

Concepts of Object-Oriented Programming

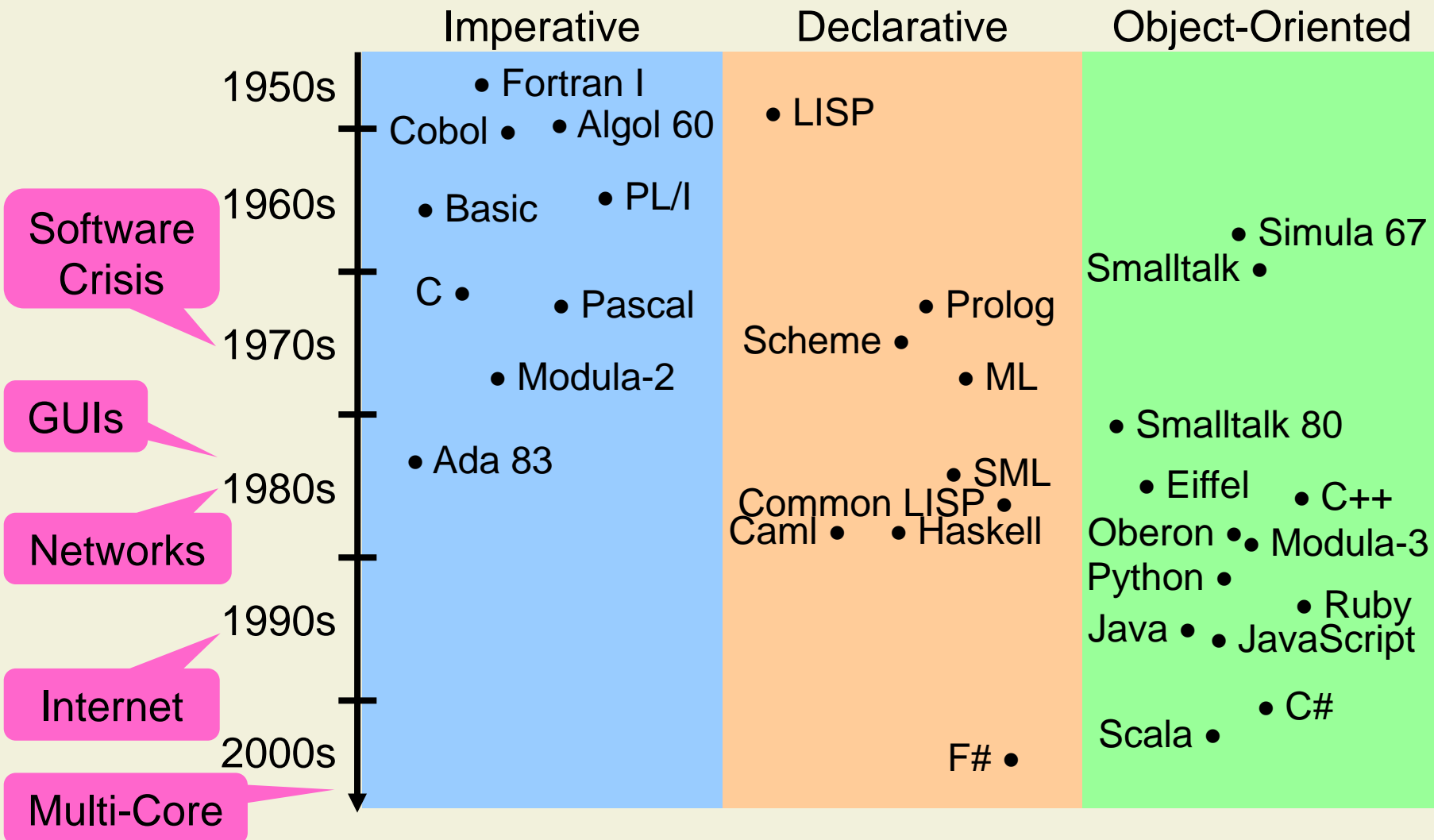
Peter Müller

Chair of Programming Methodology

Autumn Semester 2017

ETH zürich

History of Programming Languages



1. Introduction

1.1 Requirements

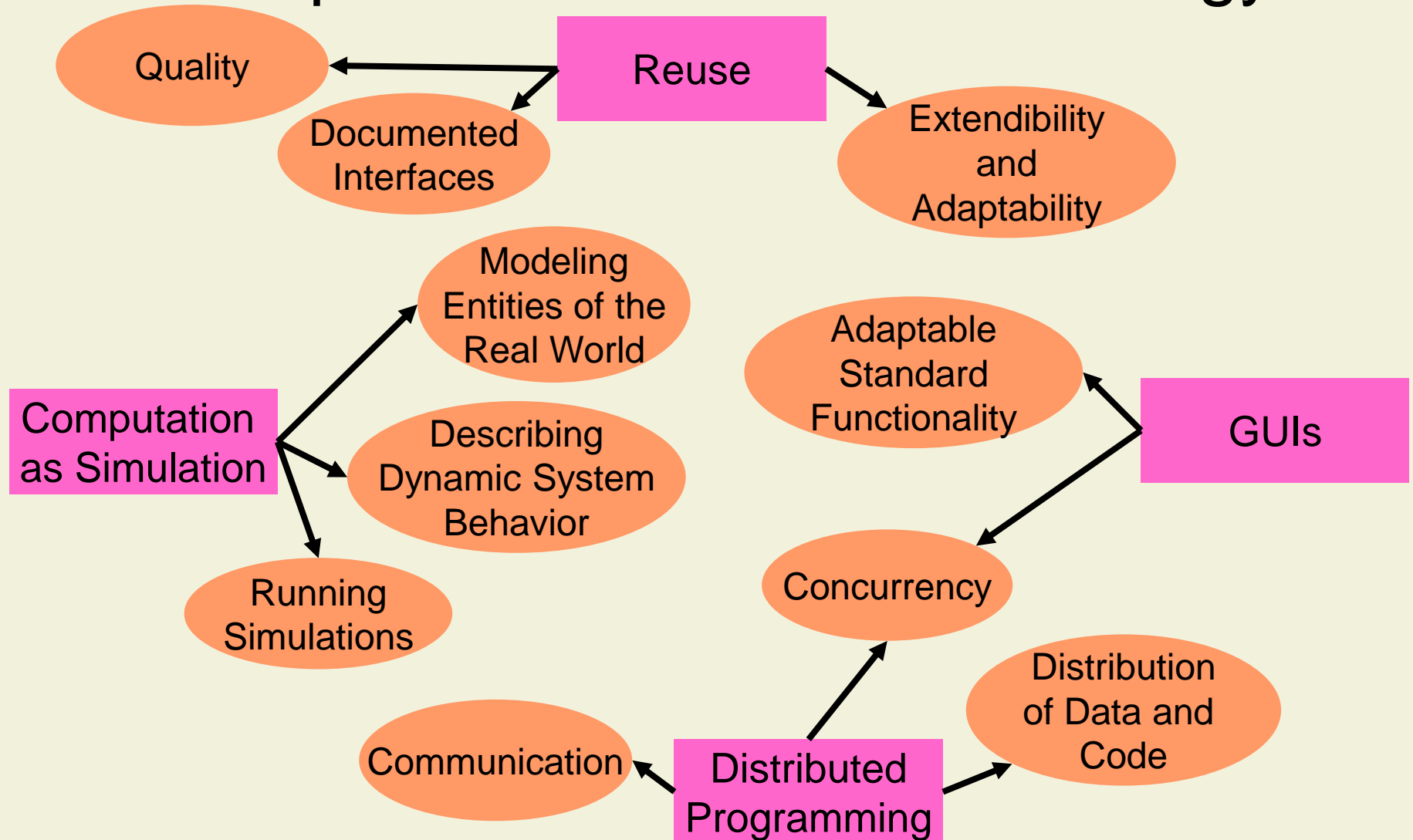
1.2 Core Concepts

1.3 Language Concepts

1.4 Course Organization

1.5 Language Design

New Requirements in SW-Technology



Example: Reusing Imperative Programs

- Scenario: University Administration System
 - Models students and professors
 - Stores one record for each student and each professor in a repository
 - Procedure printAll prints all records in the repository

An Implementation in C

```
typedef struct {  
    char *name;  
    char *room;  
    char *institute;  
} Professor;
```

```
typedef struct {  
    char *name;  
    int regNum;  
} Student;
```

```
void printStudent( Student *s )  
    { ... }
```

```
void printProf( Professor *p )  
    { ... }
```

An Implementation in C (cont'd)

```
typedef struct {  
    enum { STU,PROF } kind;  
    union {  
        Student *s;  
        Professor *p;  
    } u;  
} Person;
```

```
typedef Person **List;
```

```
void printAll( List l ) {  
    int i;  
    for ( i=0; l[ i ] != NULL; i++ )  
        switch ( l[ i ] -> kind ) {  
            case STU:  
                printStudent( l[ i ] -> u.s );  
                break;  
            case PROF:  
                printProf( l[ i ] -> u.p );  
                break;  
        }  
}
```

Extending and Adapting the Program

