# 301AA - Advanced Programming

Lecturer: **Andrea Corradini**
andrea@di.unipi.it
http://pages.di.unipi.it/corradini/

Course pages:
http://pages.di.unipi.it/corradini/Didattica/AP-18/

***AP-2018-04***: *Runtime Systems and intro to JVM*

# Overview

- Runtime Systems
- The Java Runtime Environment
- The JVM as an abstract machine
- JVM Data Types
- JVM Runtime Data Areas
- Multithreading
- Per-thread Data Areas
- Dynamic Linking
- JIT compilation
- Method Area

# Runtime system

- Every programming language defines an **execution model**
- A **runtime system** implements (part of) such execution model, providing support during the execution of corresponding programs
- **Runtime support** is needed both by *interpreted* and by *compiled* programs, even if typically less by the latter

# Runtime system (2)

- The runtime system can be made of
  - Code in the executing program generated by the compiler
  - Code running in other threads/processes during program execution
  - Language libraries
  - Operating systems functionalities
  - The interpreter / virtual machine itself

# Runtime Support needed for…

- Memory management
  - Stack management: Push/pop of activation records
  - Heap management: allocation, garbage collection
  - → **Chapter 7 of "Dragon Book"**
- Input/Output
  - Interface to file system / network sockets / I/O devices
- Interaction with the **runtime environment**,
  - state values accessible during execution (eg. environment variables)
  - active entities like disk drives and people via keyboards.

# Runtime Support needed for… (2)

- Parallel execution via threads/tasks/processes
- Dynamic type checking and dynamic binding
- Dynamic loading and linking of modules
- Debugging
- Code generation (for JIT compilation) and Optimization
- Verification and monitoring

# Java Runtime Enviroment - JRE

- Includes all what is needed to run compiled Java programs
  - JVM – Java Virtual Machine
  - JCL – Java Class Library (Java API)
- We shall focus on the JVM as a real runtime system covering most of the functionalities just listed
- Reference documentation:
  - The Java$^{TM}$ Virtual Machine Specification, Java SE 8 Edition
    **https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf**
  - The Java Language Specification, Java SE 8 Edition
    **https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf**

# New "short-term" releases of Java

- ➔ **Java 9**, released in September 2017
  - Added module system
  - https://www.oracle.com/corporate/features/understanding-java-9-modules.html
- ➔ **Java 10**, released in March 2018
  - Type inference of local variables
  - https://developer.oracle.com/java/jdk-10-local-variable-type-inference
  - https://medium.com/the-java-report/java-10-sneak-peek-local-variable-type-inference-var-3022016e1a2b
- ➔ **Java 11 (LTS)**, to be released in September 2018

