```
//bytecode.py
BYTECODE = {}


def bytecode(code):
    def cl(func):
        BYTECODE[hex(code)] = func
        return func

    return cl


def get_operation(code):
    return BYTECODE.get(code)


def get_operation_name(code):
    if code in BYTECODE:
        return BYTECODE[code].__name__
    return ""

//checkcast.py
"""Major definitions for classes and instances"""

from pyjvm.prim import PRIMITIVES
```

```
def checkcast(s, t, vm):
    '''Check if a is instance of b
    both are classes from vm.get_class(...) - JavaClass
    '''
    if s.is_array:
        if t.is_array:
            s_name = s.this_name
            assert s_name[0] == '['
            t_name = t.this_name
            assert t_name[0] == '['
            s_name = s_name[1:]
            t_name = t_name[1:]
            if s_name[0] == 'L':
                s_name = s_name[1:-1]
            else:
                s_name = PRIMITIVES[s_name]
            if t_name[0] == 'L':
                t_name = t_name[1:-1]
            else:
                t_name = PRIMITIVES[t_name]
            sc = vm.get_class(s_name)
            tc = vm.get_class(t_name)
            if sc.is_primitive and tc.is_primitive:
                if sc == tc:
                    return True
                else:
                    return False
            return checkcast(sc, tc, vm)
        elif t.is_interface:
            for i in s.interfaces:
                if i == t.this_name:
                    return True
            return False
        else:
            if t.this_name == 'java/lang/Object':
                return True
```

```
        else:
            return False
    if s.is_interface:
        if t.is_interface:
            while s is not None:
                if t == s:
                    return True
                s = s.super_class
            return False
        if t.this_name == 'java/lang/Object':
            return True
        else:
            return False
    # S is object class
    if t.is_interface:
        while s is not None:
            for i in s.interfaces:
                i_c = vm.get_class(i)
                while i_c is not None:
                    if t == i_c:
                        return True
                    assert len(i_c.interfaces) < 2
                    if len(i_c.interfaces) == 1:
                        i_c = vm.get_class(i_c.interfaces[0])
                    else:
                        i_c = None
            s = s.super_class
        return False
    while s is not None:
        if t == s:
            return True
        s = s.super_class
    return False
```

//class_loader.py

```
'''Class loader. Binary to python representation'''

import logging
```

```
import os
import struct
import zipfile

from pyjvm.jvmo import JavaClass

logger = logging.getLogger(__name__)


def class_loader(class_name, (lookup_paths, jars, rt)):
    '''Get JavaClass from class file.
    Order of lookup: rt.jar, other jars from class path, folders
    from class path
    '''
    logger.debug("Loading class {0}".format(class_name))
    assert class_name[0] != '['  # no arrays
    file_path = class_name + ".class"
    f = None
    zip_file = None
    if file_path in rt:
        path = rt[file_path]
        zip_file = zipfile.ZipFile(path, "r")
        f = zip_file.open(file_path)
        logger.debug("Loading %s from %s", file_path, path)
    elif file_path in jars:
        path = jars[file_path]
        zip_file = zipfile.ZipFile(path, "r")
        f = zip_file.open(file_path)
        logger.debug("Loading %s from %s", file_path, path)
    else:
        for directory in lookup_paths:
            path = os.path.join(directory, file_path)
            if os.path.exists(path) and not os.path.isdir(path):
                f = open(path, "rb")
                break
            logger.debug("Loading from file %s", path)
    if f is None:
        raise Exception("Class not found " + class_name)
```

```
    # file discovered, read step by step
    try:
        cafebabe(f)
        jdk7(f)
        constant_pool = read_constant_pool(f)
        class_flags = access_flags(f)
        (this_name, super_name) = this_super(f)
        all_interfaces = interfaces(f)
        all_fields = fields(f)
        all_methods = methods(f)
    except Exception:
        raise
    finally:
        if zip_file is not None:
            zip_file.close()
        f.close()
    return make_class(this_name, super_name, constant_pool, all_fields,
                      all_methods, all_interfaces, class_flags)


# EXACTLY THE SAME RESULTS COME FROM PYTHON's struct MODULE
# Here both approaches are used to make real reading process cleaner

def getU1(f):
    '''Single byte'''
    byte1 = f.read(1)
    return ord(byte1)


def getU2(f):
    '''Two bytes'''
    byte1 = f.read(1)
    byte2 = f.read(1)
    return (ord(byte1) << 8) + ord(byte2)
```

```
def getU4(f):
    '''4 bytes'''
    byte1 = f.read(1)
    byte2 = f.read(1)
    byte3 = f.read(1)
    byte4 = f.read(1)
    return (ord(byte1) << 24) + (ord(byte2) << 16) + (ord(byte3) << 8) \
        + ord(byte4)


def getUV(f, length):
    '''variable length'''
    data = f.read(length)
    return data


def cafebabe(f):
    '''Make sure this is java'''
    cb = [0xCA, 0xFE, 0xBA, 0xBE]
    index = 0
    while index < 4:
        byte = getU1(f)
        if byte != cb[index]:
            raise Exception("No CAFEBABE")
        index += 1


def jdk7(f):
    '''Make sure this is java 7 class'''
    getU2(f)
    major = getU2(f)
    if major != 0x33:  # 52 - jdk7
        raise Exception("Not a jdk7 class")


def read_constant_pool(f):
    '''Constant pools starts with index 1'''
    pool = ["ZERO"]
```

```
    cp_size = getU2(f)
    count = 1
    while count < cp_size:
