Theia Virtual Machine

Kirill Z., George S.

March 2023

Table of contents

Introduction

The Theia Virtual Machine (TVM) is a versatile tool that offers just-in-time (JIT) compilation capabilities for a range of computing applications. It is designed to be equally effective for both blockchain and desktop environments, making it an ideal solution for developers who need a powerful and flexible virtual machine. With TVM, developers can create high-performance applications that can be executed quickly and efficiently, regardless of the platform or technology stack they are working with. This makes TVM a valuable asset for anyone who is looking to build robust and scalable software solutions that can keep up with the demands of modern computing environments. TVM was inspired by JVM and inside TVM everything represented as object too.

Memory

Heap introduction

The TVM's heap is designed to be highly optimized for the specific needs of the virtual machine, providing fast and efficient memory allocation and management for a wide range of computing tasks. The heap can allocate and manage memory for a variety of different object types, including primitive types like integers and doubles, as well as complex data structures like arrays and other objects. For reasons of perfomance was chosen pairing heap.

Operation	find-min	delete-min	insert	decrease-key	meld
Pairing heap	$\Theta(1)$	$O(\log n)$[1]	$\Theta(1)$	$o(\log n)$[2]	$\Theta(1)$

[1] Amortized time. Fredman, Michael L.; Sedgewick, Robert; Sleator, Daniel D.; Tarjan, Robert E. (1986) "The pairing heap: a new form of self-adjusting heap" (PDF)

[2] Amortized time. Fredman, Michael L.; Sedgewick, Robert; Sleator, Daniel D.; Tarjan, Robert E. (1986) "The pairing heap: a new form of self-adjusting heap" (PDF)

Table 1. Time complexity[3] of pairing heap.

Memory Structure

It's necessary to say a little bit about memory inside heap.

Memory is devided on the two huge areas of memory:

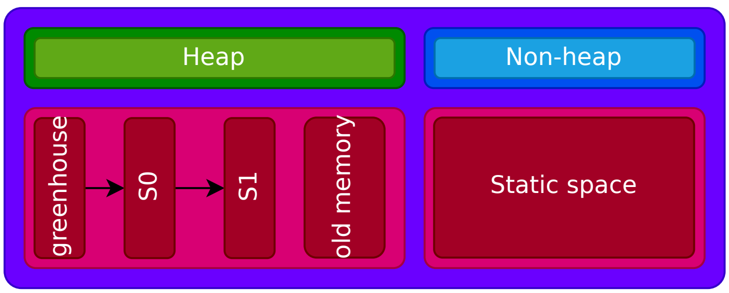
- Young generation.
- Old generation.

Young generation separated on three area:

- greenhouse - it's an area where every object is growing.
- survivors 0 - area where objects are stored after first cycle of collecting garbage.
- survivors 1 - area where objects are stored after second cycle of collecting garbage.
- old memory - area where stored objects after three and more cycles of collecting garbage.

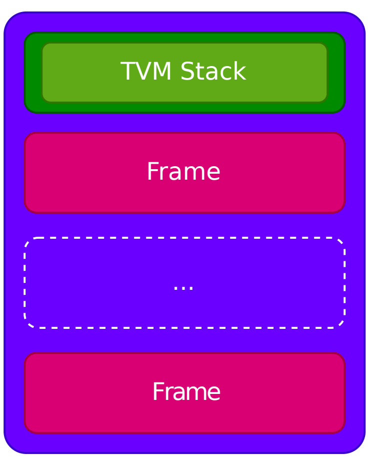
Old generation separated on:

- Static space - area where stored static data(constants etc.).



Allocator introduction

TVM runtime memory representation TVM is a hybrid virtual machine that combines both stack-based and heap-based architectures. While the TVM's stack-based architecture is used for pushing and popping frames, which is used for executing bytecode and manipulating data at runtime, the TVM's heap-based architecture is used for allocating and managing memory.



The stack is a region of memory that is organized as a Last-In-First-Out (LIFO) data structure, which means that the most recently pushed value is always at the top of the stack.

