# NaturalX-Anwendungen entwickeln

Dieses Dokument beschreibt, wie eine Anwendung durch Definition und Benutzung von Klassen entwickelt wird.

Es umfasst die folgenden Themen:

- Entwickungsumgebungen
- Klassen definieren
- Klassen und Objekte benutzen

---

## Entwickungsumgebungen

- **Klassen auf Windows-Plattformen entwickeln**
  Auf Windows-Plattformen stellt Natural den Class Builder als Werkzeug zur Entwicklung von Natural-Klassen zur Verfügung. Der Class Builder präsentiert eine Natural-Klasse in einer strukturierten hierarchischen Reihenfolge und ermöglicht es dem Benutzer, die Klassen und ihre Komponenten effizient zu verwalten. Wenn Sie den Class Builder benutzen, sind überhaupt keine Vorkenntnisse oder auch nur Grundkenntnisse der unten beschriebenen Syntax-Elemente erforderlich.

- **Klassen mit SPoD entwickeln**
  In einer Natural Single Point of Development-Umgebung (SPoD), der zentralen Umgebung für Entwicklungen einschließlich eines Großrechners und/oder UNIX Remote Development Servers (DFÜ-Entwicklungsserver) können Sie den Class Builder benutzen, der zusammen mit der Natural Studio-Benutzeroberfläche zur Entwicklung von Klassen auf Großrechnern und/oder UNIX-Plattformen zur Verfügung steht. In diesem Fall sind keine Vorkenntnisse oder auch nur Grundkenntnisse der im Folgenden beschriebenen Syntax-Elemente erforderlich.

- **Klassen auf Großrechner- oder UNIX-Plattformen entwickeln**
  Wenn Sie SpoD nicht benutzen, entwickeln Sie mittels des Natural-Programm-Editors Klassen auf diesen Plattformen. In diesem Fall sollten Sie die Syntax der unten beschriebenen Klassen-Definition kennen.

## Klassen definieren

Wenn Sie eine Klasse definieren, müssen Sie ein Natural-Klassenmodul definieren, innerhalb dessen Sie ein `DEFINE CLASS`-Statement erstellen. Mittels des `DEFINE CLASS`-Statements können Sie der Klasse einen extern verwendbaren Namen zuweisen und ihre Schnittstellen, Methoden und Eigenschaften definieren. Der Klasse kann auch ein Objekt-Datenbereich zugewiesen werden, der das Layout einer Instanz der Klasse beschreibt. Das `DEFINE CLASS`-Statement dient außerdem dazu, einen Global Unique Identifier für diejenigen Klassen zur Verfügung zu stellen, die als COM-Klassen registriert werden sollen.

Dieser Abschnitt umfasst die folgenden Themen:

- Natural-Klassenmodul erstellen
- Klasse spezifizieren
- Schnittstelle definieren
- Objekt-Datenvariable einer Property zuweisen
- Subprogramm einer Methode zuweisen
- Methoden implementieren

## Natural-Klassenmodul erstellen

**Um ein Natural-Klassenmodul zu erstellen**

- Benutzen Sie das `CREATE OBJECT`-Statement zur Erstellung eines Natural-Objekts des Typs Klasse.

## Klasse spezifizieren

Das `DEFINE CLASS`-Statement definiert den Namen der Klasse, die Schnittstellen, die die Klasse unterstützt, und die Struktur ihrer Objekte. Das `DEFINE CLASS`-Statement dient außerdem dazu, einen Global Unique Identifier und die Activation Policy für diejenigen Klassen zur Verfügung zu stellen, die als COM-Klassen registriert werden sollen.

**Um eine Klasse zu spezifizieren**

- Benutzen Sie das `DEFINE CLASS`-Statement, das in der *Statements*-Dokumentation beschrieben ist.

## Schnittstelle definieren

Jede Schnittstelle einer Klasse wird mit einem `INTERFACE`-Statement innerhalb der Klassen-Definition angegeben. Ein `INTERFACE`-Statement gibt den Namen der Schnittstelle und eine Anzahl von Eigenschaften und Methoden an. Für Klassen, die als COM–Klassen registriert werden sollen, gibt es auch die Globally Unique ID der Schnittstelle an.

