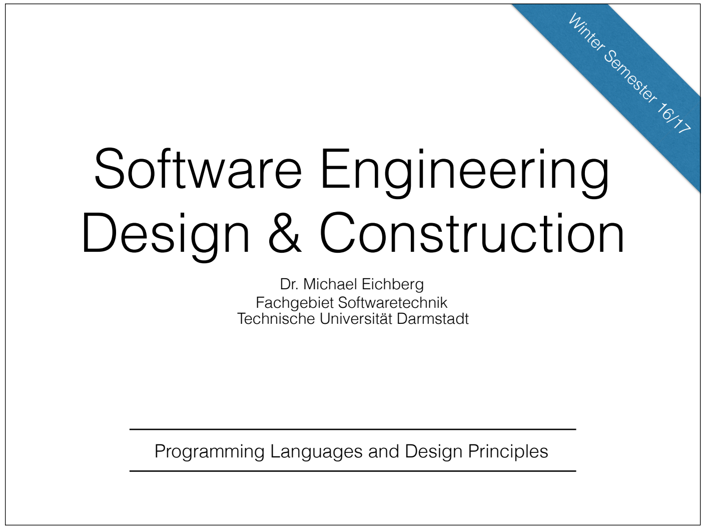


Winter Semester 16/17

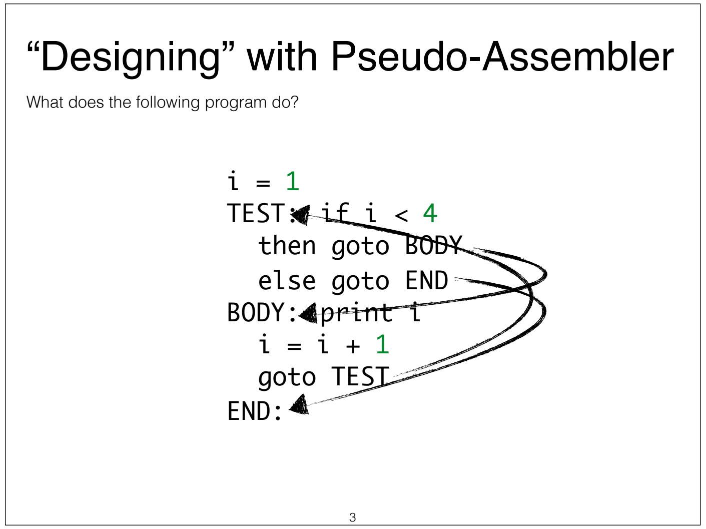
# Software Engineering Design & Construction

Dr. Michael Eichberg
Fachgebiet Softwaretechnik
Technische Universität Darmstadt

Programming Languages and Design Principles

In the following, we will discuss the development of programming languages as a means to improve their ability to capture the software design at ever increasing abstraction levels. Or, from another point of view, we discuss why advances in programming language technology are driven by the need to make programming languages capable of capturing higher-level designs.

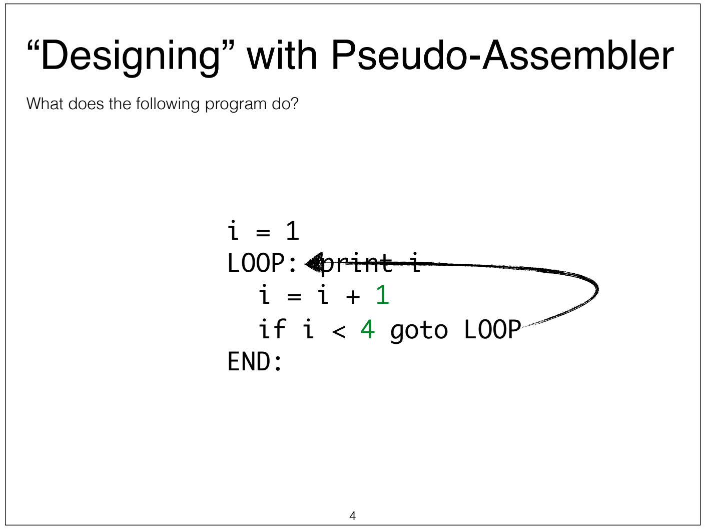
## Making Code Look Like Design



## “Designing” with Pseudo-Assembler

What does the following program do?

```
i = 1
TEST: if i < 4
  then goto BODY
  else goto END
BODY: print i
  i = i + 1
  goto TEST
END:
```



