Gold Training

eWAM Training

Introduction

- **Types**
- **Constants**
- **Variables**
- **Methods**
- **Statements**
- **Intrinsics**
- **Keywords/Misc**
- **Important methods**

Types

- **Basic Types**
- **Advanced Types**
- **Reference Types**

Types

- **GOLD is a strongly typed language**

  Types define the values that a variable can hold or that an expression can produce.

```
type <Identifier>: <SimpleType>
type <Identifier>: <StructuredType>
type <Identifier>: <ReferenceType>
```

Types

- **compare to Java:**

- Java is also strongly typed.
- However, Java has no analogous construct which can be used to define alternate names for basic types.
- **classes** or **enums** may be used instead for similar purposes.

- **compare to C++:**

- C++ is also strongly typed.
- typedef

```
typedef <SimpleType> <Identifier>;
typedef <StructType> <Identifier>;
typedef <PointerType> <Identifier>;
```

Basic Types

- **Integer**

```
type  tMeters : Int4
var   HeightInMeters : tMeters
```

Type	Range	Format
Int1	0 .. 255	8 bits, unsigned
Int2	-32768 .. 32767	16 bits, signed
Int4	-2147483648 .. 2147483647	32 bits, signed
Int8	-9.22 e+18 .. 9.22 e+18	64 bits, signed

Basic Types

- **Number**
  - `type tLength : Num8`
    `var SizeInMillimeters : tLength`

Type	Range	Decimal
Num4	1.17.. e–38 .. 3.40.. e+38	9
Num8	2.22.. e–308 ..1.79.. e+308	16
Num10	1.9.. e-4951 .. 1.1.. e+4952	20

Basic Types

- *in* GOLD*:*

```
Int1
Int2
Int4
Int8
Num4
Num8
Num10
```

```
type tMeters : Int4
var HeightInMeters : tMeters

type tLength : Num8
var SizeInMils : tLength
```

- *compare to* Java*:*

```
byte
short
int
long
float
double
(n/a)
```

```
int heightInMeters;

double sizeInMils;
```

- *compare to* C++*:*

```
char
short int
int (or long)
long long
float
double
long double
```

```
typedef int tMeters;
tMeters heightInMeters;

typedef double tLength;
tLength sizeInMils;
```

Basic Types

- **Boolean**
  - ```
    type tIsOk : Boolean
    var Response : tIsOk
    Response = False
    ```

Booleans are TRUE or FALSE like java. While storing in database, it comes 0 (false) and 1 (true)

True and False constants
    ```

Basic Types

- Character
  - ```
    type tLetter : char
    var KeyStroke : tLetter
    KeyStroke = 'A'
    ```
    ```

Basic Types

- **String (Limited to 255 characters)**
  - CString
    - `type <Identifier>: CString(<Size>)`
    - Null terminated strings
  - String
    - `type <Identifier>: String(<Size>)`
    - First byte reserved for the length of the string
  - Already defined type strings

```
type String31 : String(31)
type String : String(255)
type CString31 : CString(31)
type CString : CString(255)
```

  - User defined type strings

```
type tName : CString(63)
type tComment : CString
type tSerialNumber : CString(10)
```

Basic Types

- **Pointer**
  - ```
    type tpName : .CString ;(pointer type to a string)
    var pName : tpName
    var Name : CString

    Name = 'John'
    pName = @Name ;(address of the variable Name)
    pName. = 'Bill' ;(variable Name now = 'Bill')
    ```
  - Pointer to anything
    ```
    var Ptr : Pointer
    var Name : Cstring
    Ptr = @Name
    tpCString(Ptr). = 'Bill'
    ```
  - Predefined pointer types
    - tpCString
    - tpInt4
    - tpNum8
    - …
    ```

Basic Types

- *in* GOLD*:*

```
Boolean
Char
String (or String(255))
Cstring (or CString(255))
(quoted string literal)
Text
Pointer
```

- *compare to* Java*:*

```
boolean
char
char[255]
char[255]
String
StringBuilder
(n/a, Java only has refs)
```

- *compare to* C++*:*

```
Bool
char
char[255]
char[255]
“I am a text string”
string
char *
```

Basic Types

- *in* GOLD*:*

```
type tpCString : .CString
var pName : tpName
var Name : CString

Name = 'John'
pName = @Name
pName. = 'Bill'

var Ptr : Pointer
Ptr = @Name
tpCString(Ptr). = 'Bill'
```

- *compare to* Java*:*

```
// Note: Since Java does not have
// pointers, this looks different.

class StringRef { String s; }
StringRef pName;
StringRef name = new StringRef();

name.s = "John";
pName = name;
pName.s = "Bill";

Object ptr;
ptr = name;
((StringRef)ptr).s = "Bill";
```

- *compare to* C++*:*

```
typedef string *tpCString;
string *pName;
string Name;

Name = “John”;
pName = &Name;
*pName = “Bill”;

void *Ptr;
Ptr = Name;
*((string *) Ptr) = “Bill”
```

- **Enumeration**
  - `type tPhoneKind : (FixedLine, Cellular, Satellite, PhoneBooth)`
    `var PhoneKind : tPhoneKind`

```
if PhoneKind = PhoneBooth
 Alert('Please insert a dime')
endIf
```

Text of constants can be specified for graphical presentations

Advanced types

- *in* GOLD*:*

```
type tPhoneKind :
   (FixedLine,
   Cellular,
   Satellite,
   PhoneBooth)

var PhoneKind :
   tPhoneKind

if PhoneKind =
   PhoneBooth
   Alert('Please
   insert $.50')
endIf
```

- *compare to* Java*:*

```
enum TPhoneKind {
   FIXED_LINE,
   CELLULAR,
   SATELLITE,
   PHONE_BOOTH
}

TPhoneKind
   phoneKind;

if(phoneKind ==
   TPhoneKind.PHONE_
   BOOTH) {
   JOptionPane.showM
   essageDialog(null
   , "Please insert
   $.50");
}
```

- *compare to* C++*:*

```
typedef enum {
   FixedLine,
   Cellular,
   Satellite,
   PhoneBooth
} tPhoneKind;

tPhoneKind
   PhoneKind;

if (PhoneKind ==
   PhoneBooth) {
   cout << "Please
   insert \$.50";
}
```

Advanced Types

- **Set**
  - `type tCarOption : (TurboCharged, Convertible, V8Engine, Automatic, AirScoop)`

```
type tCarOptions : [tCarOption]
var Options : tCarOptions

Options = [TurboCharged,Convertible]
if TurboCharged in Options
  Alert('Don''t drive too fast')
endIf
```

  - Operators: In, +, -, *
  - Empty set: []

Advanced types

- *in* GOLD*:*

```
type tCarOption :
    (TurboCharged,
    Convertible,
    V8Engine, Automatic,
    AirScoop)

type tCarOptions :
    [tCarOption]

Options =
    [TurboCharged,
    Convertible]



if TurboCharged in
    Options
     Alert('Don''t drive
    too fast')
endIf
```

- *compare to* Java*:*

