Basic Concepts in Object Oriented Programming

Raúl Ramos-Pollán / IT User Support

It's about facing "complex" problems

We have **LIMITATIONS** ⟶ We use **TRICKS**

Abstraction + Decomposition + Organisation

Paradigm

Logic

Functional

Object Oriented

Java

C++

SmallTalk
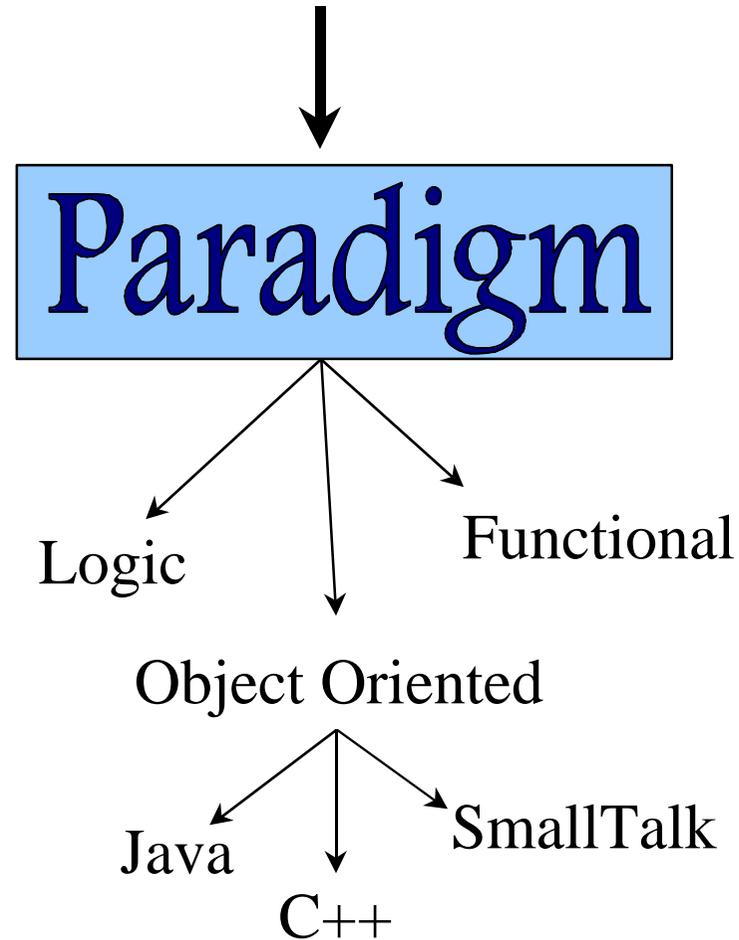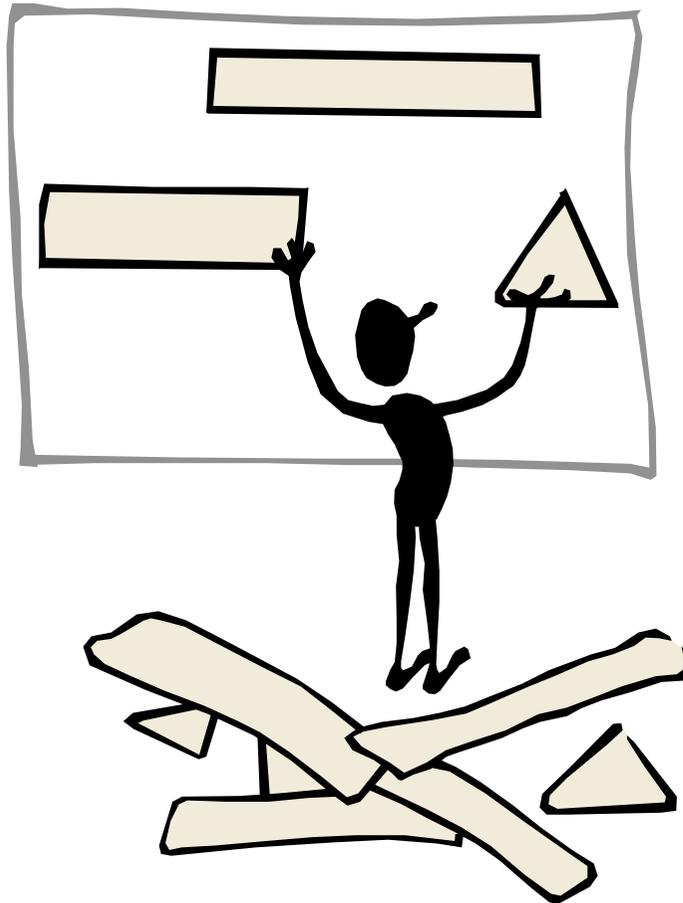
Functional Paradigm

