Object-Oriented Programming

Bernard Boigelot

E-mail : bernard.boigelot@uliege.be
WWW : https://people.montefiore.uliege.be/boigelot/
https://people.montefiore.uliege.be/boigelot/courses/oop/

References:

- *An Introduction to Object-Oriented Programming*, 3rd ed., Timothy Budd, Addison-Wesley, 2002.
- *The Java Programming Language*, 4th ed., Ken Arnold, James Gosling, and David Holmes, Addison-Wesley, 2005.

Chapter 1

The Object-Oriented Approach

Programming Paradigms

For fundamental reasons, all programming languages share the same expressive power:
For every pair (A, B) of languages, any problem that can be solved by a program expressed in language A can also be tackled by a program written in language B.

There exist however

- many approaches to solving problems algorithmically,
- various programming mechanisms, and
- several programming styles.

Each of these choices can affect the desired properties of programs: Correctness, efficiency, modularity, conciseness, readability, ergonomics, . . .

Example: Imperative Structured Programming (C language)

Main principles:

- The executable instructions are arranged into sequences that form blocks.
- A block plays the same syntactic role as a single instruction.
- Control structures are limited to loops as well as binary and n-ary conditional decisions.
- Some instruction blocks can be turned into functions.
- Sets of functions can be grouped into modules.
- The scope of a variable can be either defined as global, or restricted to a function or a module.

Advantages:

- The restricted control structures improve the readability and maintainability of code.
- The programs can be made modular:
  - Variables that are local to a function or a module remain well decoupled from other ones.
  - The interface and implementation of modules can be cleanly separated.

Drawback:

- The modularity of programs is not optimal.

Illustration: Word Indexing in C

```c
/*  index.c  */

#include "index.h"

static struct
{
  ...
} index;

void index_init(void)
{
  ...
}

void index_insert(char *word, int val)
{
  ...
}

int index_lookup(char *word)
{
  ...
}
```

```c
/*  index.h  */

void index_init(void);
void index_insert(char *, int);
int  index_lookup(char *);
```

Advantages of this solution:

- The interface and implementation are well decoupled from each other.

  It is therefore possible to use the index module without knowing precisely its implementation details.

- The variable `index` is protected against naming conflicts and spurious access from other modules.

Drawback:

- It is not possible to work with more than one index in a given program!

Another Solution

```c
/*  index.c  */

#include "index.h"

index *index_new(void)
{
  ...
}

void index_insert(index *h, char *word, int val)
{
  ...
}

int index_lookup(index *h, char *word)
{
  ...
}

void index_free(index *h)
{
  ...
}
```

```c
/*  index.h  */

typedef struct
{
  ...
} index;

index *index_new(void);
void   index_insert(index *, char *, int);
int    index_lookup(index *, char *);
void   index_free(index *);
```

Advantage:

- One can now handle more than one index.

Drawback:

- This code is not modular enough: The internal structure of an index is defined as a part of its interface!

  As a consequence, a program that uses an index is able to manipulate its internal structure, which may lead to

  - problematic dependence constraints between modules, and
  - data corruption issues.

- It is difficult to guarantee that a pointer passed as an argument always corresponds to a valid index.

The Object-Oriented Approach: Goals

The *object-oriented* paradigm is an approach to solving algorithmic problems and to programming. It is aimed at producing programs that have good modularity properties.

Goals: Programmers should be able to

- develop part of a program without knowing precisely the internal details of the other parts,
- apply local modifications to a module without influencing the rest of the program,
- reuse fragments of code across different projects or program components.

The Object-Oriented Approach: Principles

- Objects are data structures that are present in memory during program execution.
- An object combines data and executable code. This code corresponds to operations that can be performed by the object.
- An object has an interface and an implementation. The environment of the object can only access its interface.
- Data managed by an object is kept in variables that are internal to this object.
- Operations that can be performed by an object are described by a set of methods.

- The interface of an object characterizes each method by a header, composed of
  - a method name,
  - a list of zero or more parameters, and
  - the type of an optional return value.
- The implementation of an object contains
  - its variables, and
  - the body of its methods, composed of executable instructions.
- The execution of a program is seen as a sequence of interactions between objects. These interactions take the form of messages. A message sent from an object A to an object B represents a request from A to B for performing an operation. The reception of a message by an object triggers the invocation of a method.

Note: The instructions executed upon invoking a method are able to

- manipulate object data,
- send messages, and
- create new objects.

Classes and Objects

In order to be able to create an object, its desired features have to be precisely specified, in particular

- the number, type and name of its variables, and
- the interface and body of its methods.

The set of all these features forms a *class*. A class can be instantiated in order to construct an object.

An object is an instance of a class.

An object-oriented program mainly takes the form of a set of class definitions.

Note: Objects instantiated from the same class have

- distinct variables, but with identical types, and
- identical methods.

Illustration: Word Indexing in Java

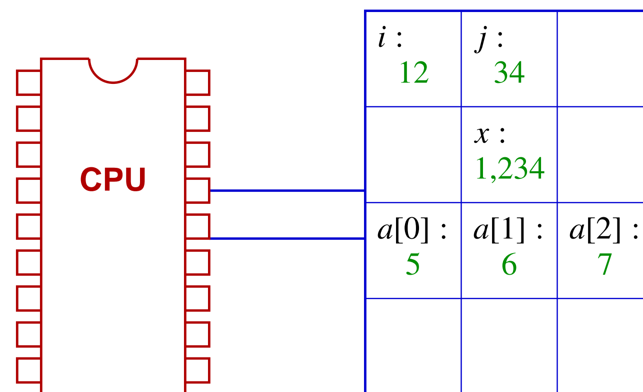
```
/*  Index.java  */

public class Index                              //  Class name
{
   private int nbWords;                         //  Variable
   ...
   ...

   public void insert(String word, int val)     //  Method header
   {
      ...                                       //  Method body
   }

   public int lookup(String word)
   {
      ...
   }
   ...
}
```

Advantages

- One can easily manipulate several indices in the same program, by instantiating the class `Index` as many times as needed.
- Data stored inside each index
  - is kept separate from the data of other objects, and
  - remains unreachable from the environment.
- By decoupling the implementation from the interface, it becomes possible to modify the implementation details of the index data structure without affecting other parts of the program.

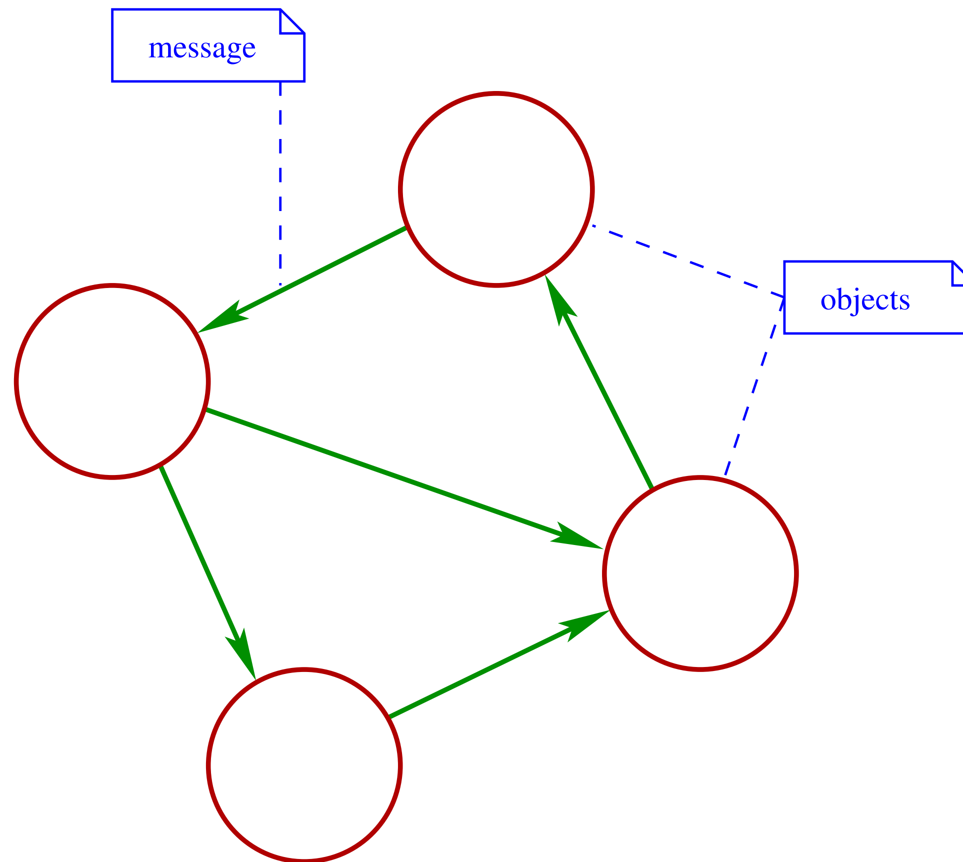
The Object-Oriented View of Programming

Traditionally, the execution of an imperative program is seen in the following way:



A processor executes a sequence of instructions that read and write values in a common memory.

The object-oriented approach is based on a different view:



Program execution is now distributed among several objects, that mutually exchange requests for performing operations.

Delegating Responsibilities

When an object receives a message, it accepts the responsibility of performing the corresponding operation.

Part of this operation may consist in sending messages to other objects. In such a case, the object delegates part of its responsibility to those objects.

Example (word processor): An object representing a line of text receives a message that requires this text to become justified at the margins.

1. The line object sends messages to all words composing the line, in order to ask them to compute their dimension in screen pixels.
2. The line object computes the optimal position of each word.
3. The line object sends to each word a message requesting to update its position.

Of course, objects cannot indefinitely delegate their responsibilities. In addition to sending messages, objects are also able to perform primitive (or native) operations, that are directly executable.

Developing an object-oriented program thus essentially amounts to

1. specifying classes corresponding to the elements of the problem to be solved,
2. assigning responsibilities to those classes, and organizing the delegation of these responsibilities,
3. defining variables and methods representing the data and operations of objects.

Chapter 2

Classes and Methods

Encapsulation

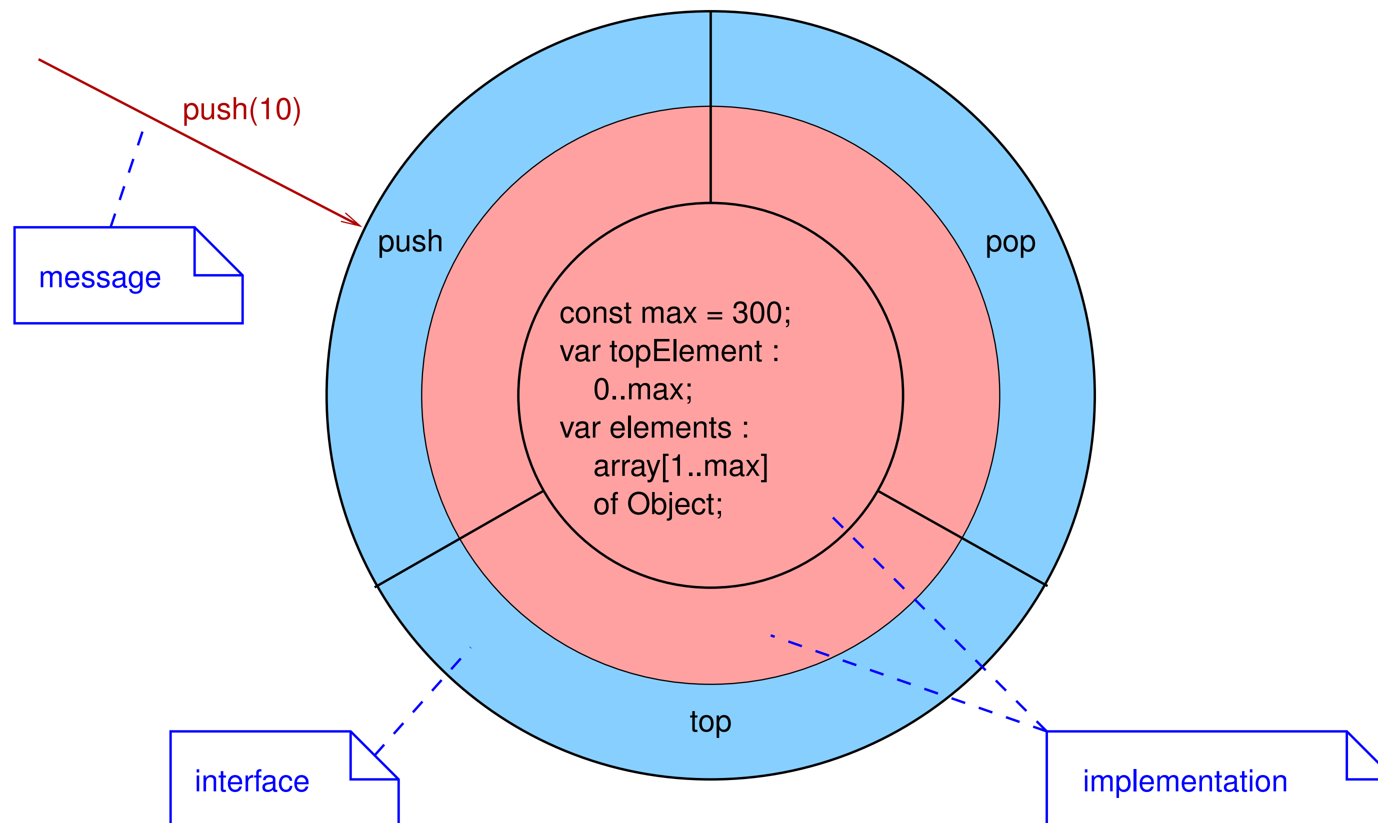
A class can be seen as an abstract data type, combining

- an interface specifying the set of operations that an instance of the class is able to perform, and
- an implementation that describes
  - the internal variables that store the state of an instance, and
  - the body of the methods that express how to perform operations.

Encapsulation principle: *Only the interface of an object can be accessed from outside its class.*

Example

Definition of a data structure representing a stack:



Information Hiding

- A class definition must specify the least possible amount of information needed for interacting with its instances, and nothing else.

- The implementation of a method must rely on the simplest possible specification of the task that it is expected to perform, and and on nothing else.

