Sistemas Distribuídos

About Java and Programming Methodologies

António Rui Borges

Summary

- *Java*
  - *Historical note*
  - *Main features*
- *General description techniques of a problem solution*
- *Programming methodologies*
  - *Procedural or imperative programming*
  - *Modular programming*
  - *Object oriented programming*
  - *Concurrent programming*
  - *Distributed programming*
- *Suggested reading*

Historical note - 1

Java, originally called *Oak*, was created in the mid of the 90s by a *Green* project working group of Sun MicroSystems, headed by James Gosling

- its syntactic structure is based on the programming languages C and C++
- the initial target area for the language was building control and communication environments for embedded systems of consumer electronic appliances
  - its popularity, however, arises in a different context, that of *internet*: first, with *applets* (small programs written in Java that can be executed inside a *browser*) and, more recently, with *web services*
- in a somewhat broader perspective, Java tends to be the language of choice to write distributed applications, since it provides an integrated and uniform environment for interprocess communication over a computer network (Java in action).

Historical note - 2

Java initial objectives, as a programming language, included

- *to be totally independent of the underlying hardware platform*, this has lead to be simultaneously thought of as a *language* and as an *execution environment*: thus, programs can be transferred in run time and executed in any node of the processing mesh
- *to be inherently robust*, minimizing programming errors: it is, therefore, organized as a strongly semantic language where some C++ features, of multiple inheritance and of operator overloading, for instance, were eliminated for being judged potential sources of error; a mechanism of implicit management of dynamic memory (*garbage collection*) was also introduced and any attempt of definition of data structures outside the scope of the `class` constructor was forbidden; in this sense, some people claim that ***Java is C++ done right***
- *to promote security* in environments that are continuously exchanging information and sharing code: *pointers* were thus eliminated, trying to prevent that non-authorized programs were able to access data structues resident in memory.

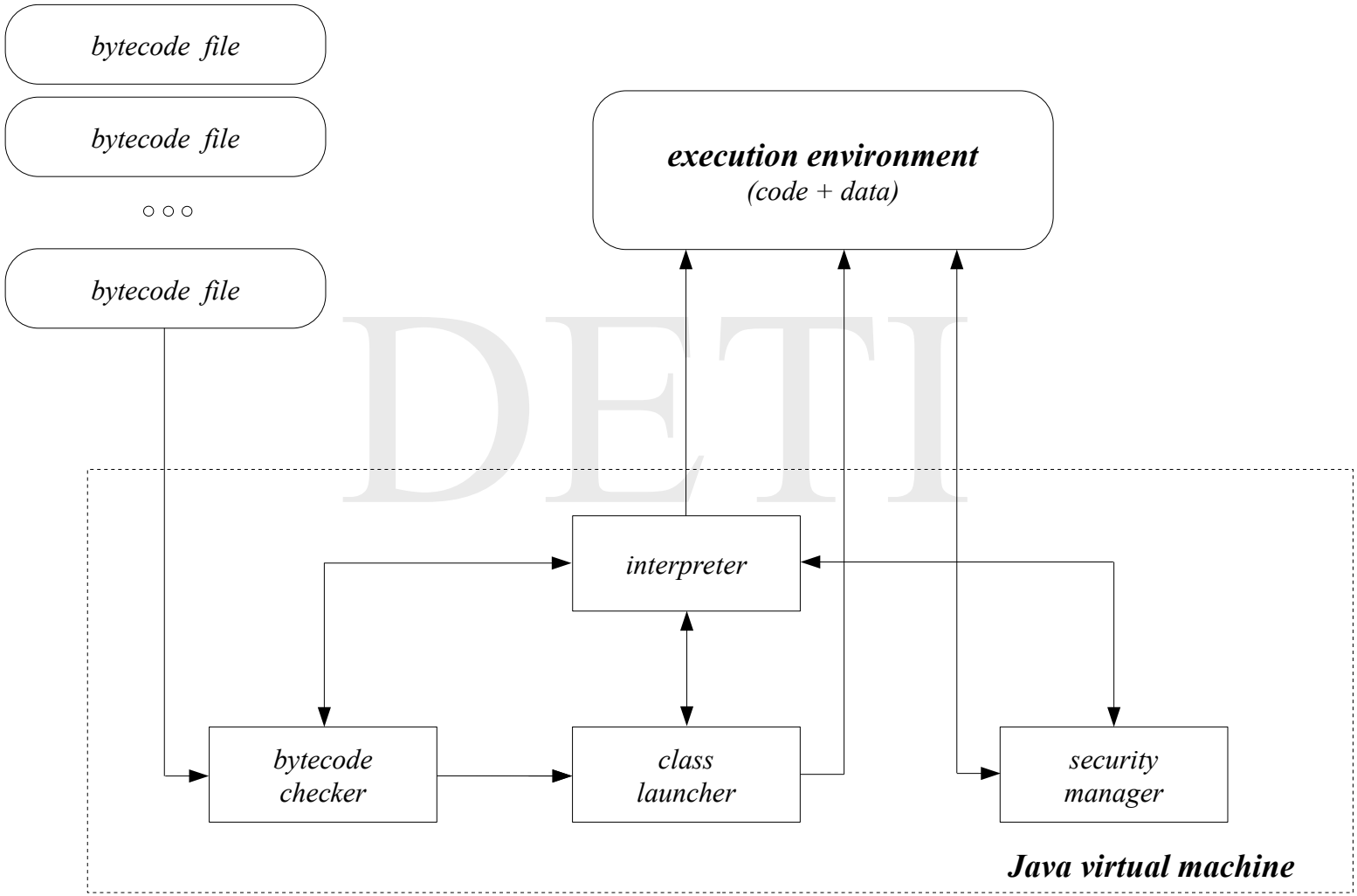
Main features - 1

- the language follows the principles of the *object-oriented paradigm*
  - there is a minimum number of semantic constructs, enabling a compact and inherently safe description
  - the inheritance mechanism is strictly linear
  - the `interface` constructor, as a special case of the `class` constructor, was introduced to enable a precise and clear separation between specification and implementation and to serve as a connecting point between variables of distinct data types
- it provides concurrency constructs
  - there is an explicit support for building *multithreading* environments
  - *objects* are implicitly transformed into *monitors* [of the Lampson / Redell type] through synchronization of its public methods
- it supports distributed programming by providing libraries which implement the two major communication paradigms
  - message passing: *sockets*
  - shared variables: *remote method of invocation* (RMI)