## “Designing” with Pseudo-Assembler

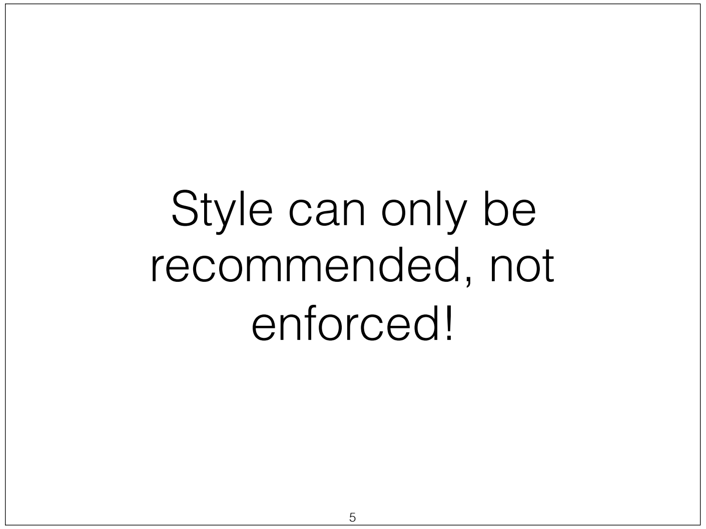
What does the following program do?

```
i = 1
LOOP: print i
  i = i + 1
  if i < 4 goto LOOP
END:
```

Though both programs just print out "123" the second one is easier to read and comprehend. It has a better style:

- Clear structure
- No crossing gotos
- Better names
- Code structure closer to what we want to express.
  "Print out i, i smaller than 4“

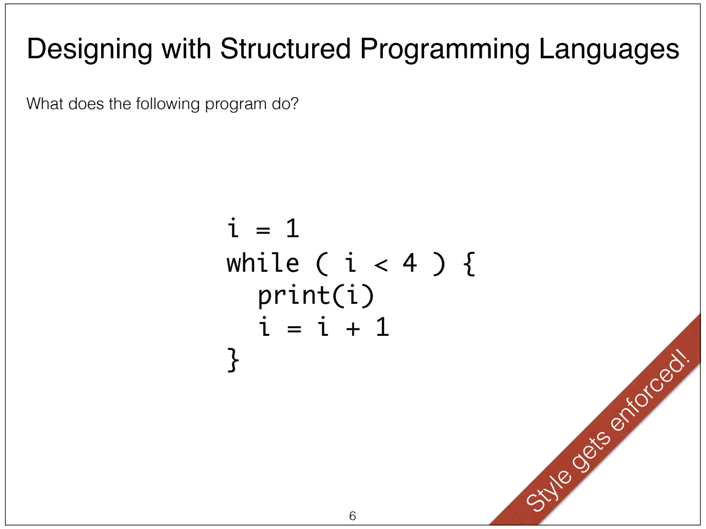
Hence, the second variant, though functionally identical, is easier to understand, debug, change.



Style can only be recommended, not enforced!

5

Enforcing a specific style is always very laborious and (without proper tool support) most often fails for large(r) (distributed)   groups.



## Designing with Structured Programming Languages

What does the following program do?

```
i = 1
while ( i < 4 ) {
  print(i)
  i = i + 1
}
```

Style gets enforced!

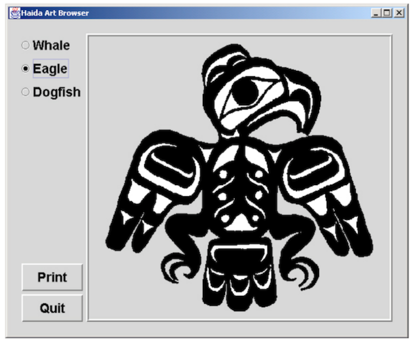
6

In the 1960th programming language support for better structuring of code emerged. **`Goto`s were replaced by loops (`while`)** and conditionals (`if`/`else`). Furthermore, procedures were introduced to support user-defined abstractions.