- Scenario: University Administration System
 - Models students and professors
 - Stores one record for each student and each professor in a repository
 - Procedure printAll prints all records in the repository
- Extension: Add assistants to system
 - Add record and print function for assistants
 - Reuse old code for repository and printing

Step 1: Add Record and Print Function

```
typedef struct {  
    char *name;  
    char *room;  
    char *institute;  
} Professor;
```

```
typedef struct {  
    char *name;  
    int regNum;  
} Student;
```

```
typedef struct {  
    char *name;  
    char PhD_student; /* 'y', 'n' */  
} Assistant;
```

```
void printStudent( Student *s )  
    { ... }
```

```
void printProf( Professor *p )  
    { ... }
```

```
void printAssi( Assistant *a )  
    { ... }
```

Step 2: Reuse Code for Repository

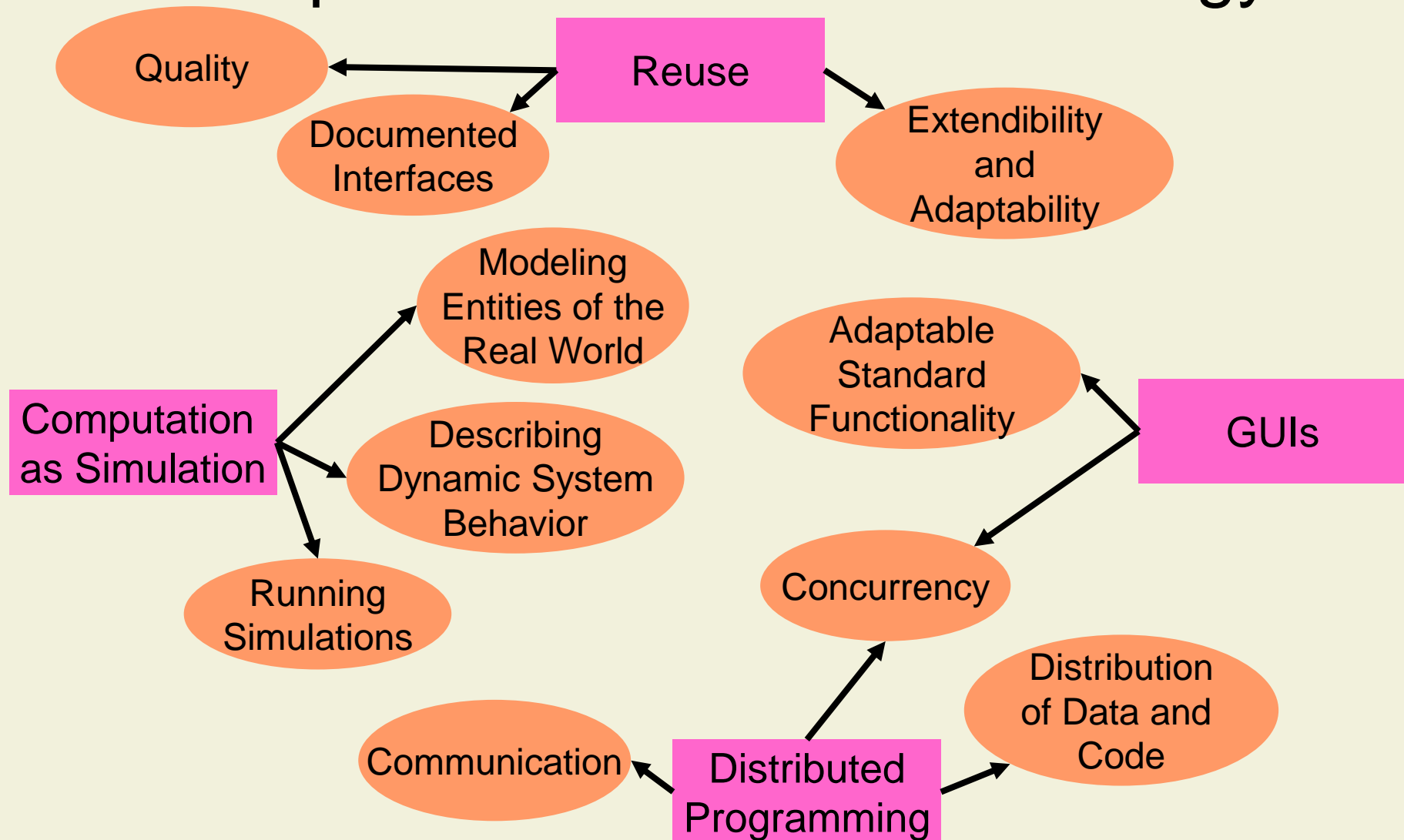
```
typedef struct {  
    enum { STU,PROF,ASSI } kind;  
    union {  
        Student *s;  
        Professor *p;  
        Assistant *a;  
    } u;  
} Person;  
  
typedef Person **List;
```

```
void printAll( List l ) {  
    int i;  
    for ( i=0; l[ i ] != NULL; i++ )  
        switch ( l[ i ] -> kind ) {  
            case STU:  
                printStudent( l[ i ] -> u.s );  
                break;  
            case PROF:  
                printProf( l[ i ] -> u.p );  
                break;  
            case ASSI:  
                printAssi( l[ i ] -> u.a );  
                break;  
        }  
}
```