Main features - 2

- the Java execution environment, called *Java virtual machine* (JVM), constitutes a *middleware* layer which makes the applications to be totally independent of the hardware platform and the operating system where they run, implementing a three-layered security model that protects the system against non-trustable code
  - the *bytecode checker* parses the bytecode presented to execution and ensures that the basic rules of the Java grammar are obeyed
  - the *class launcher* delivers the required data types to the Java interpreter
  - the *security manager* deals with the problems which put potentially at stake the application related system security, controlling the access conditions of the running program to the file system, to the network, to external processes and to the window system

Main features - 3



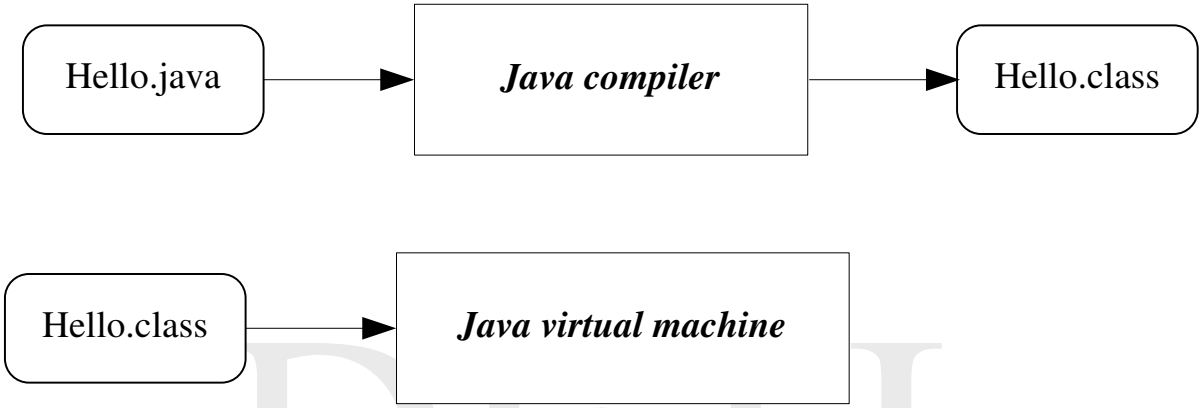
Main features - 4

- it supports internationalization
  - ASCII code is replaced by Unicode in the internal representation of characters to allow the consideration of the alphabets and ideographic symbols of different world languages
  - separation of locale information from the executable code, together with the storage of text elements outside the source code and its dynamic access, ensures that there will be a conformance of applications to the language and other specific cultural traits of the final user
- it includes tools that ease the production of program documentation
  - by taking into account the *documentation comments* inserted in the source files to describe the reference data types, its internal data structures and its access methods, it becomes trivial to generate documentation in *html* format through the application of the `javadoc` tool.

Trivial example - 1

```
/**
 *    General description:
 *     in this case, it may be the program prints the sentence
 *      <em>Hello <b>person name</b>, how are you?</em> in the monitor screen.
 *
 *    @author António Rui Borges
 *    @version 1.0 - 13/2/2017
 */

public class Hello

{
  /**
   *    Main program,
   *      it is implemented by the main method of the data type.
   *    <p>Any application must contain a static method named main in its
   *       start data type.</p>
   *
   *    @param args nome da pessoa a saúdar
   */

   public static void main (String [] args)

   {
      System.out.println ("Hello " + args[0] + ", how are you?");
   }
}
```

Trivial example - 2



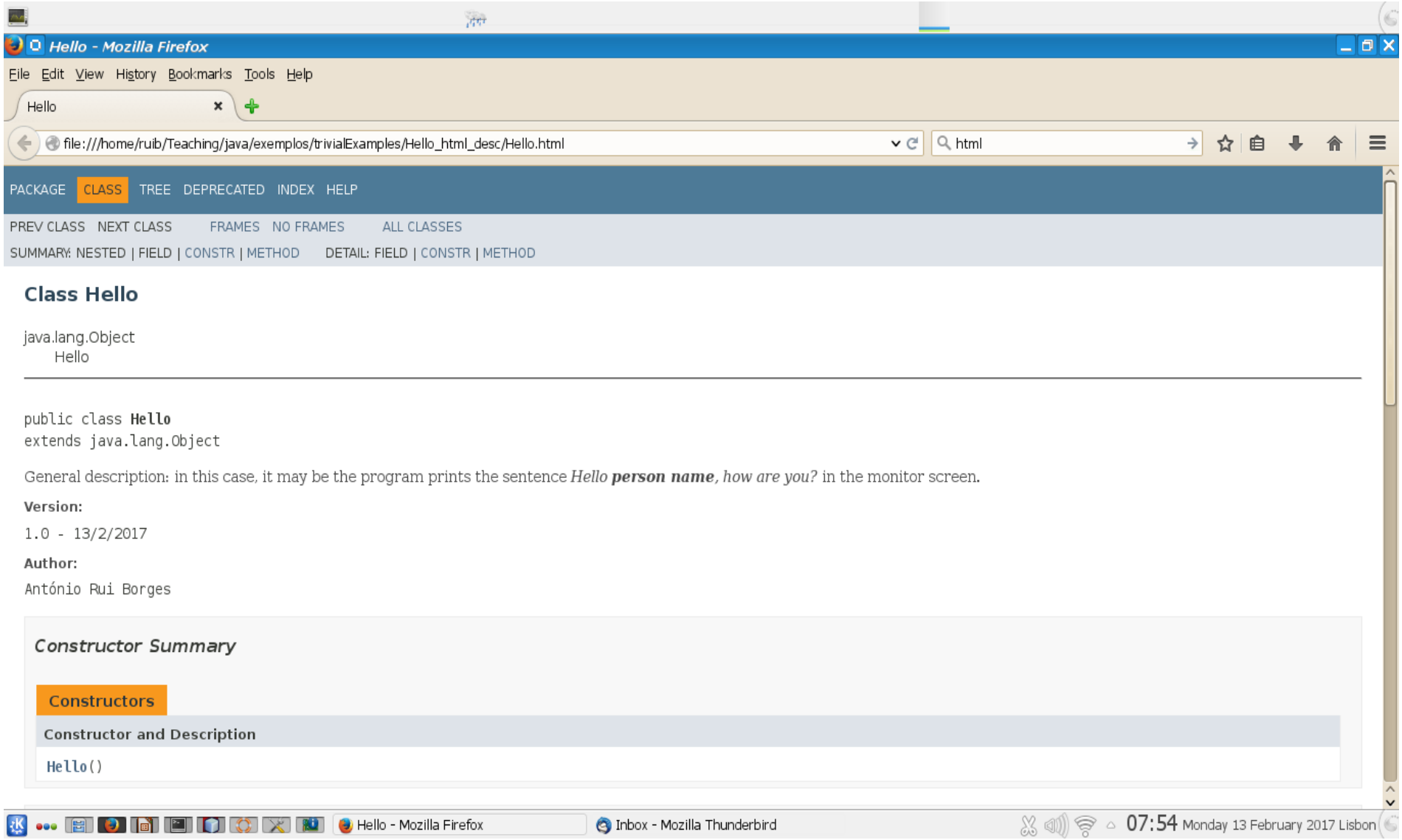

```
[ruib@ruib-laptop1 trivialExamples]$ javac Hello.java
[ruib@ruib-laptop1 trivialExamples]$ java Hello Pedro

Hello Pedro, how are you?

[ruib@ruib-laptop1 trivialExamples]$ ll
total 24
-rw-r--r-- 1 ruib ruib 599 Feb 13 07:42 Hello.class
-rw-r--r-- 1 ruib ruib 654 Feb 13 08:06 Hello.java
-rwxr--r-- 1 ruib ruib 131 Feb 13 06:42 Hello.javadoc
-rw-r--r-- 1 ruib ruib 424 Feb 13 08:07 HelloWorldApp.class
-rw-r--r-- 1 ruib ruib 149 Nov 26  2015 HelloWorldApp.html
-rw-r--r-- 1 ruib ruib 374 Feb 13 08:03 HelloWorldApp.java
[ruib@ruib-laptop1 trivialExamples]$
```

