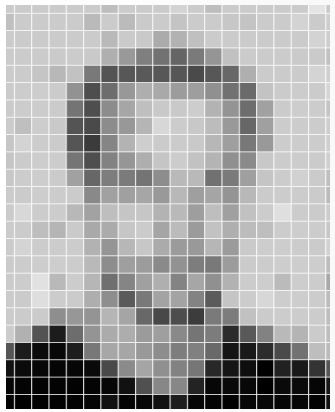
The EIFFEL Programming Language



Claus Brabrand

(((brabrand@itu.dk)))

Associate Professor, Ph.D.

(((Software & Systems)))

IT University of Copenhagen

[**Acknowledgement:** Some of the slides are inspired by a presentation on Eiffel by **Martin Nordio**, ETHZ]

Eiffel

- Statically typed “pure” **OO** programming language (like Java and C#)
- Inherent support for ***"Design by Contract"*** *(DbC)*:
  - ***pre-conditions***, ***post-conditions***, ***invariants***
- Promotes a particular way of thinking about the design, architecture, and implementation of **Object-Oriented Software Systems !**

Recap: Virtualization

```
class A {
  void m() {
    print("A");
  }
}

class B extends A {
  void m() {
    print("B");
  }
}

class C extends A {
  void m() {
    print("C");
  }
}
```

```
C c = new C();
c.m(); // what is printed here?
```

Answers:

- **A)** "A"
- **B)** "B"
- **C)** "C"
- **D)** *** compile-time error! ***
- **E)** *** runtime error! ***
- **F)** I don't know ?!?

Recap: Virtualization

```
class A {
  void m() {
    print("A");
  }
}

class B extends A {
  void m() {
    print("B");
  }
}

class C extends A {
  void m() {
    print("C");
  }
}
```

```
A a = new B();
a.m(); // what is printed here?
```

Answers:

- **A)** "A"
- **B)** "B"
- **C)** "C"
- **D)** *** compile-time error! ***
- **E)** *** runtime error! ***
- **F)** I don't know ?!?

Recap: Virtualization

```
class A {
  void m() {
    print("A");
  }
}

class B extends A {
  void m() {
    print("B");
  }
}

class C extends A {
  void m() {
    print("C");
  }
}
```

```
B b = new A();
b.m(); // what is printed here?
```

Answers:

- **A)** "A"
- **B)** "B"
- **C)** "C"
- **D)** *** compile-time error! ***
- **E)** *** runtime error! ***
- **F)** I don't know ?!?

Recap: Object Orientation

OO benefits?

- **1)** Real World Apprehension (modeling)
- **2)** Stability of Design
- **3)** Code Reuse

Scandinavia

America

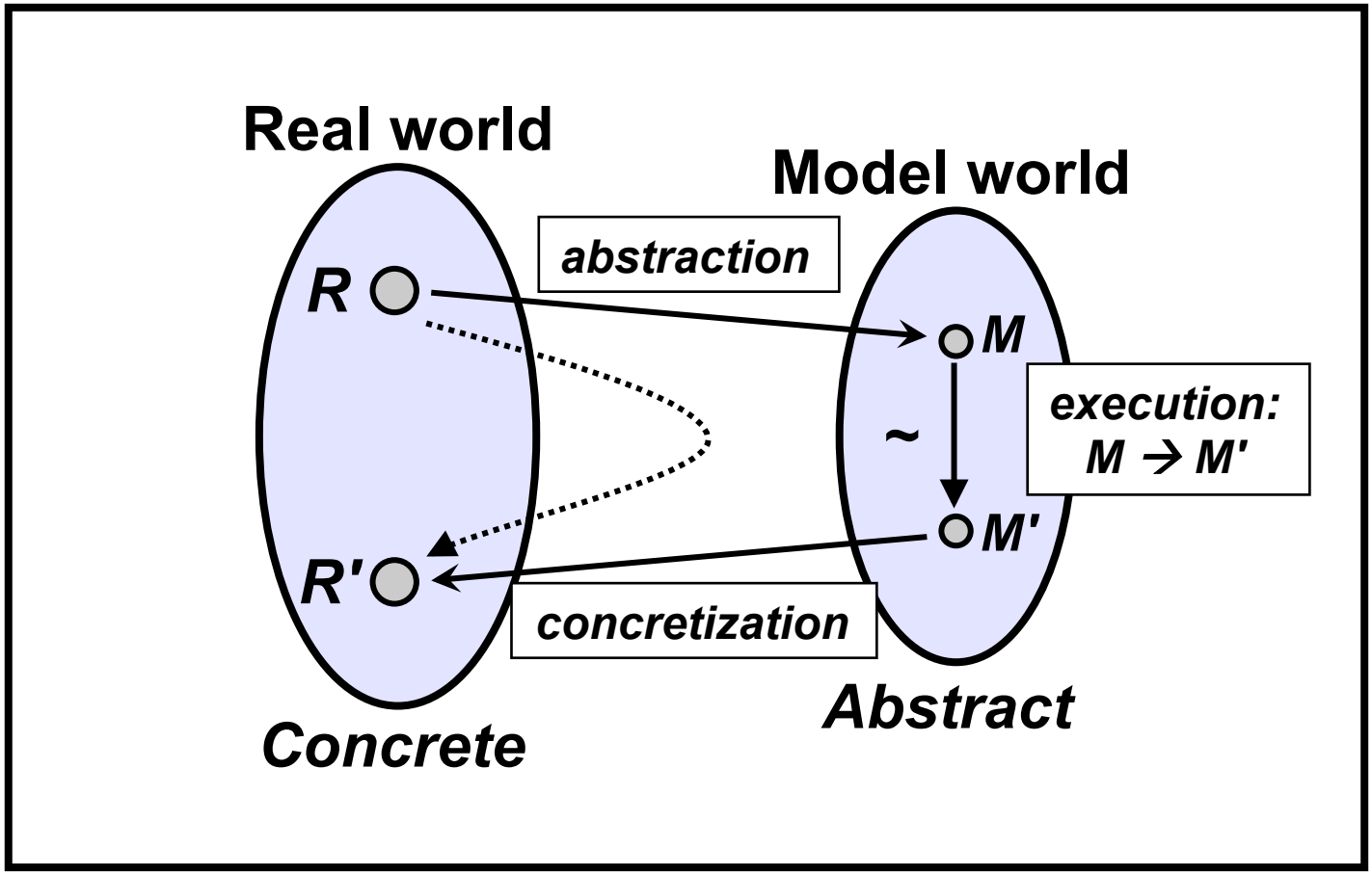
- **+)** It's what everyone else is using