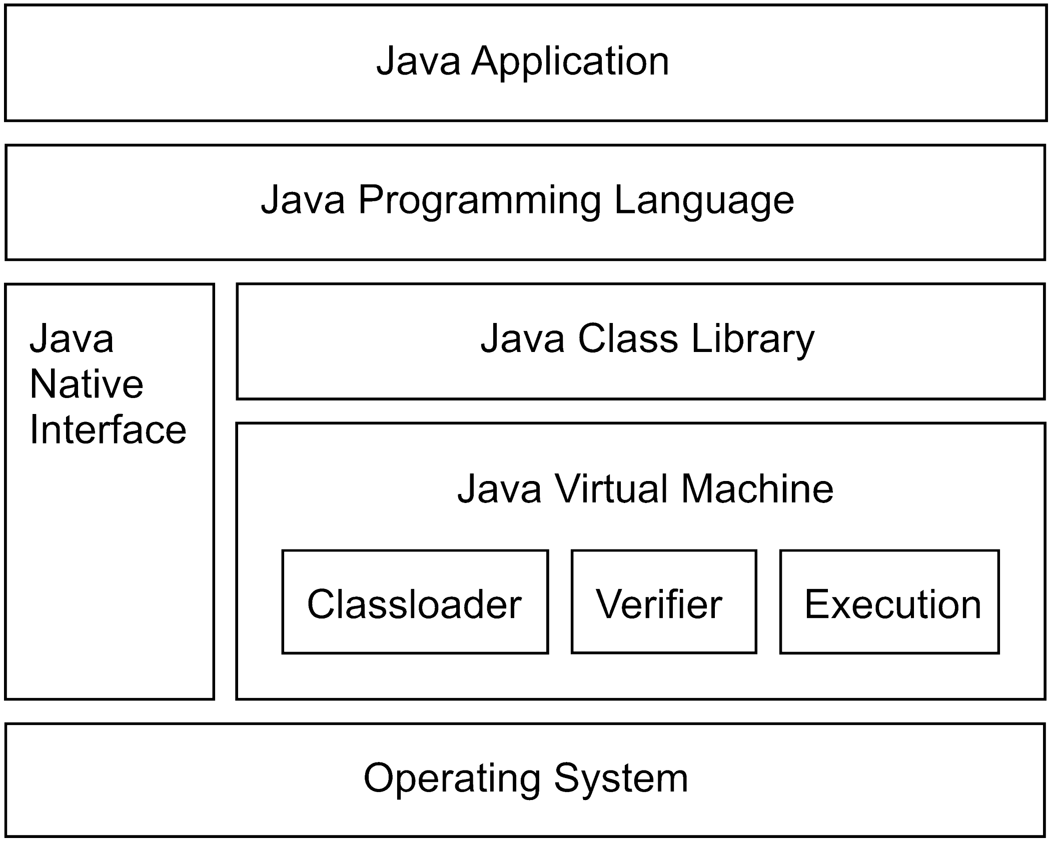
# What is the JVM?

- The **JVM** is an **abstract** machine in the true sense of the word.
- The JVM specification does *not* give implementation details like memory layout of run-time data area, garbage-collection algorithm, internal optimization (can be dependent on target OS/platform, performance requirements, etc.)
- The JVM specification defines a machine independent “**class file format**” that all JVM implementations must support
- The JVM imposes **strong syntactic** and **structural constraints** on the code in a class file. Any language with functionality that can be expressed in terms of a valid class file can be hosted by the JVM

