The KScr Language

Definition of the KScr Language

comroid

The KScr Language as defined by this document is a compiled high-level language whose goal is to provide runtime optimizations from a different approach.

Team:
Kaleidox

Contents

1 Source Model **3**
- 1.1 Modifiers . . . 3
  - 1.1.1 Accessibility Modifiers . . . 3
  - 1.1.2 Other Modifiers . . . 3
- 1.2 Classes . . . 3
  - 1.2.1 Class Types . . . 3
  - 1.2.2 Class Names . . . 4
  - 1.2.3 Type Generics . . . 4
  - 1.2.4 `extends` and `implements` inheritance setters . . . 4
- 1.3 Class Members . . . 4
  - 1.3.1 Static Initializer . . . 4
  - 1.3.2 Constructors . . . 5
  - 1.3.3 Methods . . . 5
  - 1.3.4 Properties . . . 5

2 The KScr VM **6**
- 2.1 Built-in Types . . . 6
  - 2.1.1 `interface void` . . . 6
  - 2.1.2 `class num<T>` . . . 6
  - 2.1.3 `class str` . . . 7
  - 2.1.4 `class object` . . . 7
  - 2.1.5 `class array<T>` . . . 7
  - 2.1.6 `class tuple<T...>` . . . 7
  - 2.1.7 `class enum<T>` . . . 7
  - 2.1.8 `class type<T>` . . . 7
  - 2.1.9 `class pipe<T...>` . . . 7
- 2.2 Literals . . . 7
  - 2.2.1 `null` . . . 7
  - 2.2.2 `Numeric Literals` . . . 7
  - 2.2.3 `String Literals` . . . 7
  - 2.2.4 `Array Literals` . . . 8
  - 2.2.5 `Tuple Literals` . . . 8
  - 2.2.6 `Other Literals` . . . 8

3 Core Features **9**
- 3.1 `Convert`-Class . . . 9
- 3.2 Property Caches . . . 9
- 3.3 Pipe Functionality . . . 9

4 KScr Build System **10**
- 4.1 Build Files . . . 10

1 Source Model

KScr expects source files to have the extension `.kscr`, and outputs binaries with the extension `.kbin`.
All source files must start with a package declaration:

```
package org.comroid.kscr;
```

Following the package declaration, an optional list of import statements follows; importing either canonical class paths or all static members of a class.

1.1 Modifiers

1.1.1 Accessibility Modifiers

Modifier	Function
`public`	Accessible from everywhere
`internal`	Accessible during compilation; compiled to `private`
`protected`	Accessible from inheritors
`private`	Accessible from inside only

1.1.2 Other Modifiers

Modifier	Function
`static`	Cannot be invoked dynamically
`final`	Cannot be overridden or changed
`abstract`	Must be implemented in inheritors
`synchronized`	Invocations are synchronized
`native`	Must be implemented by a native module

1.2 Classes

A class is declared by a class header of the following structure:

```
<accessibility modifiers> [class type] [class name]<type generics> <inherits> <body>
```

Note that the class body can be optional when using a semicolon.

1.2.1 Class Types

A source class can be of different types:

1.2.1.1 `class`-Type

A normal class that can be instantiated. Allowed modifiers are:

Modifier	Function
`static`	Cannot be instantiated and behaves like a Singleton
`final`	Cannot be inherited by other classes
`abstract`	Cannot be instantiated directly and must be inherited by other classes

1.2.1.2 `enum`-Type An enumeration of runtime-constants that follow a class-like pattern. This type does not allow modifiers.

1.2.1.3 `interface`-Type An interface that declares basic structure requirements for implementing classes. Cannot be instantiated directly.

1.2.1.4 `annotation`-Type A marker for most components of code. Used for flow control by enforcing rules at compile time, setting markers or carrying information at runtime.

1.2.2 Class Names

It is suggested that class names start with an upper case letter.

1.2.3 Type Generics

Type Generics are initially defined in a classes header, detailing the name with a postfix. They are surrounded by arrow-brackets `<...>`:

```
public class num<T> {}
```

There is two special kinds of Type Generics;

1.2.3.1 `n`-Generic The `n`-Generic serves as a type-based declaration for tuple types.
Its usages must be integers, and their value is available at runtime in a semi-static field `public final int n`.
Writing `string<2>` is the same as writing `tuple<2, string>`

1.2.3.2 Listing Generic A listing Generic serves as a varargs-Generic.
Its instance is an array of types which can be accessed at runtime in a semi-static field.

1.2.4 `extends` and `implements` inheritance setters

The `extends` and `implements` keywords follow the detailed class name definition and declare what classes or interfaces a class inherits.
Both the `extends` and the `implements` listing can contain multiple members.