When a thread executes a method, the TVM tracks the current method, current frame, current class, and current constant pool. The current frame stores parameters, local variables, intermediate computations, and other data related to the

[3] WIkipedia - Big O notation

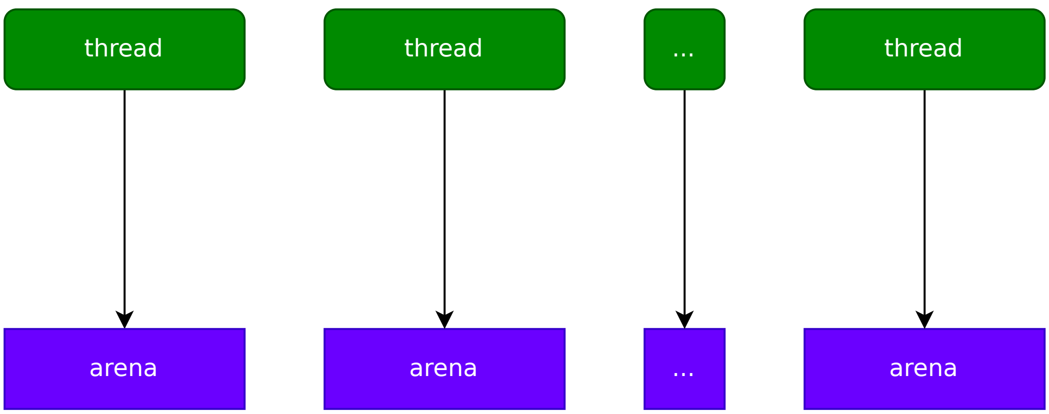
method's execution. As the thread invokes a new method, the TVM creates and pushes a new frame onto the stack, which becomes the current frame for the executing method.

The TVM's allocator uses a unique approach called ”arenas”, which helps to improve its performance and scalability. With this approach, each CPU thread is assigned its own arena, which is a separate region of memory that contains its own dedicated heap. This means that each thread can allocate and manage memory independently, without needing to synchronize with other threads or compete for resources.

Each arena contains its own heap, which is managed by a specialized allocator that is optimized for fast allocation and deallocation of small objects. The allocator uses a variety of techniques, such as slab allocation and object caching, to minimize overhead and reduce fragmentation of the heap.

By using arenas, the TVM is able to achieve high levels of concurrency and scalability, even on multi-core systems. Each CPU thread can operate independently, which allows the TVM to fully utilize all available processing power and avoid contention for shared resources. This makes the TVM a highly efficient and performant virtual machine that is well-suited for a wide range of computing applications.

Overall, the use of arenas is a key feature of the TVM's architecture, and it helps to make the virtual machine a powerful and flexible tool for developers who need to build high-performance software solutions.



Stack Frame The stack frame is divided into two components: local variables, operand stack. The sizes of the local variables and operand stack, measured in words, vary based on the specific needs of each method. These sizes are predetermined during compilation and are included in the class file data for each method. The size of the frame data, however, is dependent on the implementation.

When a method is invoked by the virtual machine, it examines the class data to determine the number of words required by the method in the local variables and operand stack. The virtual machine then creates a stack frame of the appropriate size for the method and adds it to the Theia stack.

Local variables Local variables is a zero-based array. In TVM, a local variable can store a value of various types, including **boolean**, **byte**, **char**, **short**, **int**, **reference**, or **returnAddress**. On the other hand, if the value being stored is of type **long** or **double**, then two local variables are required to hold it.

```
class Example {
  public:
  int example(int a, String b, long c, Object d) {
    return 0;
  }
}
```

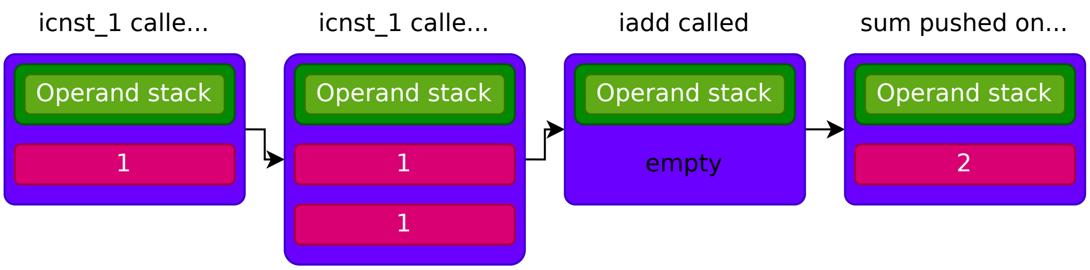
In this example local variables will be set like this:

Index	Type	Parameter
0	int	int a
1	reference	String b
2	long	long c

Index	Type	Parameter
3	reference	Object d

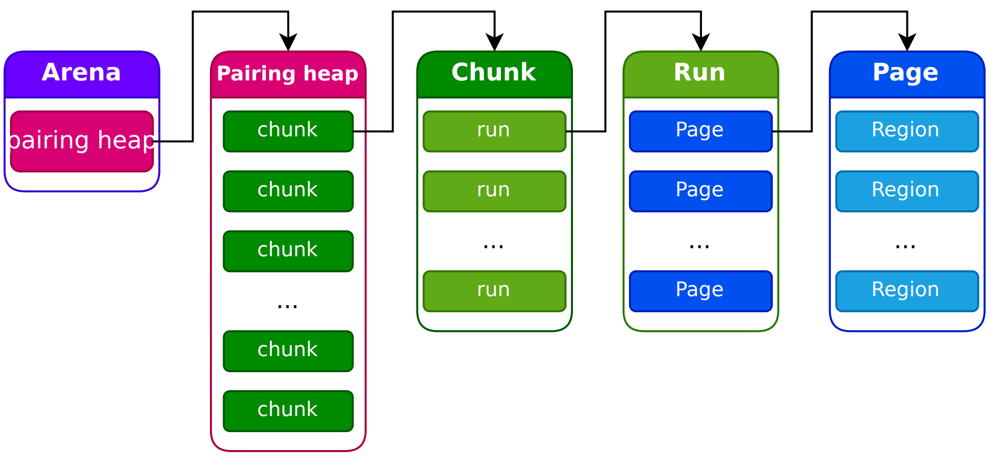
Operand Stack Each stack frame contains a last-in-first-out (LIFO) stack. When function is called - empty operand stack is created.

Operand stack used for storing data, for example, when ***icnst_1*** called, the integer value ”1” push onto the stack, if we add more one ”1” onto the top of the stack and call ***iadd*** both ”1” will be popped and then their sum will be pushed back onto the stack.



Allocator structure

Pairing heap in allocator's arena contains chunks. Chunks is a fixed structure which consists of `run`-list, each run contains page[4] with regions. Local variables are accessed using an index-based system. An integer is treated as an index into the local variable array only if it falls between zero and the size of the array minus one.

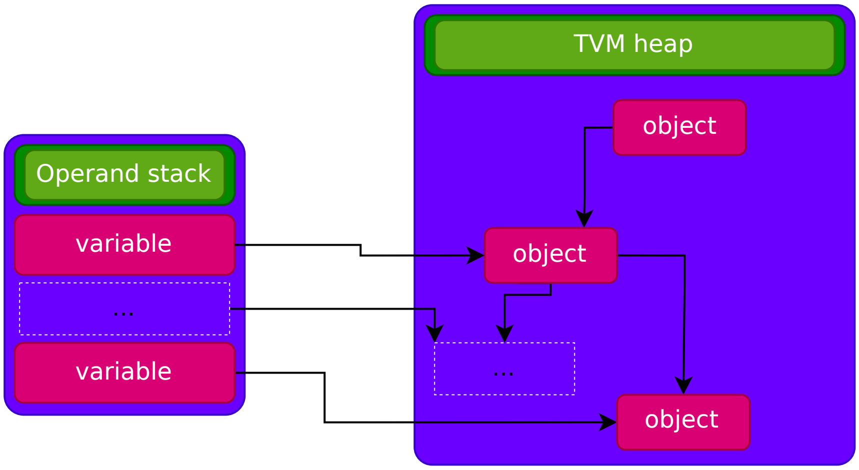


Arena also contains auxiliary data structures such as Red-Black tree[5] and R-tree[6]. Red-black tree used for searching small-size objects, while R-tree used for searching huge objects.

Allocation

TVM has opcodes which starts with ***new*** word, for example opcode ***newarray***. They're used for allocating object in heap. When object is created, it pushed onto the stack frame.

[4]Wikipedia - Page (computer memory)
[5]Wikipedia - Red–black tree
[6]Wikipedia - R-tree



When program asking for allocating memory, allocator looking for requested memory size in auxiliary data-structures depending on the requested size: Red-black tree[7] or R[8]-tree - if there is no free memory in heap, new memory will be asked through system call(e.g **mmap** or **VirtualAlloc**). However, if there is no available memory(both RAM and memory swap are full), then program throttling will be turned on, till necessary size of memory become available.

All allocations/deallocations are tracked by memory tracker, it allows to solve memory leaking.