Visibility

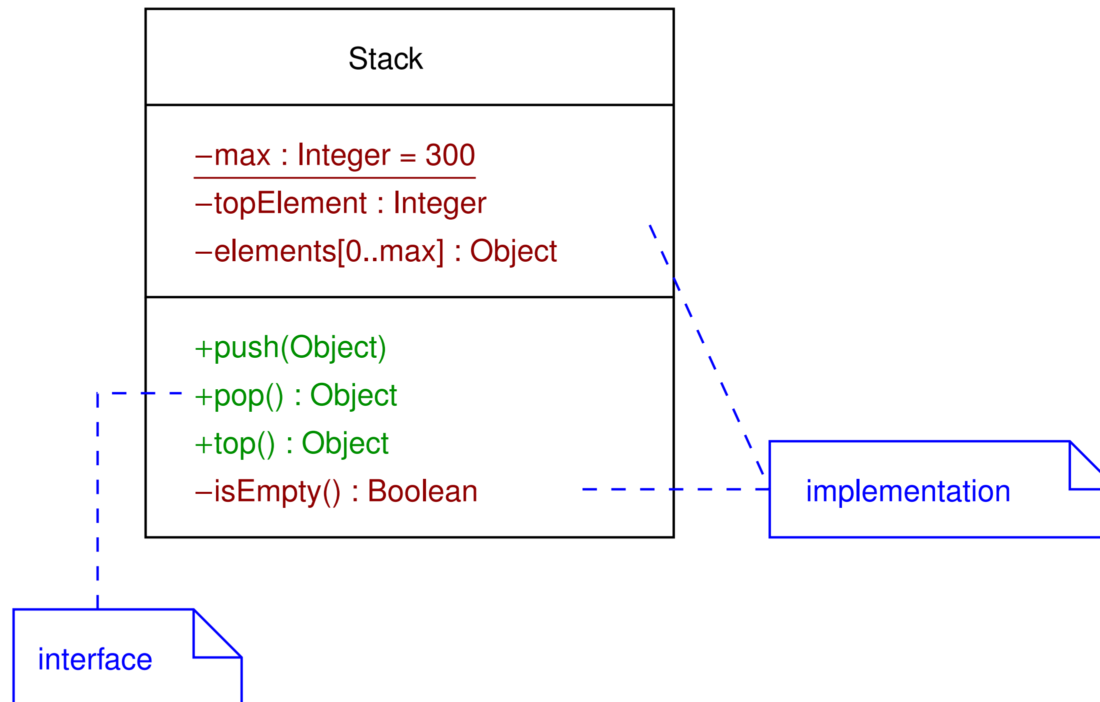
In a class definition, the distinction between interface and implementation is expressed by visibility markers associated to the class elements (variables and methods):

- a public element (denoted “+”) can be accessed from every class.

- a private element (denoted “–”) can only be accessed from its own class.

Note: There exist other visibility levels, that will be studied later.

Example



Class and Instance Variables

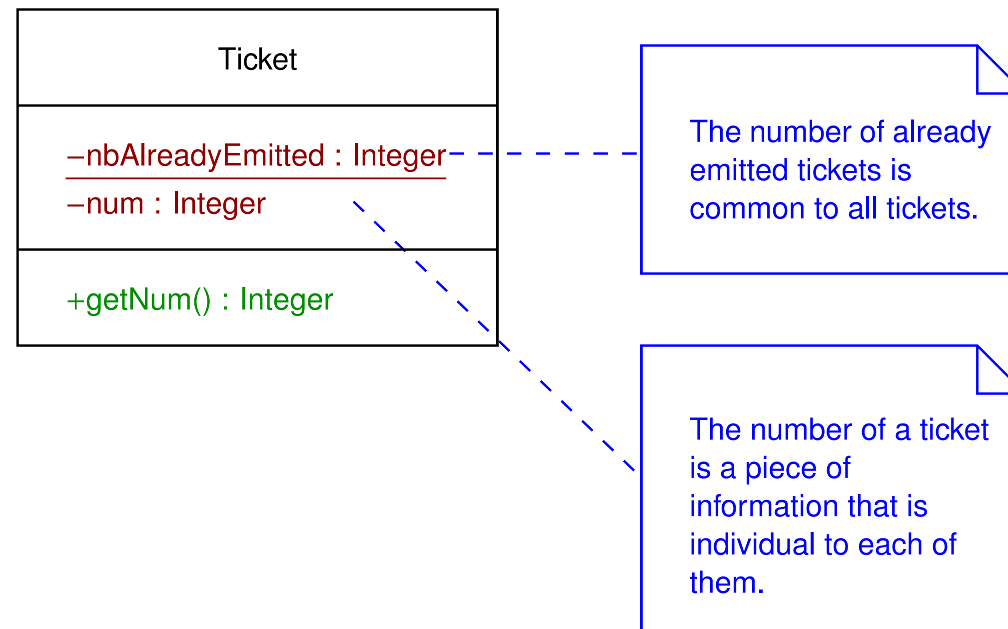
The variables defined by a class can belong to two categories:

- Class variables have a value that is shared between all instances of the class.
- Instance variables have a value that is individual to each instance of the class.

In other words, a class variable corresponds to a storage area associated to a class. An instance variable represents a storage area within objects.

Example

By convention, class elements are underlined.



Class and Instance Methods

Like variables, the methods of an object can be classified as follows:

- Class methods can only access class variables.
- Instance methods are able to read and write all the variables of the object.

Notes:

- Unlike variables, instance methods are shared between multiple instances of a same class.
- Class methods can be invoked by sending a message either to the class to which they belong, or to any instance of this class.

The Java Programming Environment

Principles:

- The semantics of the programming language is defined independently from hardware details.
- Source code does not compile into machine code, but into intermediate code (*bytecode*) that is independent from the host architecture.
- Bytecode is either interpreted or compiled *Just In Time (JIT)* by a virtual machine. This machine can either be standalone, run inside an application (such as a browser), or be implemented in hardware.
- The programming environment provides a standard library of general-purpose classes.

Advantages:

- One can develop and distribute software products without knowing the details of their runtime environment.
- Good protection against malicious code can be obtained by sandboxing the Java virtual machine.

The Structure of Java Programs

In its simplest form, a Java program is composed of a sequence of class definitions:

```
class FirstClass
{
...
...
}

class SecondClass
{
...
...
}

class ThirdClass
{
...
...
}

...
```

Note: The following problems will be addressed later:

- How can classes be instantiated in order to create objects?
- How can a complex program be decomposed into several modules?

Defining a Class

A class definition is composed of

- a declaration that specifies the name of the class as well as some of its properties, and
- a body that defines the variables and the methods of the class.

In its simplest form, a class declaration is expressed as follows:

```
[ public ] class ClassName
```

- `public`: The scope of the class extends to the whole program (by default: to only the current module).
- There exist other class qualifiers, that will be studied later.

Note: Java compilers impose that public classes share their name with the source file in which they are defined.

Inside the body of a class, one finds

- variable declarations, and
- method declarations.

It is good programming practice to declare variables before methods, but this is not compulsory.

Declaring a Variable

Within the body of a class, a variable declaration takes the following form:

[visibility] [attributes] type variableName [= initialValue];

The visibility marker can be one of the following keywords:

- `public`: The variable can be accessed from any class.
- `private`: The variable can only be accessed from the same class.

  Note: Access to a `private` instance variable is not limited to the object that owns this variable. Any instance of the same class can read or write such a variable!

A variable that does not declare a visibility marker can only be accessed from the classes that belong to the same module.

Variable attributes are specified by a combination of the following keywords:

- `static`: The variable is a class rather than an instance variable.
- `final`: The value of the variable cannot be modified after its first assignment.

  Note: This property has to be enforced at compile time, without executing the program. This is achieved by imposing that it holds for every potential execution path.
- `transient`: The value of the variable is not considered to be part of the state of the object.
- `volatile`: The value of the variable can be read or written by mechanisms that are external to the current fragment of code.

Primitive Types

In Java, the type of a variable can take several forms. A variable with a primitive type represents a storage area suited for a simple value.

The following primitive data types are available:

`byte`	Signed integer	8 bits
`short`	Signed integer	16 bits
`int`	Signed integer	32 bits
`long`	Signed integer	64 bits
`float`	Floating-point number (IEEE 754)	32 bits
`double`	Floating-point number (IEEE 754)	64 bits
`char`	Unicode character	16 bits
`boolean`	Boolean value	1 bit

Notes: The properties of those data types are hardware-independent.

Reference Types

A variable declared with a reference type (also known as a *reference variable*) is able to store a reference to an instance of a given class (in other words, a pointer to this object).

Such a variable is declared by using the referenced class name as a type:

```
[ ... ]  ClassName  variableName  [ = initialValue ];
```

Notes:

- The special value `null` denotes an empty reference, that does not point to any object.
- The keyword `this` denotes a reference to the current object.

- The instance element (variable or method) `el` of the object referenced by the variable `v` is accessed by evaluating the expression `v.el`.

- The class element `el` of a class `C` can be accessed by evaluating either the expression `C.el`, or `v.el` for any variable `v` referencing an instance of `C`.

Array Types

An *array type* is a particular case of reference type. An array variable stores a reference to a vector of data. The components of this vector can either have a primitive or reference type, or be themselves arrays.

The declaration of an array variable takes one of the following forms:

```
primitiveType[] variableName [ = initialValue ];
primitiveType[][]...[] variableName [ = initialValue ];
className[] variableName [ = initialValue ];
className[][]...[] variableName [ = initialValue ];
```

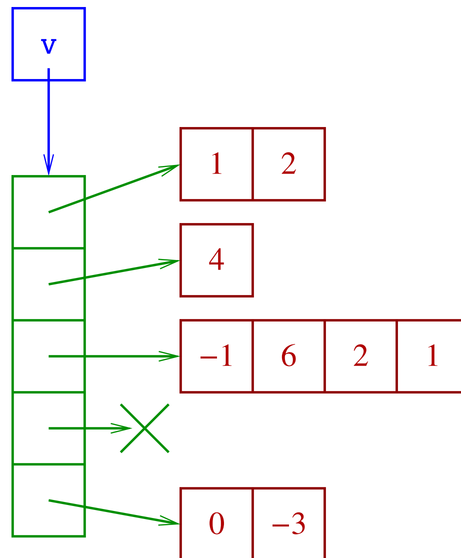
Example: The declaration

```
int[][] v;
```

defines a variable v that stores a reference to a vector. Each component of this vector stores a reference to a vector of integers.

The size of the vectors is not specified in their declaration, and can be freely chosen at instantiation time. The declaration in the previous example can thus be used for referring to a non-rectangular array.

Illustration:



Declaring a Method

A method declaration takes the following form:

```
[ visibility ] [ attributes ] returnType methodName(type1 param1,
                                                    type2 param2, ...)
{
  ...
}
```

The visibility marker is identical to the one of variables. The attributes are specified by a combination of the following keywords:

- `static`: The method is a class rather than an instance method.
- `native`: The method is implemented outside the program (possibly, in a different language from Java).

Notes:

- Other attributes will be studied later.
- The return value and the parameters of the method can either have a primitive, reference, or array type.
- The pseudo return type `void` specifies that the method does not return a value.
- For a method that does not take arguments, the list of parameters is empty.
- The methods labeled `native` have their body replaced by a semicolon (“`;`”).

Polymorphic Methods

A class may contain several methods that share the same name *(overloading)*, provided that the number and/or type of their parameters differ.

Upon receiving a message, the method to be invoked is then identified on the basis of the message name together with the number and types of the provided arguments (*signature* of the method).

Example:

```
class GeometricalShape
{
...

  public void move(int x, int y)
  {
  ...
  }

  public void move(float x, float y)
  {
  ...
  }

  public void move(IntVector v)
  {
  ...
  }

...

}
```

The Body of a Method

The statements contained in the body of a method belong to three main groups:

- declarations of local variables,
- control instructions, and
- expressions (including assignments).

Each statement is terminated by a semicolon ";".

A sequence of statements can be grouped into a block, delimited by curly braces "{ }". A block can generally play the same syntactic role as a single instruction. Blocks can be nested into other blocks. The body of a method can be seen as a block that contains the implementation of this method.

Local Variables

Local variables are declared in much the same way as instance or class variables, except that such declarations cannot have visibility markers. The `final` attribute is allowed.

```
[ final ] type variableName [ = initialValue ];
```

A local variable declaration can appear at any place in a block. The scope of a local variable starts immediately after its declaration, and ends with the innermost block containing this declaration.

Local variables define temporary storage areas that are allocated when their declaration instruction is executed, and freed upon leaving the block containing this declaration. Local variables are not stored in objects or classes!

Note: The parameters of a method can be considered to be a particular case of local variables. When the method is invoked, such variables are initialized with the arguments accompanying the corresponding message.

One can thus modify the value of a parameter in a method, like for any other local variable. The effect of such a modification is limited to the body of the invoked method, and is not visible to the calling method. (In Java, parameter passing is performed *by value*.)

Control Instructions

Branching instructions:

- Binary decision:

```
  if (expression)
    instruction;
[ else
    instruction; ]
```

- Multiple decision:

```
switch (expression)
{
  case value:
    instructions;
    break;
  case value:
    instructions;
    break;
  ...
[ default:
    instructions; ]
}
```

Loop instructions:

- Loop with check at the beginning:

```
while (expression)
  instruction;
```

- Loop with check at the end:

```
do
  instruction;
while (expression);
```

- General loop:

```
for ([ expression1 ]; [ expression2 ]; [ expression3])
   [ instruction ];
```

This statement is equivalent to the following construct:

```
{
   expression1;
   while (expression2)
      {
         instruction;
         expression3;
      }
}
```

Notes:

- *expression1* can be replaced by a local variable declaration.
- It is possible for *expression1* and *expression3* to be composed of several subexpressions separated by commas (“,”).
- Modern versions of Java also support an enhanced form of the `for` instruction, suited for iterating over arrays or containers of objects.

Control break instructions:

- End of method:

```
return [ expression ];
```

- Exit from a loop or multiple decision:

```
break [ label ];
```

- Jump to the next iteration:

```
continue [ label ];
```

Note: The `label` that appears in the two previous statements makes it possible to specify precisely the instruction targeted by the control break in the case of nested loops.