```
enum TCarOption {
    TURBO_CHARGED,
    CONVERTIBLE,
    V8_ENGINE, AUTOMATIC,
    AIR_SCOOP
}

Set<CarOption> options;
options =
    EnumSet.noneOf(TCarOp
    tion.class);
options.add(TCarOption.TU
    RBO_CHARGED);
options.add(TCarOption.CO
    NVERTIBLE);
if(options.contains(TCarO
    ption.TURBO_CHARGED))
    {
    JOptionPane.showMessa
    geDialog(null, "Don't
    drive too fast!");
}
```

- *compare to* C++*:*

```
typedef enum {
    TurboCharged,
    Convertible,
    V8Engine, Automatic,
    AirScoop
} tCarOption;
typedef set<tCarOption>
    tCarOptions;

(Options =
    tCarOptions()).inser
    t(TurboCharged);
    Options.insert(Conve
    rtible);

if(Options.find(TurboCh
    arged) !=
    Options.end()) {
    cout << "Don't drive
    too fast";
}
```

Advanced Types

- **SubRange**
  -
```
type tNumberOfDay : 1 to 7
type tUpperCase : 'A' to 'Z'
var theDayNumber : tNumberOfDay
var theLetter : tUpperCase

theDayNumber = 1
theLetter = 'B'
theDayNumber = 8 ;(Generates error)
theDayNumber = 1 to 'a' ;(Generates error)
```

Advanced Types

- **Array**
  - ```
    type tWeek : array [tNumberOfDay] of String31
    type tWeekHours : array [tNumberOfDay] [0 to 23]
    of CString31
    var Days : tWeek
    var Hours : tWeekHours

    Days[1] = 'Sunday'
    Days[7] = 'Saturday'
    Hours[1][1] = 'One' (on Sunday)
    ```
  - Array assignments, comparisons
    ```

Advanced types

- *in* GOLD*:*

```
type tWeek : array
   [tNumberOfDay] of
   String31
type tWeekHours :
   array
   [tNumberOfDay] [1
   to 24] of
   CString31
var Days : tWeek
var Hours :
   tWeekHours

Days[1] = 'Sunday'
Days[7] = 'Saturday'
Hours[1][1] =
'One AM on Sunday'
```

- *compare to* Java*:*

```
String[1] days = new
   String[7];
String[1][1] hours =
   new String[7][24];
days[1] = "Sunday";
days[7] = "Saturday";
hours[1][1] = "One";
```

- *compare to* C++*:*

```
typedef string
   [tNumberOfDay]
   tWeek;
typedef string
   [tNumberOfDay] [24]
   tWeekHours;

tWeek Days;
tWeekHours Hours;

Days[1] = "Sunday";
Days[7] = "Saturday";
Hours[1][1] = "One";
```

Advanced Types

- **Record**
  - ```
    type tHouse : record
     NumberOfFloors : Int4
     Area : Int4
     Color : tColors
     HasGarden : Boolean
    endRecord
    var myHouse : tHouse

    myHouse.NumberOfFloors = 4
    myHouse.HasGarden = True
    ```
  - Record assignments, comparisons
    ```

Advanced types

- *in* GOLD*:*

```
type tHouse : record
   NumberOfFloors :
   Int4
   Area : Int4
   Color : tColors
   HasGarden :
   Boolean
endRecord

var myHouse : tHouse


myHouse.NumberOfFloo
   rs = 4
myHouse.HasGarden =
   True
```

- *compare to* Java*:*

```
class THouse {
   int
   numberOfFloors;
   int area;
   TColors color;
   boolean
   hasGarden;
}

THouse myHouse = new
   Thouse();

myHouse.numberOfFloo
   rs = 4;
myHouse.hasGarden =
   true;
```

- *compare to* C++*:*

```
typedef struct {
int NumberOfFloors;
int Area;
tColors Color;
bool HasGarden;
} tHouse;



tHouse myHouse;

myHouse.NumberOfFloo
   rs = 4;
myHouse.HasGarden =
   true;
```

Advanced Types

- **Text**
  - String of any size. Text types resize automatically as text grows.
  - ```
    var Greetings : text

    Write(Greetings,'Hello, ', theClient)
    Writeln(Greetings,'. How''s the weather in `,
    theClient.myAddress.City, `?')
    ```
  - Manipulated with intrinsics, method types
    ```

Advanced types

- *in* GOLD*:*

```
var Greetings : text


Write(Greetings,
   'Hello, ',
   theClient)

Writeln(Greetings,
   '. How''s the
   weather in ',
   theClient.myAddre
   ss.City, '?')
```

- *compare to* Java*:*

```
StringBuilder
   greetings = new
   StringBuilder();


greetings.append("Hell
   o, " + theClient);

greetings.append(".
   How's the weather
   in " +
   theClient.myAddress
   .City + "?\n");
```

- *compare to* C++*:*

```
ostringstream gstream;
string Greetings;


gstream << "Hello, "
   << theClient;

gstream << ". How's
   the weather in " <<
   theClient.myAddress
   .City << "?" <<
   endl;

Greetings =
   gstream.str();
```

Reference Types

- **RefTo**
  - A link between one instance of a referencing class and one instance of a referenced class.
  - For instance variables of descendents of aFullObject
  - Features
    - Persistency
    - Typed (InTransaction, Versioned, Inverse, …)
    - Overrideable (Role class)

Reference Types

- **RefTo usage**
  - Declaration

```
var theVehicle:aVehicle
myVehicle : refTo aVehicle
```

  - Manipulation

```
Self.myVehicle = theVehicle
Self.myVehicle = Nil
if Self.myVehicle <> Nil
  Self.myVehicle.Color = cRed
endIf
```

Reference types

- *in* GOLD*:*

```
myVehicle : refTo
   aVehicle

Self.myVehicle =
   theVehicle

Self.myVehicle = Nil


if Self.myVehicle <>
   Nil
   Self.myVehicle.Co
   lor = cRed
endIf
```

- *compare to* Java*:*

```
AVehicle myVehicle;


myVehicle =
   theVehicle;

myVehicle = null;


if(myVehicle !=
   null) {
   myVehicle.color =
   C_RED;
}
```

- *compare to* C++*:*

```
aVehicle *myVehicle;


this->myVehicle =
   theVehicle

this->myVehicle =
   NULL

if(this->myVehicle
   != NULL) {
   this->myVehicle-
   >Color = cRed;
}
```

Reference Types

- **ListOf**
  - A link between one instance of a referencing class and many instances of a referenced class
  - For instance variables only of descendents of aFullObject
  - Features
    - Persistency
    - Typed (InTransaction, Versioned, Inverse, …)
    - Overrideable (Role class)

Reference Types

- **ListOf usage**
  - Declaration

```
myVehicles : listOf aVehicle
```

  - Manipulation

```
Self.myVehicles.AppendObject(theVehicle)
Self.myVehicles.InsertObjectAt(theVehicle, 5)
Self.myVehicles.DeleteObjectAt(5)
curVehicle = Self.myVehicles[1]
forEach curVehicle in Self.myVehicles
  curVehicle.Color = cRed
endFor
```

