C.6 The Evolution of the Object Model

1 Generations of Programming Languages

- First generation (1954 - 1958)
  - Mathematical Expressions (FORTRAN I, ALGOL 58)
- Second generation (1959 - 1961)
  - Subroutines, separate compilation (FORTRAN II)
  - Block structure, data types (ALGOL 60)
  - Data description, file handling (COBOL)
  - List processing, pointers, garbage collection (Lisp)
- Third generation (1962 - 1970)
  - various successors to ALGOL 60 (ALOGOL 68, Pascal)
  - Classes, data abstraction (Simula)

1 Generations of Programming Languages (2)

- *The Generation Gap* (1970 - 1980)

  many different languages were invented,
  but few endured …
  - C
  - Modula
  - Ada
- Object-oriented programming languages (1980 - today)
  - Smalltalk (successor to Simula)
  - C++ (evolved from C and Simula)
  - Eiffel (evolved from Simula and Ada)
  - Java

C.7 Fundamental Concepts of the OO Paradigm

- Abstraction
  - Encapsulation
  - Abstract data type
- Modularization
- Hierarchy
- Typing
  - Hierarchy of types
  - Polymorphism
  - Genericity
- Concurrency
- Persistence

1 Abstraction

Fundamental concept for solving complex problems

- Emphasize details which are relevant for the entire solution
- Suppress details which are (for the moment) immaterial or diversionary

→ Object orientation
- important:
  - ➤ Signature of an object
  - ➤ Semantics of an object

  } Outside view

  ➥ contract model: Outside view = contract with other objects
- unimportant:
  - ➤ Implementation of an object

- Describe the abstraction first and think about the implementation later

2 Encapsulation

= Information Hiding
Concealing of the implementation of an abstraction from the users of the abstraction

- Complement to abstraction
  - ➤ Abstraction exposes the external properties of an object
  - ➤ Encapsulation hides the internals
- Fundamental for abstraction

  B. Liskov *For abstraction to work, implementations must be encapsulated*
- Encapsulation & object orientation
  - ◆ Representation of the object state
  - ◆ Implementation of the methods

➔ Abstract data type

3 Abstract Data Type

? How can we provide a complete, precise & unambiguous description of an abstraction?

- ADT: Model to describe
  - ➤ Properties of data structures (Semantics!)
  - ➤ Operations of the data + semantic effects
  - ➤ NOT: Implementation of the data structures
- Specification → Description of all essential properties
- Overspecification: Details about the representation/implementation

➔ ADT & object orientation
- ◆ Class = Implementation of an ADT
- ◆ Data abstraction: Object state is accessible only through methods

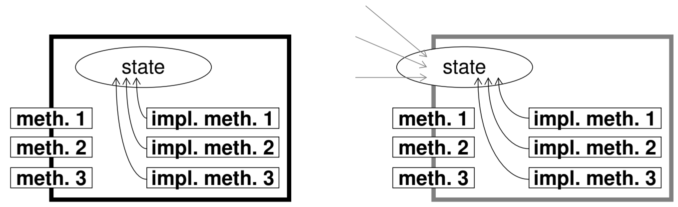
3 Abstract Data Type (2)

- Example: Specification of an ADT “Stack”

```
TYPES
        STACK[X]

FUNCTIONS
        empty: STACK[X] → BOOLEAN
        new: → STACK[X]
        push: X × STACK[X] → STACK[X]
        pop: STACK[X] → STACK[X]
        top: STACK[X] → X

PRECONDITIONS
        pre pop(s: STACK[X]) = (not empty(s))
        pre top(s: STACK[X]) = (not empty(s))

AXIOMS
        ∀x:X, s:STACK[X]:
        empty(new())
        not empty(push(x,s))
        top(push(x,s)) = x
        pop(push(x,s)) = s
```


3 Abstract Data Type (3)

- Data abstraction in OOP
  - ➔ the object state is accessible only through the methods of the object



object as implementation of an ADT   object without data abstraction

- Data abstraction in C++ and Java
  - ◆ Scope rules (private / protected)

4 Modularity

Partitioning a program into individual components can reduce its complexity

- ◆ Problem partitions are easier to comprehend
- ◆ Different problem partitions may be assigned to different developer teams
- ◆ Module = separate unit in development

- most important: partitioning creates boundaries = interfaces
  - ➔ well-defined
  - ➔ documented

  } invaluable for the comprehension of the whole problem
- Many programming languages make a difference between interface and implementation of a module
- Structured Design: Grouping of subprograms
- OOD: Grouping of classes and objects (from the design's logical structure)

5 Hierarchy

- ◆ Abstraction & encapsulation help to hide details of components
- ◆ Modularity helps to cluster related abstractions

- Comprehension of large problems still difficult
  - ➔ too many abstractions
  - ➔ additional means to organize abstractions
- Abstractions often form hierarchies
  - ➤ common properties → more general abstractions
  - ➤ differences → specialization
  - ➔ Hierarchy: Ordering of abstractions
- Hierarchy & object orientation
  - ➤ Class structure: Inheritance → “is a” hierarchy
  - ➤ Object structure: Aggregation → “part of” hierarchy

