ELEMENTARY PRINCIPLES OF OBJECT-ORIENTED PROGRAMMING

Gérald MASINI
Senior Researcher

L O R I A
ISA Research Group (`http://www.loria.fr/isa`)

Post: B.P. 239, 54506 Vandœuvre-lès-Nancy Cedex, France
E-mail: Gerald.Masini@loria.fr
Tel: (+33) (0)3.83.59.20.68
Fax: (+33) (0)3.83.41.30.79

INTRODUCTION

Size and complexity
of software systems
go increasing

☛ Methods and tools to design quality software

☛ Reasonable costs in time and money

☛ Ability to reuse (reusability) existing software

during the 80's, it has been estimated that less than 15% of the software produced each year in the world was quite new

BREAKDOWN OF SOFTWARE MAINTENANCE COSTS

From: B.P. Lientz and E.B. Swanson. *Software Maintenance: A User/Management Tug of War*. Data Management, pp. 26–30, 1979

Changes in user requirements	**41.8%**
Changes in data format	**17.4%**
Emergency fixes	*12.4%*
Routine debugging	*9.0%*
Hardware changes	6.2%
Documentation	**5.5%**
Efficiency improvements	4.0%
Other	3.4%

487 large software systems

- **Systems are harder to change than they should be**
  $\Longrightarrow$ lack of extendibility
- **Data format affects too many parts of the systems**
  $\Longrightarrow$ access data by external properties (*abstract data types*) rather than physical implementation
- Low percentage of documentation: **It is either done during the system development, or not done**
  $\Longrightarrow$ documentation must be embedded within programs, with special tools to extract it
- *Emergency bug fixes $>$ routine debugging, because they are performed under heavy pressure and tend to introduce new errors*

SOFTWARE ENGINEERING REQUIREMENTS

REUSABILITY

- **correctness**
  programs exactly perform their tasks, as defined by their specification

- **robustness**
  programs function (i.e. do not abort), even in abnormal conditions

- **extendibility**
  programs can be easily adapted to changes of specification

- **compatibility** (modular composability)
  programs can be easily combined with others

EASY SOFTWARE MAINTENANCE

- **modular understandability**
  modules can be separately understood by a human reader

- **modular continuity**
  a small change in specification results in a change of *few* modules

- **modular protection**
  the effect of an abnormal condition at run-time remains confined in the module where it occurs

ARTIFICIAL INTELLIGENCE REQUIREMENTS

SOFTWARE ENGINEERING REQUIREMENTS, TOO

REAL-WORLD KNOWLEDGE MODELLING

- **large heterogeneous knowledge bases**
- **multi-paradigm programming**
  logical programming, functional programming, access-oriented programming, etc.

PERSISTENT KNOWLEDGE

- **concurrent access** to data
- **secure access**
  some data must remain confidential
- **backup/recovery system**

DISTRIBUTED SYSTEMS

- **autonomous processes**
- **parallelism**
- **asynchronous communication**

S Y N T H E T I C A L   S U M M A R Y

DATA ABSTRACTION

- get free from physical structure of data

  $\Longrightarrow$ design in terms of “concepts”

  $\Longrightarrow$ easily understand high-level source code

MODULARITY

- **libraries**

  $\Longrightarrow$ easy and rapid prototyping

SOPHISTICATED PROGRAMMING ENVIRONMENTS

- user-friendly
- self-documented
- extendable and customizable

SOFTWARE DESIGN METHODS

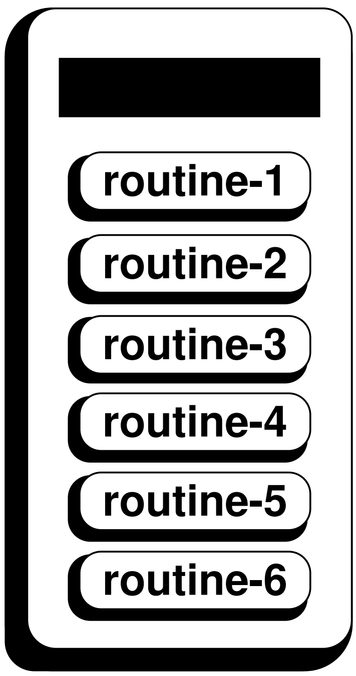
- **PROCESS-ORIENTED**

  algoritms + global data = programs

- **DATA-ORIENTED**

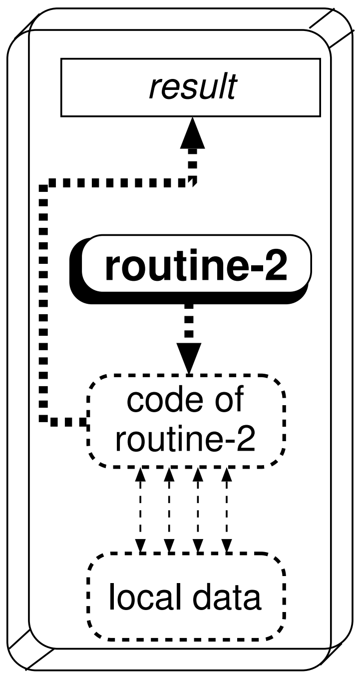
  routines + local data = objects

encapsulation



object

information hiding



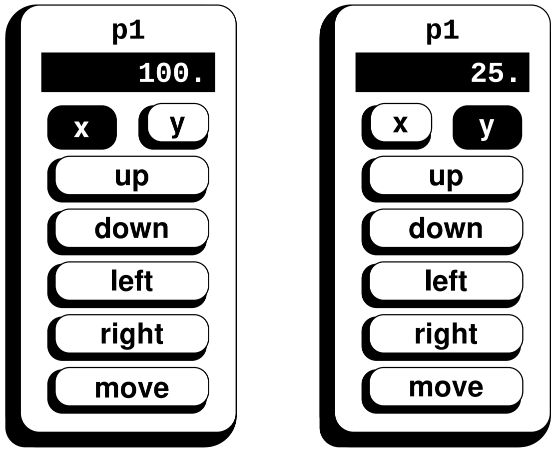
- An object is known by its **interface**
  i.e. the set of available operations (buttons)

- Users communicate with objects by **sending messages** (pushing buttons)

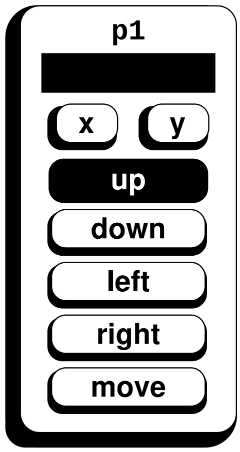
P E R F O R M   A N   O P E R A T I O N

p1 is an object representing a point

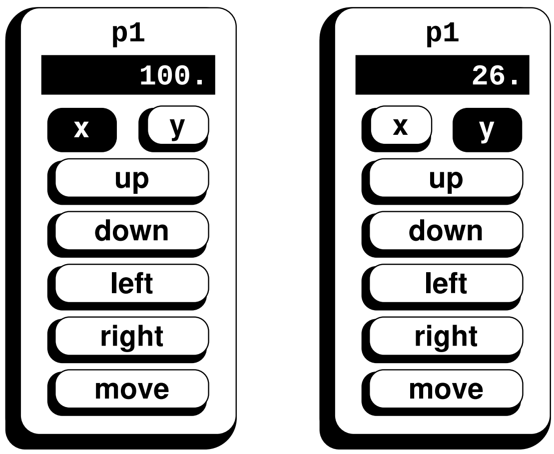
read data (x and y coordinates)



change data (move up one unit)



read data



O B J E C T S

AN OBJECT IS DEFINED BY ITS BEHAVIOUR, INDEPENDENTLY FROM ITS PHYSICAL STRUCTURE

All previously requirements are met:

- **modularity**

  objects = (implementations of) modules

- **data abstraction**
  - "direct modelling" of entities to be represented (design in terms of concepts)
  - different objects may supply same-named operations

- **reusability**

  thanks to interfaces

- **understandability**

  implementation details remains hidden

- **compatibility**

  thanks to libraries

- **extendibility**

- **etc.**

F A M I L I E S   O F   O B J E C T S

C L A S S E S

A class is the implementation of a(n abstract data) type

☛ especially used for software engineering
software design, construction and maintenance

- A class defines a pattern to create objects

- Encapsulation hides the details of implementation
  The behaviour of objects is described by the **interface** of their class

- **Inheritance** allows the reuse of existing classes to define new classes

- **Dynamic binding** and **overloading** allow the specification of operations independently from the implementation

LANGUAGES WITH STATIC TYPING

inspired by Simula (1966) and Algol (1960)

C++ (1983), Eiffel (1986), Ada95 (1995), Java (1995)...

- S.B. Lippman and J. Lajoie. *C++ Primer, Third Edition*. Addison-Wesley, Reading (MA), 1998
- B. Meyer. *Object-Oriented Software Construction, Second Edition*. The Object-Oriented Series, Prentice Hall, Englewood Cliffs (NJ), USA, 1997
- J. Barnes. *Programming in Ada 95*. Addison-Wesley, Reading (MA), 1996
- D. Flanagan. *Java in a Nutshell, Second Edition*. O'Reilly & Associates, Sebastopol (CA), USA, 1997

LANGUAGES WITH DYNAMIC TYPING

inspired by Smalltalk (1972)

Smalltalk-80 (1980), Objective-C (1983)...

- W. LaLonde. *Discovering Smalltalk*. Addison-Wesley, Reading (MA), USA, 1994
- *Object-Oriented Programming and the Objective-C Language*. Next Computer Inc., Addison-Wesley, Reading (MA), USA, 1993

lineage of LISP (often used for AI)

Flavors (1980), CLOS (Common Lisp Object System, 1984)...

