# Dream Semantics

## Block Structure

The Dream language is a block-structured language. Blocks are created by 1) defining a class and 2) defining a method.

Identifiers are visible from the point of declaration to the end of the block in which they are defined. There are basically three scope levels in Dream:

- the global scope, containing predefined identifiers and class names
- class scope, containing a class's methods and attributes
- local scope, containing local variables and method parameters

Note that a method name can be used within the body of the method definition, for the purposes of recursion and the assignment of return values. Also, it is legal for a class definition to reference itself. For example:

```
class Foo is
  f: Foo ~ this is legal
  ...
end Foo
```

## Predefined Identifiers

Identifiers must be declared before they are used, with the exception of these predefined identifiers:

- `in` refers to an object of class Reader (defined in stdlib.Dream). The Reader class contains two methods:

  `readline(): string` - reads a line from standard input and returns it
  `readint(): int` - reads a line from standard input, converts it to an integer, and returns it

- `out` refers to an object of class Writer (defined in stdlib.Dream). The Writer class contains these methods:

  `writeint(num: int): Writer` - outputs base 10 representation of *num* followed by a newline; returns me
  `writechar(char: int): Writer` - outputs ASCII character for *char*; returns me
  `write(str: string): Writer` - outputs characters in *str*; returns me

  These methods print to standard output. writeint prints a newline after printing the integer; writechar and write do not.

- `me` refers to the current object instance. It can be used only in ID expressions (such as those on the right-hand side of an assignment statement, or in a parameter list of a method call statement).

# Expressions and Variables

An expression computes a value. Expressions in Dream yield either a primitive value (int or boolean) or a reference to an object.

### Identifier Expression

An expression that is an identifer (msg, x, in) must refer to a predefined identifier, local variable, method parameter, or an instance variable defined in the currrent class.

### Call Expression

A method call expression may be of the form *expr*.*method-name*(parms) or just *method-name*(parms).

- If *expr* is specified, then *method-name* must be the name of a method defined in the class of *expr*, or in an ancestor of that class
- If *expr* is not specified, then *method-name* must be the name of a method defined in the class being defined, or in an ancestor of this class

Methods called in a call expression must be non-void.

### Constructing Objects

Objects can be constructed in the following ways:

- Writing a literal value. Example:

  ```
  msg := "Hello, Suzie"
  ```

  A new String object is automatically created when "Hello Suzie" is evaluated; the value of the expression is a reference to the new String object.

- Using the

  `new` *type*

  expression. *type* must be a class name or an array type with a size expression. At runtime, an instance of that class is created, and instance variables are initialized to 0 / false / null. The value of the expression is a reference to the new object. Example:

  ```
  pt := new Point
  ```

### Null Values

The expression *null* denotes a reference to a nonexistent object. It is illegal to invoke a method on a null reference: for example, if s is null, s.length( ) results in a runtime error. In general, it is not possible to determine at compile time whether an expression is null, so the null check must occur at runtime.

Object variables hold either a reference to an object or the *null* reference. Uninitialized instance variables variables (except array variables) hold a reference to null.

It is an error to assign the null value to an int or boolean variable.

## Statements

### Assignment Statement

LHS := RHS

At runtime, the assignment statement evaluates the expression on the RHS to compute its value. It stores the resulting value in the variable on the LHS, after verifying that the type of the LHS is compatible with the type of the object on the RHS.

For B version implementations, the type of the LHS must exactly match the type of the RHS. For A version implementations, the type compatibility is more complicated to support polymorphism; see Inheritance Features below.

## User Defined Classes

### Attributes

Class-level variables must be initialized, depending on their type, to 0 (`int`), false (`boolean`), or null (`String` and user defined class types) when the class is instantiated.

### Methods

Methods may be defined with or without a return type. If no return type is defined, the method is a "void method" and may be called only in a call statement, not in an expression.

The compiler must generate code to initialize local variables to 0 (`int`), false (`boolean`), or null (`String` and user defined class types) when the method begins executing.

If a return type is defined, the code inside the method may assign a return value using an assignment statement, where the LHS is the name of the method:

```
getLength(): int
begin
  getLength := length
end
```

Note that this assignment may occur anywhere in the method, and the assignment does not cause the method to terminate.

Method overloading is not permitted: no two methods in the same class may have the same name.

Methods that call other methods in the same class may call only call methods defined **before** the current method (or the current method – via recursion).

### Object Variables

A variable whose type is a user-defined class holds a reference to an object. Example, where `point` is a user-defined class:

```
p: point
```

When an object variable is defined, it does not have a reference to any object, but contains null. *null* is a special reference that refers to no object at all. The *null* reference is type compatible with all object variables, so it can be assigned to them explicitly, as well as compared with them:

```
p := null
...
if p = null then ...
```

An object is instantiated using the *new* keyword, like this:

```
p := new point
```

The expression `new point` allocates memory for `point`'s instance variables from the heap, initializes them to 0 / null, and returns a pointer to that memory. Thus, the assignment statement stores the pointer in p.

There is no constructor syntax. However, the following is a common idiom in Dream:

```
p := (new point).init(5, 10)
```

Many classes define an init method that returns the object on which it is invoked, and can thus be used to do the work of a constructor.

### Member Visibility

All object attributes are private (accessible only to code in their class), and all methods are public.