        cp_type = getU1(f)
        if cp_type == 10:  # CONSTANT_Methodref
            pool.append([10, getU2(f), getU2(f)])
        elif cp_type == 11:  # CONSTANT_InterfaceMethodref
            pool.append([11, getU2(f), getU2(f)])
        elif cp_type == 9:  # CONSTANT_Fieldref
            pool.append([9, getU2(f), getU2(f)])
        elif cp_type == 8:  # CONSTANT_String
            pool.append([8, getU2(f)])
        elif cp_type == 7:  # CONSTANT_Class
            pool.append([7, getU2(f)])
        elif cp_type == 6:  # CONSTANT_Double
            value = struct.unpack('>d', f.read(8))[0]
            pool.append([6, value])
            count += 1  # double space in cp
            pool.append("EMPTY_SPOT")
        elif cp_type == 1:  # CONSTANT_Utf8
            length = getU2(f)
            data = getUV(f, length)
            value = unicode("")
            index = 0
            while index < length:
                c = struct.unpack(">B", data[index])[0]
                if (c >> 7) == 0:
                    value += unichr(c)
                    index += 1
                elif (c >> 5) == 0b110:
                    b = ord(data[index + 1])
                    assert b & 0x80
                    c = ((c & 0x1f) << 6) + (b & 0x3f)
                    value += unichr(c)
                    index += 2
                elif (c >> 4) == 0b1110:
                    y = ord(data[index + 1])
                    z = ord(data[index + 2])
```

```
                c = ((c & 0xf) << 12) + ((y & 0x3f) << 6) + (z & 0x3f)
                value += unichr(c)
                index += 3
            elif c == 0b11101101:
                v = ord(data[index + 1])
                w = ord(data[index + 2])
                # x = ord(data[index + 3]) No need this is marker
                y = ord(data[index + 4])
                z = ord(data[index + 5])
                c = 0x10000 + ((v & 0x0f) << 16) + ((w & 0x3f) << 10) \
                    + ((y & 0x0f) << 6) + (z & 0x3f)
                value += unichr(c)
                index += 6
            else:
                raise Exception("UTF8 is not fully implemented {0:b}"
                                .format(c))
            pool.append([1, value])
        elif cp_type == 4:  # CONSTANT_Float
            value = struct.unpack('>f', f.read(4))[0]
            pool.append([4, value])
        elif cp_type == 12:  # CONSTANT_NameAndType
            pool.append([12, getU2(f), getU2(f)])
        elif cp_type == 3:  # CONSTANT_Int
            data = f.read(4)
            value = struct.unpack('>i', data)[0]
            # pool.append([3, getU4(f)])
            pool.append([3, value])
        elif cp_type == 5:  # CONSTANT_Long
            value = struct.unpack('>q', f.read(8))[0]
            pool.append([5, value])
            count += 1  # double space in cp
            pool.append("EMPTY_SPOT")
        else:
            raise Exception("Not implemented constant pool entry tag: %s",
                            str(cp_type))
        count += 1
    return pool
```

```
def access_flags(f):
    '''Read flags'''
    flags = getU2(f)
    return flags


def this_super(f):
    '''Constant pool indexes for this/super names.
    Resolve later to unicode/class
    '''
    this_name = getU2(f)
    super_class = getU2(f)
    return (this_name, super_class)


def interfaces(f):
    '''Not really used at runtime, other than casts'''
    data = []
    int_count = getU2(f)
    for i in range(int_count):
        index = getU2(f)
        data.append(index)
    return data


def fields(f):
    '''Read all fields from .class'''
    fields_count = getU2(f)
    data = []
    for i in range(fields_count):
        flags = access_flags(f)
        name = getU2(f)
        desc = getU2(f)
        attributes_count = getU2(f)
        attrs = []
        for k in range(attributes_count):
            attr_name = getU2(f)
```

```
            attr_len = getU4(f)
            attr_data = getUV(f, attr_len)
            attrs.append((attr_name, attr_data))
        # flags, name and description, attrs
        data.append((flags, name, desc, attrs))
    return data


def methods(f):
    '''Read all methods from .class'''
    methods_count = getU2(f)
    data = []
    for i in range(methods_count):
        flag = getU2(f)
        name = getU2(f)
        desc = getU2(f)
        attr_count = getU2(f)
        attrs = []
        for k in range(attr_count):
            attr_name = getU2(f)
            attr_len = getU4(f)
            attr_data = getUV(f, attr_len)
            attrs.append((attr_name, attr_data))
        data.append((flag, name, desc, attrs))
    return data


def make_class(this_name, super_name, constant_pool, all_fields,
all_methods,
               all_interfaces, class_flags):
    '''Actually construct java class from data read earlier'''
    jc = JavaClass()
    jc.flags = class_flags
    if class_flags & 0x0200:  # is interface
        jc.is_interface = True
    jc.constant_pool = constant_pool
    jc.this_name = resolve_to_string(constant_pool, this_name)
    if super_name != 0:
```

```
        jc.super_class = resolve_to_string(constant_pool, super_name)
    add_fields(jc, constant_pool, all_fields)
    add_methods(jc, constant_pool, all_methods)
    add_interfaces(jc, constant_pool, all_interfaces)
    return jc


def resolve_to_string(constant_pool, index):
    '''Unicode string for constant pool entry'''
    data = constant_pool[index]
    if data[0] == 1:
        return unicode(data[1])
    elif data[0] == 7:
        return resolve_to_string(constant_pool, data[1])
    elif data[0] == 12:
        return resolve_to_string(constant_pool, data[1])
    else:
        raise Exception("Not supported string resolution step: {0}".