Common Motivation

Modeling Reality

Computing = "Informations·representations·transformation"

- **Transformation** of **Representation** of **Information**:



Recap: Inheritance

DEF:

```
class Vehicle {
  void move() {
    ...
  }
}
```

OO benefits:

- **1)** Real World Apprehension
- **2)** Stability of Design
- **3)** Code Reuse

```
class Train
extends Vehicle {
  void move() {
    ...
  }
}
```

```
class Plane
extends Vehicle {
  void move() {
    ...
  }
}
```

USE:

```
Vehicle v;
...
v.move(); // shared: same method for a train and a plane! :-)
```

Recap: Inheritance

DEF:

```
class Vehicle {
  void move() {
    ...
  }
}
```

OO benefits:

- **1)** Real World Apprehension
- **2)** Stability of Design
- **3)** Code Reuse

```
class Train
extends Vehicle {
  void move() {
    ...
  }
}
```

```
class Car
extends Vehicle {
  void move() {
    ...
  }
}
```

```
class Plane
extends Vehicle {
  void move() {
    ...
  }
}
```

USE:

```
Vehicle v;
...
v.move();  // unchanged (works for cars without modification)!
```

Recap: Pre-OO Alternative

DEF:

```
datatype VEHICLE = union {// union type!
  train: int train_id; .. // vars for train
| plane: int plane_id; .. // vars for plane

}
```

~~OO benefits:~~

- ~~1) Real World Apprehension~~
- ~~2) Stability of Design~~
- ~~3) Code Reuse~~

USE:

```
void reserve(VEHICLE v) {
  if (v is train) {
    ... v.train_id ...
  } elseif (v is plane) {
    ... v.plane_id ...


  } else {
    error("runtime error: uninitialized vehicle!");
  }
}
```

Recap: Pre-OO Alternative

DEF:

```
datatype VEHICLE = union {// union type!
  train: int train_id; .. // vars for train
| plane: int plane_id; .. // vars for plane
| car:   int car_id;   .. // vars for car
}
```

~~OO benefits:~~

- ~~1) Real World Apprehension~~
- ~~2) Stability of Design~~
- ~~3) Code Reuse~~

USE:

```
void reserve(VEHICLE v) {
  if (v is train) {
    ... v.train_id ...
  } elseif (v is plane) {
    ... v.plane_id ...
  } elseif (v is car) {
    ... v.car_id ...
  } else {
    error("runtime error: uninitialized vehicle!");
  }
}
```

EIFFEL

Class Declaration

- **Java:**

```
class MyAccount {

   ...

}
```

Convention: so-called *CamelCase* for class names

- **Eiffel:**

```
class MY_ACCOUNT

   ...

end
```

Convention: capitalization (possibly with underscores) for class names

Syntax: `"begin .. end"` style instead of curly braces `"{ .. }"`

Constructors

- **Java:**

```
class Account {

  Account() {
    ...
  }

  Account(int b) {
    ...
  }

}
```

- **Eiffel:**

```
class ACCOUNT
  create
    make
    make_balance

  feature
    make
      do ... end

    make_balance(i: INTEGER)
      do ... end
end
```

a ***routine*** *(method)* named "***make***"

another ***routine***, "***make_balance***"

Constructors have names; use "**create**" clause to declare ***routines*** (methods) as constructors

Constructors

- **Java:**

```
class Account {

  Account() {
    ...
  }

  Account(int b) {
    ...
  }

  Account(String s) {
    ...
  }

}
```

- **Eiffel:**

```
class ACCOUNT
  create
    make
    make_balance
    make_name

  feature
    make
      do ... end

    make_balance(i: INTEGER)
      do ... end

    make_name(s: STRING)
      do ... end
end
```

Constructors are given **logical names**;
i.e., "**`string`**" *vs* "**`make_name`**"

Basic Types

Java:

- boolean <-> Boolean
- char, byte <-> Character
- short, int <-> Integer
- long <-> Long
- float <-> Float
- double <-> Double
- String

NB: *Boxing* / *Unboxing*:
"primitive types" <->
"object wrapper classes"

```
int i;
...
Integer integer = new Integer(i);
```

Boxing: `int -> Integer`

```
Integer integer;
...
int i = integer.intValue();
```

Unboxing: `int <- Integer`

Eiffel:

- BOOLEAN
- CHARACTER
- INTEGER
- INTEGER_64
- REAL
- DOUBLE
- STRING

Just normal classes!
(no boxing/unboxing issues)

No Overloading

- **Java:**

```
class Printer {

  void print(boolean b) {
    ...
  }

  void print(int i) {
    ...
  }

  void print(String s) {
    ...
  }

}
```

- **Eiffel:**

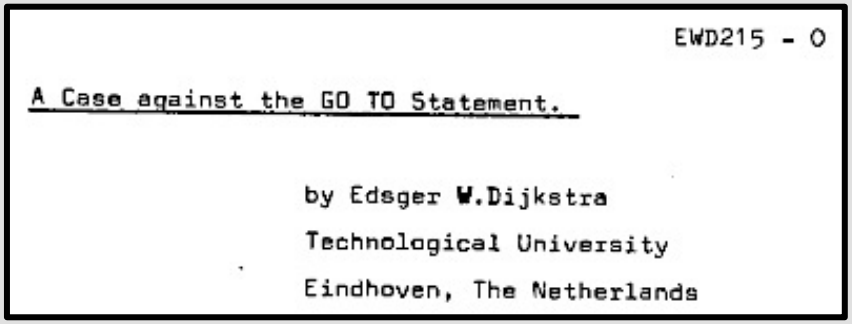
```
class PRINTER

  feature
    print_boolean(b: BOOLEAN)
      do ...
      end

    print_int(i: INTEGER)
      do ...
      end

    print_string(s: STRING)
      do ...
      end
end
```

