Advanced OOABL

Agenda

- History of OOABL
- OO Basics
- Pattern
- Exercises

History of object oriented ABL

- OpenEgde ABL was developed in the 80s
- Various paradigm shifts in the last 30 years
- In the beginning purly procedural architecture
- Version 7 – Event driven UI architecture
- Version 9 - PUBLISH, SUBSCRIBE SUPER procedures
- Version 10.1 – First real OO implementation
- OpenEdge makes it possible, to combine all programming styles

Progress OpenEdge versions

- 10.1A – First version with OO implementation
- 10.1A - Basics: Classes, Methods
- 10.1B - Properties, Interfaces, USING statement
- 10.1C – First dynamic OO statements, static class members, properties in Interfaces
- 10.2A - .NET Bridge , Garbage Collection
- 10.2B - Abstract classes , events , expanded .NET support
- 10.2B - OO implementation ready for (productiv/commercial) use
- 11.4 - Serialization
- 11.6 - Enumerations, reflection

Agenda

- History of OOABL
- OO Basics
  - Classes
  - Attributes
  - Methods
  - Constructors
  - Destructors
  - Inheritance
  - Overriding
  - Overloading
  - Interfaces
  - Events
- Pattern
- Exercises

Definition: OOP

Object oriented programing (OOP)

- Structuring according to human thinking
- Depicting things through classes and objects
- Interplay of cooperating objects

Definition: classes

- Are blueprints for real-world constructs
- Have state (attributes) and behavior (methods)
- A class defines the properties and functionality of objects at compiling
- Classes can be seen as a kind of template for objects
- Templates are used to create objects at runtime

What is an object?

- Instance (concrete implementation) of a class
- Attributes have their own values
- Objects only exist at runtime of the program
- An object has precisely defined properties and functionality and interacts with other objects in a defined way

Create objects

- The keyword **NEW <classname>()** creates a new object of a certain class and returns it as an object reference (handle).

```
DEFINE VARIABLE oParrot AS Bird NO-UNDO.
oParrot = NEW Bird(“Cora”).
```

Constructor

- The keyword **CONSTRUCTOR** defines a constructor of a class
- The constructor is used to initialize the object and is always called when it is created
- A constructor can define parameters that must be passed when it is created
- Each class can define multiple constructors with different signatures
- Constructors always have the same name as the class (without package)

```
CONSTRUCTOR PUBLIC Bird(cName AS CHARACTER):
  THIS-OBJECT:cName = cName.
END CONSTRUCTOR.
```

Destructor

- An object can be explicitly deleted using **DELETE OBJECT <object reference>**
- The garbage collector handles implicit deletion
- As soon as no part of the program has a valid reference to a specific object instance, the object is automatically deleted by the garbage collector
- Destructors are always PUBLIC

Attributes: variable

- An attribute, like a variable, holds a value of a specific data type
- The names of the attributes are always unique within a class
- Variable values can be read and set at any time

```
DEFINE VARIABLE oKonf AS Konfiguration NO-UNDO.

oKonf = Konfiguration:oInstance.
```

Attribute: property

- Properties are variables encapsulated in a class
- Describe properties or characteristics of the class
- Program code can be executed when setting the value as well as when reading the value

```
DEFINE PUBLIC STATIC PROPERTY oInstance AS configuration
    PUBLIC GET():
        IF oInstance = ? THEN
            oInstance = NEW configuration().
        RETURN oInstance.
    END GET.
    PRIVATE SET.
```

Methods

- Methods are used to implement an object's actions or capabilities
- Like functions in procedures, methods can define parameters to be passed at execution
- The names of the methods are not unique, but can be used multiple times as long as the parameters differ

Methods: implementation

```
/*Bird.cls*/
CLASS Bird:
    DEFINE VARIABLE cText AS CHARACTER NO-UNDO.

    METHOD VOID SetText(pcText AS CHARACTER):
        cText= pcText.
    END METHOD.

    METHOD CHARACTER GetText():
        RETURN(cText).
    END METHOD.

END CLASS.

/*Story.p*/
DEFINE VARIABLE oMethod AS Bird NO-UNDO.
oMethod = NEW Bird().
oMethod:SetText("tschirp tschirp").
MESSAGE oMethod:GetText() VIEW-AS ALERT-BOX.
```

- **METHOD** {return-type} method-name (<Param1>,<Param2>, ...): method-body
- Methods without a return value use the data type **VOID**

Visibilities

```
CLASS Bird:
    METHOD PUBLIC CHARACTER GetText():
        RETURN(cText).
    END METHOD.
END CLASS.
```

Visibilities

- Regulate access to items
- Visible: Element can be read/written
- Not visible: Element cannot be read and changed
- Within a class, all elements of the class are always visible

Overview keywords: visibilities

- PUBLIC
  - The member can be accessed from anywhere
  - Defines the interface of the class of objects
- PRIVATE
  - The member can only be accessed by an object instance of the defining class itself
- PROTECTED
  - The member can only be accessed by an object instance of the defining class itself or a derived class

Static members

- **STATIC** keyword applicable to attributes and methods
- Attribute or method is defined for the class, not for the object
- Instantiated when a static member in the class is referenced or a dynamic object instance is created
- Cannot call "super" methods

Static members, final class

```
CLASS of1.const.urban-colors FINAL:

    DEFINE PUBLIC STATIC PROPERTY BGColorComponent        AS CHARACTER NO-UNDO   /*   */
        GET.
        PRIVATE SET.

    DEFINE PUBLIC STATIC PROPERTY FGColorComponent        AS CHARACTER NO-UNDO   /*  */
        GET.
        PRIVATE SET.

    CONSTRUCTOR STATIC urban-colors (  ):
        BGColorComponent        = of1.const.urban-colors:white.
        FGColorComponent        = of1.const.urban-colors:greyish-brown.
    END CONSTRUCTOR.

    METHOD STATIC PUBLIC CHARACTER getInvertedColor(cColor AS CHARACTER):
        DEFINE VARIABLE iR   AS INTEGER   NO-UNDO.
        DEFINE VARIABLE cInv AS CHARACTER NO-UNDO.

        iR = INTEGER(ENTRY(1,cColor)).
        RETURN cInv.
    END METHOD.
END CLASS.
```