                        format(data[0]))


def add_fields(jc, constant_pool, data):  # list of (flag, name, desc)
    '''Both static and instance fields'''
    for field in data:
        static = True if field[0] & 0x0008 > 0 else False
        name = resolve_to_string(constant_pool, field[1])
        desc = resolve_to_string(constant_pool, field[2])
        if static:
            default_value = default_for_type(desc)
            jc.static_fields[name] = [desc, default_value]
        else:
            jc.member_fields[name] = desc


def default_for_type(desc):
    '''Default values for primiteves and refs'''
    if desc == "I":
        return 0
```

```
    elif desc == "J":  # long
        return ("long", 0)
    elif desc[0] == "[":  # array
        return None
    elif desc[0] == 'L':  # object
        return None
    elif desc == 'Z':  # boolean
        return 0
    elif desc == 'D':  # double
        return ('double', 0.0)
    elif desc == 'F':  # float
        return ('float', 0.0)
    elif desc == 'C':  # float
        return 0
    elif desc == 'B':  # byte
        return 0
    elif desc == 'S':  # short
        return 0
    raise Exception("Default value not yet supported for " + str(desc))


def parse_code(code, constant_pool):
    '''Each non abstract/native method has this struc'''
    nargs = (ord(code[2]) << 8) + ord(code[3])
    code_len = (ord(code[4]) << 24) + (ord(code[5]) << 16) + \
        (ord(code[6]) << 8) + ord(code[7])
    ex_len = (ord(code[8 + code_len]) << 8) + ord(code[8 + code_len + 1])
    ex_base = 8 + code_len + 2
    extable = []
    for i in range(ex_len):
        data = code[ex_base + i*8:ex_base + i*8 + 8]
        start_pc = struct.unpack('>H', data[0:2])[0]
        end_pc = struct.unpack('>H', data[2:4])[0]
        handler_pc = struct.unpack('>H', data[4:6])[0]
        catch_type = struct.unpack('>H', data[6:8])[0]
        type_name = None
        if catch_type > 0:
            cp_item = constant_pool[catch_type]
```

```
            assert cp_item[0] == 7
            type_name = constant_pool[cp_item[1]][1]
        e = (start_pc, end_pc, handler_pc, catch_type, type_name)
        extable.append(e)
    return (code[8:8+code_len], nargs, extable)


def parse_exceptions(data, constant_pool):
    '''See jvm 7 spec for details'''
    count = (ord(data[0]) << 8) + ord(data[1])
    exceptions = []
    for i in range(count):
        index = struct.unpack('>H', data[i*2 + 2:i*2+4])[0]
        cp_item = constant_pool[index]
        assert cp_item[0] == 7
        ex = constant_pool[cp_item[1]][1]
        exceptions.append(ex)
    return exceptions


def add_methods(jc, constant_pool, data):
    '''Add methods information'''
    # data is a list list of flag, name, desc, attrs; attr list of name/data
    for method in data:
        flags = method[0]
        name = resolve_to_string(constant_pool, method[1])
        desc = resolve_to_string(constant_pool, method[2])
        code = None
        exceptions = []
        for attr in method[3]:
            attr_name = resolve_to_string(constant_pool, attr[0])
            if attr_name == "Code":
                code = attr[1]
            elif attr_name == "Exceptions":
                exception = parse_exceptions(attr[1], constant_pool)
                # ignore
            elif attr_name in ("Signature", "Deprecated",
                               "RuntimeVisibleAnnotations"):
```

```
                pass
            else:
                raise Exception("Unsupported attr {0} in {1}".format(attr_name,
                                name))
        if code is None and (flags & (0x0100 + 0x0400)) == 0:
            raise Exception("No code attr in {0}".format(name))
        if name not in jc.methods:
            jc.methods[name] = {}
        m = jc.methods[name]
        if code is not None:
            code = parse_code(code, constant_pool)
        else:
            code = ("<NATIVE>", 0, [])
        m[desc] = (flags, code[1], code[0], code[2], exceptions)


def add_interfaces(jc, constant_pool, all_interfaces):
    for i in all_interfaces:
        name = resolve_to_string(constant_pool, i)
        jc.interfaces.append(name)

//class_path.py
'''Class path for jar files and directories. Cache all jars content.
JAVA_HOME must be set.

Class path is list of jar files and folders for classes lookup.
Separator ":", (";", ",") are also supported

See START.txt for details
'''

import os
import zipfile


def read_class_path(class_path):
    '''Cache content of all jars.
    Begin with rt.jar
```

```
    '''

    # folders for lookup for class files
    lookup_paths = []
    # content of all jars (name->path to jar)
    jars = {}
    # content of rt.jar
    rt = {}

    # first check local rt.jar
    local_path = os.path.dirname(os.path.realpath(__file__))
    RT_JAR = os.path.join(local_path, "../rt/rt.jar")
    if not os.path.isfile(RT_JAR):
        JAVA_HOME = os.environ.get('JAVA_HOME')
        if JAVA_HOME is None:
            raise Exception("JAVA_HOME is not set")
        if not os.path.isdir(JAVA_HOME):
            raise Exception("JAVA_HOME must be a folder: %s" %
JAVA_HOME)

        RT_JAR = os.path.join(JAVA_HOME, "lib/rt.jar")
        if not os.path.exists(RT_JAR) or os.path.isdir(RT_JAR):
            RT_JAR = os.path.join(JAVA_HOME, "jre/lib/rt.jar")
            if not os.path.exists(RT_JAR) or os.path.isdir(RT_JAR):
                raise Exception("rt.jar not found")

    if not zipfile.is_zipfile(RT_JAR):
        raise Exception("rt.jar is not a zip: %s" % RT_JAR)

    read_from_jar(RT_JAR, rt)

    current = os.getcwd()

    splitter = None
    if ":" in class_path:
        splitter = ":"
    elif ";" in class_path:
        splitter = ";"
```

```
    elif "," in class_path:
        splitter = ","
    else:
        splitter = ":"
    cpaths = class_path.split(splitter)
    for p in cpaths:
        p = p.strip()
        path = os.path.join(current, p)
        if not os.path.exists(path):
            raise Exception("Wrong class path entry: %s (path not found %s)",
                            p, path)
        if os.path.isdir(path):
            lookup_paths.append(path)
        else:
            if zipfile.is_zipfile(path):
                read_from_jar(path, jars)
            else:
                raise Exception("Class path entry %s is not a jar file" % path)

    return (lookup_paths, jars, rt)


def read_from_jar(jar, dict_data):
    '''Read file list from a jar'''
    if not zipfile.is_zipfile(jar):
        raise Exception("Not a jar file: %s" % jar)
    with zipfile.ZipFile(jar, "r") as j:
        for name in j.namelist():
            if name.endswith(".class"):  # at some point save all files
                dict_data[name] = jar

//frame.py
'''Major execution component.

Created for every method execution and placed to thread's stack
'''

f_counter = 1  # make it easy to debug
```

```
class Frame(object):
    '''Frame is created for every method invokation'''

    def __init__(self, _thread, _this_class, _method, _args=[], _desc=""):
        self.thread = _thread
        if _thread is not None:
            self.vm = _thread.vm
        self.this_class = _this_class
        self.pc = 0  # Always points to byte code to be executed
        self.method = _method
        self.code = _method[2]  # method body (bytecode)
        self.stack = []
        self.args = _args
        self.ret = None  # return value for non void
        self.has_result = False  # flag if return value is set
        self.desc = _desc
        global f_counter
        self.id = f_counter
        # to support multithreaded environment
        self.cpc = 0
        self.monitor = None
        f_counter += 1

//jassert.py
'''Java related asserts'''

from pyjvm.jvmo import JArray


def jassert_float(value):
    assert type(value) is tuple and value[0] == "float"


def jassert_double(value):
    assert type(value) is tuple and value[0] == "double"
```

```
def jassert_int(value):
    assert type(value) is int or type(value) is long
    assert -2147483648 <= value <= 2147483647


def jassert_long(value):
    assert type(value) is tuple and value[0] == "long"
    assert -9223372036854775808 <= value[1] <= 9223372036854775807


def jassert_ref(ref):
    assert ref is None or (type(ref) is tuple and ref[0] in ("ref", "vm_ref"))


def jassert_array(array):
    assert array is None or isinstance(array, JArray)

//jvmo.py
"""Major definitions for classes and instances"""

import logging

from pyjvm.prim import PRIMITIVES
from pyjvm.utils import default_for_type

logger = logging.getLogger(__name__)


class JavaClass(object):
    '''Java class representation inside python.