No overloading in Eiffel; ***use type suffixes*** to distinguish routines

Restriction not Necessarily Bad

Example (1/2):

- **Gotos** (e.g., in Basic):
  - **Syntax:** `goto 120;`
  - **Semantics:** *jump (go) to program line number 120*
  - Structural reasoning breaks down!
    - Cannot reason structurally
    - In particular, you'll never know if someone is or will be "jumping into the middle" of your code


EWD215 - 0

A Case against the GO TO Statement.

by Edsger W.Dijkstra
Technological University
Eindhoven, The Netherlands

"Go-to Statement Considered Harmful"
by **Edsgar W. Dijkstra**
(Communications of the ACM, 1968, 11(3): 147–148)

Restriction not Necessarily Bad

Example (2/2):

- **Pointer Arithmetic** (e.g., in C):
  - **Syntax:** `*(p+7)` // Assuming p pointer `Person *p;`
  - **Semantics:** *go to the cell p is pointing to; forward 7 Person structures (i.e., sizeof(Person) number of cells); then dereference the content of that cell.* Like array:
  - **In fact:** `p[7]` is translated into `*(p+7)` *after* parsing, but *before* semantic analysis!
  - **Consequently:**

    `p[7]` ⇒ `*(p+7)` $\cong$ (*commutativity of addition '+'*) `*(7+p)` ⇐ `7[p]`

  - I.e., you might just as well write array lookup as: `7[p]`

No Overloading

- **Java:**

```
class Printer {

  void print(boolean b) {
    ...
  }

  void print(int i) {
    ...
  }

  void print(String s) {
    ...
  }

}
```

- **Eiffel:**

```
class PRINTER

  feature
    print_boolean(b: BOOLEAN)
      do ...
      end

    print_int(i: INTEGER)
      do ...
      end

    print_string(s: STRING)
      do ...
      end
end
```

No overloading in Eiffel; ***use type suffixes*** to distinguish routines

Object Creation

- **Java:**

```
class Bank {

  void payBill() {
    Account a;
    a = new Account();
  }

}
```

- **Eiffel:**

```
class BANK

  feature
    pay_bill
      local
        a: ACCOUNT
      do
        create a.make
      end
end
```

local variable declaration

Use "**create**" for object creation...

...and name the constructor explicitly (aka, ***creation call***)

Multiple Object Creation

- **Java:**

```
class Bank {

  void payBill() {
    Account a1, a2;
    a1 = new Account();
    a2 = new Account(100);
  }

}
```

- **Eiffel:**

```
class BANK

  feature
    pay_bill
      local
        a1, a2: ACCOUNT
      do
        create a1.make ;
        create a2.make_balance(100)
      end
end
```

Semi-colon ';' optional
(for statement separation)

Default Object Creation

- **Java:**

```
class Main {

  Main(int i) {
    super();
    ...
  }

}
```

Implicit invocation of "**super**" as first statement of all constructors (for object creation and initialization, by invoking the constructor of the superclass: "**Object**")

...inheriting the methods:

- *clone()*
- *equals(Object)*
- *finalize()*
- *getClass()*
- *hashCode()*
- *toString()*
- *notify()* / ...
- *wait()* / ...

- **Eiffel:**

```
create a
```

Default creation routine: "**default_create**" (inherited from "**ANY**")

=

```
create a.default_create
```

Methods inherited from "**ANY**":

- *same_type(other)*
- *equal(other)*
- *deep_equal(other)*
- *clone(other)*
- *deep_clone(other)*
- *print(x)*
- *argument(n)*
- *system(s)*
- ...

Default Object Creation

- **Java:**

```
class Main {

  Main(int i) {
    super();
    ...
  }
}
```

Implicit invocation of "**super**" as first statement of all constructors (for object creation and initialization, by invoking the constructor of the superclass: "**Object**")

...inheriting the methods:

- *clone()*
- *equals(Object)*
- *finalize()*
- *getClass()*
- *hashCode()*
- *toString()*
- *notify() / ...*
- *wait() / ...*

- **Eiffel:**

```
create a
```

Default creation routine: "**default_create**" (inherited from "**ANY**")

=

```
create a.default_create
```

```
class BANK inherit ANY

  redefine
    default_create
  end

  feature
    default_create
      do ... end
end
```

"**default_create**" is redefinable !

Groping by Feature(s)

- **Java:**

```
class Account {
  // -- attribute (field) -------
  int balance;

  // -- constructors ------------
  Account() { .. }
  Account(int b) { .. }
  Account(String s) { .. }

  // -- methods -----------------
  void deposit(int i) { .. }
  void withdraw(int i) { .. }
  void transfer(Account a) { .. }
}
```

- **Eiffel:**

Comment syntax:
'--' (rest of line)

```
class ACCOUNT
  feature -- Access
    balance:INTEGER // attribute

  create
    make
    make_balance
    make_name

  feature -- Initialization
    make do .. end
    make_balance do .. end
    make_name do .. end

  feature -- Basic operations
    deposit(i: INTEGER) do .. end
    withdraw(i: INTEGER) do .. end
    transfer(a: ACCOUNT) do .. end
end
```

The "**feature**" clause is used for ***grouping routines*** and for ***information hiding*** [later].

Getter vs Functional Attributes

- **Java:**

DEF

```
class Account {
  // -- attribute ---------------
  private int balance;

  // -- getter ------------------
  int getBalance() {
    // other code
    return balance;
  }

  // -- setter ------------------
  void setBalance(int balance) {
    // other code
    this.balance = balance;
  }
}
```

