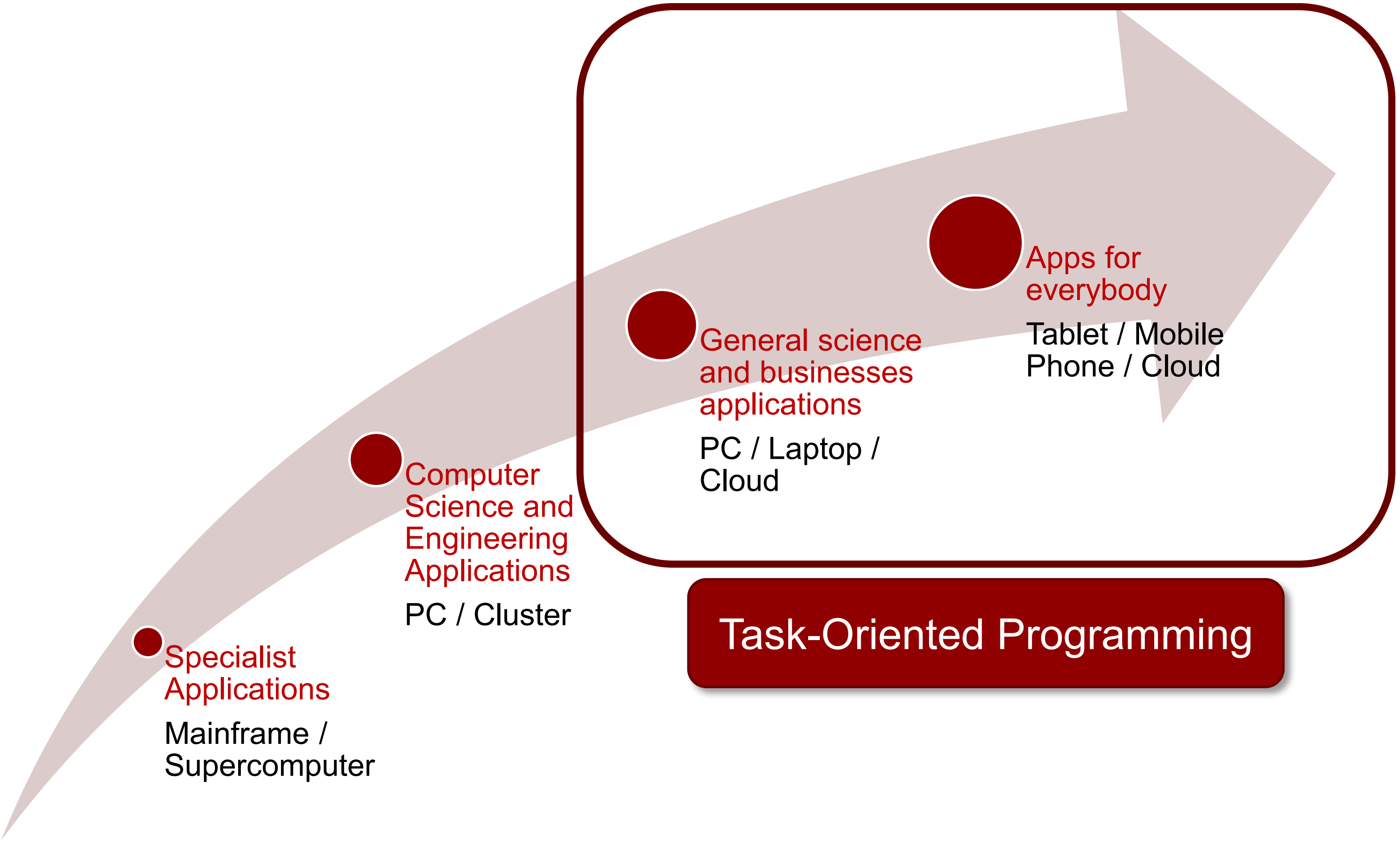
Task-Oriented Programming

Sven-Bodo Scholz, **Peter Achten**

Advanced Programming

part 1/2

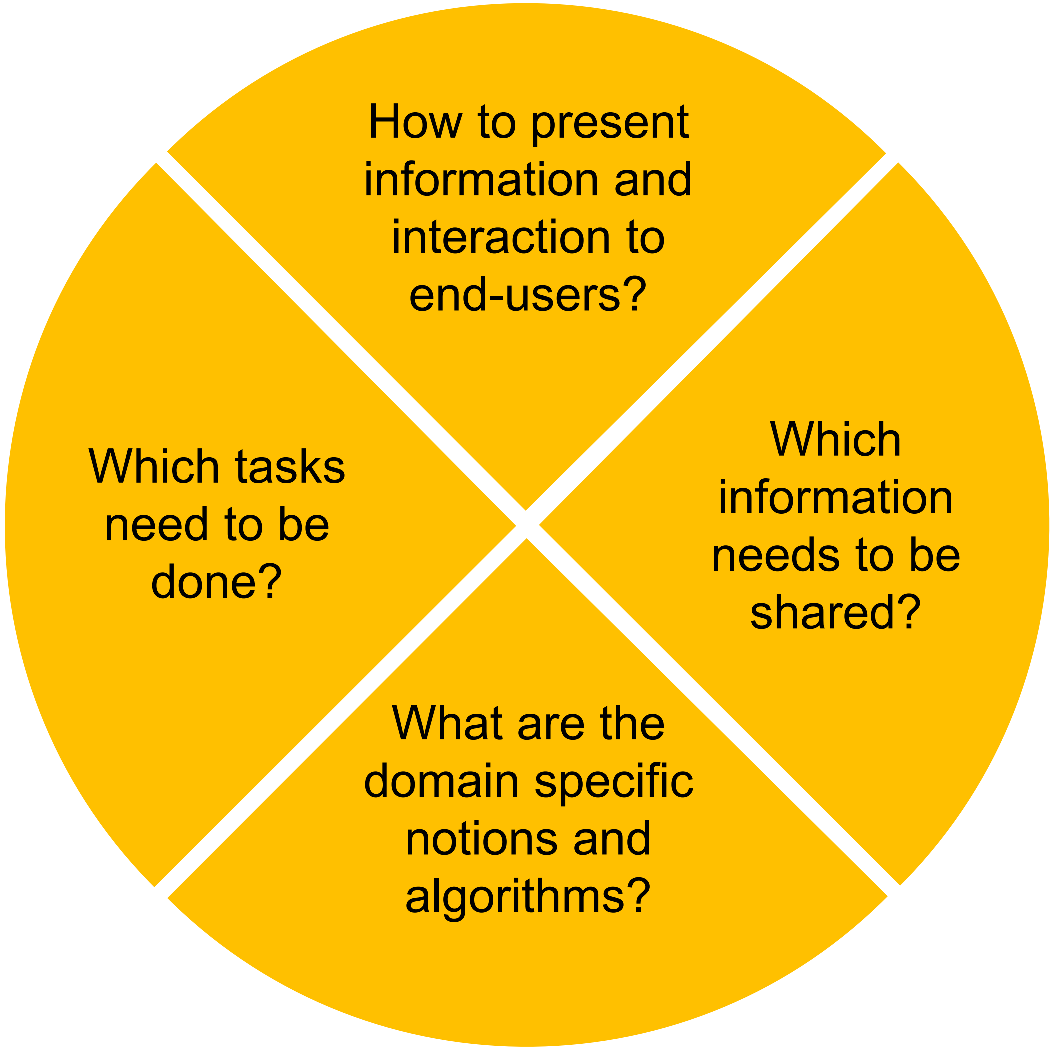
What this lecture is about

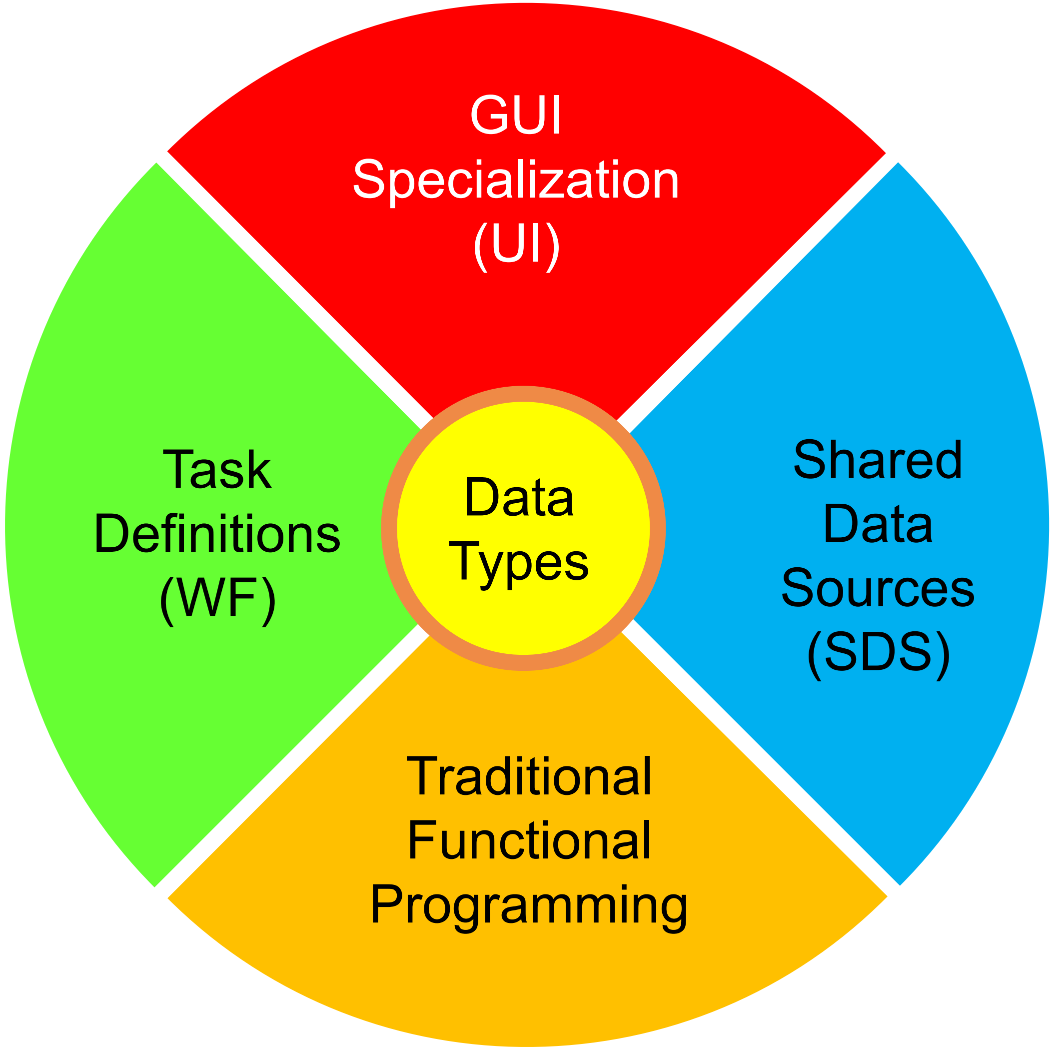


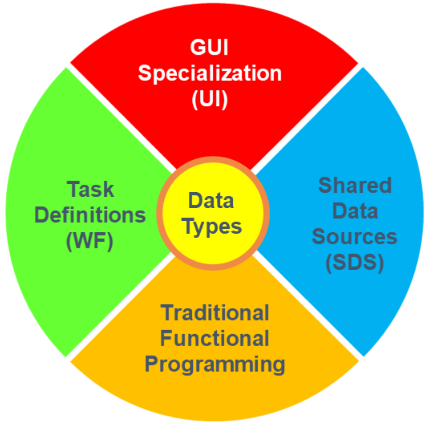
HP^3

- High Performance
  - No delay
  - Nice images
  - Sufficient information
  - Sufficiently complex
- High Productivity
  - Quickly available
  - Easy to put together for beginners
  - Stable
- High Portability
  - Run anywhere
  - Have the same performance anywhere

Task-Oriented Programming


How to present information and interaction to end-users?
Which tasks need to be done?
Which information needs to be shared?
