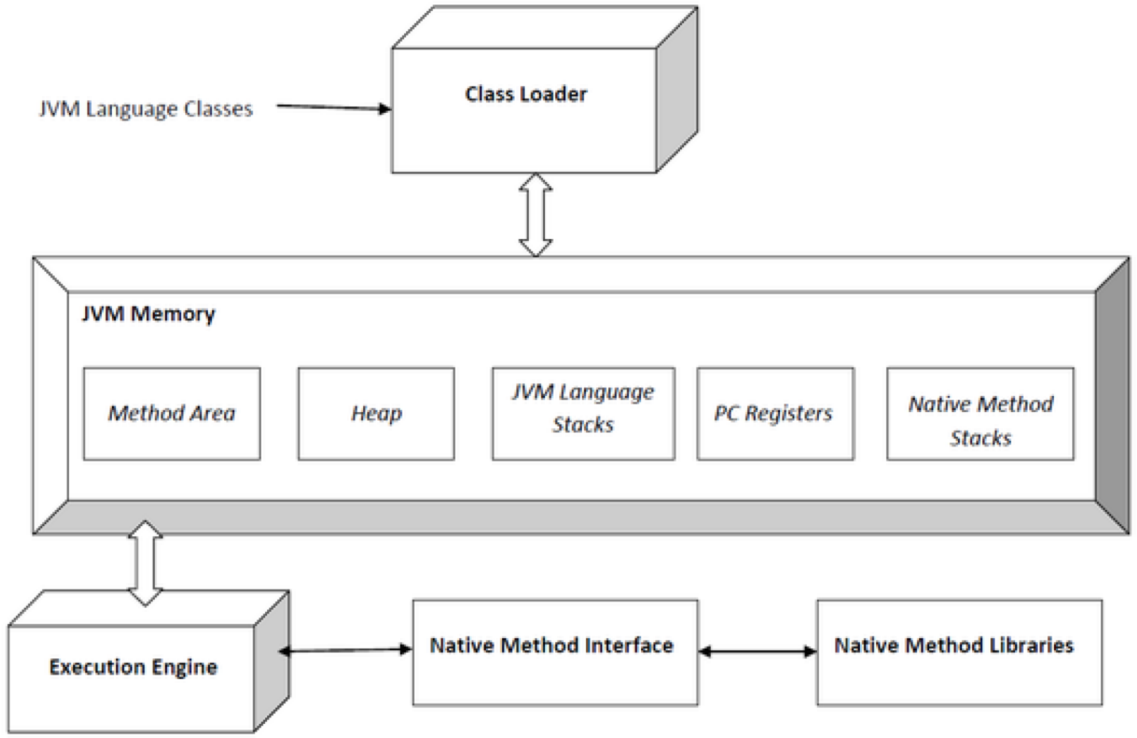
Architecture of the Java Virtual Machine



Linking In The Java Virtual Machine

- Java's approach to linking is completely dynamic.
- Individual `.class` files are loaded, linked, and initialized when they are first referenced.
  - Allows for greater flexibility and portability for the user.
  - Makes the necessary run-time data structures and algorithms more elaborate.

Linking In The Java Virtual Machine

- Java's approach to linking is completely dynamic.
- Individual `.class` files are loaded, linked, and initialized when they are first referenced.
  - Allows for greater flexibility and portability for the user.
  - Makes the necessary run-time data structures and algorithms more elaborate.

```
% java HelloWorld "Hello!" "Hi!"

public static void HelloWorld.main(args)   args.length = 2
                                            args[0] = "Hello!"
                                            args[1] = "Hi!"
```

Linking In The Java Virtual Machine

- Java's approach to linking is completely dynamic.
- Individual `.class` files are loaded, linked, and initialized when they are first referenced.
  - Allows for greater flexibility and portability for the user.
  - Makes the necessary run-time data structures and algorithms more elaborate.

```
% java HelloWorld "Hello!" "Hi!"
```

```
public static void HelloWorld.main(args)   args.length = 2
                                            args[0] = "Hello!"
                                            args[1] = "Hi!"
```

- Game plan
  - The `class` file format.
  - JVM startup and exit actions.
  - Class loading, linking, initialization (simplified: fast path only).