- S. Slade. *Object-Oriented Common Lisp*. Prentice Hall, Englewood Cliffs (NJ), USA, 1997

FAMILIES OF OBJECTS

FRAMES and KNOWLEDGE REPRESENTATION

The construction of an intelligent system requires a language for representing the knowledge, and tools for managing the knowledge

The language and tools must be designed to deal with dynamic knowledge, as human knowledge is dynamic: It may be incomplete at a given moment and be completed or denied later

A frame is a typical representation (a prototype) of a concept

- Three levels:
  - frame ➡ a concept
  - slot ➡ an attribute of the concept
  - facet ➡ a property of an attribute
- Access-oriented programming (like in a data base)

  Reflexes are routines triggered by read and write accesses
- Concepts are hierarchically organized by inheritance
- Specific facilities for IA:
  Spatial (object aggregation) and temporal relationships, viewpoints, etc.

(PURE) FRAME-BASED LANGUAGES

only historical interest

KRL (1977), FRL (1977), UNITS (1979), etc.

OBJECT-CENTRED KNOWLEDGE-BASED REPRESENTATION LANGUAGES

Large sophisticated commercial systems:
LOOPS (1981), KEE, ART...
Academic systems:
SHIRKA/TROEPS (1985), YAFOOL/Y3 (1984)...
General-purpose languages:
CLOS (1984)...

- Frame+class-based model
- Environment providing special tools to manage knowledge: pattern-matching, filtering, classification, etc.
- Different programming paradigms: functions, rules, etc.

- G. Masini, A. Napoli, D. Colnet, D. Léonard, and K. Tombre. *Object-Oriented Languages*. The APIC Series 34, Academic Press, London, 1991

  P.H. Winston. *Artificial Intelligence, Third Edition*. Addison-Wesley, Reading (MA), USA, 1992

  R. Ducournau, J. Euzénat, G. Masini, and A. Napoli, editors. *Langages et modèles à objets*. Collection Didactique, N° 19, INRIA, 1998

F A M I L I E S O F O B J E C T S

A C T O R S

Actors are autonomous and active entities

☛ concurrent programming and distributed systems

- An actor knows **acquaintances** (= data)
- Its behaviour is decribed by a **script** (= methods)
  The script filters the messages (pattern-matching) and activates the appropriate part of the behaviour
- Communications may be asynchronous
- An actor is produced by duplicating an other actor
- A message which cannot be processed is **delegated** to a **proxy**
  delegation is equivalent to inheritance

An actor is similar to a machine working in the way specified by its script

THE ACTOR MODEL

Developed at MIT by Carl E. Hewitt (1973–77) for a prototype multi-processor machine called *The Apiary*

Actor semantics was formalized by Gul Agha (MIT, 1986)

HISTORICAL LANGUAGES

PLASMA (1977): first language based on the actor model
☛ *Patrick Sallé at IRIT (Toulouse, France) since 1980*

Incorporation of parallelism:
  Act 1 (Henry Lieberman, MIT, 1982)
  Act 2 (Daniel G. Theriault, MIT, 1983)

ACADEMIC LANGUAGES

ABCL/1 (A. Yonezawa, Tokyo Institute of Technology, 1986)
extension of Common Lisp
`ftp://camille.is.s.u-tokyo.ac.jp/pub/abcl1/`

ActTalk (Jean-Pierre Briot, LAFORIA, 1989)
Smalltalk + actors
"Actalk is a testbed for modeling, classifying and experimenting with object-oriented concurrent programming languages."

- Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge (MA), USA, 1986
- R. Guerraoui, O. Nierstrasz, and M. Riveill, editors. *Object-Based Distributed Programming*. Lecture Notes in Computer Science 791, Springer-Verlag, Berlin, 1994
- J.-P. Briot, J.-M. Geib, and A. Yonezawa, editors. *Object-Based Parallel and Distributed Computation*. Lecture Notes in Computer Science 1107, Springer-Verlag, Berlin, 1996

C L A S S E S

PART I

DYNAMIC/WEAK TYPING

Smalltalk-like world

The correctness of typing is checked at run-time

1.1 DEFINITION OF A CLASS

A class **describes** a set of objects:

- state of the objects ➡ fields (data)
- behaviour of the objects ➡ methods (procedures or functions)

```
// C++/Java-like syntax

class Article {
  // FIELDS:
    code;
    description;
    basePrice;
    quantity;
  // METHODS:
  // Access methods
    price()
      { return(1.2 * basePrice); }
    transportCost()
      { return(0.05 * basePrice); }
  // Transformation methods
    remove(q)
      { quantity := quantity - q; }
    add(q)
      { quantity := quantity + q; }
};
```

INFORMATION HIDING

```
class Article {
  private:
    _code;
    _description;
    _basePrice;
    _quantity;
  public:
    // Access methods
    code() { return(_code); }
    description() { return(_description); }
    basePrice() { return(_basePrice); }
    quantity() { return(_quantity); }
    price() { return(1.2 * _basePrice); }
    transportCost() { return(0.05 * _basePrice); }
    // Transformation methods
    setCode(c) { _code) := c; }
    setDescription(d) { _description := d; }
    setBasePrice(b) { _basePrice := b; }
    setQuantity(q) { _quantity := q; }
    remove(q) { _quantity := _quantity - q; }
    add(q) { _quantity := _quantity + q; }
};
```

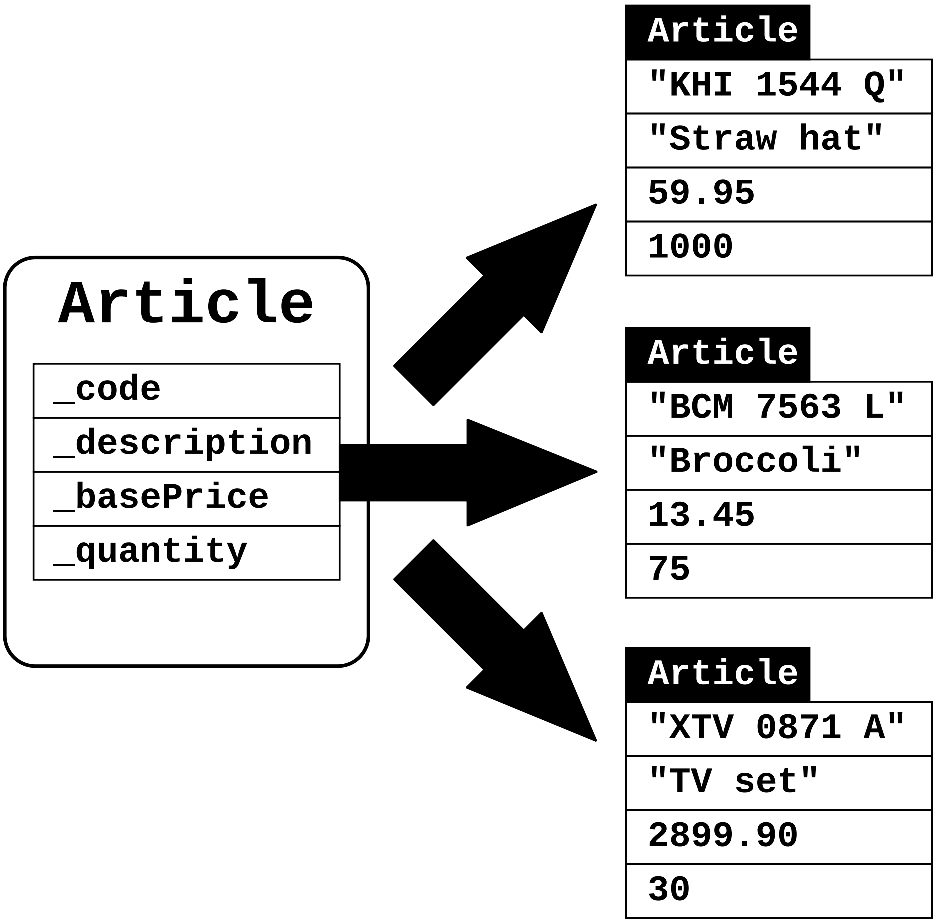
- The **private** part is hidden to “external” users, called **clients**
- The **public** part defines the user’s guide of the objects, i.e. operations clients may perform on the objects

  It should not appear any data in this part

THE INTERFACE

```
class Article

// ACCESS
  // Returns article code
    code()

  // Returns article description
    description()

  // Returns article base price (not including taxes)
    basePrice()

  // Returns the quantity available in stock
    quantity()

  // Returns article price (including taxes)
    price()

  // Returns article transport cost
    transportCost()

// TRANSFORMATION
  // Sets article code to ‘c’
    setCode(c)

  // Sets article description to ’d’
    setDescription(d)

  // Sets article base price to ‘b’
    setBasePrice(b)

  // Sets quantity available in stock to ‘q’
    setQuantity(q)

  // Withdraw ‘q’ items from the stock
    remove(q)

  // Add ‘q’ items in the stock
    add(q)
```

1.2 CREATION OF AN OBJECT



- An object is **created** by duplicating the pattern defined by the data specified by the class
- An object “knows” its class: There is an instantiaton relationship between the object and its class

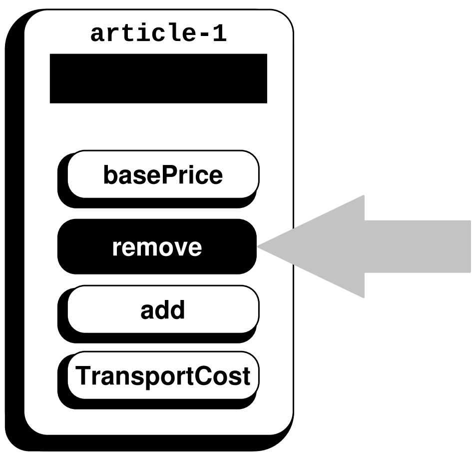
1.3 MESSAGE SENDING

- A method is performed using a request called a message

- A message includes:
  - **a target**: the object to which the message is sent
  - **a selector**: the name of the method to be performed

    ☛ the selector must be included in the interface of the class of the object
  - **parameters**: the eventual effective parameters of the method