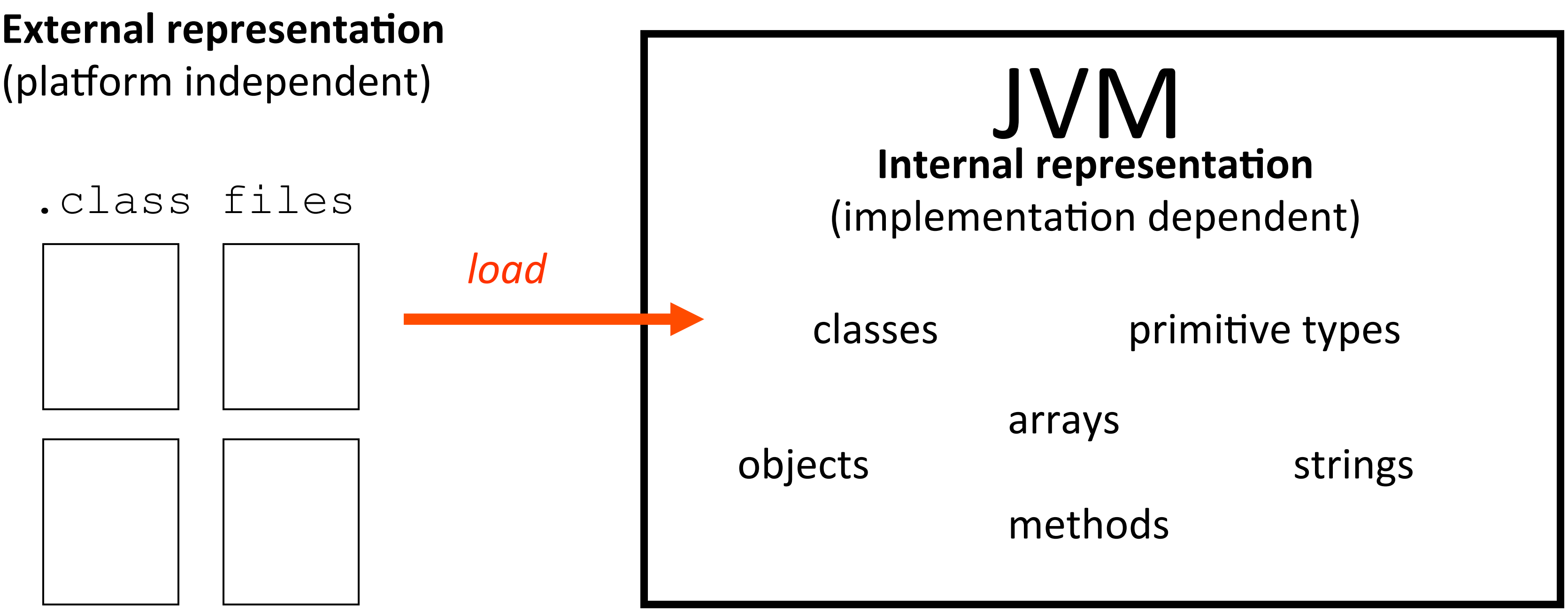
# Execution model

- JVM is a *multi-threaded stack based machine*
- JVM instructions
  - implicitly take arguments from the top of the operand stack of the current frame
  - put their result on the top of the operand stack
- The operand stack is used to
  - pass arguments to methods
  - return a result from a method
  - store intermediate results while evaluating expressions
  - store local variables

# Java Abstact Machine Hierarchy



# Class Files and Class File Format



# JVM Data Types

**Primitive types:**
- boolean: `boolean` (support only for arrays)
- numeric integral: `byte`, `short`, `int`, `long`, `char`
- numeric floating point: `float`, `double`
- internal, for exception handling: `returnAddress`