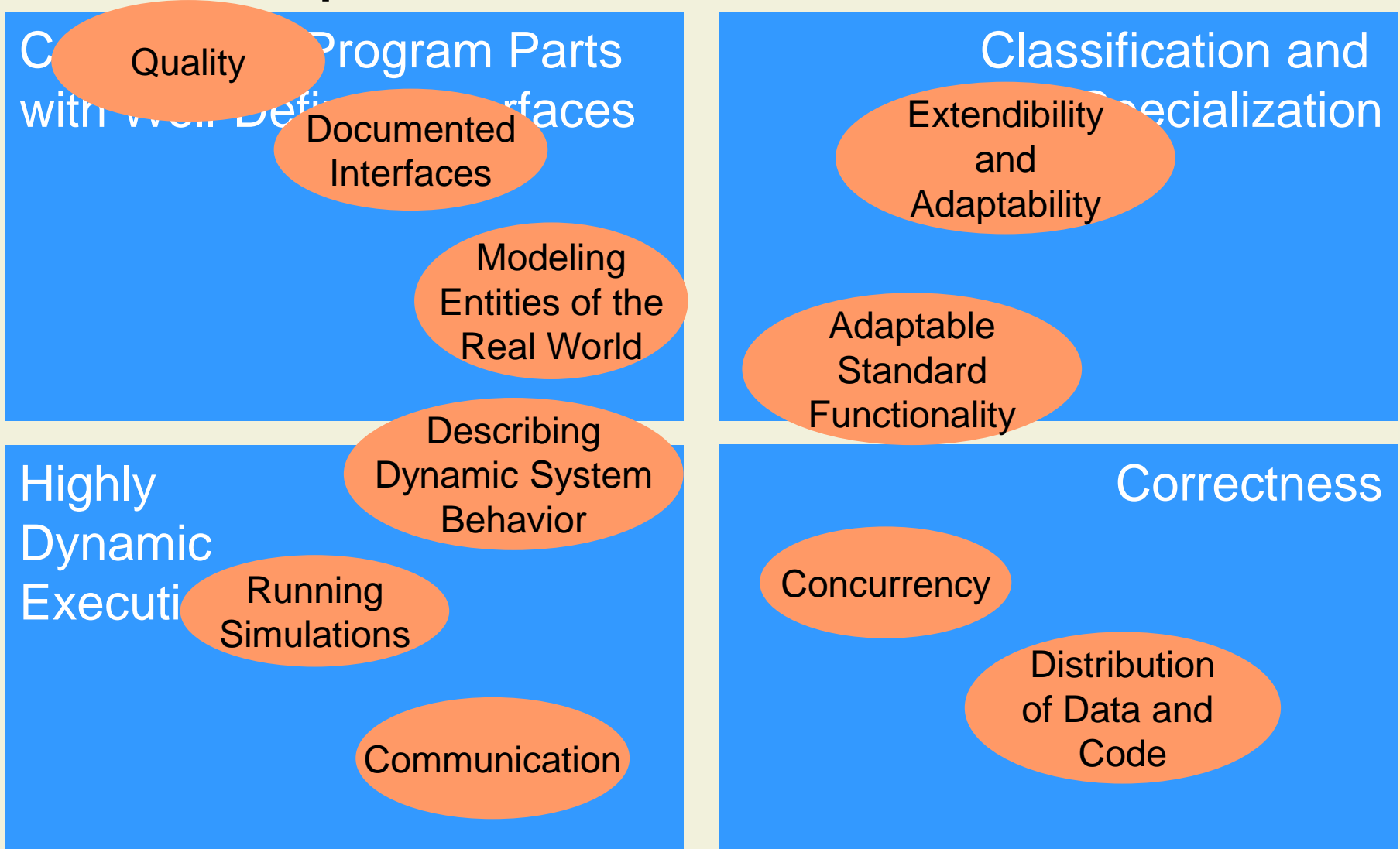
Reuse in Imperative Languages

- No explicit language support for extension and adaptation
- Adaptation usually requires modification of reused code
- Copy-and-paste reuse
 - Code duplication
 - Difficult to maintain
 - Error-prone

New Requirements in SW-Technology



Core Requirements



From Requirements to Concepts

What are the concepts of a programming paradigm

- That structure programs into **cooperating program parts with well-defined interfaces**?
- That are able to express **classification and specialization** of program parts without modifying reused code?
- That enable the **dynamic adaptation** of program behavior?
- That facilitate the development of **correct programs**?

1. Introduction

1.1 Requirements

1.2 Core Concepts

1.3 Language Concepts

1.4 Course Organization

1.5 Language Design

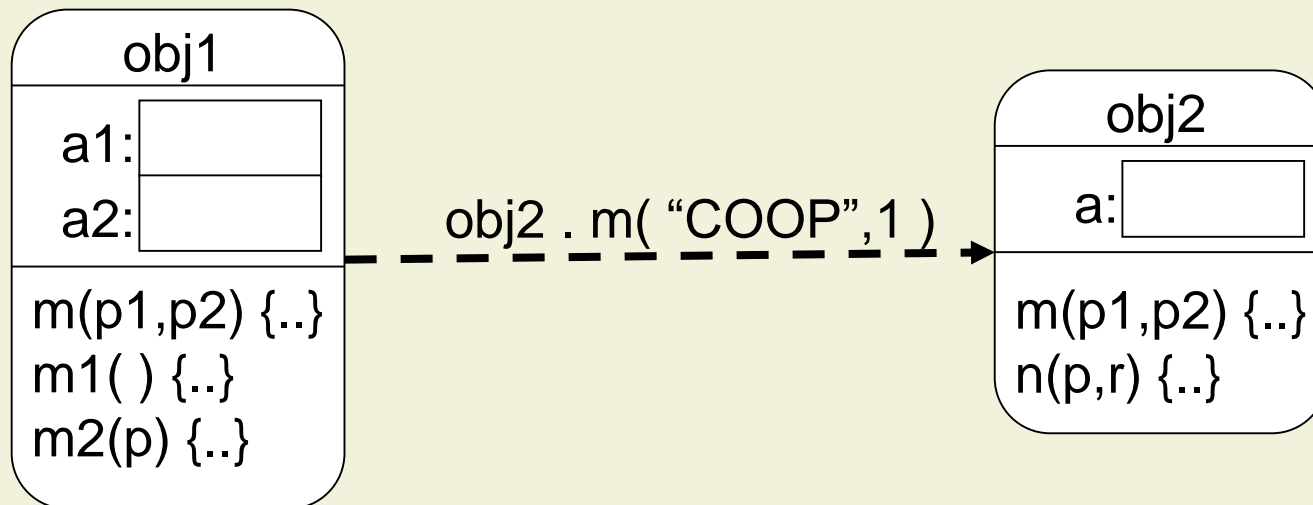
Object Model: The Philosophy

“The basic philosophy underlying object-oriented programming is to make the programs as far as possible reflect that part of the reality they are going to treat. It is then often easier to understand and to get an overview of what is described in programs. The reason is that human beings from the outset are used to and trained in the perception of what is going on in the real world. The closer it is possible to use this way of thinking in programming, the easier it is to write and understand programs.”