  Example: `send(article-1, "remove", 2);`



- *remote procedure call* would be a more appropriate terminology, as message sending is synchronous

field	instance variable	*Smalltalk*
	data member	*C++*
	attribute	*Eiffel*
	slot	*CLOS*
method	method	*Smalltalk, CLOS*
	function member	*C++*
	routine	*Eiffel*
object	instance	*(Smalltalk)*
create	instantiate	*Smalltalk, CLOS*
	make	*Eiffel*

message sending routine / function / procedure call (or invocation)

selector routine / function / procedure name

OBJECT TO WHICH A MESSAGE IS SENT

```
// In the message
send(obj, "netPrice'');
```

target	receiver	*Smalltalk*
	receiving object	

```
// In the code of the corresponding method
netPrice() {
  p := send(self, "basePrice");
  t := send(self, "transportCost");
  return(p + t);
}
```

current object	`self`	*Smalltalk*
	`Current`	*Eiffel*
	`this`	*C++, Java*

1.4 SINGLE INHERITANCE

- A class factorizes knowledge (data + methods)

- An object set may be defined relatively to an other set, by **specializing** the knowledge of the latter set:
  - by adding new data or methods (class `Clothing`)
  - by **redefining**, or shadowing, existing data or methods (class `LuxuryArticle`)

- The specialized class is called the **superclass**, or the **base class**

  The specializing class is called the **subclass**, or the **derived class**

  The subclass **inherits** the data and methods of its superclass

- The inheritance relationship hierarchically organizes classes into an **inheritance tree/graph**

```
class Article {
  private:
    _code;
    _description;
    _basePrice;
    _quantity;
  public:
    code() { return(_code); }
    description() { return(_description); }
    basePrice() { return(_basePrice); }
    quantity() { return(_quantity); }
    price() { return(1.2 * _basePrice); }
    transportCost() { return(0.05 * _basePrice); }
    setCode(c) { _code) := c; }
    setDescription(d) { _description := d; }
    setBasePrice(b) { _basePrice := b; }
    setQuantity(q) { _quantity := q; }
    remove(q) { _quantity := _quantity - q; }
    add(q) { _quantity := _quantity + q; }
};
```

```
// CLASS Clothing INHERITS FROM CLASS Article

class Clothing : Article {
  private:
    // Extra data
    _size;
    _colour;
  public:
    // Extra methods
    size() { return(_size); }
    colour() { return(_colour); }
    setSize(s) { _size := s; }
    setColour(c) { _colour := c; }
};
```

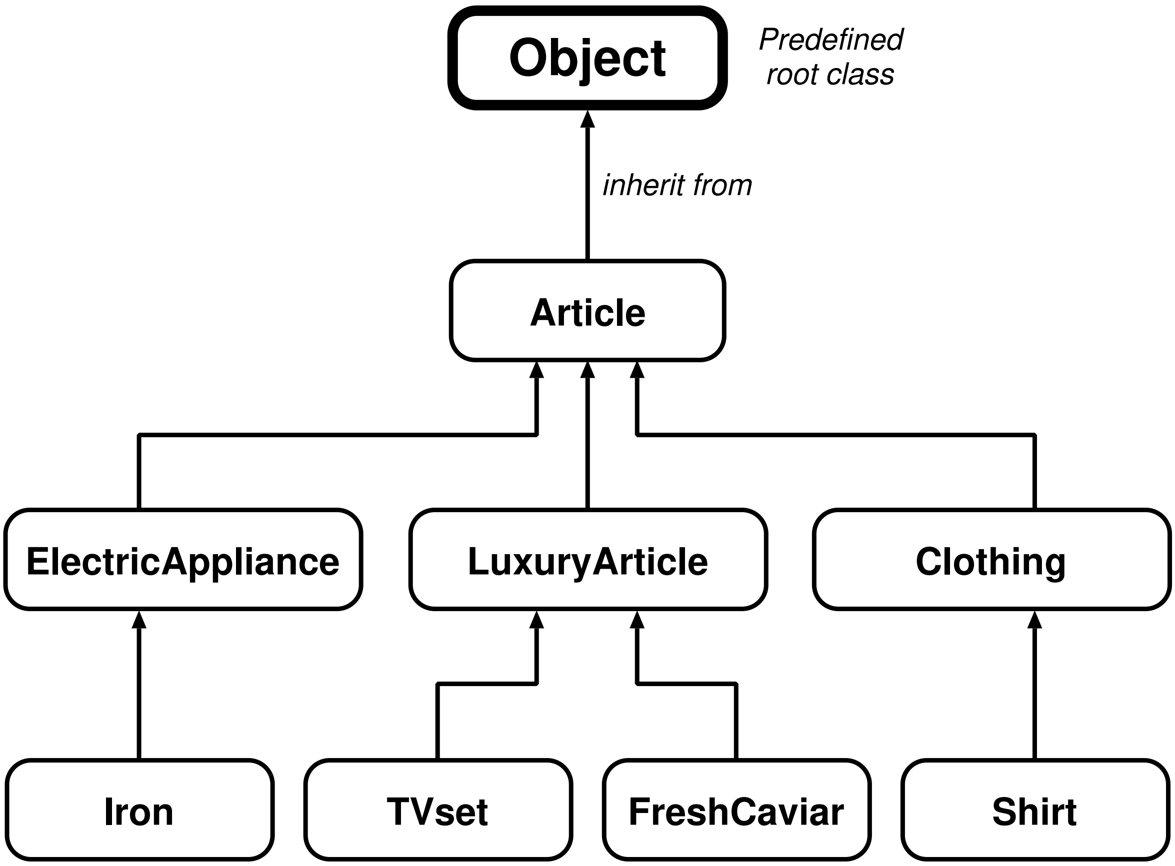
```
// THE COMPLETE DEFINITION OF CLASS Clothing COULD BE
// OBTAINED BY INCLUDING THE DEFINITION OF CLASS Article
// (REPRESENTED BY FRAMED BOXES) INTO THE INITIAL
// DEFINITION OF CLASS Clothing

class Clothing {
  private:
    _code;
    _description;
    _basePrice;
    _quantity;
    _size;
    _colour;
  public:
    code() { return(_code); }
    description() { return(_description); }
    basePrice() { return(_basePrice); }
    quantity() { return(_quantity); }
    price() { return(1.2 * _basePrice); }
    size() { return(_size);
    colour() { return(_colour);
    transportCost() { return(0.05 * _basePrice); }
    setCode(c) { _code) := c; }
    setDescription(d) { _description := d; }
    setBasePrice(b) { _basePrice := b; }
    setQuantity(q) { _quantity := q; }
    setSize(s) { _size := s; }
    setColour(c) { _colour := c; }
    remove(q) { _quantity := _quantity - q; }
    add(q) { _quantity := _quantity + q; }
};
```

```
// CLASS LuxuryArticle INHERITS FROM CLASS Article

class LuxuryArticle : Article {
public:
    // Redefinition of the inherited 'price()' function
    // (special tax rate for luxury articles)
    price()
      { return(1.3 * _basePrice); }
};
```

```
// RESULTING DEFINITION OF CLASS LuxuryArticle:
// THE INHERITED DEFINITION OF METHOD price()
// IS SUBSTITUTED FOR THE NEW DEFINITION

class LuxuryArticle {
  private:
    _code;
    _description;
    _basePrice;
    _quantity;
  public:
    code() { return(_code); }
    description() { return(_description); }
    basePrice() { return(_basePrice); }
    quantity() { return(_quantity); }
    price() { return(1.3 * _basePrice); }
    transportCost() { return(0.05 * _basePrice); }
    setCode(c) { _code) := c; }
    setDescription(d) { _description := d; }
    setBasePrice(b) { _basePrice := b; }
    setQuantity(q) { _quantity := q; }
    remove(q) { _quantity := _quantity - q; }
    add(q) { _quantity := _quantity + q; }
};
```

SINGLE INHERITANCE

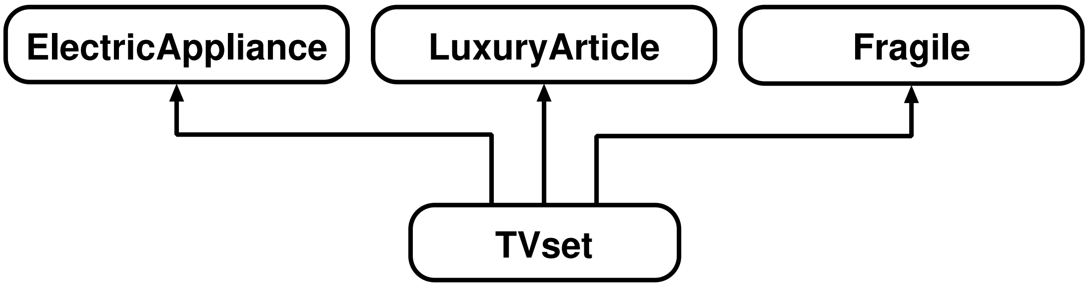
- Each class (excepting `Object`) inherits from one and only one other class



- Class `Object` defines the data and methods common to all objects: e.g. `print` (an object)

MULTIPLE INHERITANCE

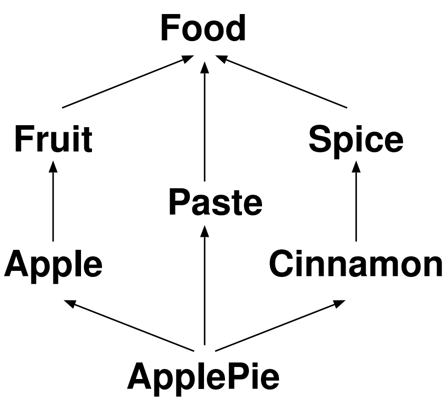
- Each class (excepting `Object`) may inherit from one or more other classes

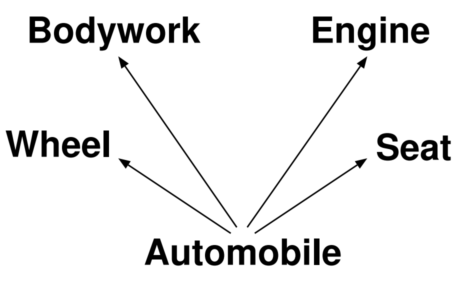


- Each superclass corresponds to a different **point of view**

  A TV set is defined as the union of the characteristics of an electric appliance, a luxury article, and a fragile article (for transport)

- Warning: **to inherit is not to aggregate**





An apple pie should not be defined as the union of the characteristics of apples, paste and cinnamon.

