

Motivation

Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory management

Programming Paradigms

Imperative

Declarative

OO

Concurrent

Other paradigms

Interaction-based

References

Theme 1. Introduction (Part 1)

Programming Languages, Technologies and Paradigms (LTP)

DSIC, ETSInf



Motivation

Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory management

Programming Paradigms

Imperative

Declarative

OO

Concurrent

Other paradigms

Interaction-based

References

- 1 Motivation
- 2 Essential concepts in programming languages
 - Types and type systems
 - Polymorphism
 - Reflection
 - Procedures and control flow
 - Memory management
- 3 Main Programming Paradigms: imperative, functional, logic, object oriented, concurrent
 - Imperative Paradigm
 - Declarative Paradigm
 - Object-Oriented Paradigm
 - Concurrent Paradigm
- 4 Other paradigms: interaction-based, emerging
 - Interaction-based Paradigm
- 5 References

Main goals

Motivation

Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory management

Programming Paradigms

Imperative

Declarative

OO

Concurrent

Other paradigms

Interaction-based

References

- Knowing about the evolution of programming languages and their contributions and impact in the design of other languages.
- Understanding existing (and emerging) programming paradigms (imperative, functional, logic, OO, and concurrent paradigms) and their specific features.
- Understanding different abstraction mechanisms (genericity, inheritance, use of modules) and evaluation strategies (eager/lazy).
- Identifying essential aspects of programming languages: dynamic/static scope, memory management.
- Understanding the criteria for choosing the appropriate programming paradigm/language according to the targeted application, its size, and the programming methodology.
- Understanding the main features of programming languages with regard to the underlying model (paradigm) and main components (type/class system, execution model, abstractions).
- Understanding the concerns of programming language expressivity in the implementation and execution costs.

The story goes back to 1950...

Motivation

Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory management

Programming Paradigms

Imperative

Declarative

OO

Concurrent

Other paradigms

Interaction-based

References

THE FIFTIES:

- Programmer time was cheap; the machines were expensive:

keep the machine busy

- Sometimes, programs were directly written in machine code. Hand-made compilations were frequent to achieve the best performance for a given hardware:

direct connection language-hardware

NOWADAYS:

- Programmer time is expensive; the machines are cheap:

keep the programmer busy

- Programs are intended to be efficient, but automatically compiled to generate (portable) code which should also be as efficient as possible:

direct connection between program and language design: objects, concurrency, etc.

Motivation

Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory management

Programming Paradigms

Imperative

Declarative

OO

Concurrent

Other paradigms

Interaction-based

References

Teaching PLs

Three approaches

- 1 Programming is a job
- 2 Programming as a branch of mathematics
- 3 Programming by concepts

Motivation

Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory management

Programming Paradigms

Imperative

Declarative

OO

Concurrent

Other paradigms

Interaction-based

References

1. Programming is a job

- You learn just a paradigm and a single language
- Self-defeating: having a bad experience in, for instance, handling lists in some languages can lead to the erroneous conclusion that managing lists is costly and complicated

1. Programming is a job

ZIP lists in Java

```
class Pair<A, B> {
    private A left;
    private B right;

    public Pair(A left, B right) {
        this.left = left;
        this.right = right;
    }

    public A left() { return left; }
    public B right() { return right; }
    public String toString() {
        return "(" + left + "," +
            right + ")";
    }
}

public class MyZip {
    public static <A, B> List<Pair<A, B>> zip(List<A> as, List<B> bs) {
        Iterator<A> it1 = as.iterator();
        Iterator<B> it2 = bs.iterator();
        List<Pair<A, B>> result = new ArrayList<>();
        while (it1.hasNext() && it2.hasNext()) {
            result.add(new Pair<A, B>(it1.next(), it2.next()));
        } return result;
    }

    public static void main(String[] args) {
        List<Integer> x = Arrays.asList(1, 2, 3);
        List<String> y = Arrays.asList("a", "b", "c");
        List<Pair<Integer, String>> zipped = zip(x, y);
        System.out.println(zipped);
    }
}
```

Output

```
[(1,a), (2,b), (3,c)]
```

ZIP lists in Haskell

```
zip :: [a] -> [b] -> [(a,b)]
```

```
zip [] xs           = []
zip (x:xs) []       = []
zip (x:xs) (y:ys) = (x,y):zip xs ys
```

Use

```
: zip [1,2,3] ["a","b","c"]
[(1,"a"), (2,"b"), (3,"c")]
```

2. Programming as a branch of mathematics

Motivation

Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory management

Programming Paradigms

Imperative

Declarative

OO

Concurrent

Other paradigms

Interaction-based

References

- You either learn an ‘ideal’, restricted language (Dijkstra) or else you run out from the real world.

2. Programming as a branch of mathematics

Example: formal verification (of a single line (!) program)

The program

```
while (x<10) x:=x+1;
```

The proof

One can then prove the following *Hoare triple*:

```
{ $x \leq 10$ } while (x<10) x:=x+1 { $x=10$ }
```

The condition C of the while loop is $x < 10$. A useful loop invariant is $x \leq 10$. Under these assumptions it is possible to prove the following Hoare triple

```
{ $x < 10 \wedge x \leq 10$ } x:=x+1 { $x \leq 10$ }
```

While this triple can be derived formally from the rules of Floyd-Hoare logic governing the assignment, it is also intuitively justified: *computation starts in a state where $x < 10 \wedge x \leq 10$ is true, which means simply that $x < 10$ is true. The computation adds 1 to x , which means that $x \leq 10$ is true (for integer x).* Under this premise, the rule for while loops permits the following conclusion:

```
{ $x \leq 10$ } while (x<10) x:=x+1 { $\neg(x < 10) \wedge x \leq 10$ }
```

However, the post-condition $\neg(x < 10) \wedge x \leq 10$ is logically equivalent to $x = 10$.