Reference types

- *in* GOLD*:*

```
myVehicles : listOf
    aVehicle
Self.myVehicles.AppendO
    bject( theVehicle)
Self.myVehicles.InsertO
    bjectAt( theVehicle,
    5)
Self.myVehicles.DeleteO
    bjectAt(5)
curVehicle = Self.
    myVehicles[1]
forEach curVehicle in
    Self.myVehicles
    curVehicle.Color =
    cRed
endFor
```

- *compare to* Java*:*

```
List<AVehicle> myVehicles
    = new
    ArrayList<AVehicle>()
    ;
myVehicles.add(theVehicle
    );
myVehicles.add(5,
    theVehicle);
myVehicles.remove(5);

curVehicle =
    myVehicles.get(1);

for (curVehicle :
    myVehicles) {
    curVehicle.color =
    C_RED;
}
```

- *compare to* C++*:*

```
vector<aVehicle>
    myVehicles;
this->myVehicles.
    push_back(
    theVehicle);
this->myVehicles. insert(
    myVehicles.
    begin()[5],
    theVehicle)
this->myVehicles. erase(
    myVehicles.
    begin()[5] )
curVehicle = this->
    myVehicles[1];
for(curVehicle= this->
    myVehicles.begin();
    curVehicle !=this->
    myVehicles.end();
    curVehicle++) {
    curVehicle->Color =
    cRed; }
```

Constants

- **Integer constant**
  - `const cMaxInt4 = 2147483647`
  - `const cReallyBigInt = 12345678901234567L`
- String constant
  - `const cMessage = 'Hello'`
  - `const cError = 'You can''t do that'`
- Number constant
  - `const cPi = 3.1415926535`

Constants

- *in* GOLD*:*

```
const cMaxInt4 =
   2147483647
const cReallyBigInt
   =
   12345678901234567
   L


const cMessage =
   'Hello'
Const cError = 'You
   can''t do that'


const cPi =
   3.1415926535
```

- *compare to* Java*:*

```
static final int
   C_MAX_INT4 =
   2147483647;
static final long
   C_REALLY_BIG_INT =
   12345678901234567L;


static final String
   C_MESSAGE =
   "Hello";
static final String
   C_ERROR = "You
   can't do that";
static final double
   C_PI =
   3.1415926535;
```

- *compare to* C++*:*

```
const int cMaxInt4 =
   2147483647;
const long long
   cReallyBigInt =
   12345678901234567LL
   ;


const char cMessage[]
   = "Hello";
const char cError[] =
   "You can't do that"


const double cPi =
   3.1415926535;
```

Variables

- Scope
  - Instance (Class Variables)
  - Local (Method Variables)
  - Global (Module Variables)

- keywords
  - Memory
  - Absolute
  - Override
  - Parameter

Variables

- **Instance**
  - Member variable of a Class

```
Counter : Int4
Name : CString

    procedure SendStatement
        if self.Name <> 'Smith'
            …
        endIf
```

  -- All Instance Variables of aFullObject are Stored unless otherwise specified.

Variables

- *in* GOLD*:*

```
Counter : Int4
Name : CString

procedure
   SendStatement
   if self.Name <>
   'Smith'
        …
   endIf
```

- *compare to* Java*:*

```
int counter;
String name;

void sendStatement() {
   if
   (!name.equals("Smit
   h“)) {
        …
   }
}
```

- *compare to* C++*:*

```
int Counter;
string Name;

void SendStatement()
{
   if(this->Name !=
   "Smith") {
        …
   }
}
```

Variables

- **Memory**
  - Transient variable of an instance
  - ; aCar(Def Version:2) (Implem Version:2)

```
memory Counter : Int4
Name : CString

procedure SendStatement
    self.Counter++
    …
```

  - Counter is never stored in data base
  - Only has meaning for descendents of aFullObject
  - Memory implies that this variable should not be stored in the database. Such temporary variables often display dynamic data derived from calculated results from functions or queries.

Variables

- **Absolute**
  - Declare a variable occupying the same physical memory as another variable.
  - Avoids recasting a variable.
  - ```
    type tByteArray : array [0 to 255] of Int1
    var UpCaseString : CString
    var ByteArray : tByteArray absolute UpCaseString
    ```
    ```

Variables

- *in* GOLD*:*

```
type tByteArray :
   array [0 to 255]
   of Int1

var UpCaseString :
   CString

var ByteArray :
   tByteArray
   absolute
   UpCaseString
```

- *compare to* Java*:*

N/A!

- *compare to* C++*:*

```
typedef char[256]
   tByteArray;


union ucs {
char
   UpCaseString[256];
tByteArray ByteArray;
};
```

- **Override**
  - The override keyword indicates that an instance variable is redefined as a compatible type in a descendent class.
    - In the parent class aPerson, the MarriedTo variable is defined with the following code:
    - `MarriedTo : refTo aPerson`
    - In the descendent class aWoman, the MarriedTo variable is overridden with the code:
    - `MarriedTo : refTo aMan override`

Variables

- **Parameter**
  - By value
    - The original value cannot be affected.
    - It's the default. No parameter modifier is needed.
  - `procedure WriteDate(SqlServerDate : tSqlServerDate, Format : tDateFormat)`
  - By reference
    - The original value can be affected.
    - The parameter modifier inOut is needed.
  - `function ReadCarString(inOut theLicensePlate : tLicensePlate) return CString`

Variables

- *in* GOLD*:*

```
procedure
   WriteDate(SqlServ
   erDate :
   tSqlServerDate,
   Format :
   tDateFormat)


function
   ReadCarString(
   inOut
   theLicensePlate :
   tLicensePlate)
   return CString
```

- *compare to* Java*:*

```
Pass by value

Pass by reference
```

- *compare to* C++*:*

```
void WriteDate(
   tSqlServerDate
   SqlServerDate,
   tDateFormat Format)


string ReadCarString(
   tLicensePlate
   &theLicensePlate)
```

- **Parameter (2)**
  - Special Parameters
    - Self is automatically declared in methods. It represents the 'current' instance on which the method has been invoked. It doesn't exist in module procedures and functions.
    - `Self.Day = Monday`
    - _Result is automatically declared in all functions. It represents the returned value of a function. When function terminates, the value of _Result is returned.
    - `_Result = 1`

Variables

- *in* GOLD*:*

```
Self.Day = Monday

_Result = 1
```

- *compare to* Java*:*

```
this.day = MONDAY;

[_Result has no analog
   in Java. You could
   create a variable
   called result, and
   include:
return result;
   before just before
   the function ends.]
```

- *compare to* C++*:*

```
this->Day = Monday;

[_Result has no analog
   in C++.  You could
   create a variable
   called result, and
   include:
return result;
   before just before
   the function ends.]
```

Variables

- **Local**
  - Declared in procedures and functions
  - Exist only inside method
  -
```
procedure TestVar
    var Counter : Int4
    var Name : Cstring