1.5 BINDING

```
...
tv := new(TVset, code: "501.R2D2",
                 description: "Stereo TV set",
                 basePrice: 998.99,
                 quantity: 5, ...);
...
send(tv, "remove", 2);
...
p := 2 * send(tv, "price");
    // Code of method from class Article
    // or from class LuxuryArticle?
...
```

- Binding is to relate a given method call to the code to be effectively executed
- It may be performed at compile-time (**static binding**) or at run-time (**dynamic binding**)
- A simplified algorithm:

  `send(`*`target`*`, `*`selector`*`, `*`param-1 ... param-N`*`);`

  is interpreted as:

  apply
  method associated with (*selector*, class-of(*target*))
  to
  param-1 ... param-N

OVERLOADING

- Same-named methods may belong to the interface of different classes
- Each method performs the corresponding operation according to the specific nature of the set of objects described by the class

☛ data abstraction is increased

```
article[1] := new(TVset, code: "501.R2D2",
                         description: "Stereo TV set",
                         basePrice: 1000.00,
                         quantity: 5, ...);
article[2] := new(Shirt, code: "888.C3PO",
                         description: "Hawaiian shirt",
                         basePrice: 100.00,
                         quantity: 25, ...);
...
for (i := 1 to nbArticles) {
  p := send(article[i], "price");
  print("Article ", i, ": ", p, " F");
}
```

produces the following output:

```
Article 1: 1300 F
Article 2: 120 F
...
```

Luxury tax rate (30%) has been used
Ordinary tax rate (20%) has been used

THE “SUPERMETHOD”

- Calling the supermethod is calling the redefined method in the body of the redefining method

- Example: The tax rate of a luxury article is 10% more than the tax rate of an ordinary article:

  price of a Luxury Article = 1.1 ∗ price of an Article

```
class Article {
  ...
public:
    ...
    price() { return(1.2 * _basePrice); }
    ...
};
```

```
class LuxuryArticle {
public:
    // Redefinition of the inherited ‘price()’ function:
    // special tax rate for luxury articles
    price() { return(1.1 * send(super, "price")); }
    ...
};
```

- In some way, it results in specializing a method:
  The `price` method of class `LuxuryArticle` is defined relatively to the `price` method of class `Article`

☛ reusability is increased
software maintenance is easier

1.6 METACLASSES

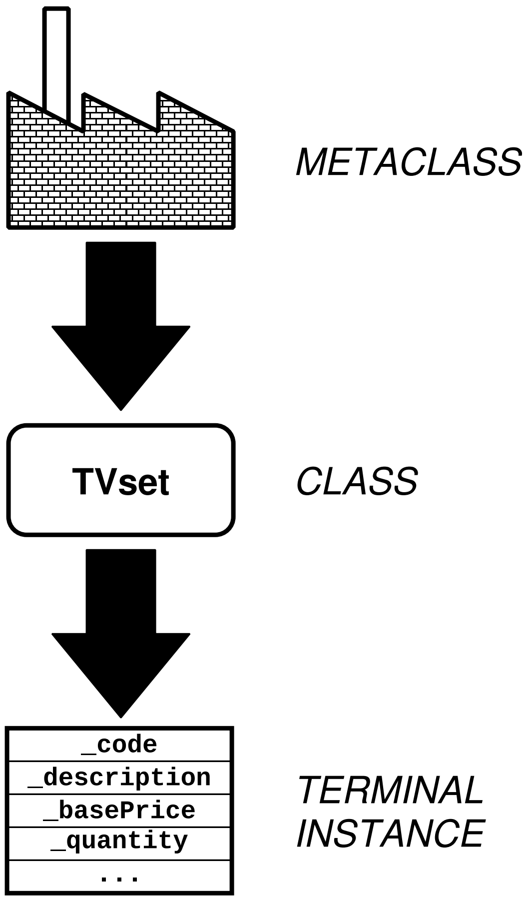
- An object is an instance of a class
  It can receive messages:

```
send(tv, "remove", 2);
```

- A class is a description
  It is not instantiated and cannot receive messages
  ⇒ an object must be created using built-in function `new`:

```
tv := new(TVset, code: "501.R2D2",...  );
```

☛ Let a class be an object: it is instantiated from an other class, called a **metaclass**



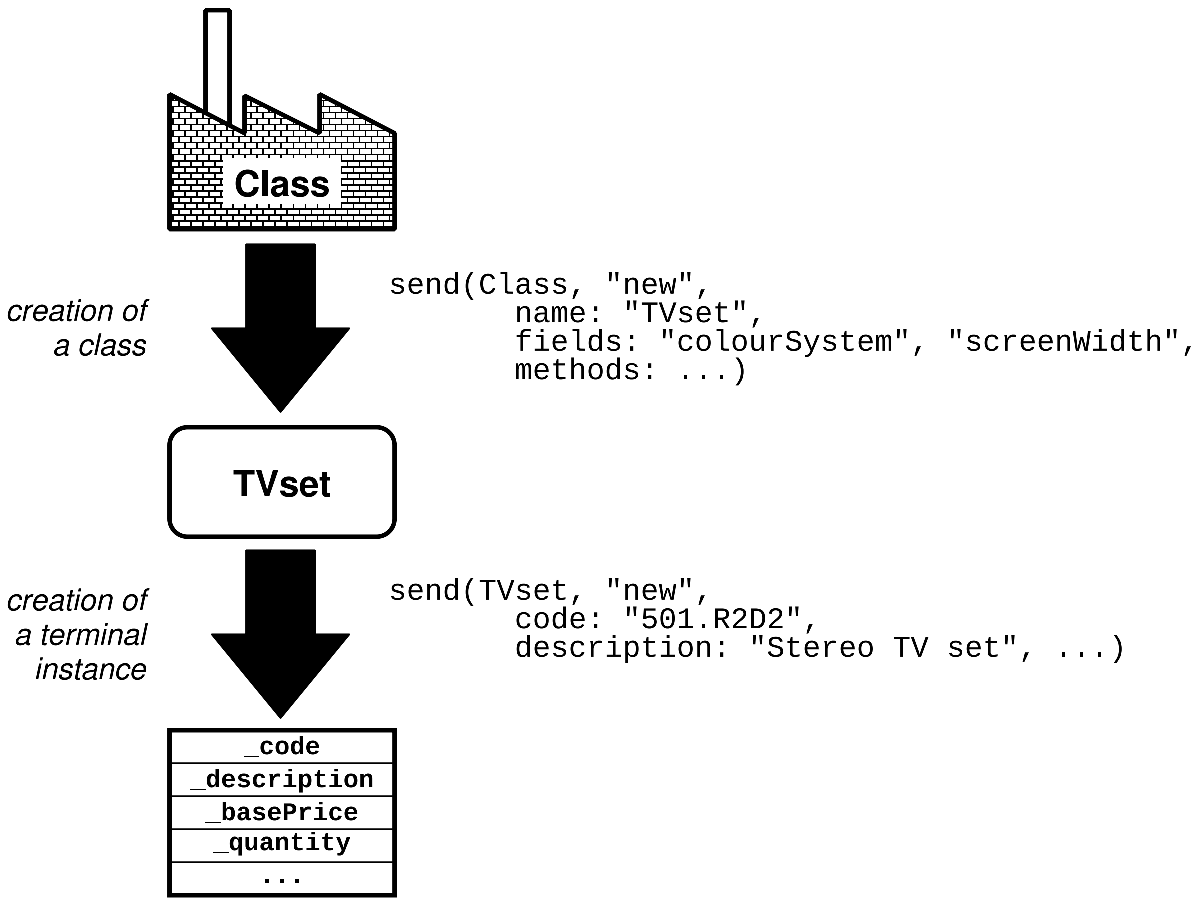
- A metaclass describes a class
  It generates **classes**