Convention:
getName ~ name
setName ~ name

- **Eiffel:**

DEF

```
class ACCOUNT
  feature -- attribute
    balance:INTEGER
end
```

DEF

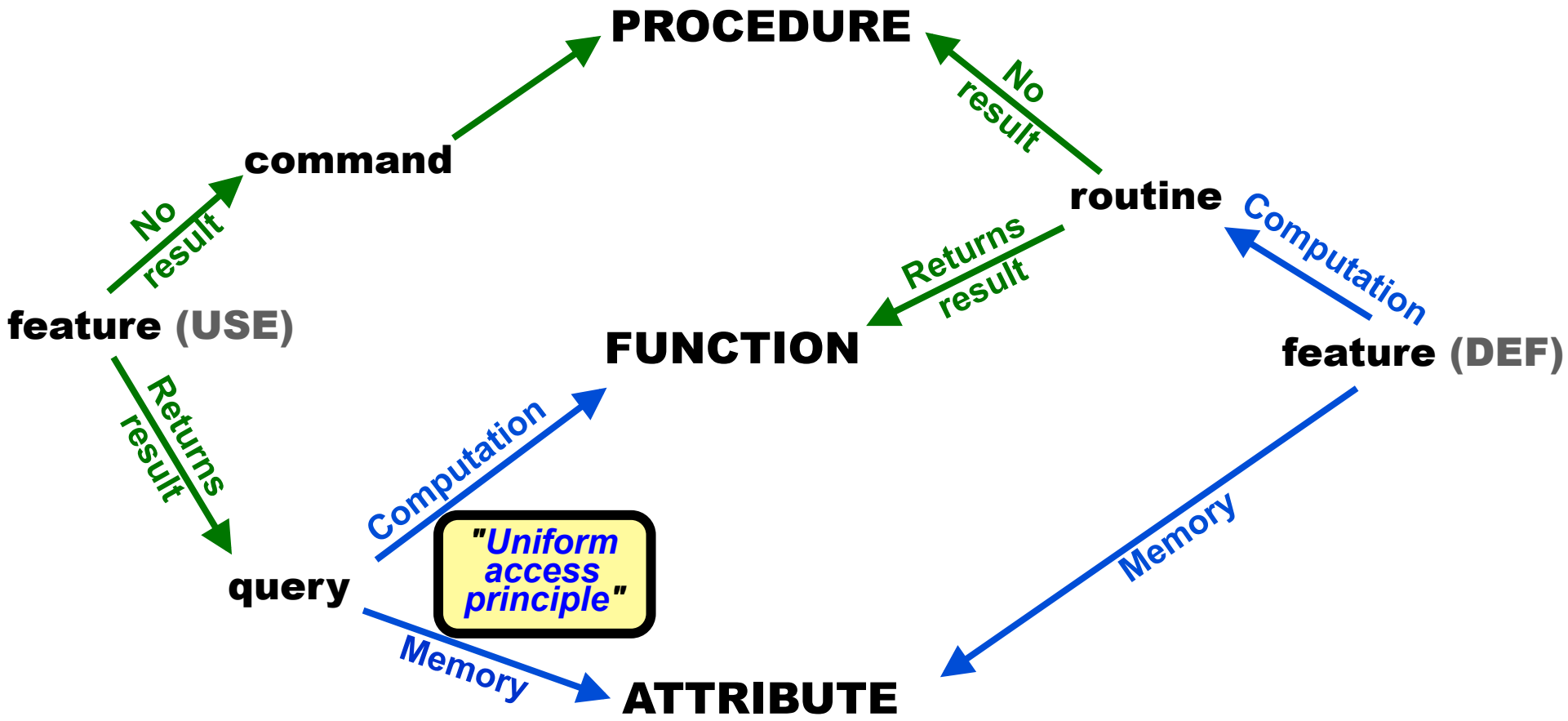
```
class ACCOUNT
  feature -- function
    balance:INTEGER do .. end
end
```

"**balance**" is now a ***function*** *returning an* **INTEGER** *(no longer an attribute)*

USE

```
  x := a.balance + 7;
```

Terminology



Expressions && Conditionals

- **Java:**

Strict conjunction

```
void foo(){
  if (x & (y | z)) {
    x = 42;
    ...
  }
}
```

Strict disjunction

Lazy conjunction

```
void foo(){
  if (x && (y || z)) {
    x = 42;
    ...
  }
}
```

Lazy disjunction

- **Eiffel:**

Strict conjunction

```
feature
  foo
    do
      if x and (c or d) then
        x := 42
        ...
      end
    end
```

Strict disjunction

Lazy conjunction

```
feature
  foo
    do
      if x and then (c or else d) then
        x := 42
        ...
      end
    end
```

Lazy disjunction

Loops

- **Java:**

```
void foo() {
  for (int i=0; i<10; i++) {
    print(i);
  }
}
```

- **Eiffel:**

```
feature
  foo
    local
      i:INTEGER
    do
      from i:=0 until i>=10 loop
        print(i);
        i := i+1;
      end
    end
```

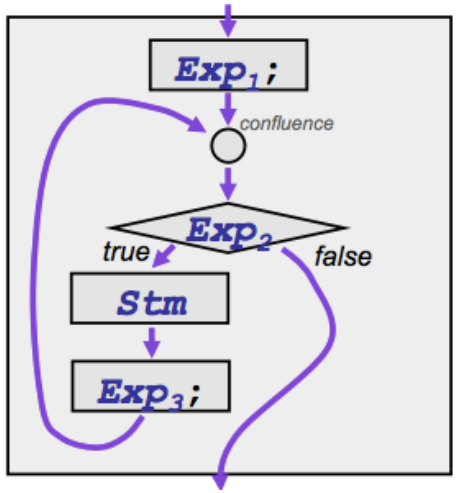
NB: negated condition "`from-until-loop`"

- **'for' ⇒ 'while':**

```
for (Exp1; Exp2; Exp3) {
  Stm
}
```

⇒

```
{
  Exp1 ;
  while ( Exp2 ) {
    {
      Stm
      Exp3 ;
    }
  }
}
```



Inspect

- **Java:**

```
class SwitchExample {
  String prettyPrint(int x) {
    switch(x) {
      case 1:
        return "one";
      case 2:
        return "two";
      case 3:
        return "three";
      default:
        return "another";
    }
  }
}
```

- **Eiffel:**

```
class INSPECT_EXAMPLE
  feature
    pretty_print(x: INTEGER):STRING
      do
        inspect x
        when 1 then Result := "one"
        when 2 then Result := "two"
        when 3 then Result := "three"
        else Result := "another"
        end
      end
    end
```

A routine produces a result by assigning to a ***special variable*** "`Result`".