    Counter = 1
    Name = 'Bill'
endProc
```

Variables

- *in* GOLD*:*

```
procedure TestVar
var Counter : Int4
var Name : Cstring
Counter = 1
Name = 'Bill'
endProc
```

- *compare to* Java*:*

```
void testVar() {
    int counter;
    String name;

    counter = 1;
    name = "Bill";
}
```

- *compare to* C++*:*

```
void TestVar()
{
    int Counter;
    string Name;

    Counter = 1;
    Name = "Bill";
}
```

Variables

- **Global**
  - Member variable of a Module
  - ; Banks(Def Version:2) (Implem Version:2)
    TotalCount : Int4
    LastTreatedClient : Cstring
  - Accessed via module name

```
; aBank(Def Version:2) (Implem Version:2)
  procedure SendStatement
    if self.curClient <> Banks.LastTreatedClient
        …
    endIf
```

Variables

- *in* GOLD*:*

```
; Banks(Module…)
TotalCount : Int4
LastTreatedClient :
    Cstring

; aBank(Class…)
procedure SendStatement
if self.curClient <>
    Banks.LastTreatedCli
    ent
    […]
endIf
endproc
```

- *compare to* Java*:*

```
class Banks {
    static int
    totalCount;
    static String
    lastTreatedClient;
}

class ABank {
[…]
    void sendStatement()
    {
    if(!curClient.equals(
    Banks.lastTreatedClie
    nt)) {
          […]
    }
    }
}
```

- *compare to* C++*:*

```
[in header]
extern int TotalCount;
extern string
    LastTreatedClient;

[in module cpp]
string LastTreatedClient
    = "";

[in class cpp]
void SendStatement()
{
    if(this->curClient
    !=Banks.LastTreatedCl
    ient) {
          […]
    }
}
```

Methods

- Method Types
  - Procedure (does not return a variable)
  - Function (returns a variable)
- Keywords
  - Override
  - External
  - Forward

Methods

- **Procedure**
  - A procedure is a routine in a class or a module that does not return a value.
-
```
procedure ResetInsurancePremiumTotal

    Self.PremiumTotal = 0
endProc
```

Methods

- *in* GOLD*:*

```
procedure
   ResetInsurancePre
   miumTotal
   Self.PremiumTotal
   = 0
endProc
```

- *compare to* Java*:*

```
void
   resetInsurancePremi
   umTotal() {
   premiumTotal = 0;
}
```

- *compare to* C++*:*

```
void
   ResetInsurancePremi
   umTotal()
{
   PremiumTotal = 0;
}
```

Methods

- **Function**
  - A function is a routine in a class or a module that returns a value.
- ```
  function Tax return Num4
      _Result = Self.Price * TaxPercent
  endFunc
  ```
  ```

Methods

- *in* GOLD*:*

```
function Tax return
   Num4
   _Result =
   Self.Price *
   TaxPercent
endFunc
```

- *compare to* Java*:*

```
float tax() {
   return price *
   taxPercent;
}
```

- *compare to* C++*:*

```
float Tax()
{
return Price *
   TaxPercent;
}
```

Methods

- **Override**
  - The override keyword indicates that a method is being specialized in a descendent class.
  - ```
    procedure PrintColor override
      inherited Self.PrintColor
      WriteLn('Here we can implement behavior
      specific to this subclass')
    endProc
    ```
  - ```
    Function GetTotal(pUnits:int4,pQuanitity: int4)
    override
      _result = inherited self.GetTotal
    end
    ```

Methods

- *in* GOLD*:*

```
procedure PrintColor
    override
inherited
    Self.PrintColor
Write('Here we can
    implement behavior
    specific to this
    subclass')
endProc
```

- *compare to* Java*:*

```
@Override
void printColor() {
    super.printColor();
    System.out.print("Her
    e we can implement
    behavior specific to
    this subclass");
}
```

- *compare to* C++*:*

```
class Parent {
    […]
    void PrintColor();
};

class Child : public
    Parent
{
    […]
void PrintColor()
    {
    Parent::PrintColor();
    cout << "Here we can
    implement behavior
    specific to this
    subclass";
    }
};
```

Methods

- **External**
  - The external keyword lets you declare routines you can call in external DLLs.
  - It gives you access to other libraries.
  - External methods must be declared in Modules
  - `function Encrypt(theString : CString) return CString external 'Encrypt.EncryptStr'`

- **Forward**
  - The forward keyword lets you declare a method before you specify its implementation. This keyword is used when one method calls a second method which is not implemented until after the first method.
  - `procedure Display(i : Int4, j : Int4) forward`
  -
```
procedure Show(x : Num4, y : Num4)
    Self.Display(4, 5)
endProc
procedure Display(i : Int4, j : Int4)
    Self.Show(5.2, 6.3)
endProc
```

Methods

- *in* GOLD*:*

```
procedure Display(i
   : Int4, j : Int4)
   forward

procedure Show(x :
   Num4, y : Num4)
   Self.Display(4,
   5)
endProc


procedure Display(i
   : Int4, j : Int4)
   Self.Show(5.2,
   6.3)
endProc
```

- *compare to* Java*:*

Not needed.

- *compare to* C++*:*

```
void Display();



void Show(float x,
   float y)
{
   this-
   >Display(4,5);
}

void Display(int i,
   int j)
{
   this->Show(5.2,
   6.3);
}
```

Statements

- **Basic Statements**
  - Comments, Assignment, Special Assignment, If, While, For, Repeat, Loop, Switch, Return, Continue, Exit.
- **Advanced Statements**
  - ForEach, OQL Select

Basic Statements

- **Comments**
  - `; Your comments ...`
  - Only valid inside code. Comments for vars, methods, etc. should be linked to those entities.
- **Assignment**
  - The assign statement '=' assigns an expression to an l-value.
  - ```
    i = 12
    i = 123*4 + 5
    s = 'First letters of the alphabet : '
    s = s + 'abcdef'
    ```
    ```

Basic Statements

- *in* GOLD*:*

```
i = 12
i = 123 * 4 + 5

s = 'First letters
   of the alphabet :
   '
s = s + 'abcdef'
```

- *compare to* Java*:*

```
i = 12;
i = 123 * 4 + 5;

s = "First letters of
   the alphabet : ";
s = s + "abcdef";
```

- *compare to* C++*:*

```
i = 12;
i = 123 * 4 + 5;

s = "First letters of
   the alphabet : ";
s = s + "abcdef";
```

Basic Statements

- **Special Assignment**
  - The '++' statement increments a variable
    - `i++ ; i = i + 1`
  - The '--' statement decrements a variable
    - `i-- ; i = i - 1`
  - The "+=" statement increments a variable by another value.
    - `i+= 5 ; i = i + 5`
  - The "-=" statement decrements a variable by another value.
    - `I-= 5 ; i = i - 5`

Basic Statements

- **If**
  - The if statement allows conditional execution of a sequence of statements.
  - ```
    if City = 'Paris'
     WriteLn('This train is leaving from
     Paris.')
    elseif City = 'London'
     WriteLn('This train is leaving from
     London.')
    else
     WriteLn('This train is not leaving from
     Paris or London.')
    endIf
    ```
    ```