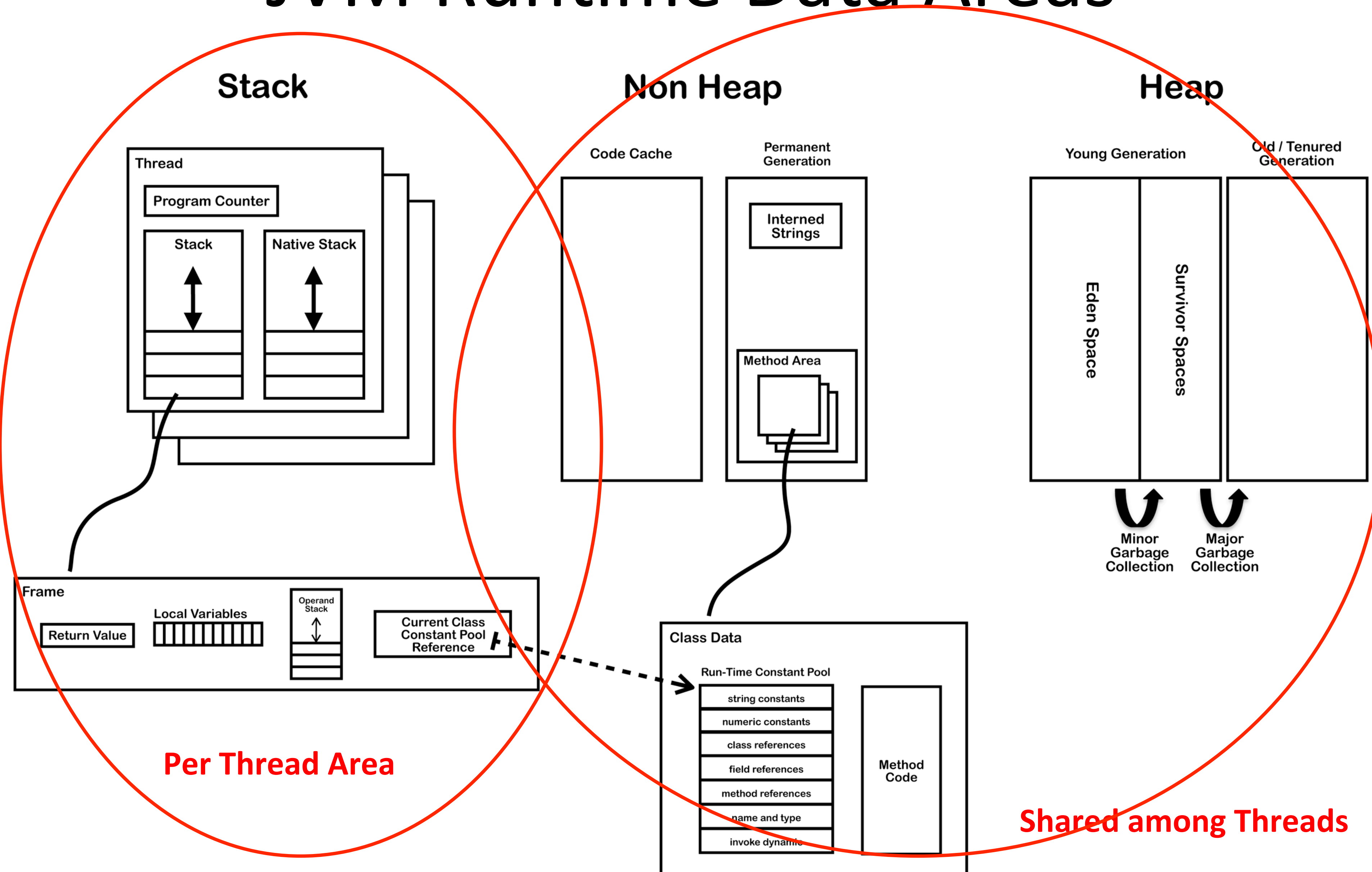
**Reference types:**
- class types
- array types
- interface types

**Note:**
- No type information on local variables at runtime
- Types of operands specified by opcodes (eg: `iadd`, `fadd`, ….)

# Object Representation

- Left to the implementation
  - Including concrete value of `null`
- This adds extra level of indirection
  - need pointers to instance data and class data
  - make garbage collection easier
- Object representation must include
  - mutex lock
  - GC state (flags)