Uncontrolled Control Structures

- **Java:**

```
void foo() {
  for (int i=0; i<10; i++) {
    print(x);
    ..
      continue;
    ..
      break;
    ..
      return;
    ..
  }
  print(y);
  ..
}
```

- **Eiffel:**

```
feature
  foo
    local
      i:INTEGER
    do
      from i:=0 until i>=10 loop
        print(x);
        ..
          -- structured control flow!
        ..
      end
      print(y);
      i := i+1;
    end
```

No "non-structural" control structures
(i.e., no "**continue**", no "**break**", no "**return**")!

Infix Notation

- **Java:**

Not supported!

- **Eiffel:**

binary infix notation!

DEF

```
class MY_CLASS

feature
  plus alias "+" (other:MY_CLASS): MY_CLASS
    do
      ...
    end
```

USE

```
-- a, b, c:MY_CLASS

c = a.plus(b)

c = a + b                    -- c = a.plus(b)
```

Infix Notation

- **Java:**

Not supported!

- **Eiffel:**

binary infix notation!

DEF

```
class MY_POINT

feature
  distance alias "|-|" (other:MY_POINT): INTEGER
    do
      ...
    end
```

USE

```
-- p, q :MY_POINT

if p.distance(q) > 10 then .. end

if (p |-| q) > 10 then .. end
```

NB: be aware of ***precedence*** and ***associativity*** rules

Infix Notation

- **Java:**

Not supported!

- **Eiffel:**

unary prefix notation!

DEF

```
class MY_CLASS

feature
  uminus alias "-" : MY_CLASS
    do
      ...
    end
```

USE

```
-- x, y :MY_CLASS

y = x.uminus()

y = -x                    -- y = x.uminus
```

Equivalence Relation '='

(S, '=') ***equivalence relation*** '=' $\subseteq$ S×S:

- **Reflexive:**
  - $\forall x \in S$: $x = x$
- **Symmetric:**
  - $\forall x,y \in S$: $x = y \Leftrightarrow y = x$
- **Transitive:**
  - $\forall x,y,z \in S$: $x = y \wedge y = z \Rightarrow x = z$

Partial Order Relation '≤'

(S, '≤') ***partial order relation*** '≤' ⊆ S×S:

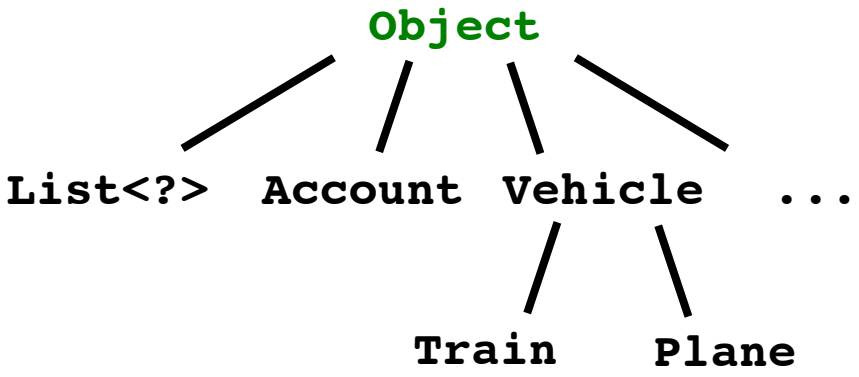
- **Reflexive:**
  - $\forall x \in S$: $x \leq x$
- **Anti-Symmetric:**
  - $\forall x,y \in S$: $x \leq y \wedge x \leq y \Rightarrow x = y$
- **Transitive:**
  - $\forall x,y,z \in S$: $x \leq y \wedge y \leq z \Rightarrow x \leq z$

Inheritance

Inheritance

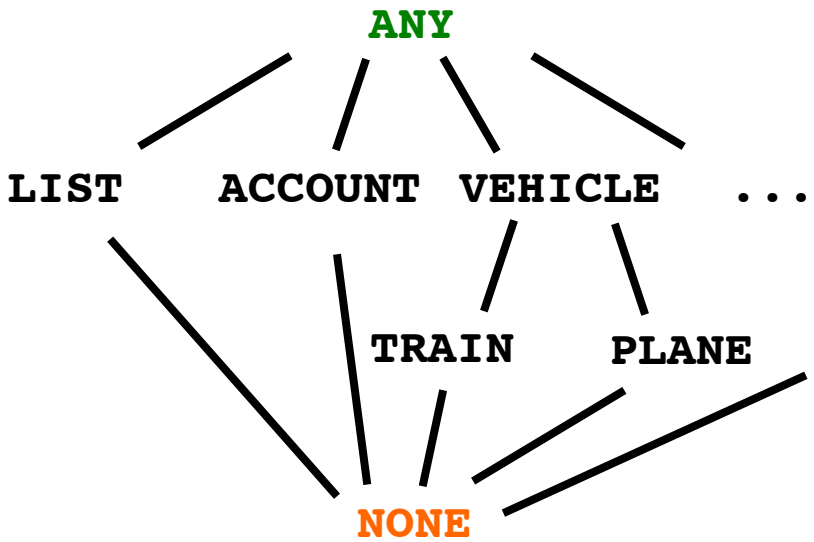
- **Java:**

```
class Account extends Object {

  ...

}
```



- **Eiffel:**

```
class ACCOUNT inherit ANY

  ...

end
```



Multiple Inheritance with "NONE" as the subclass of all classes!
[more about this later]

Redefinition of Routines

- **Java:**

```
class Account extends Object {

  String toString() { // override
    return "abc";
  }

}
```

Method "**out**" is ***implicitly*** overwritten

- **Eiffel:**

```
class ACCOUNT inherit ANY

  redefine out end

  feature
    out:STRING -- redefinition
      Result := "abc"
    end
end
```

Routine: "**out**" is ***explicitly*** redefined

Precursor (super)

- **Java:**

```
class Account extends Object {

  String toString() { // override
    return super();
  }

}
```

"`super()`" will cause invocation of the ***method*** with the same name (here, "`toString()`") in the ***super class*** (here, "`Object`")

- **Eiffel:**

```
class ACCOUNT inherit ANY

  redefine out end

  feature
    out:STRING -- redefinition
      Result := Precursor {ANY}
    end
end
```

"`Precursor()`" will cause invocation of the ***routine*** with the same name (here "`out()`") in the ***heir class*** (here, "`ANY`")

In case of ***multiple inheritance***, the heir has to be specified "`{ANY}`" (optional in case of a single heir)

Multiple Inheritance

- **Java:**

Java does *not* support multiple inheritance!