Basic Statements

- *in* GOLD*:*

```
if City = 'Paris'
    WriteLn('This train
    is leaving from
    Paris.')
elseif City = 'London'
    WriteLn('This train
    is leaving from
    London.')
else
    WriteLn('This train
    is not leaving from
    Paris or London.')
endIf
```

- *compare to* Java*:*

```
if(city.equals("Paris"))
    {
    System.out.println("T
    his train is leaving
    from Paris.");
} else if
    (city.equals("London"
    )) {
    System.out.println("T
    his train is leaving
    from London.");
} else {
    System.out.println("T
    his train is not
    leaving from Paris or
    London.");
}
```

- *compare to* C++*:*

```
if(City == "Paris") {
    cout << "This train
    is leaving from
    Paris." << endl;
} else if (City =
    "London") {
    cout << "This train
    is leaving from
    London." << endl;
} else {
    cout << "This train
    is not leaving from
    Paris or London." <<
    endl;
}
```

- **For**
  - The for statement repeats a sequence of statements. This sequence statement is executed each time a counter variable is incremented or decremented until it goes beyond its limit value.
  - ```
    for i = 10 to 30 step 2
      WriteLn(i)
    endFor
    ```
    ```

Basic Statements

- **While**
  - The while statement executes a sequence of statements as long as value of an expression is true (or the sequence is exited by a break, return, or exit statement).
  - ```
    while (Amount > 0)
      Amount -= Self.Withdraw(Amount)
      AccountType++
    endWhile
    ```
    ```

Basic Statements

- *in* GOLD*:*

```
while (Amount > 0)
    Amount -=
    Self.Withdraw(
    Amount)
    AccountType++
endWhile
```

- *compare to* Java*:*

```
while (amount > 0) {
    amount -=
    withdraw(amount);
    accountType++;
}
```

- *compare to* C++*:*

```
while(Amount > 0) {
    Amount -= this->
    Withdraw(Amount);
    AccountType++;
}
```

Basic Statements

- **Repeat**
  - The repeat statement repeats a sequence of statements until a terminating expression evaluates to True.
  - The until keyword specifies the terminating expression of the repeat statement.
  - ```
    repeat
      Sum += theArray[i]
      i = wUtil.RANDOM(27)
    until i > 25
    ```
    ```

Basic Statements

- *in* GOLD*:*

```
for i = 10 to 30
   step 2
   WriteLn(i)
endFor
```

- *compare to* Java*:*

```
for(i = 10; i <= 30;
   i += 2) {
   System.out.printl
   n(i);
}
```

- *compare to* C++*:*

```
for(i = 10; i <= 30;
   i+=2) {
   cout << i <<
   endl;
}
```

Basic Statements

- *in* GOLD*:*

```
repeat
Sum += theArray[i]
i = wUtil.RANDOM(27)
until i > 25
```

- *compare to* Java*:*

```
Random r = new
   Random();

do {
   sum +=
   theArray[i];
   i =
   r.nextInt(27);
} while (i <= 25);
```

- *compare to* C++*:*

```
do {
   Sum +=
   theArray[i];
   i = rand() % 27;
} while (i <= 25);
```

- **Loop**
  - The loop statement creates an infinite loop. The only way to exit a loop is to have a break, return, or exit statement in the loop statement.
  - The endLoop keyword terminates the loop statement.
  -
```
loop
  if a > 10
    break
  endIf
  a++
endLoop
```

Basic Statements

- *in* GOLD*:*

```
loop
   if a > 10
         break
   endIf
   a++
endLoop
```

- *compare to* Java*:*

```
do {
   if (a > 10) {
         break;
   }
   a++;
} while (true);
```

- *compare to* C++*:*

```
do {
   if (a > 10) {
         break;
   }
   a++;
} while (true);
```

- **Switch**
  - The switch statement transfers control to one of several groups of statements depending on the value of an expression.
  - The endSwitch keyword terminates the switch statement.
  - The when and endWhen keywords are used to specify statement(s) for a particular value of the expression.
  - The Else statement can be used in a switch

Basic Statements

- **Switch (2)**

```
switch Self.ModeOfTransportation
     when byLand
       WriteLn('We will arrange a car for you')
     endWhen
     when byAir
       WriteLn('We will arrange an flight for you')
     endWhen
     when bySea
       WriteLn('We will arrange a Boat for you')
     endWhen
     else WriteLn('Transport Mode not recognized')

endSwitch

Switch thePerson.Name
     when 'Bill', 'Bob'
       WriteLn('You''re name starts with B')
     endWhen
     when 'Kate'
       WriteLn('You''re Great')
     endWhen
     else
       WriteLn('No cleverness coded for your name')
endSwitch
```

Basic Statements

- *in* GOLD*:*

```
switch
Self.ModeOfTransport
      when byLand
        WriteLn('We will
arrange a car for you')
      endWhen
      when byAir
        WriteLn('We will
arrange an flight for
you')
      endWhen
      when bySea
        WriteLn('We will
arrange a Boat for you')
else WriteLn('Transport
Mode not recognized')
      endWhen

endSwitch
```

- *compare to* Java*:*

```
switch (ModeOfTransport) {
case byLand:
    System.out.println(" We
    will arrange a car for you
    ");
    break;
case byAir:
    System.out.println(" We
    will arrange an flight for
    you ");
    break;
case bySea:
    System.out.println(" We
    will arrange an flight for
    you.");
    break;
default:
    System.out.println("
    Transport Mode not
    recognized "); }
```

- *compare to* C++*:*

```
switch (ModeOfTransport)
{
    case byLand:
          cout << " We will
    arrange a car for you " <<
    endl;
          break;
    case byAir:
          cout << " We will
    arrange an flight for you
    " << endl;
          break;
    case bySea:
          cout << " We will
    arrange an flight for
    you." << endl;
          break;
    default:
          cout << " Transport
    Mode not recognized " <<
    endl;
}
```

Basic Statements

- **Return**
  - The return statement exits a function and returns a value. Takes precedence over _Result.
  -
```
function GetFullName return CString
 if Self.IsMale
  return 'Mr. ' + Self.Name
 else
  return 'Ms. ' + Self.Name
 endIf
endFunc
```

Basic Statements

- *in* GOLD*:*

```
function GetFullName
   return CString
if Self.IsMale
   return 'Mr. ' +
   Self.Name
else
   return 'Ms. ' +
   Self.Name
endIf
endFunc
```

- *compare to* Java*:*

```
##
```

- *compare to* C++*:*

```
string GetFullName()
{
if(this->isMale()) {
   return
   string("Mr. ") +
   this->Name;
} else {
   return
   string("Ms. ") +
   this->Name;
}
}
```

- **Continue**
  - The continue statement forces execution to cycle the currently executing 'loop' (repeat, while, for, forEach, loop) statement.
  - The continue statement jumps over the remaining statements and passes to the next iteration of the 'looping' statement.
  -
```
for i = 0 to 20
  if i = 10
    continue
  endIf
  WriteLn(i)
endFor
```

Basic Statements

- *in* GOLD*:*