[Object-oriented Programming in the BETA Programming Language]

The Object Model

- A software system is a set of cooperating objects
- Objects have state and processing ability
- Objects exchange messages



Characteristics of Objects

- Objects have
 - State
 - Identity
 - Lifecycle
 - Location
 - Behavior

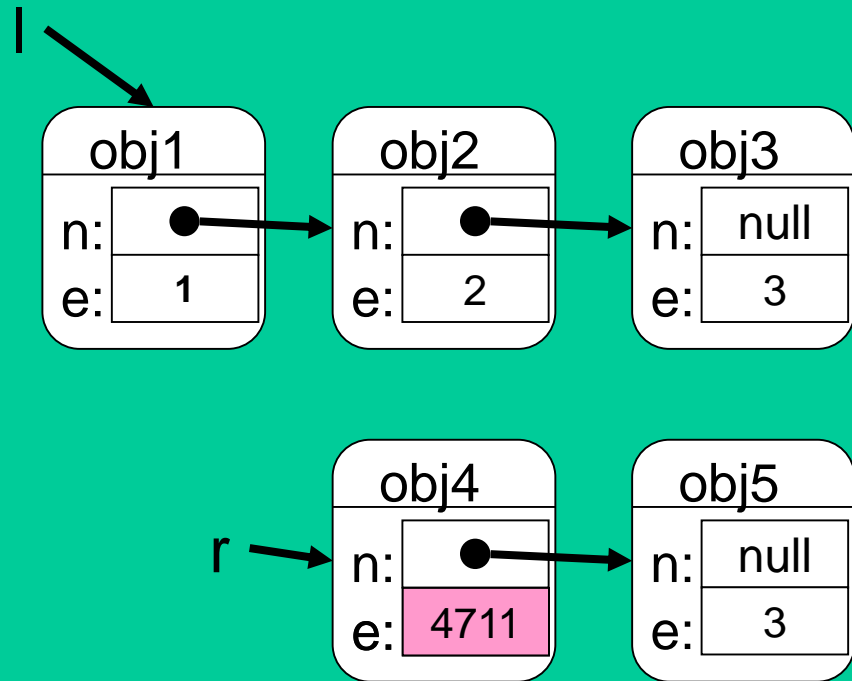
- Compared to imperative programming,
 - Objects lead to a different program structure
 - Objects lead to a different execution model

Object Identity: Example

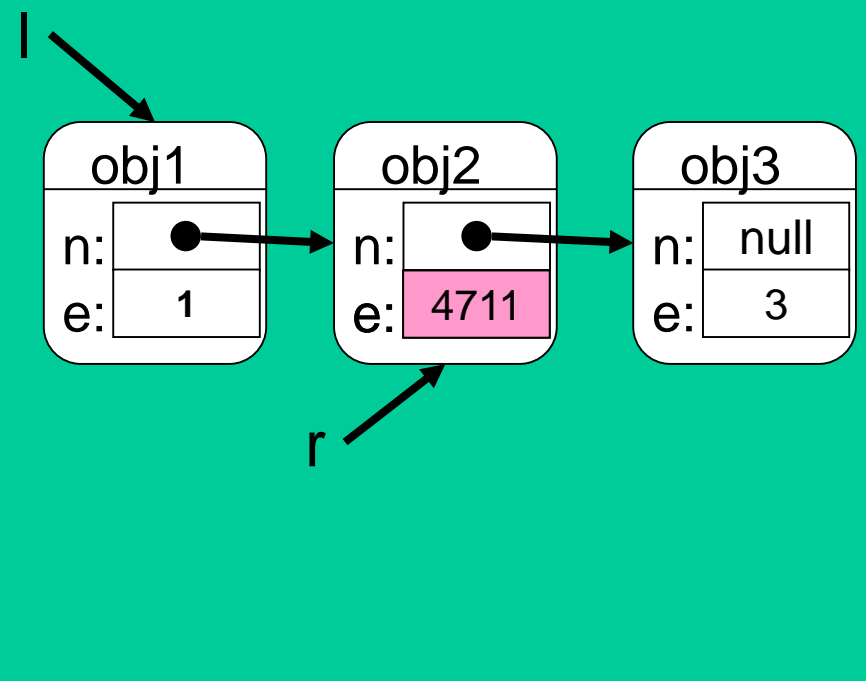
- Consider

```
r = l.rest( ); r.set( 4711 ); int i = l.next.get();
```

Variant 1: copying

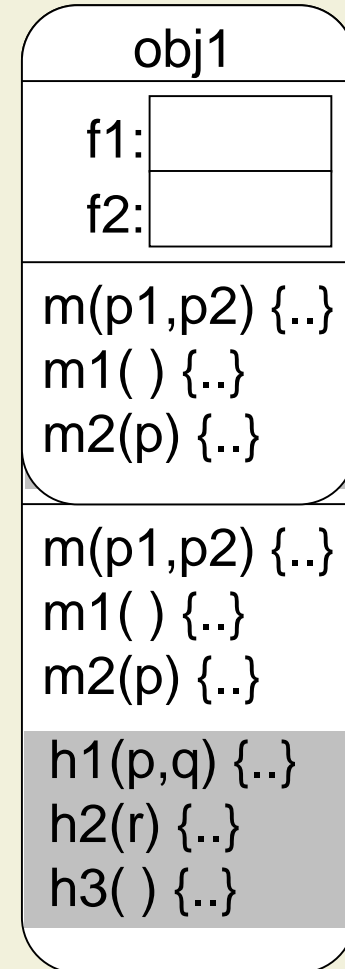


Variant 2: sharing



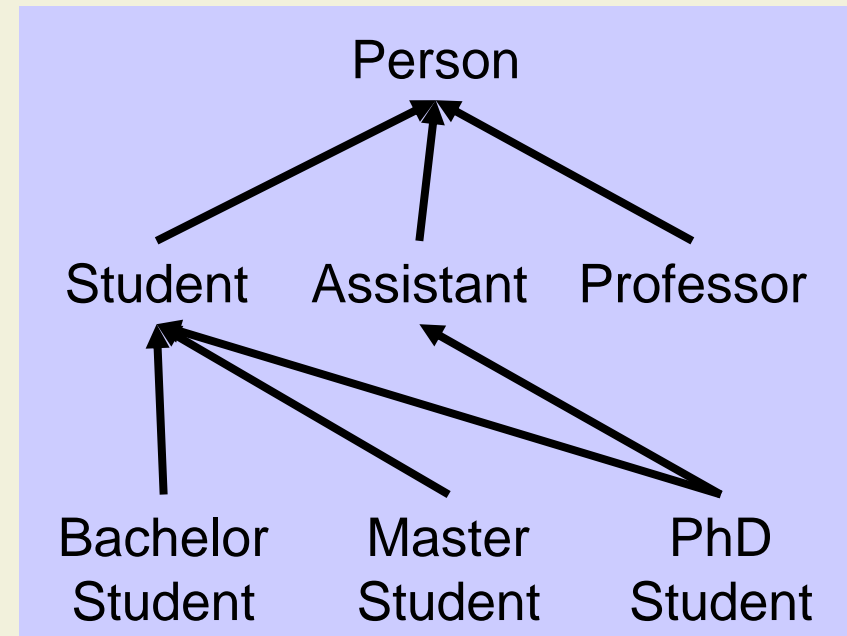
Interfaces and Encapsulation

- Objects have well-defined interfaces
 - Publicly accessible fields
 - Publicly accessible methods
- Implementation is hidden behind interface
 - Encapsulation
 - Information hiding
- Interfaces are the basis for describing behavior