What are the domain specific notions and algorithms?

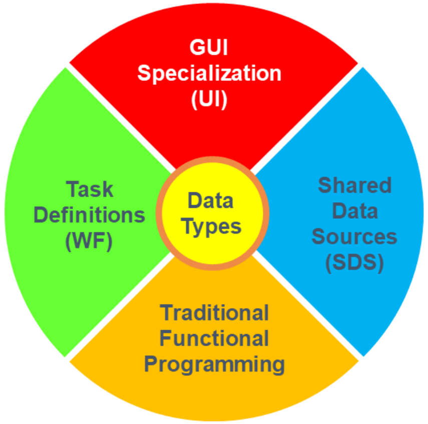

GUI
Specialization
(UI)
Task
Definitions
(WF)
Data
Types
Shared
Data
Sources
(SDS)
Traditional
Functional
Programming



Task-Oriented Software Development (TOSD)

- Separation of concerns (Stutterheim et al[2018])
  1. UoD modeling: identify the entities (data types) and their relations (functions)
  2. SDS modeling: identify existing or required sources of information
  3. Task modeling: identify the user and application tasks
  4. UI modeling: identify the required UI experience
- High Productivity, High Portability
  - Generate all code based on type information (generics)
  - Serialize any code on any platform (dynamics)

Task-Oriented Programming (TOP)