```
for i = 0 to 20
   if i = 10
         continue
   endIf
   WriteLn(i)
endFor
```

- *compare to* Java*:*

```
##
```

- *compare to* C++*:*

```
for(i = 0; i <= 20;
   i++) {
   if (i == 10) {
         continue;
   }
   cout << i << endl;
}
```

Basic Statements

- **Exit**
  - The exit statement exits the currently executing procedure or function. Note that exit works in both functions and procedures, whereas return only works for functions.
    - Note: We recommend you not use the exit statement. Better programming style would be to use appropriate if constructs.
  -
```
for a = 0 to 1000
 if a > 100
   exit
 endIf
endFor
```

Basic Statements

- *in* GOLD*:*

```
for i = 0 to 1000
   if i > 100
         exit
   endIf
endFor
```

- *compare to* Java*:*

```
##
```

- *compare to* C++*:*

```
for(i = 0; i <= 1000;
   i++) {
   if (i > 100) {
         return;
   }
}
```

Basic Statements

- **Break**
  - The break statement exits the current loop (for, while, repeat, forEach, loop)
  -
```
for a = 0 to 1000
  if TestValue(a) = cError
    break
  endIf
endFor
```

Basic Statements

- *in* GOLD*:*

```
for a = 0 to 1000
   if TestValue(a) =
   cError
         break
   endIf
endFor
```

- *compare to* Java*:*

```
##
```

- *compare to* C++*:*

```
for(a = 0; a <= 1000;
   a++) {
   if (TestValue(a) ==
   cError) {
         break;
   }
}
```

Break?

- **ForEach**
  - The forEach statement executes a sequence of statements for each element of a traversable list.
  - The keyword endFor terminates a forEach statement.
  - Commonly used for listof variables and OQL selects, but exists for other listable entities.
  -
```
forEach curVehicle in Self.myVehicles
  _Result += curVehicle.CalculateInsurance
endFor
```

- **ForEach using Rank**
  - Sometimes it useful to know the rank of the object in the list that is in the current iteration. The **using** keyword will cause the rank to be “carried” by a int4 variable of your choosing.

```
Var Rank : int4


forEach curVehicle in Self.myVehicles using Rank
    _Result += curVehicle.CalculateInsurance
endFor
```

Advanced Statements

- *in* GOLD*:*

```
forEach curVehicle
   in
   Self.myVehicles
_Result +=
   curVehicle.Calcul
   ateInsurance
endFor
```

- *compare to* Java*:*

```
##
```

- *compare to* C++*:*

```
for(curVehicle =
   myVehicles.begin();
   curVehicle !=
   myVehicles.end();
   curVehicle++) {
result += curVehicle->
   CalculateInsurance(
   );
}
```

Advanced Statements

- OQL Select
  - OQL Select is used to create queries of your data base based on business criteria (instead of data base structure). The query is formulated using the OQL language.
  -
```
OQL select * from curPerson in aPerson++ where
curPerson.LastName like 'S' using cursor
```
  -
```
forEach curInfo in OQL select x.FirstName, x.Name,
x.Age from x in aPerson++ where x.Name like 'S'
 WriteLn(curInfo.Name, ' is ', curInfo.Age, '
years old')
endFor
```

Intrinsics

- **WAM Intrinsics**
  - New, Dispose
  - First, Last, Ord, Succ, Pred
  - Write, Writeln, Concat, Str
  - Length, SizeOf
  - UpCase
  - Member

Intrinsics

- **New**
  - The new intrinsic allocates an instance of a class or a pointed element of a pointer type. When creating new instances, the init method is called.
  -
```
var thePerson : aPerson

new(thePerson)
thePerson.FirstName = 'John'
```

Intrinsics

- *in* GOLD*:*

```
var thePerson :
   aPerson

new(thePerson)
thePerson.LastName =
   'John'
```

- *compare to* Java*:*

```
##
```

- *compare to* C++*:*

```
aPerson *thePerson;

thePerson = new
   aPerson;
thePerson -> LastName
   = "John";
```

- **Dispose**
  - The dispose intrinsic deletes an instance or a pointer and frees any memory it occupies.
  -
```
var theBoat : aBoat
var pString : .String

new(pString)
pString. = 'Hello!'
WriteLn(pString.)
dispose(pString)

new(theBoat)
theBoat.Name = 'Chris Craft'
WriteLn(theBoat)
dispose(theBoat)
```

Intrinsics

- *in* GOLD*:*

```
var theBoat : aBoat
var pString :
    .String

new(pString)
pString. = 'Hello!'
WriteLn(pString.)
dispose(pString)

new(theBoat)
theBoat.Name =
    'Chris Craft'
WriteLn(theBoat)
dispose(theBoat)
```

- *compare to* Java*:*

```
##
```

- *compare to* C++*:*

```
aBoat theBoat;
string *pString;

pString = new string;
*pString = "Hello!"
cout << *pString <<
    endl;
delete pString;

theBoat = new aBoat;
theBoat.Name = "Chris
    Craft";
cout << theBoat->Name
    << endl;
delete theBoat;
```

- **First**
  - The first intrinsic returns the first element of an ordinal type. First is most often used for enum types, for which it returns the first enum constant in the type, but can also be used for ints or subranges for which it returns the minimum value.
  - ```
    type tWorkDay : (Mon, Tue, Wed, Thu, Fri)
    var Today : tWorkDay

    if Today = first(tWorkDay)
     WriteLn('I hope you had a good weekend')
    endIf
    ```
    ```

- **Last**
  - The last intrinsic returns the last element of an ordinal type. Last is most often used for enum types, for which it returns the last enum constant in the type, but can also be used for ints or subranges for which it returns the maximum value.
  - ```
    type tWorkDay : (Mon, Tue, Wed, Thu, Fri)
    var Today : tWorkDay

    if Today = last(tWorkDay)
     WriteLn('Have a good weekend!')
    endIf
    ```
    ```

Intrinsics

- **Ord**
  - The ord intrinsic returns the integral value of an element in an ordinal type.
  - ```
    type tCar : (Ferrari, Porsche, Jaguar)
    procedure TestOrdInEnum
     var Car : tCar
     Car = succ(Porsche)
     WriteLn(ord(Car))
    endProc
    ```
  - Ord is Zero based and works much like Rank in a list.
    ```

Intrinsics

- *in* GOLD*:*

```
type tCar :
   (Ferrari,
   Porsche, Jaguar)

procedure
   TestOrdInEnum
   var Car : tCar
   Car =
   succ(Porsche)
   WriteLn(ord(Car))
endProc
```

- *compare to* Java*:*

```
##
```

- *compare to* C++*:*

```
typedef enum {
   Ferrari,
   Porsche,
   Jaguar
} tCar;

void TestOrdInEnum()
{
tCar Car;
Car = Porsche++;
cout << int(Porsche)
   << endl;
}
```