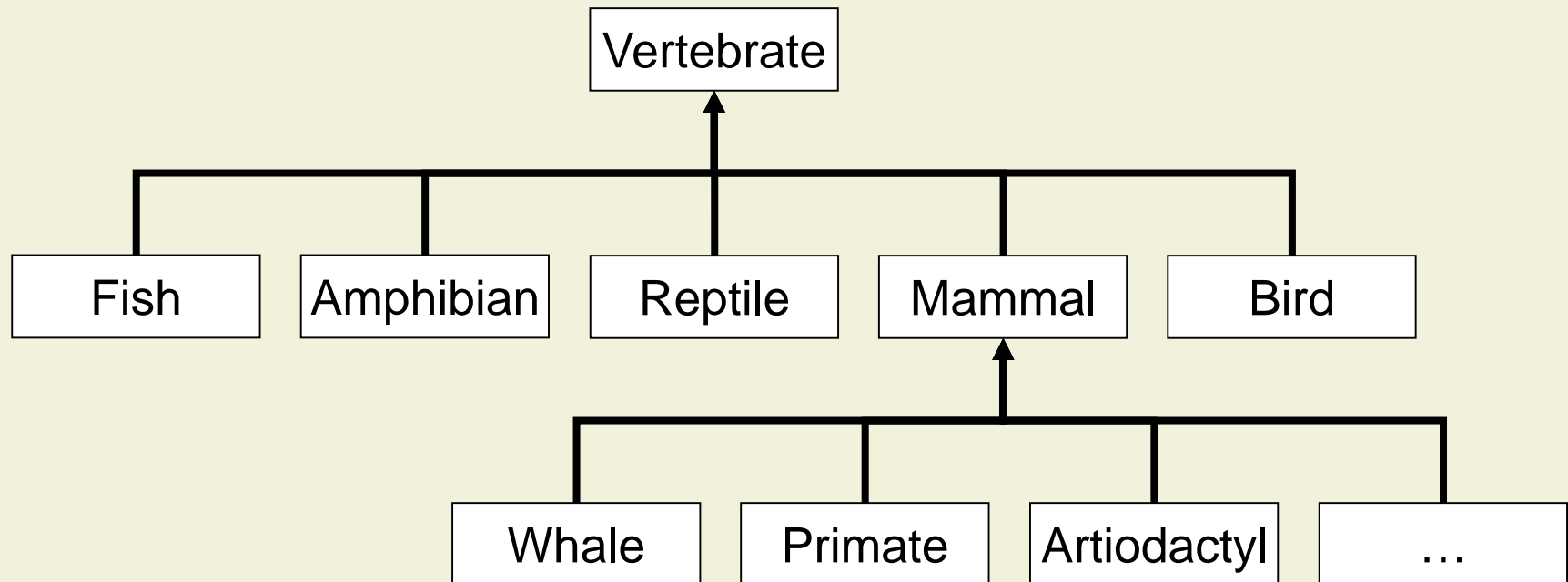


Classification and Polymorphism

- **Classification:**
Hierarchical structuring of objects
- Objects belong to different classes simultaneously
- **Substitution principle:**
Subtype objects can be used wherever supertype objects are expected



Classification of Vertebrates



Arrows represent
the “is-a” relation

Polymorphism

- Definition of *Polymorphism*:

The quality of being able to assume different forms

[Merriam-Webster Dictionary]

- In the context of programming:

A program part is polymorphic if it can be used for objects of several classes

Subtype Polymorphism

- Subtype polymorphism is a direct consequence of the substitution principle
 - Program parts working with supertype objects work as well with subtype objects
 - Example: printAll can print objects of class Person, Student, Professor, etc.

- Other forms of polymorphism (not core concepts)
 - Parametric polymorphism (generic types)
 - Ad-hoc polymorphism (method overloading)

Parametric Polymorphism: Example

```
class List<G> {  
    G[ ] elems;  
    void append( G p ) { ... }  
}
```

```
List<String> myList;  
myList = new List<String>( );  
myList.append( "String" );
```

```
myList.append( myList );
```

- Parametric polymorphism uses **type parameters**
- One implementation can be used for different types
- Type mismatches can be detected at compile time

Ad-hoc Polymorphism: Example

```
class Any {  
    void foo( Polar p ) { ... }  
    void foo( Coord c ) { ... }  
}
```

```
x.foo( new Coord( 5, 10 ) );
```

- Ad-hoc polymorphism allows several methods with the **same name but different arguments**
- Also called **overloading**
- No semantic concept: can be modeled by **renaming** easily

Specialization

- Definition of *Specialization*:
Adding specific properties to an object or refining a concept by adding further characteristics.
- Start from general objects or types
- Extend these objects and their implementations (add properties)
- Requirement: Behavior of specialized objects is compliant to behavior of more general objects
- Program parts that work for the more general objects work as well for specialized objects

Example: Specialization

- Develop implementation for type Person
- Specialize it

```
class Person {  
    String name;  
    ...  
    void print( ) {  
        System.out.println( name );  
    }  
}
```