3. Programming by concepts

Motivation

Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory management

Programming Paradigms

Imperative

Declarative

OO

Concurrent

Other paradigms

Interaction-based

References

- Learn the **semantic concepts** and **implementation structures** leading to a natural description of the languages and their implementations

3. Programming by concepts

Features from different
'blocks' may coexist in a given programming language

Functional language

- (+) Polymorphism
- (+) Strategies
- (+) Higher-order

Logic language

- (+) Nondeterminism
- (+) Logic variables
- (+) Unification

***Kernel* language**

- (+) Data abstraction
- (+) Recursion
- (+) ...

Imperative language

- (+) Explicit state
- (+) Modularity
- (+) Components

OO Language

- (+) Classes
- (+) Inheritance

***Dataflow* language**

- (+) Concurrency

Essential concepts

Motivation

Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory management

Programming Paradigms

Imperative

Declarative

OO

Concurrent

Other paradigms

Interaction-based

References

The following concepts are essential in our presentation of PLs:

- Types and type systems
- Polymorphism
- Reflection
- Parameter passing
- Variable scope
- Memory management

Types and type systems

Motivation

Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory management

Programming Paradigms

Imperative

Declarative

OO

Concurrent

Other paradigms

Interaction-based

References

A **type** represents the set of values that can be given to a variable or expression. Types:

- Avoid programming **errors**

*Only programs that make a consistent use of expressions of a given type are **legal***

- Are helpful to give **structure** to the information

We think of types as collections of values sharing a number of properties

- Are helpful to handle **data structures**

Types tell us how data structures should be used

Types and type systems

Typed languages

Motivation

Concepts

Types and type systems

- Polymorphism
 - Overloading
 - Coercion
 - Genericity
- Inclusion
- Reflection
- Procedures and control flow
 - Parameter passing
 - Variable scope
- Memory management

Programming Paradigms

- Imperative
- Declarative
- OO
- Concurrent

Other paradigms

- Interaction-based

References

- In **typed** languages all variables have a type (e.g., C, C++, C#, Haskell, Java, Maude).
- **Untyped** languages do not restrict the values adopted by the variables (e.g., Lisp, Prolog).
 - All values have a single (*universal*) type

Types and type systems

Typed languages

Motivation

Concepts

Types and type systems

- Polymorphism
- Overloading
- Coercion
- Genericity
- Inclusion
- Reflection
- Procedures and control flow
- Parameter passing
- Variable scope
- Memory management

Programming Paradigms

- Imperative
- Declarative
- OO
- Concurrent

Other

paradigms

- Interaction-based

References

The type system establishes which kind of values can be associated to variables:

- The type of the **value** and the type of the variable must coincide (e.g., C, Haskell)
- The type of the **value** is *compatible* (according to the type system) with the type of the variable (e.g., C++, C#, Java).
- Besides, the type of the **value** associated to a given variable may change:
 - **Static typing**: the type of the value does not change during the execution
 - **Dynamic typing**: the type of the value may change during the execution

Types and type systems

Typed languages

Motivation

Concepts

Types and type systems

- Polymorphism
- Overloading
- Coercion
- Genericity
- Inclusion
- Reflection
- Procedures and control flow
 - Parameter passing
 - Variable scope
- Memory management

Programming Paradigms

- Imperative
- Declarative
- OO
- Concurrent

Other paradigms

- Interaction-based

References

$$\text{Typed Languages} = \text{Program Expressions} + \text{Type System}$$

- In languages with **explicit** typing, types are part of the syntax
- In languages with **implicit** typing, types do **not** need to be part of the syntax

Typed languages

Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory

management

Programming Paradigms

Imperative

Declarative

00

Concurrent

Other paradigms

Interaction-based

References

- ```
object(key).
object(ball).
thing(X) <- object(X).
```

Variable `x` has no associated type

- ```
int x;  
x = 42;
```

All variables must be declared. Variable declarations include a type

- ```
fac 0 = 1
fac x = x * fac (x-1)
```

The type system automatically infers the type of variable  $x$

# Types and type systems

## Motivation

## Concepts

### Types and type systems

- Polymorphism
- Overloading
- Coercion
- Genericity
- Inclusion
- Reflection
- Procedures and control flow
- Parameter passing
- Variable scope
- Memory management

## Programming Paradigms

- Imperative
- Declarative
- OO
- Concurrent

## Other paradigms

- Interaction-based

## References

Types are described by means of a *language of type expressions*.

## Example of a language of type expressions:

- Basic or primitive types: `Bool`, `Char`, `Int`, ...
- Type variables: `a`, `b`, `c`, ...
- Type constructors:
  - $\rightarrow$  to define functions,
  - $\times$  for pairing,
  - `[]` to define lists
  - ...
- Rules to build type expressions:

$$\tau ::= \text{Bool} \mid \text{Char} \mid \text{Int} \mid \dots \mid \tau \rightarrow \tau \mid \tau \times \tau \mid [\tau]$$

## Motivation

## Concepts

## Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

# Types and type systems

## Monomorphic and polymorphic types

- Types whose type expression contains **no** type variable are called **monomorphic** types or just **monotypes**
- Types whose type expression contain type variables are called **polymorphic** types or just **polytypes**
- A polymorphic type represents an infinite number of monotypes

# Types and type systems

## Examples of type expressions

### Motivation

### Concepts

#### Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory management

### Programming Paradigms

Imperative

Declarative

OO

Concurrent

### Other paradigms

Interaction-based

### References

- Predefined type expressions. Basic types: `Bool`, `Int`, ...