Garbage collector

Garbage collector is used for managing memory, collecting garbage or garbagging - is a process when unused objects will be destroyed. Basically it works on counting references to the data. As was mentioned above - there are three global areas for objects in heap: greenhouse, S0, S1, old memory. When objects are only allocated - it's placed into greenhouse till first time of collecting garbage, after collecting garbage and if object is still alive - object moves to S0 space and so on, till collecting garbage in S1, if object can be destroyed - it moves into old memory and possibly won't be deleted over the lifetime of the program.

By default in OpenTVM object destroying itself when quantity of reference to object is 0. At the end of the program's lifetime, garbagging is started to check reference cycles.

Static space

Static space - is a hash map which represents global variables[9]. Static space are storing data which will never be deleted and can be accesed by all threads, for example, it can be burning address for units in blockchain, quantity of cpu threads or path where program were executed[10].

Metaspace

Metaspace is a memory area where all required data for program stored, for example, **classes**, **functions**, **variables**, **access flags** and other data.

Object

In spark implementation of TVM, there is no data structure which represents object itself.

Data types

Data type	Note
bool	true/false

[7] Wikipedia - Red–black tree
[8] Wikipedia - R-tree
[9] Can't be set manually by user.
[10] Not available for all threads in blockchain.

Data type	Note
byte	signed type from -0x7F to 0x7F, inclusive
short	signed type from -0x7FFF to 0x7FFF, inclusive
int	signed type from -0x7FFFFFFF to 0x7FFFFFFF, inclusive
long	signed type from -0x7FFFFFFFFFFFFFFF to 0x7FFFFFFFFFFFFFFF, inclusive
uint32	unsigned type from 0 to 0xFFFFFFFF
uint64	unsigned type from 0 to 0xFFFFFFFFFFFFFFFF
uint128	unsigned type from 0 to 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
uint256	unsigned type from 0 to 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
double	in range: 1.7E +/- 308
char	`'\u0000'` to `'\uffff'` inclusive or else from 0 to 65535
address	address of object

Opcodes

TVM has necessary opcodes for both blockchain and off-chain developing. For blockchain version of TVM there're few opcodes that are used inside it.

hex num	name	args	examples	Stack [before]→[after]	Can be used off-chain
000	stop	-	stop	* → [empty]	+
001	go_to	int	go_to 2	[no change]	+
002	swap	int,int	swap 1 3	value1,value2→value2,value1	+
003	iadd	num, num		value1,value2→result	+
004	isub	num, num		value1,value2→result	+
005	idivide	num, num		value1,value2→result	+
006	imul	num, num		value1,value2→result	+
007	imod	int, int		value1,value2→result	+
008	ixor	int, int		value1,value2→result	+
~~009~~	-	-	-	-	-
010	ilshift	int, int		value1,value2→result	+
011	irshift	int		value1,value2→result	+
012	pop	object		value→	+
013	pop2	object, object	-	value1,value2→	+
014	dup	object		value→value1,value1	+
015	ior	num		value1,value2→result	+
016	iand	int		value1,value2→result	+
017	pshnull	-	-	→null	+
018	sha3	object		value→sha3(value)	-
019	balance	string		→value	-
020	timestamp	-		→value	-
021	blockhash	-		→value	-
022	chainid	-		→value	-
023	create	-		[no change]	-
024	destruct	string		[no change]	-
025	waddress	-		→value	-
026	invalid	-	-	-	+
027	icnst_0	int	-	→ 0	+
028	icnst_1	int	-	→ 1	+
029	icnst_2	int	-	→ 2	+
030	icnst_3	int	-	→ 3	+
031	icnst_4	int	-	→ 4	+
032	u64cnst_0	-	-	→ 0	+
033	u64cnst_1	-	-	→ 1	+
034	checktype	int	checktype	value→value	+
035	u32cnst_0	-	-	→ 0	+