    Is loaded from .class by class loader
    '''

    def __init__(self):
        '''Init major components.
        See models.txt in docs.
        '''
```

```
        self.constant_pool = []
        # each field is name -> (desc, value)
        self.static_fields = {}
        # each field is name -> desc
        self.member_fields = {}
        self.methods = {}  # name-> desc-> (flags, nargs, code)
        self.interfaces = []  # names

        self.this_name = None
        self.super_class = None
        self.flags = 0
        self.is_interface = False
        self.is_primitive = False
        self.is_array = False

        # Reference to java.lang.Class
        self.heap_ref = None

    def print_constant_pool(self):
        '''Debug only purpose'''
        index = 0
        for record in self.constant_pool:
            print str(index) + ":\t" + str(record)
            index += 1

    def static_contructor(self):
        '''Find static constructor among class methods'''
        if "<clinit>" in self.methods:
            return self.methods["<clinit>"]["()V"]
        return None

    def find_method(self, name, signature):
        '''Find method by name and signature in current class or super'''
        if name in self.methods:
            if signature in self.methods[name]:
                return self.methods[name][signature]
        if self.super_class is not None:
            return self.super_class.find_method(name, signature)
```

```
        return None

    def get_instance(self, vm):
        '''Make class instance to be used in java heap'''
        logger.debug("Creating instance of " + str(self.this_name))
        return JavaObject(self, vm)

    def __str__(self):
        s = "JavaClass: "
        s += str(self.this_name) + "\n"
        if self.super_class is None:
            pass
        elif type(self.super_class) is unicode:
            s += "Super: *" + self.super_class + "\n"
        else:
            s += "Super: " + self.super_class.this_name + "\n"
        s += "Static fields: "
        for k in self.static_fields:
            s += "{0}{1} ".format(k, self.static_fields[k])
        s += "\n"
        s += "Member fields: "
        for k in self.member_fields:
            s += "{0}:{1} ".format(k, self.member_fields[k])
        s += "\n"
        s += "Methods:\n"
        for k in self.methods:
            s += "\t" + k + ": "
            for t in self.methods[k]:
                s += t + "::" + str(self.methods[k][t][1]) + ", "
            s += "\n"
        return s


class JavaObject(object):
    '''Java class instance.
    Piece of memory with all instance fields.
    Is created in heap.
    '''
```

```
    def __init__(self, jc, vm):
        self.java_class = jc
        self.fields = {}
        self.fill_fields(jc, vm)
        self.waiting_list = []  # wait/notify/notifyall

    def fill_fields(self, jc, vm):
        '''Init all fields with default values'''
        if jc is None:
            return
        for name in jc.member_fields:
            tp = jc.member_fields[name]
            if tp[0] == 'L':
                #vm.get_class(tp[1:-1])
                pass
            self.fields[name] = default_for_type(jc.member_fields[name])
        self.fill_fields(jc.super_class, vm)

    def __str__(self):
        return "Instance of {0}: {1}".format(self.java_class.this_name,
                                             self.fields)

    def __repr__(self):
        return self.__str__()


class JArray(object):
    '''Java array

    Lives in heap and has corresponding java_class
    '''
    def __init__(self, jc, vm):
        self.java_class = jc
        self.fields = {}
        self.values = []
```

```
def array_class_factory(vm, name):
    assert name[0] == '['
    name = name[1:]
    if name[0] == 'L':
        name = name[1:-1]
        vm.get_class(name)  # make sure it's in
        jc = JavaClass()
        jc.is_array = True
        jc.this_name = "[L" + name + ";"
        jc.super_class = vm.get_class("java/lang/Object")
        jc.interfaces = ["java/lang/Cloneable", "java/io/Serializable"]
        return jc
    if name[0] == '[':
        jc = JavaClass()
        jc.is_array = True
        jc.this_name = "[" + name
        jc.super_class = vm.get_class("java/lang/Object")
        jc.interfaces = ["java/lang/Cloneable", "java/io/Serializable"]
        return jc
    assert name in PRIMITIVES

    vm.get_class(PRIMITIVES[name])  # make sure class is in

    jc = JavaClass()
    jc.is_array = True
    jc.interfaces = ["java/lang/Cloneable", "java/io/Serializable"]
    jc.this_name = "[" + name
    jc.super_class = vm.get_class("java/lang/Object")
    return jc

//natives.py
'''Natives methods handler '''

import logging

from pyjvm.platform.java.lang.clazz import *
from pyjvm.platform.java.lang.double import *
from pyjvm.platform.java.lang.float import *
```

```
from pyjvm.platform.java.lang.object import *
from pyjvm.platform.java.lang.runtime import *
from pyjvm.platform.java.lang.string import *
from pyjvm.platform.java.lang.system import *
from pyjvm.platform.java.lang.thread import *
from pyjvm.platform.java.lang.throwable import *
from pyjvm.platform.java.io.filedescriptor import *
from pyjvm.platform.java.io.fileinputstream import *
from pyjvm.platform.java.io.fileoutputstream import *
from pyjvm.platform.java.io.filesystem import *
from pyjvm.platform.java.security.accesscontroller import *
from pyjvm.platform.sun.misc.unsafe import *
from pyjvm.platform.sun.misc.vm import *
from pyjvm.platform.sun.reflect.nativeconstructoraccessorimpl import *
from pyjvm.platform.sun.reflect.reflection import *

logger = logging.getLogger(__name__)


def exec_native(frame, args, klass, method_name, method_signature):
    '''Handle calls to java's native methods.
    Create function name from class and method names and call that
    implementation.
    See native.txt in documentation.