`Bool` is the type of boolean values `True` and `False`

- Functional type expressions.

`Int → Int` is the type of function `fact` (see above), which returns the factorial of a number.

- Parametric type expressions.

`[a] → Int` is the type of function `length`, which computes the length of a list.

# Types and type systems

## Examples of type expressions

### Motivation

### Concepts

#### Types and type systems

Polymorphism  
Overloading  
Coercion  
Generality  
Inclusion  
Reflection  
Procedures and control flow  
Parameter passing  
Variable scope  
Memory management

### Programming Paradigms

Imperative  
Declarative  
OO  
Concurrent

### Other paradigms

Interaction-based

### References

- Predefined type expressions. Basic types: `Bool`, `Int`, ...

`Bool` is the type of boolean values `True` and `False`

- Functional type **monomorphic types**

`Int → Int` is the type of function `fact` (see above), which returns the factorial of a number.

- Parameterized types **polymorphic type**

`[a] → Int` is the type of function `length`, which computes the length of a list.

**type variable**

**type constructor**

# Polymorphism

## Motivation

## Concepts

Types and type systems

### Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

- It is a feature of programming languages which permits the use of values of different types under a uniform interface
- Both functions and types can be polymorphic:
  - A function can be polymorphic with respect to some of its arguments.

*a single addition operator (+) can be applied to integers, reals, ...*
  - A data type can be polymorphic with regard to the types of its components.

*lists of elements of an arbitrary type*

## Motivation

## Concepts

Types and type systems

## Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory

management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

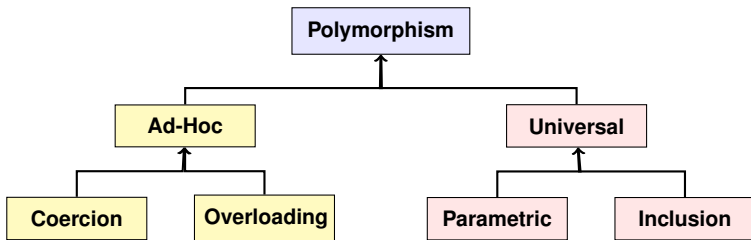
## Other paradigms

Interaction-based

## References

# Polymorphism

Different kinds of polymorphism



- Ad-hoc: a finite number of unrelated types is considered
  - Overloading
  - Coercion (type-casting)
- True or Universal: infinitely many types sharing a *common structure* are considered.
  - Parametric polymorphism (genericity)
  - Inclusion polymorphism (inheritance)

# Polymorphism. Overloading

## Ad-Hoc polymorphism: Overloading

### Motivation

### Concepts

Types and type systems

Polymorphism

**Overloading**

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory management

### Programming Paradigms

Imperative

Declarative

OO

Concurrent

### Other paradigms

Interaction-based

### References

- **Overloading:** several functions share a single name (with possibly different implementations).

- Arithmetic operators  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\dots$ : the monotypes

$$(+) :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$$

$$(+) :: \text{Float} \rightarrow \text{Float} \rightarrow \text{Float}$$

$$(+) :: \text{Complex} \rightarrow \text{Complex} \rightarrow \text{Complex}$$

$$(+) :: \text{Int} \rightarrow \text{Float} \rightarrow \text{Float}$$

correspond to different uses of  $+$

- The operator  $+$  cannot be given a polytype

$$(+) :: a \rightarrow a \rightarrow a$$

because the meaning of (and how to implement!) the ‘sum’ of characters, functions, lists, etc., is unclear



# Polymorphism. Overloading

## Example of overloading in Java (1)

In Java, overloaded methods are distinguished by examining the number and types of their parameters

```
/* overloaded methods */

int myAdd(int x, int y, int z) {
 ...
}

double myAdd(double x, double y, double z) {
 ...
}
```

# Polymorphism. Overloading

## Example of overloading in Java (2)

```
public class Overload {
 public void numbers(int x, int y) {
 System.out.println("Method that gets integer numbers");
 }
 public void numbers(double x, double y, double z) {
 System.out.println("Method that gets real numbers");
 }
 public int numbers(String st) {
 System.out.println("The length of " + st + " is " +
 st.length());
 }
 public static void main(...) {
 Overload s = new Overload();
 int a = 1;
 int b = 2;
 s.numbers(a,b);
 s.numbers(3.2,5.7,0.0);
 a = s.numbers("Madagascar");
 }
}
```

No match in the number  
or types of the parameters/result is necessary

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

**Coercion**

Genericity

Inclusion

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

# Polymorphism. Coercion

## Ad-Hoc polymorphism: Coercion

- **Coercion:** (implicit or explicit) type conversion of arguments.
- Implicit conversion usually relies on an underlying type hierarchy.

*Most languages provide implicit coercion between integer and real arguments of arithmetic operators*

- Some languages allow for an explicit coercion.

