TADS 3 Language Reference

A QUICK GUIDE TO THE LANGUAGE

This is a quick-reference to common language features of the TADS 3 language. For the full story, see the *TADS 3 System Manual*.

Literals and Datatypes

`nil` and `true;` `nil` is false or an empty value.
Integer: -2147483658 to +2147483647
Hexadecimal `0xFFFF`
Enumerators `enum red, blue, green`
Property ID `&myProp`
List `[item1, item2, item3, item4, ... itemn]`
BigNumber `12.34` or `1.25e9` ; can store up to 65,000 decimal digits in a value between 10^{32767} and 10^{-32767}

String: a string is an ordered set of Unicode characters. A string constant is written by enclosing a sequence of characters in single quotation marks: `local str = 'Hello world! ';`
Strings can achieve special characters including:
`\"` - a double-quote mark
`\'` - a single-quote mark
`\n` - a newline character
`\b` - a “blank” line (paragraph break)
`\^` - a “capitalize” character; makes the next character capitalized
`\v` - a “miniscule” character, makes the next character lower case
`\ ` - a quoted space
`\t` - a horixontal tab
`\uXXXX` - the Unicode character XXXX (in hexadecimal digits)
`<.p>` - single paragraph break
`<q>` - smart opening quote mark “ or ‘
`</q>` - smart closing quote mark ” or ’

Identifiers

An identifier (object, class, function, property, method or variable name) must start with an alphabetic character or underscore followed by zero or more alphabetic characters, underscores, or the digits 0-9. The usual convention is that class names begin with a capital letter, and other identifiers with a lower case letter. Note that TADS 3 identifiers are case-sensitive.

Expressions and Operators

Arithmetic/logical operators:

`a + b`	addition
`a - b`	subtraction
`a * b`	multiplication
`a / b`	division
`a % b`	modulo (remainder)
`a++`	increments `a` by 1; evaluates to original value
`++a`	increments `a` by 1; evaluates to new value
`a--`	decrements `a` by 1; evaluates to original value
`--a`	decrements `a` by 1; evaluates to original value
`a += b`	equivalent to `a = a + b`
`a -= b`	equivalent to `a = a - b`
`a *= b`	equivalent to `a = a * b`
`a /= b`	equivalent `to a = a / b`
`a & b`	bitwise AND
`a | b`	bitwise OR
`a ? b : c`	if `a` is true evaluates to `b`, otherwise `c`

Conditional expressions, return true or nil (i.e. false)

`a == b`	`a` is equal to b
`a != b`	a is not equal to b
`a > b`	a is greater than b
`a < b`	a is less than b
`a >= b`	a is greater than or equal to b
`a <= b`	a is less than or equal to b
`a is in (x, y, z)`	`a` is equal to x, y or z
`a not in (x, y, z)`	a is not x, y or z

Boolean expressions, return true or nil (i.e. false)

`a && p`	both a and b are true (not nil or 0)
`a || b`	either or or b is true (not nil or 0)
`!a`	a is nil (false)

Object/class operators

`x = new MyClass`	dynamically create a new instance
`inherited`	invokes the method that the current method overrides
`delegated OtherClass`	like `inherited`, but invokes the corresponding method on `OtherClass`

Classes and Objects

To declare a class:

```
class MyClass: Class1, Class2, Class3...
   myProperty = 12
   myMethod(x)
   {
        myProperty = x;
   }
;
```

To declare an object

```
myObj: Class1, Class2 ...
    myProperty = 0
    myMethod(x)
    {
          myProperty = x;
    }
    myNestedObject: SomeClass { prop = 12 }
;
```

OR

```
myObj: Class1, Class2 ...
{
    myProperty = 0
    myMethod(x)
    {
          myProperty = x;
    }
    myNestedObject: SomeClass { prop = 12 }
}
```

Statements

Each statement is terminated by a semicolon “:”
A *statement_block* is a single statement or series of statements enclosed in braces {...}.
A pair of slashes, //, starts a comment; the rest of the line is ignored.
Anything between /* and */ is also a comment.

A common statement is the assigment:
`variable = *expr*;`

Use `local` to declare a local variable anywhere in a code block

Flow Control

To execute statements if *expr* is true; optionally, to execute other statements if *expr* is nil (false):

```
if(expr)
   statement_block

if(expr)
   statement_block
else
   statement_block
```

To execute statements depending on the value of *expr*:

```
switch(expr)
{
     case value1: statement; ... statement;
     case value2: statement; ... statement;
     ...
     default: statement; ... statement;
}
```

Note that an explicit `break` statement is needed to prevent fall-through.

Loop Control

To execute statement while *expr* is true:

```
while(expr)
   statement_block
```

To execute statements while *expr* is true, executing them at least once:

```
do
   statement_block
while(expr);
```

To execute statements while a variable changes:

```
for( initializer; condition; updater)
    statement_block
```

To execute statement for all objects in a list:

```
foreach (obj in list)
    statement_block
```

To jump out of the current innermost loop or switch:

```
 break;
```

To immediately start the next iteration of the current loop:

```
  continue;
```

Methods and Functions

To define a function:

```
function_name(param_name, param_name...)
{
    function_body
}
```

To replace or modify a function:

```
replace someFunc(a, b)
{
    // new code here
}
```

A method definition looks just like a function definition, except that it is attached to some object:

```
class MyClass: object
    getOwner()
   {
        // code goes here
   }
;
```

Shorthand method definition for a method that takes no parameters:

```
class MyClass: object
    getOwner = ( myOwner ?  myOwner.owner : nil)
```

Varying parameter lists:

```
printf(fmt, ....)
{
    // code goes here
}
```

Retrieve the *n*th argument with `getArg(n)`, `argcount` gives the total number of arguments.

Alternate form of varying parameter list:

```
printf(fmt, [lst])
{
    foreach(local x in lst)
       // do something
}
```

To return a value from a method or function:

```
  return expr;
```

To define an anonymous function:

```
  new function(x) { "x = <<x>>\n"; }
```

To define a short-form anonymous function:

```
  {a, b:  a + b}
```

N.B. a semicolon is not allowed in an anonymous function.

An anonymous function may be assigned to a variable or passed as an argument to a function call

Displaying Text

To output a list of values:

```
  say(value1, value2, .... value);
```

Where each value can be a string, an integer, a BigNumber, or nil.

To display a string:

```
  "string";
```

To display a string containing an embedded expression:

```
  "string <<expr>> text";
```

To change font attributes:

`<b> ... </b>` bold
`<i> ... </i>` italic
`<u> ... </u>` underline

Selected Intrinsic Functions

`dataType(val)` returns the data type of val as one of the TypeXXX values.
`firstObj(cls, flags?)` returns the first object of class *cls*.
`nextObj(obj, cls, flags?)` returns the next object after *obj* of class *cls*.
Use `firstObj()` and `nextObj()` together to iterate over all objects of a certain class in the game; flags is an optional parameter which you normally won't need to supply.
`rand(n)` returns a random number between 0 and n-1
`rand(val1, val2, ... valn)` or `rand([list])` randomly selects one of the list elements and returns it.
`randomize()` seeds the random number generator.
`toInteger(val)` converts *val* to an integer, where *val* can be an integer, string, BigNum, true or nil.
`toString(val)` converts *val* to a string.