Dynamic Objects Cheat Sheet

jBASE Getting Started Guide

https://docs.zumasys.com/jbase/dynamic-objects/

jBASE Dynamic Objects (jDO) is a subset of JabbaScript, the enhanced language extension to the jBASE development language, jBC (BASIC).

Objects

To activate JabbaScript so that Dynamic Objects can be used in a BASIC program, the program name either must have a **.jabba** extension or add the line **$option jabba** to the program. If the program is a subroutine or a function, then that line must be placed before the **SUBROUTINE** or **FUNCTION** line.

A Dynamic Object is created using the `new` keyword:

```
book = new object() ;* Create an object called `book`
```

Our **book** object is initially void so we can now add **properties** to the object and then display it:

```
$option jabba
include jabba.h
book = new object()
book->title = "Hitchhiker's Guide to the Galaxy"
book->author = "Douglas Adams"
book->copyright = 1979
book->pages = 215
crt book->$tojson(JABBA_TOJSON_VERBOSE + JABBA_TOJSON_USE_SPACE)
crt "Our object has ":book->$size():" properties."
```

When this program is run, it produces the following output:

```
{
    "title":"Hitchhiker's Guide to the Galaxy",
    "author":"Douglas Adams",
    "copyright":1979,
    "pages":215
}
Our object has 4 properties.
```

There are a few important things of which to take note:

- The exclusive format that Dynamic Objects uses is JSON.
- **properties** can be likened to attributes of a record but do not take the analogy too far.
- The properties of the object, e.g., title, author, copyright, and pages, belong to the object and will not conflict with other local variables of the same name. IOW, **book->title** is a different variable than **title**. The concept here is that the properties that belong to the object, **book** in this case, are only accessible to the object. This is called **encapsulation** or "data hiding" and is one of the most important concepts of Object-Oriented Programming (OOP). More on this later when introducing classes and methods.
- **$tojson()** and **$size()** are two of the many built-in methods that are part of Dynamic Objects. **$tojson()** is used to convert the object to JSON format. **$size()** returns the number of properties in the object.
- All built-in methods begin with a **$**.
- The **$tojson()** arguments, defined in the **jabba.h** header file, cause the object to be formatted. Without arguments, the object would be represented in a manner suitable for sending across the network or storing in a database, e.g.:

```
{"title":"Hitchhiker's Guide to the Galaxy","author":"Douglas Ad
ams","copyright":1979,"pages":215}
```

- The **->** symbol is the default object pointer, but can be changed to use a period, **.**, by changing the **$option** statement to:

```
•   $option jabba,period
```

Note

Use the command **jpp2 -?** to see all options available to the **$option** compiler directive as well as other ways to configure program behavior.

A JSON string can be converted to a Dynamic Object using the built-in **$fromjson()** method like so:

```
json = \{"title":"Hitchhiker's Guide to the Galaxy","author":"Douglas
 Adams","copyright":1979,"pages":215}\
book = json->$fromjson()  ;* convert a JSON string to an object
```

To make a local variable a property of an object, use the **@** indirection symbol in the same way as it is used to make an indirect subroutine CALL:

```
obj = new object()          ;* create an object
property = "name"           ;* assign a value to the local variable `p
roperty`
* assign the local variable 'property' to the object as the property
'name'
obj->@property = "Arthur"
print obj->name             ;* Displays "Arthur".
```

Objects, Classes and Methods

Dynamic Objects could not sport its name without being able to support Classes and methods.

The following examples are intentionally simple, so that we can focus on the language without getting diverted by complicated algorithms and data structures.

First, we need to define some nomenclature:

An **Object** is a structure that defines State and Behavior.

The **State** of an object are called the properties of the object.

The **Behavior** of an object are called methods which are procedures for operating on properties.

A **Class** is a template used to create objects.

When an object is created from a Class, that object is called an **instance** of the Class.

Let us create a bank account Class called **Checking**. Our Class will have 2 properties and 3 methods. All methods can be contained in the same record:

```
    class_checking.jabba
     method Checking::Checking(account_number)
         this->account_number = account_number
         this->balance = 0
     end method

     method Checking::deposit(amount)
         this->balance = this->balance + iconv(amount, "md2")
         return this
     end method

     method Checking::withdraw(amount)
         this->balance = this->balance - iconv(amount, "md2")
         return this
     end method

     method Checking::balance()
         return oconv(this->balance, "md2$")
     end method
```

The first method is the **constructor**. It is a Magic Method, as it is automatically invoked when the object is created with the **new** keyword. None of the other methods can be used until an instance of the Class is created.