Trivial example - 3

Documentation production

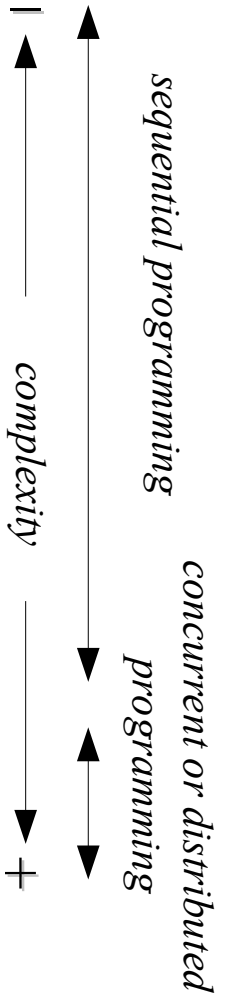
```
[ruib@ruib-laptop1 trivialExamples]$ cat Hello.javadoc
javadoc -d Hello_html_desc -author -version -breakiterator -charset "UTF-8" Hello.java
ln -s Hello_html_desc/Hello.html Hello.html

[ruib@ruib-laptop1 trivialExamples]$ ./Hello.javadoc
Loading source file Hello.java...
Constructing Javadoc information...
Creating destination directory: "Hello_html_desc/"
Standard Doclet version 1.8.0_31
Building tree for all the packages and classes...
Generating Hello_html_desc/Hello.html...
Generating Hello_html_desc/package-frame.html...
Generating Hello_html_desc/package-summary.html...
Generating Hello_html_desc/package-tree.html...
Generating Hello_html_desc/constant-values.html...
Building index for all the packages and classes...
Generating Hello_html_desc/overview-tree.html...
Generating Hello_html_desc/index-all.html...
Generating Hello_html_desc/deprecated-list.html...
Building index for all classes...
Generating Hello_html_desc/allclasses-frame.html...
Generating Hello_html_desc/allclasses-noframe.html...
Generating Hello_html_desc/index.html...
Generating Hello_html_desc/help-doc.html...
[ruib@ruib-laptop1 trivialExamples]$
```

Trivial example - 4



General description techniques of a problem solution

- *hierarchical decomposition*
  - recourse is made to a metalanguage of description, preferably similar to the programming language that will be used
  - information is encapsulated through
    - the construction of suitable data types to the characteristics of the problem
    - the definition of new operations in the context of the language
  - strict data dependencies are established
- *decomposition in interactive autonomous structures*
  - precise specification of an interaction mechanism
  - clear separation between *interface* and *implementation*
    - data abstraction and protection
    - virtualization of access operations
- *decomposition in interactive autonomous entities*
  - precise specification of a communication model
  - definition and implementation of synchronization mechanisms.



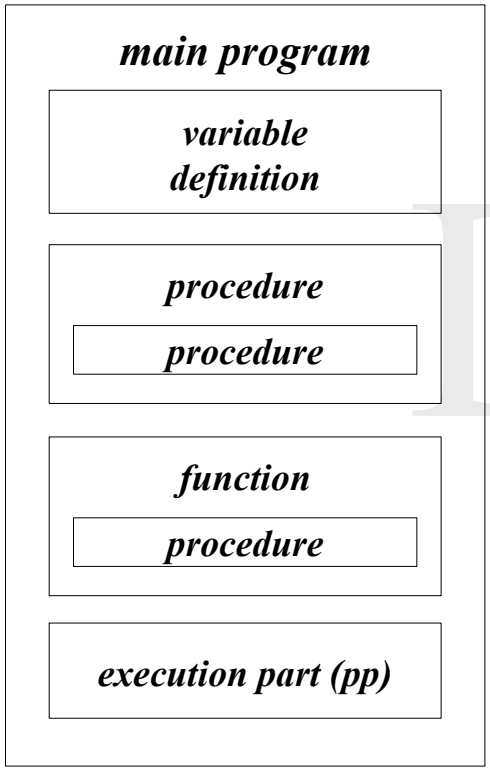
Programming methodologies - 1

Procedural or imperative programming

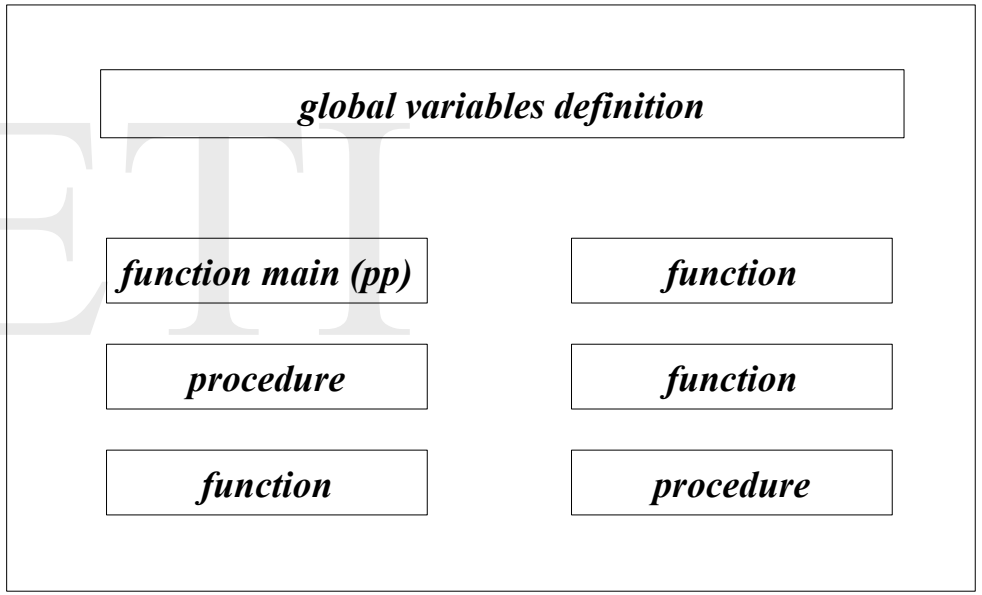
- emphasis is placed in a more or less accurate implementation of the hierarchical decomposition of the solution
- complexity is controlled by the intensive use of
  - *functions* and *procedures*, as a means to describe operations at different levels of abstraction
  - *data type constructors*, to organize the associated information in the most suitable way to the characteristics of the problem
- variable space is *concentrated*
  - all relevant data to the problem solution is defined at the *main program* level, or is global; variables local to the remaining functions and procedures only require temporary storage (they only exist when the functions or the procedures are invoked)
  - access of the different operations to data is conceived in strict obedience to the principle "*what they need to know*", using a communication model based on parameter transfer