- Programming language paradigm
- Core concepts:
  - Types & Functions
  - Tasks & Combinators
  - Shared Data Sources (part 2)
- TOP implementations
  - iTask: distributed web applications in Clean (Plasmeijer et al[2007…])
  - mTask: IoT (Koopman and Lubbers[2016…])
  - µTask: non-interruptible embedded systems in Haskell (Piers[2016])
  - TopHat: formal semantics of TOP in Haskell (PhD Theses Naus[2020], Steenvoorden[2022])
  - Toppyt: Python web applications (Lijnse[2022])
  - LTask: TOP implementation in Lua (van Gemert[2022])

iTask

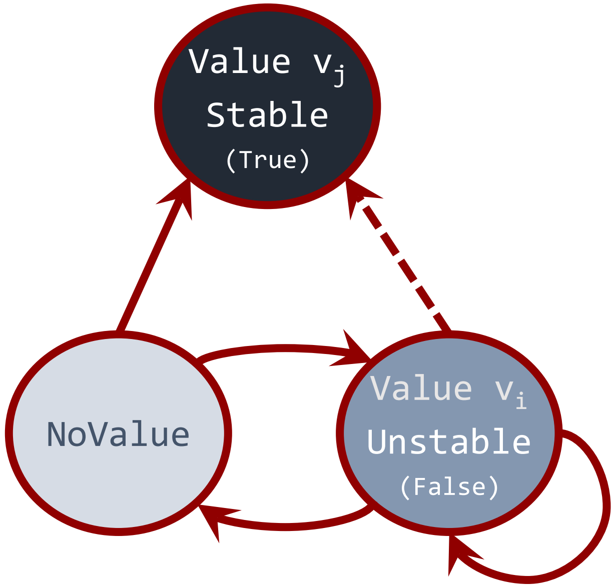
- Implementation of TOP
- Shallowly Embedded Domain Specific Language (EDSL) for applications that coordinate people and systems on the internet
- Host language of the EDSL is Clean
  - strongly typed, pure, lazy, functional programming language
  - built-in support for generic programming and dynamic types

Task-Oriented Programming (TOP)

```
:: Task a
```

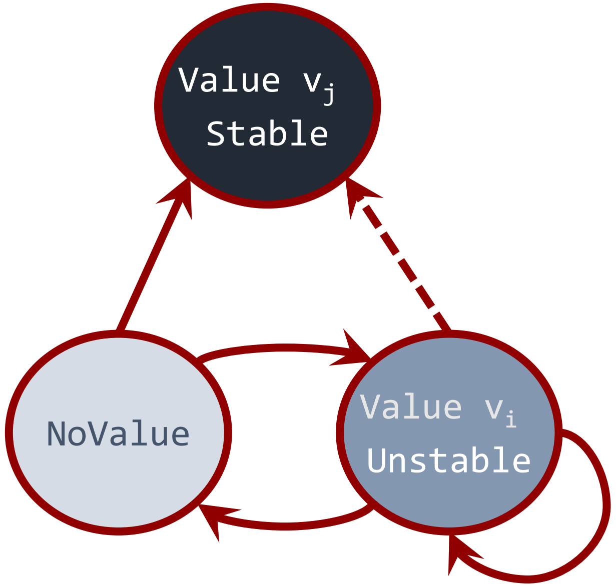
- Abstraction of work performed by human or computer
- Key paradigm design decisions:
  - typed interface of the task to its environment
  - the task controls its task value
  - the environment controls what to do with it

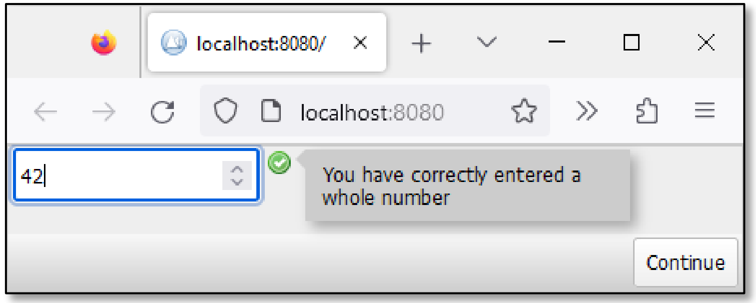
```
:: TaskValue a = NoValue
               | Value !a !Stability
:: Stability :== Bool
```

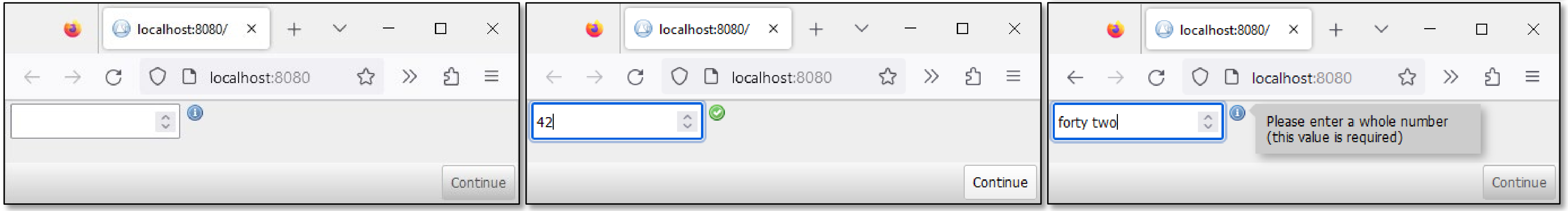


Interactive tasks, part 1

```
module example
import iTasks, StdEnv

Start :: *World -> *World
Start world = doTasks task1 world

task1 :: Task Int
task1 = enterInformation [] >>? return
```