- ## We think in terms of **functions** acting on **data**

  - ABSTRACTION: Think of the problem in terms of a process that solves it.

  - DECOMPOSITION: Break your processing down into smaller manageable processing units (functions).

  - ORGANIZATION: Set up your functions so that they call each other (function calls, arguments, etc.)

- ## FIRST: define your set of data structures (types, etc.)

- ## THEN: define your set of functions acting upon the data structures.

Object Oriented Paradigm

- We think in terms of objects interacting:

  - ABSTRACTION: Think in terms of independent agents (objects) working together.

  - DECOMPOSITION: Define the kinds of objects on which to split the global task.

  - ORGANIZATION: Create the appropriate number of objects of each kind.

- FIRST: Define the behavior and properties of objects of the different kinds we have defined.

- THEN: Set up objects of each kind and put them to work.

An Scenario

- We want to make an implementation to analyze data from physics events.

- Data is stored somewhere else (db, file, …)

- Our implementation must:

    – Provide data structures to hold data once it is organized in run, events, tracks, etc.

    – Provide the algorithms to:

        - Populate the data structures sources (db, file, …)

        - Manipulate the data structures to obtain: graphical representations, analysis, etc.

An Scenario

- With our implementation we will make a library which will be made available to other physicists so that they build their own programs to manipulate events.

WARNING

This is a VERY simplified scenario for educational purposes. Real life conceptualizations may differ in their philosophy and final implementations.

A Functional Approach

Data Definition:

```
Library EventLibrary;


structure Track {
    real coordinates[];    // Array of coordinates
    real angles[];         // Array of angles
}
structure Event {
    integer eventNumber;
    Track tracks[];        // Array of Tracks
}
structure Run {
    Time initTime;
    Time endTime;
    Event events[];        // Array of Events
}
```

A Functional Approach

Functions Definition

```
Library EventLibrary;
Track retrieveTrack(db, tID) { access db, fill arrays, ... }
void drawTrack(Track t) { . . . }


Event retrieveEvent(db, eID)
        { . . . while (i) tracks[i]=retrieveTrack(db, i); . . .}
void drawEvent(Event e) { for t in tracks drawTrack(t); }
real calculateFactor (Event e) {  … access tracks of e …  }
boolean isHiggs (Event e) { … analize tracks of e …  }


Run retrieveRun(rID) { … while (i) events[i]=retrieveEvent(db,i)… };
void analyseRun(Run r)
        { … c=0; for e in allEvents c=++calculateFactor(e); print c; …  };
Event searchForHiggs(Run r) { … access events of r … };
```

A Functional Approach

User Program:

```
use EventLibrary;
```

any function in our program, maybe the entry point (void main)
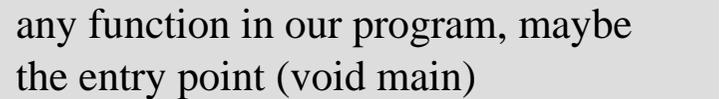
```
some_function (Event e) {
 … initialize database, etc …
  Run myRun = retrieveRun(our_db, 1045);


  // Do things with the event passed on to this function
  drawEvent(e);
  print "Factor for event 105 is " +calculateFactor(e);


  // Do things with the run
  if (searchForHiggs(myRun)!=null) print "found it!!!";
  analyzeRun(myRun);
}
```

