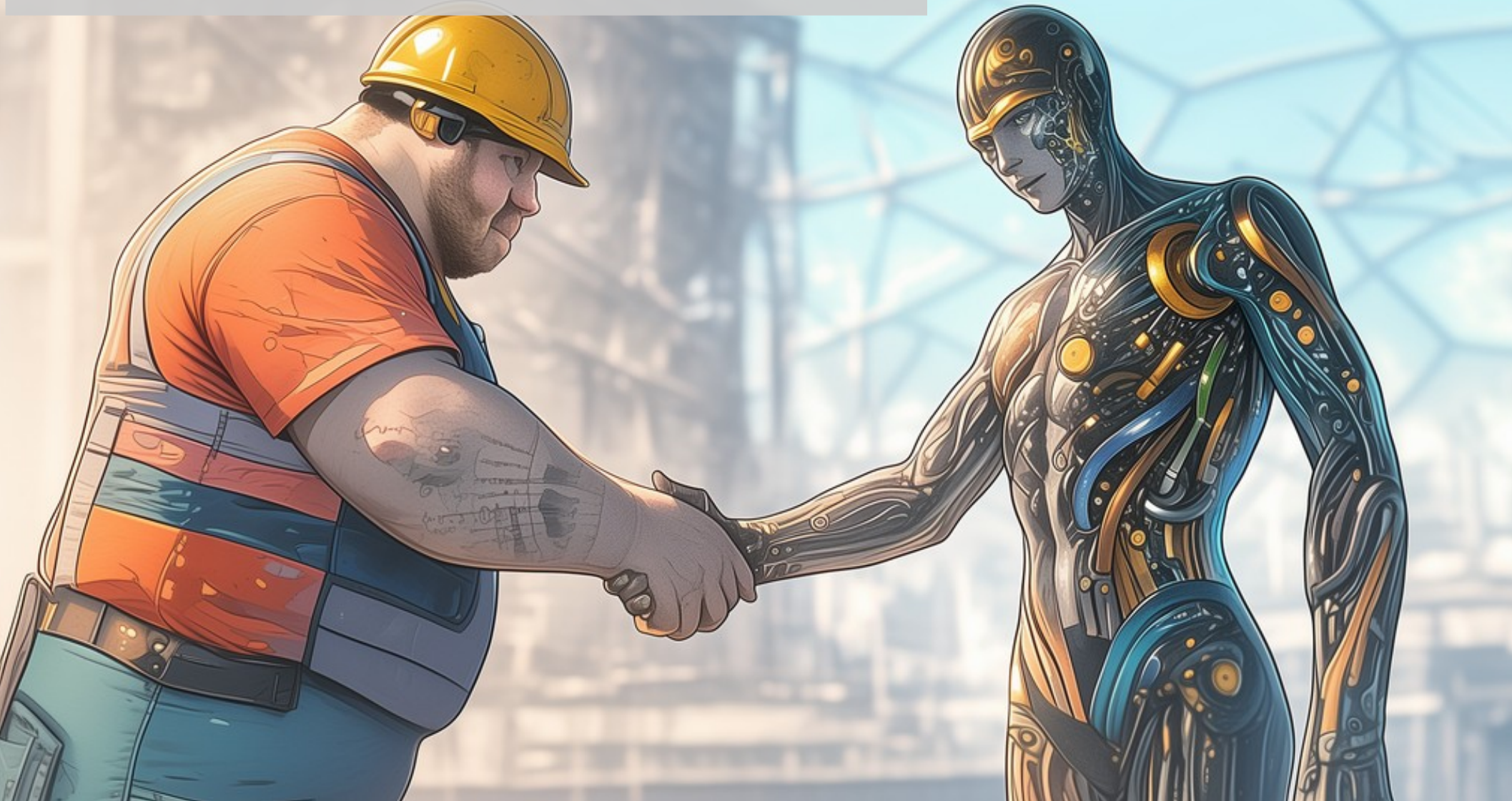


Fearless





Fearless has two souls working together



Fearless has two souls working together

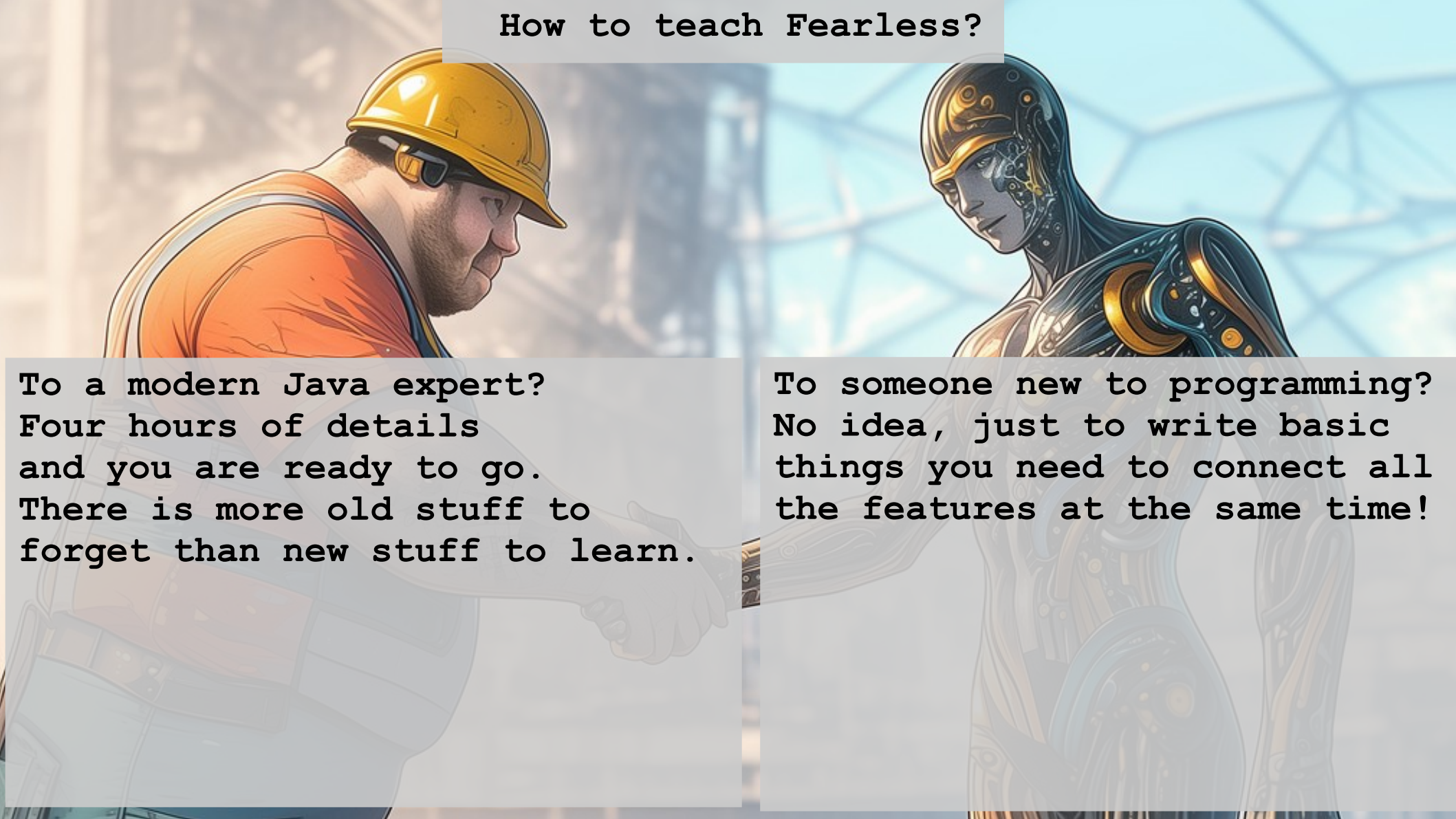
Object oriented

Functional



You need to understand both mindsets to be effective in fearless.

How to teach Fearless?

An illustration of a construction worker in a yellow hard hat and orange shirt shaking hands with a blue and gold robot. The background is a blurred construction site with scaffolding.

To a modern Java expert?
Four hours of details
and you are ready to go.
There is more old stuff to
forget than new stuff to learn.