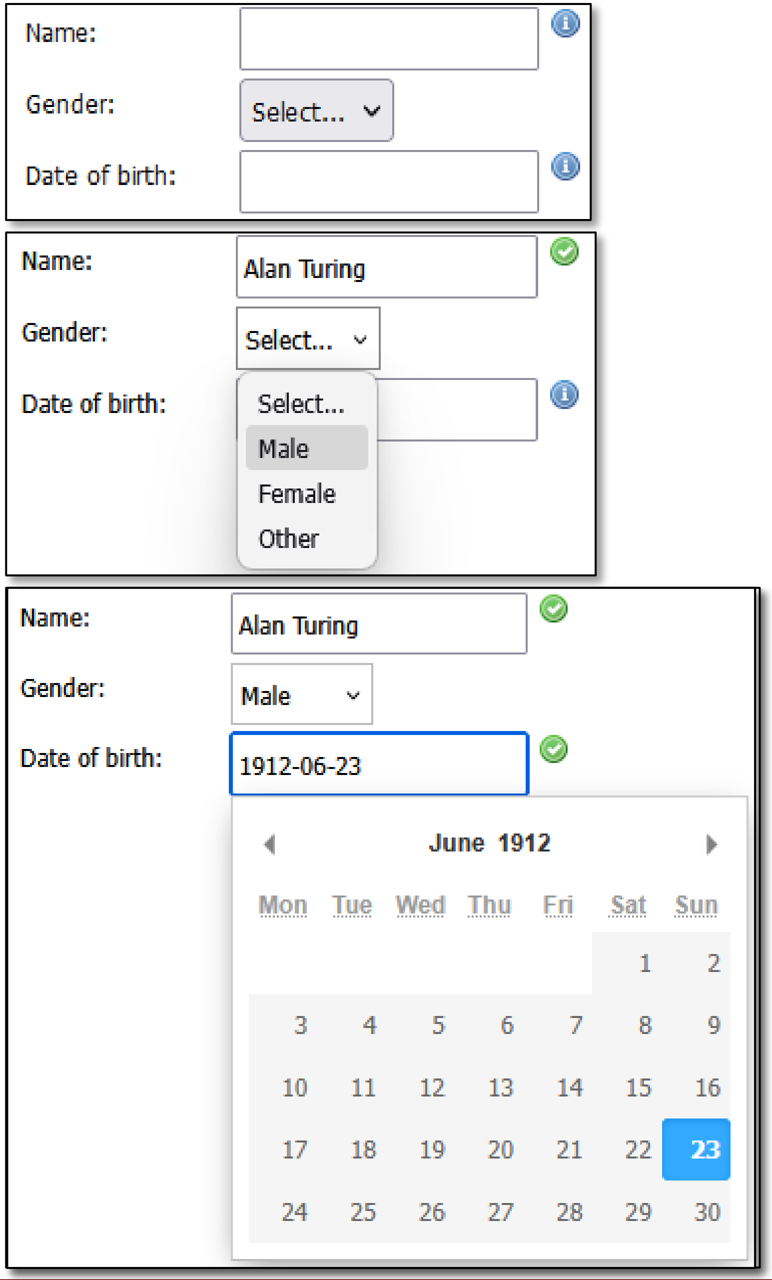
Under the hood, part 1

```
:: Task a
  =: Task (Event TaskEvalOpts *IWorld -> *(TaskResult a, *IWorld))
:: TaskResult a
   = ValueResult (TaskValue a) TaskEvalInfo UIChange (Task a)
   | ExceptionResult TaskException // something went wrong
   | DestroyedResult               // task finalizes and cleaned up
```

- Observations:
  - referential transparent: we need an event and a world in each step
  - task does not change value at arbitrary moments, only at an event

Interactive tasks, part 2

```
:: Person
 = { name        :: String
   , gender      :: Gender
   , dateOfBirth :: Date
   }
:: Gender
 = Male | Female | Other String
derive class iTask Person, Gender
import iTasks.Extensions.DateTime

task2 :: Task Person
task2 = enterInformation []
```

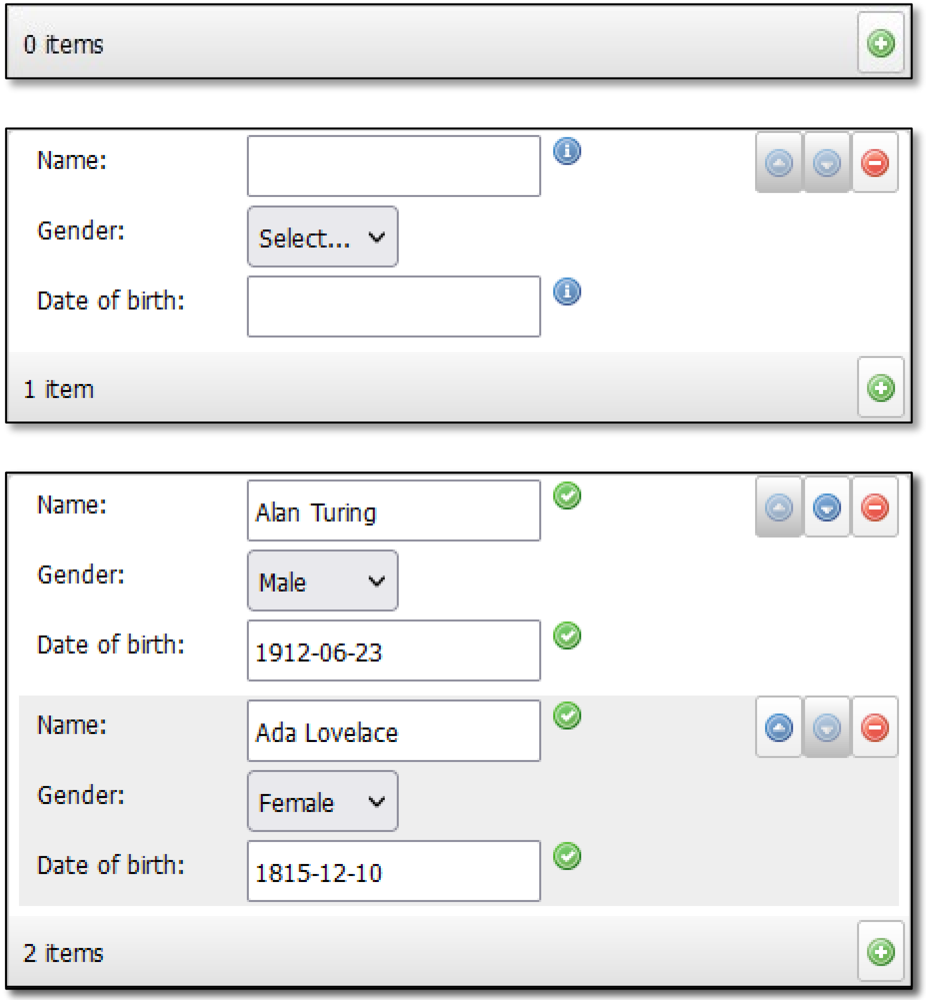


Under the hood, part 2

- Generic class `iTask`:
  - defines a collection of generic functions for the basic types and custom types
- Collection:
  - `gEditor{|*|}`: create an interactive task (view and/or update information)
  - `gText{|*|}`: ‘print’ a value in several ways
  - `JSONEncode{|*|}` and `JSONDecode{|*|}`: serialize and deserialize values for transmission via internet protocols
  - `gEq{|*|}`: compare two values for equality
  - `TC`: unlock storage and interchange of arbitrary values (including functions)
- How to use?
  1. define new types (algebraic data types or record types)
  2. add the following line for each new type `T`:
     ```
     derive class iTask T
     ```

