301AA - Advanced Programming

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

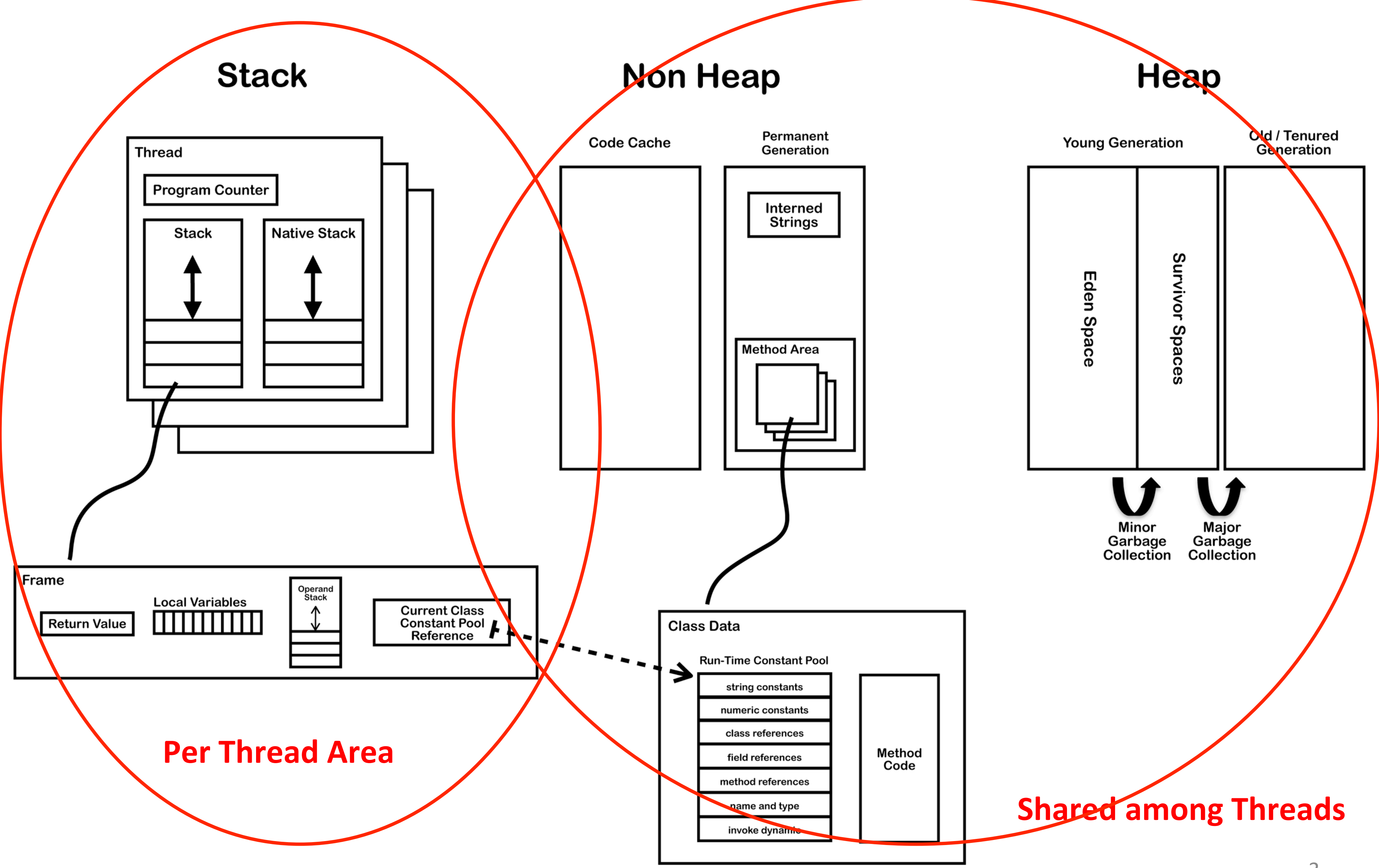
Course pages:
http://pages.di.unipi.it/corradini/Didattica/AP-18/

