Programming Paradigms
Unit 2 — Imperative and Object-oriented Paradigm

J. Gamper

Free University of Bozen-Bolzano
Faculty of Computer Science
IDSE
Outline

1. Imperative Programming Paradigm
2. Abstract Data Types
3. Object-oriented Approach
Outline

1. Imperative Programming Paradigm
2. Abstract Data Types
3. Object-oriented Approach
The imperative paradigm is the oldest and most popular paradigm. Based on the von Neumann architecture of computers, imperative programs define sequences of commands/statements for the computer that change a program state. Commands are stored in memory and executed in the order found. Commands retrieve data, perform a computation, and assign the result to a memory location.

The hardware implementation of almost all computers is imperative. Machine code, which is native to the computer, and written in the imperative style.

Machine code:
8B542408 83FA0077 06B80000
C9010000 008D0419 83FA0376
B84AEBF1 5BC3
Central elements of imperative paradigm:

- **Assignment statement:** assigns values to memory locations and changes the current state of a program
- **Variables** refer to memory locations
- **Step-by-step** execution of commands
- **Control-flow statements:** Conditional and unconditional (GO TO) branches and loops to change the flow of a program

Example of computing the factorial of a number:

```c
unsigned int n = 5;
unsigned int result = 1;
while(n > 1) {
    result *= n;
    n--;
}
```
Procedural Programming

- **Procedural programming** is a refinement of the imperative paradigm adding subroutines (or procedures)
  - Procedures can be used the same way that built-in commands are used (allows re-usability)
  - Some state changes are localized in this way
- Creating a procedure from the previous example:

  ```c
  int factorial(unsigned int n) {
      unsigned int result = 1;
      while(n > 1) {
          result *= n;
          n--;
      }
      return result;
  }
  ```
EARLIEST IMPERATIVE LANGUAGES WERE THE MACHINE LANGUAGES OF THE COMPUTERS

- Very simple instructions
- Made hardware implementation easier, but difficult to create complex programs

IN 1954, FORTRAN WAS DEVELOPED BY JOHN BACKUS AT IBM

- First major programming language
- Removed problems of machine code for the creation of complex programs
- Many features that are common in imperative languages, e.g., named variables, complex expressions, subprograms, etc.
- FORTRAN was a compiled language

THE NEXT TWO DECADES SAW THE DEVELOPMENT OF A NUMBER OF OTHER MAJOR HIGH-LEVEL IMPERATIVE PROGRAMMING LANGUAGES

- In the late 1950s, ALGOL (short for ALGOarithmic Language) was developed in order to allow mathematical algorithms to be more easily expressed
History of Imperative Paradigm/2

- COBOL (1960) and BASIC (1964) were both attempts to make programming syntax look more like English.
- In the 1970s, Niklaus Wirth at ETH Zurich developed Pascal as a small and efficient language intended to encourage good programming practices using structured programming and data structuring.
- C was created by Dennis Ritchie at Bell Laboratories.
  - Used to (re-)implement the Unix operating system.
  - Has become one of the most widely used programming languages of all time.
In the 1980s, there is a rapid growth in interest in object-oriented programming

- Imperative in style, but added features to support objects

Simula was the first OO-language, and influenced other languages (1960s)
Smalltalk-80 was released in 1980 by the Xerox Palo Alto Research Center
Bjarne Stroustrup designed C++, an object-oriented language based on C

OO languages in the late 1980s and 1990s:

- Perl (Larry Wall in 1987)
- Python (Guido van Rossum, 1990)
- Visual Basic and Visual C++ (Microsoft, 1991 and 1993)
- PHP (Rasmus Lerdorf, 1994)
- Java (Sun Microsystems, 1994)
- Ruby (Yukihiro Matsumoto, 1995)
Outline

1. Imperative Programming Paradigm
2. Abstract Data Types
3. Object-oriented Approach
Abstract Data Types

- The procedural approach in imperative programming was taken further by introducing **abstract data types (ADT)**
- In ADTs, everything related to a type is **encapsulated** in one bundle, most importantly
  - data itself and
  - operations on the data
- This **hides** the underlying representation and actual implementation (information hiding)
What are the advantages of information hiding?