Eine Klasse kann eine oder mehrere Schnittstellen haben. Für jede Schnittstelle wird ein `INTERFACE`-Statement in der Klassen-Definition kodiert. Jedes `INTERFACE`-Statement enthält eine oder mehrere `PROPERTY`- und `METHOD`-Klauseln. Gewöhnlich stehen die in einer Schnittstelle enthaltenen Eigenschaften und Methoden miteinander in einem technischen oder betriebswirtschaftlichen Zusammenhang.

Die `PROPERTY`-Klausel definiert den Namen einer Eigenschaft und weist der Eigenschaft eine Variable vom Objekt-Datenbereich zu. Diese Variable wird zum Speichern des Wertes der Eigenschaft benutzt.

Die `METHOD`-Klausel definiert den Namen einer Methode und weist der Methode ein Subprogramm zu. Dieses Subprogramm wird zum Implementieren der Methode benutzt.

**Um eine Schnittstelle zu definieren**

- Benutzen Sie das INTERFACE-Statement (siehe *Statements*-Dokumentation).

## Objekt-Datenvariable einer Property zuweisen

Das PROPERTY-Statement wird nur benutzt, wenn mehrere Klassen dieselbe Schnittstelle auf verschiedene Art und Weise implementieren sollen. In diesem Fall benutzen die Klassen dieselbe Schnittstellen-Definition gemeinsam und beziehen sie von einem Natural-Copycode ein. Das PROPERTY-Statement wird dann benutzt, um *außerhalb* der Schnittstellen-Definition eine Variable vom Objekt-Datenbereich einer Eigenschaft zuzuweisen. Wie die PROPERTY-Klausel des INTERFACE-Statements definiert das PROPERTY-Statement den Namen einer Eigenschaft und weist eine Variable vom Objekt-Datenbereich der Eigenschaft zu. Diese Variable wird zum Speichern des Wertes der Eigenschaft benutzt.

**Um eine Objekt-Datenvariable einer Eigenschaft zuzuweisen**

- Benutzen Sie das PROPERTY-Statement (siehe *Statements*-Dokumentation).

## Subprogramm einer Methode zuweisen

Das METHOD-Statement wird nur benutzt, wenn mehrere Klassen dieselbe Schnittstelle auf verschiedene Arten implementieren sollen. In diesem Fall benutzen die Klassen dieselbe Schnittstellen-Definition gemeinsam und beziehen sie von einem Natural-Copycode ein. Das METHOD-Statement wird dann benutzt, um außerhalb der Schnittstellen-Definition der Methode ein Subprogramm zuzuweisen. Wie die METHOD-Klausel des INTERFACE-Statements definiert das METHOD-Statement den Namen einer Methode und weist der Methode ein Subprogramm zu. Dieses Subprogramm wird zum Implementieren der Methode benutzt.

**Um einer Methode ein Subprogramm zuzuweisen**

- Benutzen Sie das METHOD-Statement (siehe *Statements*-Dokumentation).

## Methoden implementieren

Eine Methode wird in der folgenden allgemeinen Form als ein Natural-Subprogramm implementiert:

```
DEFINE DATA
*
* Implementation code of the method
*
END
```

Informationen zum DEFINE DATA-Statement siehe *Statements*-Dokumentation.

Alle Klauseln des DEFINE DATA-Statements sind optional.

Um die Daten-Konsistenz sicherzustellen, empfiehlt es sich, dass Sie keine Inline-Datendefinitionen, sondern Data Areas benutzen.

Wenn eine PARAMETER clause-Klausel angegeben wird, kann die Methode Parameter und/oder einen Rückgabewert haben.

Mit BY VALUE in der Parameter Data Area markierte Parameter sind Eingabe-Parameter der Methode.

Nicht mit BY VALUE markierte Parameter werden "By Reference" übergeben und sind Eingabe/Ausgabe-Parameter. Dies ist die Voreinstellung.

Der erste mit BY VALUE RESULT markierte Parameter wird als Rückgabewert für die Methode zurückgegeben. Wenn mehr als ein Parameter auf diese Art markiert wird, werden die anderen als Eingabe/Ausgabe-Parameter behandelt.