- **Pred**
  - The pred intrinsic returns the predecessor of an element of an ordinal type.
    - Note: The predecessor of the first element of an ordinal type is the first element of that type.
  -
```
type tCarCategory : (SubCompact, Compact, MidSize, Luxury)
var RequestedCar : tCarCategory

WriteLn('Sorry, there is no ', RequestedCar, ' cars.')
WriteLn('Would you like a ', pred(RequestedCar), ' car?')
```

- **Succ**
  - The succ intrinsic returns the successor of an element of an ordinal type.
    - Note: The successor of the last element of an ordinal type is the last element of that type.
  -
```
type tCarCategory : (SubCompact, Compact, MidSize, Luxury)
var RequestedCar : tCarCategory

WriteLn('Sorry, there is no ', RequestedCar, ' cars.')
WriteLn('Would you like a ', succ(RequestedCar), ' car?')
```

- **Write**
  - Write converts a list of writeable elements to a string and either adds it to a text variable or writes it to the Standard Report window.
    - Note: Write does not advance to the next line.
  -
```
var Profession : CString

Profession = 'writer'
Write(Self.myText, Self, ' is the greatest ',
Profession, ' of all time, ')
Write(Self.myText, 'even though he has only
written ', Self.Books.count, ' books')
```

Intrinsics

- *in* GOLD*:*

```
var Profession :
   CString

Profession =
   'writer'
Write(Self.myText,
   Self, ' is the
   greatest ',
   Profession, ' of
   all time, ')
Write(Self.myText,
   'even though he
   has only written
   ',
   Self.Books.count,
   ' books')
```

- *compare to* Java*:*

```
##
```

- *compare to* C++*:*

```
string profession;
ostringstream stream;

profession = "writer";

stream << *this << "
   is the greatest "
   << profession << "
   of all time, ";
stream << "even though
   he has only written
   ", this-
   >Books.size() << "
   books"

this->myText = stream;
```

- **WriteIn**
  - WriteLn converts a list of writeable elements to a string and either adds it to a text variable or writes it to the Standard Report window.
    - Note: WriteLn advances to the next line.
  -
```
var Title : CString

Self.Name = 'James Joyce'

Title = 'Finnegans Wake'
WriteLn(Self.Name, ' wrote ', Title)
WriteLn('It''s a whale of a book.')
```

Intrinsics

- *in* GOLD*:*

```
var Title : CString

Self.Name = 'James
   Joyce'

Title = 'Finnegans
   Wake'
WriteLn(Self.Name, '
   wrote ', Title)
WriteLn('It''s a
   whale of a
   book.')
```

- *compare to* Java*:*

```
##
```

- *compare to* C++*:*

```
string title;

name = "James Joyce";

title = "Finnegans
   Wake"

cout << name <<
   "wrote" << title <<
   endl;
cout << "It's a whale
   of a book.";
```

- **Concat**
  - Concat converts a list of writeable elements to a string and returns the string.
  - ```
    var CS : CString
    var UnitsOrdered : Int4
    var Dollars : Num8

    type tProduct : (e-WAM, WydeWeb, WydeFramework)
    var Product : tProduct
    UnitsOrdered = 25
    Dollars = 554016.15
    Product = e-WAM
    CS = Concat(Self, ' has ordered ', UnitsOrdered, '
    of ', Product, ' for the sum of : $', Dollars)
    ```
    ```

Intrinsics

- *in* GOLD*:*

```
var CS : CString
var UnitsOrdered : Int4
var Dollars : Num8

type tProduct : (eWAM,
    WydeWeb,
    WydeFramework)
var Product : tProduct
UnitsOrdered = 25
Dollars = 554016.15
Product = e-WAM
CS = Concat(Self, ' has
    ordered ',
    UnitsOrdered, ' of
    ', Product, ' for
    the sum of : $',
    Dollars)
```

- *compare to* Java*:*

```
##
```

- *compare to* C++*:*

```
string cs;
int unitsOrdered;
double dollars;

typedef enum {
    eWAM, WydeWeb,
    WydeFramework
} tProduct;
tProduct product;

unitsOrdered = 25;
dollars = 554016.15;
product = eWAM;
cs = string(*this) + "
    has ordered " +
    unitsOrdered + " of "
    + product + " for the
    sum of : $" +
    dollars;
```

- **Str**
  - Str converts a list of writeable elements to a string and stores it in the first parameter.
  - ```
    var CS : CString
    var UnitsOrdered : Int4
    var Dollars : Num8
    type tProduct : (e-WAM, WydeWeb, WydeFramework)
    var Product : tProduct

    UnitsOrdered = 25
    Dollars = 554016.15
    Product = e-WAM
    Str(CS, Self, ' has ordered ', UnitsOrdered, ' of
    ', Product, ' for the sum of : $', Dollars)
    ```
    ```

Intrinsics

- **Length**
  - The length intrinsic function returns the length of a string expression or the length of the text contained in a Text variable.
  -
```
var Message : Text


if length(Message) < 255
  …
endIf
```

Intrinsics

- *in* GOLD*:*

```
var Message : Text

if length(Message)
   < 255
   […]
endIf
```

- *compare to* Java*:*

```
##
```

- *compare to* C++*:*

```
string message;

if(message.length() <
   255) {
   […]
}
```

Intrinsics

- **Sizeof**
  - The sizeof intrinsic function returns the size in bytes of a type or a variable (the size of the variable's type).
  - Different from Length for strings
  - ```
    procedure InitData
       Self.Buffer = GetMem(sizeof(tHouse))
    endProc
    ```
    ```

- **Upcase**
  - The upcase intrinsic function returns a given string converted to upper case.

```
LastNameArg: Cstring
```

  - OQL `select * from x in aPerson++ where upcase(x.LastName) like upcase(LastNameArg)`

  -
```
function IsGoodPerson return Cstring
   return Upcase(Self.Name) ;would return JOYCE
endFunc
```

Intrinsics

- *in* GOLD*:*

```
function
   IsGoodPerson
   return Boolean
return
   Upcase(Self.Name)
   = 'JOYCE '
endFunc
```

- *compare to* Java*:*

##

- *compare to* C++*:*

##

- **Member**
  - The member intrinsic allows you to determine if an instance is a descendent of the specified class.
    - Note: One should use the **member** intrinsic sparingly. In most cases can be replaced by the object-oriented technique of a method being overridden for the class.

- **Member (2)**
  - `var thePerson : aPerson`

```
thePerson = Self.FindPerson
if member(thePerson, aClient)
 WriteLn('Thank you, ', thePerson.LastName, ' for
your order of e-WAM’)
else
 WriteLn('Thank you, ', thePerson.LastName, ' for
your interest in e-WAM')
endIf
```

Intrinsics

- *in* GOLD*:*

```
var thePerson : aPerson

thePerson =
    Self.FindPerson
if member(thePerson,
    aClient)
    WriteLn('Thank you,
    ',
    thePerson.LastName,
    ' for your order of
    e-WAM’)
else
    WriteLn('Thank you,
    ',
    thePerson.LastName,
    ' for your interest
    in e-WAM')
endIf
```

- *compare to* Java*:*