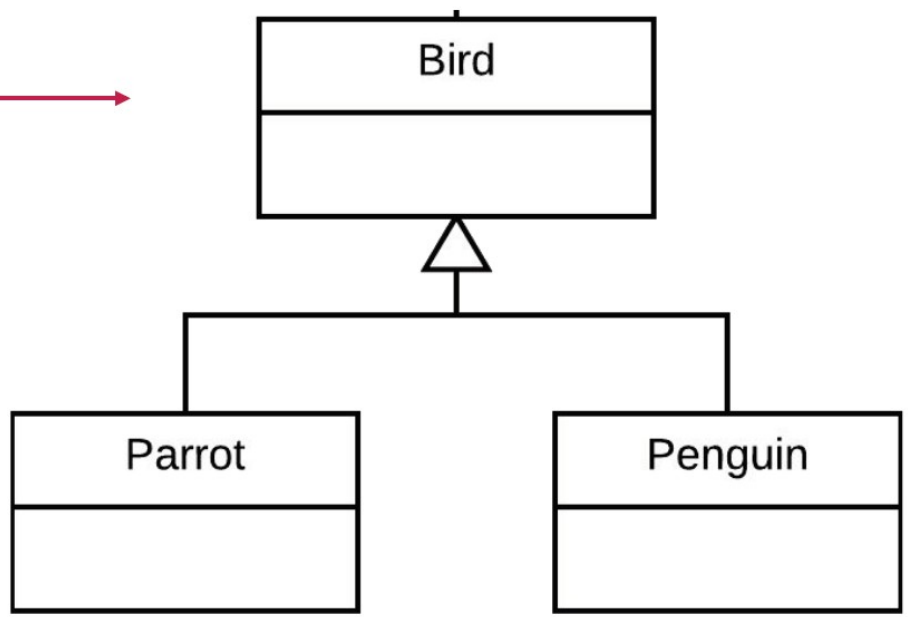
- **FINAL** keyword applicable to classes, attributes, and methods
- Final methods cannot be overridden
- Final classes cannot be upgraded

Why inheritance?

Inheritance

- A parrot is a bird
- A penguin is a bird

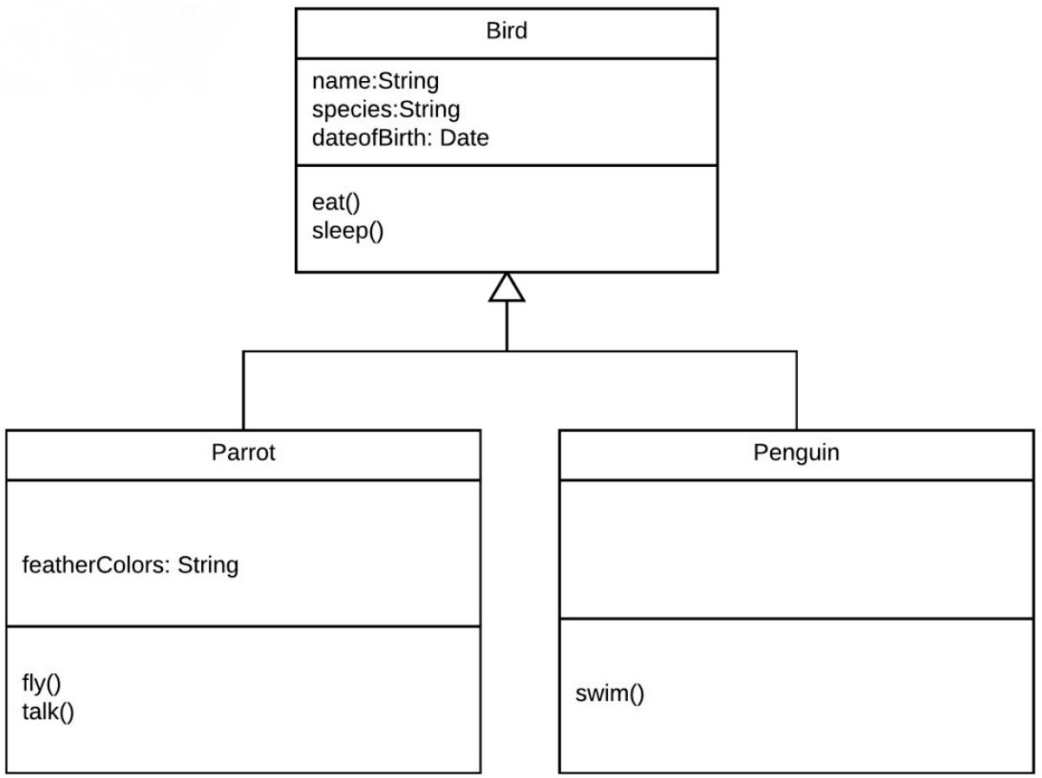
Superclass
(al(more er)
abstract)



Subclass
(more concrete)

Superclass Bird

- Attributes and methods are inherited from the superclass to the subclass(es).
- Subclasses can add their own methods and attributes



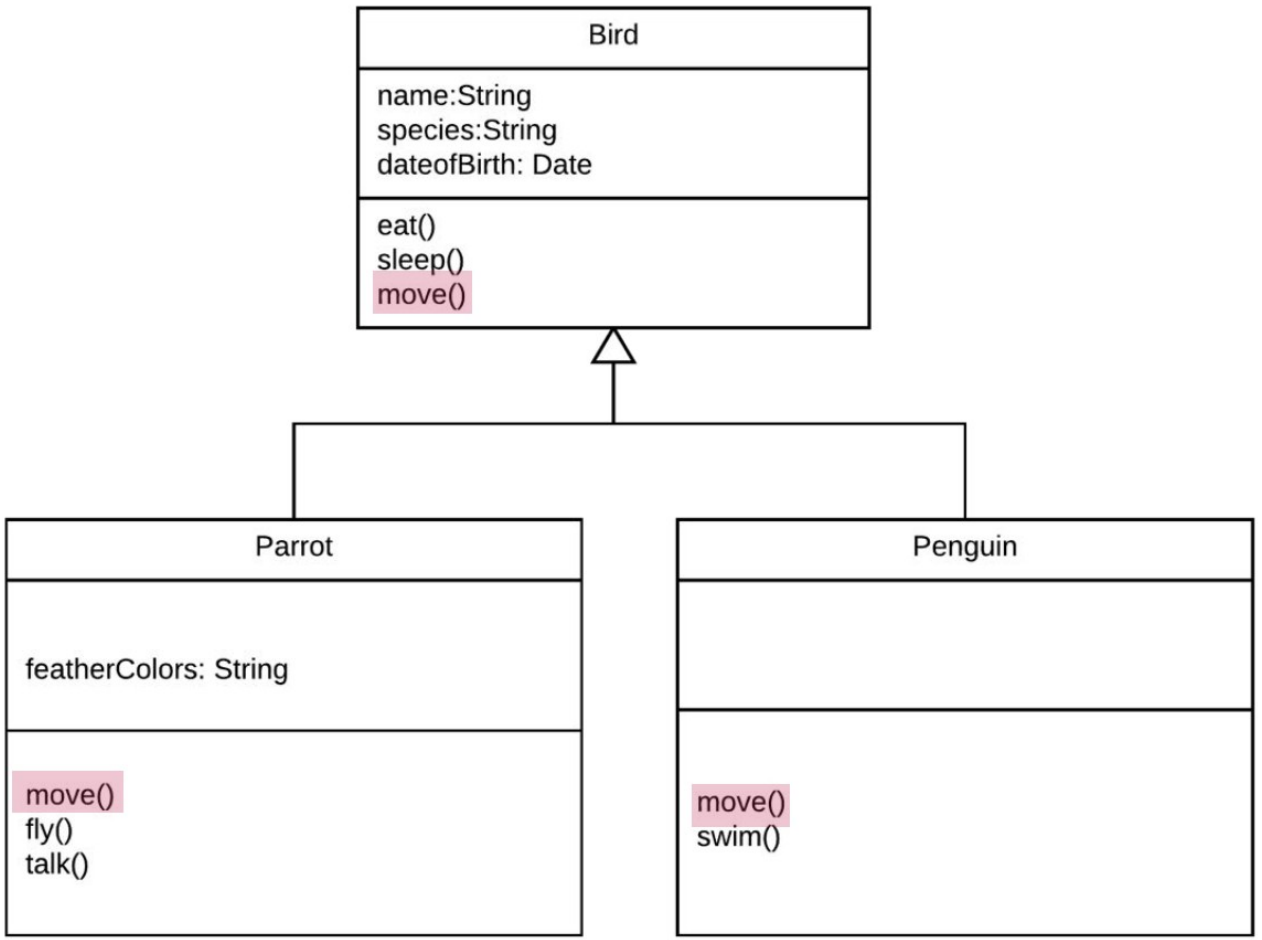
Subclasses - keyword: inherits

```
CLASS Bird:
    DEFINE PUBLIC PROPERTY cName AS CHARACTER NO-UNDO GET. PRIVATE SET.

    METHOD PUBLIC VOID sayHello( ):
    END METHOD.

END CLASS.

CLASS Parrot INHERITS Bird:
END CLASS.
```