To someone new to programming?
No idea, just to write basic
things you need to connect all
the features at the same time!

booleans slide here

Embedded DSL

- A library define terms and how to combine them. It is a language nested inside the programming language.
- An **E**mbodied **D**omain **S**pecific **L**anguage is a library embracing this idea.
- Code using this library looks like is using a different language, with different rules and conventions.
- Example: Java streams
- Languages with minimal / flexible syntax are great for EDSLs

Find the tallest!

```
TallestPerson:Comparator[Person]{p1,p2->  
  NumComparator.compare(p1.height, p2.height)  
}
```

//Declarative

```
Find-The-TallestPerson-In-myList
```

//Declarative/Relational

```
Query.findThe TallestPerson .in myList
```

//Functional/Declarative

```
myList.flow.max TallestPerson .get
```

//Imperative

```
Block#
```

```
.var res = {myList.get(0)}
```

```
.for i = {myList.range}
```

```
.do{ res.set(TallestPerson.greater(res#,myList.get(i))) }
```

```
.return {res}
```



Find the tallest!

```
TallestPerson:Comparator[Person]{p1,p2->  
  NumComparator.compare(p1.height, p2.height)  
}
```

//Declarative

```
Find-The-TallestPerson-In-myList
```

//Declarative/Relational

```
Query.findThe TallestPerson .in myList
```

//Functional/Declarative

```
myList.flow.max TallestPerson .get
```

//Imperative

```
Block#
```

```
.var res = {myList.get(0)}
```

```
.for i = {myList.range}
```

```
.do{ res.set(TallestPerson.greater(res#,myList.get(i))) }
```

```
.return {res}
```



Find the tallest!

```
TallestPerson:Comparator[Person]{p1,p2->  
  NumComparator.compare(p1.height, p2.height)  
}
```

//Declarative

```
Find-The-TallestPerson-In-myList
```

//Declarative/Relational

```
Query.findThe TallestPerson .in myList
```

//Functional/Declarative

```
myList.flow.max TallestPerson .get
```

//Imperative

```
Block#
```

```
.var res = {myList.get(0)}
```

```
.var i={1}
```

```
.loop {Block#
```

```
  .if {i#>myList.size} .break
```

```
  .do {res.set(TallestPerson.greater(res#,myList.get(i#)))}
```

```
  .continue}
```

```
.return {res}
```



Embedded DSL in Fearless

- flows
- block
- tests
- guis/forms/IQL
- regexes
- parsing
- webAPI <->

Example, some code to test

```
Fractions: F[Int,Int,Fraction]{num,den -> Fraction: {  
  .numerator: Int -> num,  
  .denominator: Int -> den,  
  .divide: Float -> num.float / (den.float),  
  .str:Str-> "Fraction["+num+", "+den+"]",  
}}
```

Fluent tests

```
TestMyApp: TestMain{sys,runner->runner
    .withReporter(runner.stdoutPrinter(sys.io))

.test TestStaysPositive: Test{Block#
    .do {Assert!(Fractions#(+5, +4).divide > 0.0)}
    .do {Assert!(Fractions#(+1, +2).divide > 0.0)}
    .done
}

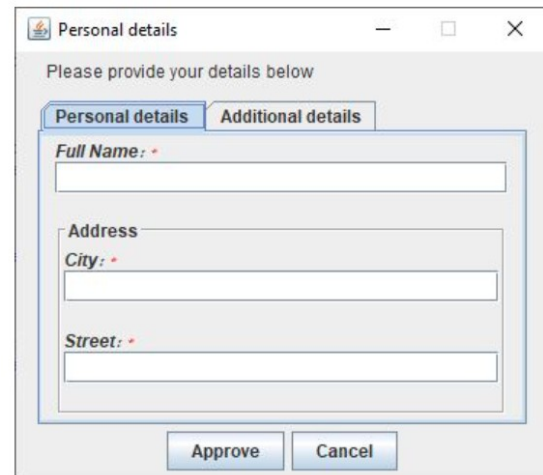
.testLog TestGetsNegative: Test{log->Block#
    .let expected= {0.0}
    .do {log#{Assert!(Fractions#(+5, -4).divide < expected)}}
    .do {log#{Assert!(Fractions#(+1, -2).divide < expected)}}
    .done
}

.test TestStrRepr: Test{Block#
    .do{"Fraction[+5,+4]".assertEq(Fractions#(+5, +4).str)}
    .done
}

.testSuite MoreTestFractions
}
```