- Allowing access to data only via a specified set of operations increases type safety
- An implementation of an ADT can be replaced by a different (more efficient) one without having to rewrite other parts of the code
- Code becomes more portable and easier to reuse
Limits of Data Abstraction

- While ADTs exhibit important features such as encapsulation and information hiding, there are still shortfalls.
- We will have a look at these with a concrete (though simplified) example.
  - Assume we want to define an ADT implementing a counter.
ADT Counter

This ADT provides a counter of type integer, which can be read, incremented, and reset.

abstracttype Counter {
  type
    Counter = int x;
  operations
    int get(Counter x) { return x; }
    void inc(Counter x) { x++; }
    void reset(Counter x) { x := 0; }
}
Assume we want to extend this type by adding an operation that tells us how many times we have reset the counter.

We could define a completely new ADT

abstracttype NewCounter {
  type
    NewCounter = struct {
        int c;
        int noOfResets = 0;
    }
  operations
    int get(NewCounter x) {
        ...
    }
    void inc(NewCounter x) {
        ...
    }
    void reset(NewCounter x) {
        ...
    }
    int howManyResets(NewCounter x) {
        ...
    }
}
ADT NewCounter/2

- In terms of encapsulation and information hiding this is fine
- However, we have to redefine and re-implement all operations, even though most of them work exactly the same way as in Counter
- Gets worse if we want more extensions to the type Counter or NewCounter
- Adding more types leads to redundancy
  - This causes unnecessary work (and increases the size of the code)
  - More difficult to maintain, may result in inconsistencies
Another approach would be to **re-use the ADT Counter** when defining NewCounter

```plaintext
abstracttype NewCounter {
  type
    NewCounter = struct {Counter c; int noOf Resets = 0;}
  operations
    int get(NewCounter x) { ...};
    void inc(NewCounter x) { ...};
    void reset(NewCounter x) { ...};
    int howManyResets(NewCounter x) { ...};
}
```
ADT NewCounter/4

- What would the implementation of the operators looks like in this case?
- Re-uses the implementation of Counter:

```c
int get(NewCounter x) {
    return get(x.c);
}
...
void reset(NewCounter x) {
    reset(x.c);
    x.noOfResets++;
}
...
This solution still has drawbacks

- We have to map new operators explicitly to old operators
- If we extend NewCounter again, an operator is mapped to the NewCounter operator, which is mapped to the Counter operator . . .
- It would be great if the derived ADT could just inherit the operators from the original ADT
Further problems with **typing**

Assume we have a group of counters, some of type `Counter` and some of type `NewCounter`

- If we want to store these in an array, what would the array look like?
- `Counter Z[20]` cannot store `NewCounter`
- `NewCounter Z[20]` cannot store `Counter`
When we introduced type systems, we briefly mentioned compatibility rules, i.e., one type can be substituted for another.

Let us define *type compatibility* between type $S$ and type $T$ in more detail:

- $T$ is compatible with $S$ when all operations over values of type $S$ are also possible over values of $T$.
- So NewCounter is compatible with Counter.

Substitutability allows us to use NewCounter whenever a Counter is expected.
Using substitutability we can define an array Counter $Z[20]$ and put counters of type `NewCounter` into it.

However, what happens if we run the following code?

```java
for(int i; i < 20; i++)
    reset(Z[i]);
```
During compilation a type checker will be satisfied, as `reset()` is a valid operation on `Counter`.

But which version of `reset()` will be executed?

If this is determined statically (e.g. during compilation), then:
- this will be `reset()` of `Counter`, as `Z` is of type `Counter`.
- As a consequence, `noOfResets` of `NewCounter` will not be updated.

In order to make this work we need to check the “true” type of a counter stored in `Z` and select the correct operator _dynamically_.

**ADT NewCounter**
Outline

1. Imperative Programming Paradigm
2. Abstract Data Types
3. Object-oriented Approach
Object-oriented Approach

In the **object-oriented paradigm**, all the previously mentioned issues are resolved

- There is **encapsulation** and **information hiding**
- Under certain conditions, **inheritance** of operator implementations is permitted
- Types can be **substituted** for one another if they are compatible
- Operators are selected **dynamically** depending on the actual type
Objects encapsulate both the data and the operations on the data (well, at least conceptually)

Operations are usually called methods and are sent as a message to an object

- This means, that the object receiving the message is an implicit parameter
- So for our example, the operations on Counter would not need any additional parameters
  - Assuming we have an object o implementing a counter, then we would increase it by calling

  o.inc();
Sending Messages
So a **Counter** object would look like this:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>get</td>
<td><code>return this.x;</code></td>
</tr>
<tr>
<td>inc</td>
<td><code>this.x++;</code></td>
</tr>
<tr>
<td>reset</td>
<td><code>this.x = 0;</code></td>
</tr>
</tbody>
</table>

However, when implementing object-oriented languages, the code for operations is not stored explicitly with every object.
A class acts as a blueprint for objects and basically defines a type