- A class describes an object set
  It generates **terminal instances**

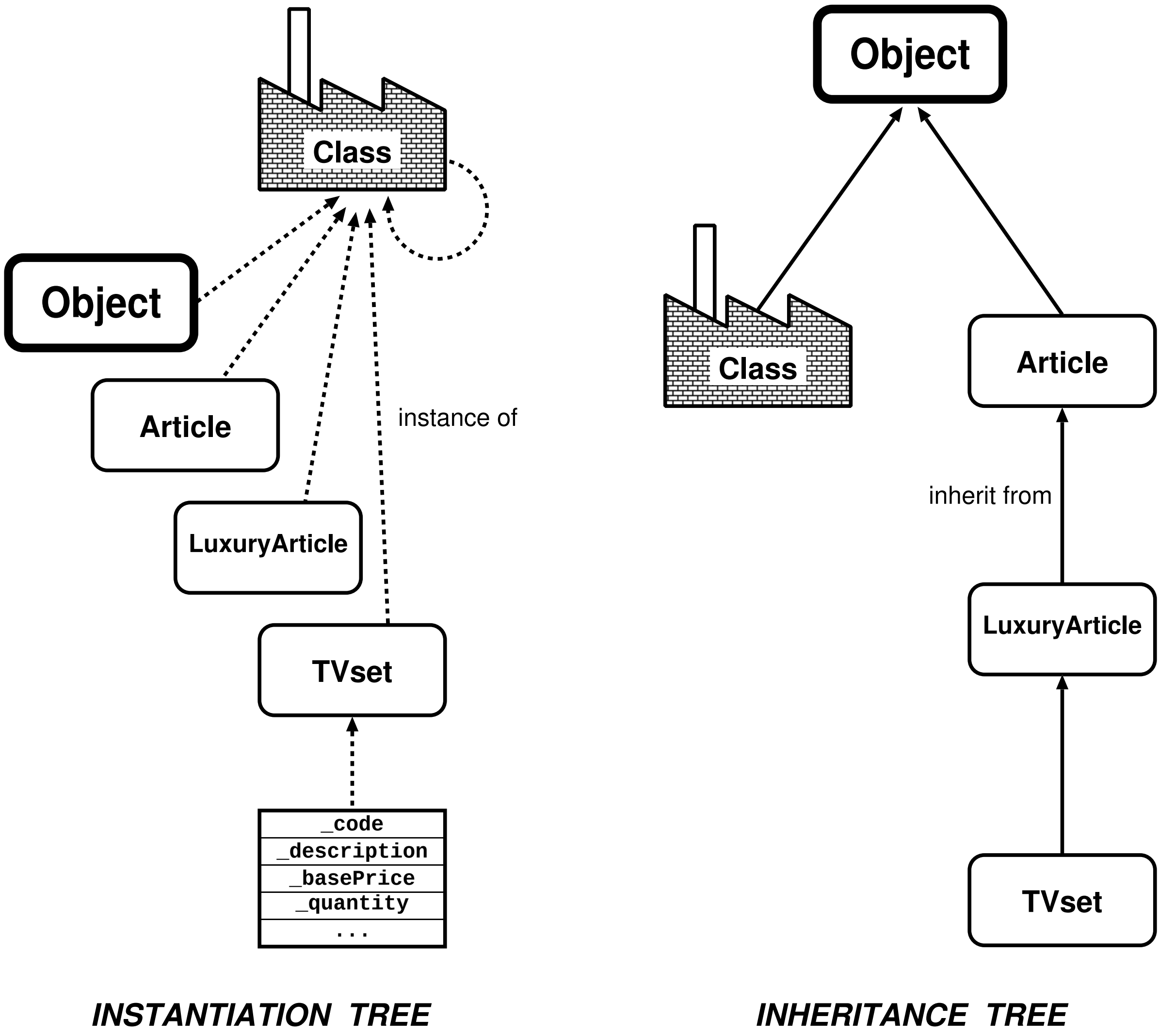
- ☛ Terminal instances, classes, and also metaclasses, now are all **objects**

  They are instantiated from classes

- Objects are created and manipulated using a unique control structure: message sending

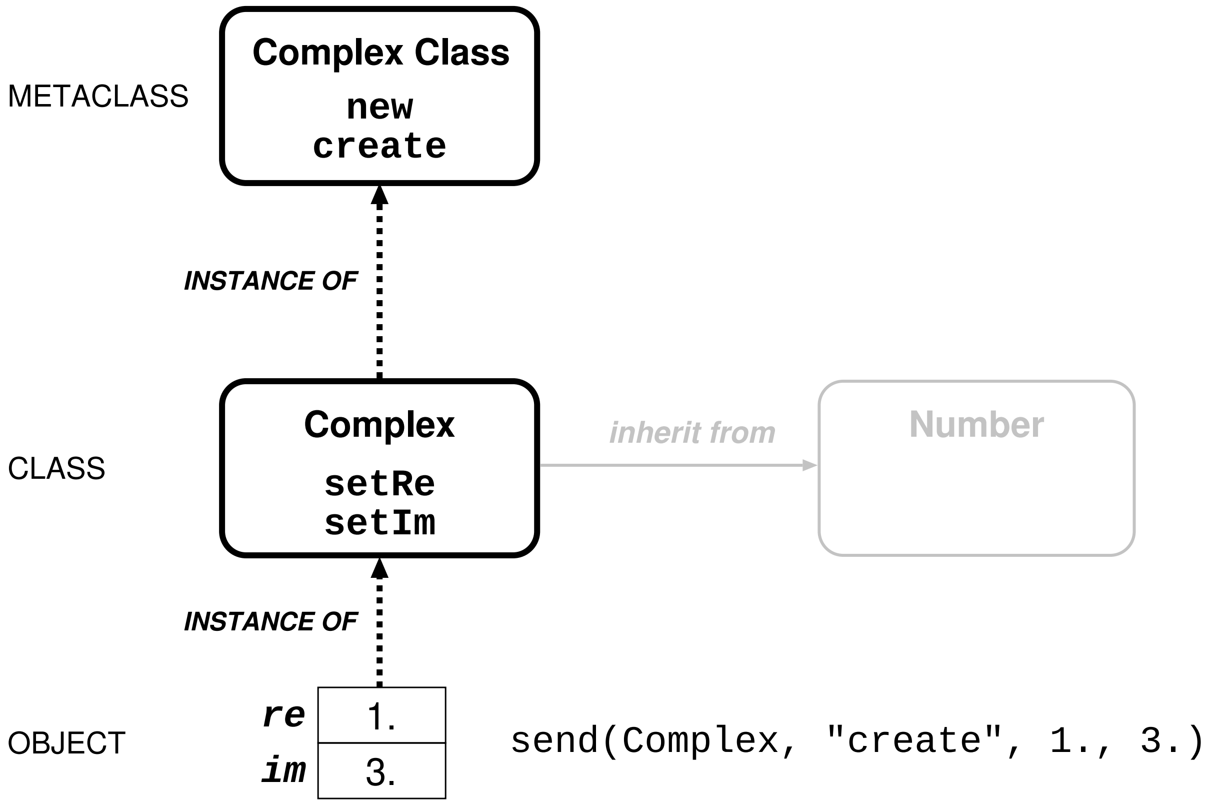


EXAMPLE: THE SMALLTALK-76 MODEL



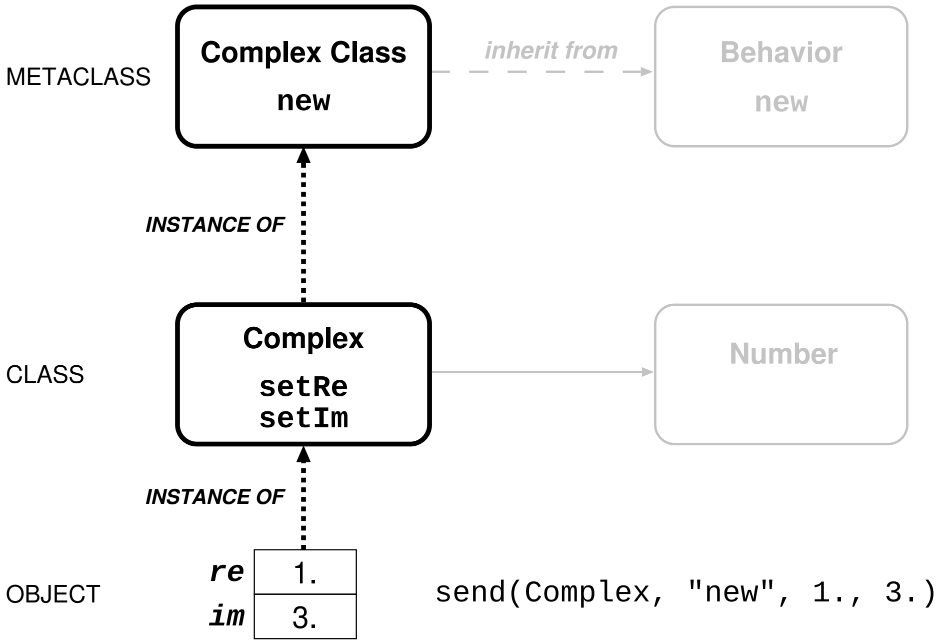
- Class `Class` generates classes:
  - `Class` defines the behaviour and structure of classes $\Rightarrow$ All classes have same behaviour and structure
  - `Class` is an instance of itself (to close the model)
- The `Object` class defines the behaviour common to all terminal instances

EXAMPLAR USE OF METACLASSES:
Customizing the creation of a (Smalltalk-80) object




```
create(realPart, imPart) {
  // Creation of a complex number
  // by instantiating myself, i.e. class Complex
    newComplex := send(self, "new");
  // Set the real part of the new object
    send(newComplex, "setRe", realPart);
  // Set the imaginary part of the new object
    send(newComplex, "setIm", imPart);
  // Return the new object
    return(newComplex);
}
```

REAL CASE



- Standard instantiation method `new` (no parameter) is defined by class `Behavior`
- Metaclass `Complex Class` inherits the standard instantiation method `new` from class `Behavior`
- Method `new` is **redefined** in metaclass `Complex Class` so that it admits two parameters

```
new(realPart, imPart) {
  // Creation of a complex number
  // by instantiating myself, i.e. class Complex
    newComplex := send(super, "new");
  // Set the real part of the new object
    send(newComplex, "setRe", realPart);
  // Set the imaginary part of the new object
    send(newComplex, "setIm", imPart);
  // Return the new object
    return(newComplex);
}
```

PART II

STATIC/STRONG TYPING

C++/Eiffel/Java-like world

The correctness of typing is checked at compile-time,

before running programs

VOCABULARY

- A **routine** is a procedure or a function
- A class defines a **type**
- A type must be declared for each variable, parameter of routine and result of function

  This type is called the **static type**
- At run-time, a variable (or a parameter, or a result of function) contains an object of a certain type

  This type is called the **dynamic type**
- Routine call syntax uses dot notation:

  target **.** *routine*(*param-1*,... *param-N*) ;

2.1 POLYMORPHISM and DYNAMIC BINDING

```
character code;
float p;
Shirt sh;
TVset tv;
Article a;
...
tv := new(TVset, "501.R2D2", "Stereo TV set", 1000.00,...);
sh := new(Shirt, "888.C3PO", "Hawaiian shirt", 100.00,...);
...
read(code);
if (code = 's')
  a := sh;
else
  a := tv;
p := a.price();  // Dynamic binding
...
```