Fluent API Forms

```
Iql
.title "Personal details"
.single "Please provide your details below"
.tab("Personal details",
  Iql.entry("name", "Full Name:", "String"),
  Iql.group("Address",
    Iql.entry("city", "City:", "String"),
    Iql.entry("street", "Street:", "String")
  )
)
.tab("Additional details",
  Iql.entry("married", "Married:", "Boolean"),
  Iql.entry("age", "Age:", "Integer"),
  Iql.entry("weight", "Weight:", "Decimal")
)
.queryUser(system) //--> gives a flow of Map[String,Data]
```



Personal details

Please provide your details below

Personal details Additional details

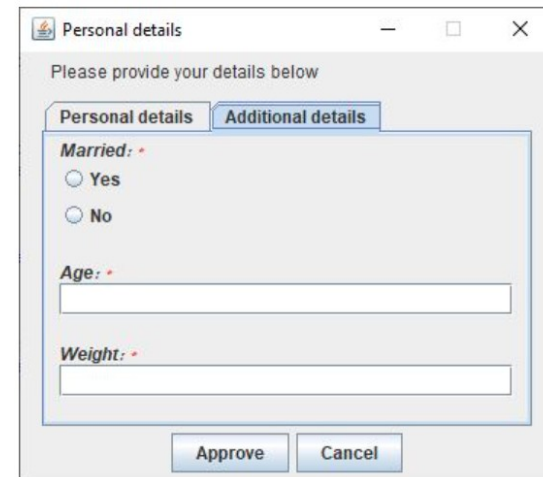
Full Name: *

Address

City: *

Street: *

Approve Cancel



Personal details

Please provide your details below

Personal details Additional details

Married: *

☐ Yes

☐ No

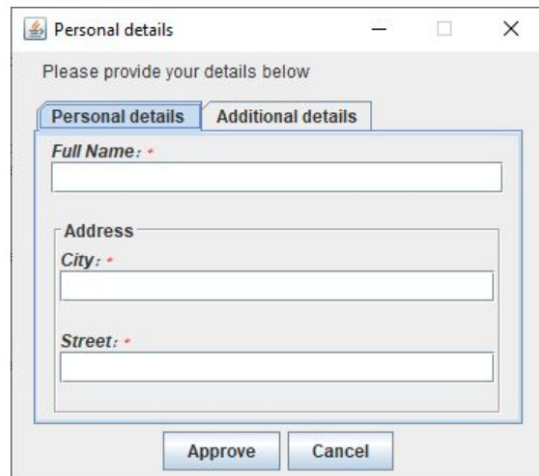
Age: *

Weight: *

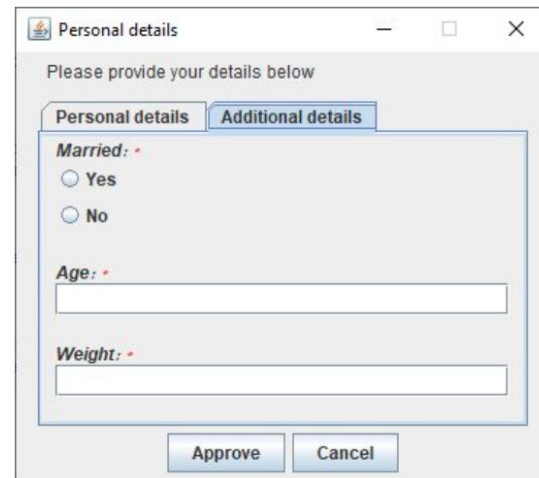
Approve Cancel

Fluent API Forms

```
Iql
.title "Personal details"
.single "Please provide your details below"
.tab "Personal details"
  .entry("name", "Full Name:", "String")
  .group "Address"
    .entry("city", "City:", "String"),
    .entry("street", "Street:", "String")
  !
!
.tab "Additional details"
  .entry("married", "Married:", "Boolean")
  .entry("age", "Age:", "Integer")
  .entry("weight", "Weight:", "Decimal")
!
.queryUser(system) //--> gives a flow of Map[String,Data]
```



The screenshot shows a window titled "Personal details" with a subtitle "Please provide your details below". It has two tabs: "Personal details" (selected) and "Additional details". The "Personal details" tab contains three text input fields: "Full Name:" (with a red asterisk), "City:" (with a red asterisk), and "Street:" (with a red asterisk). At the bottom of the window are two buttons: "Approve" and "Cancel".