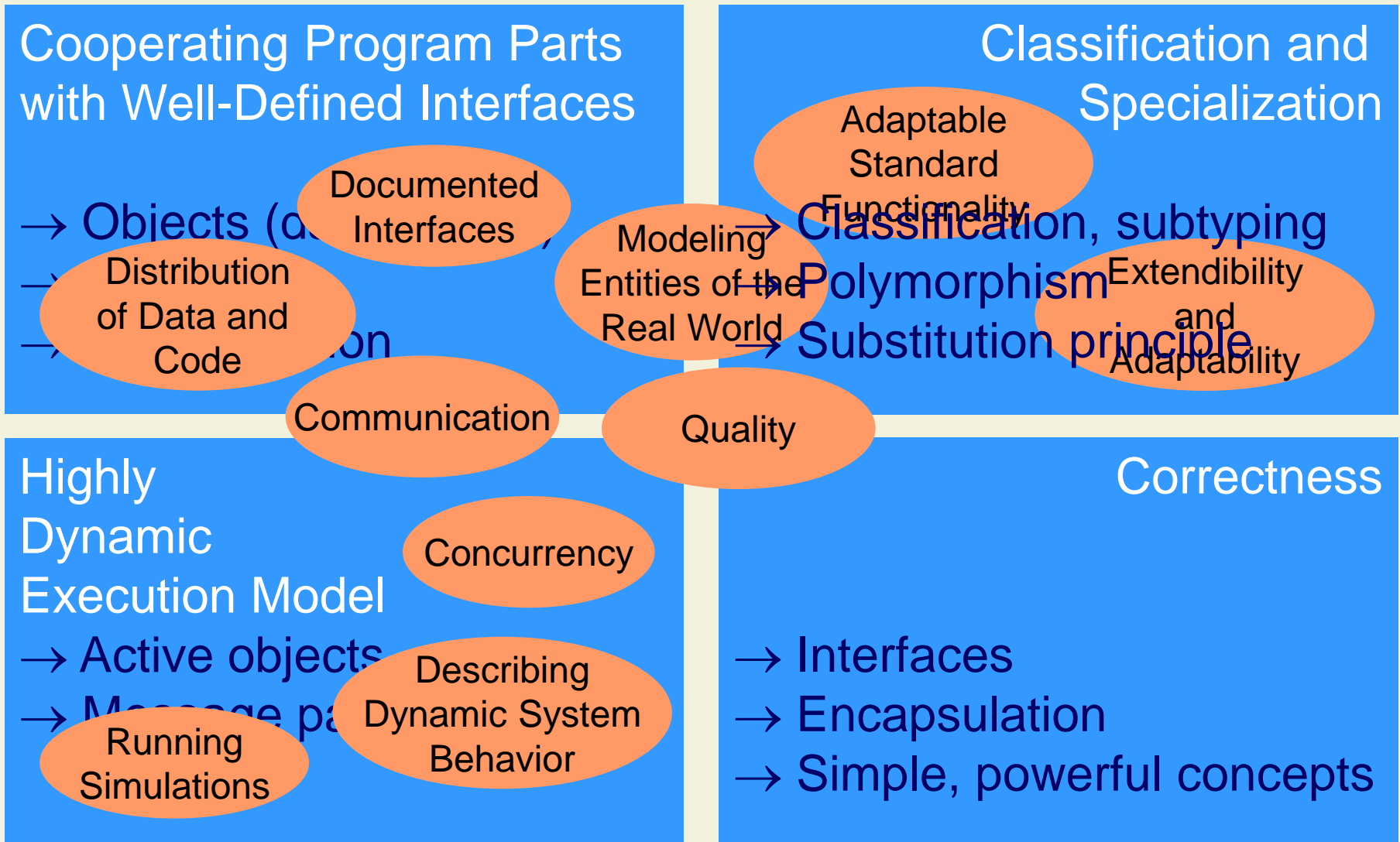
Example: Specialization (cont'd)

- Inheritance of
 - Fields
 - Methods
- Methods can be overridden in subclasses

```
class Student extends Person {  
    int regNum;  
  
    ...  
    void print( ) {  
        super.print( );  
        System.out.println( regNum );  
    }  
}
```

```
class Professor extends Person {  
    String room;  
  
    ...  
    void print( ) {  
        super.print( );  
        System.out.println( room );  
    }  
}
```

Meeting the Requirements



Core Concepts: Summary

- Core concepts of the OO-paradigm
 - Object model
 - Interfaces and encapsulation
 - Classification and polymorphism
- Core concepts are **abstract concepts** to meet the new requirements
- To apply the core concepts we need ways to **express them in programs**
- **Language concepts** enable and facilitate the application of the core concepts

1. Introduction

1.1 Requirements

1.2 Core Concepts

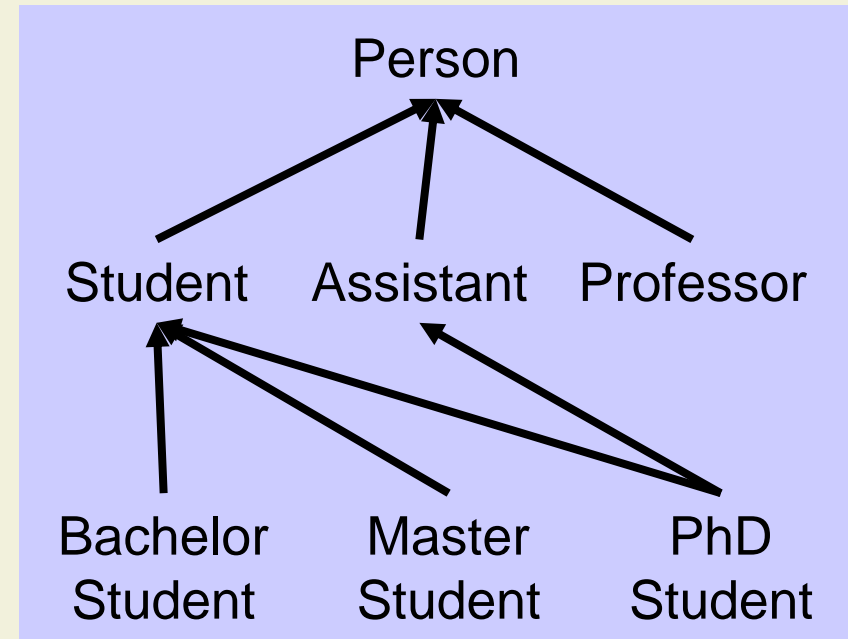
1.3 Language Concepts

1.4 Course Organization

1.5 Language Design

Example: Dynamic Method Binding

- Classification and polymorphism
 - Algorithms that work with supertype objects can be used with subtype objects
 - Subclass objects are specialized
- Dynamic binding: Method implementation is selected at runtime, depending on the type of the receiver object



```
void printAll( Person[ ] l ) {  
    for (int i=0; l[ i ] != null; i++)  
        l[ i ] . print( );  
}
```

OO-Concepts and Imperative Languages

- What we have seen so far
 - New concepts are needed to meet **new requirements**
 - **Core concepts** serve this purpose
 - **Language concepts** are needed to express core concepts in programs
- Open questions
 - Why do we need **OO-programming languages**?
 - Can't we use the language concepts as **guidelines** when writing imperative programs?
- Let's do an experiment ...
 - Writing object-oriented programs in C

Types and Objects

- Declare types

```
typedef char*      String;  
typedef struct sPerson Person;
```

- Declare records with

- Fields
- Methods
(function pointers)

```
struct sPerson {  
    String name;  
    void   ( *print )( Person* );  
    String ( *lastName )( Person* );  
};
```

Methods and Constructors

- Define methods

```
void printPerson( Person *this ) {  
    printf("Name: %s\n", this->name);  
}
```

```
String LN_Person( Person *this )  
{ ... }
```

- Define constructors

```
Person *PersonC( String n ) {  
    Person *this = (Person *)  
                    malloc( sizeof( Person ) );  
    this -> name      = n;  
    this -> print     = printPerson;  
    this -> lastName  = LN_Person;  
    return this;  
}
```