- *cast* sentence in C-like languages
- Type transformations in Java:
  - Primitive variables can change their basic type
  - From class to superclass

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

**Coercion**

Genericity

Inclusion

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

# Polymorphism. Coercion

## Example of coercion in Java

### Implicit conversion in Java:

```
int num1 = 100 // 4 bytes
long num2 = num1 // 8 bytes
```

### Explicit conversion in Java:

```
int num1 = 100 // 4 bytes
short num2 = (short) num1 // 2 bytes
char c = (char) num1 // 2 bytes

String s = Integer.toString(num1)
```

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

**Genericity**

Inclusion

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

# Polymorphism. Genericity

## Universal polymorphism: Genericity

- **Genericity/Parametric:** the function definition or class declaration has a common structure for a potentially infinite set of types

- In Haskell we can define and use generic types and functions
- In Java we can define and use generic classes and methods

# Polymorphism. Genericity

## Example of genericity in Haskell

### Motivation

### Concepts

Types and type systems

Polymorphism

Overloading

Coercion

**Genericity**

Inclusion

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory management

### Programming Paradigms

Imperative

Declarative

OO

Concurrent

### Other paradigms

Interaction-based

### References

By using a **generic type** (with type variables), we can define a data structure to represent a *dictionary* and use its entries (of any type):

```
type Entry k v = (k,v)

getKey :: Entry k v -> k
getKey (x,y) = x

getValue :: Entry k v -> v
getValue (x,y) = y
```

With a **generic function** we can compute the length of a list of elements of any type:

```
length :: [a] -> Integer
length [] = 0
length (x:xs) = 1 + (length xs)
```

# Polymorphism. Genericity

Example of genericity in Java (1/2)

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

**Genercity**

Inclusion

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

We can use a generic class (with parameters) to define an entry of a *dictionary*:

```
public class Entry<K,V>{
 private final K mKey;
 private final V mValue;

 public Entry(K k, V v){
 mKey = k;
 mValue = v;
 }

 public K getKey() {
 return mKey;
 }

 public V getValue() {
 return mValue;
 }
}
```

We can define a **generic method** to compute the length of an array of entries of *any* type:

```
public static <T> int lengthA(T[] inputArray){
 ...
}
```

# Polymorphism. Genericity

## Example of genericity in Java(2/2)

### Motivation

### Concepts

Types and type systems

Polymorphism

Overloading

Coercion

**Genercity**

Inclusion

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory management

### Programming Paradigms

Imperative

Declarative

OO

Concurrent

### Other paradigms

Interaction-based

### References

Use of a dictionary:

parameterization

```
Entry<Integer,String> elem1 = new Entry<Integer,String>;
elem1(3,"Programming");
System.out.println(elem1.getValue());
```

Use of a generic method to compute the length of an array

```
Integer[] intArray = { 1, 2, 3, 5};
Double[] doubleArray = { 1.1, 2.2, 3.3};

System.out.println("Array length =" + lengthA(intArray));
System.out.println("Array length =" + lengthA(doubleArray));
```



# Polymorphism. Genericity

## Some considerations about genericity in Java

### Motivation

### Concepts

Types and type systems

Polymorphism

Overloading

Coercion

**Genericity**

Inclusion

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory management

### Programming Paradigms

Imperative

Declarative

OO

Concurrent

### Other paradigms

Interaction-based

### References

- A generic class is a normal one, except that the declaration uses a **variable type** (parameter), that will be instantiated (to a type) when used.
- We can use other generic classes within a generic class
- Generic classes may have several parameters

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

**Inclusion**

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

# Polymorphism. Inclusion

## Universal Polymorphism: Inclusion/Inheritance

- **Inclusion or Inheritance:** the definition of a function is made on types that are related by an inclusion hierarchy.
- In OO Programming, inheritance is the usual mechanism for software **reuse** and **extensibility**.

*Classes are organized into a hierarchical structure based on class inheritance*

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

**Inclusion**

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

# Polymorphism. Inclusion

## Universal Polymorphism: Inclusion/Inheritance

### IDEA:

If a class B inherits from a class A, then B has the structure and behavior of class A. Besides, we can:

- add new attributes to B
- add new methods to B

Depending on the language, we will be able to:

- redefine inherited methods
- inherit from several classes (a Java class may inherit from a single class, though)

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

**Inclusion**

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory

management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other

## paradigms

Interaction-based

## References

# Polymorphism. Inclusion

Example of inheritance in Java (1/2)

```
public class Bicycle {
 protected int cadence;
 protected int gear;
 protected int speed;
 public Bicycle (int startCad, int startSpeed,
 int startGear) {
 cadence = startCad;
 speed = startSpeed;
 gear = startGear;
 }
 public void setCadence(int newValue) {
 cadence = newValue; }
 public void setGear(int newValue) {
 gear = newValue; }
 public void applyBrake(int decrement) {
 speed -= decrement; }
 public void speedUp(int increment) {
 speed += increment; }
}
```

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

**Inclusion**

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory

management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other

## paradigms

Interaction-based

## References

# Polymorphism. Inclusion

## Example of inheritance in Java (2/2)

```
public class MountainBike extends Bicycle {
 public int seatHeight;
 public MountainBike(int startHeight, int startCad,
 int startSpeed, int starteGear){
 super(startCad, startSpeed, StartGear);
 seatHeight = startHeight;
 }
}
```

- Subclasses are introduced using the keyword **extends**
- New attributes and methods can be added. Methods can be redefined as well.

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

**Inclusion**

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory

management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other

## paradigms

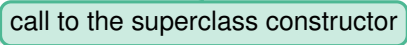
Interaction-based

## References

# Polymorphism. Inclusion

Example of inheritance in Java (2/2)

```
public class MountainBike extends Bicycle {
 public int seatHeight;
 public MountainBike(int startHeight, int startCad,
 int startSpeed, int starteGear){
 super(startCad, startSpeed, StartGear);
 seatHeight = startHeight;
 }
}
```



call to the superclass constructor

- Subclasses are introduced using the keyword **extends**
- New attributes and methods can be added. Methods can be redefined as well.