hex num	name	args	examples	Stack [before]→[after]	Can be used off-chain
036	u32cnst__1	-	-	→ 1	+
037	u32str__0	-	-	value→	+
038	u32str__1	-	-	value→	+
039	u32str__2	-	-	value→	+
040	u32str__3	-	-	value→	+
041	u64str__0	-	-	value→	+
042	u64str__1	-	-	value→	+
043	u64str__2	-	-	value→	+
044	u64str__3	-	-	value→	+
045	astorec	-	-	arrayref, index, value →	+
046	aloadc	-	-	-	+
047	u128str__0	int		value→	+
048	u128str__1	int		value→	+
049	u128str__2	int		value→	+
050	u128str__3	int		value→	+
051	ldc	int index		→ value	+
052	ild__0	-	iload 1	→ value	+
053	ild__1	-	iload 1	→ value	+
054	ild__2	-	iload 1	→ value	+
055	ild__3	-	iload 1	→ value	+
056	swap	-	swap	value1,value2→value2,value1	+
057	if__acmpeq	object, object	if__acmpeq 23	[no change]	+
058	if__acmpne	object , object	if__acmpne 23	[no change]	+
059	if__icmpeq	object, object	if__icmpeq 23	[no change]	+
060	if__icmpge	num,num	if__icmpge 23	[no change]	+
061	if__icmpgt	num,num	if__icmpgt 23	[no change]	+
062	if__icmple	num,num	if__icmple 23	[no change]	+
063	if__icmplt	num,num	if__icmplt 23	[no change]	+
064	if__icmpne	num,num	if__icmpne 23	[no change]	+
065	ifeq	num	ifeq 23	[no change]	+
066	ifge	num	ifge 23	[no change]	+
067	ifgt	num	ifgt 23	[no change]	+
068	ifle	num	ifle 23	[no change]	+
069	iflt	num	iflt 23	[no change]	+
070	ifne	num	ifne 23	[no change]	+
071	ifnonnull	Object	ifnonnull 23	[no change]	+
072	ifnull	Object	ifnull 23	[no change]	+
073	nop	-	nop	[no change]	+
~~074~~	-	-	-	-	+
075	dcnst__0	-	dcnst__0	→ 0.0	+
076	dcnst__1	-	dcnst__1	→ 1.0	+
077	lcnst__0	-	lcnst__0	→ 0	+
078	lcnst__1	-	lcnst__0	→ 1	+
~~079~~	-	-	-	-	+
080	dadd	num, num		value1,value2→value1+value2	+
081	u128cnst__0	-	-	→ 0	+
082	u128cnst__1	-	-	→ 1	+
083	u256cnst__0	-	-	→ 0	+
084	u256cnst__1	-	-	→ 1	+
083	dsub	num, num		value1,value2→value1-value2	+
084	lsub	long,long	-	value1,value2→value1-value2	+
085	lmul	long,long	-	value1,value2→value1*value2	+
086	ldiv	long,long	-	value1,value2→value1/value2	+
087	u256str__0	int		value→	+
088	u256str__1	int		value→	+

hex num	name	args	examples	Stack [before]→[after]	Can be used off-chain
089	u256str__2	int		value→	+
090	u256str__3	int		value→	+
091	rtcall	objectref	-	objectref,[arg1,arg2,...]→result	+
092	stcall	objectref	-	objectref,[arg1,arg2,...]→result	+
093	itfcall	objectref	-	objectref,[arg1,arg2,...]→result	+
094	spcall	objectref	-	objectref,[arg1,arg2,...]→result	+
095	lld__0	-	-	→ value	+
096	lld__1	-	-	→ value	+
097	lld__2	-	-	→ value	+
098	lld__3	-	-	→ value	+
099	aloadi	-	-	-	+
100	astorei	-	-	-	+
~~101~~	-	-	-	-	+
102	dinv	num		value→!value	+
103	ddiv	num, num		value1,value2→value1/value2	+
104	dmul	num, num		value1,value2→value1*value2	+
105	aloadl	-	-	-	+
106	astorel	-	-	-	+
107	aloadd	-	-	-	+
108	astored	-	-	-	+
109	aloadb	-	-	-	+
110	astoreb	-	-	-	+
111	-	-	-	-	+
112	iinc	int, num	iinc 1 4	[No change]	+
113	i2d	num,num	i2d	(int)value→(double)result	+
114	i2u64	num,num	i2u64	(int)value→(uint64)result	+
115	i2u128	num,num	i2u128	(int)value→(uint128)result	+
116	i2u256	num,num	i2u256	(int)value→(uint256)result	+
117	i2b	int	i2b	(int)value→(byte)result	+
118	i2c	int	i2c	(int)value→(char)result	+
119	i2l	int	i2l	(int)value→(long)result	+
120	i2s	int	i2s	(int)value→(short)result	+
121	d2i	double	d2i	(double)value→(int)result	+
122	d2l	double	d2l	(double)value→(long)result	+
123	ireturn	int	ireturn	value→[empty]	+
124	lreturn	long	lreturn	value→[empty]	+
125	return	-	return	-	+
126	areturn	objectref	refreturn	objectRef → [empty]	+
127	new	int	new 115	→ objectRef	+
128	newarray	uint64__t	newarray	(array__size)value→arrayref	+
129	new__mdarray	byte1,byte2,dimensions	-	count1, [count2,...] → arrayref	+
130	dreturn	-	-	value→[empty]	+
131	u32return	-	-	value→[empty]	+
132	u64return	-	-	value→[empty]	+
133	u128return	-	-	value→[empty]	+
134	u256return	-	-	value→[empty]	+
135	aloadu32	-	-	-	+
136	astoreu32	-	-	-	+
137	aloadu64	-	-	-	+
138	astoreu64	-	-	-	+
139	aloadu128	-	-	-	+
140	astoreu128	-	-	-	+
141	aloadu256	-	-	-	+
142	astoreu256	-	-	-	+
143	aloada	-	-	-	+