Using the “Object”

- Declaration
- Use constructors, fields, and methods

```
struct sPerson {  
    String name;  
    void   ( *print )( Person* );  
    String ( *lastName )( Person* );  
};
```

```
Person *p;  
p = PersonC( "Tony Hoare" );  
p->name = p->lastName( p );  
p->print( p );
```

Inheritance and Specialization

- Copy code
- Adapt function signatures
- Define specialized methods

```
typedef struct sStudent Student;  
struct sStudent {  
    String name;  
    void ( *print )( Student* );  
    String ( *lastName )( Student* );  
    int regNum;  
};
```

```
void printStudent( Student *this ) {  
    printf("Name: %s\n", this->name);  
    printf("No: %d\n", this->regNum);  
}
```

Inheritance and Specialization (cont'd)

- Reuse LN_Person for Student
- View Student as Person (cast)

```
Student *StudentC( String n, int r ) {  
    Student *this = (Student *)  
        malloc( sizeof( Student ) );  
  
    this -> name      = n;  
    this -> print     = printStudent;  
  
    this -> lastName  =  
        (String (*)(Student*)) LN_Person;  
    this -> regNum    = r;  
  
    return this;  
}
```

Subclassing and Dynamic Binding

- Student has all fields and methods of Person
- Casts are necessary
- Array I can contain Person and Student objects
- Methods are selected dynamically

```
Student *s;  
Person *p;  
s = StudentC( "Susan Roberts", 0 );  
p = (Person *) s;  
p -> name = p -> lastName( p );  
p -> print( p );
```

```
void printAll( Person **I ) {  
    int i;  
    for ( i=0; I[ i ] != NULL; i++ )  
        I[ i ] -> print( I[ i ] );  
}
```


Discussion of the C Solution: Pros

- We can express **objects**, **fields**, **methods**, **constructors**, and **dynamic method binding**
- By imitating OO-programming, the union in Person and the switch statement in printAll became dispensable
- The behavior of reused code (Person, printAll) can be **adapted** (to introduce Student) **without changing the implementation**

Discussion of the C Solution: Cons

- Inheritance has to be replaced by **code duplication**
- Subtyping can be simulated, but it requires
 - Casts, which is **not type safe**
 - **Same memory layout** of super and subclasses (same fields and function pointers in same order), which is **extremely error-prone**
- Appropriate language support is needed to apply object-oriented concepts

A Java Solution

```
class Person {  
    String name;  
    void print( ) {  
        System.out.println( "Name: " +  
            name );  
    }  
    String lastName( ) { ... }  
    Person( String n ) { name = n; }  
}
```

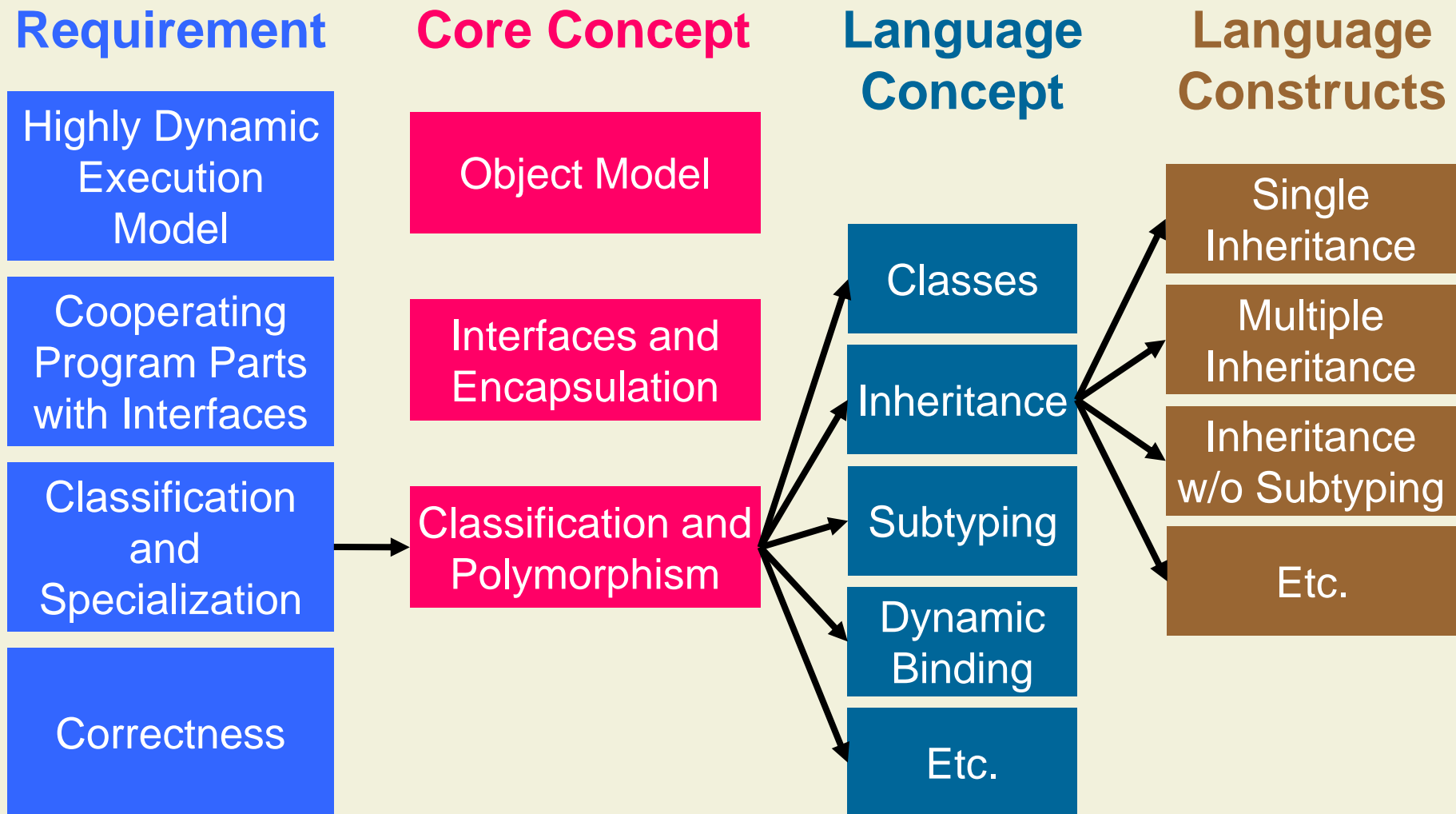
```
void printAll( Person[ ] l ) {  
    for ( int i=0; l[ i ] != null; i++)  
        l[ i ].print( );  
}
```

```
class Student extends Person {  
    int regNum;  
    void print( ) {  
        super.print( );  
        System.out.println( "No: " +  
            regNum );  
    }  
    Student( String n, int i ) {  
        super( n );  
        regNum = i;  
    }  
}
```