# JVM Runtime Data Areas



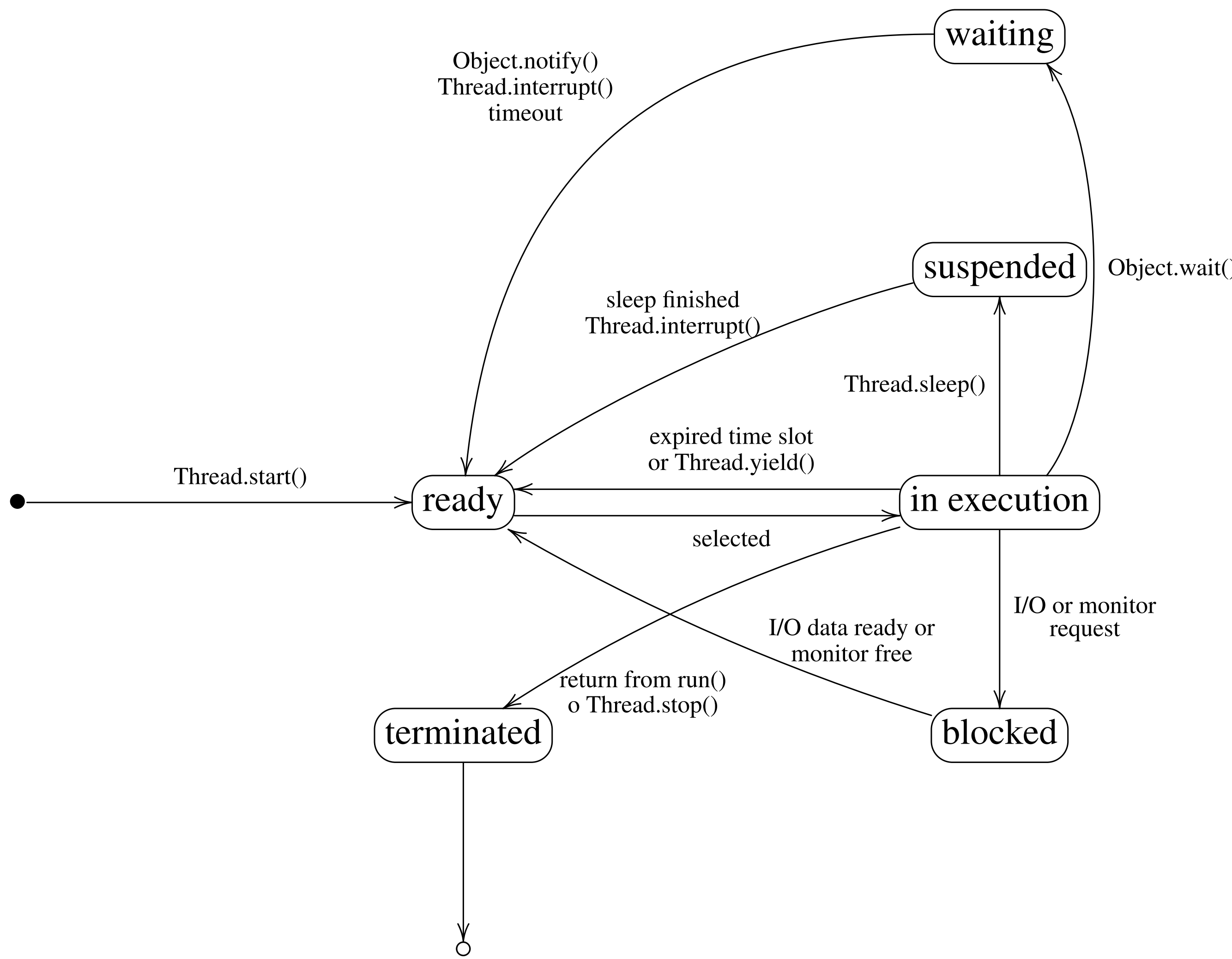
# Threads

- JVM allows multiple threads per application, starting with `main`
- Created as instances of `Thread` invoking `start()` (which invokes `run()`)
- Several background (daemon) system threads for
  - Garbage collection, finalization
  - Signal dispatching
  - Compilation, etc.
- Threads can be supported by time-slicing and/or multiple processors

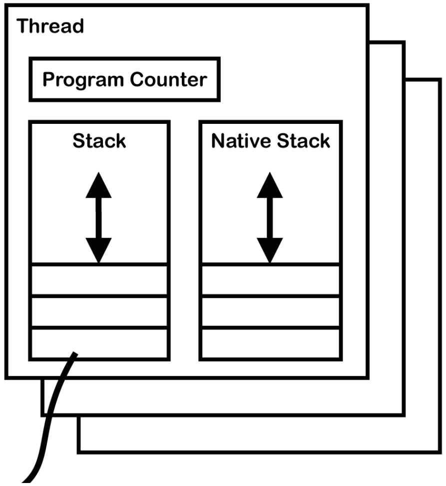
# Threads (2)

- Threads have shared access to heap and persistent memory
- Complex specification of consistency model
  - volatiles
  - working memory vs. general store
  - non-atomic longs and doubles
- The *Java programming language memory model* prescribes the behaviour of multithreaded programs (JLS-8 Ch. 17)