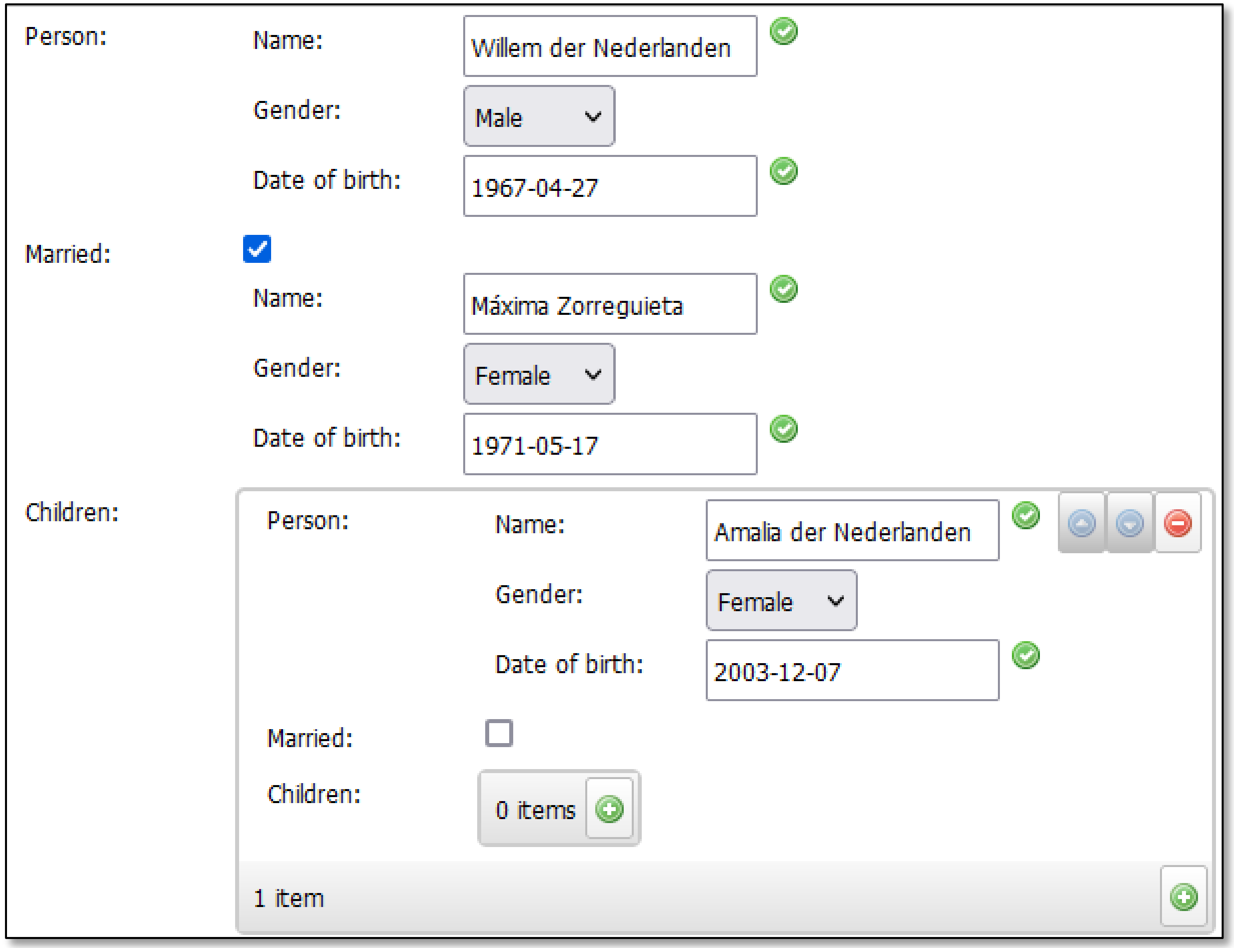
Interactive tasks, part 3

```
:: Person
 = { name        :: String
   , gender      :: Gender
   , dateOfBirth :: Date
   }
:: Gender
 = Male | Female | Other String
derive class iTask Person, Gender
import iTasks.Extensions.DateTime

task3 :: Task [Person]
task3 = enterInformation []
```



Interactive tasks, part 4

```
:: Family
 = { person   :: Person
   , married  :: ? Person
   , children :: [Family]
   }

derive class iTask Family

task4 :: Task Family
task4 = enterInformation []
```



Tuning interactive tasks

```
(<<@) infixl 2 :: !item !option -> item | tune option item
(@>>) infixr 2 :: !option !item -> item | tune option item

instance tune Title (Task a)
instance tune Hint  (Task a)
instance tune Label (Task a)
instance tune Icon  (Task a)
```

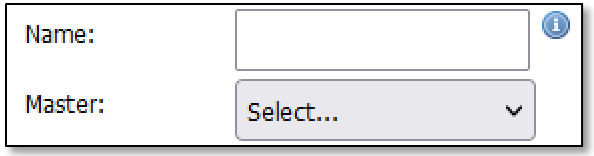
iTasks.UI.Tune

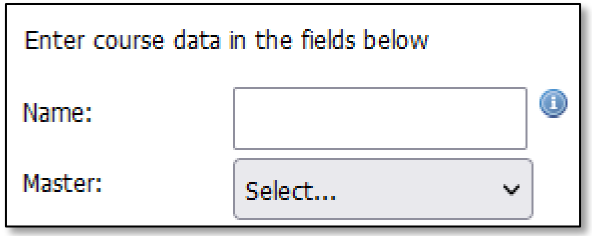
```
:: Title = Title !String
:: Hint  = Hint  !String
:: Label = Label !String
:: Icon  = Icon  !String
```

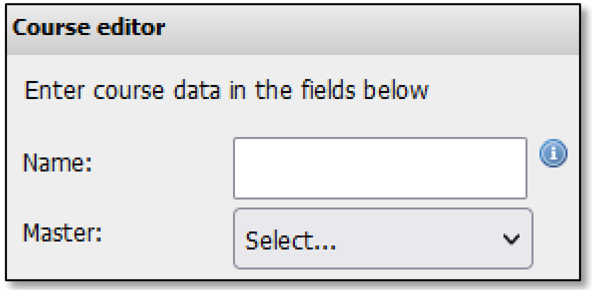
iTasks.UI.Definition

Interactive tasks, part 5

```
:: Course
 = { name   :: String
   , master :: Bool
   }

derive class iTask Course

task5 :: Task Course
task5 = enterInformation []
        <<@ Hint "Enter course data in the fields below"
        <<@ Title "Course editor"
```



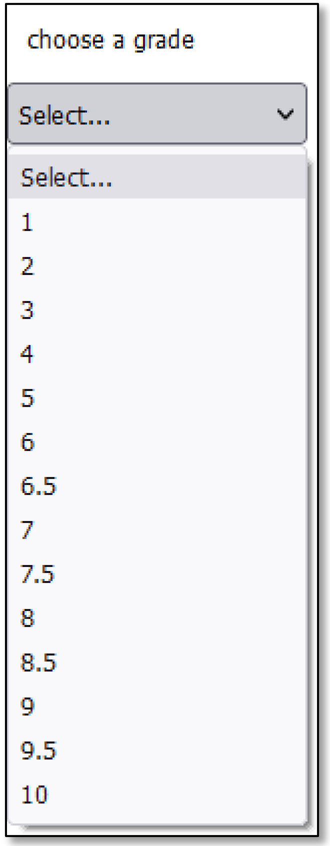




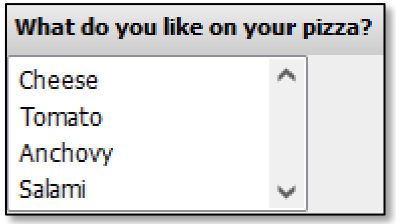
Interactive tasks, part 6

- Module iTasks.WF.Tasks.Interaction offers many interactive tasks

```
chooseGrade :: Task Real
chooseGrade = enterChoice [] ([1.0..5.0] ++ [6.0,6.5..10.0])
                <<@ Hint "choose a grade"
```