- The derived class inherits all PUBLIC and PROTECTED members of the superclass
- Each class can derive from exactly one other class using **INHERITS**
- Note: If no superclass is specified, a class automatically inherits from the class Progress.Lang.Object (Ultimate Superclass). All classes therefore basically have all the properties of the Progress.Lang.Object class

Method overriding

- Subclasses can override methods and thus change inherited behavior individually



Overriding

```
CLASS Bird:
    DEFINE PUBLIC PROPERTY cName AS CHARACTER NO-UNDO GET. PRIVATE SET.
    METHOD PUBLIC VOID sayHello( ):
        MESSAGE "tschirp tschirp" VIEW-AS ALERT-BOX.
    END METHOD.
END CLASS.

CLASS Parrot INHERITS Bird:
    METHOD OVERRIDE VOID sayHello():
        MESSAGE "Cora wants peanuts" VIEW-AS ALERT-BOX.
    END METHOD.
END CLASS.
```

- Overriding class properties
  - To do this, the method is defined again in the derived class and must be provided with the keyword **OVERRIDE**
  - The implementation of the super class is not lost. It can be invoked using the **SUPER** keyword

Overloading of methods

```
METHOD PUBLIC INTEGER Sum(INPUT x AS INTEGER, INPUT y AS INTEGER):
    RETURN x + y.
END METHOD.

METHOD INTEGER Sum(INPUT x AS INTEGER, INPUT y AS INTEGER, INPUT z AS INTEGER):
    RETURN x + y + z.
END METHOD.

METHOD PRIVATE DECIMAL Sum(INPUT x AS DECIMAL, INPUT y AS DECIMAL):
    RETURN x + y.
END METHOD.
```

- Different methods have the same name within a class
- Has **nothing** to do with inheritance
- **Must** have different parameter lists
- Return types can be different
- Visibility may vary

Motivation abstract classes

```
oPenguin = NEW Penguin().
```

```
oBird = NEW Bird().
```

```
oAnimal = NEW Animal().
```

?

- Some classes shouldn't be instantiable!

Abstract Classes

```
CLASS Animal ABSTRACT:
    DEFINE PUBLIC ABSTRACT PROPERTY cName AS CHARACTER NO-UNDO
        GET.
        PROTECTED SET.

    METHOD PUBLIC ABSTRACT VOID eat( ).

END CLASS.
```

- The keyword **ABSTRACT** in the CLASS statement is sufficient to define an abstract class
- An object cannot be created from an abstract class

Abstract methods

```
CLASS Bird ABSTRACT INHERITS Animal:

    METHOD PUBLIC ABSTRACT VOID fly( ).

END CLASS.
```

- Are created by the keyword **ABSTRACT**
- Have no method body and end with a period
- Must be overridden (and thus defined) by the first concrete subclass
- Abstract methods can only exist in abstract classes

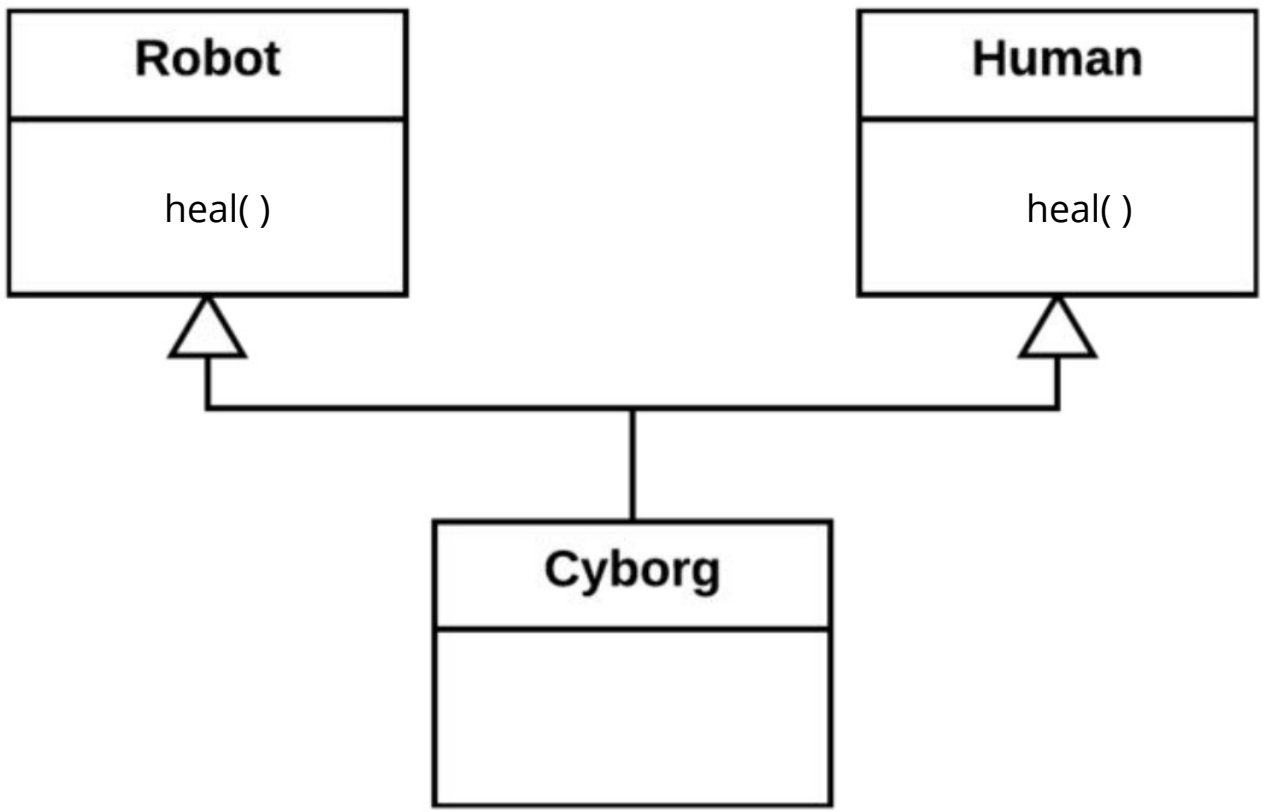
First concrete subclass

```
CLASS Parrot INHERITS Bird:
    DEFINE OVERRIDE PUBLIC PROPERTY cNAME AS CHARACTER NO-UNDO
    GET.
    SET.

    METHOD OVERRIDE PUBLIC VOID eat(  ):
    ...
    END METHOD.

    METHOD OVERRIDE PUBLIC VOID fly(  ):
    ...
    END METHOD.

END CLASS.
```