Example:

```
out: for (int i = 0; i < 10; i++)
       for (int j = 0; j < 20; j++)
         {
           if (f(i) < f(j))
             break out;
           ...
         }
```

Expressions

The Java language defines the following operators, in decreasing order of precedence:

<table>
<tr><td>access to elements</td><td>.</td></tr>
<tr><td>postfix operators</td><td>expr++, expr--</td></tr>
<tr><td>unary operators</td><td>++expr, --expr, +expr, -expr, ~, !</td></tr>
<tr><td>type casting</td><td>(type) expr</td></tr>
<tr><td>object creation</td><td>new</td></tr>
<tr><td>products</td><td>*, /, %</td></tr>
<tr><td>sums</td><td>+, -</td></tr>
<tr><td>shifts</td><td><<, >>, >>></td></tr>
<tr><td>comparisons</td><td><, >, <=, >=, instanceof</td></tr>
<tr><td>equality test</td><td>==, !=</td></tr>
<tr><td>binary and</td><td>&</td></tr>
<tr><td>binary xor</td><td>^</td></tr>
<tr><td>binary or</td><td>|</td></tr>
<tr><td>logical and</td><td>&&</td></tr>
<tr><td>logical or</td><td>||</td></tr>
<tr><td>conditional evaluation</td><td>? :</td></tr>
<tr><td>assignments</td><td>=, +=, -=, *=, /=, %=, &=, ^=, |=,
<<=, >>=, >>>=</td></tr>
</table>

Notes:

- Expressions are usually evaluated from left to right, except in specific cases (e.g., assignments are evaluated from right to left).

- The evaluation of operands of logical operators (&&, `||`) relies on short circuiting. In other words, the right-hand operand is not evaluated if the value of the left-hand operand suffices for deducing the value of the expression. On the other hand, the binary operators (&, `|`, `^`) always evaluate both of their operands.

- The instantiation operator `new` will be studied in the next chapter.

Example of class definition

```
public class Counter
{
  private int val = 0;

  public void countUp()
    {
      countUp(1);
    }
  public void countUp(int n)
    {
      n += val;
      if (n > val)
        val = n;
    }
  public void countDown()
    {
      if (val > 0)
        val--;
    }
  public boolean isZero()
    {
      return val == 0;
    }
}
```

Chapter 3

Messages, Instantiation, and Initialization of Objects

Messages

Recall that a message is a request sent to an object or a class in order to perform a specific operation. A message is labeled by the name of this operation, accompanied with an optional set of arguments (i.e., values for the parameters of the operation).

Sending a message is done synchronously: The execution of the invoking method is suspended while the message is sent, and resumes only when the invoked method has finished its execution. When a method finishes, it has the possibility of returning a value to its caller.

In Java, messages are sent by evaluating expressions of the form

```
ref.methodName(expr1, expr2, ...)
```

where

- *ref* is either a reference to the recipient of the message, or the name of a class.
- *methodName* is the label of the message, and corresponds to the name of the method to be invoked.
- (*expr1*, *expr2*, . . .) evaluates into the list of arguments of the message.

The value of such an expression becomes equal to the value (if any) returned by the invoked method, once this method has finished its execution.

Note: The interactions between an object and its environment are not limited to exchanging messages. Remember that non-private variables can also be read or written by code that is external to the class!

Creating Objects

The simplest way of creating an object consists in evaluating an expression of the form

```
new Class()
```

where *Class* is the name of the class that has to be instantiated.

The evaluation of such an expression creates a new object, and returns a reference to this object.

It is of course possible to assign an initial value to a reference variable using an instantiation expression.

Example:

```
Stack s = new Stack();
```

Instantiation expressions can also appear as components of more complex expressions.

Example:

```
int n = new IntVector().maxSize();
```

Instantiating Arrays

The simplest way of instantiating an array is to evaluate an expression of the form

```
new type[expr]
```

where *type* defines the type of the array elements, and the evaluation of *expr* provides the number of elements to be allocated.

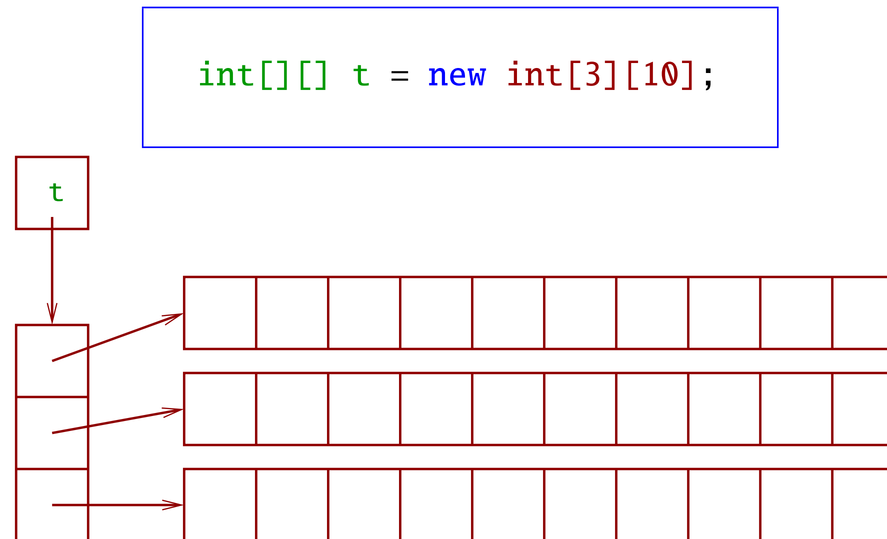
Example:

```
int[] t = new int[10];
```

Instantiating multi-dimensional arrays is done in a similar way:

```
new type[expr1][expr2]...
```

Example:



It is not mandatory to specify the size of all components of an array. The following construct is allowed:

$$\texttt{new}\ \mathit{type}\ \underbrace{\texttt{[}\mathit{expr}\texttt{][}\mathit{expr}\texttt{]}\ldots}_{n}\ \underbrace{\texttt{[][]}\ldots}_{m}$$

Such an expression instantiates a n-dimensional array, each element of which is a reference to a (non instantiated) m-dimensional array.

Example: Construction of a triangular array:

```
int[][] triangle = new int[10][];

for (int i = 0; i < 10; i++)
  triangle[i] = new int[i + 1];
```

Notes:

- In Java, array elements are always indexed starting from 0.
- All array objects possess a public final variable `length`, the value of which is equal to their number of allocated elements.

Array Initializers

One can both instantiate and initialize an array in a single expression:

```
new type[]{val1, val2, ...}
```

Example:

```
int[] t = new int[]{10, 20, 30};
```

Notes:

- Arrays can also be initialized (as well as implicitly instantiated) in variable declarations:

```
int[] t = {10, 20, 30};
```

- The latter construct is less general, since it is only valid in variable declarations.
- Both variants generalize to multimensional arrays by nesting braces.

Constructors

When an object is created, its instance variables are initialized using the expressions specified in their declaration.

It is however not always possible to describe all initialization operations by expressions. An alternative is to define a constructor, which contains a set of instructions to be executed each time that the class is instantiated.

Constructor definitions appear at the same place as method definitions, and take the following form:

```
[ visibility ] Class()
{
  ...
}
```

where *Class* is the name of the instantiated class.

The visibility marker of a constructor is syntactically identical to the one of a method. Such a marker controls the possibility of instantiating the class.

Example: A class with a constructor declared `private` can only be instantiated by instances of the same class, as well as from its own class methods.

Instantiation Parameters

All instances of a class do not necessarily need to be instantiated in exactly the same way. When instantiating a class, one may provide arguments that can influence how the object is initialized. The instantiation expression then takes the following form:

```
new Class(expr1, expr2, ...)
```

In order for this expression to be valid, the class needs to define a constructor that accepts the corresponding combination of parameters. Such a constructor is declared as follows:

```
[ visibility ] Class(type1 param1, type2 param2, ...)
{
...
}
```

Notes:

- A class can define several constructors, provided that the number and/or type or their parameters differ *(constructor polymorphism)*.
- By default, a class without explicitly defined constructors is automatically provided with an empty constructor, that accepts an empty list of parameters.
- An instantiation expression is only valid when the corresponding class admits a constructor with parameters that match (in number and type) the arguments of the operation.
- A constructor can invoke another constructor with the keyword this:

```
this(expr1, expr2, ...);
```

Such an operation is only valid if it appears as the first instruction of the caller constructor.

- If an instance variable that is declared final is not initialized in its declaration, then it must be assigned a value in every constructor.

Static Initialization Blocks

Constructors are only executed when their class is instantiated, hence they are not able to initialize class variables before their first use.

For this purpose, one employs instead static initialization blocks, defined as follows (at the same place as constructors):

```
static
{
    ...
}
```

A class can contain any number of initialization blocks. These blocks are executed the first time that the class is loaded by the runtime environment. They are always executed in their order of appearance in the class definition.

Note: Final class variables must be initialized either in their declaration or in a static initialization block.

Destroying Objects

Objects are destroyed by means of a garbage collection mechanism: The memory allocated to objects that are not reachable anymore by currently active code is automatically freed.

Notes:

- Garbage collection is generally performed asynchronously, i.e., at the same time as the program runs. It can however become synchronous (suspending temporarily program execution) in the case of insufficient memory.
- When large objects are not useful anymore to the program, it is a good idea to drop all references to these objects.
- The garbage-collection mechanism is able to correctly detect and handle cyclic references between unreachable objects.

Running a Java Program

There exist several types of Java programs. In order to build a standalone application, one needs to define a public class that implements a `main` method with the following signature:

```
public static void main(String[] args)
{
  ...
}
```

This class method is then invoked by the runtime environment after the public class has been loaded.

The parameter *args* contains a reference to an array that provides the command-line arguments of the program.

Example:

```
public class Stack
{
  private final static int max = 300;
  private int nbElements;
  private Object[] contents;

  public Stack()
  {
    nbElements = 0;
    contents = new Object[max];
  }

  public void push(Object e)
  {
    if (nbElements < max)
      contents[nbElements++] = e;
  }

  public Object pop()
  {
    if (nbElements > 0)
      {
        Object e = contents[nbElements - 1];
        contents[--nbElements] = null;
        return e;
      }
    else
      return null;
  }

...
```

```
...

  public Object top()
  {
    if (nbElements > 0)
      return contents[nbElements - 1];
    else
      return null;
  }
}
```

Sample test program:

```
public class TestStack
{
  public static void main(String[] args)
  {
    Stack s = new Stack();

    s.push(Integer.valueOf(10));
    s.push(Double.valueOf(3.14));
    System.out.println(s.pop());
    System.out.println(s.pop());
  }
}
```

Notes:

- The method `valueOf()` of the classes `Integer` and `Double` returns a reference to an object (either already existing or newly allocated), representing the value specified by its argument.
- Such a conversion of primitive values into representing objects can be performed automatically by modern versions of Java.
- We will learn later (in Chapter 6) how to better handle error situations.

Chapter 4

Object-Oriented Development Methodology

Development Tasks

Generally speaking, developing an object-oriented program amounts to carrying out the following tasks:

1. Specifying a set of usage scenarios describing the expected behavior of the system in typical situations.
2. Defining the components of the program.
3. Assigning to each component a set of responsibilities, and organizing their delegation.

Basic rules:

- Each software component must be assigned responsibilities that are simple and well defined.
- The interactions between components must be as limited as possible.

CRC Cards

A simple way of identifying components and defining their responsibilities consists in playing the scenarios while imagining that the system has already been developed, reasoning about its expected behavior.

During this process, the features of each component are summarized in a *CRC (Component, Responsibilities, Collaborators) card*, of the following form:

Component Name	Collaborators
Description of the responsibilities assigned to the component	List of other components with which the component interacts

Principles:

- The text written on each card must remain simple. In particular, non-essential details should be omitted.
- It is allowed to erase and rewrite the contents of cards when studying the scenarios.

Example of Development: Interactive Recipe Book

The project consists in developing an application for managing a database of recipes, and planning meals.

1. Choice of scenarios

After having considered several possibilities, one chooses to make the program display a list of possible operations when it starts up:

1. Consulting recipes.
2. Adding a new recipe to the database.
3. Modifying or commenting a recipe from the database.

4. Planning new meals.
5. Consulting or modifying planned meals.
6. Generating a shopping list for the ingredients of the meals planned for the following days.

One notices that these operations naturally belong to two main groups:

- Managing recipes (from 1 to 3),
- Planning meals (from 4 to 6).

2. Main components

By studying the scenarios, one identifies three main components:

- The welcome screen, aimed at presenting a list of possible operations to the user, and prompting him/her to choose one of them.
- The recipe database, responsible for storing and querying recipes, making it possible to display and modify them, as well as to add new recipes.
- The meal planner, that manages the assignment of meals to some calendar days, and allows to display and modify planned meals.

The description of these components motivates the definition of three additional components:

- The recipe, characterized by a list of ingredients and a description.

- The meal, that corresponds to a recipe prepared in a specific quantity.
- The day menu, that assigns one or several meals to a given calendar day.

3. Assignment of responsibilities

Welcome screen

Collaborators

1. Displays a list of possible operations.
2. Reads the choice made by the user.
3. Depending on this choice, invokes the recipe database or the meal planner.

- Recipe database
- Meal planner

Recipe database

Collaborators

Manages a set of recipes.

- Makes it possible to select a recipe in order to consult, modify, or print it, and then invokes this recipe.
- Makes it possible to select a recipe and returns this recipe.
- Makes it possible to create a new recipe, that is then added to the database.

Collaborators:

- Recipes

Meal planner

Collaborators

- Makes it possible to select a calendar day in order to plan a corresponding day menu, or to consult, modify, or print the menus planned for that day. Invokes the relevant day menus.
- Makes it possible to select a time interval, and then generates the list of ingredients needed for preparing the day menus planned for that period, by successively querying these day menus.

Collaborators:

- Day menus

Recipe

Collaborators

—

Maintains a list of ingredients and a description.

- Makes it possible to consult, modify, or print the recipe.
- Is able to generate a list of ingredients needed by the recipe.

Meal

Collaborators

- Recipes
- Recipe database

Associates a given preparation quantity to a recipe.

- Makes it possible to choose a recipe (by making a request to the recipe database), and to specify the quantity in which it needs to be prepared.
- Is able to display or print a description of the meal.
- Is able to generate a list of ingredients needed by the meal.