    '''
    if method_name == "registerNatives" and method_signature == "()V":
        logger.debug("No need to call native registerNatives()V for class:
%s",
                     klass.this_name)
        return
    lookup_name = "%s_%s_%s" % (klass.this_name, method_name,
method_signature)
    lookup_name = lookup_name.replace("/", "_")
    lookup_name = lookup_name.replace("(", "_")
    lookup_name = lookup_name.replace(")", "_")
    lookup_name = lookup_name.replace("[", "_")
    lookup_name = lookup_name.replace(";", "_")
    lookup_name = lookup_name.replace(".", "_")
```

```
    if lookup_name not in globals():
        logger.error("Native not yet ready: %s:%s in %s", method_name,
                     method_signature, klass.this_name)
        raise Exception("Op ({0}) is not yet supported in natives".format(
                        lookup_name))
    logger.debug("Call native: %s", lookup_name)
    globals()[lookup_name](frame, args)

//prim.py
"""Mapping between JDK type id and class name"""


PRIMITIVES = {'B': 'byte', 'C': 'char', 'D': 'double',
              'F': 'float', 'I': 'int', 'J': 'long', 'S': 'short',
              'Z': 'boolean'}

//thread.py
'''JMV threads'''


class Thread(object):
    '''JMV thread.
    See threads.txt in documentation for details.
    '''

    def __init__(self, _vm, _java_thread):
        '''Init pyjvm thread
        _vm reference to current vm
        _java_thread reference to java's Thread instance in heap
        '''
        # One frame per method invocation
        self.frame_stack = []
        self.vm = _vm
        # Support looping for multi-threaded apps
        self.next_thread = None
        self.prev_thread = None
        # Reference to java's Thread instances
        self.java_thread = _java_thread
```

```
        self.is_alive = False
        self.waiting_notify = False
        self.is_notified = False
        self.monitor_count_cache = 0
        # For sleep(long) support
        self.sleep_until = 0
        if _java_thread is not None:
            obj = _vm.heap[_java_thread[1]]
            obj.fields["@pvm_thread"] = self


class SkipThreadCycle(Exception):
    '''Thread may skip his execution quota in case when a monitor
    is busy or sleep was called
    '''
    pass

//throw.py
'''Java Exception'''


class JavaException(Exception):
    '''PY excpetion.

    Real heap reference is stored in ref
    '''

    def __init__(self, _vm, _ref):
        self.vm = _vm
        self.ref = _ref
        self.stack = []

    def __str__(self):
        ex = self.vm.heap[self.ref[1]]
        return str(ex)

//utills.py
"""Common utils"""
```

```
def arr_to_string(str_arr):
    '''Convert string's array to real unicode string'''
    result_string = ""
    for char_ in str_arr:
        result_string += str(unichr(char_))
    return result_string


def str_to_string(vm, ref):
    '''Convert java string reference to unicode'''
    if ref is None:
        return "NULL"
    heap_string = vm.heap[ref[1]]
    value_ref = heap_string.fields["value"]
    value = vm.heap[value_ref[1]]  # this is array of chars
    return arr_to_string(value.values)


def args_count(desc):
    '''Get arguments count from method signature string
    e.g. ()V - 0; (II)V - 2 (two int params)
    '''
    count = _args_count(desc[1:])
    return count


def _args_count(desc):
    '''Recursive parsing for method signuture'''
    char_ = desc[0]
    if char_ == ")":
        return 0
    if char_ in ["B", "C", "F", "I", "S", "Z"]:
        return 1 + _args_count(desc[1:])
    if char_ in ["J", "D"]:
        return 2 + _args_count(desc[1:])
    if char_ == "L":
```

```
        return 1 + _args_count(desc[desc.index(";") + 1:])
    if char_ == "[":
        return _args_count(desc[1:])
    raise Exception("Unknown type def %s", str(char_))


def default_for_type(desc):
    '''Get default value for specific type'''
    if desc == "I":
        return 0
    elif desc == "J":  # long
        return ("long", 0)
    elif desc[0] == "[":  # array
        return None
    elif desc[0] == 'L':  # object
        return None
    elif desc == 'Z':  # boolean
        return 0
    elif desc == 'D':  # double
        return ("double", 0.0)
    elif desc == 'F':  # float
        return ("float", 0.0)
    elif desc == 'C':  # char
        return 0
    elif desc == 'B':  # boolean
        return 0
    raise Exception("Default value not yet supported for " + desc)


def category_type(value):
    '''Get category type of a variable according to jdk specs

    long, double are 2, others are 1'''
    if type(value) is tuple and value[0] in ('long', 'double'):
        return 2
    else:
        return 1
```

```
//vmo.py
("vm_ref", x), where x < 0; versus normal heap owned objects:
("ref", y), y > 0
When a method is called on these vm owned instances, python code is
executed. This is different from handling native methods.

Example of vm owned object is STDOUT (print something on the screen).
"""

import os
import logging
import sys

from pyjvm.jassert import jassert_array
from pyjvm.utils import str_to_string

logger = logging.getLogger(__name__)

VM_OBJECTS = {
    "Stdout.OutputStream": -1,
    "System.Properties": -2,
    "JavaLangAccess": -3,
    "Stdin.InputputStream": -4,
    "FileSystem": -5
    }

VM_CLASS_NAMES = {
    -1: "java/io/OutputStream",
    -2: "java/util/Properties",
    -3: "sun/misc/JavaLangAccess",
    -4: "java/io/InputStream",
    -5: "java/io/FileSystem"
}


def vm_obj_call(frame, args, method_name, method_signature):
    '"Called by invoke method operations when instance ref is ("vm_ref",
x).
```

```
    This methods converts call to function name defined in this file. It is
    executed (python code) instead of original byte code.