Programming methodologies - 2

Procedural or imperative programming



hierarchical organization of a source file – Pascal like



horizontal organization of a source file – C language like (sea of functions structure)

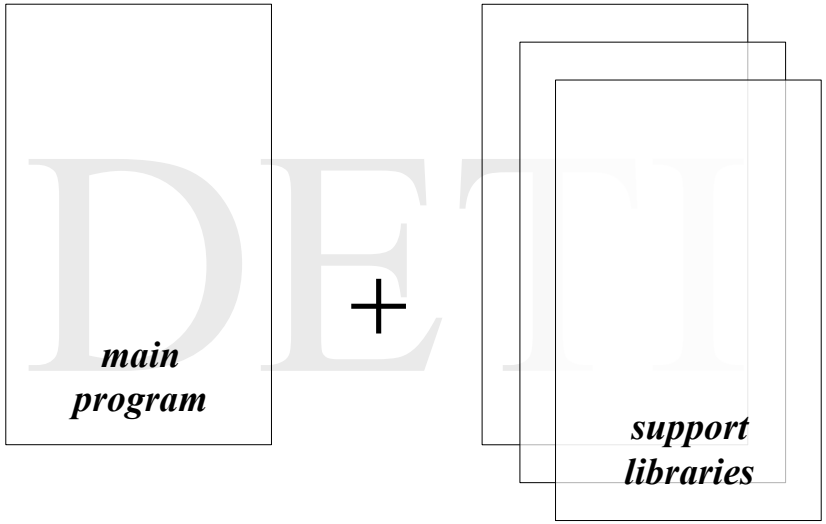
Programming methodologies - 3

Procedural or imperative programming

- an issue that immediatly comes to mind is *code reuse*
- different operations, implemented by functions and procedures in the context of a specific application, are often useful in other applications
- its automatic inclusion in the code of new applications can be done by splitting the source file into multiple files, each targeted for separate compilation and put together later on in a single executable file at the linking stage
- thus, the code of a particular application is scattered by
  - a source file that contains the main program and, eventually, other private code
  - multiple source files that contain well-defined functionalities, implemented by functions and procedures which were independently written and which will be now put to work in the context of the present application, the so called *support libraries*

Programming methodologies - 4

Procedural or imperative programming



- it becomes necessary in this type of organization to insert in the piecewise source files, whenever its code refers to external operations, the name of the source file where they are declared (*interface file*) to ensure consistency at compile time.

Programming methodologies - 5

Modular programming

- as the description complexity of the solution increases, the centralized management of the space of variables becomes progressively more difficult and emphasis moves from operation conception to the development of a finer data organization
- the space of variables becomes in fact *distributed* and the concept of *module* arises as a more convenient means to deal with its management
  - a *module* is, by definition, an autonomous structure, described in a separate source file, that encapsulates a well-defined functionality owning in general a proprietary space of variables and a set of operations to manipulate it
  - the space of variables is usually *internal*, which means that access to it can only be done by the specified operations, the so-called *access primitives*
  - it should be noted that, when the proprietary space of variables has no external meaning, a *module* becomes what was formerly called a *support library*