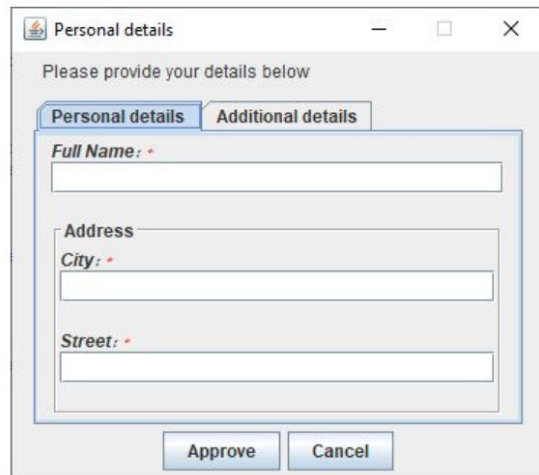


The screenshot shows the same "Personal details" window, but with the "Additional details" tab selected. This tab contains two radio button options: "Married:" (with a red asterisk) with "Yes" and "No" choices, and "Age:" (with a red asterisk) followed by a text input field. Below these is "Weight:" (with a red asterisk) followed by another text input field. The "Approve" and "Cancel" buttons remain at the bottom.

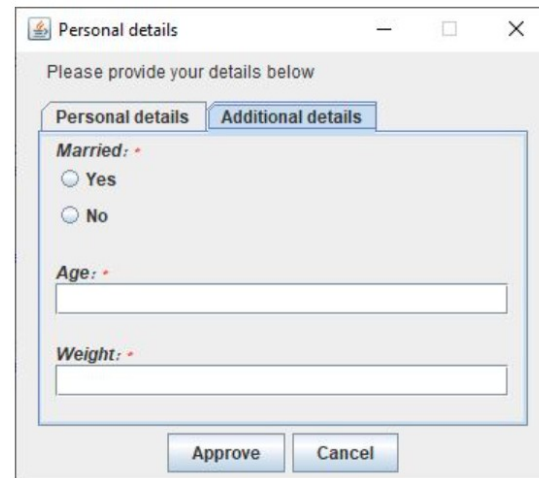
Fluent API Forms

Iql

```
.title "Personal details"
.single "Please provide your details below"
.tab{::#"Personal details"
  .entry("name", "Full Name:", "String")
  .group{::#"Address"
    .entry("city", "City:", "String"),
    .entry("street", "Street:", "String")
  }
}
.tab{::#"Additional details"
  .entry("married", "Married:", "Boolean")
  .entry("age", "Age:", "Integer")
  .entry("weight", "Weight:", "Decimal")
}
.queryUser(system) //--> gives a flow of Map[String,Data]
```



A screenshot of a web application window titled "Personal details". The window has a standard title bar with minimize, maximize, and close buttons. Below the title bar, there is a subtitle "Please provide your details below". The main content area contains two tabs: "Personal details" (which is selected) and "Additional details". Under the "Personal details" tab, there are three input fields: "Full Name:" (with a red asterisk), "City:" (with a red asterisk), and "Street:" (with a red asterisk). At the bottom of the form, there are two buttons: "Approve" and "Cancel".