    '''
    ref = args[0]
    assert type(ref) is tuple
    assert ref[0] == "vm_ref"
    assert ref[1] < 0
    logger.debug("VM owned obj call: %s", ref[1])
    lookup_name = "vmo%s_%s_%s" % (ref[1] * -1, method_name,
method_signature)
    lookup_name = lookup_name.replace("/", "_")
    lookup_name = lookup_name.replace("(", "_")
    lookup_name = lookup_name.replace(")", "_")
    lookup_name = lookup_name.replace("[", "_")
    lookup_name = lookup_name.replace(";", "_")
    lookup_name = lookup_name.replace(".", "_")
    if lookup_name not in globals():
        logger.error("VMOcall not implemented: %s:%s for %d",
method_name,
                     method_signature, ref[1])
        raise Exception("Op ({0}) is not yet supported in vmo".format(
            lookup_name))
    globals()[lookup_name](frame, args)


def vmo_check_cast(vm, vmo_id, klass):
    '''check cast for specific vmo object

    vmo_id is less than zero, klass is JavaClass
    True if vmo is subclass of klass or implements interface klass
    '''
    this_klass = VM_CLASS_NAMES[vmo_id]
    klass_name = klass.this_name
    while klass is not None:
        if klass.this_name == this_klass:
            return True
        else:
            klass = klass.super_class
```

```
    vmo_klass = vm.get_class(this_klass)
    for i in vmo_klass.interfaces:
        if i == klass_name:
            return True
    return False


def vmo1_write___BII_V(frame, args):
    '''java.io.OutputStream
    void write(byte[] b, int off, int len)
    '''
    buf = args[1]
    offset = args[2]
    length = args[3]
    arr = frame.vm.heap[buf[1]]
    jassert_array(arr)
    chars = arr.values
    for index in range(offset, offset + length):
        sys.stdout.write(chr(chars[index]))


def vmo2_getProperty__Ljava_lang_String__Ljava_lang_String_(frame,
args):
    '''java.lang.System
    public static String getProperty(String key)
    This is call to java.util.Properties object
    '''
    s_ref = args[1]
    value = str_to_string(frame.vm, s_ref)
    # refactor this code someday
    # ok for now, as all refs are cached
    props = {}
    props["file.encoding"] = frame.vm.make_heap_string("utf8")
    props["line.separator"] = frame.vm.make_heap_string("\n")
    if value in props:
        ref = props[value]
        assert type(ref) is tuple and ref[0] == "ref"
        frame.stack.append(ref)
```

```
        return
    frame.stack.append(None)


def vmo4_read___BII_I(frame, args):
    '''In will be truncated at 8k'''
    # TODO all exception checks
    ref = args[1]
    offset = args[2]
    length = args[3]
    o = frame.vm.heap[ref[1]]
    array = o.values

    c = sys.stdin.read(1)
    if c == '':
        frame.stack.append(-1)
    array[offset] = ord(c)
    if ord(c) == 10:
        frame.stack.append(1)
        return
    i = 1
    while i < length:
        c = sys.stdin.read(1)
        if c == '':
            break
        array[offset + i] = ord(c)
        i += 1
        if ord(c) == 10:
            break
    frame.stack.append(i)


def vmo4_available___I(frame, args):
    '''This is always zero. No support for buffering'''
    frame.stack.append(0)


def vmo4_read___I(frame, args):
```

```
    '''Read single byte'''
    c = sys.stdin.read(1)
    if c == '':
        frame.stack.append(-1)
    else:
        frame.stack.append(ord(c))


def vmo5_getSeparator___C(frame, args):
    '''Always slash'''
    frame.stack.append(ord('/'))


def vmo5_getPathSeparator___C(frame, args):
    '''Do not check operating system'''
    frame.stack.append(ord(':'))


def vmo5_normalize__Ljava_lang_String__Ljava_lang_String_(frame,
args):
    '''Normalize according api rules'''
    s_ref = args[1]
    value = str_to_string(frame.vm, s_ref)
    norm = os.path.normpath(value)
    if value != norm:
        s_ref = frame.vm.make_heap_string(norm)
    frame.stack.append(s_ref)


def vmo5_prefixLength__Ljava_lang_String__I(frame, args):
    '''This is shortcut'''
    # s_ref = args[1]
    # value = str_to_string(frame.vm, s_ref)
    frame.stack.append(0)  # for now


def vmo5_getBooleanAttributes__Ljava_io_File__I(frame, args):
    '''See javadoc for details. Subset of all attributes is supported'''
```

```
    ref = args[1]
    assert ref is not None  # NPE
    o = frame.vm.heap[ref[1]]
    path_ref = o.fields['path']
    path = str_to_string(frame.vm, path_ref)
    result = 0
    if os.path.exists(path):
        result |= 0x01
        if not os.path.isfile(path):
            result |= 0x04
    frame.stack.append(result)

//vm.py
Initialization, threads, frame management.
'''

import logging
from collections import deque

from pyjvm.bytecode import get_operation, get_operation_name
from pyjvm.class_loader import class_loader
from pyjvm.class_path import read_class_path
from pyjvm.frame import Frame
from pyjvm.jvmo import array_class_factory
from pyjvm.jvmo import JArray
from pyjvm.jvmo import JavaClass
from pyjvm.thread import Thread
from pyjvm.thread import SkipThreadCycle

from pyjvm.throw import JavaException

from pyjvm.vmo import VM_OBJECTS

from pyjvm.ops.ops_names import ops_name
from pyjvm.ops.ops_arrays import *
from pyjvm.ops.ops_calc import *
from pyjvm.ops.ops_cond import *
from pyjvm.ops.ops_convert import *
```

```
from pyjvm.ops.ops_fields import *
from pyjvm.ops.ops_invokespecial import *
from pyjvm.ops.ops_invokestatic import *
from pyjvm.ops.ops_invokevirtual import *
from pyjvm.ops.ops_invokeinterface import *
from pyjvm.ops.ops_misc import *
from pyjvm.ops.ops_ret import *
from pyjvm.ops.ops_setget import *
from pyjvm.ops.ops_shift import *

logger = logging.getLogger(__name__)


def vm_factory(class_path="."):
    '''Create JVM with specific class path'''
    return VM(class_path)


class VM(object):
    '''JVM implementation.