Als OPTIONAL markierte Parameter brauchen nicht angegeben zu werden, wenn die Methode aufgerufen wird und Sie die *n*X-Notation im SEND METHOD-Statement benutzen.

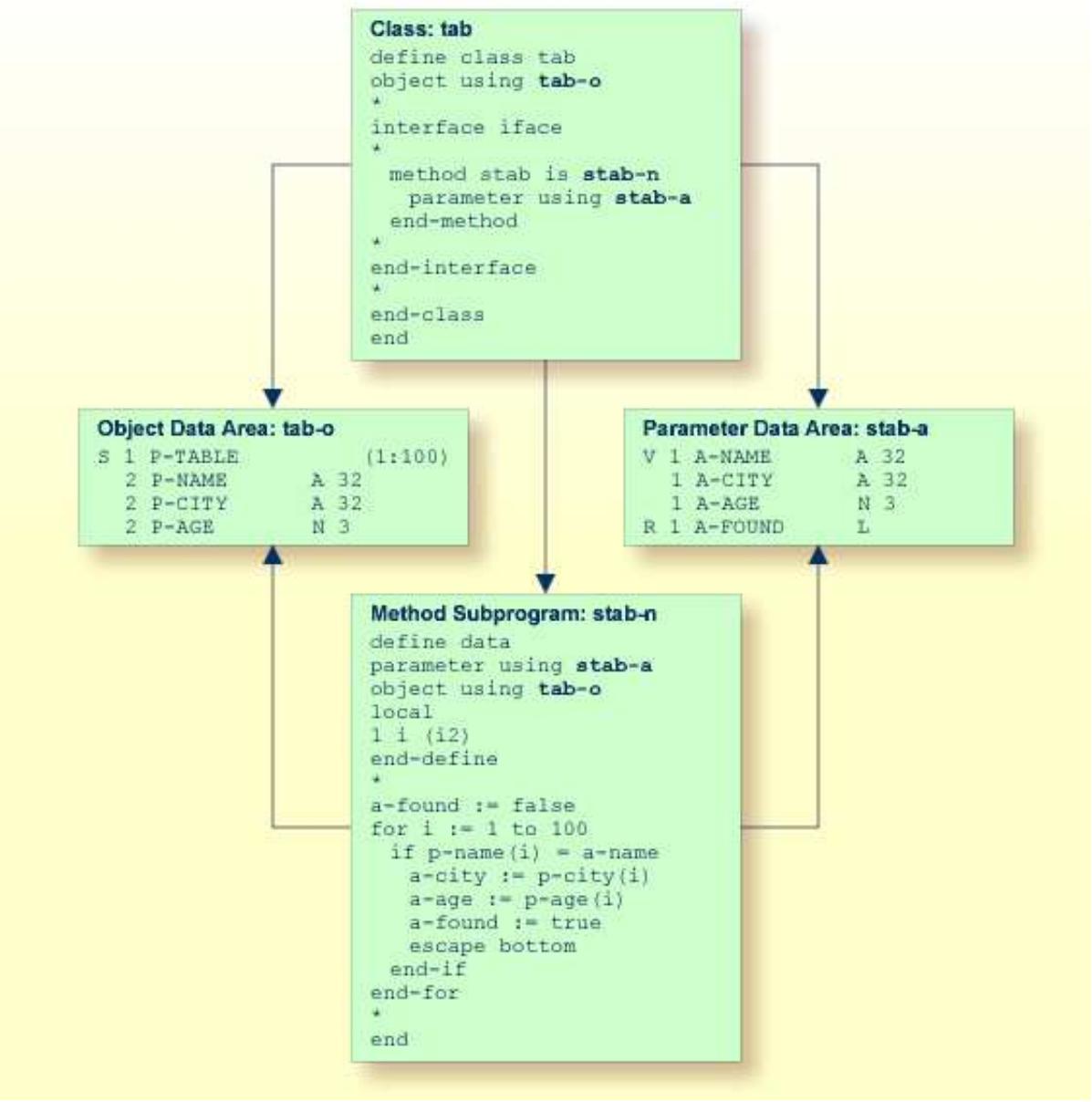
Um sicherzustellen, dass das Methoden-Subprogramm genau dieselben Parameter akzeptiert, wie in dem entsprechenden METHOD-Statement in der Klassen-Definition angegeben, benutzen Sie eine Parameter Data Area anstatt von Inline-Datendefinitionen. Benutzen Sie dieselbe Parameter Data Area wie in dem betreffenden METHOD-Statement.