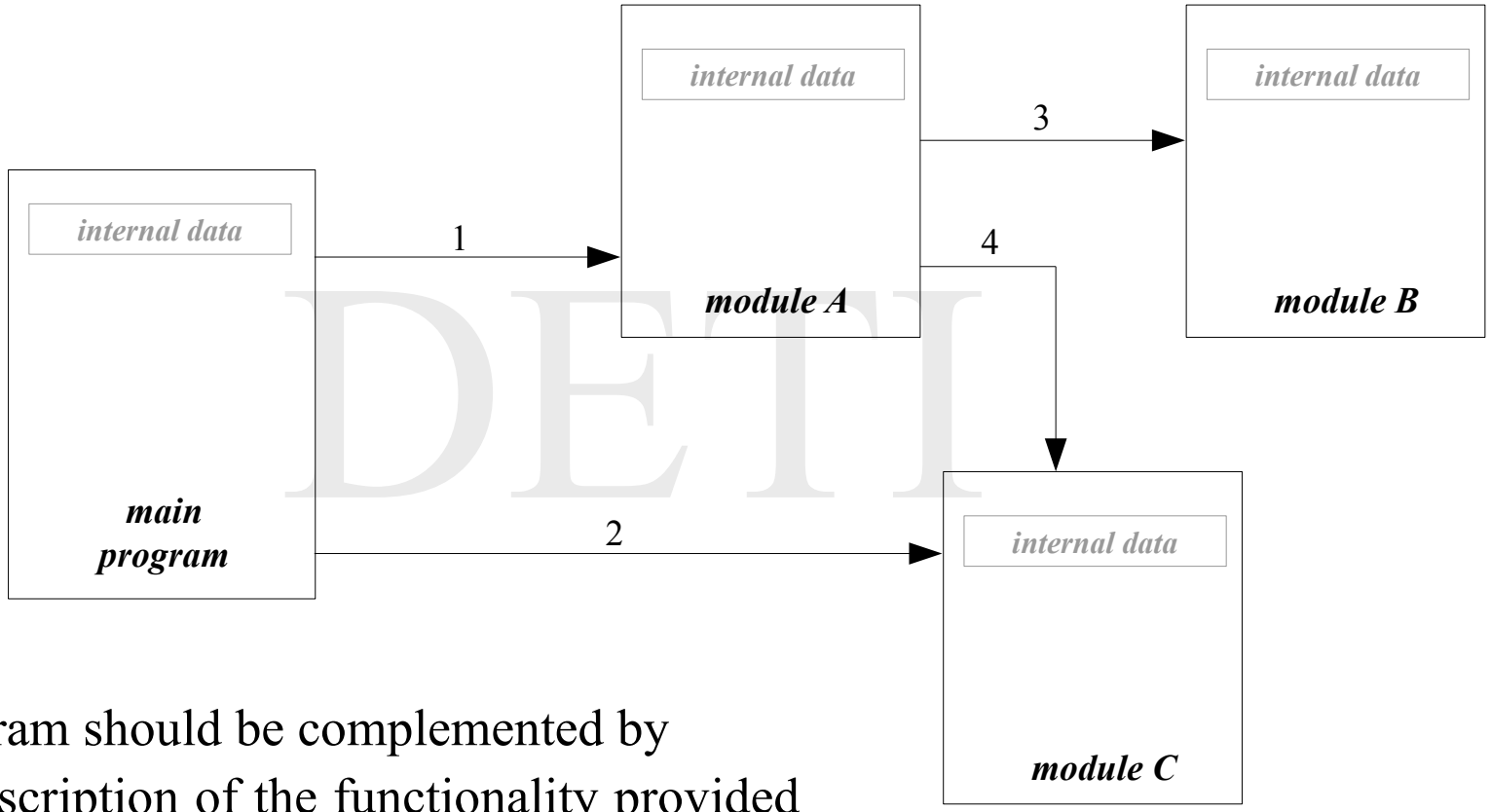
Programming methodologies - 6

Modular programming

- the application of a functional approach to solution decomposition generates the specification of a set of autonomous interacting structures and one is able to deal with much more complex problems through the creation of a clear task division and the promotion of cooperative work
  - the description is now done by specifying precisely which are the interacting structures at play and by establishing for each of them what is its role in the interaction and how access to it is to be made
  - the obvious separation between access specification, the *interface*, and implementation allows its use in the design of other modules, even before the original module has been fully concluded and validated

Programming methodologies - 7

Modular programming



- the diagram should be complemented by
  - a description of the functionality provided by each of the modules
  - a listing of the operations involved in the interaction

Programming methodologies - 8

Modular programming

- the consistent application of a *divide and conquer* strategy, which underlies any description methodology and, particularly, when a functional decomposition of the solution is carried out, enables the development of *robust* code, embodied by
  - *design for test*: as a module is an autonomous structure, it functionality can and should be validated according to the previously established specification before its inclusion in the application it is aimed to be a part of
  - *programming by contract*: the programmer, to whom it was assigned the task of its implementation, should devise means that ensure that only the described functionality is made available at *runtime*
    - all access primitives must return *status* information about the operation: *success* or *an error has occurred*, with a specific indication of the error in the latter case
    - the consistency of the internal data structure must be ensured prior to the execution of the first operation (*pre-invariant*)
    - whenever an operation is called, the value of each variable in the input parameter list must be checked to assert if it is within the allowed range, before operation execution
    - the consistency of the internal data structure must be ensured after operation execution (*post-invariant*).

Programming methodologies - 9

Object-oriented programming

- a modular implementation of a functional decomposition may become rather inflexible in terms of *code reuse* whenever the new application requires
  - the multiple instantiation of a module
  - the introduction of small changes to it
- the programmer inevitably tries to deal with the situation by adjusting the pre-existing modules to the present requirements, leading to a progressive pulverization of his/her repertoire of implemented functionalities and the consequent loss of efficiency on managing support software for the development of future applications
- a possible solution to this conundrum is an approach of increased abstraction

Programming methodologies - 10

Object-oriented programming

- the first problem may be solved by an *extension* to the concept of *data type*
- in fact, if it were possible to associate with each module instantiation a different variable, the ambiguity is removed and multiple instantiations become trivial
- in this sense, the notion of *data type* as a set of rules which allow the storage in memory of values with certain properties, is now extended to contemplate not only the storage of an aggregate of values, but also the identification of the operations that can be executed on them
- although improperly, these data types are sometimes called *abstract data types*; *reference data types*, or *user-defined data types*, are better names which are also commonly used

Programming methodologies - 11

Object-oriented programming

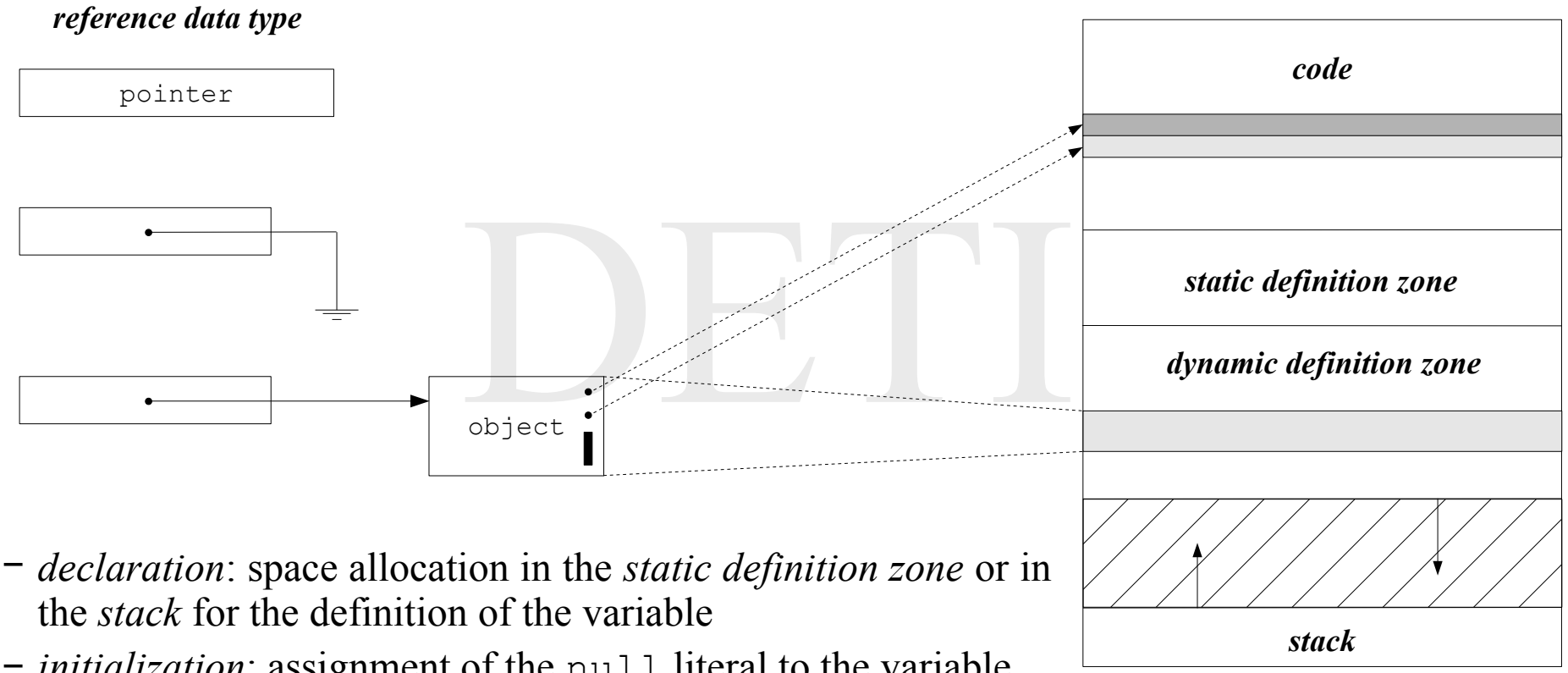
- programming languages that support them, have a constructor, whose traditional name is `class`, which basically allows the association of a module definition to an identifier
- from this point on, it becomes possible to declare variables of this type
- an important detail is that the pure declaration of variables of this type does not allocate memory space for its storage, it only allocates memory space for a *pointer*
- space allocation for the value itself is only made by the *type instantiation* in *runtime*; then, space is allocated in the dynamic definition zone of the process address space and the consequent initialization of the internal data structure takes place; furthermore, whenever the variable is no longer necessary, the storage space in dynamic memory should be freed
- the instantiated values are called *objects*, thus the paradigm name