Day menu

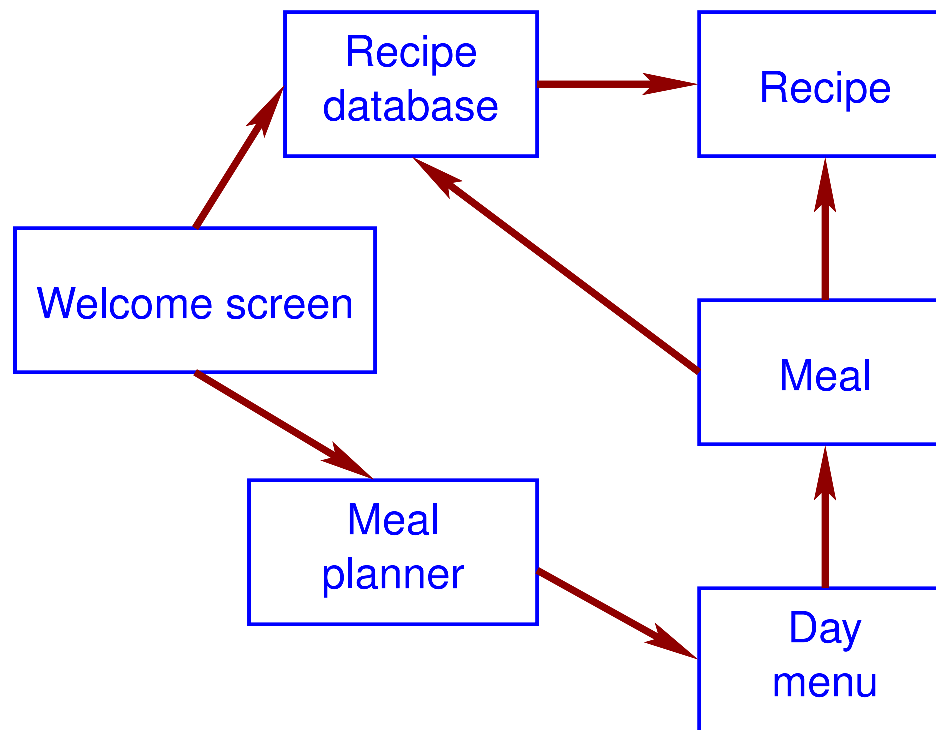
Assigns one or many meals to a calendar day.

- Makes it possible to consult, modify, or print the meals that compose the menu (by invoking these meals).
- Is able to generate a list of ingredients needed by the meals that compose the menu.

Collaborators

- Meals

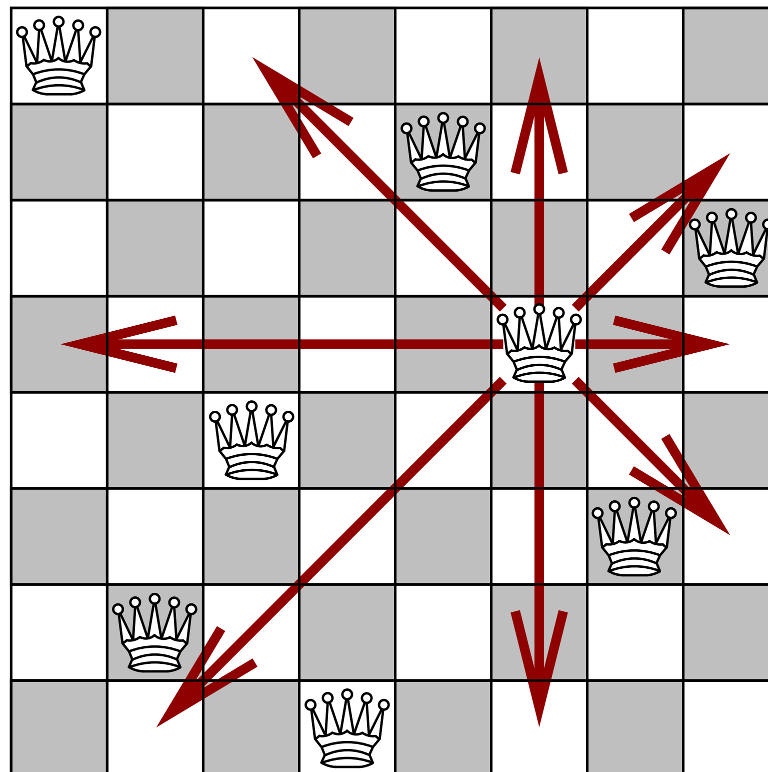
Summary of Interactions



Example 2: The Eight Queens Puzzle

The goal is to write a program that is able to solve the following problem:

> *How can eight queens be placed on a chessboard in such a way that they do not threaten each other?*



1. Choice of scenarios

There is only one scenario: the program searches for a solution, in other words, suitable coordinates on the board for each queen. If successful, it then displays this solution.

2. Main components

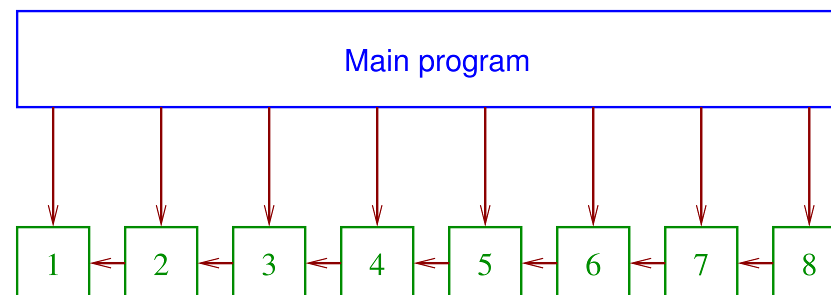
A simple strategy is to define the following components:

- the main program, responsible for initiating the search for a solution, and
- eight queens that will collaborate with each other in order to find a solution.

3. Assignment of responsibilities

For the sake of simplicity, one makes the following choices:

- Each queen is permanently assigned a fixed column on the chessboard. While searching for a solution, only the row in which a queen is located can thus be modified.
- Each queen can only communicate with its nearest left neighbor.
- The main program can interact with all queens.



Main program

Collaborators

1. Creates successively eight queens and places them in their respective column, proceeding from left to right. Asks each queen to search for a solution that involves itself as well as all the queens to its left.
2. If successful, asks the eighth queen to display the complete solution.

- The eight queens

Queen

Collaborator

- Nearest left neighbor

Remains located in a fixed column, and can move within this column.

- Can search for a solution for itself as well as all the queens to its left, by moving within its column, and asking some operations to its nearest left neighbor.
- Is able to check whether a given square on the chessboard is threatened by itself or a queen to its left.
- Is able to advance to the next row, the queens to its left maintaining a configuration in which they do not threaten each other.
- Assuming that a solution has been found for itself and the queens to its left, is able to search for another solution.
- Can display its position on the chessboard as well as the position of all the queens to its left.

Java Implementation

Interface of classes:

- Main program: class `EightQueens`:
  - `public static void main(String[] args)`
- Queen: class `Queen`:
  - `public Queen(int c, Queen n)`
  - `public boolean findSolution()`
  - `public boolean threatens(int c, int r)`
  - `public boolean moveOn()`
  - `public boolean nextSolution()`
  - `public void display()`

Main program:

```
public class EightQueens
{
  public static void main(String[] args)
  {
    Queen q = null;

    for (int i = 1; i <= 8; i++)
      {
        q = new Queen(i, q);
        if (!q.findSolution())
          {
            System.out.println("No solution found.");
            return;
          }
      }

    q.display();
  }
}
```

Queen:

```
class Queen
{
  private final int column;
  private int row;
  private Queen neighbor;

  public Queen(int c, Queen n)
  {
    this.column = c;
    row  = 1;
    neighbor = n;
  }

  public boolean findSolution()
  {
    if (neighbor == null)
      return true;

    while (neighbor.threatens(column, row))
      if (!moveOn())
        return false;

    return true;
  }

  public boolean threatens(int c, int r)
  {
    if (c == column || r == row || c - column == r - row || c - column == row - r)
      return true;
```

```
    if (neighbor == null)
      return false;

    return neighbor.threatens(c, r);
  }

  public boolean moveOn()
  {
    if (row < 8)
      {
        row++;
        return true;
      }

    if (neighbor == null)
      return false;

    row = 1;
    return neighbor.nextSolution();
  }

  public boolean nextSolution()
  {
    return moveOn() && findSolution();
  }

  public void display()
  {
    if (neighbor != null)
      neighbor.display();

    System.out.println("(" + column + ", " + row + ")");
  }
}
```

Chapter 5

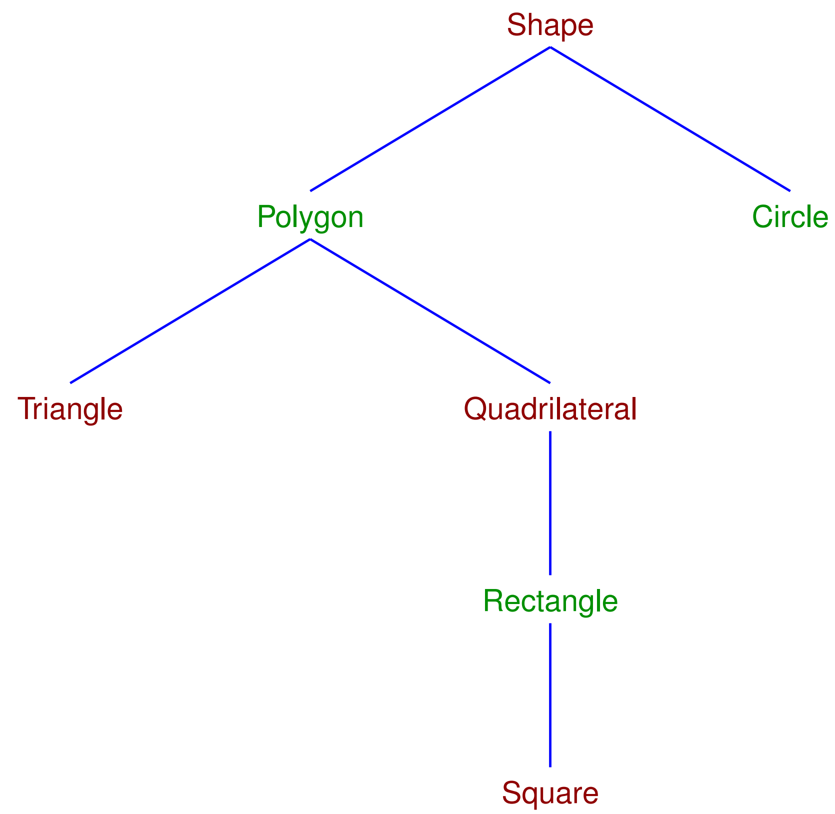
Inheritance

The Class Hierarchy

The classes that compose an object-oriented program are generally not independent, but are linked together by a hierarchical relation:

- The descendants of a class (its *subclasses*) are specializations of this class.
- The ancestors of a class (its *superclasses*) correspond to generalizations.

Illustration:



Inheritance

Inheritance is a mechanism related to the hierarchical organization of classes.

Principles:

- By default, a subclass inherits the variables and methods of its superclasses.
- A subclass is however able to define new variables and methods, as well as to replace *(override)* inherited elements by its own.
- Inheritance is transitive.

Object Polymorphism

Instances of a subclass must be able to accept all messages that are accepted by instances of its superclasses.

In other words, the interface of a subclass necessarily has to include the interface of its superclasses.

If `ClassS` is a direct or indirect subclass of `ClassG`, then one can thus consider that any instance of `ClassS` is also an instance of `ClassG`.

As a consequence, one can assign to a reference variable of type `ClassG` a reference to an object instantiated from `ClassS` (or from any other subclass of `ClassG`).

Notes:

- At compile time, it is sometimes impossible to know precisely the class of an object referenced by a variable *(object polymorphism)*.
- A reference expression of type `ClassC` can only be employed for sending messages that correspond to the methods declared by `ClassC` (or its superclasses), or for accessing the variables defined by `ClassC` (or its superclasses).

Abstract Classes

It is sometimes useful to define classes that are not intended to be instantiated, but that define elements that can be inherited by other classes. Such classes are said to be *abstract*.

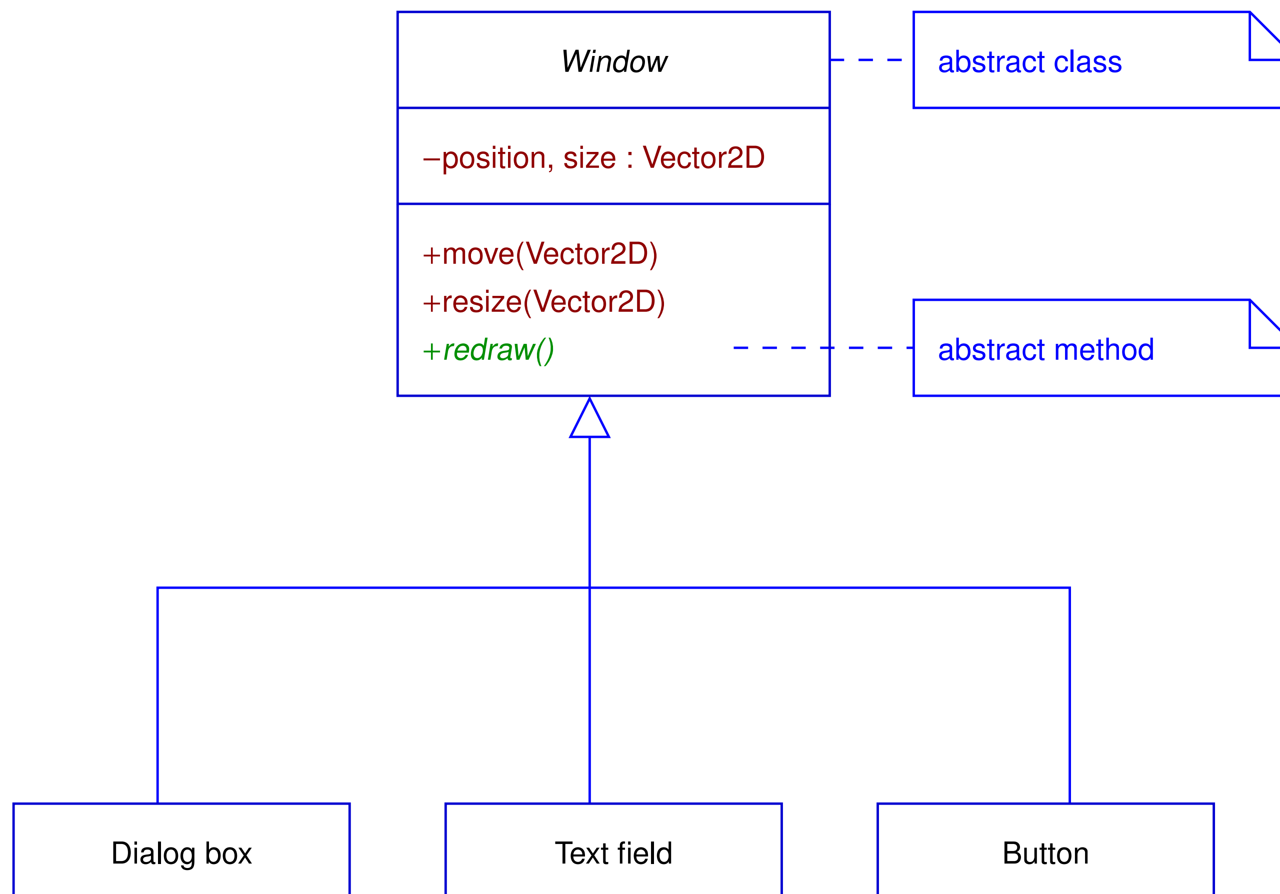
An abstract class does not necessarily need to provide a body for each of its methods. An *abstract method* is a method without an implementation. A non-abstract class (*concrete class*) that inherits an abstract method is then required to implement it.