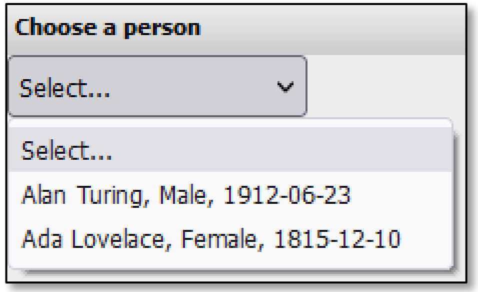



```
pizzaWith :: Task [String]
pizzaWith = enterMultipleChoice []
                ["Cheese", "Tomato", "Anchovy", "Salami"]
            <<@ Title "What do you like on your pizza?"
```



Tasks first, fine-tune later

```
choosePerson1 :: [Person] -> Task Person
choosePerson1 persons
 = enterChoice [] persons
   <<@ Title "Choose a person"
```




```
choosePerson2 :: [Person] -> Task Person
choosePerson2 persons
 = enterChoice [ChooseFromGrid id] persons
```

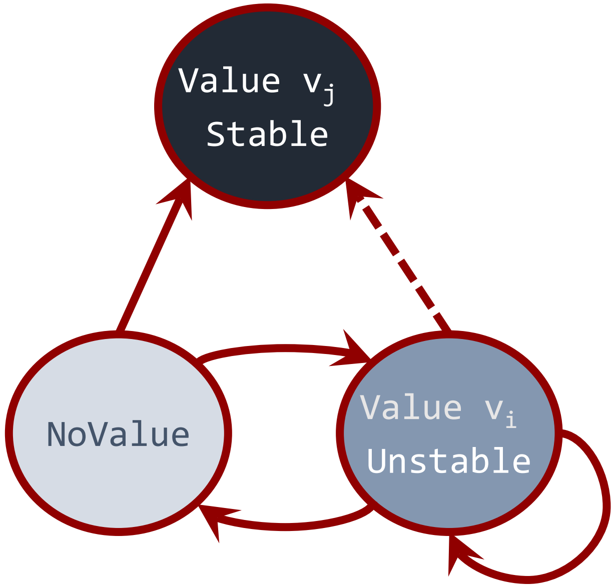

```
choosePerson3 :: [Person] -> Task Person
choosePerson3 persons
 = enterChoice [ChooseFromCheckGroup (\p = p.Person.name)] persons
```

Task Combinators

- Combine tasks to new task
- Atomic
  - `return` (`iTasks.WF.Tasks.Core`)
  - `@` (`iTasks.WF.Combinators.Common`)
- Sequential
  - `step` (`iTasks.WF.Combinators.Core`)
  - derived combinators such as `>>*`, `>>?`, `>>-`, `>>|`, …
  - similar to monadic operations, but with side-effects in the GUI
- Parallel
  - `parallel` (`iTasks.WF.Combinators.Core`)
  - derived combinators such as `-&&-`, `-||-`, `-||`, `||-`, …
- ‘Do it yourself’

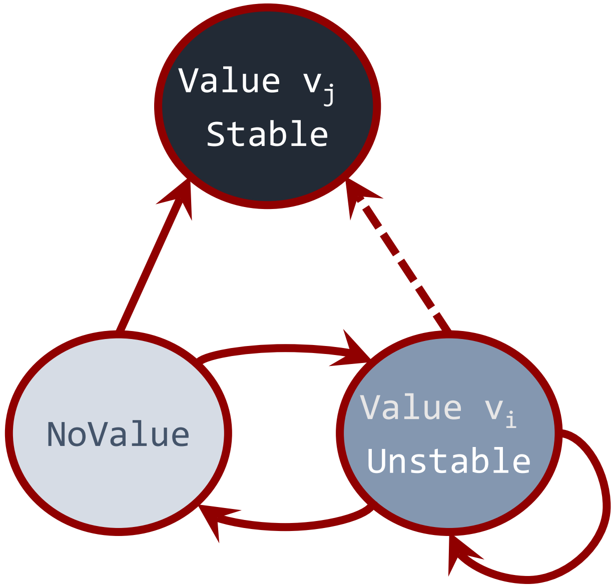
Atomic task combinator: return

```
return :: !a -> Task a
```



Atomic task combinator: transform

```
(@)  infixl 1 :: (Task a) (a -> b) -> Task b
(@!) infixl 1 :: (Task a) b -> Task b
```



Sequential task combinators: step

- Core combinator:

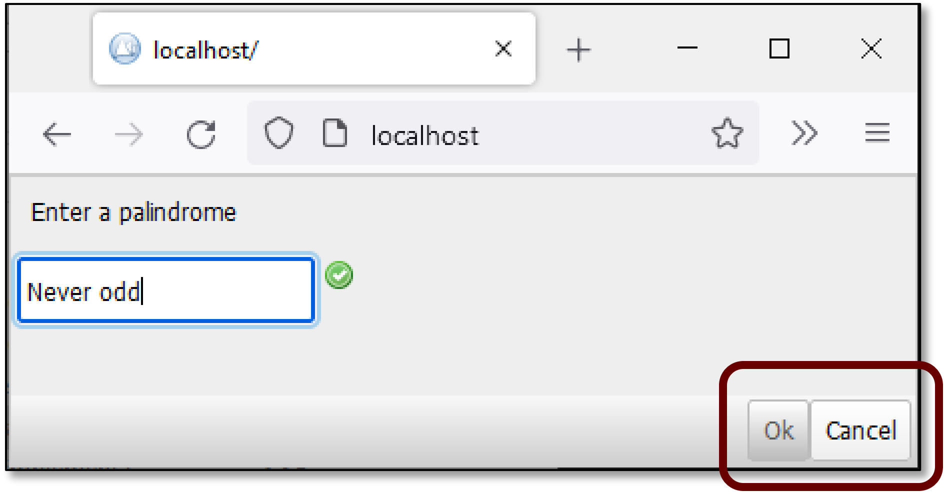
```
step           :: !(Task a) ((?a) -> ?b) [TaskCont a (Task b)] -> Task b
                | TC, JSONEncode{|*|} a

(>>*) infixl 1 :: !(Task a) ![TaskCont a (Task b)] -> Task b
                | TC, JSONEncode{|*|} a
(>>*) task steps = step task (const ?None) steps

:: TaskCont a t
 =      OnValue          ((TaskValue a) -> ?t)
 |      OnAction Action  ((TaskValue a) -> ?t)
 | E.e: OnException      (e      ->  t) & iTask e
 |      OnAllExceptions  (String ->  t)
```