A Object Oriented Approach

Run
Beginning Time
End Time

Event
number

· · ·

Event
number

Track
angles & coords

· · ·

Track
angles & coords

⟶ Means "is composed of", "contains"

OBJECTS DEFINITIONS
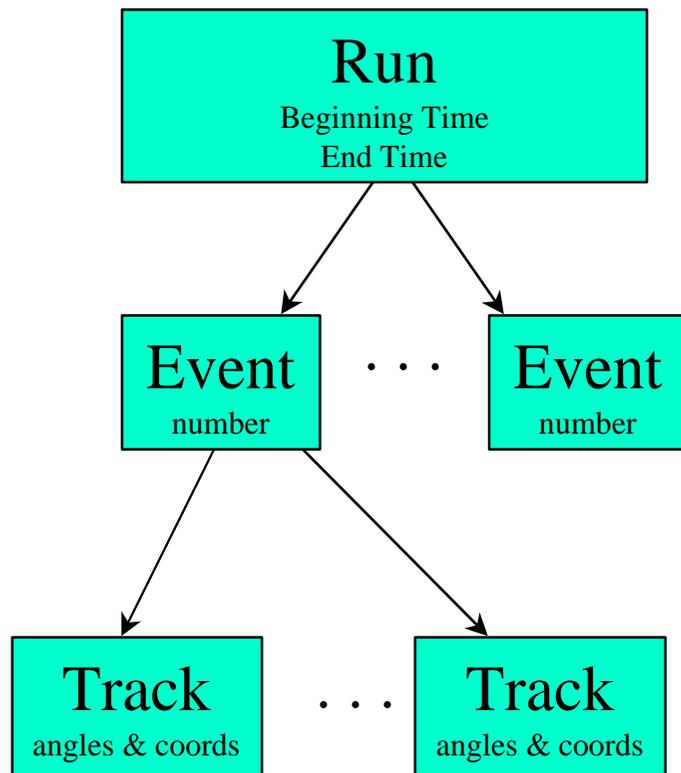
A **Track** contains a set of <u>coordinates</u> and <u>angles</u> and it's able to <u>draw</u> itself

An **Event** contains a set of <u>Tracks</u> and and it's able to <u>draw</u> itself, to <u>tell</u> if it signals a Higgs boson, and to <u>calculate</u> a factor about itself.

A **Run** contains a set of <u>Events</u> and it's able to <u>search</u> for the Higgs boson within its Events and to <u>calculate</u> some global figure from its Event's factors.

An Object Oriented Approach

- We have three kinds of objects:
  - Run, Event, Track

- We may have several objects of each kind.

- OO is about defining of objects not about defining processes.

- To define objects we have to define two things: **properties** (state) and **behavior**.

- In the previous informal definitions:
  - Track: coords+angles   draw
  - Event: tracks+nb   draw + calc.Factor + Higgs?
  - Run: events+times   search Higgs + calc. Global

Defining objects: STATE

Encompasses all the properties of an object.
Each property has a value or a reference to another object.
All objects of the same "kind" have the same properties
(although may have different values).

Properties are implemented with **_Object Variables_**

Every **_Track_** has a set of coordinates and angles.
Different Tracks contain different sets of coords. and angles.

Every **_Event_** has an event number and is composed of a set of tracks.
Different Events have different numbers and tracks.

Every **_Run_** is has a beginning and ending time and a set of events.
Different Runs have different times and events.

Defining objects: BEHAVIOR

Is how an object reacts, in terms of state changes and interaction with other objects.

It is defined with ***Object Methods***

Every ***Track*** can <u>draw</u> itself

A specific Track will draw itself according to its own data.

Every ***Event*** can <u>draw</u> itself (by ordering the tracks it contains to themselves), can <u>calculate a factor</u> about itself and <u>tell</u> if it contains a Higgs boson.

Every Event will perform this operations according to their own properties (tracks, number).

Every ***Run*** can <u>search for Higgs</u> bosons within its Events and <u>calculate</u> some global figure.