Defining an abstract class makes it possible

- to share common pieces of implementation between subclasses (non-abstract methods), as well as
- to specify common interface elements between subclasses (abstract methods).

Example:

Class hierarchy of a windowing toolkit (the name of abstract elements is written in italics).



The Substitution Principle

The inheritance mechanism can be exploited in several ways. A first possibility consists in requiring that subclasses respect all the specifications of their superclasses.

Substitution principle: *In all situations in which a class can be employed, it should be possible to substitute a subclass without breaking the application. In other words, all the properties of the class that may affect the correctness of programs must be preserved by the subclass.*

This principle expresses an ideal view of the class hierarchy. It is not always respected.

If the definition of a subclass satisfies the substitution principle, then this subclass represents a specialized version of its superclasses. Such a subclass is then said to provide a subtype of the types defined by the superclasses.

Notes:

- No language or programming environment offers automatic mechanisms for enforcing the substitution principle in subclass definitions. In object-oriented programming, a subclass is not always a subtype.
- In some specific cases, it is useful to deliberately exploit inheritance without respecting the substitution principle.

We are now going to study some of the most frequent applications of the inheritance mechanism.

1. Specialization

Description: *An instance of the subclass represents a particular case of its superclass instances.*

From this definition, it follows that the subclass satisfies all the properties that hold for its superclasses, and therefore the substitution principle is respected. The subclass is thus a subtype.

Example: In a windowing toolkit, a class *Window* implements some operations over a rectangular zone of the screen (resize, move, ...). The class is specialized into subclasses *TextField*, *DialogBox*, *PulldownMenu*, ... Each of these subclasses satisfies all the properties of *Window* instances.

Specialization is the most frequent and most relevant application of inheritance.

2. Specification

Description: *A class defines behavior that it does not implement itself, but that is meant to be implemented by its subclasses.*

This mechanism makes it possible to impose a common interface (in other words, the requirement of implementing similar methods) across several classes.

This application can be seen as a particular case of specialization.

Example: In simulation software, the abstract class *Animable* defines the set of all the methods that must be implemented by an object in order for this object to be involved in a simulation.

3. Construction

Description: *A subclass exploits functionalities implemented by its superclasses, without becoming a subtype of these classes.*

This mechanism is often employed in order to modify the interface of a class, by altering the name of its methods or the type of its variables.

Examples:

- A library contains a class *Dictionary* that stores pairs of the form (*key*, *value*), where the type of *key* can be freely chosen. For a compiler application, one defines a subclass *SymbolTable* that associates values to the identifiers appearing in the source code being compiled.

A *SymbolTable* cannot always replace a *Dictionary* (since it cannot accept arbitrary keys), but the implementation of this class can benefit from inheritance in order to reuse some of the functionalities of its superclass.

- A library of input/output classes contains a class *BinaryOutput* aimed at redirecting a stream of data to a file. From this class, one can define a subclass *TextOutput* for saving character strings to a file.

The substitution principle is not satisfied by this application of inheritance.

4. Generalization

Description: *A subclass modifies or completely overrides some inherited operations, in order to make them more general.*

This application can be seen as being totally opposed to specialization, since the behavior of the subclass is now more general than the one of its superclasses.

This mechanism makes it possible to generalize the properties of a class in situations where it is impossible or forbidden to modify its definition (in particular, to place it elsewhere in the class hierarchy).

Example: A windowing toolkit is organized around a class *Window* that displays its content in black and white.

In order to generalize this class to full-color display, one can define a subclass *ColorWindow* in which all drawing methods inherited from *Window* are overridden by full-color equivalents.

This application of inheritance explicitly violates the substitution principle. It should only be used in situations where it is impossible to reorganize the class hierarchy.

5. Extension

Description: *A subclass adds new operations to those inherited from its superclasses, without affecting the inherited operations.*

The difference between *extension* and *generalization* is that the former is limited to extending the functionalities of its superclasses while preserving all their properties. The substitution principle therefore remains satisfied.

Example: A class *Collection* is able to store arbitrary sets of objects. This class can be extended into a subclass *IndexedCollection* that satisfies all the properties of a collection, but adds a mechanism for searching efficiently for an element with a given value.

Note: The distinction between *specialization* and *extension* is not strict.

6. Limitation

Description: *A subclass restricts the usage modalities of some inherited operations.*

Example: A library contains the class *DoublyLinkedList* that allows to add and remove elements at any position in a list. A programmer that needs to implement a stack data structure can define the class *Stack* as a subclass of *DoublyLinkedList*, and then override the methods for adding and removing elements, so as to make then inoperative at all positions other than the origin of lists.

Like in the case of *generalization*, this mechanism explicitly produces subclasses that are not subtypes. It should thus be used only when there is no other alternative.

7. Variation

Description: *A subclass and its direct superclass are variants of each other, the direction of the hierarchical relation between them being chosen arbitrarily.*

Example: A *Mouse* and a *PenTablet* are both pointing devices. Defining one of these classes as a subclass of the other makes it possible to share common parts of their implementation.

In this example, a better solution would be to define an abstract class *PointingDevice*, and express both *Mouse* and *PenTablet* as subclasses of this abstract class.

Variation leads to violating the substitution principle. Like *generalization* and *limitation*, it should be used only in situations where the class hierarchy cannot be modified by the programmer.

8. Combination

Description: *A subclass inherits the elements from more than one direct superclass.*

This is only possible if the programming language or environment allows multiple inheritance. In this case, the hierarchical relation between classes does not necessarily take the form of a tree, but only of a directed acyclic graph (DAG).

Multiple inheritance is a source of subtle problems:

- Ambiguities are present when several superclasses define elements that share the same signature.
- An element can be transitively inherited from a class by following more than one path in the hierarchy graph.

It is then not obvious to select the constructors that need to be invoked when a subclass is instantiated, and to define the order in which they must be executed.

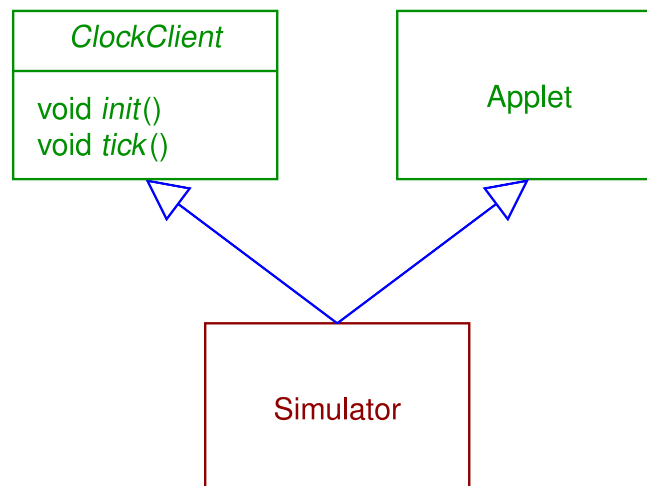
- It is difficult to enforce the substitution principle.
- . . .

There is however one situation for which it is quite useful to employ multiple inheritance: When a subclass is a subtype of one of its direct superclasses, and the other branches of the hierarchical relation involving this subclass only express specification.

Example: In Java, an *applet* is a program that is designed to run inside another application, for instance a WWW browser. Such a program can easily be constructed using *specialization*, by defining the main class of the program as a subclass of the class *Applet* provided by the Java standard library.

Consider a program that contains an abstract class *ClockClient* defining all the methods that need to be implemented by a class in order to subscribe to a periodic notification service.

In order to develop an applet implementing a simulator, one needs to update periodically the position of the simulated objects, using the notification service. This can be achieved by defining the following class hierarchy:



One observes that the branch *Simulator* → *Applet* expresses *specialization*, and the branch *Simulator* → *ClockClient* corresponds to *specification*.

Inheritance in Java

A class *ClassS* is defined as a direct subclass of the class *ClassG* by declaring it as follows:

```
class ClassS extends ClassG
{
...
...
}
```

The subclass then inherits all the elements of the superclass.

Notes:

- Multiple inheritance between classes is not allowed in Java. (In other words, a class can only be the direct subclass of at most one class.)
- There exists a class located at the root of the hierarchy, called `Object`.

- Classes declared with the attribute `final` cannot be specialized into subclasses. Methods declared `final` cannot be overridden.

- Abstract classes and methods are declared using the attribute `abstract`.

Object Polymorphism in Java

Object polymorphism can sometimes lead to ambiguous situations.

Example: Assume that `ClassS` and `ClassG` both define a method `m()`, with different implementations.

Question: Does the following fragment of code invoke the method from `ClassS` or `ClassG`?

```
ClassG v;

v = new ClassS();
v.m();
```

There are two possibilities:

- The type of the object referenced by the variable v could be determined from the declaration of v *(static link)*.
- The type of the object referenced by the variable v could be a proper feature of this object, set when this object is instantiated *(dynamic link)*.

The Java language uses the following policy:

- Instance variables of an object are accessed by static link.
- Methods of an object are accessed by dynamic link.

Accessing Superclass Elements

Consider a class `ClassS` that declares a variable `v` that already appears in the declaration of its direct superclass `ClassG`.

Since variables are accessed by static link, it is necessary for an instance of `ClassS` to store two copies of `v`: One for the type `ClassS`, and another one for the type `ClassG`.

As a consequence, the variable `v` in `ClassS` does not really override the variable `v` of `ClassG`, but only masks it.

One can still access the variable `v` of `ClassG` from the methods of `ClassS`. This can be done either

- by explicitly modifying the type of the reference to the current object:

```
((ClassG) this).v
```

- or by using the keyword `super`:

```
super.v
```

In the case of methods, since access is by dynamic link, one cannot invoke a method `m` of `ClassG` overridden in `ClassS` by simply changing the type of the reference.

One can however use the keyword `super` for this purpose, in the same way as for variables:

```
super.m()
```

Inheritance and Visibility

Taking into account the inheritance mechanism, a class actually offers two interfaces:

- one to code that instantiates the class, and
- another to code that specializes the class into subclasses.

In Java, the visibility level `protected` makes it possible to assign different privileges to these two interfaces.

Principles:

- A class element that is declared `protected` can be accessed from a subclass, as well as from the same module as its declaration.

- In practice, if `ClassG` defines a `protected` element, and `ClassS` is a subclass of `ClassG`, then this element can be accessed from `ClassS` using the the keyword `super`.
- If needed, a `protected` method inherited from `ClassG` can be overridden in `ClassS` with a `public` one, in order to make this method visible to other classes.

Notes:

- On diagrams describing class elements, the `protected` visibility level is denoted by the symbol "#".
- A method can only be overridden by one with an equal or more permissive visibility.

Constructors and Inheritance

Consider a class `ClassS` defined as a direct subclass of `ClassG`.

Every instance of `ClassS` is also an instance of `ClassG`. Upon instantiating `ClassS`, one thus needs to invoke a constructor of `ClassG` in addition to a constructor of `ClassS`.

In Java, this constructor chaining is carried out from superclass to subclass. In other words, the constructor of `ClassG` is invoked first.

In such a case, the constructor of `ClassS` is selected on the basis of the instantiation arguments, and receives those arguments. On the other hand, by default, the constructor of `ClassG` that is invoked is the one that does not take parameters.

One can however modify this default behavior, by explicitly invoking a particular constructor of the superclass from within a constructor of the subclass. This is done thanks to a particular usage of the keyword `super`:

```
class ClassG
{
  ClassG(int x)
  {
    ...
  }
}

class ClassS extends ClassG
{
  ClassS(int x)
  {
    super(x);
    ...
  }
}
```

This mechanism can only be used if the invocation of the superclass constructor appears as the first instruction of a subclass constructor.

Multiple Inheritance in Java

Recall that in Java, multiple inheritance is forbidden between classes. The language however admits a restricted form of multiple inheritance, corresponding to a unique *specialization* associated with multiple *specification*.

In specific Java terminology, an interface is a collection of public method declarations (without implementation). An interface can also define public constants.

An interface is declared as follows (such a definition can appear at the same place in programs as a class definition):

```
interface InterfaceName
{
   returnType methodName(type param, ...);
   returnType methodName(type param, ...);

   typeConst constName = constValue;
   ...
}
```

The hierarchical place of a class with respect to a set of interfaces is defined as follows:

```
class ClassName implements Interface1, Interface2, ...
{
   ...
   ...
}
```

Such a declaration makes it mandatory for the class to implement the methods declared in *Interface1*, *Interface2*, ... The constants defined in these interfaces can also be used inside the class definition.

Notes:

- It is possible to define empty interfaces. The fact that a class implements such an interface represents a commitment of this class to respect some structural or behavioral properties expressed by the interface.

- Like classes, interfaces are linked by a hierarchical relation that supports inheritance.