- In addition to abstract properties, abstract classes can contain completely implemented properties as well as constructors and destructors

principle of multiple inheritance

- A subclass Cyborg inherits from (potentially) different inheritance hierarchies
- The Human and Robot classes define the heal( ) method in different ways
- Which of the implementations does the cyborg inherit if heal( ) is not redefined?



Solution: interfaces

- Why did we want multiple inheritance?
- → It should be ensured that both the methods of the Human class and the methods of the Robot class are present

- Interfaces
  - ABL does not support multiple inheritance
  - Interfaces provide a “contract” that the implementing class must abide by
  - The properties are not implemented, only specified in the form of names and signatures

Interfaces

```
/*Healable.cls*/
INTERFACE IHealable:

  METHOD PUBLIC VOID healObj ( ).
  METHOD PUBLIC VOID healObj (INPUT pcWounds AS INTEGER).
  METHOD PUBLIC VOID logObj (INPUT pcFilename AS CHARACTER).

  DEFINE PUBLIC PROPERTY Name AS CHARACTER NO-UNDO
    GET.
    SET.

END INTERFACE.
```

- Interfaces do not contain any executable code
- Only PUBLIC is allowed as visibility
- Interfaces contain no constructor and no destructor
- An object cannot be created from an interface

Interfaces

```
CLASS Cyborg IMPLEMENTS IHealable:
    /* Must implement methods defined in the IHealable interface */

    METHOD PUBLIC VOID healObj( ):
    ...
    END METHOD.

    /* Second version of healObj */
    METHOD PUBLIC VOID printObj (INPUT piWounds AS INTEGER):
    ...
    END METHOD.

    /* Method to log information */
    METHOD PUBLIC VOID logObj (INPUT pcFilename AS CHARACTER):
    ...
    END METHOD.

    DEFINE PUBLIC PROPERTY Name AS CHARACTER NO-UNDO
        GET():
        END GET.
        SET():
        END SET.

END CLASS.
```

- Each class can implement multiple interfaces
- All prototypes must be implemented

Enumerations

- Enums
  - Own data type for variable with a finite value set
  - For example, as a list "RED, GREEN, BLUE", or min-max values
  - Realisation within cls file

```
// const.objstatus.cls
ENUM const.objstatus:
  DEFINE ENUM
    Offer       = 1
    Order       = 2
    Calculated  = 3
    Completed   = 8.
END ENUM.
```

Enumerated type

- Makes it easier for the developer to deal with constants and code

```
// So far
bObjekte.ObjStatus = 1.

// Assignment
bObjekte.ObjStatus = objstatus:Offer:GetValue().

//Unfortunately, this is not possible because different data types are used.
bObjekte.ObjStatus = objstatus:Offer.
```

- Are treated like regular data types

```
DEFINE PUBLIC PROPERTY iObjStatus AS const.objstatus NO-UNDO
    GET.
    SET.

iObjStatus = objstatus:Offer.
```

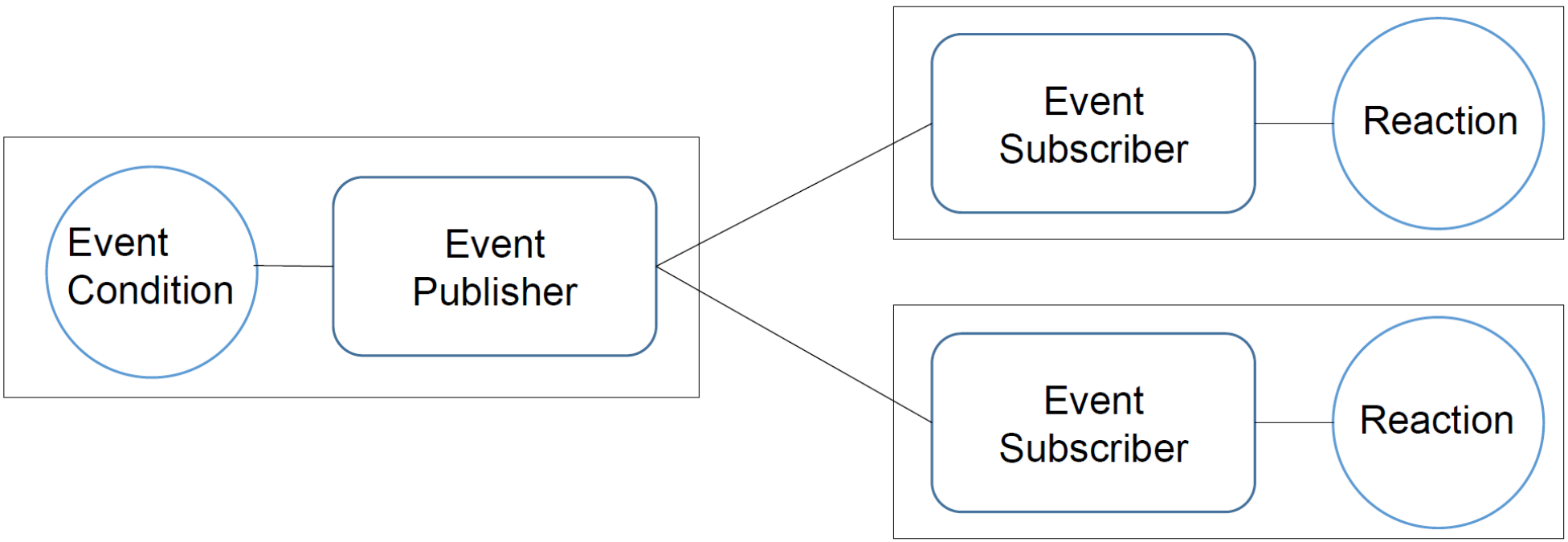
Events

- Events are used to perform actions in specific, defined situations
- Events are defined in classes with DEFINE EVENT
- The definition provides a signature in the form of parameters
- The names of the events are always unique within a class.

```
DEFINE PUBLIC EVENT SimpleEvent SIGNATURE VOID(p_msg AS CHAR).
```