hex num	name	args	examples	Stack [before]→[after]	Can be used off-chain
144	astorea	-	-	-	+
145	aconst_null	-	-	→null	+
146	setfield	-	-	objectref, value→	+
147	setstatic	-	-	value→	+

hex num	name	description
000	stop	stop execution of the program
001	go_to	goes to another instruction at *branchoffset*
002	swap	swaps two references. indexing starts from top of stack
003	iadd	adding value
004	isub	subtract value
005	idivide	devide value
006	imul	multiply value
007	imod	a % b
008	ixor	a ^ b
~~009~~	-	-
010	ilshift	a « val
011	irshift	a » val
012	pop	pop value
013	pop2	pop 2 values on top of the stack
014	dup	duplicate from top of the stack
015	ior	a || b
016	iand	a & b
017	pshnull	push null reference on top of the stack
018	sha3	apply sha3_256 value on top of the stack.
019	balance	get balance of address
020	timestamp	get timestamps
021	blockhash	get blockhash
022	chainid	returns chain_id
023	create	create contract
024	destruct	destruct contract and returns all holdings to their holders
025	waddress	wallet address of current contract
026	invalid	invalid
027	icnst_0	push int value 0 onto the stack.
028	icnst_1	push int value 1 onto the stack
029	icnst_2	push int value 2 onto the stack
030	icnst_3	push int value 3 onto the stack
031	icnst_4	push int value 4 onto the stack
032	u64cnst_0	push uint64_t value 0 onto the stack
033	u64cnst_1	push uint64_t value 1 onto the stack
034	checktype	check if objectref's type equals type in the classes virtual pool by index of class in pool
035	u32cnst_0	push uint32_t value 0 onto the stack
036	u32cnst_1	push uint32_t value 1 onto the stack
037	u32str_0	push uint32_t into register 0
038	u32str_1	push uint32_t into register 1
039	u32str_2	push uint32_t into register 2
040	u32str_3	push uint32_t into register 3
041	u64str_0	push uint64_t into register 0
042	u64str_1	push uint64_t into register 1
043	u64str_2	push uint64_t into register 2
044	u64str_3	push uint64_t into register 3
045	astorec	store char into array
046	aloadc	load char from array