Programming methodologies - 12

Object-oriented programming

- in *object-oriented* terminology, one calls *fields* to the variables of the internal data structure of the reference data type, and *methods* to the access primitives; an *object* is then determined by its *state* (set of values presently stored in the different fields) and by its *behavior* (set of access methods provided)
- programming languages that support the paradigm, usually allow *data abstraction* in the specification of the internal data structure, giving rise to generic data types with a wider application range
  - instead of creating a data type which implements a stack for character storage, for instance, one may create a data type which implements a stack for the storage of any kind of values
- furthermore, one may still ensure that a single type of values is stored there in *run-time* by using *parametrization*
  - taking the former example, one may create a data type that implements a stack for the storage of values of an unspecified data type `T` and postpone to the declaration and the subsequent instantiation of variables of this type the information about what type `T` really means

Object-oriented programming



- *declaration*: space allocation in the *static definition zone* or in the *stack* for the definition of the variable
- *initialization*: assignment of the `null` literal to the variable
- *instantiation*: space allocation in the *dynamic definition zone* for *object* creation (in *runtime*)

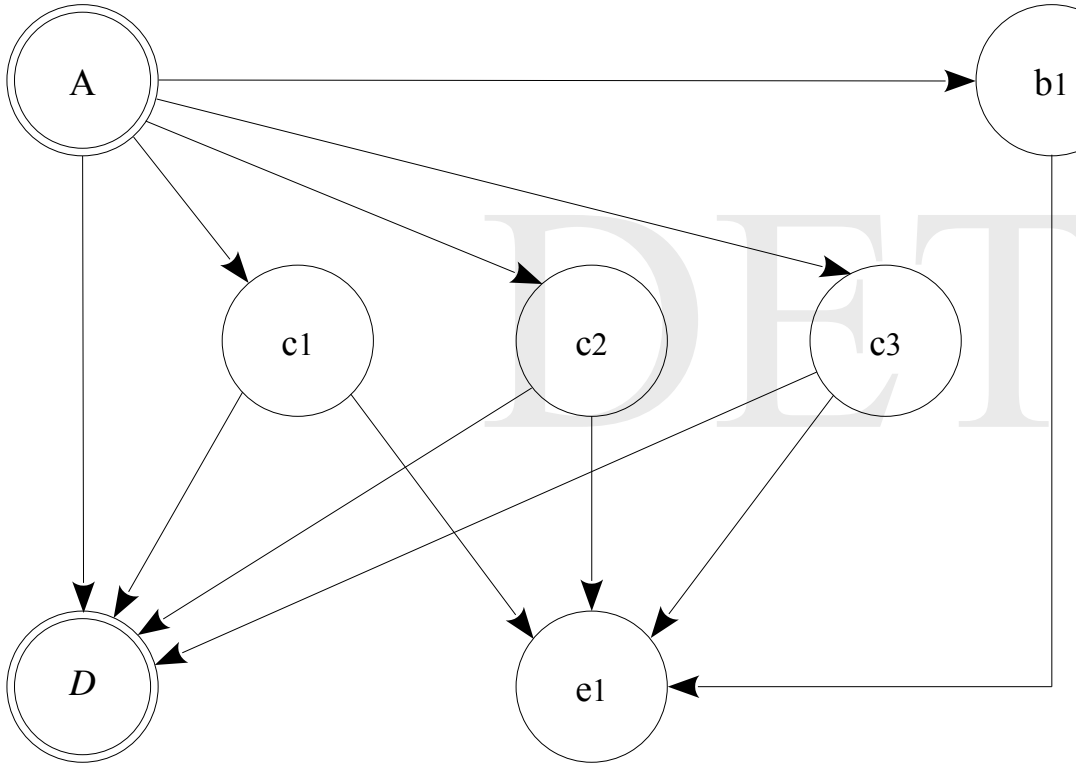
Programming methodologies - 14

Object-oriented programming

- a program organized according to the object-oriented paradigm consists of multiple source files, each defining a particular data type
- as it happens in modular programming, the interaction description is portrayed by a diagram that includes the *non-instantiated* and the *instantiated* data types, the former named by the type identifiers and the latter by the identifiers of the associated variables; the way they interact is depicted by an oriented graph which expresses the geometry of access; the diagram should be complemented with a description of the functionality each data type provides and by the listing of the operations
- as a rule, *non-instantiated* data types represent the *main program* and groups of generic operations organized in *libraries*; all the remaining data types are in principle *instantiated*

Programming methodologies - 15

Object-oriented programming



`A` – data type that contains the main program
`b1` – instantiation of data type `B`
`c1` – instantiation of data type `C`
`c2` – instantiation of data type `C`
`c3` – instantiation of data type `C`
`D` – data type that contains generic methods
`e1` – instantiation of data type `E`
→ – instantiation, access to a field or method invocation

sea of non-instantiated and instantiated data types (classes and objects)