Events

- Publish & Subscribe



Events

- Publish
  - Triggering an event is done by **<Event>:PUBLISH(<Param1>,<Param2>, ...)**.

```
/* Event Definitions */
DEFINE PUBLIC EVENT SimpleEvent SIGNATURE VOID(p_msg AS CHAR).

/* Event Publishers */
THIS-OBJECT:SimpleEvent:Publish(p_msg).
```

Events

- Subscribe
  - If an object wants to register for a specific event, it registers one of its methods with a corresponding signature using **<Event>:SUBSCRIBE(<Method>)**

```
/* Event Definitions */
DEFINE PUBLIC EVENT SimpleEvent SIGNATURE VOID(p_msg AS CHAR).

/* Event Subscriber */
oPublishingClass:SimpleEvent:Subscribe(oSubscribingClass:EventMethod).
```

Agenda

- History of OOABL
- OO Basics
- Pattern
  - Builder
  - Singleton
  - Lazy Loading
  - Adapter
- Exercises

Design Pattern

- „In software engineering, a software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design.“ -Wikipedia

- Design Patterns: Elements of Reusable Object-Oriented Software
- Three types:
  - Production patterns
  - Structure pattern
  - Behavioural patterns

Pattern: Builder

- Type: Production patterns
- Prepare creation of one object by another
- When:
  - Constructors with many parameters
- Why:
  - Readability
  - Fewer errors with parameters
  - Auto-complete
  - Easy to expand

Pattern: Builder

Starting point

User
- cFirstName: CHARACTER - cLastName: CHARACTER - iAge: INTEGER - cPhone: CHARACTER - cAddress: CHARACTER
+ User(cFirstName: CHARACTER, cLastName: CHARACTER) + User(cFirstName: CHARACTER, cLastName: CHARACTER, iAge: INTEGER) + User(cFirstName: CHARACTER, cLastName: CHARACTER, iAge: INTEGER, cPhone: CHARACTER) + User(cFirstName: CHARACTER, cLastName: CHARACTER, iAge: INTEGER, cPhone: CHARACTER, cAddress: CHARACTER)

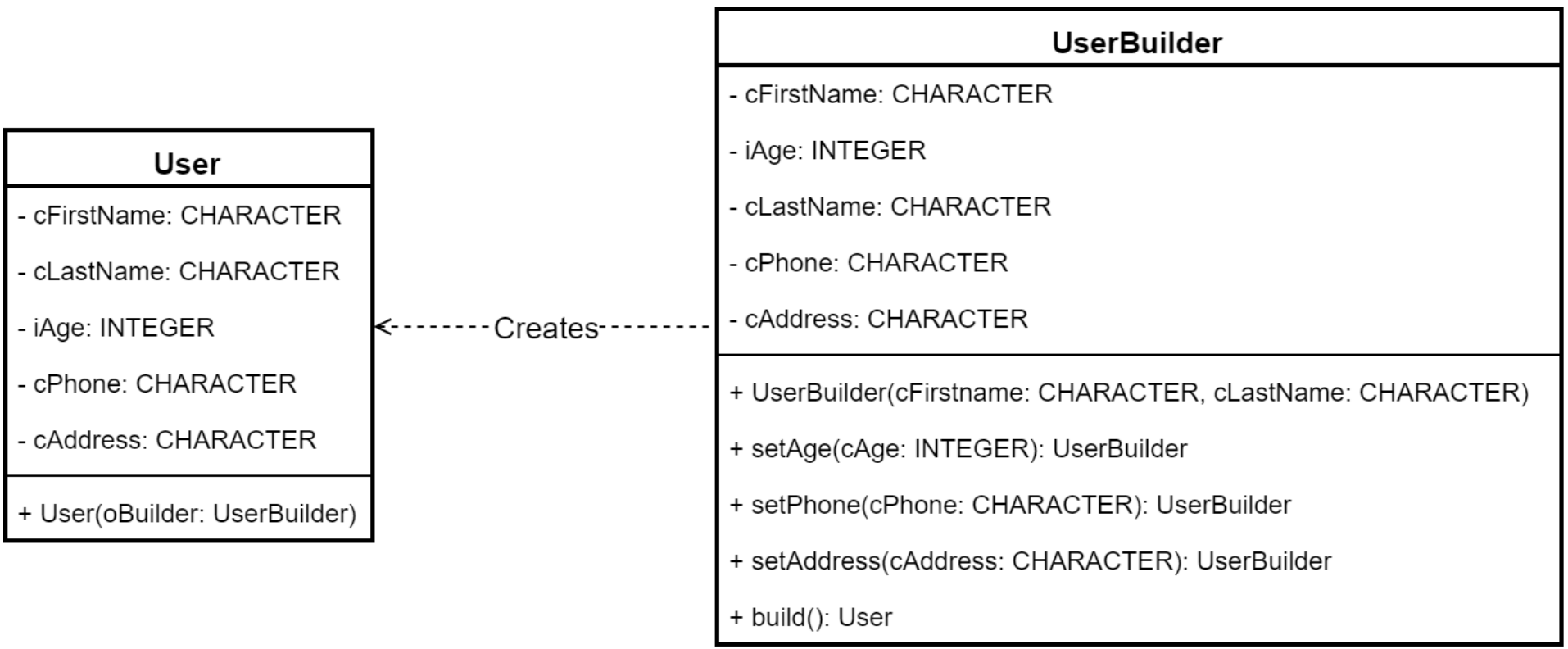
Pattern: Builder

Call with many parameters:

```
DEFINE VARIABLE oUser AS User NO-UNDO.

oUser = NEW User(
  "Max",
  "Mustermann",
  23,
  "+49 40-30 68 03-26",
  "Valentinskamp 30, 20355 Hamburg"
).
```

Pattern: Builder

With Builder Pattern:



Pattern: Builder

Builder Section of a setter:

```
CLASS UserBuilder:
…
  METHOD PUBLIC UserBuilder setAge(iAge AS INTEGER):
    THIS-OBJECT:iAge = iAge.
    RETURN THIS-OBJECT.
  END METHOD.
…
END CLASS.
```

Pattern: Builder

Call Builder:

```
DEFINE VARIABLE oUser AS User NO-UNDO.
oUser =
  (NEW UserBuilder("Max", "Mustermann")
  :setAge(23)
  :setPhone("+49 40-30 68 03-26")
  :setAddress("Valentinskamp 30, 20355 Hamburg")
  :build()).
```

Pattern: Builder

Small excerpt:

```
RUN StatusCreate IN l-Import-Library-
Handle
   ( INPUT    l-DB-Cust,

     INPUT    "",

     INPUT    150,
...
     INPUT
       "QtyType=" + OrderQtyQualifier
     + "{&T}"
       + "UTCTime=" + l-UTCTime
       + "{&T}"
       + "ConC-ID=" + SSCO-Ord.ConC-ID
...
) NO-ERROR.
```

Example of many parameters:

```
RUN StatusCreate IN l-Import-Library-Handle
  ( INPUT   l-DB-Cust,                  /* Cust Code */
    INPUT   "",                         /* Cnee Code */
    INPUT   150,                        /* status numeric */
  /* tb, 100304; export 8645 with O-E instead of O-I */
  &IF ("{&Exp_8645_with_O-E_v1}") = "TRUE" &THEN
    INPUT   "CreateNewRep2" + SSCO-Ord.OrderType + ",StartOrderExport665",  /* Create report flag */
  &ELSE
    INPUT   "CreateNewRep" + SSCO-Ord.OrderType + ",StartOrderExport665",  /* Create report flag */
  &ENDIF
    INPUT   TRUE,                       /* Report NEW = YES */
    INPUT   SSCO-o-Movement.Movement-ID, /* NOT Ord-ID */
    INPUT   "O",                        /* Status Type */
    INPUT   0,                          /* Suborder Number */
    INPUT   0,                          /* ? */
    INPUT   l-StatusDate,               /* Status Date */
    INPUT   l-StatusTime,               /* Status Time */
  /* tb, 050801 */
    INPUT   "Customer EDI",          /* User Code */
    INPUT   FALSE,                      /* Print 1 */
    INPUT   FALSE,                      /* Print 2 */
    INPUT   ?,                          /* default is Today */
    INPUT   "",                         /* Remarks */
    INPUT   SSCO-Ord.OrdQty,            /* Qty */
    INPUT   0,                          /* info code */
    INPUT   SSCO-Ord.Send-ID,           /* Send-ID */
    INPUT   SSCO-Ord.Send-Code,         /* Send-Code */
    /* no transmission to CIEL for Road orderlines */
    &IF ("{&Road_Order}") = "TRUE" &THEN
    INPUT   (SSCO-Ord.TrnsType-Code <> "R" AND b-Cust.Released), /* IsTransmit */
    &ELSE
    INPUT   b-Cust.Released,            /* IsTransmit */
    &ENDIF
    INPUT   l-Import-Date-asDate,       /* created on */
    INPUT   l-Import-Time-asChar,       /* time on */
    INPUT   "",                         /* knref */
    INPUT   "",                         /* damaged code */
    INPUT   "",                         /* address type-code */
    INPUT   ?,                          /* docs delivery date */
    INPUT   "",                         /* docs delivery time */
    INPUT   0,                          /* invoice header ID */
    INPUT   TRUE,                       /* check for duplicate status ? */
    INPUT   "",                         /* Reason Code */
    INPUT   "",                         /* Export/Import Flag */
    INPUT   "",                         /* SubStatus */
    INPUT   "QtyType=" + l-tt-{&ShipType}660.OrderQtyQualifier       + "{&T}" +
            "UTCTime=" + l-UTCTime                                   + "{&T}" +
            "ConC-ID=" + STRING(SSCO-Ord.ConC-ID), /* additional Fields ({&T}-separated list */
    OUTPUT  l-Stat-Code,             /* status code. if ? then status invalid */
    OUTPUT  l-Return-Code            /* returncode passed by called procedure */
) NO-ERROR.
```

Pattern: Builder

Call with Builder (small section):

```
DEFINE VARIABLE oStatusAttachment AS StatusAttachment NO-UNDO.

oStatusAnlage =
  (NEW StatusAttachmentBuilder()
  :setCustCode(l-DB-Cust)
  :setStatusNumeric(150)
...
  :setQtyType(OrderQtyQualifier)
  :setUTCTime(l-UTCTime)
  :setConCID(SSCO-Ord.ConC-ID)
...
  :build()).
```

Pattern: Singleton

- Type: Production pattern
- A replacement for Global Objects in OO
- When:
  - Only one object is needed in several components of the application
- Why:
  - Inheritance is possible
  - Logic during generation
  - Resource-efficient
- Example:
  - Configuration
  - Hard drive
  - Printer

Pattern: Singleton

Class with singleton pattern:

```
CLASS Configuration:
...
  DEFINE PUBLIC STATIC PROPERTY oInstance AS Configuration
    PUBLIC GET():
      IF oInstance = ? THEN
        oInstance = NEW Configuration().
      RETURN oInstance.
    END GET.
    PRIVATE SET.

  CONSTRUCTOR PRIVATE Configuration():
    reloadConfig().
  END CONSTRUCTOR.
...
END CLASS.
```

Pattern: Singleton

Singleton call:

```
DEFINE VARIABLE oConf AS Configuration NO-UNDO.

oConf = Configuration:oInstance.

oConf:setValue("mode", "debug").
oConf:saveToFile().
```

Pattern: Lazy Loading

- Type: Production pattern
- Delayed until first query:
  - Object generation
  - Value calculations
  - Other costly processes
- Often together with Singleton
- When:
  - Initialisation of a class takes a long time
- Why:
  - Performance
  - Resource-efficient

Pattern: Lazy Loading

Example part 1 (Property):

```
CLASS Invoice:
...
  DEFINE PUBLIC PROPERTY cCustomerName AS CHARACTER NO-UNDO INITIAL ? PRIVATE SET.
    PUBLIC GET:
      IF cCustomerName = ? THEN DO:
        DEFINE VARIABLE iCN AS INTEGER NO-UNDO.
        iCN = THIS-OBJECT:iCustNum.
        DEFINE BUFFER bCustomer FOR Customer.
        FIND FIRST bCustomer WHERE bCustomer.CustNum = iCN NO-LOCK NO-ERROR.
        IF AVAILABLE bCustomer THEN DO:
          cCustomerName = bCustomer.Name.
        END.
      END.

      RETURN cCustomerName.
    END GET.
...
END CLASS.
```