Now let's create some client code to exercise our class:

```
    bank.jabba
     account_number = 123
*
* Create an instance of the Checking class
*
    account = new object("Checking", account_number)
*
* Make a few transactions.
*
    account->deposit(100.00)
    account->withdraw(10.00)
    crt "Account number ":account-
>account_number:" has a balance of ":account->deposit(500.00)-
>balance()
*
* Display the object (unformatted)
*
    crt account->$tojson()
```

Running this code produces the following output:

```
Account number 123 has a balance of $590.00
{"account_number":123,"balance":59000}
```

Note that we are accessing the properties of the object through the methods of the class. This is the Object Oriented (OO) concept of data hiding called **Encapsulation**. We could, of course, access the properties directly, e.g., **account->balance**, but this is not an accepted way to code in an Object-Oriented manner.

Inheritance

Dynamic Objects supports the OO concept of **Inheritance**. To illustrate this, we will first create another bank account class called **Savings**:

```
   class_savings.jabba
    method Savings::Savings(account_number)
        this->account_number = account_number
        this->balance = 0
    end method

     method Savings::deposit(amount)
        this->balance = this->balance + iconv(amount, "md2")
        return this
    end method

    method Savings::withdraw(amount)
        this->balance = this->balance - iconv(amount, "md2")
        return this
    end method

    method Savings::balance()
        return oconv(this->balance, "md2$")
    end method
```

Notice that both classes have exactly the same 3 methods: deposit(), withdraw(), balance()

We can refactor that code into a parent class we will call **BankAccount** and then inherit those methods in the Checking and Savings account child classes:

```
   class_bankaccount.jabba
    method BankAccount::BankAccount()
        this->balance = 0
    end method

    method BankAccount::deposit(amount)
        this->validate_amount(amount)
        this->balance = this->balance + iconv(amount, "md2")
        return this
    end method

    method BankAccount::withdraw(amount)
        this->validate_amount(amount)
        this->balance = this->balance - iconv(amount, "md2")
        return this
    end method

    method BankAccount::balance()
        return oconv(this->balance, "md2$")
    end method

    method BankAccount::validate_amount(amount)
        if not(amount matches "1N0N'.'2N") then
            throw "Invalid amount ":dquote(amount):" passed to an
instance of the ":this->$classname():" class."
        end
    end method

    class_checking.jabba
    method Checking::Checking(account_number)
        this->$inherit("BankAccount")
        this->account_number = account_number
    end method

    class_savings.jabba
    method Savings::Savings(account_number)
        this->$inherit("BankAccount")
        this->account_number = account_number
    end method
```

In Object-Oriented parlance, the BankAccount class would be considered abstract, in that you would never create an instance of it, only of its sub-classes, Checking and Savings.

We've also added a new method, **validate_amount**, to the BankAccount class which is called from the deposit() and withdraw() methods. If an invalid amount is passed to one of those methods, then it will throw an exception. Exceptions are another OO concept whereby error code is separated from the mainline code.

Here is where that rearrangement pays off. Let us say we now want to create another type of bank account, a Money Market account. All that is needed is to create the constructor and tell it to inherit from the BankAccount class:

```
   class_moneymarket.jabba
    method Moneymarket::Moneymarket(account_number)
        this->$inherit("BankAccount")
        this->account_number = account_number
    end method
```

Client code can now create three different types of bank accounts:

```
   ```
   bank1.jabba
    checking_account_number = "C123"
    savings_account_number = "S456"
    moneymarket_account_number = "M789"
*
* Create instances of each of the 3 classes
*
    checking_account = new object("Checking",
checking_account_number)
    savings_account = new object("Savings", savings_account_number)
    moneymarket_account = new object("Moneymarket",
moneymarket_account_number)
*
* Deposit a different amount in each account
*
    try
        checking_account->deposit("100.00")
        savings_account->deposit("200.00")
        moneymarket_account->deposit("300.00")
    catch ex
        print ex->message_long
        stop
    end try
*
* Display the balance in each account
023 *
    crt "Checking Account ":dquote(checking_account>account_number):"
has a balance of ":checking_account->balance()
    crt "Savings Account ":dquote(savings_account->account_number):"
has a balance of ":savings_account->balance()
    crt "Money Market Account
":dquote(moneymarket_account>account_number):" has a balance of
":moneymarket_account->balance()
```
```

Running this code produces the following output:

```
Checking Account "C123" has a balance of $100.00
Savings Account "S456" has a balance of $200.00
Money Market Account "M789" has a balance of $300.00
```

If we were to change the savings account deposit amount to "200", which is an invalid format, then running that code would now throw this error:

```
   Invalid amount "200" passed to an instance of the Savings class.
```

When an exception is thrown, an exception object is created, `ex` in the above example, and the code in the `catch` block is executed.

The exception object, like any other object, has various properties which get populated when the exception in thrown. For example, change line 18 to:

```
   crt ex->$tojson(1)    ;* Display the entire exception object
```

For details on Exception Handling, refer to:

[Exceptions: try / catch / throw / $setcatch() | Zumasys Documentation]