  Example and notation:

```
interface InterfaceName extends Interface1, Interface2, ...
{
   ...
   ...
}
```

- The type of a reference variable may be specified using an interface name. Such variables can reference instances of any class that implements the interface. They can only be used for sending messages that correspond to the methods that belong to the interface.

Chapter 6

Exceptions and Packages

Exceptions

The execution of a program can run into various kinds of unexpected situations: arithmetic overflow, insufficient memory for instantiating objects, input/output error, . . .

In such a case, the method that experiences the problem cannot generally continue its execution. There are several possible approaches to handling the exception:

- returning an error code to the invoking method,
- returning a default value, and/or
- performing some finalization operations, e.g., closing files, freeing resources, . . .

In some programming languages such as C, exception handling needlessly complicates the structure of code.

Example: Opening a file and dynamically allocating memory in C:

```
...

f = fopen("filename", "r");
if (!f)
  return ERROR;

m1 = malloc(10);
if (!m1)
  {
    fclose(f);
    return ERROR;
  }

m2 = malloc(20);
if (!m2)
  {
    free(m1);
    fclose(f);
    return ERROR;
  }

...
```

The main problem is that the instructions that are responsible for handling exceptions are interleaved with the nominal sequence of operations.

Exceptions in Object-Oriented Programming

Most object-oriented languages offer convenient mechanisms for handling exceptions.

In such languages, an *exception* is a signal that indicates the occurrence of an unexpected situation. This signal contains data that describes the nature of the error.

An exception takes the form of an object that is instantiated when the unexpected situation occurs. Control is then transferred either

- to the invoking method, or
- to a dedicated exception handler.

In both cases, a reference to the exception object is provided in order to be able to identify the cause of the problem.

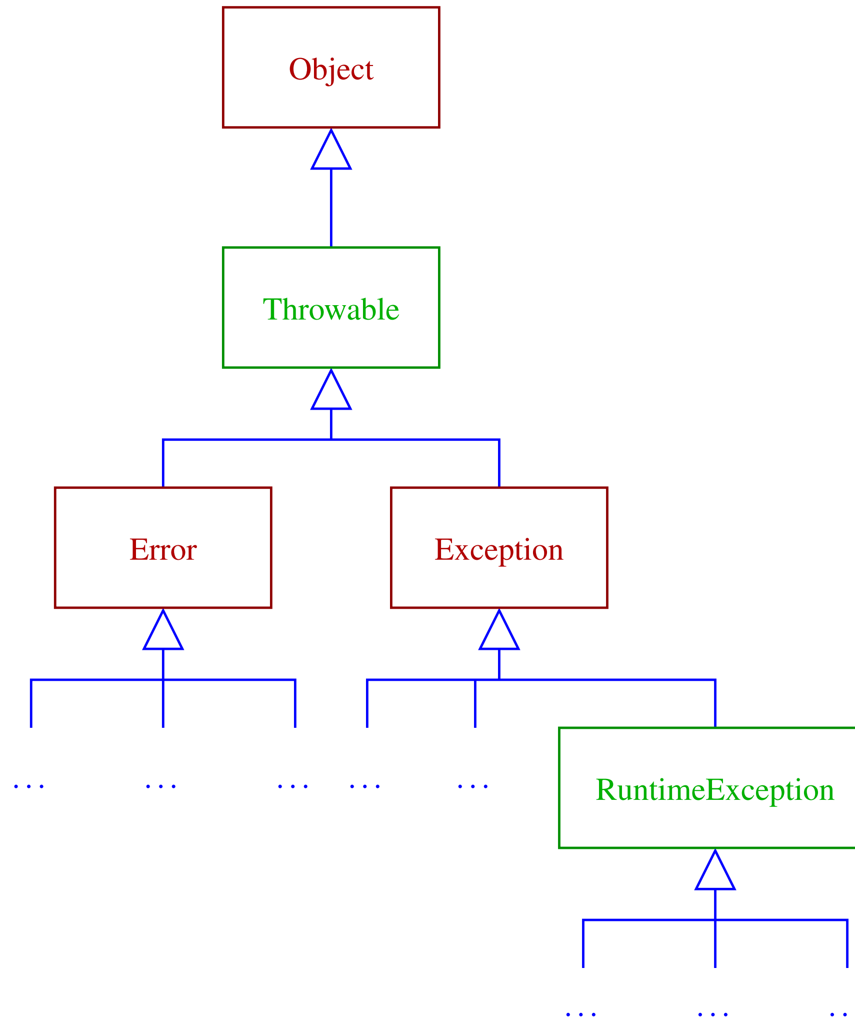
Exceptions in Java

In Java, there are two categories of exceptions:

- *Runtime Exceptions*: Such exceptions can be triggered essentially everywhere in programs: insufficient memory, arithmetic exceptions, attempt to follow an empty reference, array index out of bounds, . . .
- *Checked Exceptions*: They correspond to exceptions that only occur while performing specific operations: input/output errors, invalid data format during a conversion, . . .

In the class hierarchy, all exceptions classes are direct or indirect subclasses of `Throwable`. Runtime exceptions correspond to subclasses of either `Error` or `RuntimeException`. The other subclasses of `Throwable` are the checked exceptions.

Overview of the exceptions hierarchy:



Handling Exceptions

The following Java instruction makes it possible to react to exceptions:

```
try
{
    ...
}
catch (ExceptionClass1 e1)
{
    ...
}
catch (ExceptionClass2 e2)
{
    ...
}
...
finally
{
    ...
}
```

Principles:

- The block that follows the `try` keyword contains instructions that are executed until an exception is triggered.
- If an exception occurs, and this exception belongs to one of the classes *ExceptionClass1*, *ExceptionClass2*, . . . , then the block that follows the corresponding `catch` clause is executed. The parameter (*e1*, *e2*, . . .) of this clause then contains a reference to the object that represents the exception.

  Note: In case of multiple relevant `catch` clauses, only the first one is run.
- The block that follows the keyword `finally` is always executed, after the other instructions (even if the `try` block contains a control break instruction).

  Note: `catch` and `finally` clauses can be omitted.

- In recent versions of Java, it is possible to handle multiple exception types with a common `catch` clause.

  Syntax:

```
try
{
  ...
}
catch (ExceptionClass1 | ExceptionClass2 | ...| ExceptionClassk e)
{
  ...
}
```

Delegating Exceptions

If an exception triggered during the execution of a method is not caught by a `catch` clause, then this method is aborted, and the exception is transmitted to the invoking method.

This *delegation* of exceptions proceeds from invoked to invoker method, until reaching an appropriate `catch` clause. If no such clause can be found, then the exception is handled by the virtual machine, which then reports an error to the user.

There is however a rule to be respected: *Every method declaration must define all the checked exceptions that can be triggered without being handled in this method.*

Method declarations then take the following form:

```
returnType methodName(type1 param1, type2 param2, ...)
    throws ExceptionClass1, ExceptionClass2, ...
```

Notes:

- Constructor declarations must also indicate which checked exceptions can be triggered during their execution.
- Since all exceptions are subclasses of `Throwable`, the following code declares a method in which any exception can potentially be triggered without being handled:

```
returnType methodName(type1 param1, ...) throws Throwable
```

Triggering Exceptions

A method can explicitly trigger an exception with the statement `throw`, after having instantiated the appropriate exception object:

```
throw new ExceptionClass("message");
```

The exception classes can be defined as follows:

```
class ExceptionClass extends Exception
{
  public ExceptionClass() { super(); }
  public ExceptionClass(String s) { super(s); }
}
```

Example : Statistics application

```
import java.io.*;

public class Statistics
{
  private int n = 0;
  private int total = 0;
  private boolean isFinished = false;
  private FileInputStream   fis = null;
  private InputStreamReader isr = null;
  private BufferedReader    br  = null;

  public Statistics(String fileName) throws StatFileError
  {
    try
      {
        fis = new FileInputStream(fileName);
        isr = new InputStreamReader(fis);
        br  = new BufferedReader(isr);
      }
    catch (IOException e)
      {
        this.close();
        throw new StatFileError("Opening file");
      }
  }

  public void next() throws StatFileError
  {
    if (isFinished())
      throw new StatFileError("Reading past end of file");
```

```
  try
    {
      total += Integer.parseInt(br.readLine());
      n++;
    }
  catch (Exception e)
    {
      throw new StatFileError("Invalid value");
    }
}

public boolean isFinished() throws StatFileError
{
  try
    {
      return !br.ready();
    }
  catch (IOException e)
    {
      throw new StatFileError("File access");
    }
}

public double mean() throws StatComputationError
{
  if (n == 0)
    throw new StatComputationError("Missing values");

  return (double) total / (double) n;
}
```

```
  public void close()
  {
    try
      {
        if (br != null)  br.close();
        if (isr != null) isr.close();
        if (fis != null) fis.close();
      }
    catch (IOException e) { }
    finally
      {
        br  = null;
        isr = null;
        fis = null;
      }
  }
}

class StatFileError extends Exception
{
  public StatFileError()          { super(); }
  public StatFileError(String s)  { super(s); }
}

class StatComputationError extends Exception
{
  public StatComputationError()          { super(); }
  public StatComputationError(String s)  { super(s); }
}
```

Test program for this class:

```
import java.io.*;

public class TestStat
{
  public static void main(String[] args) throws Throwable
  {
    if (args.length != 1)
      {
        System.out.println("usage: java TestStat <file>");
        return;
      }

    Statistics s = new Statistics(args[0]);

    do
      {
        s.next();
      }
    while (!s.isFinished());

    System.out.println("Mean : " + s.mean());

    s.close();
  }
}
```

Structuring Java Applications

An object-oriented program is essentially a collection of class definitions. Programs containing a large number of classes need to be structured in order to keep their source code manageable.

In Java, the set of classes that compose a program is structured in the following way:

- The program is partitioned into several packages, which are groups of classes sharing common functional goals (for instance, user interface, computation module, file manager, . . .). Packages should remain as loosely coupled as possible from each other.
- Each package is defined by some number of source files. It is good practice to define only one class, or a small number of closely interrelated classes, in each source file.

Packages

Each package is identified by its name, which must be unique among the Java developers community.

This is achieved by structuring the package names. Such names are composed of a sequence of identifiers, from the most general to the most specific one, separated by dots ("`.`"). Unique package names can be obtained by following this convention:

- One starts with a prefix that identifies the company or institution that is responsible for the development of the package. This prefix can take the form of the company name, or its internet domain name read from right to left.
- Optionally, one adds to this prefix the name of the person or team that develops the package.

- One then appends a suffix that defines the package inside the project.

Examples:

- `be.uliege.boigelot.courses.oop.ex.eightqueens`
- `com.google.api.services.analytics`

Notes: Some prefixes such as `java` and `javax` are reserved for packages that belong to the standard library, in particular:

`java.applet`	Applets
`java.awt`	Graphical user interface
`java.awt.event`	Graphical events
`java.awt.image`	Image processing
`java.io`	Input/output operations
`java.lang`	Fundamental classes
`java.math`	Arithmetic
`java.net`	Network operations
`java.text`	Internationalization of text
`java.util`	Miscellaneous utilities
`java.util.zip`	Data compression
⋮	⋮

Defining Packages

The correspondence between source files and packages is specified as follows:

- Source files must be placed in a directory structure that matches the package identifiers. The root directory must be known to the compiler or development environment.

  Example (UNIX): Files belonging to the package `uliege.admin.students` are located in the directory `uliege/admin/students/`.

- The first statement of a source file mentions the package to which it belongs:

```
package packageName;
```

Note: If a source file does not contain a `package` declaration, then it is considered to belong to the default package, the name of which is empty.

Accessing Package Elements

A class can make use of classes belonging to other packages only if these are declared `public`.

In such a case, class names must be prefixed by their package name in order to resolve any potential ambiguities.

Example: Instantiation of the class `BigDecimal` belonging to the package `java.math`:

```
java.math.BigDecimal v = new java.math.BigDecimal("12");
```

This mechanism is not very convenient. It is however possible to *import* classes or entire packages in a source file, in order to make their name known to the compiler.

The import instruction admits two forms:

```
import className;
import packageName.*;
```

Example: `import java.io.*;` makes it possible to refer to the input/output classes of the standard library without having to prefix their name with "`java.io.`".

Note: The classes that belong to the package `java.lang` are implicitly imported in all source files.

Packages and Visibility

The visibility of class elements (variables, methods, and constructors) can take into account the structure of the application.

- An element defined `public` can be accessed from every class, whatever its package.
- An element defined `protected` can be accessed from every class in the same package, as well as from all the subclasses of the class in which it is defined (cf. Ch. 5).

In practice, the visibility markers are set so as to decouple as much as possible the packages that compose an application.

Note: In Java, one cannot limit the visibility of a class or a class element to a single source file.

Chapter 7

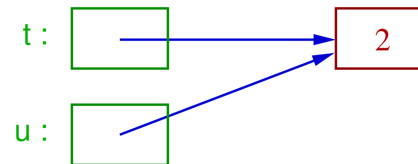
Cloning, Equivalence Checking, and Serialization

Copying Objects

In many object-oriented languages, objects are accessed by means of reference variables. Copying the value of such variables does not replicate the resources allocated to the referenced objects.

Example:

```
...

int[] t = new int[1];
int[] u = t;

u[0] = 1;
t[0] = 2;

System.out.println(u[0]);

...
```



This mechanism can thus be used for copying objects only if their state cannot be modified (*immutable* objects).

Cloning

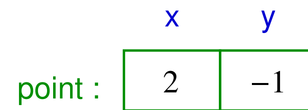
In order to duplicate an object that dynamically changes its state, it is not sufficient to copy a reference to this object. The following operations need to be performed:

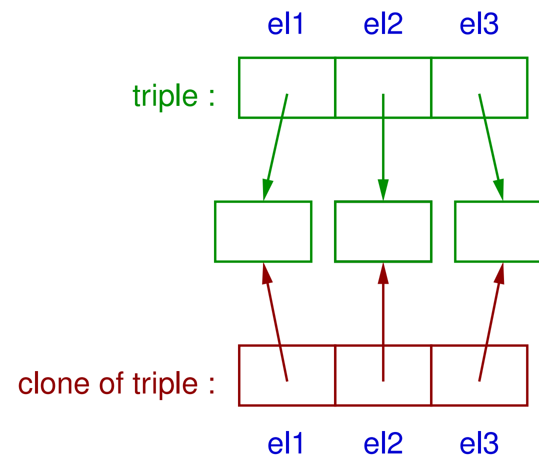
1. Instantiating a new object from the same class as the object to be copied.
2. Assigning to the instance variables of the new object the same values as those of the original object.