    See vm.txt in docs
    '''

    # Mark for vm caching
    serialization_id = 0
    initialized = False

    def __init__(self, _class_path="."):
        logger.debug("Creating VM")

        # Major memory structures
        self.perm_gen = {}
        self.heap = {}
        self.heap_next_id = 1
        #todo clean up self.cache_klass_klass = {}
        self.global_strings = {}

        # Handle for linked list of threads
```

```
        self.threads_queue = deque()
        self.non_daemons = 0

        self.top_group = None
        self.top_thread = None
        self.top_group_ref = None
        self.top_thread_ref = None

        self.class_path = read_class_path(_class_path)

        self.init_default_thread()

        # Load System and init major fields
        system_class = self.get_class("java/lang/System")

        # Set System.props to vm owned object
        system_class.static_fields["props"][1] = ("vm_ref",
                                            VM_OBJECTS[
                                                "System.Properties"])

        # STDout initialization using vm owned object
        ps_class = self.get_class("java/io/PrintStream")
        ps_object = ps_class.get_instance(self)
        ps_ref = self.add_to_heap(ps_object)
        method = ps_class.find_method("<init>",
"(Ljava/io/OutputStream;)V")
        std_out_ref = ("vm_ref", VM_OBJECTS["Stdout.OutputStream"])
        thread = Thread(self, None)
        frame = Frame(thread, ps_class, method, [ps_ref, std_out_ref],
                      "PrintStream init")
        thread.frame_stack.append(frame)

        logger.debug("Run PrintStream init")
        self.run_thread(thread)  # Run exclusive thread
        system_class.static_fields["out"][1] = ps_ref

        system_class.static_fields["in"][1] = \
            ("vm_ref", VM_OBJECTS["Stdin.InputputStream"])
```

```
        # Additional parameters
        system_class.static_fields["lineSeparator"][1] = \
            self.make_heap_string("\n")

        # Load additional classes to speed up booting
        self.touch_classes()

        self.initialized = True

        logger.debug("VM created")

    def init_default_thread(self):
        '''Create initial thread group and thread.
        Both are java's objects
        '''
        tg_klass = self.get_class("java/lang/ThreadGroup")
        t_klass = self.get_class("java/lang/Thread")
        tg = tg_klass.get_instance(self)
        t = t_klass.get_instance(self)

        tg.fields["name"] = self.make_heap_string("system")
        tg.fields["maxPriority"] = 10
        t.fields["priority"] = 5
        t.fields["name"] = self.make_heap_string("system-main")
        t.fields["blockerLock"] = self.add_to_heap(
            self.get_class("java/lang/Object").get_instance(self))

        tg_ref = self.add_to_heap(tg)
        t_ref = self.add_to_heap(t)
        t.fields["group"] = tg_ref

        # Add thread to threadgroup; call byte code of void add(Thread)
        pvm_thread = Thread(self, t_ref)
        pvm_thread.is_alive = True
        method = tg_klass.find_method("add", "(Ljava/lang/Thread;)V")
        args = [None]*method[1]
        args[0] = tg_ref
```

```
        args[1] = t_ref
        frame = Frame(pvm_thread, tg_klass, method, args, "system tg init")
        pvm_thread.frame_stack.append(frame)
        self.run_thread(pvm_thread)

        self.top_group = tg
        self.top_thread = t
        self.top_group_ref = tg_ref
        self.top_thread_ref = t_ref

    def run_vm(self, main_klass, method, m_args):
        '''Run initialized vm with specific method of a class.
        This is class entered from command line. Method is looked up
        void main(String args[]).
        For more details see methods.txt in docs.
        '''
        t_klass = self.get_class("java/lang/Thread")
        t = t_klass.get_instance(self)
        t.fields["priority"] = 5
        t.fields["name"] = self.make_heap_string("main")
        t.fields["blockerLock"] = self.add_to_heap(
            self.get_class("java/lang/Object").get_instance(self))
        t_ref = self.add_to_heap(t)
        t.fields["group"] = self.top_group_ref

        pvm_thread = Thread(self, t_ref)
        pvm_thread.is_alive = True
        frame = Frame(pvm_thread, main_klass, method, m_args, "main")
        pvm_thread.frame_stack.append(frame)

        self.add_thread(pvm_thread)
        logger.debug("run thread pool")
        self.run_thread_pool()

    def get_class(self, class_name):
        '''Returns initialized class from pool (perm_gen) or loads
        it with class loader (and running static constructor).
        Getting a class might result in loading it's super first.
```

```
        '''
        if class_name is None:
            return  # this is look up for Object's super, which is  None
        if class_name in self.perm_gen:
            return self.perm_gen[class_name]
        if class_name[0] == '[':  # special treatment for arrays
            java_class = array_class_factory(self, class_name)
            lang_clazz = self.get_class("java/lang/Class")
            clazz_object = lang_clazz.get_instance(self)
            clazz_object.fields["@CLASS_NAME"] = class_name
            ref = self.add_to_heap(clazz_object)
            java_class.heap_ref = ref
            self.perm_gen[class_name] = java_class
            return java_class
        if class_name in ['byte', 'char', 'double', 'float', 'int', 'long',
                          'short', 'boolean']:
            java_class = JavaClass()
            self.perm_gen[class_name] = java_class
            java_class.is_primitive = True
            java_class.this_name = class_name
            lang_clazz = self.get_class("java/lang/Class")
            clazz_object = lang_clazz.get_instance(self)
            clazz_object.fields["@CLASS_NAME"] = class_name
            ref = self.add_to_heap(clazz_object)
            java_class.heap_ref = ref
            return java_class
        logger.debug("Class {0} not yet ready".format(class_name))
        java_class = class_loader(class_name, self.class_path)
        super_class = java_class.super_class
        if type(super_class) is unicode:  # lame check
            super_class = self.get_class(super_class)
            java_class.super_class = super_class
        logger.debug("Loaded class def\n{0}".format(java_class))
        self.perm_gen[class_name] = java_class
        # create actual java.lang.Class instance
        lang_clazz = self.get_class("java/lang/Class")
        clazz_object = lang_clazz.get_instance(self)
        clazz_object.fields["@CLASS_NAME"] = class_name
```

```
        ref = self.add_to_heap(clazz_object)
        java_class.heap_ref = ref
        self.run_static_constructor(java_class)
        return java_class

    def get_class_class(self, klass):
        '''Get class of class.
        Basically this is heap owned version of java.lang.Class
        '''
        return klass.heap_ref

    def run_static_constructor(self, java_class):
        '''Static constructor is run for every class loaded by class loader.
        It is executed in thread exclusive mode.