Defining objects: CLASSES

A CLASS is a set of objects that share the same properties and behavior. It is the intuitive notion of a "*kind*" of objects.

It's where **Variables** and **Methods** are defined

OO IS MAINLY ABOUT DEFINING CLASSES

An object which follows the definition of a class is said to be an INSTANCE of that CLASS.

Every **Track** is an instance of the   **Track CLASS**
Every **Event** is an instance of the   **Event CLASS**
Every **Run** is an instance of the   **Run CLASS**

OO Programming Languages

- Functional programming languages (C, Pascal, FORTRAN, etc.) provide mechanisms to manipulate the basic conceptualizations:

  - define functions, call functions, etc.

- OO Programming languages provide mechanisms to:

  - define classes: `CLASS { ...... }`

  - create instances: `new CLASSNAME`

  - etc.

- The following examples are in no particular OO programming language. Specific OO languages provide similar constructs.

CLASSES definition

```
Library EventLibrary;
CLASS Track {
  real coordinates[]; // Array of coordinates
  real angles[];      // Array of angles


  constructor Track(database db, int tID) {
        …
        access database
        fill in coordinates[] and angles[] arrays from db
        …
        other initialization code.
  }


  void method draw(){
        foreach i in coordinates[] {
                drawLineWithAngle (coordinate[i], angle[i]);
        }
  }
}
```

Two properties for every Track

Constructor is called whenever a new object (instance) of this class is created.

One thing every Track knows how to do

CLASSES definition

```
Library EventLibrary;


CLASS Event {
  Track tracks[]; // Array of tracks
  int eventNumber;


  constructor Event(database db, int eID) {
        … access db, retrieve numer of tracks …
        while (i) { tracks[i] = new Track(db, i);      }
  }


  void method draw() {
        forall t in tracks[] t.draw();
  }


  real method calculateFactor() { … access tracks, calculate … }
  boolean method isHiggs() { … access tracks, analize … }
}
```

To create new objects we use the **new** operator, and the class from where to take the definition for the new object. Arguments are defined in the Track constructor

We ask an object to do something. Since t is a Track, the definition for the Track CLASS must have an implementation for the draw method.

CLASSES definition

```
CLASS Run {
    Date begin, end;
    Event events[];                    // Array of events

    constructor Run(database db, int rID) {
        … access database, retrieve number of events
        while (i) { events[i] = new Event (db, i); }
    }

    Event method searchForHiggs() {
        … access events[], calculate,
            return Event object or null …
    }

    void method analize() {
        … c=0;
        for e in events[] { c=++e.calculateFactor(); }
        print c;
    }
}
```

We create new objects.

We ask an object to do something.

User Program

```
use EventsLibrary;
class Sample {
 some_method (Event e) {


   … initialize database, etc …
   Run myRun = new Run(our_db, 1045);



   // Do things with the event passed when calling this method
   e.draw();
   print "Factor for event 105 is " + e.calculateFactor();


   // Do things with the run
   if (myRun.searchForHiggs()!=null) print "found it!!!";
   myRun.analyze();


 }
}
```

Create an object. Notice that since in the Run constructor we create new Event objects which in turn create Track objects.

Ask objects to do things

Functional vs OO Approaches

- What we obtain is the same.

- We have basically reorganized the code.

- The difference in the calls signals the philosophies:
    - Functional: WE DO SOMETHING TO A DATA STRUCTURE **drawEvent(e); calculateFactor(e);**

    - OO: WE ASK AN OBJECT TO DO SOMETHING **e.draw(); e.calculateFactor();**

- OO Provides the mechanisms to:
    - Invoke the correct constructor when objects are created
    - Invoke the correct method when asking an object to do something

The First OO Principle

Encapsulation

- Hides the behavior of an object from its implementation
- Separates what an object looks like from how it does it implements its behavior.

Nobody but themselves knows how a **Track** draws itself or how an **Event** calculates its factor

Extending the Library

- We have distributed the library and we have people making programs with it.