AP-2018-05: *The Java Virtual Machine (cont.)*

Overview

- JVM Runtime Data Areas: The Method Area
- Class File Structure
- Field and Method Data
- Method Code
- Disassembling Class Files
- The Constant Pool
- Loading, Linking and Initializing
- Class Loaders
- The Verification Process
- Initialization and Finalization
- The JVM Exit

JVM Runtime Data Areas



Class file structure

ClassFile {

u4 magic;	**0xCAFEBABE**

u2 minor_version;
u2 major_version;
	Java Language Version

u2 constant_pool_count;
cp_info contant_pool[constant_pool_count–1];
	Constant Pool

u2 access_flags;	**access modifiers and other info**

u2 this_class;
u2 super_class;
	References to Class and Superclass

u2 interfaces_count;
u2 interfaces[interfaces_count];
	References to Direct Interfaces

u2 fields_count;
field_info fields[fields_count];
	Static and Instance Variables

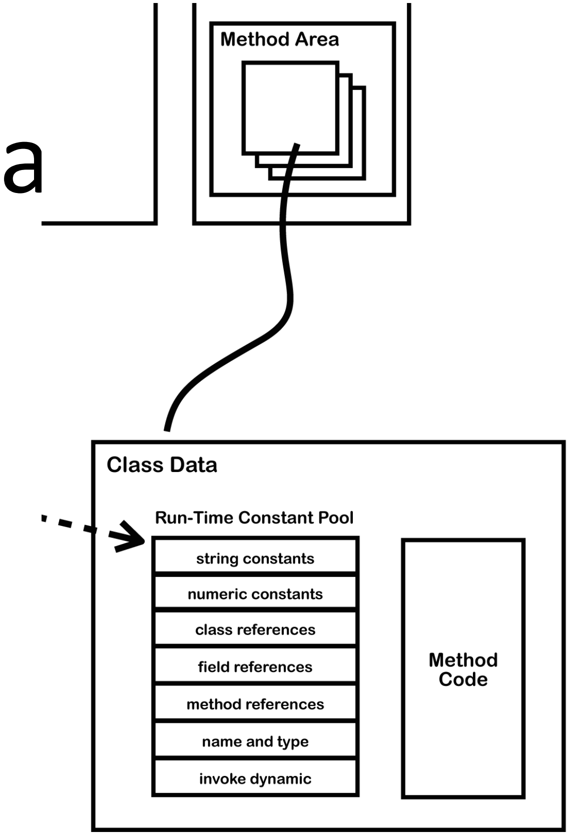
u2 methods_count;
method_info methods[methods_count];
	Methods

u2 attributes_count;
attribute_info attributes[attributes_count];
	Other Info on the Class

}

Non-Heap: the Method Area

- **Classloader Reference**
- From the `class` file:
  - **Run Time Constant Pool**
  - **Field data**
    - Name
    - Type
    - Modifiers
    - Attributes
  - **Method data**
    - Name
    - Return Type
    - Parameter Types (in order)
    - Modifiers
    - Attributes
  - **Method code…**



Method code

Per method:

- Bytecodes
- Operand stack size
- Local variable size
- Local variable table
- Exception table
- LineNumberTable – which line of source code corresponds to which byte code instruction (for debugger)