New words, new grammars, new abstractions enable developers to directly express looping/conditional computations, instead of emulating them by jumps. Using a – by then – modern structured programming language, it was no longer possible  to write crossing `goto`s!

## Better languages, More challenging tasks…

A simple image browser with structured programming



## Code for Image Browser Structured into Procedures

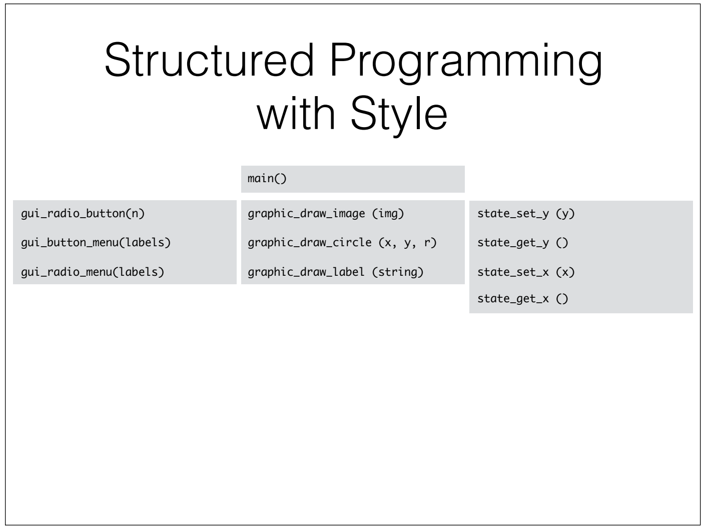
Try to identify which method calls which method!

```
main () {
draw_label(“Art Browser”)
   m = radio_menu(
      {“Whale”, “Eagle”,
       “Dogfish”})
   q = button_menu({“Quit”})
   while ( !check_buttons(q) ) {
      n = check_buttons(m)
      draw_image(n)
   }
}

set_x (x) {
   current_x = x
}

draw_circle (x, y, r) {
   %%primitive_oval(x, y, 1, r)
}

set_y (y) {
   current_y = y
}
```

```
radio_menu(labels) {
   i = 0
   while (i < labels.size) {
      radio_button(i)
      draw_label(labels[i])
      set_y(get_y()
         + RADIO_BUTTON_H)
      i++
   }
}

radio_button (n) {
   draw_circle(get_x(),
      get_y(), 3)
}

get_x () {
   return current_x
}

get_y () {
   return current_y
}
```

```
draw_image (img) {
   w = img.width
   h = img.height
   do (r = 0; r < h; r++)
      do (c = 0; c < w; c++)
         WINDOW[r][c] = img[r][c]
}

button_menu(labels) {
   i = 0
   while (i < labels.size) {
      draw_label(labels[i])
      set_y(get_y()
         + BUTTON_H)
      i++
   }
}

draw_label (string) {
   w = calculate_width(string)
   print(string, WINDOW_PORT)
   set_x(get_x() + w)
}
```

In this case, the code is structured, but the procedures are not! It is hard, if not nearly impossible, to maintain or even extend this code.



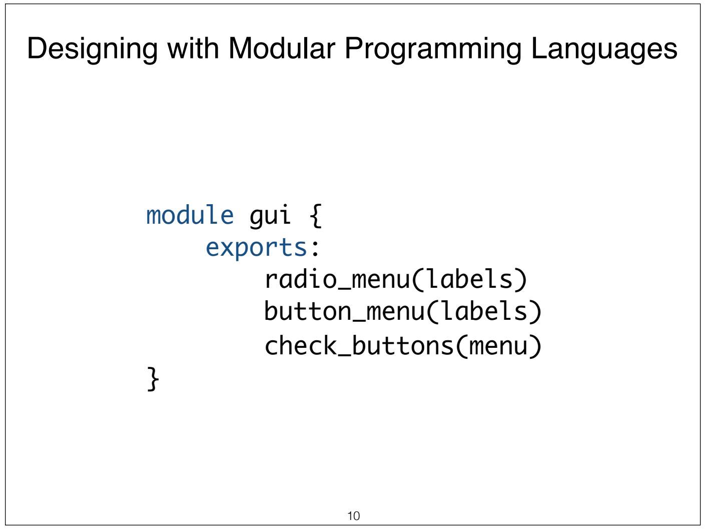
## Structured Programming with Style

```
                              main()

gui_radio_button(n)           graphic_draw_image (img)        state_set_y (y)
gui_button_menu(labels)       graphic_draw_circle (x, y, r)   state_get_y ()
gui_radio_menu(labels)        graphic_draw_label (string)     state_set_x (x)
                                                              state_get_x ()
```