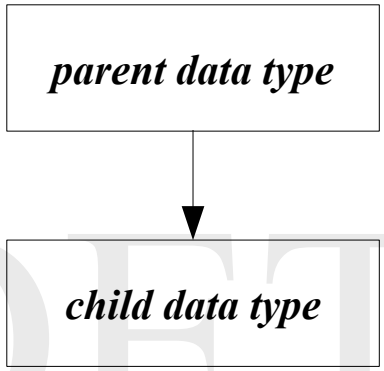
Programming methodologies - 16

Object-oriented programming

- the second problem may be solved by the application of the concept of *reuse* to the construction of data types
- indeed, if it were possible to define new data types from pre-existing data types, one can achieve their differentiation to make them suitable to new situations with a minimum of effort
- this mechanism, called *enheritance*, lets one extend in a hierarchical fashion to the new data type, *child data type* or *subtype*, the properties of the data type in which the definition is based, *parent data type* or *supertype*
- specifically, this means that
  - the *protected* fields of the internal data structure
  - the *public* methods

  of the *base* data type (parent data type) are directly accessible and, in the latter case, modifiable in the *derived* data type (child data type)

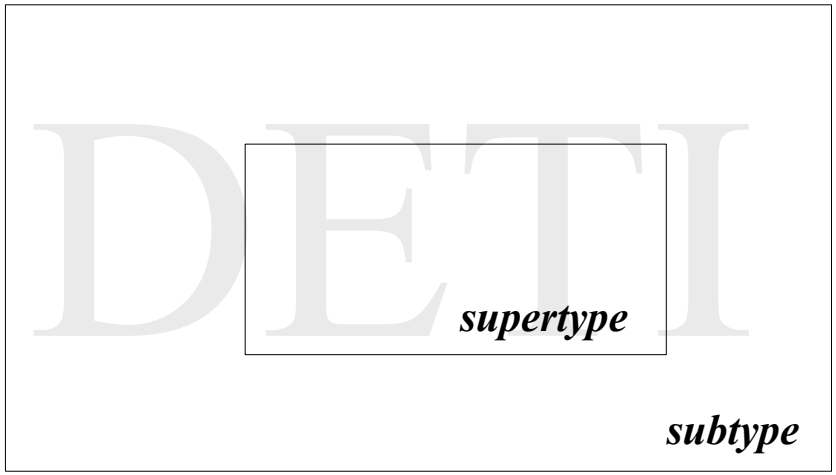
Object-oriented programming



- the creation mechanism of new data types, called *derivation* mechanism, allows one
  - to introduce new fields in the internal data structure
  - to introduce new public methods
  - to *override* methods: redefinition of [previously implemented] public methods of the base data type
  - to *implement virtual* methods: definition of public methods whose interface is in the base data type

Programming methodologies - 18

Object-oriented programming



- compatibility wise, the subtype is *compatible* with the supertype it is derived from, but the converse is not always true

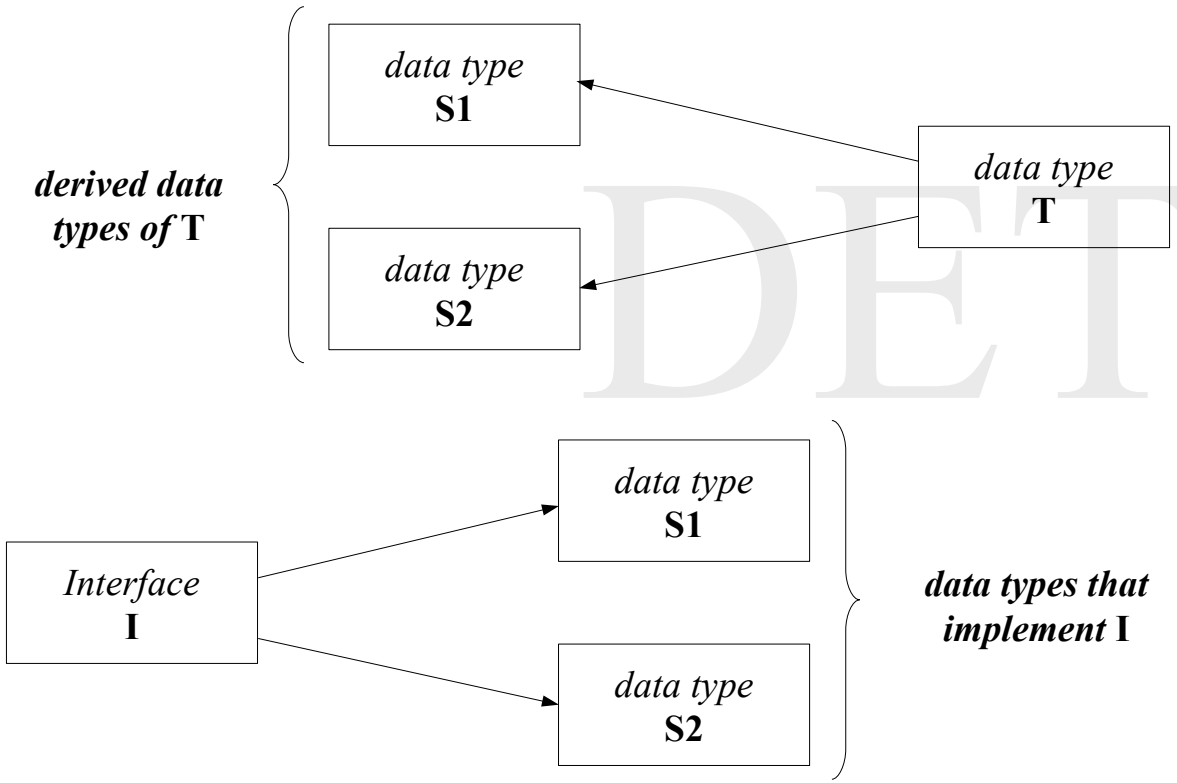
Programming methodologies - 19

Object-oriented programming

- the compatibility relation can be used to establish a very powerful principle of the object-oriented paradigm, *polymorphism*, which consists of the possibility of using the same variable to reference objects of different data types, derived from the same base type
- let `T` be a reference data type, `S1` and `S2` two distinct data types derived from the data type `T` and `t`, `s1` and `s2` variables of each one of these types where objects of the respective data types were assigned to the last two; consider now that one assigns any of these variables to the variable `t` (always a valid operation because `s1` and `s2` are compatible with `t`, although they are not mutually compatible) and one calls on it a method defined in `T`
- the association of the method suitable to the object being referenced is automatically selected and carried out
  - at the compilation stage, *static association*, when the above-referred method was not modified in the definition of any of the derived data types at play
  - in *runtime*, *dynamic association*, otherwise.