- **Polymorphism**

  A given variable, `a` here, may succesively refer to objects of different types at run-time

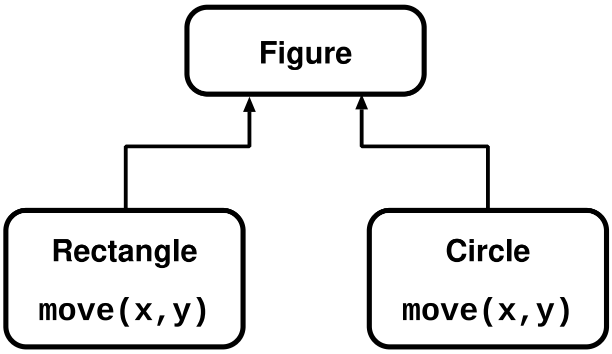
- **Type conformance**

  The types of `sh` and `tv` must conform to (or be a subtype of) the type of `a`

- **Dynamic binding**

  Routine `price()` is invoked according to the dynamic type of `a`

STATIC TYPE CHECKING MUST REMAIN POSSIBLE

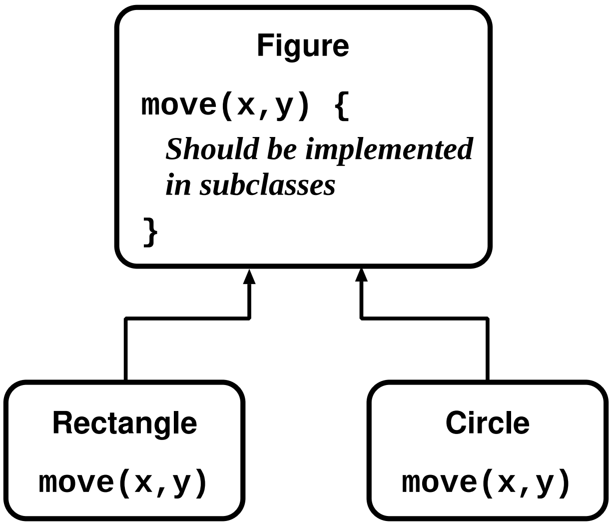



```
character code;
Figure p;
Rectangle r;
Circle c;
...
r := new(Rectangle, ...);
c := new(Circle, ...);
read(code);
if (code = 'c')
  p := c;
else
  p := r;
p.move(6, 35);     // Error at compile-time
...
```

- The static type of `p` is `Figure`:
  class `Figure` does not provide any `move()` routine
  $\Rightarrow$ compilation of `p.move(6,35);` fails

- But the dynamic type of `p` may be `Rectangle` or `Circle` at run-time...

ABSTRACT CLASSES



- It is specified that all subclasses of class `Figure` implement a `move()` routine

  ⇒ static type checking becomes possible

- The implementation of some operations is deferred to the subclasses: class `Figure` describes the concept of "figure" in an abstract way

  ⇒ class `Figure` is called an **abstract class**

- The interface of an abstract class can be used by programmers as a guide for the implementation of specific subclasses (figures, described by subclasses such as `Rectangle` or `Circle`)

  ⇒ programmers do know the routines the subclasses must be provided with

- It does not make sense to instantiate an abstract class

2.2 GENERICITY

- A concept like “stack” applies to many data types
- A **generic class** is parameterized by (formal parameter) types to avoid the definitions of many similar classes:
  `IntegerStack` for integers, `ArticleStack` for articles, etc.

```
class Stack[TYPE] {
private:
  Array[TYPE] _implementation;
  integer _size;
  integer _topIndex;
public:
  ...
  push(TYPE v) {
    if (_topIndex = _size) {
      write("Error: Stack is full!\n");
      abort();
      }
    _topIndex := _topIndex + 1;
    _implementation[_topIndex] := v;
  }

  TYPE top() {
    if (self.empty()) {
      write("Error: Stack is empty!\n");
      abort();
      }
    return(_implementation[_topIndex]);
  }
  ...
};
```

INSTANTIATION OF A GENERIC TYPE

A specific type is obtained from a generic type by giving **effective types** for formal parameter types

```
Stack[integer] iStack;
Stack[Article] aStack;
Article a;
...
iStack := new integer(500)
aStack := new Article(1000);
a := new(Article, "FX-1138", "C++ book",...);
...
aStack.push(a);    // Push an article
...
iStack.push(33);   // Push an integer
...
aStack.push(357);
        // Error at compile-time:
        // An integer cannot be pushed on
        // a stack of articles...
...
```

Warning: Using inheritance for implementation may be dangerous

```
class Array[TYPE] {
  ....
  public:
    ...
    integer count() { ... }
    put(integer index, TYPE value) { ... }
    ...
};
```

```
class Stack[TYPE] : Array[TYPE] {
  private:
    integer _topIndex;
  public:
    ...
    push(TYPE v) { // Inherited methods are available
      if (_topIndex = self.count())
        write("Error: Stack is full!\n");
        abort();
        }
      _topIndex := _topIndex + 1;
      self.put(_topIndex, v);
    }
    ...
};
```

```
Stack[integer] iStack;
...
iStack.push(32767);
iStack.put(10, 0);
    // Violation of the specification of a stack:
    // A stack may be used like an array,
    // because routines inherited from class Array
    // may be called for 'iStack'
...
```

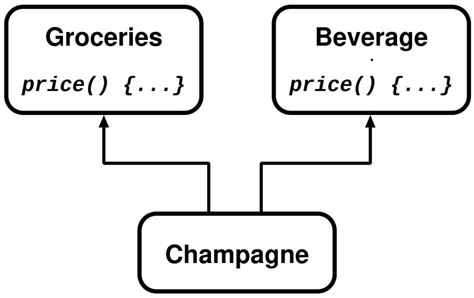
PART III

BACK TO INHERITANCE

conflicts and exceptions

INHERITANCE CONFLICTS

Classes are specified by **properties** (data and methods)



☛ *A conflict occur when a given class inherits several homonymous properties*

SOFTWARE ENGINEERING

- inheritance is mainly used to avoid duplication of source code: classes do not exist at run-time
- the correctness of typing must be statically proved before executing programs
- $\Longrightarrow$ only the programmer is supposed to know how to *correctly* solve conflicts

ARTIFICIAL INTELLIGENCE

- inheritance is used to represent human knowledge: concepts are organized into hierarchies (zoological taxonomy, for example), including conflicts and exceptions
- human knowledge may be dynamically modified
- $\Longrightarrow$ built-in tools are needed, to dynamically deal with hierarchy modifications and conflicts

EXPLICIT CONFLICT SOLVING

☛ especially in software engineering

- *Eiffel*
  Homonymous methods are renamed in the subclass, so that they have different names and can be called without ambiguity

- *C++*
  The method class is specified in the call

```
class Groceries {
  public:
    float price() { return(_basePrice * 1.05); }
    ...
};
class Beverage {
  private:
    int capacity;
  ...
  public:
    float price()
      { return(capacity * _basePrice * 1.05); }
    ...
};
class Champagne : Groceries, Beverage {
  public:
    float jeroboamPrice()
      { return(Beverage::price() * 4.); }
    ...
};
```

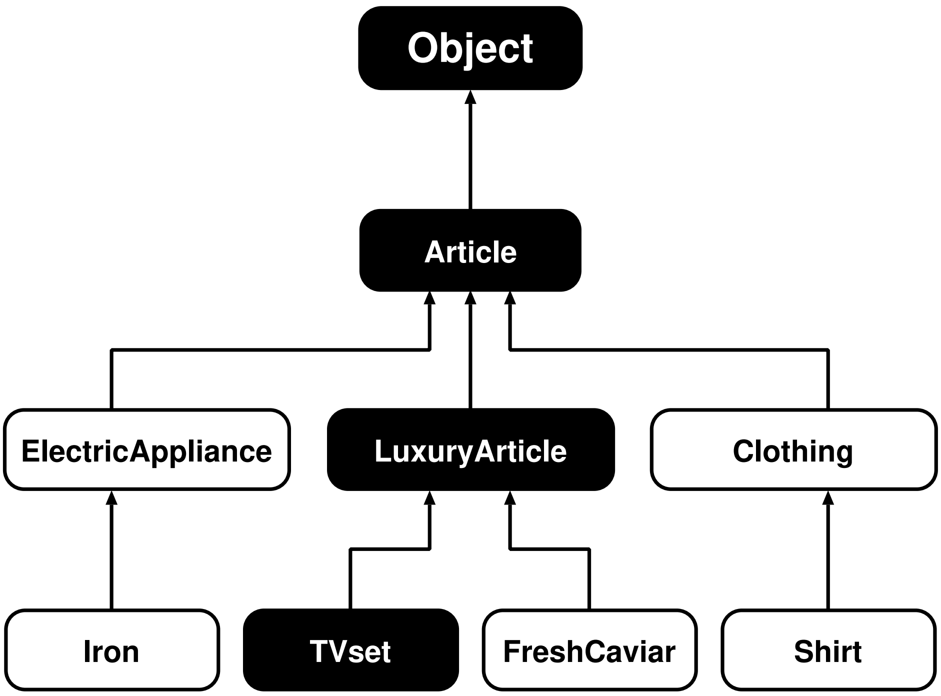
IMPLICIT CONFLICT SOLVING

☛ especially in artificial intelligence

Problem:

- Let C_1 be a class
- Let $\mathcal{G}(C_1)$ be the inheritance graph of C_1
- Let P be a property

From what class $C_2 \in \mathcal{G}(C_1)$ does class C_1 inherit P ?