Group procedures by the functionality they implement and the state they access, e.g. by naming conventions …

Advantages:

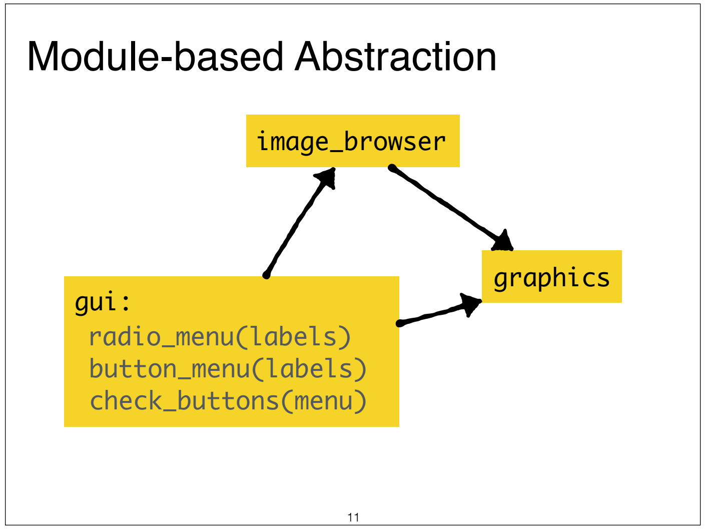
- The code is closer to what we want to express.
  "main calls gui, gui calls graphic to draw, …"
- The code is easier to understand, debug and change.



## Designing with Modular Programming Languages

```
module gui {
    exports:
        radio_menu(labels)
        button_menu(labels)
        check_buttons(menu)
}
```

Modular programming introduced modules, higher-level units/modules introduce higher-level abstractions! One can handle a whole module as if it was its interface.
Programming language mechanisms for supporting information hiding: interface hides module internals.



## Module-based Abstraction

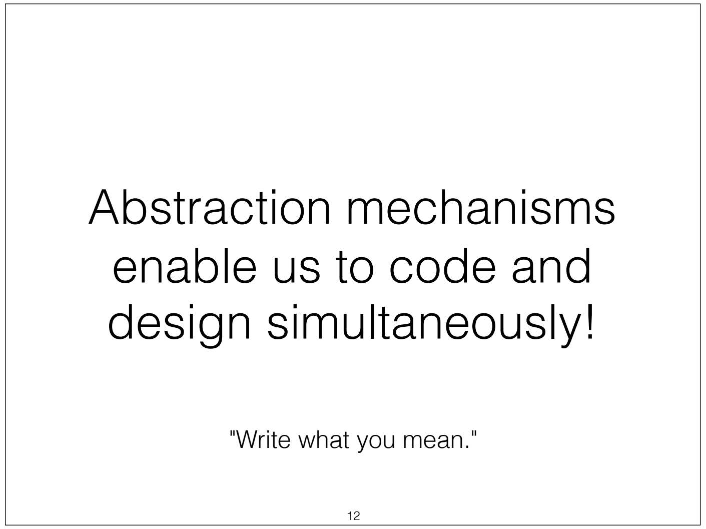
$$\begin{tikzcd}
 & \text{image\_browser} \arrow[dr] & \\
\text{gui: radio\_menu(labels), button\_menu(labels), check\_buttons(menu)} \arrow[ur] \arrow[rr] & & \text{graphics}
\end{tikzcd}$$

```
gui:
  radio_menu(labels)
  button_menu(labels)
  check_buttons(menu)
```

Abstraction enables us to:

- look at the overall structure of the system (architectural thinking).
- zoom in on individual units as needed
- with more or less details

Hence, abstraction is the key to managing complexity.