Pattern: Lazy Loading

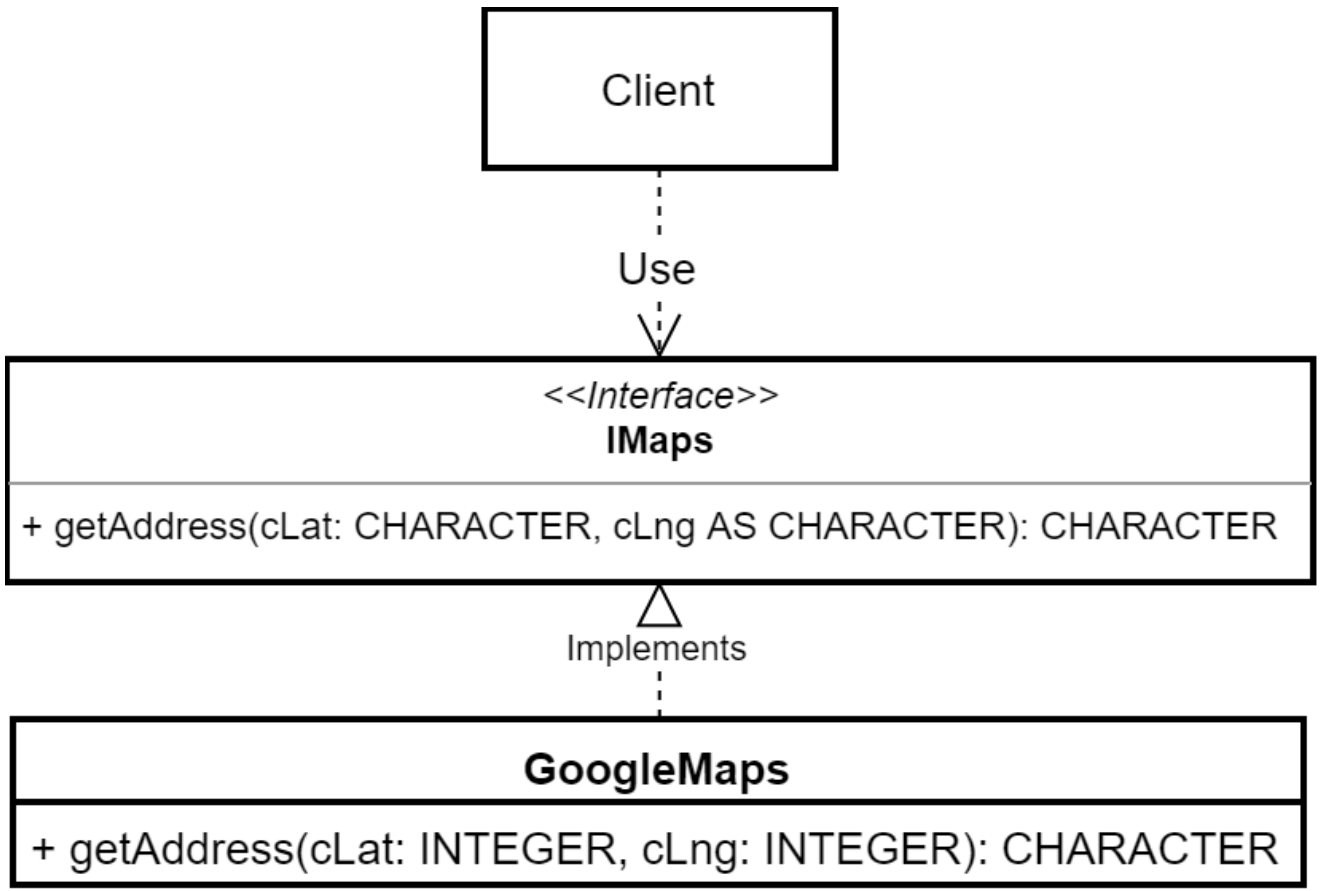
Example Part 2 (Constructor & Other Properties):

```
CLASS Invoice:
...
  CONSTRUCTOR PUBLIC Invoice(iInvoiceNum AS INTEGER):
    DEFINE BUFFER bInvoice FOR Invoice.

    FIND FIRST bInvoice WHERE bInvoice.Invoicenum = iInvoiceNum NO-LOCK
NO-ERROR.
    IF AVAILABLE bInvoice THEN DO:
      THIS-OBJECT:iInvoiceNum = iInvoiceNum.
      THIS-OBJECT:iCustNum    = bInvoice.CustNum.
    END.
  END CONSTRUCTOR.
...
END CLASS.
```