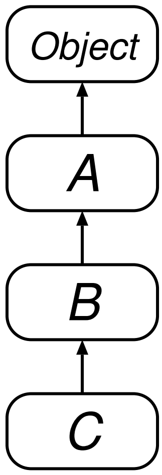


$\mathcal{G}(TVset)$: The inheritance (sub)graph of class $TVset$

THE INHERITANCE RELATIONSHIP

- It forms a **graph without cycle**, generally including a "root" without parent class (class *Object*, typically)

- It is **transitive**



class C inherits from class B
class B inherits from class A
=> class C inherits from class A

- It determines an **ordering** over the classes
  - class *C* inherits from class *B* $\Rightarrow$ $C < B$
  - the root is the "greatest class"
  - **the class precedence list** of a given class *C* is the ordered list of its superclasses ($\in \mathcal{G}(C)$)

SOLUTION TO THE PROBLEM:

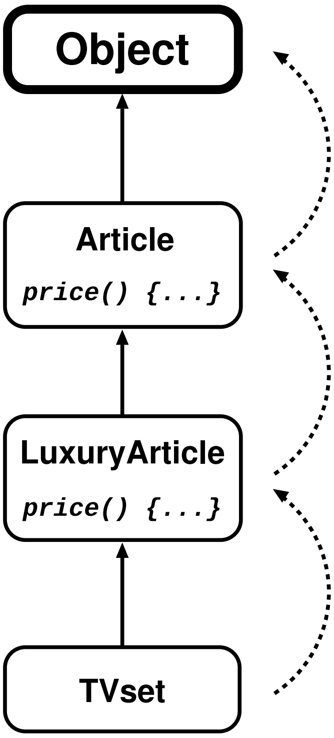
- linearize (i.e. transform into a list) the inheritance graph, using a traversal of the hierarchy respecting a **total** ordering
- the inherited property P belongs to the first class of the list which holds P

SINGLE INHERITANCE

As the inheritance hierarchy is a tree, the **transitive closure** of the inheritance relationship is a total ordering

```
TVset tv;
...
tv := new(TVset, "501.R2D2", "Stereo TV set", 1000.00,...);
...
p := tv.price();  // Inherited from LuxuryArticle
                  // SHADOWING IS EFFECTIVE
```

- CPL(*TVset*) = { *TVset*, $\boxed{\textit{LuxuryArticle}}$, *Article*, *Object* }
- The inheritance hierarchy is dynamically examined from the bottom up, following the inheritance links, until method `price()` is found

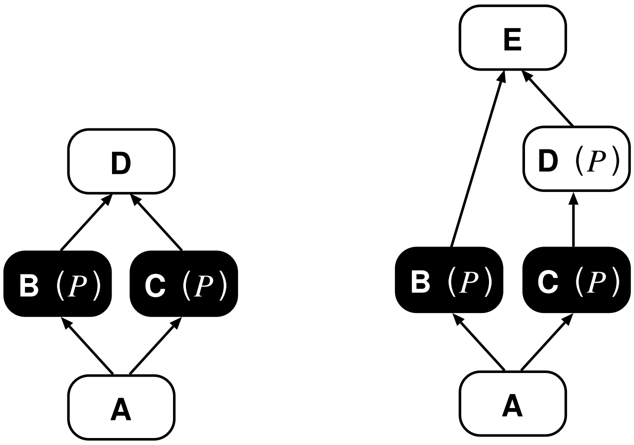


the look up process

MULTIPLE INHERITANCE

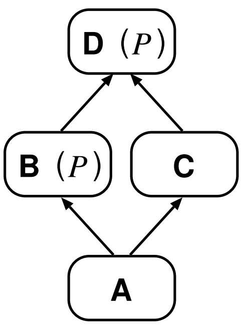
As the inheritance hierarchy is not a tree, the transitive closure of the inheritance relationship is a **partial** ordering

- Some classes are not comparable
  $\Longrightarrow$ their homonymous properties are conflicting



conflict between B and C

- comparable classes are still comparable (*B* and *D*)
  $\Longrightarrow$ their homonymous properties are not conflicting



no conflict

☛ find criteria so that the resulting precedence list best respects the designer's intentions of the class hierarchy

EXAMPLES OF CRITERIA

- The partial ordering induced by the inheritance relationship must be preserved

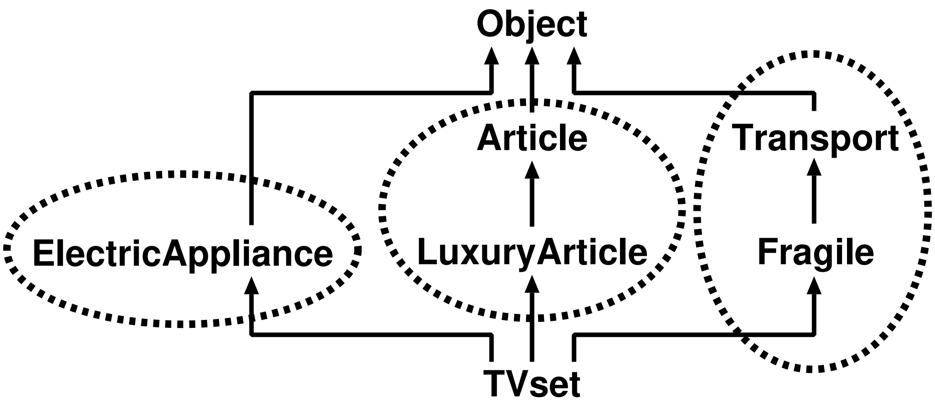
  ⇒ if class *A* inherits from class *B*, class *A* must precede class *B* in the resulting class precedence list

- Local multiplicity (ordering of the "direct" superclasses, given at definition) must be preserved

```
class Champagne : Groceries, Beverage {
public:
  float jeroboamPrice()
    { return(Liquid::price() * 4.); }
  ...
};
```

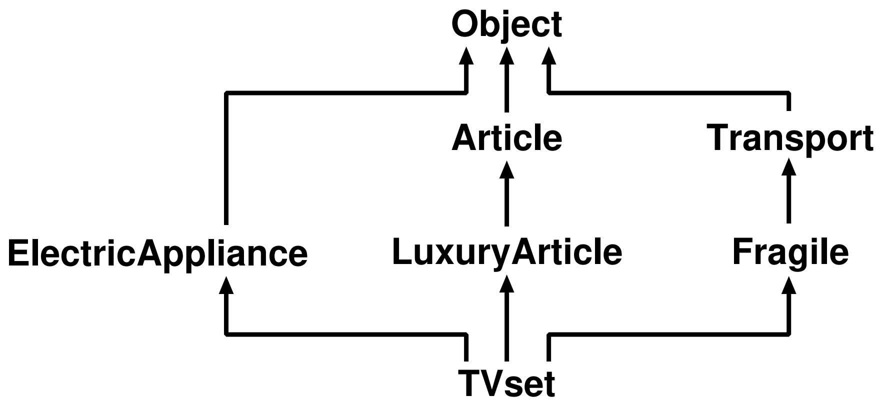
  ⇒ class *Groceries* must precede class *Beverage*

- Subgraphs (modules) corresponding to "points of view" must not be mixed in the precedence list



  ⇒ for example, no class must appear between class *Transport* and class *Fragile*

ALGORITHMS



- Depth-first search:
  inheritance partial ordering is not preserved

  { *TVset*, *ElectricAppliance*, *Object*, *LuxuryArticle*, *Article*, *Fragile*, *Transport* }

- Breadth-first search:
  inheritance partial ordering and modules are not preserved

  { *TVset*, *ElectricAppliance*, *LuxuryArticle*, *Fragile*, *Object*, *Article*, *Transport* }

- Simplified Ducournau-Habib's/CLOS algorithm
  - collect nodes in the order they are pushed during depth-first search:

    { *TVset*, *ElectricAppliance*, *Object*, *LuxuryArticle*, *Article*, *Object*, *Fragile*, *Transport*, *Object* }

  - eliminate duplicates starting from the end of the list:

    { *TVset*, *ElectricAppliance*, $\boxed{\textit{Object}}$, *LuxuryArticle*, *Article*, $\boxed{\textit{Object}}$, *Fragile*, *Transport*, *Object* }

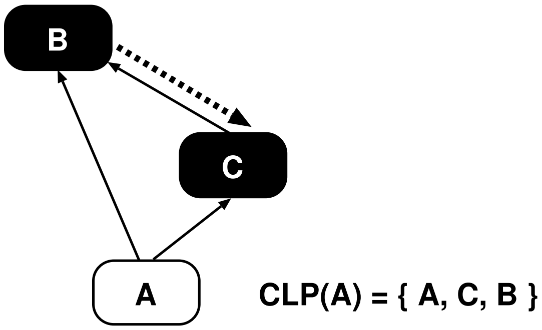
  - the final result (most often) preserves the criteria:

    { *TVset*, *ElectricAppliance*, *LuxuryArticle*, *Article*, *Fragile*, *Transport*, *Object* }

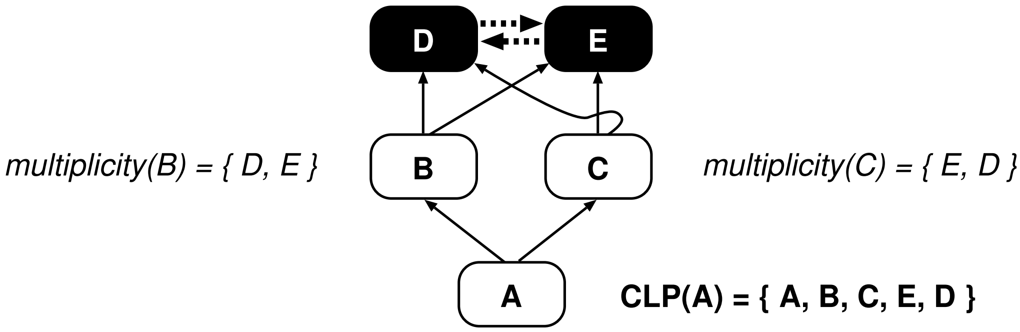
UNSOLVABLE CASES

Wrong design?