Um dem Methoden-Subprogramm Zugriff auf die Objektdaten-Struktur zu geben, kann die OBJECT-Klausel angegeben werden. Um sicherzustellen, dass das Methoden-Subprogramm korrekt auf die Objektdaten zugreifen kann, benutzen Sie eine Local Data Area anstatt von Inline-Datendefinitionen. Verwenden Sie dieselbe Local Data Area, wie in der OBJECT-Klausel des DEFINE CLASS-Statements angegeben.

Die Klauseln GLOBAL, LOCAL und INDEPENDENT können wie in jedem anderen Natural-Programm benutzt werden.

Obwohl es technisch möglich ist, macht es gewöhnlich keinen Sinn, eine CONTEXT-Klausel in einem Methoden-Subprogramm zu benutzen.

In dem folgenden Beispiel werden Daten über eine vorgegebene Person von einer Tabelle eingelesen. Der Suchschlüssel wird als ein BY VALUE-Parameter übergeben. Die sich daraus ergebenden Daten werden über “by reference”-Parameter zurückgegeben ("By Reference" ist die Standard-Definition). Der Rückgabewert der Methode wird durch die Spezifikation BY VALUE RESULT definiert.




# Klassen und Objekte benutzen

Auf in einer lokalen Natural-Session erstellte Objekte kann von anderen Modulen aus in derselben Natural-Session zugegriffen werden.

Auf Objekte, die in anderen Prozessen oder auf Remote-Maschinen erstellt wurden, kann über DCOM zugegriffen werden.

In beiden Fällen gelten dieselben Regeln für den Zugriff auf und die Verwendung von Klassen und ihren Objekten.

Das Statement `CREATE OBJECT` wird benutzt, um ein Objekt (auch als *Instance* bezeichnet) einer gegebenen Klasse zu erstellen.

Um Objekte in Natural-Programmen zu referenzieren, müssen Object-Handles im `DEFINE DATA`-Statement definiert werden. Methoden eines Objekts werden mit dem Statement `SEND METHOD` aufgerufen. Objekte können Eigenschaften haben, auf die mit der normalen Zuweisungssyntax zugegriffen wird.

**Anmerkung:**
Damit Sie eine NaturalX-Klasse über DECOM benutzen können, müssen Sie sie zuerst registrieren.

Diese Schritte sind im Folgenden beschrieben:

- Objekt-Handles definieren
- Instanz einer Klasse erstellen
- Bestimmte Methode eines Objekts aufrufen
- Properties aufrufen
- Beispielanwendung

## Objekt-Handles definieren

Um Objekte in Natural-Programmen zu referenzieren, müssen Object-Handles im `DEFINE DATA`-Statement wie folgt definiert werden:

```
DEFINE DATA
  level-handle-name [(array-definition)] HANDLE OF OBJECT
  ...
END-DEFINE
```

Beispiel:

```
DEFINE DATA LOCAL
1 #MYOBJ1 HANDLE OF OBJECT
1 #MYOBJ2 (1:5) HANDLE OF OBJECT
END-DEFINE
```

## Instanz einer Klasse erstellen

**Um eine Instanz einer Klasse zu erstellen**

- Benutzen Sie das `CREATE OBJECT`-Statement (siehe Natural *Statements*-Dokumentation).

## Bestimmte Methode eines Objekts aufrufen

**Um eine bestimmte Methode eines Objekts aufzurufen**

- Benutzen Sie das `SEND METHOD`-Statement (siehe Natural *Statements*-Dokumentation).

# Properties aufrufen

Auf Properties kann mit dem Statement ASSIGN (oder COMPUTE) wie folgt zugegriffen werden:

**ASSIGN** *operand1.property-name = operand2*

**ASSIGN** *operand2 = operand1.property-name*

## Object Handle — *operand1*

*Operand1* muss als eine Object-Handle definiert werden und identifiziert das Objekt, dessen Eigenschaft aufgerufen werden soll. Das Objekt muss bereits vorhanden sein.