### Method Calls

Every method call, even parameterless method calls, involves implicitly passing a pointer to the object as the first parameter to the method. A check should be done at runtime just before the call to see if the pointer is null. If the pointer is null, execution should abort with a "Null pointer exception on line ###" message displayed, where ### is the line number of the call.

# Strings

In addition to the Reader and Writer classes, the String class is a class defined in stdlib.Dream which contains the following methods:

- `length(): int` - returns the length of this string.

- `cat(str: String): String` - returns a copy of this string with *str* concatenated to the end of it. Neither this string nor *str* are modified.

  `catChar(char: int): String` - returns a copy of this string containing the ASCII character *char* concatenated to the end of this string. This string is not modified.

  `catInt(num: int): String` - returns a copy of this string with the base 10 representation of *num* concatenated to the end of it. This string is not modified.

- `charAt(index: int): int` - returns the ASCII character at position *index* in the string, or -1 if *index* is out of bounds

- `eq(s: String): boolean` - returns true if this string is equal to *s* using case-sensitive comparison
  `gt(s: String): boolean` - returns true if this string is lexicographically greater than *s*
  `gteq(s: String): boolean` - returns true if this string is lexicographically greater or equal to *s*

The `string` data type is an alias for the String class. Thus, when a variable of type `string` is defined, it has exactly the same semantics as if a variable of type String were defined.

Variables of type String (or its alias, `string`) hold a reference to a String object. Thus, before a value is assigned to a string variable, it holds a null reference, and it is illegal to call any of the string methods on it.

```
str: string

x := str.length() ~ causes null pointer exception
```

String literals evaluate to a string object reference. Thus, an assignment like this:

```
str := "Hello there!"
```

causes a reference to a string object containing the text "Hello there!" to be stored in *str*.

Since literal string values such as "Hello, Suzie" denote objects, methods can be called on them:

```
x := "Hello, Suzie".length( )
```

# Inheritance Features

### Oyd

The ancestor class of all Dream classes is named Oyd. It defines one method:

- `toString(): string` - returns "none". Intended to be overridden by descendants.

Classes defined without the *inherits from* clause automatically inherit methods from the Oyd class. Classes defined with the *inherits from* clause inherit attributes and methods in the indicated class, which must have been previously defined.

### Attributes

It is permissible for a class to define an attribute with the same name as an attribute in a parent.

Since all attributes are private, a child class is not allowed to access instance variables defined in the parent class. The parent class must define accessor methods for attributes to which the child needs access.

### Overriding

It is permissible for a child to define a method with the same name as a method in an ancestor. Furthermore, if the method has the same signature as a method in an ancestor, this is called overriding, and leads to polymorphic behavior at runtime. If the method does not have the same signature (including return type), no overriding occurs, and the parent's method is hidden.

## Type Checking

The Dream language is strongly typed. All the operations in the language, including array indexing, assignment, function calling, etc., expect arguments of specific types. If the types are not compatible, then there is a type error.

### Type Compatibility

An object variable of type A can reference an object of type B if A = B, or A is an ancestor of B. This is the notion of *type compatibility:* we say that a variable of type A is compatible with an object of type B if the types meet these rules.

For example, given the following type hierarchy:

Oyd > Person > American > Texan

Oyd > Yankee

A variable of type *Oyd* could hold a reference to an Oyd, a Person, an American, or a Texan. But a variable of type American could reference only an American or a Texan.

All variables except int and boolean variables are type compatible with the *null* value.

**Compile Time vs. Run Time Checking**

Some type errors can be caught at compile time; others cannot be caught until run time.

For example, the following error could be caught at compile time:

```
y: Yankee
a: American

a := y
```

Clearly, there is no object that *y* could refer to such that *a* would be type compatible with it. Now, consider this situation:

```
p: Person
a: American

p := a
```

This code is correct. *a* could refer either to an American or a Texan; since it is also legal for *p* to refer to an American or Texan, p is type compatible with a, and this can be verified at compile time. However, consider this situation:

```
p: Person
a: American

a := p
```

This code is fine in some cases and not in others. For example, *p* might refer to an American, in which case the assignment is legal. However, if *p* refers to a Person, the assignment is not legal. Since we don't know at compile time what object *p* will actually refer to, we must do some checking at runtime to verify type compatibility.

Thus, at compile time, the compiler can check only **compile-time type compatibility**, which is defined as follows:

- A variable of type A is compile-time compatible with an expression of type B if A = B, or A is an ancestor of B, or B is an ancestor of A.

The compile-time and run-time type compatibility checks must be performed not only on assignment, but also when performing method calls; the types of the actual parameters must be compatibile with the formals.

# Array Features

Note: Array features are not required for any level of Dream. They are listed here for completeness of the specification.

An array must be defined without a size:

```
list: int[]
```

Array variables (such as list) hold a reference to an array in memory. When first defined, the reference is null, until the array is instantiated like this:

```
list := new int[3]
```

Now *list* holds a pointer to a dynamically allocated memory area large enough to hold 3 integers.

**Array Assignment**

It is legal to assign one array to another:

```
list: int[]
list2: int[]

...
list := list2
```

This results in copying the reference in list2 to list, so both variables now reference the same array. (Or, if list2 was null, the null reference is copied to list).

**Bounds Checking**

All array accesses in Dream are checked at runtime to verify that the index is within bounds.