Programming methodologies - 20

Object-oriented programming




```
T t;
I i;
S1 s1 = new S1 (...);
S2 s2 = new S2 (...);

if (condition)
   t = s1;
   else t = s2;
   ...
t.m (...);
```

invocation of method m on variable t

```
   ...

if (condition)
   i = s1;
   else i = s2;
   ...
i.m (...);
```

invocation of method m on variable t

Programming methodologies - 21

Concurrent programming

- the functional decomposition of the solutions of complex problems leads quickly to the need of conceiving a group of activities that take place in a more or less autonomous manner and cooperate to achieve a common goal
- keeping a single thread of execution requires the integration in the user code of a *scheduler* which switches among the different activities
- this approach, being very complex and demanding, is also totally useless since present day operating systems allow multiprogramming, providing facilities for interprocess communication and synchronization
- furthermore, with the popularization of *multicore* processors, operating systems implement *thread* management at the *kernel* level, which means a speed up in the running of concurrent applications

Programming methodologies - 22

Concurrent programming

- two different approaches to solution design are in use
  - *event-driven approach*: the processes that implement the activities are normally blocked, waiting for an event which triggers their execution; one deals here with more or less independent processes, usually only one of them being active at a time; process communication takes place by writing and reading shared variables kept centrally; *visualization space managers*, which interact with the user through the mouse and the keyboard, are perhaps the most common example of this approach
  - *peer-to-peer approach*: the processes that implement the activities cooperate among themselves in a more or less specific manner; each is thought of as running a life cycle consisting of independent and interacting operations, the latter produce or collect information, block the process until certain conditions are attained, or wake up other processes when certain conditions are met
- whatever the case is, an always present issue is that, opposite to what happens in sequential programming, there is no guarantee of operations reproductivity, therefore, *debugging* is much more sensitive and much less effective here

Programming methodologies - 23

Concurrent programming

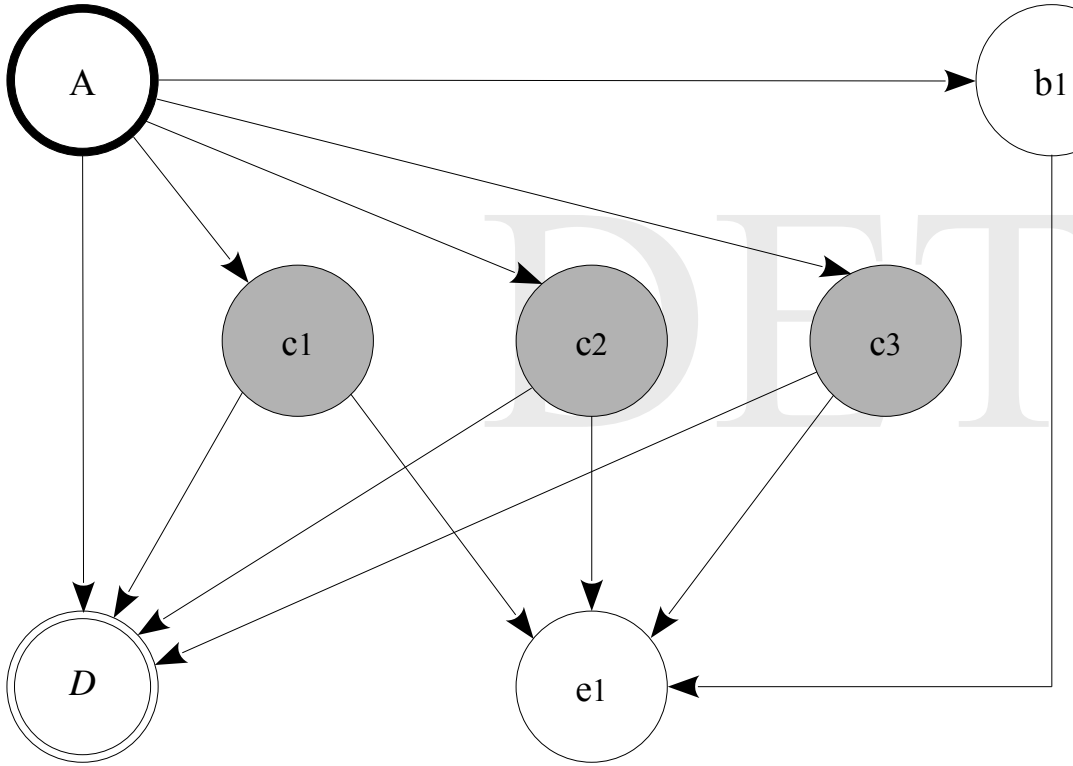
- there are two basic models for information sharing and communication
  - *shared variables*: it is the common model in *multithreaded* environments where the intervening processes write and read values stored in a centrally defined data structure; to prevent *racing conditions* that may lead to inconsistency of information, the access code to the shared region (*critical region*) has to run in mutual exclusion; there is also the need to have means for process synchronization
  - *message passing*: it is a model of universal application since it does not require the sharing of the processes addressing space; communication is carried out by message exchange between pairs of processes (*unicast*), between a process and all the others (*broadcast*), or between a process and all the others belonging to the same group (*multicast*); one assumes here that an infrastructure of communication channels, interconnecting all the processes, is available and that it is managed by an entity not belonging to the application, ensuring a mutual exclusive access to the channels and providing synchronization mechanisms

Programming methodologies - 24

Concurrent programming

- a program organized according to this methodology conforms to the principles of modular programming, or object-oriented programming, to describe the interaction and, therefore, also consists of multiple source files
- what is new here is that, by having multiple threads of execution, one needs to clearly distinguish between *active* and *passive* entities
- *active* entities represent the different intervening processes, while *passive* entities result from making explicit the different types of functionality present.

Concurrent programming



`A` – data type that contains the main program (active entity)
`b1` – instantiation of data type `B`
`c1` – instantiation of data type `C` (active entity)
`c2` – instantiation of data type `C` (active entity)
`c3` – instantiation of data type `C` (active entity)
`D` – data type that contains generic methods
`e1` – instantiation of data type `E`
→ – instantiation, access to a field or method invocation

sea of non-instantiated and instantiated data types and active and passive entities

Programming methodologies - 26

Distributed programming

- there are two main reasons that lead to moving from *concurrent programming* to *distributed programming*
  - *parallelization*: to take advantage of multiple processors and other *hardware* components of a parallel computer system to get a faster and more efficient execution of an application
  - *making a service available*: to supply a well defined functionality to an enlarged group of applications in a way that is consistent, reliable and safe
- the change in methodology entails, therefore, that one somehow finds a way to map over distinct computer systems the different processes and functionality centers a concurrent solution was previously divided

Programming methodologies - 27

Distributed programming

- the mapping can not be made in an automatic way, however; there are several issues in the concept of *parallel processing platform* which have to be carefully assessed for the migration to be possible
- some of them are
  - eventual heterogeneities among the nodes of the processing platform
  - there is no global clock to enable the chronological ordering of events
  - some of nodes of the processing platform and parts of the communication infra-structure may fail at any given time.

Suggested reading

- *On-line* support documentation for Java program developing environment by Oracle (Java Platform Standard Edition 8)