This procedure is known as *cloning* the object.

Note: In this simple form, the cloning operation is *shallow*: If some instance variables reference other objects, then these auxiliary objects are themselves not cloned.

Examples:





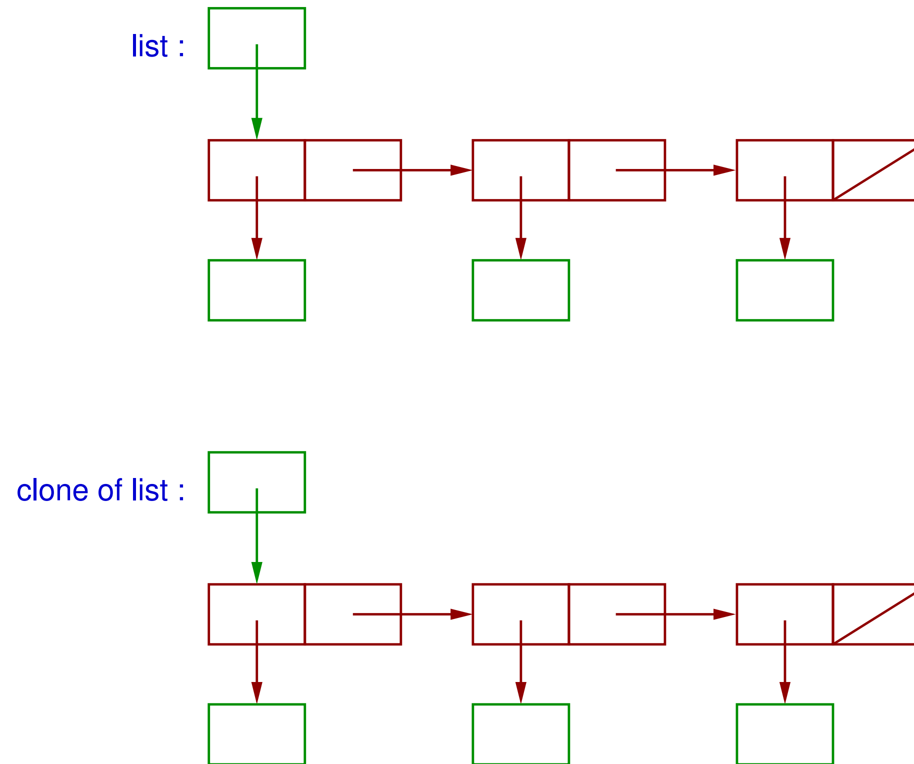
Deep Cloning

For applications in which a main object maintains references to auxiliary objects that are also considered to be part of its state, one needs to carry out *deep cloning*.

This operation amounts to

1. deep cloning recursively the objects referenced by the main object,
2. shallow cloning the main object,
3. assigning to each reference instance variable of the main object clone a reference to the clone of the corresponding auxiliary object.

Example:



Notes:

- The cloning operation can be restricted to only a subset of the reference variables of an object.
- Generally, an object is responsible for cloning correctly its auxiliary objects.

Cloning in Java

In Java, cloning an object is performed by sending the message `clone()` to this object. This operation returns a reference of type `Object` to the newly created clone.

In order to be clonable, an object must implement a method `clone()`. The implementation of this method follows these principles:

- Shallow cloning of the object is carried out by invoking the method `clone()` inherited from `Object`. This invocation can either be direct (for direct subclasses of `Object`), or indirect (through the method `clone()` of the superclass).

- Deep cloning is performed by sending `clone()` messages to the appropriate auxiliary objects.

Notes:

- The method `clone()` is defined with a `protected` visibility in `Object`. It is thus generally impossible to send a message `clone()` to an object whose class does not override this method.
- The method `clone()` implemented in `Object` is only able to clone instances of classes that implement the interface `Cloneable`. Otherwise, it triggers a `CloneNotSupportedException`.
- The shallow cloning operation performed by the method `clone()` of `Object` is not equivalent to instantiating an object of the same class using `new`. Indeed, `clone()`
  - dynamically assigns the instantiation class of the clone to the same class as the cloned object, and
  - creates an object without executing its constructors.

Example: Stack of References

```
public class Stack implements Cloneable
{
  private int nbElements;
  private Object[] contents;

  ...

  public Object clone()
  {
    Stack copy;

    try
      {
        copy = (Stack) super.clone();
        copy.contents = (Object[]) this.contents.clone();
      }
    catch (CloneNotSupportedException e)
      {
        throw new InternalError("unable to clone");
      }

    return (Object) copy;
  }
}
```

Note: In this solution, the objects referenced by the stack are not cloned!

Equivalence Checking

The equality check operator ("==") applied to a pair of references returns true only when these references point to the same object (or if they are both equal to `null`).

Example:

```
...

Integer u = new Integer(3);
Integer v = new Integer(3);

System.out.println(u == v ? "equal" : "different");

...
```

In such a situation, a better approach would be to check equivalence by comparing the state of objects.

This can be done by providing each object with a method `equals(`u`)`, that is able to compare this object against the object referenced by the parameter u, returning a Boolean result.

Properties of the Equivalence Relation

The method `equals` must be implemented in such a way that it satisfies the following properties:

- Reflexivity: For all *x*, `x.equals(x)` returns `true`.
- Symmetry: For all *x* and *y*, `x.equals(y)` and `y.equals(x)` return the same value.
- Transitivity: For all *x*, *y* and *z*, if `x.equals(y)` and `y.equals(z)` both return `true`, then `x.equals(z)` also returns `true`.
- Consistency: Provided that the objects referenced by *x* and *y* do not change their state, multiple evaluations of `x.equals(y)` always return the same value.
- Comparison against null: For all *x*, `x.equals(null)` returns `false`.

Equivalence Checking in Java

The class `Object` defines a method `equals`. This method can either be simply inherited by other classes, or overridden. By default, this method implements the same comparison as the equality check operator: It returns `true` if and only if its argument references the same object as the one executing the method.

Whenever a class overrides the method `equals`, it must also necessarily override the method `hashCode()` inherited from `Object`, which computes an integer *hash value* for the current object. This method can be freely implemented, provided that the following requirement is met:

> *For all x and y such that* x`.equals(`y`)` *returns* `true`*, the values returned by* x`.hashCode()` *and* y`.hashCode()` *must be equal.*

In practice, one should try to distribute as evenly as possible the hash values among the range of integer values.

Example:

```
public class Stack
{
  private int nbElements;
  private Object[] contents;

  ...

  public boolean equals(Object u)
  {
    if (!(u instanceof Stack))
      return false;

    Stack s = (Stack) u;

    if (s.nbElements != this.nbElements)
      return false;

    for (int i = 0; i < this.nbElements; i++)
      if (!this.contents[i].equals(s.contents[i]))
        return false;

    return true;
  }

  public int hashCode()
  {
    int code = this.nbElements;

    for (int i = 0; i < this.nbElements; i++)
      code += this.contents[i].hashCode();

    return code;
  }
}
```

Serialization

One sometimes needs to collect all the information that characterizes a given object, in other words:

- the class from which it has been instantiated,
- its state, that is, the value of its instance variables, and
- similar information about the auxiliary objects that it references.

This operation is called *serialization* of the object. In particular, serialization makes it possible to

- persistently store an object in order to reuse it in a future execution of the program, and
- transmit an object from a program to another.

Serialization in Java

In Java, objects can be serialized by invoking the method `writeObject` of `ObjectOutputStream`, and reconstructed using the method `readObject` of `ObjectInputStream`.

Principles:

- Only instances of classes that implement the interface `Serializable` can be serialized.
- The values of class variables as well as of those declared `transient` are not taken into account by the serialization process.
- The serialization operation automatically considers all objects that are directly or indirectly referenced by the object being serialized. Cycles of references are correctly detected and handled.

- When an object is deserialized, all the objects that it directly or indirectly references are deserialized as well, and their mutual references are reconstructed.
- Since a serialized object can be reconstructed by a different program, it is important to check that the class from which the object has originally been instantiated has the same definition. This is done by including in the serialization information a serial number for the class, that allows to detect inconsistencies.

  This serial number is represented by the class variable

  ```
  private static final long serialVersionUID = value;
  ```

  which is defined either automatically by the programming environment when a class implements the interface `Serializable`, or explicitly by the programmer.
- Mechanisms are available for precisely controlling how serialization must be carried out.

Example:

```
import java.io.*;

public class Stack
{
  private int nbElements;
  private Object[] contents;

  ...

  public Stack(String fileName) throws IOException
  {
    this();

    FileInputStream   fis = new FileInputStream(fileName);
    ObjectInputStream ois = new ObjectInputStream(fis);

    try
      {
        int n = ((Integer) ois.readObject()).intValue();

        this.nbElements = n;

        for (int i = 0; i < n; i++)
          this.contents[i] = ois.readObject();
      }
    catch (ClassNotFoundException e)
      {
        throw new IOException("Wrong stack format.");
      }

    ois.close();
    fis.close();
  }

...
```

```
...

  public void save(String fileName) throws IOException
  {
    FileOutputStream   fos = new FileOutputStream(fileName);
    ObjectOutputStream oos = new ObjectOutputStream(fos);

    oos.writeObject(Integer.valueOf(this.nbElements));

    for (int i = 0; i < this.nbElements; i++)
      oos.writeObject(this.contents[i]);

    oos.flush();
    oos.close();
    fos.close();
  }
}
```

Chapter 8

Generics

Introduction

The use of polymorphic objects sometimes leads to defining data structures that are too general.

Example: Implementation of a linked list for storing objects of arbitrary type.

```
class LinkedListElement
{
  private Object value;
  private LinkedListElement next;

  ...

  Object getValue()
  {
    return value;
  }

  LinkedListElement getNext()
  {
    return next;
  }

  ...

}

...
```

```
...

public class LinkedList
{
  private LinkedListElement first;

  ...

  public void add(Object e)
  {
    ...
  }

  public Object get(int i)
  {
    ...
  }

  ...

}
```

If this data structure is used for handling a list of integers, it becomes necessary to perform type casting:

```
LinkedList l = new LinkedList();

...

l.add(Integer.valueOf(10));
l.add(Integer.valueOf(2));

...

Integer n1 = (Integer) l.get(0);
Integer n2 = (Integer) l.get(1);

...
```

Problem: Inconsistencies between the type of the object returned by `get()` and the expected type (in the present case, `Integer`) caused by programming errors would only be detected at runtime, by an exception triggered by the type casting operation.

Question: Could such errors be detected at compile time?

Generic Classes

Principles:

- One augments a class definition with one or many type parameters, that make it possible to specify symbolically
  - the type of the variables defined in the class, as well as
  - the return type and the type of the parameters of its methods.

Illustration: The class `LinkedList` can be turned into a class `LinkedList<T>`, where *T* is a type parameter. This parameter then defines the type of

- the instance variable representing the value of a list element,
- the parameter of the `add()` method, and
- the return value of the `get()` method.

- When a generic class is instantiated, one needs to provide one or several arguments that assign a precise type to each type parameter.

  Example: Instantiating the `LinkedList<Integer>` class creates an object that has all the features defined in the `LinkedList<T>` class, where *T* represents the type `Integer`.

Advantages:

- Data structures that only differ in the type of internal or interface elements can share the same code.
- Type checking operations can be carried out at compile time.

Genericity in Java

In Java, generic classes are defined as follows:

```
[ public ] [ abstract ] [ final ] class ClassName<ptype1, ptype2, ...>
```

where *ptype1*, *ptype2*, . . . are type parameters.

Such a declaration makes is possible to use types of the form

```
ClassName<type1, type2, ...>
```

where *type1*, *type2*, . . . are argument types,

- in variable or method declarations, and
- in instantiation expressions.

Note: Argument types cannot be primitive.

Example:

```
class LinkedListElement<T>
{
  private T value;
  private LinkedListElement<T> next;

  public LinkedListElement(T value, LinkedListElement<T> next)
  {
    this.value = value;
    this.next = next;
  }

  public T getValue()
  {
    return value;
  }

  public LinkedListElement<T> getNext()
  {
    return next;
  }
}

...
```

```
public class LinkedList<T>
{
  private LinkedListElement<T> first;

  public LinkedList()
  {
    first = null;
  }

  public void add(T value)
  {
    LinkedListElement<T> e = new LinkedListElement<T>(value, first);
    first = e;
  }

  public T get(int i) throws IndexOutOfBoundsException
  {
    LinkedListElement<T> e;

    for (e = first; i > 0 && e != null; i--, e = e.getNext());

    if (e == null)
      throw new IndexOutOfBoundsException("Invalid list index");

    return e.getValue();
  }
}
```

Sample test program:

```
public class LinkedListTest
{
  public static void main(String[] args)
  {
    LinkedList<Integer> l = new LinkedList<Integer>();

    l.add(Integer.valueOf(10));
    l.add(Integer.valueOf(20));
    l.add(Integer.valueOf(30));

    Integer n1 = l.get(2);
    Integer n2 = l.get(0);

    ...
  }
}
```

Restrictions Imposed by Java

In Java, the genericity mechanism disappears after compilation: After having performed type checking, the compiler removes all information about type parameters *(type erasure)*.

As a consequence, one cannot program operations that rely on the value of type parameters at runtime. In particular, it is forbidden

- to instantiate objects from a type parameter.

  Example:

```
class C<T>
{
  public C()
  {
    new T();  // Invalid!
  }
  ...
}
```

- to create arrays whose elements are defined with a generic type.

Example:

```
...
LinkedList<Integer>[] t = new LinkedList<Integer>[10];  //Invalid!
...
```

- to define class variables or methods using a type parameter.

Example:

```
class C<T>
{
  static T x;  // Invalid!
  ...
}
```

- to use generic types with the `instanceof` operator, or in a type casting operation.

Example:

```
...
if (l instanceof LinkedList<Integer>) ...  // Invalid!
...
```

- to define generic exception types, or `catch` clauses that rely on type parameters.