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

**Inclusion**

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory

management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

# Polymorphism. Inclusion

About inheritance in Java (1/2)

- Class `Object` is the topmost class in any Java hierarchy
- A class which is declared to be `final` cannot specialize into any subclass
- Only simple inheritance is allowed in Java
- Superclass variables may be instantiated to objects of a subclass, but **not** vice versa

## Example of a valid assignment

```
Bicycle b;
MountainBike m = new MountainBike(75, 90, 25, 8);
b = m
```

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

**Inclusion**

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

# Polymorphism. Inclusion

About inheritance in Java (2/2)

- In Java qualifiers preceding attributes and methods are used to establish visibility of instance variables and methods from a class
  - **Private**: no visible from subclasses or other classes.  
*In order to read or write private attributes, appropriate methods must be provided*
  - **Protected**: visible from subclasses only.
  - **Public**: no restrictions. Public members remain public in any other subclass in the hierarchy.



# Polymorphism. Inclusion

Example: redefining inherited methods in Java (1/2)

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

**Inclusion**

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other

## paradigms

Interaction-based

## References

```
public class Employee {
 String name;
 int nEmployee, salary;
 static private int counter = 0;
 public Employee (String name, int salary){
 this.name = name;
 this.salary = salary;
 nEmployee ++ counter;
 }
 public void increaseSalary(int wageRaise){
 salary += (int) (salary*wageRaise/100);
 }
 public String toString(){
 return "Num. Employee " + nEmployee +
 " Name: " + name + " Salary: " + salary;
 }
}
```

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

**Inclusion**

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

# Polymorphism. Inclusion

Example: redefining inherited methods in Java (2/2)

```
public class Executive extends Employee{
 int budget;
 void assignBudget(int b){
 budget = b;
 }
 public String toString(){
 String s = super.toString();
 s = s + " Budget: " + budget;
 return s;
 }
}
```

## Example of use:

```
Executive boss = new Executive("Thomas Turner", 1000);
boss.assignBudget(1500);
boss.increaseSalary(5);
```

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

**Inclusion**

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

# Polymorphism. Inclusion

## Inheritance in Java: Abstract classes

- An abstract class is one which is declared to be `abstract`
  - Classes containing an `abstract` method must be `abstract`.
  - `abstract` methods have no implementation.
  - Abstract classes cannot be instantiated.
- Non-abstract subclasses of abstract classes must implement each inherited abstract method

# Polymorphism. Inclusion

Example: use of abstract classes in Java (1/2)

```
public abstract class Shape {
 private float x, y; // Position of the shape
 public Shape (float initX, float initY){
 x = initX; y = initY;
 }
 public void move(float incX, float incY){
 x = x+incX; y = y+incY;
 }
 public float getX(){ return x; }
 public float getY(){ return y; }
 public abstract float perimeter();
 public abstract float area();
}
```

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

**Inclusion**

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory

management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

# Polymorphism. Inclusion

Example: use of abstract classes in Java (2/2)

```
public class Square extends Shape {
 private float side;
 public Square (float initX, float initY, float initSide){
 super(initX,initY); // Call to superclass constructor
 side = initSide;
 }
 public float perimeter(){ return 4*side; }
 public float area(){ return side*side; }
}

public class Circle extends Shape {
 private float radius;
 public Circle(float initX, float initY, float initRadius){
 super(initX,initY); // Call to superclass constructor
 radius = initRadius;
 }
 public float perimeter(){ return 2*pi*radius; }
 public float area(){ return pi*radius*radius; }
}
```

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

**Inclusion**

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

# Polymorphism. Inclusion

Example: use of abstract classes in Java (2/2)

```
public class Square extends Shape {
 private float side;
 public Square (float initX, float initY, float initSide){
 super(initX,initY); // Call to superclass constructor
 side = initSide;
 }
 public float perimeter(){ return 4*side; }
 public float area(){ return side*side; }
}

public class Circle extends Shape {
 private float radius;
 public Circle(float initX, float initY, float initRadius){
 super(initX,initY); // Call to superclass constructor
 radius = initRadius;
 }
 public float perimeter(){ return 2*pi*radius; }
 public float area(){ return pi*radius*radius; }
}
```

How do we extend the example to deal with a new form (e.g., triangles)?

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

**Inclusion**

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

## Polymorphism. Inclusion

## Inheritance in Java: Interfaces

An interface **declares** attributes and operations to be defined in classes implementing (**implements**) such an interface

```
public interface MyInterface {
 public int method1(...);
 ...}
```

```
public class MyClass implements MyInterface {
 public int method1(...) {...}
 ...}
```

- Interface methods are abstract and public
- Attributes are static and final (i.e., constant)
- Classes are allowed to implement several interfaces
- Interfaces are allowed to inherit from other interfaces (**extends**)

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

**Inclusion**

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

# Question: Inclusion and genericity

Consider the following class definitions:

```
class Shape { /*...*/ }
class Circle extends Shape { /*...*/ }
class Rectangle extends Shape { /*...*/ }
class Node<T> { /*...*/ }
```

Is there any compilation error in the following code? Why?

```
Node<Circle> nc = new Node<Circle>();
Node<Shape> ns = nc;
```



## Concepts

## Types and type systems

## Polymorphism

## Overloading

## Coercion

### Genericity

### Inclusion

### Reflection

## Procedures and control flow

## Parameter passing

## Variable scope

## Memory

management

## Programming Paradigms

### Imperative

### Declarative

00

Concurrent

## Other paradigms

Interaction-based

## References

## Question: Inclusion and genericity

Answer the following questions:

- Can an interface inherit from a class?
- Can we obtain instances from an interface?

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

## Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

## What is reflection

When looking yourself at the mirror, you may:

- See your reflected image and
  - React according to what you see
- 
- In **programming languages**, reflection enables a program to:
    - see its own structure and
    - manipulate its own code
  - LISP was the first language with reflection; it is also present in some scripting languages.