        '''
        logger.debug("Running static constructor for %s",
                     java_class.this_name)
        method = java_class.static_contructor()
        if method is None:
            logger.debug("No static constructor for %s",
                         java_class.this_name)
            return
        pvm_thread = Thread(self, self.top_thread_ref)
        pvm_thread.is_alive = True
        frame = Frame(pvm_thread, java_class, method, [None]*method[1],
                      "<clinit:{0}>".format(java_class.this_name))
        pvm_thread.frame_stack.append(frame)
        self.run_thread(pvm_thread)

        logger.debug("Finished with static constructor for %s",
                     java_class.this_name)

    def object_of_klass(self, o, klass_name):
        '''instanceOf implementation'''
        if o is None:
            return False
        if klass_name is None:
            return True
```

```
        klass = o.java_class
        while klass is not None:
            if klass_name == klass.this_name:
                return True
            klass = klass.super_class
        return False

    def add_to_heap(self, item):
        '''Put an item to java heap returning reference.
        Reference is in format ("ref", number)
        '''
        ref = self.heap_next_id
        self.heap[ref] = item
        self.heap_next_id += 1
        return ("ref", ref)

    def make_heap_string(self, value):
        '''Take python string and put java.lang.String instance to heap.
        String is represented by char array in background.
        Reference in heap is returned.
        Global caching is supported for all strings (same string always has
        same reference in heap)
        '''
        if value in self.global_strings:
            return self.global_strings[value]
        values = []
        for c in value:
            values.append(ord(c))
        array_class = self.get_class("[C")
        array = JArray(array_class, self)
        array.values = values
        arr_ref = self.add_to_heap(array)
        c = self.get_class("java/lang/String")
        o = c.get_instance(self)
        o.fields["value"] = arr_ref
        ref = self.add_to_heap(o)
        self.global_strings[value] = ref
        return ref
```

```
    def touch_classes(self):
        '''Touch some useful classes to speed up booting for cached vm'''
        self.get_class("java/lang/String")
        self.get_class("java/lang/Class")
        self.get_class("java/nio/CharBuffer")
        self.get_class("java/nio/HeapCharBuffer")
        self.get_class("java/nio/charset/CoderResult")
        self.get_class("java/nio/charset/CoderResult$1")
        self.get_class("java/nio/charset/CoderResult$Cache")
        self.get_class("java/nio/charset/CoderResult$2")

        thread_klass = self.get_class("java/lang/Thread")
        thread_klass.static_fields["MIN_PRIORITY"][1] = 1
        thread_klass.static_fields["NORM_PRIORITY"][1] = 5
        thread_klass.static_fields["MAX_PRIORITY"][1] = 10

    def add_thread(self, thread):
        '''Add py thread to pool'''
        self.threads_queue.append(thread)
        assert thread.java_thread is not None
        java_thread = self.heap[thread.java_thread[1]]
        if java_thread.fields["daemon"] == 0:
            self.non_daemons += 1

    def run_thread_pool(self):
        '''Run all threads.
        Threads are run one-by-one according to quota'''
        while len(self.threads_queue) > 0:
            thread = self.threads_queue.popleft()
            self.run_thread(thread, 100)
            if len(thread.frame_stack) == 0:
                thread.is_alive = False
                j_thread = self.heap[thread.java_thread[1]]
                assert j_thread is not None
                for o in j_thread.waiting_list:
                    o.is_notified = True
                java_thread = self.heap[thread.java_thread[1]]
```

```
            if java_thread.fields["daemon"] == 0:
                self.non_daemons -= 1
                if self.non_daemons == 0:
                    break
            else:
                self.threads_queue.append(thread)

    def run_thread(self, thread, quota=-1):
        '''Run single thread according to quota.
        Quota is number of byte codes to be executed.
        Quota -1 runs entire thread in exclusive mode.

        For each byte code specific operation function is called.
        Operation can throw exception.
        Thread may be busy (e.g. monitor is not available).
        Returns from syncronized methods are handled.
        '''
        frame_stack = thread.frame_stack
        while len(frame_stack) > 0:
            frame = frame_stack[-1]  # get current
            if frame.pc < len(frame.code):
                op = frame.code[frame.pc]
                frame.cpc = frame.pc
                frame.pc += 1
                # Make function name to be called
                op_call = hex(ord(op))

                logger.debug("About to execute {2}: op_{0} ({3}) in
{1}".format(
                    op_call, frame.id, frame.pc - 1, get_operation_name(op_call)))

                opt = get_operation(op_call)
                if opt is None:
                    raise Exception("Op ({0}) is not yet supported".format(
                        op_call))
                try:
                    try:
                        opt(frame)
```

```
                    logger.debug("Stack:" + str(frame.stack))
                except SkipThreadCycle:
                    # Thread is busy, call the same operation later
                    frame.pc = frame.cpc
                    break
            except JavaException as jexc:
                # Exception handling
                ref = jexc.ref
                exc = self.heap[ref[1]]
                handled = False
                while not handled:
                    for (start_pc, end_pc, handler_pc, catch_type,
                         type_name) in frame.method[3]:
                        if start_pc <= frame.cpc < end_pc and \
                                self.object_of_klass(exc, type_name):
                            frame.pc = handler_pc
                            frame.stack.append(ref)
                            handled = True
                            break
                    if handled:
                        break
                    frame_stack.pop()
                    if len(frame_stack) == 0:
                        raise
                    frame = frame_stack[-1]

        else:
            # Frame is done
            frame_stack.pop()
            if frame.monitor is not None:
                assert frame.monitor.fields["@monitor"] == frame.thread
                frame.monitor.fields["@monitor_count"] -= 1
                if frame.monitor.fields["@monitor_count"] == 0:
                    del frame.monitor.fields["@monitor"]
                    del frame.monitor.fields["@monitor_count"]
                    frame.monitor = None
            # handle possible return VALUE
            if frame.has_result:
```

```
                if len(frame_stack) > 0:
                    frame_stack[-1].stack.append(frame.ret)

            if quota != -1:
                quota -= 1
                if quota == 0:
                    break

    def raise_exception(self, frame, name):
        '''Util method to raise an exception based on name.
        e.g. java.lang.NullPointerException

        Exception is created on heap and throw op is called
        '''
        ex_klass = self.get_class(name)
        ex = ex_klass.get_instance(self)
        ref = self.add_to_heap(ex)

        method = ex_klass.find_method("<init>", "()V")
        m_args = [None]*method[1]
        m_args[0] = ref

        pvm_thread = Thread(self, None)
        pvm_thread.is_alive = True
        sub = Frame(pvm_thread, ex_klass, method, m_args, "exinit")
        pvm_thread.frame_stack.append(sub)
        self.run_thread(pvm_thread)

        frame.stack.append(ref)
        get_operation('0xbf')(frame)
```