1.3 Class Members

1.3.1 Static Initializer

The static initializer is declared by including a static member block in the class.
It is executed after compile-time (or read-time if reading binary) and can modify the members of the containing class.

```
static {
    // initialize class right before late initialization
```

```
}
```

The static initializer is compile to be a method with the header `private static final void cctor()`

1.3.2 Constructors

The constructor is used to create a dynamic instance of the class.
If a class `extends` more than one class, any of which does not support the default constructor, the constructor must declare all superconstructors like this:

```
public class Apple extends Fruit, Projectile implements Digestable {
    public Apple() : Fruit("Apple"), Projectile()
}
```

Constructors are compiled as Methods named `ctor`.

1.3.3 Methods

A method is a function of a class that can affect the class, compute a result or print "Hello, world!".
Methods are distinguished from properties by their parameter definition, which may be empty.
All methods must have an explicitly defined return type, but they may return `void`.

1.3.4 Properties

Properties are value computation access ports that can either hold a value, or compute it from a returning body.
If such return bodies contain another property, then the other property is checked for its last update time. If it has been updated before the calling property has been, then computation is skipped and the last returned value is returned again. Setting a property updates it, causing all dependent properties to be computed again on their next access.

2 The KScr VM

2.1 Built-in Types

2.1.1 `interface void`

The universal base type. Is implemented by all built-in types and can be implicitly cast to everything.

2.1.1.1 Method `void#toString(short alt)` Used to obtain a string representation of the object.
The `toString()` Method that is present in all objects has one optional parameter; a `short` that defines the alternative of the output string. Its default value is 0.
The following values must be returned by different alternatives:

Value	Predefined output string
0	A parseable representation of the object
1	A name that contains type information
2	An undetailed name of the object
3	A full undetailed name of the object
4	A detailed name of the object
5	A full detailed name of the object

2.1.1.2 Method `void#equals(void other)` Used to test two objects for equality. This method is called by the $=$ and $\neq$ operators.

2.1.1.3 Method `void#getType()` Used to obtain the exact class instance of an object.

2.1.1.4 Property `void#InternalID` Used to obtain the internal ID of the object.

2.1.2 `class num<T>`

The base type of all numerics. Contains numeric subtypes:

1. `byte` - Type-alias for `int<8>`
2. `short` - Type-alias for `int<16>`
3. `int<n = 32>`
4. `long` - Type-alias for `int<64>`
5. `float`
6. `double`

All subtypes can be used directly, or using the Type Generic T, for example: `int<24> == num<int<24»`

2.1.3 `class str`

The type of all strings.
A string represents an array of characters.

2.1.4 `class object`

The base type of all non-built-in objects.
Every foreign class that has no explicit `extends` definition implements this type implicitly.

2.1.5 `class array<T>`

The type of arrays. The type Generic `T` defines the type of the array.
Writing `array<str>` is the same as writing `str[]`.

2.1.6 `class tuple<T...>`

The type of tuples. The type Generic `T...` defines all types of held values in order.
If `T.Size == 1`, the tuple is of a singular type and might have been invoked by the n-Type-Generic; `T<n>`. Size must then be obtained by invoking `it.Size`.

2.1.7 `class enum<T>`

The type of enums. The type Generic `T` defines the output type of the enum.
A class `enum Codes` would be of type `enum<Codes>` or simply `Codes`.

2.1.8 `class type<T>`

The type of all types. The type Generic `T` defines the actual type that is defined by this type.

2.1.9 `class pipe<T...>`

The type of pipes. The type Generic `T` defines the type of data that is handled by this pipe.

2.2 Literals

2.2.1 `null`

The `null`-Literal. Is always of type `void`.

2.2.2 `Numeric Literals`

2.2.3 `String Literals`

A string literal is pre- and superceded by a double-quote `"` symbol. An escaped double-quote `\"` can be contained in the string.

2.2.3.1 Planned Behaviour: Interpolation A string supports interpolation using accolades with Formatter-support with the following syntax:

```
int hex = 1 << 3;
stdio <<- "hex: {hex:X}"
// prints "hex: 0x4"
```

2.2.4 `Array Literals`

Unimplemented.

2.2.5 `Tuple Literals`

Unimplemented.

2.2.6 `Other Literals`

2.2.6.1 `stdio` Constantly represents the program's standard IO stream.
The held value is of type `pipe<str>`

3 Core Features

3.1 `Convert`-Class

3.2 Property Caches

3.3 Pipe Functionality

4 KScr Build System

4.1 Build Files