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

KScr expects source files to have the extension `.kscr`, and outputs binaries with the extension `.kbin`.
Source files are not allowed in the source root directory. The binary root directory contains a string cache.

1.1 Package Declaration

At the beginning of every source file, KScr expects a package declaration:

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

1.2 Imports and Static Imports

Following the package declaration, a list of import statements is

1.3 Modifiers

1.3.1 Accessibility Modifiers

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

1.3.2 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

1.4 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.

1.4.1 Class Types

A source class can be of different types:

1.4.1.1 `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

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

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

1.4.1.4 `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.

1.4.2 Class Names

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

1.4.3 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;

1.4.3.1 `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>`
If the `n`-Generic is defined explicitly, then it is not used as a tuple alias.

1.4.3.2 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.

1.4.4 `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.

1.5 Class Members

1.5.1 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 compiled to be a method with the header `private static final void cctor()`

1.5.2 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`.

1.5.3 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`.

1.5.4 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 ≠ 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»`

2.1.2.1 `class int<n = 32>` The `int<n>` type defines an integer measured by `n` bytes. The default value of `n` is 32; as it is for common integers.

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 `interface pipe<T...>`

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

2.1.10 `class range`

The type of ranges. Ranges are invoked with a tilde: `start end`

2.2 Literals

2.2.1 `null`

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

2.2.2 Numeric Literals

A numeric literal may either be a decimal number, a hexadecimal string or a binary string.

Decimals In case it is an irrational decimal, it may be followed by a letter indicating the number type. Supported types are:

`0.0f`	`float`
`0.0d`	`double` (Default)

Hexadecimals A hexadecimal string must be preceded with `0x`. Output type will be an `int<n>` where `n` is chosen with a radix of 8.

```
int<8> x = 0x9;
```

Binaries A binary string must be preceded with `0b`. Output type will be an `int<n>` where `n` is chosen with a radix of 1.

```
int<5> x = 0b00101;
```

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 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>`.

2.2.6.2 `endl` Constantly represents the environment's standard Line feed style.
The held value is of type `str`.

2.3 Native Methods

KScr supports calls to native C# members. To use this, it is necessary to mark a method in KScr source code as `native`, and include a DLL that contains a method attributed with `[NativeImpl]`. At startup, the directory `include/` at the installation base path is recursively scanned for `*.dll` assemblies, which are then loaded and scanned for `[NativeImpl]` attributes.

2.3.0.1 Usage Example Considering the following KScr class:

```
package org.comroid.test;

public class NativeTest {
    public static native void callNativeMethod();
}
```

For successful execution, a C# class must be present at runtime with the following signature:

```
[NativeImpl(Package = "org.comroid.test", ClassName = "NativeTest")]
public class NativeImplementations
{
    [NativeImpl]
    public static IObjectRef callNativeMethod(RuntimeBase vm,
            Stack stack, IObject target, params IObject[] args)
    {
        return vm.ConstantNull;
    }
}
```

2.4 Implicit Conversions

All built-in types can be implicitly converted to and from each other. For example, trying to assign a `str` to a variable of type `int` will first call `int.parse()` on the string.

2.5 Implementing Operators

Implementing an operator is as simple as adding a method named `op<Operator>` to a class, where for binary operators, the first and only parameter denotes the right operand.
For example, implementing a division operator for strings would be as simple as adding the following method:

```
public str opDivide(int right)
```

2.6 Property Caches

Any property contains a value that knows when the property was last updated. If a property has a computed getter, then the getter will only be evaluated if any property used inside the getter has been updated later than the computed property.
After computation, the result is stored in an internal cache and will be returned if all used properties have not been updated after the computed property.

2.7 Lambdas

Every interface that has exactly one abstract method is a functional interface.
Functional interfaces can be invoked as a lambda instead of an object instance.

2.7.1 Anonymous Lambdas

An anonymous lambda consists of a parameter denotation in parentheses, followed by a normal arrow and either an expression or a statement body. An example for an anonymous lambda that implements an integer addition `Func<int, int, int>` is:

```
(x, y) -> x + y;

// or, using a statement body:
(x, y) -> {
    return x + y;
}
```

A lambda without parameters is denoted by empty parentheses.

2.7.2 Method Reference Lambdas

A method reference lambda may be used if the functional interface notation and the referenced method notation match. If the referenced method is non-static, then the first parameter must be assignable to the invoking target object.

```
// static method reference:
Func<int, str> parseInt = int.parse;

// dynamic method reference:
Func<str> toString = myObject.toString;
```

2.8 Pipe Functionality

KScr supports both an active and passive object pipeline through the `pipe<T>` interface. These can be used to either handle and parse incoming & outgoing data, or to read and write data finitely to or from an accessor.

2.8.1 Passive Pipes

Simple examples for passive Pipes are the stdio Keyword and all properties.
Passive Pipes can only be read and written to, by using double-arrow operators.
Additionally, an active pipe operator can be attached to listen for value changes.

```
// write text to stdout:
stdio <<- "text";

// read line from stdin:
stdio ->> str line;

// call velocityChangeListener() every time 'velocity' is changed:
velocity >>= velocityChangeListener;
// or create an active pipe that serves as an event hub:
velocityHandler = velocity =>> velocityChangeListener;
```

2.8.2 Active Pipes

An active pipe actively uses callbacks to push values downstream. They can be stored in a variable, and cascaded for advanced event processing.
For example, parsing TCP data from a Websocket to JSON may look like this:

```
// parse websocket packets from TCP data
websocketData = tcpData =>> parseWebsocketPacket;
// parse json from websocket packet body
jsonData = websocketData =>> (data -> data.body) =>> parseJson;
```

3 KScr Build System

3.1 Build Files