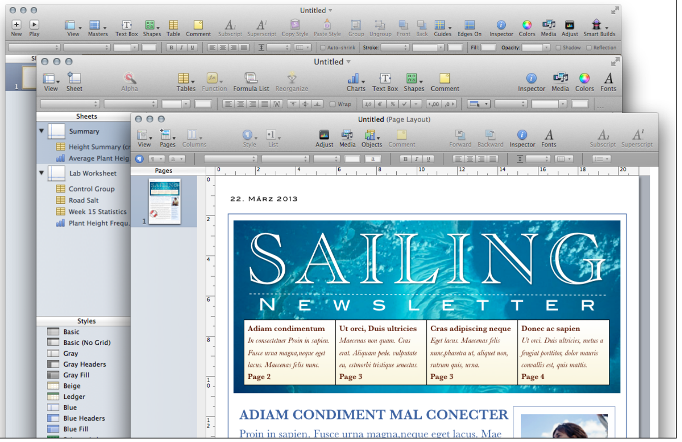


Abstraction mechanisms enable us to code and design simultaneously!

"Write what you mean."

- Makes the code easier to understand, debug and change.
- Allows structured organization of code.
- Ability to ignore details.
  Makes the code closer to what we want to express.

## Let's "develop" application families with sophisticated GUIs with uniform look and feel with modular programming…



Modeling variability with modular programming languages appeared complex…

## Designing with Object-Oriented Programming Languages

Object-oriented programming languages introduce new abstraction mechanisms:

- classes
- inheritance
- subtype polymorphism
- virtual methods

(Still) Dominating Programming Paradigm

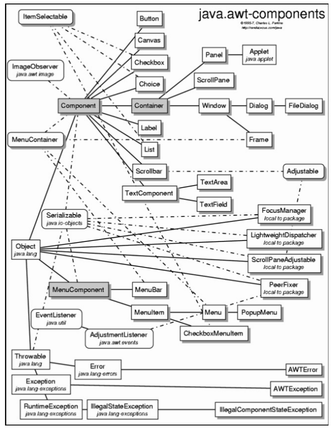
The roots of object-oriented programming languages are in the sixties.

Allan Kay,
Smalltalk 70 - 80

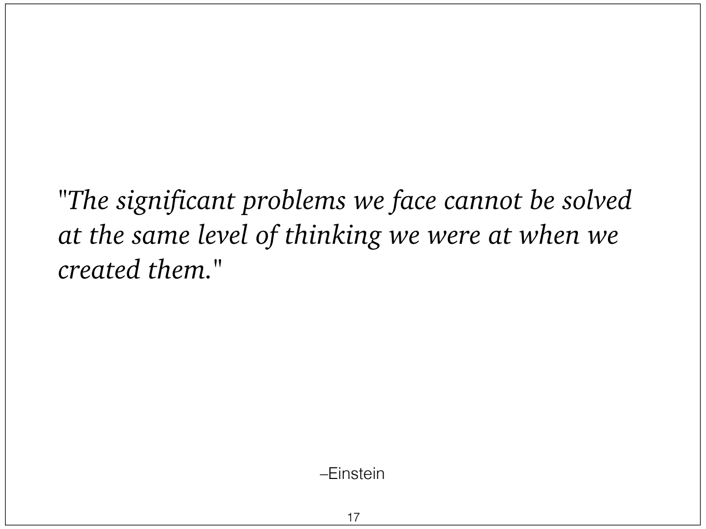
Dahl and Nygaard,
Simula 64, 68

- Object-oriented languages are popular because they make it **easier to design software and program at the same time**.
- They allow us to more **directly express high level information** about design components abstracting over differences of their variants.
- Make it easier to produce the design, and easier to refine it later.
- With stronger **type checking**, they also help the process of detecting design errors.
- Result in a more **robust design**, in essence a better engineered design.