The `class` File Format

- [Ref: JVM SE16, Chapter 4]
- A `class` file consists of a stream of 8-bit bytes.
- 16-bit and 32-bit quantities are constructed from two or four 8-bit bytes.
- Multibyte data items are always stored in big-endian order.
- Shorthand
  - `u1` for unsigned one-byte quantity.
  - `u2` for unsigned two-byte quantity.
  - `u4` for unsigned four-byte quantity.
- A class file consists of a single `ClassFile` structure.
  - Essentially, a sequence of self-describing nested tables.

The `class` File Format

```
ClassFile {
  u4 magic;                          // 0xCAFEBABE
  u2 minor_version;                  // Minor version number, in {0, 65535}
  u2 major_version;                  // Major version number, in [45,60]

  // String constants, class and interface names, field names, etc.
  u2 constant_pool_count;
  cp_info constant_pool[constant_pool_count-1];

  u2 access_flags;
  u2 this_class;                     // index into constant_pool
  u2 super_class;                    // index into constant_pool

  u2 interfaces_count;
  u2 interfaces[interfaces_count]; // indices into constant_pool

  // Complete description of a field
  u2 fields_count;
  field_info fields[fields_count];

  // Complete description of a method
  u2 methods_count;
  method_info methods[methods_count];

  // Information used both for JVM function and for tools
  u2 attributes_count;
  attribute_info attributes[attributes_count];
}
```

Constants, Fields, and Methods

- Constant pool
  - The central repository of names found in the class.
  - Consists of a number of variable-sized (but self-describing) entries of type `cp_info`, describing 17 kinds of possible constants.

```
cp_info {
  u1 tag;
  u1 info[];
}
```

Constants, Fields, and Methods

- Constant pool
  - The central repository of names found in the class.
  - Consists of a number of variable-sized (but self-describing) entries of type `cp_info`, describing 17 kinds of possible constants.
- Fields
  - Described by a `field_info` structure.
  - Variable-sized, but self-describing.

```
cp_info {
  u1 tag;
  u1 info[];
}
```

```
field_info {
  u2 access_flags;
  u2 name_index;
  u2 descriptor_index;
  u2 attributes_count;
  attribute_info
    attributes[attributes_count];
}
```

Constants, Fields, and Methods

- Constant pool
  - The central repository of names found in the class.
  - Consists of a number of variable-sized (but self-describing) entries of type `cp_info`, describing 17 kinds of possible constants.
- Fields
  - Described by a `field_info` structure.
  - Variable-sized, but self-describing.
- Methods
  - Described by a `method_info` structure.
  - Variable-sized, but self-describing.
  - A key attribute is `Code`, which contains the JVM code for the method.

```
cp_info {
  u1 tag;
  u1 info[];
}
```

```
field_info {
  u2 access_flags;
  u2 name_index;
  u2 descriptor_index;
  u2 attributes_count;
  attribute_info
    attributes[attributes_count];
}
```

```
method_info {
  u2 access_flags;
  u2 name_index;
  u2 descriptor_index;
  u2 attributes_count;
  attribute_info
    attributes[attributes_count];
}
```

Java Virtual Machine Startup and Exit

- [Ref: JVM SE16, §5.2, §5.7]
- JVM startup
  - Create an initial class (or interface) using a class loader.
  - Link this initial class.
  - Initialize this initial class.
  - Invoke the `public static` method `void main(String[])` of this class.

Java Virtual Machine Startup and Exit

- [Ref: JVM SE16, §5.2, §5.7]
- JVM startup
  - Create an initial class (or interface) using a class loader.
  - Link this initial class.
  - Initialize this initial class.
  - Invoke the `public static` method `void main(String[])` of this class.
- The invocation of this method drives all further execution, including the linking and creation of additional classes and interfaces, and the invocation of additional methods.

Java Virtual Machine Startup and Exit

- [Ref: JVM SE16, §5.2, §5.7]
- JVM startup
  - Create an initial class (or interface) using a class loader.
  - Link this initial class.
  - Initialize this initial class.
  - Invoke the `public static` method `void main(String[])` of this class.
- The invocation of this method drives all further execution, including the linking and creation of additional classes and interfaces, and the invocation of additional methods.
- JVM exit
  - The Java Virtual Machine exits when some thread invokes the `exit` method of class `Runtime` or class `System`, or the `halt` method of class `Runtime`, and the `exit` or `halt` operation is permitted by the security manager.

Run-Time Constant Pool

- [Ref: JVM SE16, §5.1]
- The JVM maintains a run-time constant pool for each class (or interface).
  - Plays a role analogous to that of a symbol table.
  - Constructed upon class creation from the `constant_pool` in the `class` file representation of the class.