With reflection we can write programs that monitor their own execution; eventually a program can introduce runtime (self)modifications to dynamically adapt its behavior to the environment

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

**Reflection**

Procedures and control flow

Parameter passing

Variable scope

Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

# Reflection

## Languages with reflection

- Languages with reflection treat their own instructions as values of a specific datatype which is a first-class type of the language (languages without reflection just see strings).
- Reflection can be seen as the ability of a language to be its own **metalanguage**

*We call metalanguage to the language which is used to write metaprograms (programs dealing with other programs: compilers, analyzers, etc.)*

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

**Reflection**

Procedures and control flow

Parameter passing

Variable scope

Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

# Reflection

Use with caution

- A bad use of reflection
  - may hinder performance

*if you can do it without reflection, do not use it*

- may compromise security by showing up unexpected details of the code in certain contexts

*reflection breaks abstraction, and private attributes and methods can be reached*

- It is an advanced but simple feature, specially in functional languages due to the natural data/program duality of these languages (homoiconicity).

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

**Reflection**

Procedures and control flow

Parameter passing

Variable scope

Memory

management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

# Reflection

## Reflection in Java

- Reflection is available in Java through an specific library (`java.lang.reflect`)

The library provides classes for the structured representation of information about classes, variables, methods, etc.

- Reflection in Java permits the runtime inspection of classes, interfaces, attributes and methods without knowing the names of the interfaces, attributes and methods in compile time. In this way, we can
  - Read a `String` from the keyboard and use it to create an object and give it a name, or call a method with this name
  - reading all getter methods (get) or modifiers (set) of a class
  - accessing private fields and methods of a class.

# Reflection

## Example: use of reflection in Java

```
import java.lang.reflect.*;

public class MyClass {...}

...
Class myClassObj = new MyClass();
// get the class information:
Class<? extends MyClass> objMyClassInfo =
 myClassObj.getClass();

// get the fields:
Field[] allDeclaredVars = objMyClassInfo.getDeclaredFields();
// travel the fields:
for (Field variable : allDeclaredVars) {
 System.out.println("Name of GLOBAL VARIABLE: " +
 variable.getName());
}
```

### Further methods defined in Class:

```
Constructor[] getConstructors();
Field[] getDeclaredFields();
Method[] getDeclaredMethods();

...
```

# Procedures and control flow

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

**Procedures and control flow**

Parameter passing

Variable scope

Memory

management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

Some important concepts in programming languages concern the control flow of program execution, and how calls to functions and procedures are processed.

- **Parameter passing.** When a method or function is called there is a change in the execution context which can be done in different ways. We discuss the main ones.
- **Variable scope.** We need to know whether an object or variable can be reached from a given program point. This can be done statically or dynamically.

# Parameter passing

## Motivation

### Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

**Parameter passing**

Variable scope

Memory management

### Programming Paradigms

Imperative

Declarative

OO

Concurrent

### Other paradigms

Interaction-based

### References

The use of functions, methods and procedures is an essential abstraction mechanism in program development. They solve specific tasks which are executed after a **call** with an appropriate parameterization.

- **CALL:**  $f(e_1, \dots, e_n)$  with  $e_1, \dots, e_n$  being expressions.
  - when the call is executed, the control flow moves to the body of function  $f$ . After completion, the control flow returns 'below' the program point issuing the call
  - $e_1, \dots, e_n$  are called **input** or **actual parameters**
- **DECLARATION:**  $f(x_1, \dots, x_n)$  with  $x_1, \dots, x_n$  being variables.
  - $x_1, \dots, x_n$  are called **formal parameters**
  - Formal parameters are variables which are local to the body of function  $f$  (i.e., reachable within the body of  $f$  only)



# Parameter passing

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

**Parameter passing**

Variable scope

Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

A number of parameter passing mechanisms can be considered:

- Call by value
- Call by reference/call by address
- Call by need

There are other parameter passing mechanisms, but these are the most frequently used in programming languages

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

**Parameter passing**

Variable scope

Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

# Parameter passing

## Call by value

The values  $v_i$  of the input parameters  $e_i$  are computed and then copied into the formal parameters  $x_i$

- within the body of the function the value is given a different memory address

```
void inc(int v)
{
 v = v + v;
}

...
int a = 10;
inc(a);
```

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

**Parameter passing**

Variable scope

Memory

management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

## Parameter passing

## Call by value

The values  $v_i$  of the input parameters  $e_i$  are computed and then copied into the formal parameters  $x_i$

- within the body of the function the value is given a different memory address

```
void inc(int v)
{
 v = v + v;
}

...
int a = 10;
inc(a);
```

$a = 10$

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

**Parameter passing**

Variable scope

Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

## Parameter passing

## Call by value

The values  $v_i$  of the input parameters  $e_i$  are computed and then copied into the formal parameters  $x_i$

- within the body of the function the value is given a different memory address

```
void inc(int v)
{
 v = v + v;
}

...
int a = 10;
inc(a);
```

$a = 10$

## Function call:

Value 10 is copied into the formal parameter  $v$

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

**Parameter passing**

Variable scope

Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

## Parameter passing

## Call by value

The values  $v_i$  of the input parameters  $e_i$  are computed and then copied into the formal parameters  $x_i$

- within the body of the function the value is given a different memory address

```
void inc(int v)
{
 v = v + v;
}

...
int a = 10;
inc(a);
```

v = 10

a = 10

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

**Parameter passing**

Variable scope

Memory

management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

## Parameter passing

## Call by value

The values  $v_i$  of the input parameters  $e_i$  are computed and then copied into the formal parameters  $x_i$

- within the body of the function the value is given a different memory address

```
void inc(int v)
{
 v = v + v;
}

...
int a = 10;
inc(a);
```

v = 20

a = 10

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

**Parameter passing**

Variable scope

Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

## Parameter passing

## Call by value

The values  $v_i$  of the input parameters  $e_i$  are computed and then copied into the formal parameters  $x_i$

- within the body of the function the value is given a different memory address

```
void inc(int v)
{
 v = v + v;
}

...
int a = 10;
inc(a);
```

$a = 10$

- Variable  $a$  is NOT modified: a working copy is used inside the body of function `inc`.