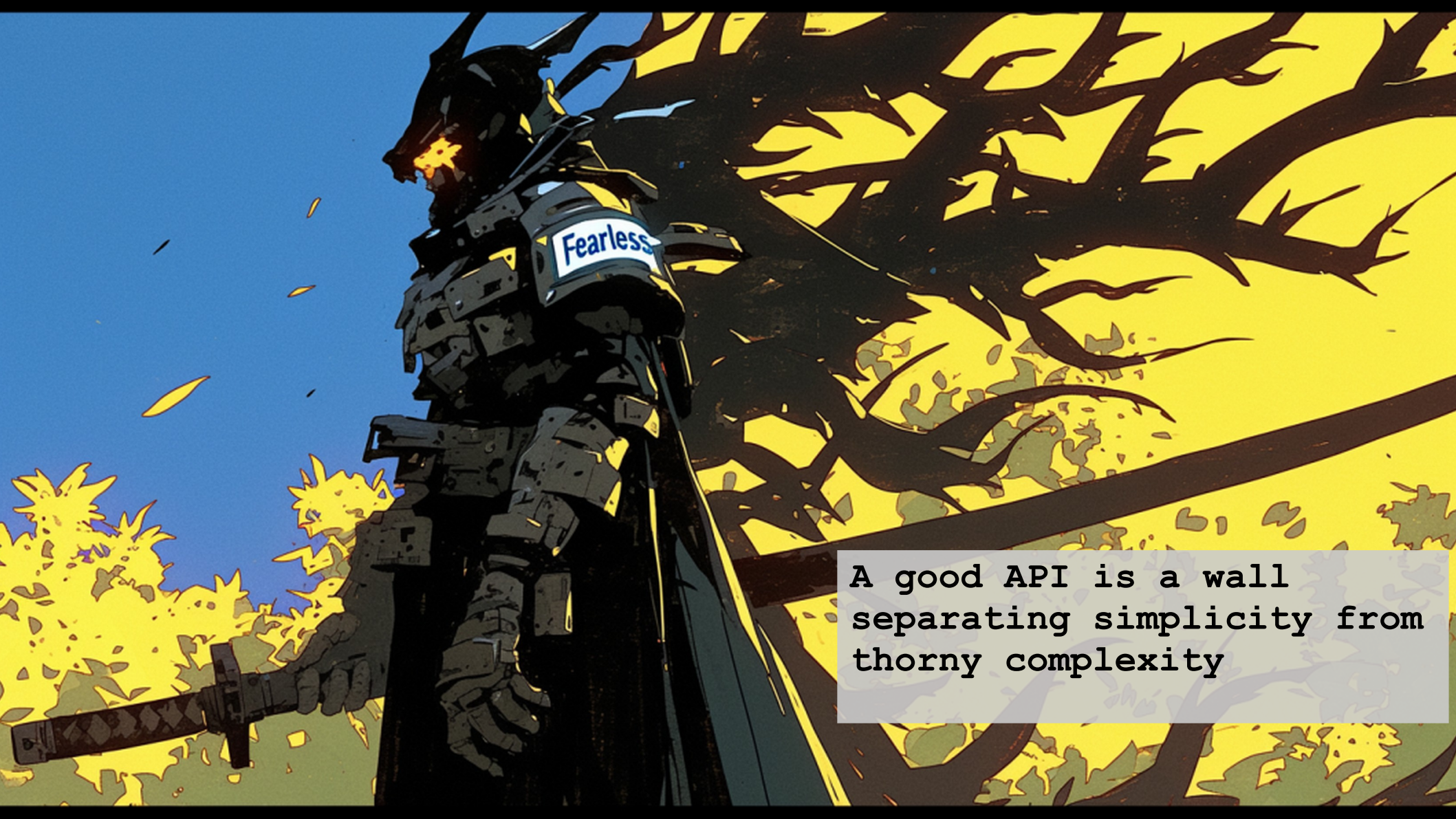


A screenshot of the same "Personal details" web application window, but with the "Additional details" tab selected. The "Personal details" tab is now greyed out. Under the "Additional details" tab, there are three input fields: "Married:" (with a red asterisk) which has two radio button options, "Yes" and "No"; "Age:" (with a red asterisk) which has a text input field; and "Weight:" (with a red asterisk) which has a text input field. At the bottom of the form, there are two buttons: "Approve" and "Cancel".

Readable API for regexes

```
.let r = Regex
.let stringLit = r.sequence(
  r.chars "\"",
  r.any(    //.any is *, .many is +, .optional is ?
    r.alternative(
      r.anyCharExcept("\"", "\\"),
      r.sequence(
        r.chars "\\",
        r.alternativeStrings("\"", "\\n", "\\t")
      )
    )
  ),
  r.chars "\"")
)
```

equivalent to the unreadable `"(?:[^\\"]|\\\\"|\\n|\\t)*"`



A good API is a wall
separating simplicity from
thorny complexity

Parsing with Flows

```
App:Main{ sys -> Block#
```

```
  .let io = {sys.io}
```

```
  .let ns = {"123\n678\n"}
```

```
  .let parsed = {Tokenize#(ns)}
```

```
  .return {io.println(parsed.flow#(Flow.str "; "))}
```

```
} //prints 123; 678
```

```
Tokenize:{#(str: Str): List[Str] -> str.flow
```

```
  .actor[mut Str, Str](iso"", {downstream, state, current -> Block#
```

```
    .if {current != "\n"}
```

```
      .return {Block#( state.add(current), ActorRes.continue)}
```

```
    .do {downstream#(state.str)}
```

```
    .do {state.clear}
```

```
    .return {ActorRes.continue}
```

```
  })
```

```
  .list
```

```
}
```