- Now, in addition to the events we already have, there is a new kind of event which contains more data and a new algorithm for drawing based on this new data.

- We need to update the library.

Refining Classes

A ColoredEvent inherits all definitions from Event (variables and methods)

```
Library EventsLibrary;
CLASS ColoredEvent INHERITS FROM Event {
   real color;

   constructor ColoredEvent (database db, int eID) {
        … access db, retrieve color into color variable.
        super(db, eID);
   }

   void method draw() {
        setBgColor(color);
        … some other faster algorithm to draw the Tracks …
   }
}
```

A ColoredEvent constructor gets color data from the database and then does the same as the constructor Event.

The draw method substitutes completely the draw method from Event.

ALL OTHER DEFINITIONS FROM EVENT ARE PRESERVED.

Refining Classes

```
use EventsLibrary;
class Sample {
  some_method () {

    … initialize db …

    Event e1 = new Event (our_db, 105);
    e1.draw();
    e1.calculateFactor();


    ColoredEvent e2 = new ColoredEvent (our_db, 246);
    e2.draw();
    e2.calculateFactor();

  }
}
```

An Event as before

When invoking e2.draw() the system actually calls the draw method defined in the ColoredEvent class

When invoking e2.calculateFactor() method the system actually calls the method defined in the Event class

This resolution is made at run-time and the code to do it's placed in our program by the OO compiler

The Second OO Principle

Inheritance

- Mechanism by which a class (*subclass*) refines the behavior
  and properties of some other class (*superclass*).
- The subclass IS A superclass plus something else.

A **ColoredEvent** is an **Event** plus extra data and some redefinitions.

This is **reuse** of code.

Functional Approach 1

```
structure ColoredEvent {

    real color;

    integer eventNumber;

    Track tracks[];

}


ColoredEvent retrieveColoredEvent(db, eID) {

        … access db, retrieve color into color variable …

        Event e = retrieveEvent (db, eID);

        eventNumber = e.eventNumber;

        tracks      = e.tracks();

}

void drawColoredEvent(ColoredEvent e) {

    setBgColor(color);

    … some other faster algorithm to draw the Tracks …

}
```

If we want to change the data definition common to Event and ColoredEvent we have to change it in both.

Reuse of code means duplicating and copying data

We have to use a new function name.
We would never want to pass an Event to a drawColoredEvent function

Functional Approach 2

```
structure ColoredEvent {
    Event eventData;
    Track tracks[];
}
```

Now, we really have common data definitions.
This is reusing code.

Now we cannot reuse the retrieveEvent function because the way to access the tracks[] array is different for ColoredEvent and Event

```
ColoredEvent retrieveColoredEvent(db, eID) {
        … access db, retrieve color into color variable …
        while (i) eventData.tracks[i]=retrieveTrack(db, i);
}
    void drawColoredEvent(ColoredEvent e) {
        setBgColor(color);
        … some other faster algorithm to draw the Tracks …
    }
```

Remember the user program

```
use EventsLibrary;

class Sample {
 some_method (Event e) {
  . . .

   // Do things with the event passed when calling this method
   e.draw();
   print "Factor for event 105 is " + e.calculateFactor();


   . . .
 }
}
class YYY {
 some_other_method() {
    … initialize some other db …
    Event myEvent = new Event(my_db, 3509);
    Sample s = new Sample();
    s.some_method(myEvent);
 }
}
```

The method that the user
had implemented (abbreviated)

This calls the draw method
of the Event class

Now we have other method
calling the previous one

Create an Event and pass
it as parameter

Independently of the previous problems this is quite analogous in functional paradigms.

Inheritance Relations

```
use EventsLibrary;


class Sample {
 some_method (Event e) {
  . . .


  // Do things with the event passed when calling this method
  e.draw();
  print "Factor for event 105 is " + e.calculateFactor();


  . . .
 }
}
class YYY {
 some_other_method() {
    … initialize some other db …
    ColoredEvent myEvent = new ColoredEvent(my_db, 5690);
    Sample s = new Sample();
    s.some_method(myEvent);
 }
}
```

some_method is not changed

the actual "draw" method invoked is only known at run-time. In this example the ColoredEvent.draw method is the one invoked.