### *operand2*

Als *operand2* geben Sie einen Operanden an, dessen Format datenübertragungskompatibel zu dem Format der Eigenschaft sein muss. Weitere Informationen siehe *Kompatibilitätsregeln zur Datenübertragung*.

Wenn auf das Objekt über DCOM zugegriffen werden soll, sind die Regeln zur Konvertierung von Datentypen im Abschnitt *Using Type Information* in der *Operations*-Dokumentation zu berücksichtigen.

### *property-name*

Der Name einer Property des Objekts.

Stimmt der Name der Property mit der Natural Identifier Syntax überein, kann er wie folgt angegeben werden:

```
create object #o1 of class "Employee"
  #age := #o1.Age
```

Wenn der Name der Property nicht mit der Natural Identifier Syntax übereinstimmt, muss er in spitze Klammern gesetzt werden:

```
create object #o1 of class "Employee"
  #salary := #o1.<<%Salary>>
```

Der Name der Property kann auch mit einem Schnittstellen-Namen qualifiziert werden. Dies ist erforderlich, wenn das Objekt mehr als eine Schnittstelle hat, die eine Property mit demselben Namen enthält. In diesem Fall muss der qualifizierte Name der Property in spitzen Klammern stehen:

```
create object #o1 of class "Employee"
  #age := #o1.<<PersonalData.Age>>
```

Beispiel:

```
define data
  local
  1 #i         (i2)
  1 #o         handle of object
  1 #p         (5) handle of object
  1 #q         (5) handle of object
  1 #salary    (p7.2)
  1 #history   (p7.2/1:10)
  end-define
```

```
  * ...
  * Code omitted for brevity.
  * ...
  * Set/Read the Salary property of the object #o.
  #o.Salary := #salary
  #salary := #o.Salary
  * Set/Read the Salary property of
  * the second object of the array #p.
  #p.Salary(2) := #salary
  #salary := #p.Salary(2)
  *
  * Set/Read the SalaryHistory property of the object #o.
  #o.SalaryHistory := #history(1:10)
  #history(1:10) := #o.SalaryHistory
  * Set/Read the SalaryHistory property of
  * the second object of the array #p.
  #p.SalaryHistory(2) := #history(1:10)
  #history(1:10) := #p.SalaryHistory(2)
  *
  * Set the Salary property of each object in #p to the same value.
  #p.Salary(*) := #salary
  * Set the SalaryHistory property of each object in #p
  * to the same value.
  #p.SalaryHistory(*) := #history(1:10)
  *
  * Set the Salary property of each object in #p to the value
  * of the Salary property of the corresponding object in #q.
  #p.Salary(*) := #q.Salary(*)
  * Set the SalaryHistory property of each object in #p to the value
  * of the SalaryHistory property of the corresponding object in #q.
  #p.SalaryHistory(*) := #q.SalaryHistory(*)
  *
  end
```

Um Arrays von Objekt-Handles und Eigenschaften mit Arrays als Werte korrekt benutzen zu können, ist es wichtig, Folgendes zu wissen:

Eine Eigenschaft einer Array-Ausprägung von Objekt-Handles wird mit der folgenden Index-Notation adressiert:

```
#p.Salary(2) := #salary
```

Auf eine Eigenschaft, die ein Array als Wert hat, wird stets als Ganzes zugegriffen. Deshalb ist keine Index-Notation mit dem Namen der Eigenschaft erforderlich:

```
#o.SalaryHistory := #history(1:10)
```

Eine Eigenschaft einer Array-Ausprägung von Objekt-Handles, die ein Array als Wert hat, wird deswegen wie folgt adressiert:

```
#p.SalaryHistory(2) := #history(1:10)
```

## Beispielanwendung

Eine Beispielanwendung finden Sie in den Libraries SYSEXCOM und SYSEXCOC. Die A-README-Objekte in diesen Libraries enthalten Information darüber, wie das Beispiel ausgeführt wird.