- **Eiffel:**

```
class A
  feature
    foo  do  print "A"  end
end
```

```
class B
  feature
    foo  do  print "B"  end
end
```

```
class C
  inherit
    A
    B  undefine foo end
  end
end
```

Name clashes can be avoided by ***undefining features*** (routines and attributes)

Multiple Inheritance

- **Java:**

Java does *not* support multiple inheritance!

- **Eiffel:**

```
class A
  feature
    foo  do  print "A"  end
end
```

```
class B
  feature
    foo  do  print "B"  end
end
```

```
class C
  inherit
    A
    B rename foo as foo_b end
  end
end
```

Name clashes can be avoided by
renaming features
(routines and attributes)

Frozen (final) Classes

- **Java:**

```
final class Account {

  ...

}
```

A ***final class*** can not be ***extended***

- **Eiffel:**

```
frozen class ACCOUNT

  ...

end
```

A ***frozen class*** can not be ***inherited***

Frozen Routines (final methods)

- **Java:**

```
class Account {

  final void deposit(int i) {
    ...
  }

}
```

A ***final method*** can not be ***overwritten***

- **Eiffel:**

```
class ACCOUNT

  feature
    frozen deposit(i:INTEGER)
      do
        ...
      end

end
```

A ***frozen routine*** can not be ***redefined***

Also, ***all of its args are frozen***

Frozen Routines (final methods)

- **Java:**

DEF

```
class A {
  void m() {
    System.out.println("test");
  }
}

class F {
  final void fin(A a) {
    a.m();
  }
}

class B extends A {
  void m() { // override
    System.out.println("he he!");
  }
}
```

USE

```
// A a; F f; ...
a = new B();
f.fin(a);
```

Output:
"he he!"

- **Eiffel:**

```
class ACCOUNT

  feature
    frozen deposit(i:INTEGER)
      do
        ...
      end

end
```

May *not* be *subclassed!*

A *frozen routine* can not be *redefined*

Also, *all of its args are frozen*

Deferred (abstract) Classes

- **Java:**

```
abstract class Vehicle {

  abstract void move(int i) {
    ...
  }

}
```

NB: A ***class*** with at least one ***abstract method*** has to be ***abstract*** !

- **Eiffel:**

```
deferred class VEHICLE

  feature
    deferred move(i:INTEGER)
    end

end
```

NB: A ***class*** with at least one ***deferred routine*** has to be ***deferred*** !

Parameter Mechanisms

Java:

DEF

```
class Main {
  void m(Account x, int y) {
    x.deposit(1);
    y = y+1;
  }
}
```

USE

```
    // Account a;  int b;
    a = new Account(2);
    b = 2;
    m(a, b);
    System.out.println(a);
    System.out.println(b);
```

Output:
3
2

Class types have ***"call by reference" semantics***; i.e., they are ***shared*** !

Primitive types have ***"call by value" semantics***; i.e., they are ***copied*** !

Eiffel:

DEF

```
class MAIN
  feature m(x:ACCOUNT; y:INTEGER)
      do  x.deposit(1)
          -- y := y+1
      end
end
```

Not allowed to change value of a parameter !

USE

```
    -- a:ACCOUNT ; b:INTEGER
    a.make_balance(2)
    b := 2
    m(a, b)
    a.print
    b.print
```

Output:
3
2

Class types have ***"call by reference" semantics***; i.e., they are ***shared*** !

Expanded types have ***"call by value" semantics***; i.e., they are ***copied*** !

Expanded Types

- **Java:**

- **Eiffel:**

DEF

```
expanded class ACCOUNT
  ...
end
```

USE

```
    -- a:ACCOUNT
    a.print
    x.f(a) -- a copied!
           -- not passed by ref!
    a.print
```

The two print statements will always print the exact same
(Routine '`f`' of object '`x`' will ***side-effect*** a ***copy*** of `a`)

Exception Handling

- **Java:**

```
class Printer {

  void print(int i) {
    try {
      ...
      throw new Exception();
      ...
    } catch(Exception e) {
      // handle exception
    }
  }
}
```

"**throw**" for raising exceptions

- **Eiffel:**

```
class PRINTER
  feature
    print_int(i:INTEGER)
      do
        ...
        (create {MY_EXCEPTION})
                              .raise
        ...
      rescue
        -- handle exception
      end

end
```

Control structure: "`try-catch`" for exception handling (via exceptional control flow)

Once Routines

- **Simulated in Java:**

```
static int fib(int n) {
  if (n==0) return 0;
  else if (n==1) return 1;
  else return fib(n-2) + fib(n-1);
}
```

```
static Integer cache = null;
static int foo() {
  if (cache==null) {
    cache = fib(30);
  }
  return cache;
}
```

Can be simulated using ***dynamic programming*** (aka, ***"memoization"***)

Relies on (**static**) ***global variables*** (singleton pattern) **!**

```
print(foo())    // calculated
print(foo())    // cached :-)
```

```
832040 [slow]
832040 [fast]
```

- **Eiffel:**

```
feature
  fib(n:INTEGER):INTEGER
    do
      if n=0 then Result := 0
      elseif n=1 then Result := 1
      else Result := fib(n-2) +
                     fib(n-1)
      end
    end
```

```
feature
  foo:INTEGER
    once
      Result := fib(30)
    end
```

"**once**" routines cache their results (built in)

```
print(foo)    -- calculated
print(foo)    -- cached :-)
```

```
832040 [slow]
832040 [fast]
```

Eiffel: Sharing Obj's via "Once"

(NB: No Java on this slide!)

```
class X
  create
    make
  feature
    make
      do ... end
```

Class to be shared

```
class SHARED_X
  feature
    the_one_and_only_x: X
      once
        Result := X.make -- instantiate X
      end
```