```cpp
class Counter {
private:
    int x;
public:
    int get();
    void inc();
    void reset();
}
```

- Parts declared private are not accessible from the outside
- Parts declared public are visible to all
So classes accomplish encapsulation and information hiding

In addition to this, `Counter` can be extended in a more elegant way

class NewCounter extending Counter {
    private:
        int noOfResets = 0;
    public:
        void reset() {
            x := 0;
            noOfResets++;
        }
        int howManyResets() {
            return noOfResets;
        }
}
We also have substitutability

- Every message understood by Counter objects is also understood by NewCounter objects

- NewCounter re-uses the methods `get()` and `inc()`
- It redefines the method `reset()`
  - This is also called **overriding**
- It also defines a new method `howManyResets()`
  - Clearly, additional methods do not pose a problem for substitutability
Inheritance

- We also say that all the methods that are not redefined inherit their implementation from the superclass.
- Some languages (such as C++) allow multiple inheritance:
  - That means a class can have more than one superclass.
  - Can be problematic due to name clashes.
Substitutability and Inheritance

- Although some languages implement substitutability and inheritance using the same constructs, these are different concepts
  - **Substitutability** allows the use of an object in another context
    - Object does not have to be of a subclass to understand same methods
  - **Inheritance** allows the re-use of code (for methods)
    - Private inheritance in C++ re-uses code, but does not allow substitutability
Dynamic Method Lookup/1

- A method defined for one object can be redefined in objects belonging to other classes.
- That means there can be many versions of a method.
- In order to figure out which one to use, we have to look at the actual type of the object the message is sent to.
- This is also called dynamic dispatch.
**Dynamic Method Lookup/2**

- Looking at our Counter example in an object-oriented setting
  
  ```java
  for(int i; i < 20; i++)
      reset(Z[i]);
  ```

  gives us the correct results using dynamic dispatch

- Not to be confused with operator overloading, in which multiple versions of a method with different parameters can exist, e.g.,

  ```java
  void reset();
  void reset(int a);
  int reset();
  ```

- Correct method would be selected by matching its signature
Polymorphism

- Nevertheless, dynamic method lookup and operator overloading are different facets of **polymorphism**
- Polymorphism means that an object or method can have more than one form
- Yet another kind of polymorphism is **parametric polymorphism** or **generics**
  - Also called **templates** in C++
Generics consist of program fragments, where some types are indicated by parameters. These parameters can then be instantiated by “concrete” types. Depending on the generics, the type used for instantiation has to implement certain methods.
Generics Example

- Implement a stack without having to re-implement it for every possible data type of its content

```java
class Elem <A> {
    A content;
    Elem <A> next;
}
class Stack <A> {
    private:
        Elem <A> top = null;
    public:
        boolean isEmpty();
        void push(A object);
        A pop();
}
```
Summary

**Imperative paradigm** is the *oldest* programming paradigm, based on von Neumann architecture

- Program consists of **sequence of statements** that change the program state

**Procedural programming** is a refinement that makes it easier to write complex programs

**Machine languages** were the earliest imperative languages, followed by FORTRAN and ALGOL

**Abstract Data Types** is a further extension of imperative programming

- Data and operations are **encapsulated** into a bundle (**information hiding**)
- This hides the underlying representation and implementation

**Object-oriented paradigm** extends ADTs

- **Classes** are blueprints for objects that encapsulate both data and operations
- Objects exchange **messages**
- Provides **encapsulation**, **information hiding**, **inheritance**, and **dynamic dispatching**