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

**Parameter passing**

Variable scope

Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

## Parameter passing

## Call by reference

A **reference** to the memory is passed so that the body of function  $f$  works on the same memory object

- Input parameters  $e_i$  which are **not** variables are treated as in call by value
- In contrast, if  $e_i$  is a variable (e.g.,  $y_i$ ), then any assignment to the formal parameter  $x_i$  in the body of  $f$  also modifies the value associated to variable  $y_i$

```
void inc(int v)
{
 v = v + v;
}

...
int a = 10;
inc(a);
```



## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

**Parameter passing**

Variable scope

Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

## Parameter passing

## Call by reference

A **reference** to the memory is passed so that the body of function *f* works on the same memory object

```
void inc(int v)
{
 v = v + v;
}

...
int a = 10;
inc(a);
```

a = 10

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

**Parameter passing**

Variable scope

Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

## Parameter passing

## Call by reference

A **reference** to the memory is passed so that the body of function *f* works on the same memory object

```
void inc(int v)
{
 v = v + v;
}
```

```
...
```

```
int a = 10;
```

```
inc(a);
```

```
a = 10
```

## Function call:

The formal parameter *v* is given the memory address of *a*

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

**Parameter passing**

Variable scope

Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

## Parameter passing

## Call by reference

A **reference** to the memory is passed so that the body of function *f* works on the same memory object

```
void inc(int v)
```

```
{
```

*v = 10*

```
 v = v + v;
```

```
}
```

```
...
```

```
int a = 10;
```

```
inc(a);
```

*a = 10*

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

**Parameter passing**

Variable scope

Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

## Parameter passing

## Call by reference

A **reference** to the memory is passed so that the body of function *f* works on the same memory object

```
void inc(int v)
{
 v = v + v;
}
```

*v* = 20

```
...
int a = 10;
inc(a);
```

*a* = 20

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

**Parameter passing**

Variable scope

Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

## Parameter passing

## Call by reference

A **reference** to the memory is passed so that the body of function *f* works on the same memory object

```
void inc(int v)
{
 v = v + v;
}

...
int a = 10;
inc(a);
```

**a = 20**

- Variable *a* IS modified: the working memory area for *v* and *a* is the same

# Parameter passing

Call by need

- Expressions from the input parameters are evaluated **only when used** in the body of the function
- Used in some functional languages

## Example

Consider the following function that returns the double of the second argument

```
sel2nd x y = 2*y
```

With call-by-need, a call `sel2nd (2*3) 5` would not evaluate the value of expression `2*3` as it is not used in the expression defining the function.

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

**Parameter passing**

Variable scope

Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

# Parameter passing

Some remarks

- In call by value, input expressions  $e_i$  are first evaluated; then, the value  $v_i$  is copied into the local variable  $x_i$  (formal parameter). This is in contrast to call by need.
- In call by reference, input expressions  $e_i$  which are not variables are also evaluated and the computed value  $v_i$  is also copied into the formal parameter  $x_i$ .

# Variable scope (1/2)

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

Parameter passing

**Variable scope**

Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

- A variable is a *name* referring to a memory address
- During the execution of the program, the variables, functions, constants, etc., which are identified by the names in the program can be reachable, or not
- The scope of a name is the code portion where it is visible (its value can be read/written).



# Variable scope (2/2)

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

Parameter passing

**Variable scope**

Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

- The instant when the binding (association) is established is called the link time
  - With **static scope**, it is defined in *compilation time*
  - With **dynamic scope**, it is defined *during the execution*
- Modern programming languages use static scope.
- Each programming language fixes its particular variable scope approach
  - Java uses `private`, `public`, and `protected` attributes and also the class/packages hierarchy system.

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

Parameter passing

**Variable scope**

Memory

management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other

## paradigms

Interaction-based

## References

## Variable scope

## Example: computing the variable scope (1/2)

```

1 program scope;
2 type
3 TArray : array [1..3]
4 of integer;
5 var
6 a: TArray;
7 procedure one;
8 procedure two;
9 a : TArray;
10 begin { * two *}
11 a[1] := 1;
12 a[2] := 2;
13 a[3] := 3;
14 swap(1, 2);
15 writeln(a[1], ' ', a[2], ' ', a[3]);
16 end { * two *};

17 procedure swap(i, j: integer);
18 var aux : integer;
19 begin { * swap *}
20 aux := a[i];
21 a[i] := a[j];
22 a[j] := aux;
23 end { * swap *};
24 begin { * one *}
25 a[1] := 0;
26 a[2] := 0;
27 a[3] := 0;
28 two;
29 end { * one *};
30 begin { * scope *}
31 one;
32 end { * scope *}

```

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory

management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

## Variable scope

Example: computing the variable scope (1/2)

```

1 program scope;
2 type
3 TArray : array [1..3]
4 of integer;
5 var
6 a: TArray;
7 procedure one;
8 procedure two;
9 a : TArray;
10 begin { * two *}
11 a[1] := 1;
12 a[2] := 2;
13 a[3] := 3;
14 swap(1, 2);
15 writeln(a[1], ' ', a[2], ' ',
16 a[3]);
17 procedure swap(i, j: integer);
18 var aux : integer;
19 begin { * swap *}
20 aux := a[i];
21 a[i] := a[j];
22 a[j] := aux;
23 end { * swap *}
24 begin { * one *}
25 a[1] := 0;
26 a[2] := 0;
27 a[3] := 0;
28 two;
29 end { * one *}
30 begin { * scope *}
31 one;
32 end { * scope *}

```

Which values are stored in array `a` at the end of the execution?  
 What does `writeln` print on the screen?