## Programming Languages are not a Panacea


java.awt-components
©1995-7, Charles L. Perkins
http://rendezvous.com/java
ItemSelectable
ImageObserver java.awt.image
Component
MenuContainer
Button
Canvas
Checkbox
Choice
Container
Label
List
Scrollbar
TextComponent
TextArea
TextField
Panel
Applet java.applet
ScrollPane
Window
Dialog
FileDialog
Frame
Adjustable
Serializable java.io-objects
Object java.lang
FocusManager local to package
LightweightDispatcher local to package
ScrollPaneAdjustable local to package
PeerFixer local to package
MenuComponent
MenuBar
MenuItem
Menu
PopupMenu
CheckboxMenuItem
EventListener java.util
AdjustmentListener java.awt.events
Throwable java.lang
Error java.lang-errors
AWTError
Exception java.lang-exceptions
AWTException
RuntimeException java.lang-exceptions
IllegalStateException java.lang-exceptions
IllegalComponentStateException

- Accessibility of object-oriented programming drives more complex designs!
- Programming languages are powerful tools, but cannot and will never guarantee good designs.
- Programming always needs to be done properly to result in good code.
- Human *creativity* remains the main factor.

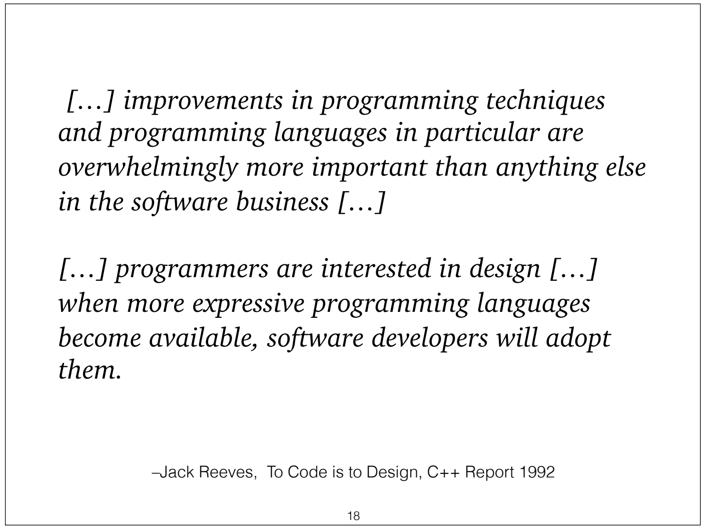


*"The significant problems we face cannot be solved at the same level of thinking we were at when we created them."*

–Einstein

E.g., trying to fix security problems in programs written in language X which are (conceptually) due to the design of language X is not going to work.

Another example are issues related to memory access, such as out-of memory issues, access violations or buffer overflows are to a great extend solved by avoiding/hiding pointers and by introducing automatic garbage collection etc.



*[…] improvements in programming techniques and programming languages in particular are overwhelmingly more important than anything else in the software business […]*

*[…] programmers are interested in design […] when more expressive programming languages become available, software developers will adopt them.*

–Jack Reeves, To Code is to Design, C++ Report 1992

Design Challenge

## Implementing `trait Col[X]{map[T](f:(X)=>T){…}}`

This is only a first approximation of the mehtod's signature.

Try to **implement** the classical **`map`** function, which performs a mapping of the values of a collection using a given function, **only once for all collection classes**.

The function should be defined by the “top-level” class (e.g., **`Collection`**).

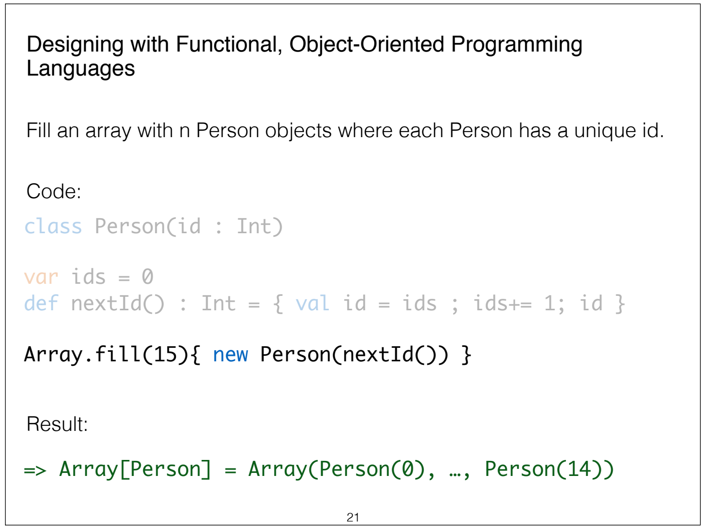
The type of the collection with the mapped values should correspond to runtime type of the source collection. If this is not possible, a *reasonable* other collection should be created. (The function should not fail!)