Superclass with ***once routine*** that creates an object (i.e., same object cached and returned on every access)

```
class SOME_CLASS inherit SHARED_X
 feature
    some_routine
      do
        ... the_one_and_only_x ...
      end
```

attribute inherited

```
class SOME_OTHER_CLASS inherit SHARED_X
  feature
    some_other_routine
      do
        ... the_one_and_only_x ...
      end
```

attribute inherited

Style Conventions

- **Java:**

Class names:
global name space
two classes cannot have same name
(even in diff clusters)

Often, classes are prefixed with a library prefix; e.g. `EV_WINDOW` (EiffelVision library)

- **Eiffel:**

All upper case.
Full words (usually without abbreviation)

```
class ACCOUNT
  feature -- Access
    balance:INTEGER // attribute

  feature -- Initialization
    make do .. end
    make_balance do .. end
    make_name do .. end

  feature -- Basic operations
    deposit(i: INTEGER) do .. end
    withdraw(i: INTEGER) do .. end
    transfer(a: ACCOUNT) do .. end

end
```

Feature names:
use full (underscored) words that clearly identify the role

Use ***tabulation*** and ***not spaces*** (for indentation)

NB: one namespace for:
- locals `l_name`
- arguments `a_name`
- attributes `name`

Information Hiding

Java: (Visibility modifiers)

- `private`
- `protected`
- *package* (default)
- `public`

visibility modifier	class	pack-age	sub-class	world
`public`	✔	✔	✔	✔
`protected`	✔	✔	✔	✘
package	✔	✔	✘	✘
`private`	✔	✘	✘	✘

Qualified: '`a.f`'
('`f`' always visible in `A` unqualified: '`f`')

Eiffel:

```
class A
  feature {NONE}
    f do .. end
end
```

***a.f** not visible anywhere (not even **A**)* !

```
class A
  feature {ANY}
    f do .. end
end
```

***a.f** visible everywhere (default)* !

```
class A
  feature {B,C}
    f do .. end
end
```

a.f** visible in **B** and **C** (and descendants), but not **A !

```
class A
  feature {A,B,C}
    f do .. end
end
```

***a.f** visible in **A**, **B**, & **C** (and descendants)* !

GENERICS

(Type Variables and
Type Parameterization)

Without Generics

- **Java (Older):**

DEF

```
class MyQueue {

  Object getItem() {
    ...
  }

  void add(Object o) {
    ...
  }
}
```

USE

```
MyQueue q;

q.add(new Integer(42))

// ** runtime error ***
String s = (String) q.get(0);
```

- **Eiffel:**

DEF

```
class MY_QUEUE
  feature
    item:ANY
      -- first item in queue
      do ... end

    extend(a_element:ANY)
      -- add new element to queue
      do ... end
end
```

USE

```
-- q:MyQueue, s:STRING

q.extend(42)

// ** type error (assignment) ***
s := q.item
```

With Generics

- **Java:**

generic type variable: 'E'

DEF

```
class MyQueue<E> {

  E getItem() {
    ...
  }

  void add(E e) {
    ...
  }
}
```

USE

```
MyQueue q<String>;


q = new MyQueue<String>();
q.add("abc");
String s = q.get(0);
```

- **Eiffel:**

generic type variable: 'G'

DEF

```
class MY_QUEUE[G]
  feature
    item:G
      -- first item in queue
      do ... end

    extend(a_element:G)
      -- add new element to queue
      do ... end
end
```

USE

```
    local
      qs: MY_QUEUE[STRING]
      s : STRING
    do
      create qs
      qs.extend("abc")
      s := qs.item
    end
```

Constrained Generic

- **Java:**

'E' class must be a ***subclass*** of 'Comparable' !

```
class MyQueue
      <E extends Comparable<E>> {
  E getItem() {
    ...
  }

  void add(E e) {
    ...
    if (e.compareTo(item) < 0)
      ...
    ...
  }
}
```

- **Eiffel:**

'G' class must ***inherit*** from 'Comparable' !

```
class MY_LIST[G -> COMPARABLE]
  feature
    item:G
      -- first item in list
      do ... end

    extend(a_element:G)
      -- add new element to list
      do
        ...
        if a_element < item then
          ...
        end
        ...
      end
end
```

EIFFEL BASE

Libraries

Libraries

- "**`EiffelStudio`**" comes with a relatively rich set of APIs that cover most of core needs of modern programming:

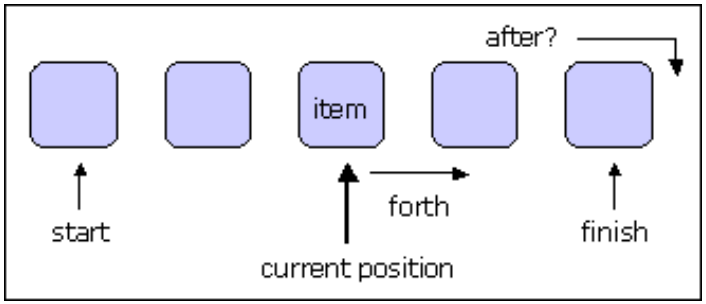
  `argument_parser, base, base_extension, cURL, diff, docking, editor, encoding, event, gobo, gobo_extension, graph, i18n, lex, memory_analyzer, net, obsolete, parse, preferences, process, store, testing, text, thread, time, uuid, vision2, vision2_extension, web, web_browser, wel.`

EiffelBase

Eiffel Base:

- `LIST [G]`
- `LINKED_LIST [G]`
- `ARRAY [G]`
- `HASH_TABLE [G,H -> HASHABLE]`
- `RANDOM` (generating random #s)
- `STRING_32` (for unicode chars)
- `SINGLE_MATH` (various math ops)
- `DOUBLE_MATH` (...with higher precision)
- ...

Seq. Structures (List, Array, ...)