# Variable scope

Example: computing the variable scope (2/2)

## Assuming static scope...

Which values are stored in array `a` at the end of the execution? What does `writeln` print on the screen?

### In compilation time:

- Lines 25 to 27 (body of function `one`) refer to variable `a` which is bound to the global variable in line 6 (note that `one` has no local declaration for any variable)
- Lines 11 to 15 (body of function `two`) refer to a variable `a` which is bound to the local variable of `two` defined in line 9: local variables hide global variables with the same name
- Lines 20 to 22 (body of function `swap`) refer to a variable `a` which is bound to the global variable in line 6: like `two`, procedure `swap` is local to `one`

# Variable scope

Example: computing the variable scope (2/2)

## Assuming static scope...

Which values are stored in array `a` at the end of the execution? What does `writeln` print on the screen?

Therefore:

- The main program calls to procedure `one` (line 31).
- The values in the global array are initialized to 0, 0 and 0 (lines 25 to 27)
- A call to `two` initializes a local array to 1, 2 and 3; the global array does not change (lines 11 to 13)
- The call to `swap` changes the values of the global array to 0, 0, and 0. The local array of `two` does not change.
- The values of the local array (1, 2 and 3) are printed

# Memory management

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

Parameter passing

Variable scope

**Memory management**

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

Concerns the different methods and procedures which are used to **maximize the memory use**.

## Impact in the design of the programming language

Some design choices can only be explained by the desire of the language designers to use a particular memory management technique.

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

Parameter passing

Variable scope

## Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

## Memory management

## The need of memory management

Some program and data elements requiring storage during program execution

- The code of the **compiled program**
- **Temporaries** in expression evaluation (e.g.,  $(x + y) \times (u + v)$ ) and in parameter passing (e.g. when a subprogram is called, a list of actual parameters must be evaluated and the resulting values stored until the evaluation of the entire list is complete)
- Subprogram **call** and **return** operations
- Input-output **Buffers**
- Data structure **insertion and destruction** operations (e.g. `new` in Java or `dispose` in Pascal)
- Component **insertion and deletion** operations in data structures (e.g. the Perl `push` function adds an element to an array)

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

Parameter passing

Variable scope

## Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

# Memory management

## Memory management approaches

Main kinds of memory management approaches:

### Static allocation

Computed and assigned in compilation time

- Efficient but incompatible with recursion or dynamic data structures

### Dynamic allocation

Computed and assigned during the execution

- stack-based
- heap-based



## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

Parameter passing

Variable scope

## Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

# Memory management

## Static allocation

Memory allocation computed in compilation time. It remains fixed throughout the execution.

It can be used with:

- **Global variables**
- Program's **machine language** translation
- **Variables** that are **local** to a single routine but retain their values from one invocation to the next
- Numeric and string-valued **constant literals**
- **Tables** produced by compilers that are used by run-time support routines for debugging, dynamic-type checking, . . .

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

Parameter passing

Variable scope

## Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

# Memory management

## Static allocation

Memory allocation computed in compilation time. It remains fixed throughout the execution.

It can be used with:

- Global variables
- Program's machine language translation
- Variables that are local to a single routine but retain their values from one invocation to the next
- Numeric and string-valued constant literals
- Tables produced by compilers that are used by run-time support routines for debugging, dynamic-type checking, . . .

Efficient but incompatible with recursive subprogram calls and data structures whose size depends on computed or input data

# Memory management

## Dynamic allocation using a stack

The simplest run-time storage-management technique to handle the activation record in function calls during the program execution (a pointer to the top of the stack suffices)

## Stack-based allocation

- Free storage when the execution starts is set up as a sequential block in memory (a stack)
- As storage is allocated, it is taken from sequential allocations in this stack block beginning at one end
- Storage must be freed in the reverse order of allocations so that a block of storage being freed is always at the top of the stack

## Motivation

## Concepts

Types and type systems

Polymorphism

Overloading

Coercion

Genericity

Inclusion

Reflection

Procedures and control flow

Parameter passing

Variable scope

Memory management

## Programming Paradigms

Imperative

Declarative

OO

Concurrent

## Other paradigms

Interaction-based

## References

# Memory management

Dynamic allocation using a *heap*

A **heap** is a region of storage in which subblocks can be allocated and deallocated at *arbitrary times*

- This kind of storage management is needed when the language allows for data structures (like sets or lists) whose size may change during the execution
- The allocated elements are always of the same **fixed size, or of variable size**
- Deallocation is
  - explicit (e.g. C, C++, Pascal)
  - implicit (when it is no longer possible to reach the allocated element from any program variable)

# Memory management

Dynamic allocation using a *heap*

A **heap** is a region of storage in which subblocks can be allocated and deallocated at *arbitrary times*

- This kind of storage management is needed when the language allows for data structures (like sets or lists) whose size may change during the execution
- The allocated elements are always of the same **fixed size, or of variable size**
- Deallocation is
  - explicit (e.g. C, C++, Pascal)
  - implicit (when it is no longer possible to reach the allocated element from any program variable)
- **Garbage collector**: language mechanism that identifies unreachable elements and returns them to the free-space