Run-Time Constant Pool

- [Ref: JVM SE16, §5.1]
- The JVM maintains a run-time constant pool for each class (or interface).
  - Plays a role analogous to that of a symbol table.
  - Constructed upon class creation from the `constant_pool` in the `class` file representation of the class.
- Two kinds of entries:
  - Symbolic references, which may later be resolved.
  - Static constants (i.e., string and numeric constants), which require no further processing.

Run-Time Constant Pool

- [Ref: JVM SE16, §5.1]
- The JVM maintains a run-time constant pool for each class (or interface).
  - Plays a role analogous to that of a symbol table.
  - Constructed upon class creation from the `constant_pool` in the `class` file representation of the class.
- Two kinds of entries:
  - Symbolic references, which may later be resolved.
  - Static constants (i.e., string and numeric constants), which require no further processing.
- All static constants and some types of symbolic references are loadable, i.e., they may be pushed onto the evaluation stack by the `ldc` family of instructions.

Class Creation and Loading

- [Ref: JVM SE16, §5.3]
- Creation of a class (or interface) C denoted by the name N consists of the construction in the method area of the JVM of an implementation-specific internal representation of C.
  - A non-array class is created by loading its binary representation using a class loader L.
  - The run-time identity of class is the pair $\langle N, L \rangle$.

Class Creation and Loading

- [Ref: JVM SE16, §5.3]
- Creation of a class (or interface) C denoted by the name N consists of the construction in the method area of the JVM of an implementation-specific internal representation of C.
  - A non-array class is created by loading its binary representation using a class loader L.
  - The run-time identity of class is the pair $\langle N, L \rangle$.
- Loading a class (when L is the bootstrap class loader)
  - The JVM passes the argument N to an invocation of a method on the bootstrap class loader to search for a representation of C in a platform-dependent manner (i.e., a `class` file).
  - Then the JVM attempts to derive a class C from the `class` file.
    - Parse the `class` file into an implementation-specific internal representation.
    - If C has a direct superclass D, resolve the symbolic reference from C to D. Likewise for any superinterfaces.

Class Linking

- [Ref: JVM SE16, §5.4]
- Linking a class (or interface) involves verifying and preparing that class, its direct superclass, and its direct superinterfaces, if necessary.
- Linking also involves resolution of symbolic references in the class or interface, though not necessarily at the same time as the class or interface is verified and prepared.
- The JVM specification allow an implementation to be flexible in scheduling linking activities “eagerly” or “lazily”, as long as the following invariants are maintained.
  - A class is completely loaded before it is linked.
  - A class is completely verified and prepared before it is initialized.

Verification, Preparation, Resolution

- [Ref: JVM SE16, §5.4.1-3]
- Verification ensures that the binary representation of a class or interface is structurally correct.

Verification, Preparation, Resolution

- [Ref: JVM SE16, §5.4.1-3]
- Verification ensures that the binary representation of a class or interface is structurally correct.
- Preparation involves creating the static fields for a class and initializing such fields to their default values.
  - This does not require the execution of any JVM code; explicit initializers for static fields are executed as part of initialization, not preparation.
  - Preparation may occur at any time following creation but must be completed prior to initialization.

Verification, Preparation, Resolution

- [Ref: JVM SE16, §5.4.1-3]
- Verification ensures that the binary representation of a class or interface is structurally correct.
- Preparation involves creating the static fields for a class and initializing such fields to their default values.
  - This does not require the execution of any JVM code; explicit initializers for static fields are executed as part of initialization, not preparation.
  - Preparation may occur at any time following creation but must be completed prior to initialization.
- Resolution is the process of dynamically determining one or more concrete values from a symbolic reference in the run-time constant pool.
  - Required when a JVM instruction like `new`, `invokevirtual`, or `getfield` is executed.
  - Initially, all symbolic references in the run-time constant pool are unresolved.

Class Initialization

- [Ref: JVM SE16, §5.5]
- Initialization of a class consists of executing its class initialization method (i.e., `<clinit>`).
  - Prior to initialization, a class must be linked, i.e., verified, prepared, and optionally resolved.
- Details of the initialization process are complicated.
  - Because the JVM is multithreaded, initialization of a class requires careful synchronization, since some other thread may be trying to initialize the same class at the same time.
  - There is also the possibility that initialization of a class may be requested recursively as part of the initialization of that class.
  - The implementation of the JVM is responsible for taking care of synchronization and recursive initialization following a specified procedure.