# Java Thread Life Cycle

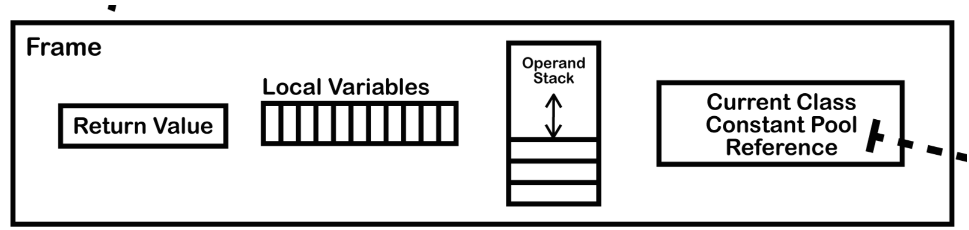


# Per Thread Data Areas




- **pc**: pointer to next instruction in *method area*
  – undefined if current method is *native*
- The **java stack**: a stack of *frames* (or *activation records*).
  – A new frame is created each time a method is invoked and it is destroyed when the method completes.
  – The JVMS does not require that frames are allocated contiguously
- The **native stack**: is used for invocation of native functions, through the JNI (Java Native Interface)
  – When a native function is invoked, eg. a C function, execution continues using the native stack
  – Native functions can call back Java methods, which use the Java stack

# Structure of frames




- **Local Variable Array** (32 bits) containing
  - Reference to `this` (if instance method)
  - Method parameters
  - Local variables
- **Operand Stack** to support evaluation of expressions and evalutation of the method
  - Most JVM bytecodes manipulate the stack
- Reference to **Constant Pool** of current class

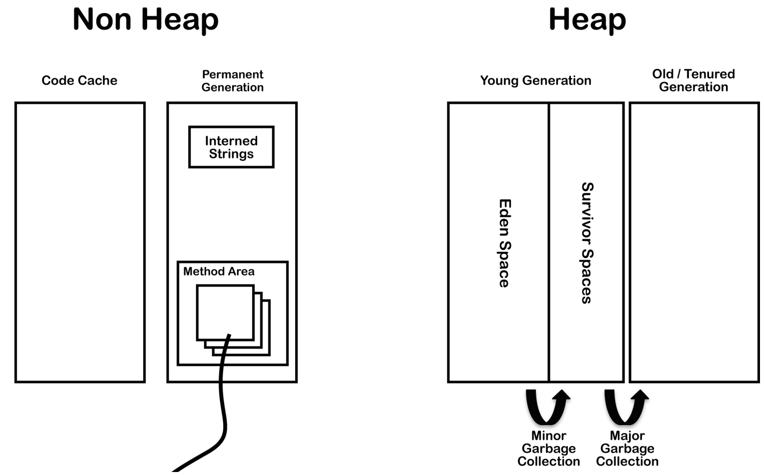
# Dynamic Linking (1)

- The reference to the constant pool for the current class helps to support **dynamic linking**.
- In C/C++ typically multiple object files are linked together to produce an executable or `dll`.
  - During the linking phase symbolic references are replaced with an actual memory address relative to the final executable.
- In Java this linking phase is done **dynamically** at runtime.
- When a Java class is compiled, all references to variables and methods **are stored in the class's constant pool as symbolic references**.