`trait Col[X]` defines a generic type with the type variable X. Furthermore, it defines the generic method map which defines the type parameter T and which gets a function f that maps a value of type X to a value of type T.
(In this context, loosely comparable to a generic abstract class in Java which has a type parameter and defines a concrete, generic method map.)

## Implementing `Col[X]{map[T](f:(X)=>T){…}}`

Initial Draft

```
trait Col[X] { def map[T](f : (X) => T) : Col[T] = {…} }
class List[X] extends Col[X] { /*does not override map!*/ …}
class BitSet[X] extends Col[X] {/*does not override map!*/ …}

val l = List(1,2,3)
l.map(i => i +1) // should result in List[Int](2,3,4)
val b = BitSet(1,2,3)
l.map(i => i +1) // should result in BitSet[Int](2,3,4)
l.map(i => "I:"+i) // should result in ???("I:1","I:2","I:3")
```



## Designing with Functional, Object-Oriented Programming Languages

Fill an array with n Person objects where each Person has a unique id.

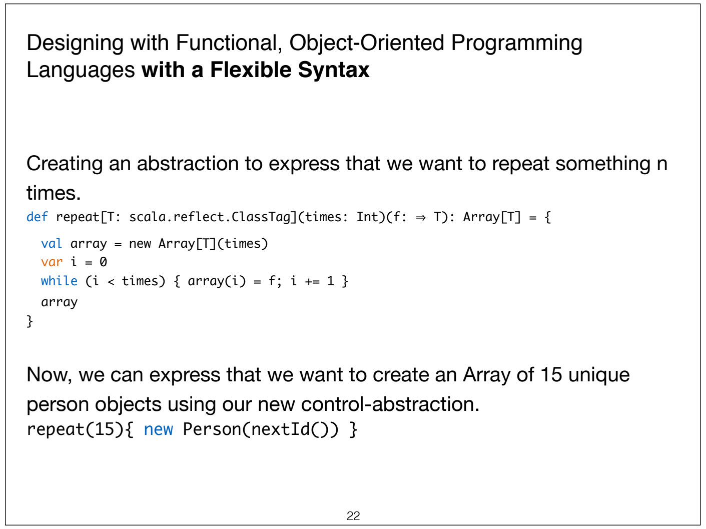
Code:

```
class Person(id : Int)

var ids = 0
def nextId() : Int = { val id = ids ; ids+= 1; id }

Array.fill(15){ new Person(nextId()) }
```

Result:

```
=> Array[Person] = Array(Person(0), …, Person(14))
```

By fusing object-oriented and functional programming we are provided with further means to raise our abstraction level. This enables us to better express our intention.



## Designing with Functional, Object-Oriented Programming Languages **with a Flexible Syntax**

Creating an abstraction to express that we want to repeat something n times.

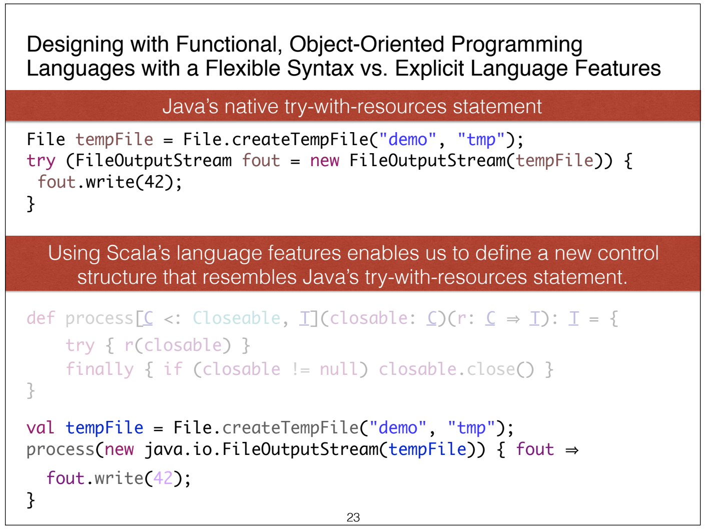
```
def repeat[T: scala.reflect.ClassTag](times: Int)(f: => T): Array[T] = {
  val array = new Array[T](times)
  var i = 0
  while (i < times) { array(i) = f; i += 1 }
  array
}
```