  Examples:

```
class C1<T> extends Throwable  // Invalid!
{
  ...
}

class C2<T>
{
  public void m()
  {
    ...
    try
    {
      ...
    }
    catch(T e)  // Invalid!
    {
      ...
    }
  }
}
```

Generic Methods

In Java, type parameters can also be associated to method declarations (for both instance and class methods). The scope of such parameters is then limited to their corresponding method.

Syntax:

```
[ visibility ] [ attributes ] <ptype1, ptype2, ...> returnType methodName(type1 param1,
                                                                        type2 param2, ...)
```

where *ptype1*, *ptype2*, . . . , are type parameters.

A message for invoking such a method can be sent by evaluating an expression of the form

```
reference.<type1, type2, ... >methodName(expr1, expr2, ...)
```

where *type1*, *type2*, . . . , are argument types.

Example:

```
public class ListUtils
{
  public static <T> void printElement(LinkedList<T> l, int i)
  {
    System.out.println(l.get(i));
  }

  ...
}
```

Test program:

```
public class TestUtils
{
  public static void main(String[] args)
  {
    LinkedList<Integer> l = new LinkedList<Integer>();

    l.add(Integer.valueOf(10));
    l.add(Integer.valueOf(20));
    l.add(Integer.valueOf(30));

    ListUtils.<Integer>printElement(l, 2);
    ListUtils.<Integer>printElement(l, 0);
  }
}
```

Bounded Type Parameters

It is not always appropriate for a type parameter to take as a value any possible type. In some cases, it is useful to restrict argument types to the subclasses of a given class. The main advantage is that the functionalities of this class can then be exploited generically.

Java syntax:

- Declaration of a generic class:

```
[ public ] [ abstract ] [ final ] class ClassName<ptype extends Class, ...>
```

- Declaration of a generic method:

```
[ visibility ] [ attributes ] <ptype extends Class, ...> returnType methodName(type1 param1,
                                                                              type2 param2, ...)
```

In both cases, the allowed values for *ptype* are limited to *Class* and its subclasses.

Example: In the standard library, the classes `Integer`, `Long`, `Short`, `Float`, `Double`, ..., are all defined as subclasses of `Number`, which provides a method `doubleValue()` that returns the floating-point value of the represented number.

The following data structure implements a generic linked list of numbers, and is able to compute their sum:

```
public class LinkedListOfNumbers<T extends Number>
{
  private LinkedListElement<T> first;

  ...

  public double sum()
  {
    double sum = 0.0;

    for (LinkedListElement<T> e = first; e != null; e = e.getNext())
      sum += e.getValue().doubleValue();

    return sum;
  }
}
```

Notes:

- In the declaration of a bounded type parameter, `extends` can also be followed by an interface name.
- A particular syntax has been introduced for the limited form of multiple inheritance allowed by Java:

```
[ options ] class className&lt;ptype extends Class1 & Interface1 & Interface2 ..., ...>
```

```
[ options ] <ptype extends Class1 & Interface1 & Interface2 ..., ...> returnType methodName(...)
```

Generic Classes and Inheritance

In the previous example, the generic class `LinkedListOfNumbers<T>` can be used in different ways in a program:

- The class `LinkedListOfNumbers<Integer>` represents lists of integer numbers.
- The class `LinkedListOfNumbers<Double>` corresponds to lists of floating-point numbers.
- The class `LinkedListOfNumbers<Number>` allows to mix different types of numbers inside the same list.

Notes:

- Those classes are not hierarchically related! Although `Integer` and `Double` are both subclasses of `Number`, the classes `LinkedListOfNumbers<Integer>` and `LinkedListOfNumbers<Double>` are not subclasses of `LinkedListOfNumbers<Number>`.

- There exist however other mechanisms for creating such class hierarchies.
- On the other hand, if one defines the generic class `LinkedListOfNumbers<T>` as a subclass of `LinkedList<T>`, then
  - `LinkedListOfNumbers<Integer>` becomes a subclass of `LinkedList<Integer>`,
  - `LinkedListOfNumbers<Double>` becomes a subclass of `LinkedList<Double>`,
  - ...

Chapter 9

Concurrency

Introduction

The programs that we have studied up to this point were sequential: Control is always located at a unique program location, and proceeds from instruction to instruction.

For some applications, it is however more convenient to develop programs in which several fragments of code are able to run simultaneously. Such programs are said to be *concurrent*, or *parallel*.

Example: Simulation tool with a graphical interface. One executes at the same time

- a computationally intensive function (that may run for a long time), and
- a module responsible for interacting with the user (that must remain reactive).

Overview of Difficulties

Concurrent programming raises issues that are not experienced in sequential programming:

- Sharing the processor. The number of tasks that need to be performed often exceeds the number of available processors. In such a case, one has to distribute the use of processors among the concurrent tasks.
- Defining the semantics of parallelism. The problem is to characterize the possible executions of a concurrent program, in particular when its tasks need to access shared variables. The challenge is to perform this characterization independently from the hardware properties of the runtime platform (such as the memory architecture or the number of processors).

- Mutual exclusion. How can concurrent tasks be prevented from simultaneously carrying out operations that can potentially corrupt the state of objects?
- Deadlocks. A *deadlock* is a situation in which all tasks are waiting for a condition that can only be lifted by the tasks themselves, which will then never happen.
- Livelocks. Those correspond to situations in which all tasks repeatedly perform some operations waiting for a condition to become true, which will then never happen.

For fundamental reasons, it is generally impossible to detect the presence of deadlocks or livelocks in a program at compile time.

Semantics of Concurrency

The goal is to be able to characterize the possible executions of a concurrent program without making hypotheses on

- the runtime platform (which may have one or several processors), or
- the relative execution speed of tasks.

This is achieved by the *interleaving semantics*, which has been adopted by multitasking operating systems, as well as by Java virtual machines.

Principles:

- Program instructions are decomposed into some number of atomic operations, the effect of which cannot be observed in the middle of their execution. (In other words, they are indivisible.)

- The concurrent execution of several tasks always yields the same effect as the successive execution of a sequence of atomic operations.

  In such a sequence, the operations belonging to a given task must all appear, in the same order as in this task.

Example: The concurrent execution of the two tasks $t_1 = \mathsf{a}; \mathsf{b}$ and $t_2 = \mathsf{c}; \mathsf{d}$ (in which operations a, b, c and d are atomic) must always yield the same result as one of the sequences

a; b; c; d
a; c; b; d
a; c; d; b
c; d; a; b
c; a; d; b
c; a; b; d.

Concurrency in Java

The Java language offers mechanisms for managing concurrency.

Principles:

- A task corresponds to a *thread*, which represents a unique control flow inside a program.

  The information maintained by a thread includes

  - a current control point that precisely identifies the next atomic operation to be executed, and
  - a runtime stack used for keeping track of the methods that are currently active (i.e., that have been invoked and have not yet finished their execution), as well as storing the local variables and temporary values manipulated by the thread.

- Objects are stored in a central memory that can be accessed from all threads.

- Copying a value from central memory to the stack of a thread, or the other way around, is performed atomically, except for the primitive types `long` and `double`.
- Operations involving (class or instance) variables are performed on the stack of the corresponding thread, and are not always immediately reflected in central memory.

  However, for variables that are declared `volatile`, every assignment becomes effective in central memory before any other operation involving them. The value of `volatile` variables is always retrieved from central memory rather than from thread stacks.
- One difficulty is that the compiler, the virtual machine and the hardware can reorder memory operations for the sake of efficiency. It is only guaranteed that
  - the memory operations of each thread considered in isolation have a global effect that is equivalent to what is written in the code being run, and
  - operations involving a `volatile` variable are never reordered with any memory operation.

Thread States

A thread is not always busy performing operations. At any given time, it can be in one of the four following states:

- Initial: The thread has just been created, but has not yet started to execute instructions.
- Runnable: The thread is able to perform operations.

  Since there are generally more runnable threads than available processors, a thread in the runnable state is not necessarily executing instructions at a given time.
- Blocked: The thread has suspended its execution and is waiting for a specific condition to become true in order to resume its operations.
- Final: The thread has finished its execution.

Scheduling

Consider a program that contains more than one runnable thread, in a single-processor environment.

A component of the virtual machine called the *scheduler* is responsible for distributing the processor among those threads. The scheduler repeatedly performs the following operations:

1. Selecting a runnable thread.
2. Assigning the processor to this thread until either it has executed a sufficient number of instructions, or it becomes blocked.

Notes:

- The scheduler is generally non deterministic: Two executions of the same program may result in different interleavings of its instructions.

- The scheduler is *fair*: A thread can never remain indefinitely runnable without being granted the processor.

  It is however possible for threads to influence their probability of being selected by the scheduler: Each thread is characterized by an integer priority that can be chosen and modified at will. Threads with a high priority are selected more often than others.

  Note: The priority mechanism can be used for improving the reactivity of threads, but not for ensuring the correctness of a program!

- In the case of a multiprocessor environment, there is a separate scheduler for each processor.

Thread Creation (I)

The Java language offers two mechanisms for creating a thread.

The first one consists in instantiating a class defined as a subclass of `Thread`. This class must satisfy the following requirements:

- Its constructors invoke the constructor of `Thread` that takes a `String` as a parameter. The value of this parameter provides a name for the thread.
- The class implements a public method `void run()` that contains the instructions to be executed by the thread.

Instantiating such a class creates a new thread, in the initial state. Sending a message `start()` to the resulting object then makes the thread runnable. The thread becomes final as soon as the method `run()` has finished its execution.

Example:

```
class Task extends Thread
{
  public Task(int n)
  {
    super("Task " + Integer.toString(n));
  }

  public void run()
  {
    ...
  }
}

public class TestTask
{
  public static void main(String[] args)
  {
    new Task(1).start();
    new Task(2).start();
    new Task(3).start();
  }
}
```

Thread Creation (II)

The first mechanism for creating a thread imposes a major restriction: The classes that implement concurrent operations must be defined as subclasses of `Thread`, hence they cannot inherit from other arbitrary classes. This complicates, in particular, the definition of concurrent applets.

The second mechanism for thread creation does not have this restriction. This mechanism consists in:

- Defining a class that implements the interface `Runnable`.

  This interface requires the class to implement a method `void run()` that contains the instructions to be executed by the thread.

- Instantiating the class `Thread` with two arguments: a reference to an instance of the class that implements `Runnable`, and the thread name.

Example:

```
class Task implements Runnable
{
  public Task(int n)
  {
    new Thread(this, "Task " +
                     Integer.toString(n)).start();
  }

  public void run()
  {
    ...
  }
}

public class TestTask
{
  public static void main(String[] args)
  {
    new Task(1);
    new Task(2);
    new Task(3);
  }
}
```

Locks

It is common to have several threads that attempt to simultaneously modify the state of a shared object, by performing non atomic sequences of operations. This may lead to corrupting this object.

In Java, this problem can be solved thanks to the concept of *monitor*:

- Each object is equipped with a *lock* that can be acquired and released by threads.
- The execution of a block of instructions can be controlled by the monitor of an object v by using the following construct:

```
synchronized(v)
  {
     ...
  }
```

When this instruction is executed, the current thread

1. attempts to acquire the lock associated to `v`. This is only possible provided that no other thread has already acquired (and not yet released) this lock. Otherwise, the current thread becomes blocked until it succeeds in acquiring the lock,
2. executes the instructions in the block,
3. releases the lock.

- A method defined with the `synchronized` attribute has its body implicitly included in a `synchronized` instruction associated to the current object.

Notes:

- If several threads compete for acquiring a lock, only one of them will succeed. The other threads then become blocked until the lock is released. At this time, the threads compete again for acquiring the lock.

- The rules governing monitors allow a thread to acquire several times the same lock, in order to avoid situations in which a thread deadlocks itself.

  In such a case, the lock can only be acquired by another thread provided that it has been released the same number of times as it has been acquired.

- One can also define class methods with the `synchronized` attribute.

  In this case, the lock is associated to the class rather than to its instances.

- Memory operations performed prior to acquiring or releasing a lock will not be reordered with operations performed afterward by the same thread.

Example (stack data structure):

```
public class Stack
{
  protected static int max = 300;
  private int nbElements;
  private Object[] contents;

  public Stack()
  {
    nbElements = 0;
    contents = new Object[max];
  }

  public synchronized void push(Object e)
  {
    if (!isFull())
      contents[nbElements++] = e;
  }

  public synchronized Object pop()
  {
    if (!isEmpty())
      {
        Object e = contents[nbElements - 1];
        contents[--nbElements] = null;
        return e;
      }
    else
      return null;
  }

...
```

```
...

  public synchronized boolean isEmpty()
  {
    return nbElements == 0;
  }
  public synchronized boolean isFull()
  {
    return nbElements == max;
  }
}
```

Thread Synchronization

When several threads exchange data, one sometimes needs to temporarily suspend a thread until another has completed some operations (for instance, producing data to be processed by the other thread).

A first solution for implementing this suspension mechanism is to program a loop in which the thread repeatedly consults shared variables that indicate whether data is available. Such *busy waiting* is however very inefficient!

A much better solution is to rely on synchronization mechanisms implemented by the class `Object` of the standard library. This class defines the three following methods, that can be invoked only if the current thread has acquired (and not yet released) the lock associated to the object:

- `wait()`: Makes the current thread blocked, and releases the lock associated to the object (the appropriate number of times).

- `notify()`: Chooses a thread waiting for the current object (following `wait()`), and makes this thread runnable again.

  After this operation, the thread that has been unblocked attempts to acquire again the lock associated to the object (the appropriate number of times).

  Notes: In the case of several threads waiting for the same object, the one chosen by `notify()` is selected arbitrarily.

- `notifyAll()`: Performs a similar operation to `notify()`, but makes runnable again all the threads that are waiting for the current object (following `wait()`).

Notes: Threads blocked on `wait()` can sometimes become unblocked for other reasons than the effect of `notify()` or `notifyAll()` (spurious wake-ups). Their wake-up condition must thus be always explicitly checked afterwards.

Example: Communication channel of capacity one.

```
public class Channel
{
  private int value;
  private boolean available = false;

  public synchronized int getValue()
  {
    while (!available)
      {
        try { wait(); }
        catch (InterruptedException e) { }
      }

    available = false;
    notifyAll();
    return value;
  }

  public synchronized void setValue(int v)
  {
    while (available)
      {
        try { wait(); }
        catch (InterruptedException e) { }
      }

    available = true;
    value = v;
    notifyAll();
  }
}
```