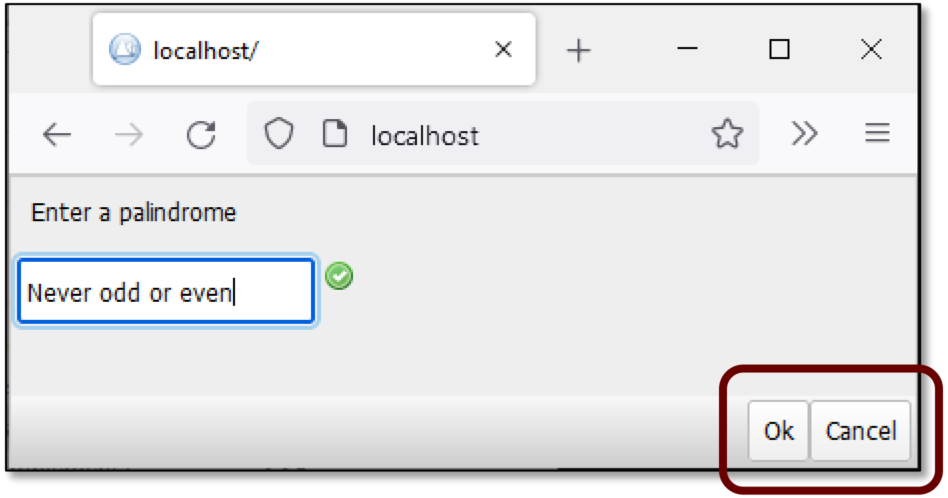
module iTasks.WF.Combinators.Common defines convenience functions, e.g.:

- `ifValue`
- `hasValue`
- `ifStable`
- `ifUnstable`

Sequential task composition

```
palindrome :: Task (?String)
palindrome
 = Hint "Enter a palindrome" @>>
   enterInformation [] >>*
     [ OnAction ActionOk
         (ifValue isPalindrome (return o ?Just))
     , OnAction ActionCancel
         (always                (return ?None))
     ]

isPalindrome :: String -> Bool
isPalindrome txt = chars == reverse chars
where
   chars = [toLower c \\ c <-: txt | not (isSpace c)]
```





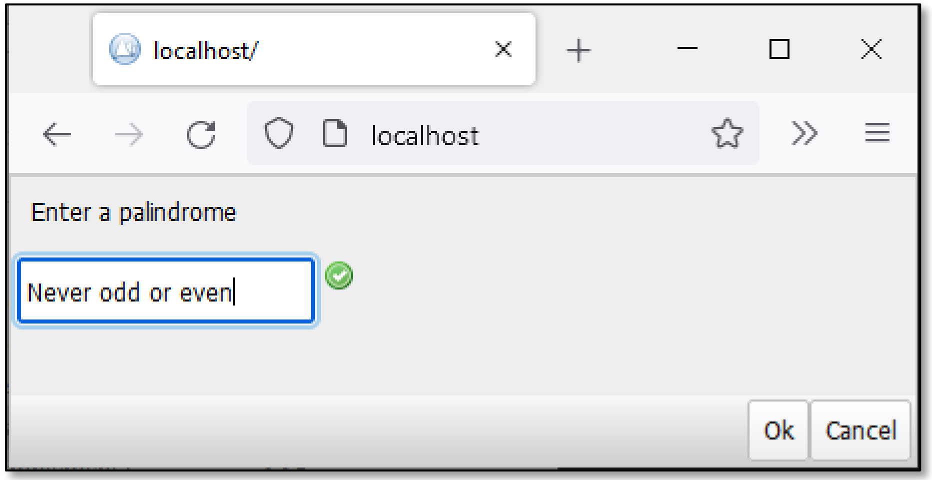
Sequential task composition

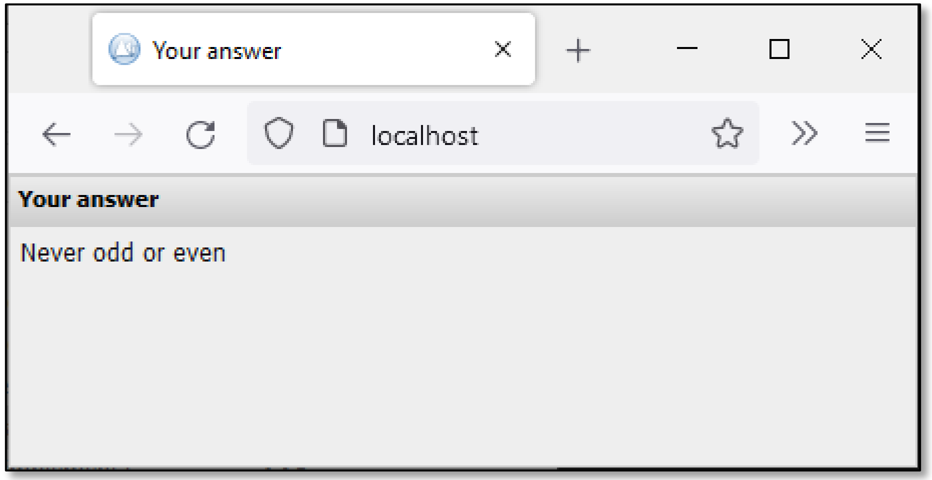
- Module iTasks.WF.Combinators.Common offers many step variations, e.g.:

```
(>>-) infixl 1 :: !(Task a) !(a -> Task b) -> Task b | TC, JSONEncode{|*|} a
(>>-) taska taskbf = taska >>* [OnValue  (ifStable taskbf)]

(>>?) infixl 1 :: !(Task a) !(a -> Task b) -> Task b | TC, JSONEncode{|*|} a
(>>?) taska taskbf = taska >>* [OnValue  (ifStable taskbf)
                               ,OnAction ActionContinue (hasValue taskbf)
                               ]

(>?|) infixl 1 :: !(Task a) !(Task b) -> Task b | TC, JSONEncode{|*|} a
(>?|) taska taskb  = taska >>* [OnValue  (ifStable (\_ = taskb))
                               ,OnAction ActionContinue (always taskb)
                               ]
```

Sequential task composition

```
task6 :: Task (?String)
task6
 =          palindrome
   >>- \s = viewInformation [] s <<@ Title "Your answer"
```





Parallel task combinator

- Core combinator:

```
parallel :: ![(ParallelTaskType,ParallelTask a)]
             [TaskCont [(Int,TaskValue a)]
             (ParallelTaskType,ParallelTask a)]
         -> Task [(Int,TaskValue a)] | iTask a
:: ParallelTaskType
   = Embedded
   | Detached !Bool !TaskAttributes
:: ParallelTask a
 :== (SharedTaskList a) -> Task a
```