Discussion of the Java Solution

- The Java solution uses
 - Inheritance to avoid code duplication
 - Subtyping to express classification
 - Overriding to specialize methods
 - Dynamic binding to adapt reused algorithms
- Java supports the OO-language concepts
- The Java solution is
 - Simpler and smaller
 - Easier to maintain (no duplicate code)
 - Type safe

Concepts: Summary



1. Introduction

1.1 Requirements

1.2 Core Concepts

1.3 Language Concepts

1.4 Course Organization

1.5 Language Design

After this Course, you should be able

- To understand the core and language concepts
- To understand language design trade-offs
- To compare OO-languages

- To learn new languages faster
- To apply language concepts and constructs correctly

- To write better object-oriented programs

Approach

- We discuss the
 - Concepts of
 - as opposed to implementations, etc.
 - Object-Oriented
 - as opposed to imperative, declarative
 - Programming
 - as opposed to analysis, design, etc.
- We study and compare solutions in different languages such as C++, C#, Eiffel, Java, Python, and Scala
 - Java is used for most examples and exercises
- We look at code and analyze programs

Course Outline

2. Types and Subtyping
3. Inheritance
4. Static Safety
5. Parametric Polymorphism
6. Object Structures and Aliasing
7. Extended Typing
8. Object and Class Initialization
9. Object Consistency
10. Reflection
11. Higher-Order Features

Classification and
Specialization

Cooperating
Program Parts

Correctness

Highly Dynamic
Execution Model

Exams

- Mid-term exam
 - Written (1 hour)
 - 30% of the overall grade
 - Friday, November 10, 11:15 – 12:15
 - No registration required
- End-term exam
 - Written (2 hours)
 - 70% of the overall grade
 - Thursday, December 21, 9:15 – 11:15
 - Registration required
- Exams will be closed-book

Course Infrastructure

- Web page:
www.pm.inf.ethz.ch/education/courses/COOP.html
- Slides will be available on the web page two days before the lecture
 - Exercise assignments and solutions are published on Friday
- Responsible assistant:
Marco Eilers
marco.eilers@inf.ethz.ch

Exercise Sessions

- Friday, **starting September 29**

- Alexandra Bugariu : 8:15 – 10:00 CHN D42
 - Last name A – L

- Caterina Urban: 8:15 – 10:00 CAB G57
 - Last name M – Z

- Lucas Brutschy: 10:15 – 12:00 CHN D42
 - Last name A – L

- Marco Eilers: 10:15 – 12:00 CAB G57
 - Last name M – Z

1. Introduction

1.1 Requirements

1.2 Core Concepts

1.3 Language Concepts

1.4 Course Organization

1.5 Language Design

What is a Good OO-Language?

- One that many people use?

- No!

- (Or do you think JavaScript is a good language?)



- One that makes programmers productive?

- No! (Or would you feel good if the Airbus flight controller was written in Python?)

- A good language should resolve design trade-offs in a way **suitable for its application domain**

Design Goals: Simplicity

- Syntax and semantics can easily be understood by users and implementers of the language
- But not small number of constructs
- Simple languages: BASIC, Pascal, C

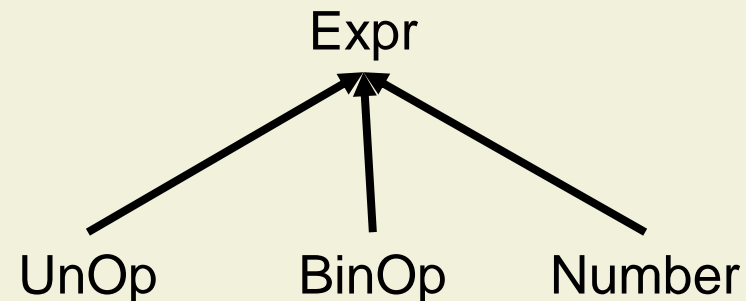
```
factorial ( i: INTEGER ): INTEGER
  require 0 <= i
  once
    if i <= 1 then Result := 1
    else
      Result := i
      Result := Result * factorial ( i - 1 )
    end
  end
end
```

Eiffel

- It took over 10 years to find out that the Java 5 type system (generics) is not decidable (and unsound)

Design Goals: Expressiveness

- Language can (easily) express complex processes and structures
- Expressive languages: C#, Scala, Python
- Often conflicting with simplicity



```
def simplify( expr: Expr ): Expr =  
  expr match {  
    case UnOp( "-", UnOp("-", e) ) => e  
    case BinOp( "+", e, Number(0) ) => e  
    case BinOp( "*", e, Number(1) ) => e  
    case _ => expr  
  }
```

Scala

Design Goals: (Static) Safety

- Language discourages errors and allows errors to be discovered and reported, ideally at compile time
- Safe languages: Java, C#, Scala
- Often conflicting with expressiveness and performance

```
int foo( List<Integer> l, int i ) {  
    if ( l.get( 0 ) != i ) return i / 5;  
    else return 0;  
}
```

Java

```
List<Integer> l;  
l = new ArrayList<Integer>();  
l.add( 7 );  
foo( l, "5" );
```

```
def foo( l, i ):  
    if l[ 0 ] != i: return i / 5  
    else: return 0
```

Python

```
l = []  
l.append( 7 )  
foo( l, "5" )
```

Design Goals: Modularity

- Language allows modules to be compiled separately
- Modular languages: Java, C#, Scala
- Often conflicting with expressiveness and performance

```
int foo( int p ) {  
    return p;  
}
```

C

```
#include <stdio.h>  
  
int main( int argc, char* argv[ ] ) {  
    printf( "%d\n", bar(5, 7) );  
    printf( "%d\n", bar() );  
    return 0;  
}
```

```
Client.o: ...  
undefined reference to `_bar'
```

Design Goals: Performance

- Programs written in the language can be executed efficiently
- Efficient languages: C, C++, Fortran
- Often conflicting with safety, productivity, and simplicity

C++ arrays

- Sequence of memory locations
- Access is simple look-up (only 2-5 machine instructions)

Java arrays

- Sequence of memory locations **plus length**
- Access is look-up **plus bound-check**

Design Goals: Productivity

- Language leads to low costs of writing programs
- Closely related to expressiveness
- Languages for high productivity:
Visual Basic, Python
- Often conflicting with static safety

```
def qsort( lst ):
    if len( lst ) <= 1:
        return lst
    pivot = lst.pop( 0 )
    greater_eq = \
        qsort( [ i for i in lst if i >= pivot ] )
    lesser = \
        qsort( [ i for i in lst if i < pivot ] )
    return lesser + [ pivot ] + greater_eq
```

Python

Design Goals: Backwards Compatibility

- Newer language versions work and interface with programs in older versions
- Backwards compatible languages: Java, C
- Often in conflict with simplicity, performance, and expressiveness

```
class Tuple<T> {  
    T first; T second;  
  
    void set( T first, T second ) {  
        this.first = first;  
        this.second = second;  
    }  
}
```

Java

```
class Client {  
    static void main( String[ ] args ) {  
        Tuple t = new Tuple();  
        t.set( "Hello", new Client() );  
    }  
}
```