di codice
- esplicitare i tipi spesso significa esplicitare il ragionamento: se ogni errore logico si rispecchiasse in un errore di tipo, il compilatore ci correggerebbe la logica del programma. (tipi come *contratti*)

Featherweight Java — Igarashi, Pierce e Walder, 1999

Un minimal core calculus che modella il type system di Java.

Un *modello formale* di un linguaggio si usa

- per **descrivere** in modo *preciso* qualche aspetto del design del linguaggio,
- per **enunciare** le sue proprietà e **dimostrarle**.
- Permette così di individuare problemi che non erano stati osservati ad un livello più informale.
- Richiede un tradeoff tra *completezza* e *compattezza*

Featherweight Java

Cosa c'è e cosa non c'è:

- *classi e sottoclassi, creazione di oggetti, invocazione di metodi, selezione di campi, cast e variabili.*
- Non ci sono gli assegnamenti (campi dati inizializzati dal costruttore): è il "**frammento funzionale**" di Java.
- è abbastanza per studiare le core features del type system di Java: *classi mutuamente ricorsive, method override e dynamic binding, method recursion tramite parametro* `this`, *subtyping, cast.*

E poi? ...

GJ=FJ+classi generiche FJ+inner classes

FJ+RMI FJ+Mixins FJ+Unknown-type

Un programma in FJ: (CT, t)

```
class A extends Object { A() { super(); } }
class B extends Object { B() { super(); } }
class Pair extends Object {
    Object fst; Object snd;
    Pair(Object fst, Object snd) { super(); this.fst=fst; this.snd=snd; }
    Pair setfst(Object newfst){ return new Pair(newfst, this.snd); }
}
```

- new Pair(new A(), new B()).snd $\longrightarrow^*$ new B()
- new Pair(new A(), new B()).setfst(new B())
  $\longrightarrow^*$ new Pair(new B(), new B())}
- ((Pair)(new Pair(new Pair(new A(), new B()), new A()).fst)).snd
  $\longrightarrow^*$ new B()

Gli errori runtime (stuck): Message Not Understood

1. l'accesso ad un campo o ad un metodo non definito nella classe corrispondente: t.f, t.foo()
2. il passaggio di parametri di numero e/o tipo scorretto
3. il tentativo di fare un cast ad un tipo che non è una superclasse del tipo run-time dell'oggetto: (Pair)new Object()

Teorema (Safety)

- In un programma ben tipato non compaiono runtime errori del tipo 1. e 2.
- Gli errori di tipo 3. non compaiono in programmi ben tipati *che non contengono down cast* .

Static Typing vs Dynamic Type Checking

Static Typing ogni *espressione* del programma ha un tipo. Le regole di typing sono usate compile time per controllare se un programma è ben tipato.

Dynamic Type Checking variabili, metodi ed espressioni di un programma in genere non sono tipati, ma ogni *oggetto* e ogni *valore* ha un tipo. Il sistema controlla a runtime se le operazioni vengono applicate ad argomenti del tipo corretto.

Duck Typing

When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.

Semantica di un oggetto

- non il suo tipo/classe
- ma l'insieme *corrente* di metodi a cui risponde

Esempio in Python:

```
fun(x) { x.walk(); x.quack(); }
```

Non interessa il tipo di `x` ma il fatto che rispode ai metodi `walk` e `quack`

Duck Typing

When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.

Esempio in Python:

```
fun(x) { x.walk(); x.quack(); }
```

```
class Duck:
    def quack(self):print("Quaaaaaack!")
    def walk(self):print("The duck walks")

class Person:
    def quack(self):print("The person imitates a duck.")
    def walk(self):print("The person imitates a duck.")
    def name(self):print("John Smith")
```

In Java... `fun(x:Duck) { x.walk(); x.quack();}`

Duck Typing

When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.

Java Adapter:

```
fun(x:Duck) { x.walk(); x.quack(); }
```

```
abstract class AsADuck extends Duck {
    void quack();
    void walk(); }

class PersonAdapter extends AsADuck {
    private Person adaptee;
    void quack(){ return adapteee.quack();}
    void walk() { return adapteee.walk();}
}
```


Duck Typing (Dynamic Typing)

When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.

Esempio in Python: `fun(x) { x.walk(); x.quack(); }`

In Java... `fun(x:Duck) { x.walk(); x.quack();}`

- controlla a runtime se il parametro ha il metodo invocato, altrimenti segnala errore
- flessibile, ma bisogna conoscere bene cosa fa il codice perché non ho il tipo che descrive il comportamento degli oggetti! (*testing!*)

If it walks like a duck and quacks like a duck, it could be a dragon doing a duck impersonation. You may not always want to let dragons into a pond, even if they can impersonate a duck.

Duck Typing (Dynamic Typing)

When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.

Esempio in Python: `fun(x) { x.walk(); x.quack(); }`

In Java... `fun(x:Duck) { x.walk(); x.quack();}`

- controlla a runtime se il parametro ha il metodo invocato, altrimenti segnala errore
- flessibile, ma bisogna conoscere bene cosa fa il codice perché non ho il tipo che descrive il comportamento degli oggetti! (*testing!*)
- Utile per far interagire componenti eterogenee, scaricate dinamicamente, o di cui non si sa a quali messaggi rispondono (interoperabilità)
- Supportato da Objective-C, JavaScript, Perl, Python, Ruby.
- Scala???

Nominal vs Structural subtyping

type system **nominali** In Java (e FJ) un tipo è definito insieme al suo nome, e un tipo/nome è sottotipo di un altro solo se è stato esplicitamente definito tale.

type system **strutturali** In ML, Haskell, il nome del tipo è irrilevante, e il subtyping definito direttamente sulla struttura dei tipi.

- Il nome dei tipi è utile a runtime. Ogni oggetto runtime ha un tag con il nome del suo tipo, utile per il runtime type test (`instanceOf` e downcast), il marshalling degli oggetti e la riflessività.
- I s. nominali gestiscono in modo semplice tipi ricorsivi (liste, alberi..)
- I s. strutturali sono più adatti in presenza di features più avanzate, come il *polimorfismo parametrico*, gli *abstract data types*, i *type-operators* definiti dall'utente, o i *funtori* (ML o Haskell).
- il rapporto tra sistemi nominali e strutturali è ancora tema della ricerca corrente.

Scala e il subtyping strutturale

Esempio in Python:

```
fun(x) {
    x.walk(); x.quack();
}
```

```
class Duck:
    def quack(self):print("Quaaaaaack!")
    def walk(self):print("The duck walks")

class Person:
    def quack(self):print("The person ...")
    def walk(self):print("The person ...")

    def name(self):print("John Smith")
```

In Scala:

```
def fun( x: { def walk():String, def quack():String } ):Unit =
    { x.walk(); x.quack() }

fun(new Duck());    fun(new Person())
```

Recupera (parte della) flessibilità del duck typing e la safety del typing statico!

Scala e il subtyping strutturale

Scala eredita da Java il typing nominale...
ma ci sono **anche i tipi strutturali:**

```
class File(name:String) {
    def getName():String = name
    def open() { /*..*/ }
    def close() { println("close file") }
}

def test(f: { def getName():String} ){println(f.getName)}

test(new File("test.txt"))
test(new java.io.File("test.txt"))
```