Now, we can express that we want to create an Array of 15 unique person objects using our new control-abstraction.

```
repeat(15){ new Person(nextId()) }
```

By fusing object-oriented and functional programming and also providing a more flexible syntax we are provided with further means to raise our abstraction level.  In this example, we demonstrate how to define our own “control-abstraction”! Defining a new control structure like this is not *reasonably possible* in Java 7 or older. In Java >= 8; the situation gets better due to closures. However, the syntax still doesn’t look like a native structure.

However, with power comes responsibility and it is easy to overdo!



## Designing with Functional, Object-Oriented Programming Languages with a Flexible Syntax vs. Explicit Language Features

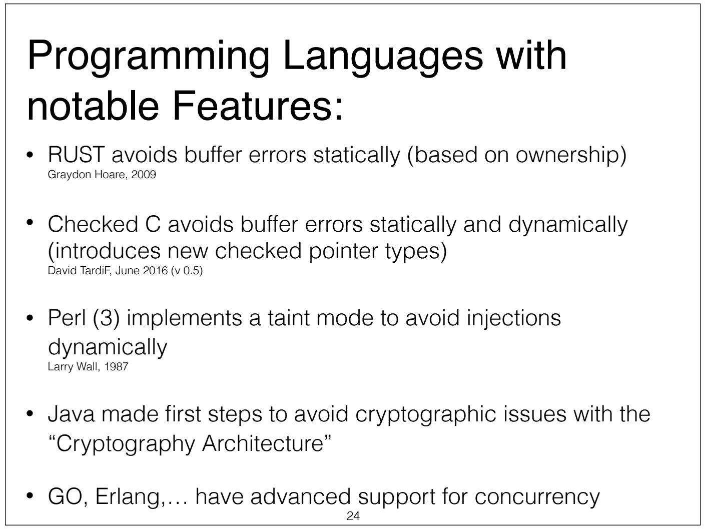
Java's native try-with-resources statement

```
File tempFile = File.createTempFile("demo", "tmp");
try (FileOutputStream fout = new FileOutputStream(tempFile)) {
 fout.write(42);
}
```

Using Scala's language features enables us to define a new control structure that resembles Java's try-with-resources statement.

```
def process[C <: Closeable, T](closable: C)(r: C ⇒ T): T = {
    try { r(closable) }
    finally { if (closable != null) closable.close() }
}

val tempFile = File.createTempFile("demo", "tmp");
process(new java.io.FileOutputStream(tempFile)) { fout ⇒
  fout.write(42);
}
```

Java 7's try-with-resources statement is more powerful/safe. However, it is an explicit language feature that was only added years after its need was identified.



## Programming Languages with notable Features:

- RUST avoids buffer errors statically (based on ownership)
  Graydon Hoare, 2009
- Checked C avoids buffer errors statically and dynamically (introduces new checked pointer types)
  David TardiF, June 2016 (v 0.5)
- Perl (3) implements a taint mode to avoid injections dynamically
  Larry Wall, 1987
- Java made first steps to avoid cryptographic issues with the “Cryptography Architecture”
- GO, Erlang,… have advanced support for concurrency

(Some languages avoid dead locks or race conditions.)

## We need good style to cope with complexity!

Help is provided through established practices and techniques, design patterns and principles.

**Good style can only be recommended, not enforced!**

Eventually style rules will have to be turned into language features to be really effective.

## General Design Principles

The following principles apply at various abstraction levels!

- Keep it short and simple
- Don't repeat yourself (also just called "DRY-Principle")
- High Cohesion
- Low Coupling
- No cyclic dependencies
- Make it testable
- Open-closed Design Principle
- Make it explicit/use Code
- Keep related things together
- Keep simple things simple
- Common-reuse/Common-closure/Reuse-release principles

Code/projects that adheres to these design principles generally have improved maintainability.

## Object-Oriented Design Principles

- Liskov Substitution Principle
- Responsibility Driven Design
- …

## Design Constraints

- **Conway's Law**
  A system's design is constrained by the organization's communication structure.