We pass a ColoredEvent. This is OK. From Inheritance every ColoredEvent is also an Event

The Third OO Principle

Polymorphism

- We can deal with objects without the need to know what exact class they belong to
- This is an extension of the inheritance concept

Sample.some_method just needs its argument to be an **Event** so it can also be an object of any class derived from **Event.** Actual methods are resolved at run time, by the OO mechisms.

This is Polymorphism

- Polymorphism can be though of as a consequence of inheritance.

- If you ask anyone if a dog is a mammal the answer is yes

- If you ask the system if a ColoredEvent is an Event the answer is always yes, including when you are passing parameters.

- This is very important. Look at the example again:

We have redefined an Event into a ColoredEvent **AFTER** the user created his Sample class.
His Sample class is now using ColoredEvent **WITHOUT ANY NEED TO CHANGE IT**

Another Example

```
CLASS Alien {
  String myName;

  constructor Alien (String Name) {
    myName = name;
  }
  public method WhoAreYou() {
    print "I'm alien "+myName;
  }
}


CLASS MarsAlien INHERITS FROM Alien {
  public method WhoAreYou() {
    super.WhoAreYou();
    print "from Mars";
  }
}


CLASS AlienKiller {
  public method KillAlien (Alien victim) {
    println "His Last Words:";
    victim.WhoAreYou();
  }
}
```

```
// Instance objects
a1 = new Alien ("Johny");
a2 = new MarsAlien ("Soujourner");
k  = new AlienKiller();


// Kill one alien
k.KillAlien(a1);
  >> His Last Words:
  >> I'm alien Johny


// Kill the other alien
k.KillAlien(a2);
  >> His Last Words:
  >> I'm alien Soujourner from Mars
```

OO Scope

OBJECT-ORIENTED **ANALYSIS**: Examines the requirements of a system or a problem from the perspective of the classes and objects found in the vocabulary of the problem domain

OBJECT-ORIENTED **DESIGN**: Architectures a system as made of objects and classes, specifying their relationships (like inheritance) and interactions.

OBJECT-ORIENTED **PROGRAMMING**: A method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes.

SUMMARY

- ## We have seen that:

  - OO is about defining object classes and instantiating objects from those classes.

  - A class is data definitions TOGETHER with code.

  - The three OO principles:

    - ENCAPSULATION + INHERITANCE + POLYMORPHISM
    - Allow for clean CODE REUSE
    - Allow for cxlean CODE INDEPENDENCE

  - OO Provides more kinds of building blocks to build complex maintainable structures.

What's good about OO

- Code reuse and uniqueness by inheritance & encapsulation

- Maintainability: changes in a superclass code are "seen" by all subclasses, ensuring uniqueness of code.

- Independence of code by encapsulation. Implementations of objects do not interfere among themselves.

- Independence through polymorphism.

- High degree of organisation and modularity of the code. This fits the needs of large projects.

- Makes you think before putting your "hands on". Fast development is "self-organised". Good for prootyping.

What's bad about OO

• Compiled programs are usually larger since we need to implement inheritance resolution at run time. This is typically done by producing look-up tables for for methods and objects.

• Compiled programs may be slower because inherited code has to be looked up when called from subclasses. In C++, calling a method is as fast as calling a function in C, because there is more information in the lookup tables produced by the compilers.

Advantages LARGELY overcome disadvantages: Optimised Compilers, Spread of Use, lots of Libraries of Classes …

MOST OF THE LARGE SW BEING DEVELOPED IS OO !!

More Documentation

UCO Books:

Grady Booch, *Object-Oriented Analysis and Design*, Addison-Wesley

Bertrand Meyer, *Object-Oriented Software Construction*, Prentice Hall

Any OO Programming Language tutorial usually includes a OO overview:
Java, C++

Software Development Tools at CERN:

http: www cern ch PTTOOLS