6 Typing

- Concept derives primarily from the theory of ADTs

  Booch: *Typing is the enforcement of the class of an object, such that objects of different types may not be interchanged, or at most, they may be interchanged only in very restricted ways*
- Typing enables a programming language to check and enforce design decisions
  - ➔ essential for programming-in-the-large [Weg87]
- Strong typing:
  Conformance of all types in an expression is guaranteed
  - ➔ Static typing:
    conformance checked completely at compilation time
    - ➤ less flexibility
      - cannot detect compatible types
      - cannot support dynamic binding
  - ➔ More flexibility with polymorphism and genericity

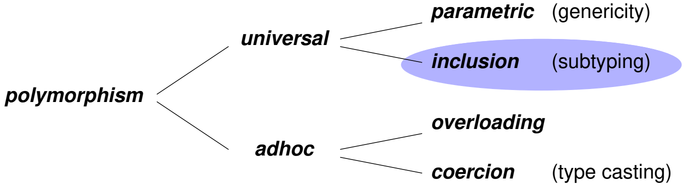
7 Type Hierarchy

- Often 1:1 relationship between classes and types — but not necessary
  - ➤ different classes may implement the same type
  - ➤ one class may implement different types
- Hierarchy of classes: superclass ← subclass
  - ◆ Objective: code inheritance
  - ◆ Subclass not necessarily confirming to the type of the superclass
- Hierarchy of types: supertype ← subtype
  - ◆ Objectives: – behavior inheritance
    – declaration of conformable types (➔ polymorphism)
- Type inheritance (subtyping) as means for deriving types
  - ➔ Relationship between types becomes comprehendible
    - ➤ allows the identification of conformable types

8 Polymorphism

- Polymorphism = *the ability to take several forms*
  - various types for values or variables
  - various types for the parameters of functions
  - various types for the operands of an operator
- Example:

  operator “+” works with operands of type `int` and `real`
- Classification



9 Polymorphism in C++

- Overloading polymorphism
  - Function-name overloading
  - Operator overloading
- Inclusion polymorphism
  - public inheritance
- Coercion polymorphism
  - Cast operator

9 Polymorphism in C++ (2)

★ **Function-name Overloading**

- several functions with the same name but different signature within one scope
- compiler selects the proper function
  - ➔ static binding (*early binding*)
- Example:

```
int max (int a, int b) { return((a>b):a?b); }
float max (float a, float b) { return((a>b):a?b); }
char *max (char *a, char *b) {
        if ((strcmp(a,b)> 0) return a; else return b; }
```

- Special cases:
  - several constructors for a class
  - operator overloading
  - overloading of a method in the context of inheritance

9 Polymorphism in C++ (3)

★ **Cast Operator / Coercion Polymorphism**

- In most programming languages automatic type conversion for basic types (e.g. short $\rightarrow$ int $\rightarrow$ double $\leftarrow$ float)
- Cast operator (cast method) allows the definition of an explicit conversion of the class type into another type

```
class::operator type(void) { ... }
```

- Cast operator is like a normal unary operator

9 Polymorphism in C++ (4)

★ **Cast operator — Example**

```
class time
{
  private:
    int hour, min, sec;
  public:
    ...                             // Konstruktor

    operator long () {              // cast-Operator
      return(hour*3600 + min*60 + sec);
    }
};
main() {
  time now(1,10,2);
  long t;

  t = now + 10;                 // ⇒ t == 4212
...
```


9 Polymorphism in C++ (5)

★ Inclusion Polymorphism — *THE Polymorphism in OOP*

Inheritance + Virtual Methods + Object References

- Object reference (pointer) has a type (= class)
  - ◆ instances of this class and all of its subclasses may be assigned to the ref.
  - ◆ on method invocation, the actual implementation of the method is designated not by the class of the pointer but by the class of the current object

➜ You may always assign any type-conformable object to an object reference and everything will work

- ◆ you may pass it as parameter to a method
- ◆ the programmer of the method did not need to know anything about your new subtype
  — as long as it conforms to the supertype his method expects

9 Polymorphism in C++ (6)

★ **Virtual Methods & Inclusion Polymorphism — Example:**

```
class geo_obj {                          // general superclass
  public:
    virtual void draw();
};

class circle : public geo_obj {          // subclass
  public:
    void draw();
}

class square : public geo_obj {          // subclass
  public:
    void draw();
}

main () {
  geo_obj *ptr;
  ptr = new circle;
  ptr->draw();
...
```


10 Types & C++ Abstract Classes

- Superclass declares methods and its signatures but does not define them
  - – **pure virtual functions**
  - ➜ superclass defines only a type
- Subclasses define various implementations of the methods
  - ➜ each subclass is one implementation of the type
- Not possible to instantiate objects from the superclass
- Example:

```
class geo_obj {                       // abstract calss
  public:
    virtual void draw() = 0;          // pure virtual function
};
class circle : public geo_obj {       // subclass
  public:
    void draw() { ... }
}
```