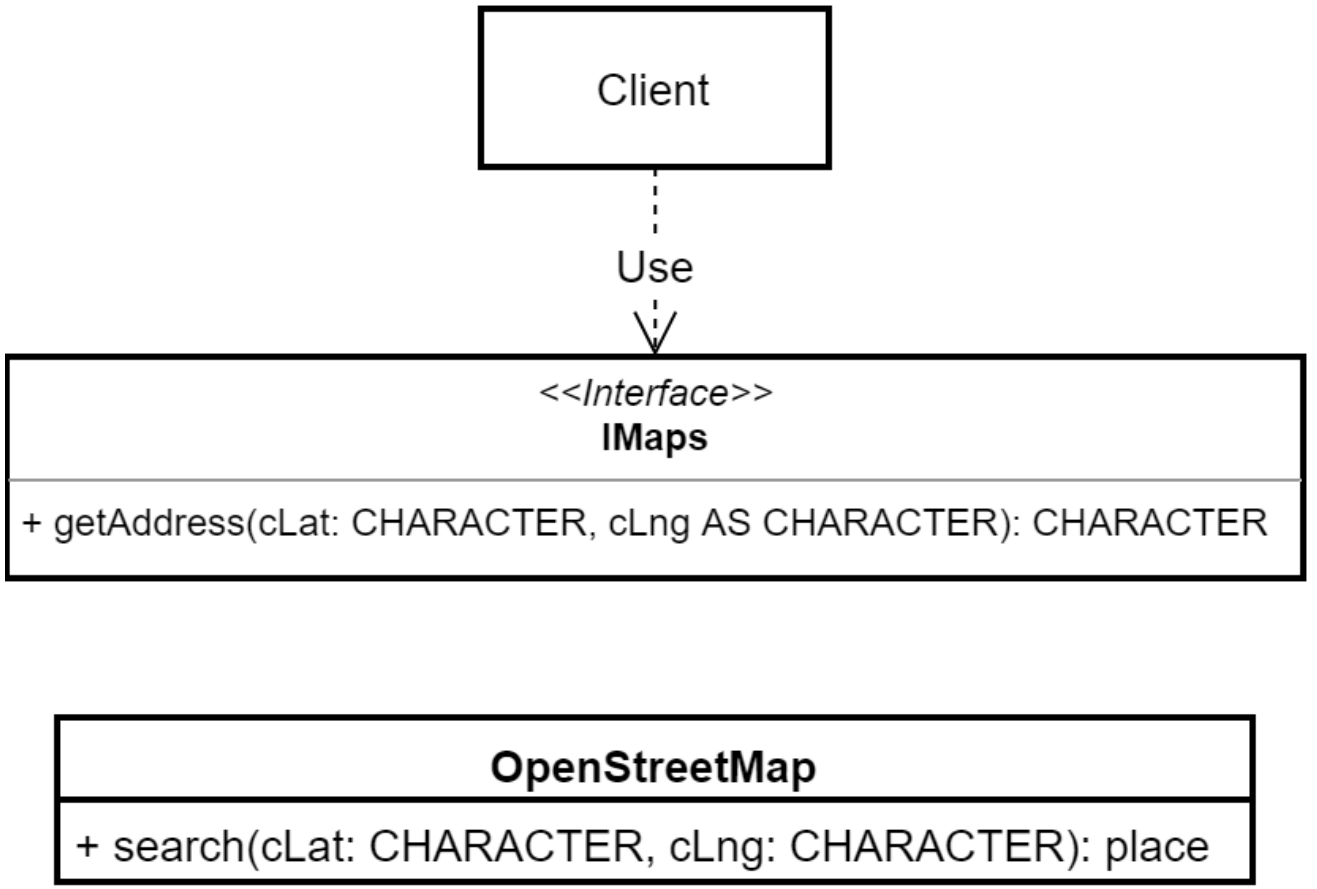
Pattern: Adapter

- Type: Structure pattern
- Connecting two incompatible interfaces
- When:
  - Using an old class in a new system
  - A new class with existing interface
- Why:
  - Old interface can be retained

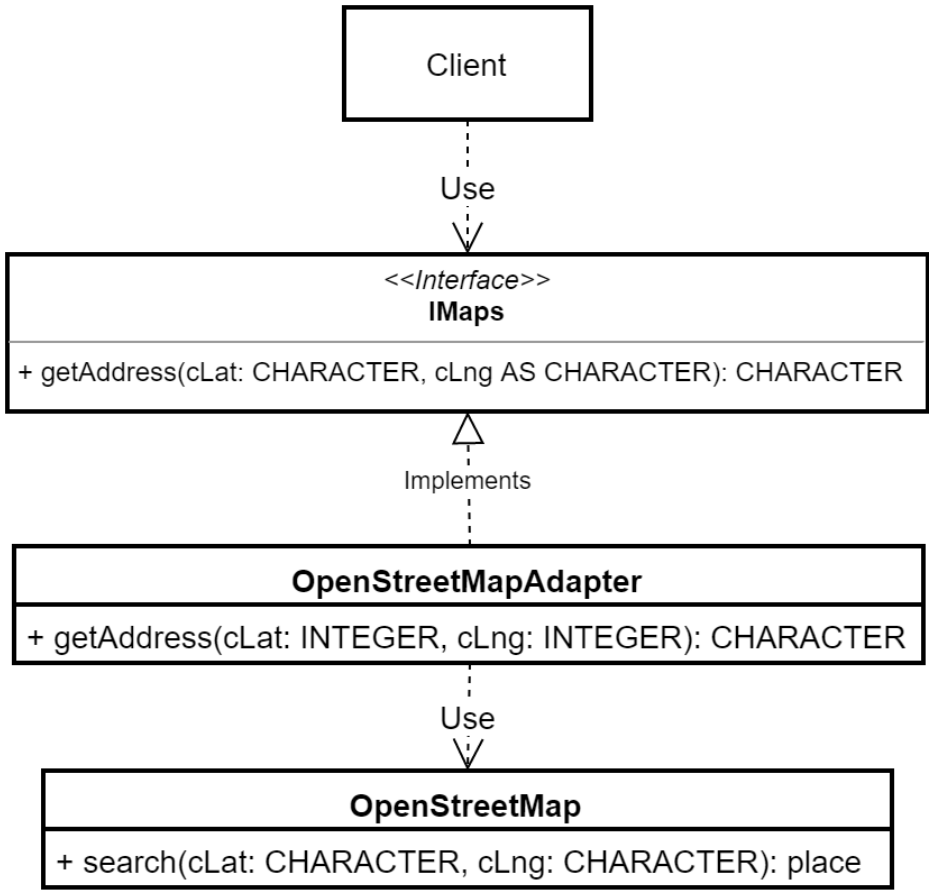
Pattern: Adapter



Pattern: Adapter



Pattern: Adapter



Pattern: Adapter

```
CLASS OpenStreetMapAdapter IMPLEMENTS IMaps:
  DEFINE PRIVATE PROPERTY oOpenStreetMap AS OpenStreetMap NO-UNDO
    PRIVATE GET.
    PRIVATE SET.

  CONSTRUCTOR PUBLIC OpenStreetMapAdapter():
    oOpenStreetMap = NEW OpenStreetMap().
  END CONSTRUCTOR.


  METHOD PUBLIC CHARACTER getAddress(cLat AS CHARACTER ,cLng AS
CHARACTER):
     RETURN oOpenStreetMap:search(cLat, cLng):Address.
  END METHOD.
END CLASS.
```

Agenda

- History of OOABL
- OO Basics
- Pattern
- Exercises

Exercise 1

- Create an order programs similar to the customer mask
- Save it under package *ui.screens.wawi.Order.cls*
- Fields: *Custnum , OrderNum , Orderdate*

Exercise 2

- Build an entity *appl.server.entities.be-order be-order.cls* analogous to *be-customer*
- Parameter are *Action(prev next)* , *CurrentRowid* , *CustNumFrom* and *CustNumTo*
- For *CustNumFrom* and *CustNumFrom* = 0, this parameter is to be ignored.

➔ Build a dynamic query

- Return type is a buffer handle

Exercise 3

- Add a button in the customer screen to call up the order programme (screen).
- The object handle of the customer screen is to be passed as a parameter to the order programme.

Exercise 5

- Create a Date-Display class based on the Fill-In class. This should always be of the type Date
- The screen value is to be incremented with "+" and decremented with "-".
- The name of this class is:
  - *appl.ui.components.native.date-display.cls*

Exercise 6

- When a control (widget) is made invisible, the associated label remains visible for the moment.
- → Change the fill-in class *appl.ui.components.native.fill-in.cls* so that the label is only visible with the fill-in.

- For testing purposes, we install a button in the screen that alternately makes the button visible.

Exercise 7

- Convert *be-order* into a singleton

Classes

- The characteristics of a class are called **member**, there are constructors, destructors, attributes, methods and events
- In ABL the definition of a class must be in files with the extension .cls
- Only one class can be defined in each file

```
CLASS Bird:
...
/* data definitions */
/* method definitions*/
/* constructor, destructor*/
...
END CLASS.
```