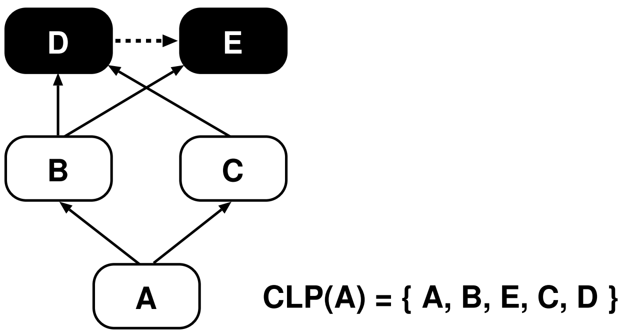
- Contradiction between inheritance and multiplicity



- Contradiction between multiplicities



- Specific case

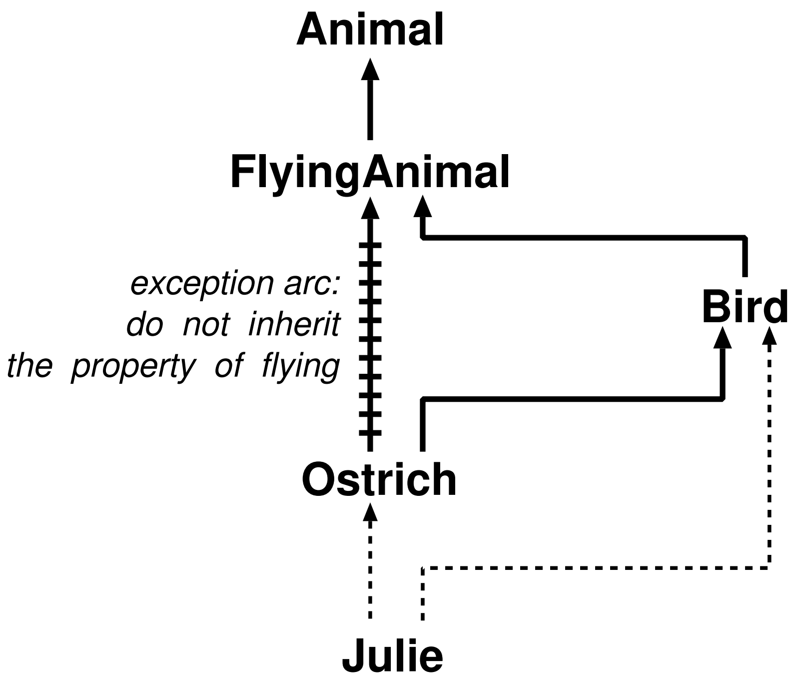


Solution: Add a virtual arc $D \rightarrow E$

3.2 INHERITANCE EXCEPTIONS

Till now, inheritance uniformly applied to all properties

☛ it is not the case in the real world:
an ostrich is a bird but does not fly



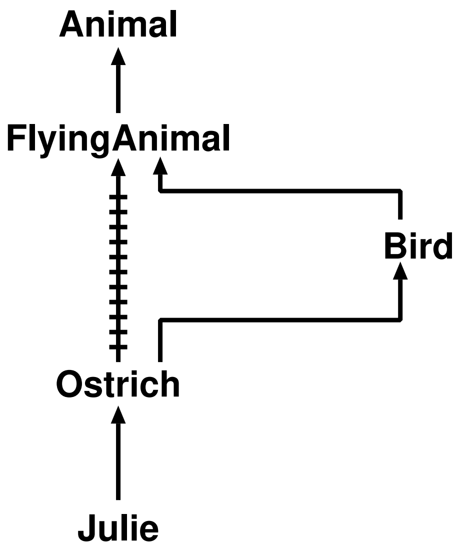
⇒ the inheritance relationship is no more transitive

Dealing with inheritance exceptions is one of the most difficult problem (in knowledge representation)

For example, (during a classification process) *Julie* is first viewed as a bird, then as an ostrich:

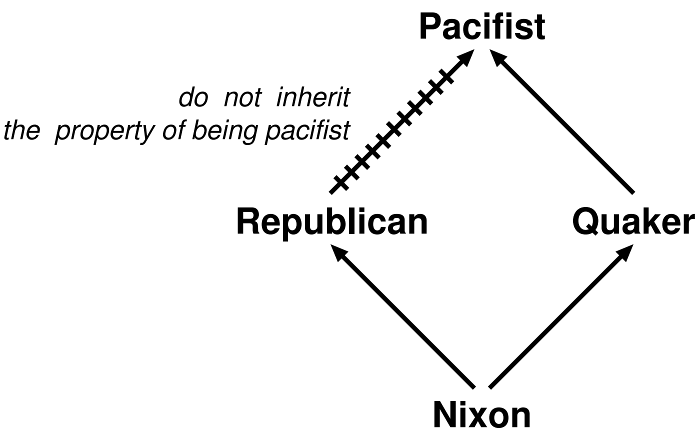
- *Julie* does not fly because it inherits from *Ostrich*
- *Julie* flies because it inherits from *Bird*

REDUNDANT CONTRADICTION



Arc *Julie* → *Bird* is redundant and introduces a contradiction
☛ suppress the redundant arc

PURE CONTRADICTION

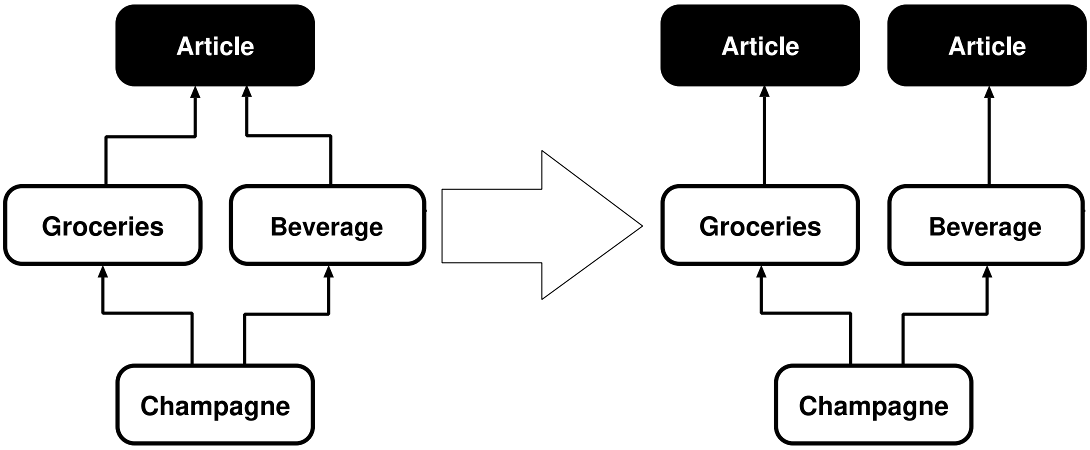


Nixon is both Quaker and Republican: Is he a Pacifist?

☛ No solution:
Ask Nixon or arbitrarily decide for a solution!

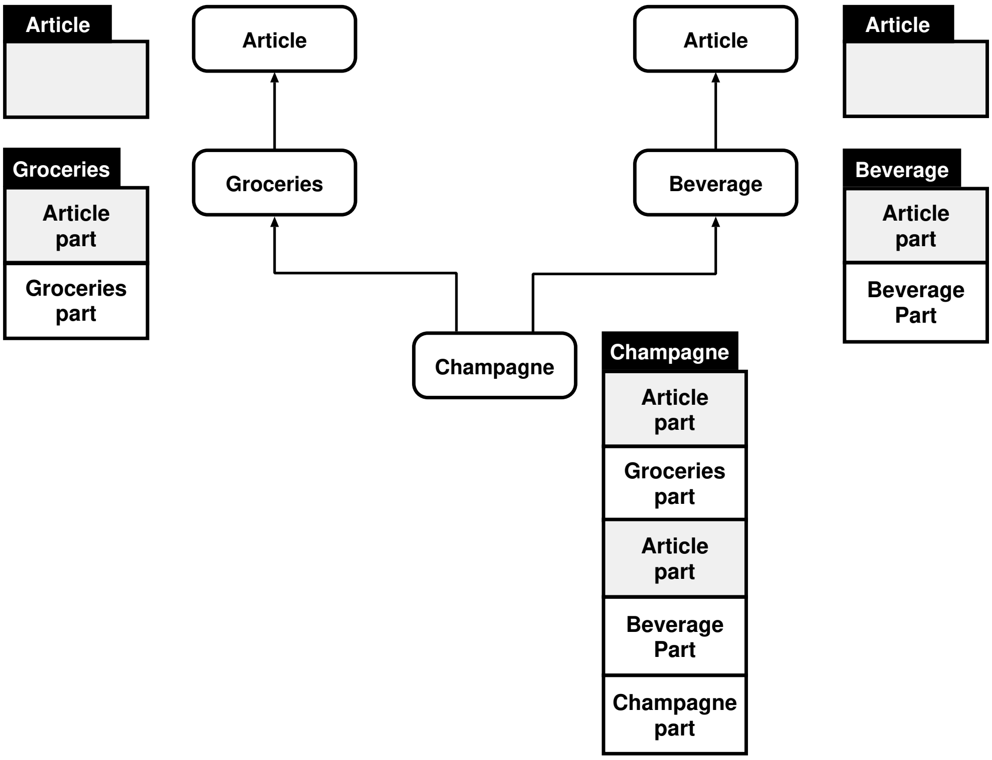
3.3 REPETEAD INHERITANCE

- Class *Champagne* inherits two times from class *Article*, by two different inheritance paths



- Inherited information may be duplicated

  C++: The structure of an object of class *Champagne* includes two instances of an object of class *Article*



CONCLUSION

- A class describes a set of objects sharing a common structure and a common behaviour
- Encapsulation allows the physical structure (data) to remain hidden
- Actions may be performed on an object using the operations specified by the interface of the object class
- A class may inherit the structure and behaviour of other classes (superclasses)
- Classes form a hierarchy organized from the most general (top) to the most specialized (bottom)
- Inheritance raises many unsolved theoretical problems, especially about name conflicts and exceptions