hex num	name	description
047	u128str_0	push uint128_t value into local 0
048	u128str_1	push uint128_t value into local 1
049	u128str_2	push uint128_t value into local 2
050	u128str_3	push uint128_t value into local 3
051	ldc	push constant with #index from a constant pool
052	ild_0	load integer from local_0 on top of stack
053	ild_1	load integer from local_0 on top of stack
054	ild_2	load integer from local_0 on top of stack
055	ild_3	load integer from local_0 on top of stack
056	swap	swaps two top elements
057	if_acmpeq	if references are equal jump to the next instruction
058	if_acmpne	if references are not equal jump to the next instruction
059	if_icmpeq	if ints are equal jump to the next instruction
060	if_icmpge	if *value1* is greater than or equal to *value2*jump to the next instruction
061	if_icmpgt	if *value1* is greater than *value2*jump to the next instruction
062	if_icmple	if *value1* is less than or equal to *value2*jump to the next instruction
063	if_icmplt	if *value1* is less than *value2*jump to the next instruction
064	if_icmpne	if ints are not equaljump to the next instruction
065	ifeq	if *value* is 0 jump to the next instruction
066	ifge	if *value* is greater than or equal to 0jump to the next instruction
067	ifgt	if *value* is greater than 0jump to the next instruction
068	ifle	if *value* is less than or equal to 0jump to the next instruction
069	iflt	if *value* is less than 0jump to the next instruction
070	ifne	if *value* is not 0 jump to the next instruction
071	ifnonnull	if *value* is not nulljump to the next instruction
072	ifnull	if *value* is nulljump to the next instruction
073	nop	perform no operation
~~074~~	-	-
075	dcnst_0	push double value 0.0 onto the stack.
076	dcnst_1	push double value 1.0 onto the stack.
077	lcnst_0	push long value 0 onto the stack.
078	lcnst_1	push long value 1 onto the stack.
~~079~~	-	-
080	dadd	adding value
081	u128cnst_0	push uint128_t value 0 onto the stack.
082	u128cnst_1	push uint128_t value 1 onto the stack.
083	u256cnst_0	push uint256_t value 0 onto the stack.
084	u256cnst_1	push uint256_t value 1 onto the stack.
083	dsub	subtract value
084	lsub	subtract long
085	lmul	multiply long
086	ldiv	divide long
087	u256str_0	push uint256_t value into local 0
088	u256str_1	push uint256_t value into local 1
089	u256str_2	push uint256_t value into local 2
090	u256str_3	push uint256_t value into local 3
091	rtcall	calling method by objectref
092	stcall	calling static method by objectref
093	itfcall	calling interface method by objectref
094	spcall	call method and push result onto the stack
095	lld_0	load long from local_0 on top of stack
096	lld_1	load long from local_1 on top of stack
097	lld_2	load long from local_2 on top of stack
098	lld_3	load long from local_3 on top of stack
099	aloadi	load integer from array

hex num	name	description
100	astorei	store integer into array
~~101~~	-	-
102	dinv	!a
103	ddiv	devide value
104	dmul	multiply value
105	aloadl	load long from array
106	astorel	store long into array
107	aloadd	load double from array
108	astored	store long into array
109	aloadb	load bool from array
110	astoreb	store bool into array
111	-	-
112	iinc	increment local variable by #index
113	i2d	convert int to double (two top values from stack)
114	i2u64	convert int to uint64 (two top values from stack)
115	i2u128	convert int to uint128 (two top values from stack)
116	i2u256	convert int to uint256 (two top values from stack)
117	i2b	convert int to byte
118	i2c	convert int to char
119	i2l	convert int to long
120	i2s	convert int to char
121	d2i	convert double to int
122	d2l	convert double to long
123	ireturn	return an integer from a method
124	lreturn	return an long from a method
125	return	return from void method
126	areturn	return reference from a method
127	new	loading object with type by index in constant pool
128	newarray	create a new array with size at the top of the stack
129	new_mdarray	create a new multidimensional array of type by classref in pool by index (indexbyte1 « 8 || indexbyte2)
130	dreturn	return a double from method
131	u32return	return a uint32 from method
132	u64return	return a uint64 from method
133	u128return	return a uint128 from method
134	u256return	return a uint256 from method
135	aloadu32	load uint32 from array
136	astoreu32	store uint32 into array
137	aloadu64	load uint64 from array
138	astoreu64	store uint64 into array
139	aloadu128	load uint128 from array
140	astoreu128	store uint128 into array
141	aloadu256	load uint256 from array
142	astoreu256	store uint256 into array
143	aloada	load reference from array
144	astorea	store reference into array
145	aconst_null	store null reference onto the stack
146	setfield	set value to object by reference
147	setstatic	set to static object

Compiled Files Structure

Classes

```
Class {
    u4              classBytes;
```

Compiled Files Structure

Classes

```
Class {
    u4              classBytes;
    u2              minorBytes;
    u2              majorBytes;
    u2              constantPoolCount;
    cpInfo          constantPool[constantPoolCount-1];
    u2              accessFlags;
  u2              thisClass;
  u2              superClass;
  u2              fieldsCount;
  fieldInfo       fields[fieldsCount];
  u2              methodsCount;
  methodInfo      methods[methodsCount];
  u2              attributesCount;
  attributeInfo   attributes[attributesCount];
  u2              interfacesCount;
  u2              interfaces[interfacesCount];
}
```

Functions

```
Function {
    u2              accessFlags;
    u2              classDescripto; // maybe null if outside class
    u2              descriptorIndex;
    u2              attributesCount;
  attributeInfo   attributes[attributesCount];
}
```