Parallel task composition

- Module iTasks.WF.Combinators.Common offers many parallel variations, e.g.:
- ‘and’: return values of all (embedded) parallel tasks:

```
allTasks           :: [Task a]          -> Task [a]          | iTask a
(-&&-) infixr 4    :: (Task a) (Task b) -> Task (a, b)       | iTask a & iTask b
```

- ‘or’: return result of (embedded) parallel tasks yielding a value as first:

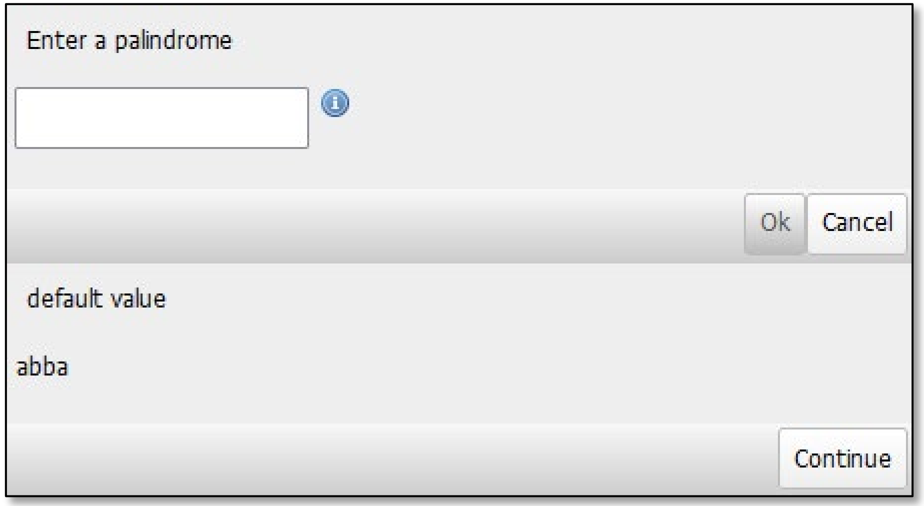
```
anyTask            :: [Task a]          -> Task a            | iTask a
(-||-) infixr 3    :: (Task a) (Task a) -> Task a            | iTask a
eitherTask         :: (Task a) (Task b) -> Task (Either a b) | iTask a & iTask b
```

- ‘one-of’: start two tasks, use the result of one of them:

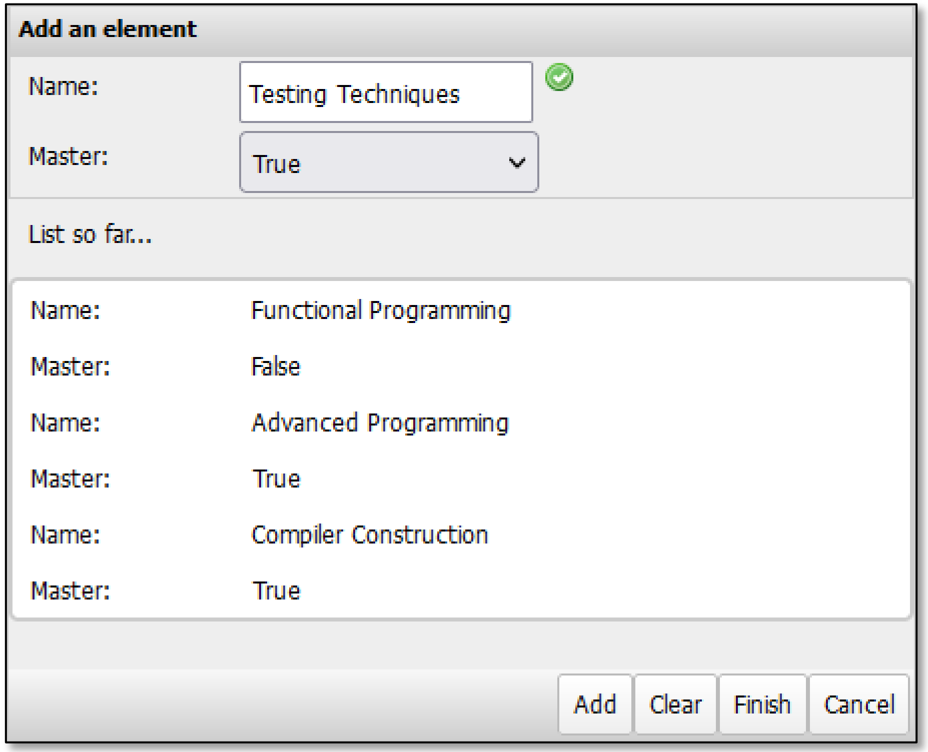
```
(-||)  infixl 3    :: (Task a) (Task b) -> Task a            | iTask a & iTask b
(||-)  infixr 3    :: (Task a) (Task b) -> Task b            | iTask a & iTask b
```

Parallel task composition

```
task7 :: Task (?String)
task7 = (  palindrome
             -||-
           (Hint "default value" @>>
            viewInformation [] "abba" >>?
            return o ?Just
           )
        )
        >>- viewInformation []
```



Deploy iTask EDSL



- Build your own abstractions
- Use recursion, higher-order functions, …

```
add1by1 :: (a [a] -> [a]) [a] -> Task [a] | iTask a
add1by1 f as
 =      (Title "Add an element" @>> enterInformation [])
    -|| (Hint "List so far..."  @>> viewInformation  [] as)
    >>* [ OnAction (Action "Add")    (hasValue (\a = add1by1 f (f a as)))
        , OnAction (Action "Clear")  (always   (add1by1 f []))
        , OnAction (Action "Finish") (always   (return as))
        , OnAction  ActionCancel     (always   (return []))
        ]
```

requires `< Course`

```
task8 :: Task [Course]
task8 = add1by1 (\a as = sort [a:as]) []
```

```
import Data.GenLexOrd
derive gLexOrd Course
instance  < Course
  where (<) c1 c2 = gLexOrd{|*|} c1 c2 === LT
```

What have we done?

- Key points:
  - HP^3 for web applications
  - TOP: a conceptual look at web applications
  - iTask: an implementation of TOP, shallowly embedded in host language Clean
- Tasks:
  - User interaction: (`view`/`enter`/`update`)`Information`
  - Atomic tasks: `return` and transform (`@`, `@!`)
  - Sequential composition: `step` (`>>*`) and friends (`>>-`, `>>?`, `>?|`)
  - Parallel composition: `parallel` and friends (`allTasks`, `-&&-`, `anyTask`, `-||-`, `-||`, `||-`)
- Next lecture:
  - Shared Data Sources
  - Distributed Programming (multi-user)
  - Guest lecture: industrial case study of TOP