# Dynamic Linking (2)

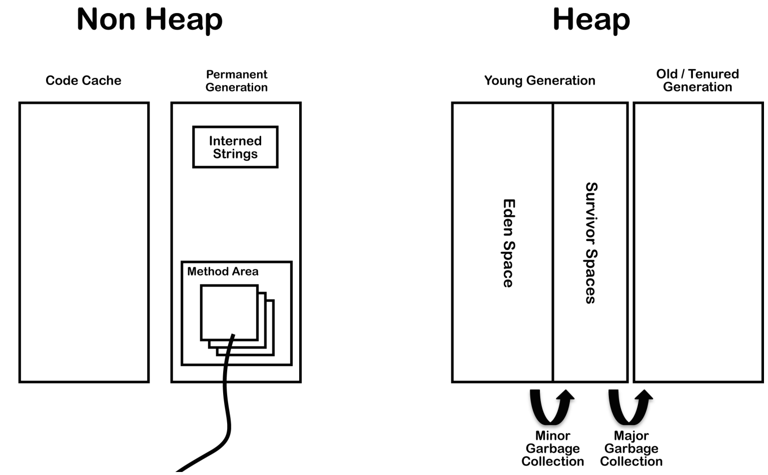
- The JVM implementation can choose when to resolve symbolic references.
  - **Eager or static resolution:** when the class file is verified after being loaded
  - **Lazy or late resolution:** when the symbolic reference is used for the first time
- The JVM **has to behave** as if the resolution occurred when each reference is first used and throw any resolution errors at this point.
- **Binding** is the process of the entity (field, method or class) identified by the symbolic reference being replaced by a direct reference
- This only happens once because the symbolic reference ***is completely replaced*** in the constant pool
- If the symbolic reference refers to a class that has not yet been resolved then this class will be loaded.

# Data Areas Shared by Threads: **Heap**




- Memory for objects and arrays; unlike C/C++ they are never allocated to stack
- Explicit deallocation not supported. Only by garbage collection.
- The HotSpot JVM includes four **Generational Garbage Collection Algorithms**

# Data Areas Shared by Threads: **Non-Heap**




- Memory for objects which are never deallocated, needed for the JVM execution
  - Method area
  - Interned strings
  - Code cache for JIT

# JIT compilation

- The Hotspot JVM (and other JVMs) profiles the code during interpretation, looking for "hot" areas of byte code that are executed regularly
- These parts are compiled to native code.
- Such code is then stored in the **code cache** in non-heap memory.

# Method area

The memory where `class` files are loaded. For each class:

- **Classloader Reference**
- From the `class` file:
  - **Run Time Constant Pool**
  - **Field data**
  - **Method data**
  - **Method code**

**Note:** Method area is shared among thread. Access to it has to be **thread safe**.

Changes of method area when:

- A new class is loaded
- A symbolic link is resolved by dynamic linking

# Class file structure

ClassFile {

| | |
|---|---|
| u4 magic; | **`0xCAFEBABE`** |
| u2 minor_version;<br>u2 major_version; | **Java Language Version** |
| u2 constant_pool_count;<br>cp_info contant_pool[constant_pool_count–1]; | **Constant Pool** |
| u2 access_flags; | **access modifiers and other info** |
| u2 this_class;<br>u2 super_class; | **References to Class and Superclass** |
| u2 interfaces_count;<br>u2 interfaces[interfaces_count]; | **References to Direct Interfaces** |
| u2 fields_count;<br>field_info fields[fields_count]; | **Static and Instance Variables** |
| u2 methods_count;<br>method_info methods[methods_count]; | **Methods** |
| u2 attributes_count;<br>attribute_info attributes[attributes_count]; | **Other Info on the Class** |

}

# Field data in the Method Area

Per field:

- Name
- Type
- Modifiers
- Attributes

# FieldType descriptors

| *FieldType* term | Type | Interpretation |
|---|---|---|
| `B` | `byte` | signed byte |
| `C` | `char` | Unicode character code point in the Basic Multilingual Plane, encoded with UTF-16 |
| `D` | `double` | double-precision floating-point value |
| `F` | `float` | single-precision floating-point value |
| `I` | `int` | integer |
| `J` | `long` | long integer |
| `L` *ClassName* `;` | `reference` | an instance of class *ClassName* |
| `S` | `short` | signed short |
| `Z` | `boolean` | `true` or `false` |
| `[` | `reference` | one array dimension |

# Method data

Per method:

- Name
- Return Type
- Parameter Types (in order)
- Modifiers
- Attributes

A *method descriptor* contains

- a sequence of zero or more *parameter descriptors* in brackets
- a *return descriptor* or V for *void descriptor*

Example: The descriptor of

```
Object m(int i, double d, Thread t) {...}
```

is:

```
(IDLjava/lang/Thread;)Ljava/lang/Object;
```