```
##
```

- *compare to* C++*:*

```
aPerson *thePerson;

thePerson = this->
    FindPerson();

if(dynamic_cast<aClient
    *>(thePerson)) {
cout << "Thank you" <<
    thePerson.lastname <<
    "for your order of e-
    WAM" << endl;
} else {
cout << "Thank you" <<
    thePerson.lastname <<
    "for your interest in
    e-WAM" << endl;
}
```

Keywords / Misc

- **Encapsulation**:
  - Private,
  - Protected,
  - Final
- **Keywords**
  - Uses
  - Pass/Inherited
  - Recast

Keywords / Misc

- **Modifiers keywords**
  - The access modifiers protected and private can be used for each class definition and each attribute (variable or method) of your class.

Keywords / Misc

- **Private**
  - The private keyword is used to specify that a class, a method (in a class), a routine (in a module), an instance variable, a module variable, a constant defined in a module, or a constant defined in a class will not be usable from anywhere else but in the compilation unit (class or module) in which it is defined.

Keywords / Misc

- **Private (2)**
  - `const BigSecret = 55 private`

```
Accessible : Int4
NeverSeen : Int4 private

class aPrivateToModule(aFullObject) private
 Age : Int4
endClass

class aVisible(aFullObject)
 Visible : Int4
 NotVisible : Int4 private

 procedure NotInvocableOutside private
   ...
 endProc
endClass
```

Keywords / Misc

- **Protected**
  - The protected keyword is used to specify that a class, a method (in a class), a routine (in a module), an instance variable, a module variable, a constant defined in a module, or a constant defined in a class will only be reachable from the compilation unit (class or module) in which it is declared, and any descendents of the compilation unit in the case of classes.

Keywords / Misc

- **Protected (2)**
  - class aBusinessman(aPerson)

```
const BigSecret = 'Didn't pay Taxes' protected

IsMarried : boolean
ClaimedMaritalStatus : boolean
FinancialDocuments : Text protected
Diary : Text private

procedure EmbezzleFunds Private
    ...
endProc

procedure ActNatural
    ...
endProc
```

Keywords / Misc

- *compare to* Java*:*

##

- *compare to* C++*:*

```
class aBusnessman : public
   aPerson
{
protected:
   const BigSecret = "Didn\'t
   pay Taxes";

   bool IsMarried;
   bool ClaimedMaritalStatus;
   string FinancialDocuments;

private:
   string Diary;

   EmbezzleFunds() { ... };

   ActNatural() { ... };
};
```

Keywords / Misc

- **Final**
  - The final keyword is used to specify that a method, variable or a class cannot be overridden.
  -
```
NonOverrideableVar : Int4 final

procedure NonOverridable final
  ...
endProc
```

Keywords / Misc

- **Uses**
  - The uses keyword indicates which modules and classes your entity (class, module, or method) references.
  - Is used to keep meta-model consistency.
  - Added and removed automatically.
  - Usage

```
uses aPerson, BillingModule, wMath
theClient : refto aClient
…
```

Keywords / Misc

- *compare to* Java*:*

```
##
```

- *compare to* C++*:*

```
#include “aperson.hpp”
#include “billing.hpp”
#include “wmath.hpp”

[…]
 aClient *theClient;
[…]
```

Keywords / Misc

- **Pass / Inherited**
  - Inherited calls the method for the parent class from within an overridden method
  - ```
    procedure PrintColor override
     inherited Self.PrintColor
     WriteLn('Here we can implement behavior
     unique to this subclass')
    endProc
    ```
  - Pass automatically expands to inherited
  - ```
    procedure PrintColor override
     pass  (expands to inherited Self.PrintColor)
     WriteLn('Here we can implement behavior
     unique to this subclass')
    endProc
    ```

Keywords / Misc

- **Pass / Inherited in Functions**

  Inherited calls the method for the parent class from within an overridden method

```
function CalculateInsurancePremium return Num8 override
    _Result = inherited self.CalculateInsurancePremium
    if self.IsTurboCharged
        _Result += cTurboChargedpercentage *self.EngineDisplacement
    endIf
endFunc
```

Inheritance

- *in* GOLD*:*

```
procedure PrintColor
   override
inherited
   Self.PrintColor
WriteLn('Here we can
   implement
   behavior unique
   to this
   subclass')
endProc
```

- *compare to* Java*:*

```
##
```

- *compare to* C++*:*

```
void
   ChildClass::PrintCo
   lor()
{
ParentClass::PrintColo
   r();
cout << "Here we can
   implement behavior
   unique to this
   subclass" << endl;
}
```

Keywords / Misc

- **Recast**
  - Tells parser to change type of recasted expression
  - `aClient(thePerson).QuantityOrdered = 1000`

  - `myVehicle refto aVehicle`
  - `var theVehicle : aSportsVehicle`
  - `New(theVehicle)`
  - `aSportsVehicle(self.myVehicle).Turbocharged=true`

Recasting

- *in* GOLD*:*

```
tpCString(ptr). =
   'Kate'


aClient(thePerson).Q
   uantityOrdered =
   1000
```

- *compare to* Java*:*

```
##
```

- *compare to* C++*:*

```
*(dynamic_cast
   <string *> (ptr)) =
   "Kate";

dynamic_cast
   <aClient *>
   (thePerson)->
   QuantityOrdered =
   1000;
```

Important Methods

- **Important Methods**
  - Interact
  - Init, InitAfterLoad, InitAfterNewVersion, Terminate
  - Accept, CancelObject
  - NewVersion, Project
  - StringExtract

Important Methods

- **StringExtract**
  - Called when the system needs to know how to describe an instance.
  - The StringExtract has multiple “Kind”s that correspond to different contexts.
  - a “Title”, a “Full” description, Technical information, Help, …
  - Can be overridden if user is interested in a particular string extract (i.e. title, name, or label only etc.)

Important Methods

- **A,B,C's of versioning**
  - Step A: Get project for instance (new, NewVersion, Project)
  - Step B: Modify object
  - Step C: Accept or CancelObject

  - Accept: indicates that you have finished modifying instance. Version is put in transaction that might own it (if not in transaction of another, instance is saved in db).
  - CancelObject: Kills instance, leaving previous version (if there is one) as the current version of instance.

Important Methods

- **Projects**
  - A project is an instance that you can modify
  - New: intrinsic that allocates a new instance
  - NewVersion: method that creates new version (error if project) of accepted instance
  - Project: method that returns project if one exists, or calls NewVersion if not.

Important Methods

- **Initialization and termination**
  - Init: called after new instance allocated to initialize its attributes
  - InitAfterLoad: called after existing instance loaded from db to initialize its memory attributes
  - InitAfterNewVersion: called after new version created to initialize its attributes
  - Terminate: called just before instance is to be killed to deallocate any allocated resources

Important Methods

- **Interact**
  - Display the object to the user
  - return rValid when the user click on button OK
  - return rCancel when the user click on button Cancel

Thank you