11 Types & Java
Abstract Classes

- As in C++
  - instantiation not possible
  - missing parts have to be completed in the subclass
- Example:

```
abstract class geo_obj {              // abstract class
  public abstract void draw();        // pure virtual function
};
class circle extends geo_obj {        // subclass
  public void draw() { ... }
}
```


12 Types & Java
Interfaces

- 2 possibilities to declare a type
  - ◆ by a class definition
    - ➔ class inheritance automatically leads to type inheritance
  - ◆ by a interface declaration
    - ➔ separate type declaration
- Example:

```
public interface Printable {
    public void Print();
}

public class MyPoint extends Object implements Printable {
    ....
    public void Print() {
        System.out.println("x="+x+" y="+y);
    }
}
```


12 Types & Java
Interfaces (2)

- Inheritance + multiple inheritance for interfaces
- One class may implement several types
- Type conformance is transitive
- Exceptions are an element of the type interface
- Examples:

```
interface Streamable extends FileIO, Printable {
    // additional Methods
    public void test() throws TestException;
}

class Test implements Streamable, Testinterface {
    ...
}
```


13 Genericity

Concept that allows the definition of the type of some unit of the programming language at invocation or instantiation by a parameter

- OOP: generic classes ➔
  - generic class ➔ instantiation (+ parameterization) ➔ actual class
  - actual class ➔ instantiation of objects
- Example:
  - ◆ general stack class
    - ➤ int stack
    - ➤ real stack
    - ➤ string stack
- Most benefit of genericity can also be achieved by inheritance
- Implemented in Ada, Eiffel and C++ (*Templates*)

14 Genericity & C++: Templates

- Objective: definition of a class without finalization of types
  - ➤ dynamic type checking at run time or
  - ➤ static type checking at compile time + parameterizable classes
- Template = parameterizable class
- Example:

```
template <class T> class stack {
  private:
    int index;
    T *array;
  public:
    void stack(int n)
              { index = 0; array = new T[n]; }
    void push(T elem)
              { array[index++] = elem; }
    T pop(void)
              { return(array[index--]); }
};
```


15 Concurrency

Several threads of control are processed in parallel on different processors or quasi-parallel on one single processor

- Concurrency is orthogonal to object orientation (in general)

  but: concurrent solutions are much more complex than sequential ones
- Granularity: concurrency / objects (capsules)
  - ➤ Finer grained concurrency
    - ➜ even object-internal concurrency
  - ➤ Coarser grained concurrency
    - ➜ just object-external concurrency
- Integration of concurrency control into OO programming languages
  - ➤ orthogonal languages
  - ➤ non-orthogonal languages
    - – uniform / non-uniform languages

14 Genericity & C++: Templates

- Parameterization of class on object instantiation
- Examples:
  - ◆ Instantiation and usage of a stack with elements of type float:

```
stack <float> s(10);
float a;
s.push(a);
a = s.pop( );
```

  - ◆ Instantiation and usage of a stack with elements of type integer:

```
stack <int> s(10);
...
```


16 Concurrency & Java

- Thread concept and synchronization mechanisms are integrated
  - ➥ non-orthogonal, non-uniform language
- Creation of threads via thread class
- Example:

```
class MyClass implements Runnable {
        public void run()  {
                System.out.println("Hello\n");
        }
}

....
MyClass o1 = new MyClass();    // create object
Thread t1 = new Thread(o1);    // create thread to run in o1

t1.start();                    // start thread

Thread t2 = new Thread(o1);    // create second thread to run in o1

t2.start();                    // start second thread
```

16 Concurrency & Java (2)

★ **Synchronization**

- Monitors: exclusive execution of methods of an object
  - ◆ Example:

```
class Bankkonto {
    int value;
    public synchronized void AddAmmount(int v) {
        value=value+v;
    }
    public synchronized void RemoveAmmount(int v) {
        value=value-v;
    }
}
...
Bankkonto b=....
b.AddAmmount(100);
```

- Conditions: releasing a monitor while waiting on an event

17 Persistence

★ **Motivation for Persistence** [ABC83]

- “active” data → programming language facilities / run-time environment
- “passive” data → DBMS or file system

➔ 2 different views of data

➔ Disadvantages for the programmer
  - ➤ Conversion between active and passive data necessary
  - ➤ Data type protection of programming language is lost

17 Persistence (2)

★ **General Definition**

- Persistence is the property of data through which its existence transcends time
  (i. e. it continues to exist after its creator ceases to exist)
  and/or space
  (i. e. its location moves from the address space in which it was created).

★ **Spectrum of Persistence**

1. Transient results in expression evaluation
2. Local variables in procedure activations
3. Global variables and heap items whose extent is different from their scope
4. Data that exists between executions of a program
5. Data that exists between various versions of a program
6. Data that outlives the program

17 Persistence (3)

★ **Persistence in Object-oriented Systems**

- Objects survive the termination of the environment (thread, application execution) in which they were instantiated or used
- Powerful mechanism in OO operating systems for
  - ➤ Data storage
  - ➤ Data transport
- Examples
  - ➤ File systems
  - ➤ Database systems
  - ➤ Persistent communication objects
- Properties of the object-oriented programming model are automatically inherited by all mechanisms which are constructed on its basis