- **`start`**: moves to the first position, if any (if "**`is_empty`**" is true then the command will have no effect)
- **`item`**: returns the item at the *current position* (the precondition is: '**`not off`**')
- **`forth`**: advance by one position (the precondition is '**`not after`**')
- **`finish`**: moves to the last position (the precondition is: '**`not is_empty`**')
- **`off`**: is false iff there is no item at the current position
- **`after`**: whether you have moved past the last position
- **`is_empty`**: whether or not the structure is empty

Traversal

- **Java:**

```
void print(List<Element> list) {
  for (Element e : list) {
    e.print();
  }
}
```

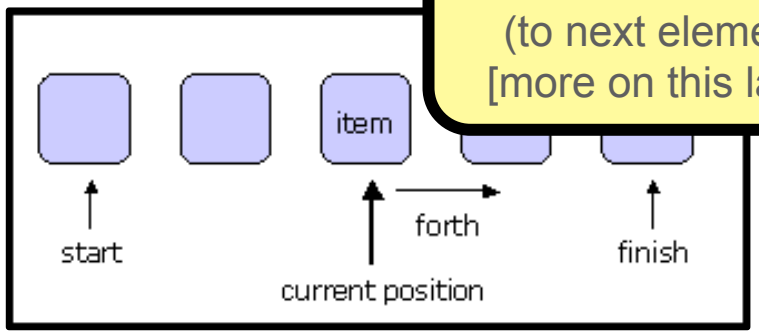
...or using an Iterator

- **Eiffel:**

```
feature
  print
    do
      from list.start
      until list.after
      loop
        list.item.print
        list.forth
      end
    end
```

Traversal with: "`from-until-loop`"

NB: routine call advances list (to next element) [more on this later]



Traversal (alternative)

- **Java:**

```
void print(List<Element> list) {
  for (Element e : list) {
    e.print();
  }
}
```

...or using an Iterator

- **Eiffel:**

```
feature
  print
    do
      across list as e loop
        e.item.print
      end
    end
```

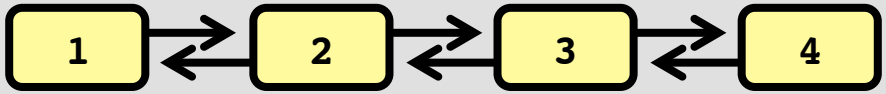
Traversal with: "across-as-loop"

NB: You have to enable ***"provisional syntax"*** to use "across-as-loop"

LinkedList vs ArrayList

LinkedList<E>

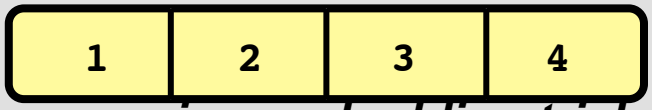
- *constant time insertion!*



- `get(int x)` O(n)
- `add(E e)` O(1)
- `add(int x, E e)` O(n)
- `remove(int x)` O(n)

ArrayList<E>

- *fast random access!*



using re-doubling trick

- `get(int x)` O(1)
- `add(E e)` *amortized:* O(1)
  worst-case: O(n)
- `add(int x, E e)` O(n)
- `remove(int x)` O(n)

ARRAY[G]

NB: Arrays start at one (1)

- **`make_empty`**
  creates an empty array
- **`item(i:INTEGER):G`**
  access to the **`i`**th element of the array
  Alternative (infix) syntax: '`a @ 7`' ⇒ '`a.item(7)`'
- **`count:INTEGER`**
  size of the array
- **`put(v:G; i:INTEGER)`**
  adds element '**`v`**' at index (valid index) '**`i`**'
- **`force(v:G; i:INTEGER)`**
  adds element '**`v`**' at index '**`i`**' (resizing as needed)
- **`has(v:G):BOOLEAN`**
  whether or not '**`v`**' appears in array

HASH_TABLE[G]

- **`make (n: INTEGER_32)`**
  Allocate a hash table for at least '**`n`**' items (the table will be resized automatically if more than '**`n`**' items are inserted)

- **`has (key: H): BOOLEAN`**
  Whether or not there is an item in the table with key '**`key`**'

- **`has_item (v: G): BOOLEAN`**
  Whether or not the hash table includes value '**`v`**'

- **`capacity: INTEGER_32`**
  Returns the number of items that may be stored

- **`count: INTEGER_32`**
  Returns the number of items in table

HASH_TABLE[G] (cont'd)

- **`item alias "[ ]" (key: H): G`**
  Item associated with '**`key`**', if present (otherwise default value of type '**`G`**')

- **`search (key: H)`**
  Search for item of key '**`key`**'. If found, set '**`found`**' to true, and set '**`found_item`**' to item associated with '**`key`**'

- **`put (new: G; key: H)`**
  Insert '**`new`**' with '**`key`**' (if no other item associated with the same key)

- **`force (new: G; key: H)`**
  Update the hash table so that '**`new`**' will be the item associated with '**`key`**'. If there was an item for that key, set '**`found`**' and set '**`found_item`**' to that item. If there was none, set '**`not_found`**' and set '**`found_item`**' to the default value.

Learning Resources

- "Eiffel Tutorial" (ET)
- "Invitation to Eiffel" (I2E)
- "Touch of Class: Learning to Program Well with Objects and Contracts"
- "Object-oriented Software Construction"
- ECMA Standard 367
- "Eiffel: The Language"
- "Eiffel Language Syntax"
- ...and that thing known as "The Internet" :-)

Thx

Any questions ?

Eiffel: "System"

- Run root method (must be a creation procedure) of root class (then: create instance of root class and run its root method)
- **Class dependency: "B ~> A"** iff:
  - **B *client-of* A** (i.e., B ***uses objects of type*** A)
  - **B *heir-of* A** (i.e., B ***extends*** A)

  - Reflexive transitive closure of root class
  - In particular, "dead classes" disregarded

- NB: no global variables!

Project Organization

- Each Eiffel class is in one cluster
- A class, `C`, is stored in a single file called '`C.e`'
- Clusters are organized by directory
- Unique cluster, class, and feature names
- Build descriptions are found in '`.ecf`' files:
  - Describe the locations of clusters
  - Point to dependent libraries
  - Summarize compiler options, etc.
  - (Quite similar to Ant)

Miscellaneous

- '`/=`': inequaliy
- '**`Current`**' (like Java's 'this')
- '**`Void`**' (initialization value for reference types)

- Default initialization values:
  - **`INTEGER`**: '**`0`**'
  - **`BOOLEAN`**: '**`False`**'
  - **`CHARACTER`**: '**`Null`**'
  - Class types: '**`Void`**'
  - *composite*: above rules apply *recursively to all fields*