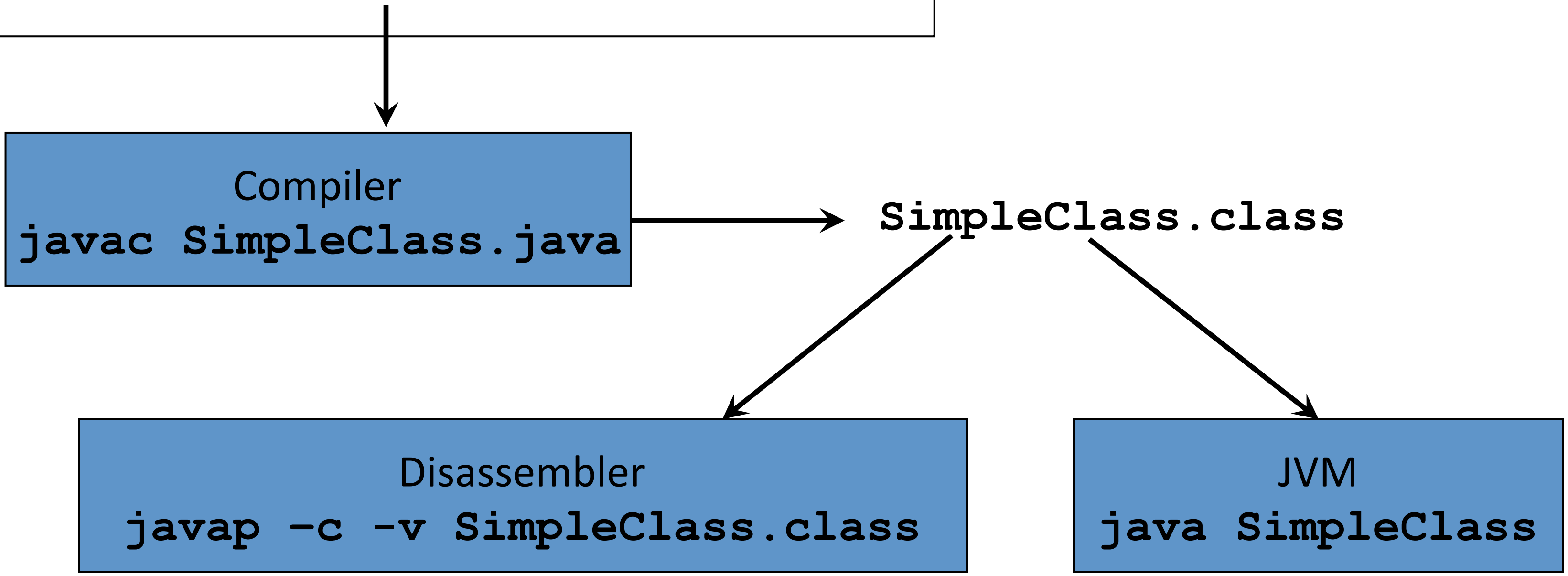
Per exception handler (one for each try/catch/finally clause)

- Start point
- End point
- PC offset for handler code
- Constant pool index for exception class being caught

Disassembling Java files: javac, javap, java

SimpleClass.java

```
package org.jvminternals;
public class SimpleClass {
    public void sayHello() {
        System.out.println("Hello");
    }
}
```



Compiler

`javac SimpleClass.java`

`SimpleClass.class`

Disassembler

`javap -c -v SimpleClass.class`

JVM

`java SimpleClass`

SimpleClass.class: constructor and method

```
package org.jvminternals;
public class SimpleClass {
    public void sayHello() {
        System.out.println("Hello");
}}
```

Local variable 0 = “this”

```
{
  public org.jvminternals.SimpleClass();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1    // Method java/lang/Object."<init>":()V
         4: return
      LineNumberTable:
        line 2: 0

  public void sayHello();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=2, locals=1, args_size=1
         0: getstatic     #2    // Field java/lang/System.out:Ljava/io/PrintStream;
         3: ldc           #3    // String Hello
         5: invokevirtual #4    // Method java/io/PrintStream.println:(Ljava/lang/
String;)V
         8: return
      LineNumberTable:
        line 4: 0
        line 5: 8
}
SourceFile: "SimpleClass.java”
```

Method descriptors

Index into constant pool

String literal

Field descriptor

The constant pool

- Similar to *symbol table,* but with more info
- Contains constants and symbolic references used for dynamic binding, suitably tagged
  - numeric literals (Integer, Float, Long, Double)
  - string literals (Utf8)
  - class references (Class)
  - field references (Fieldref)
  - method references (Mehodref, InterfaceMethodref, MethodHandle)
  - signatures (NameAndType)
- Operands in bytecodes often are indexes in the constant pool

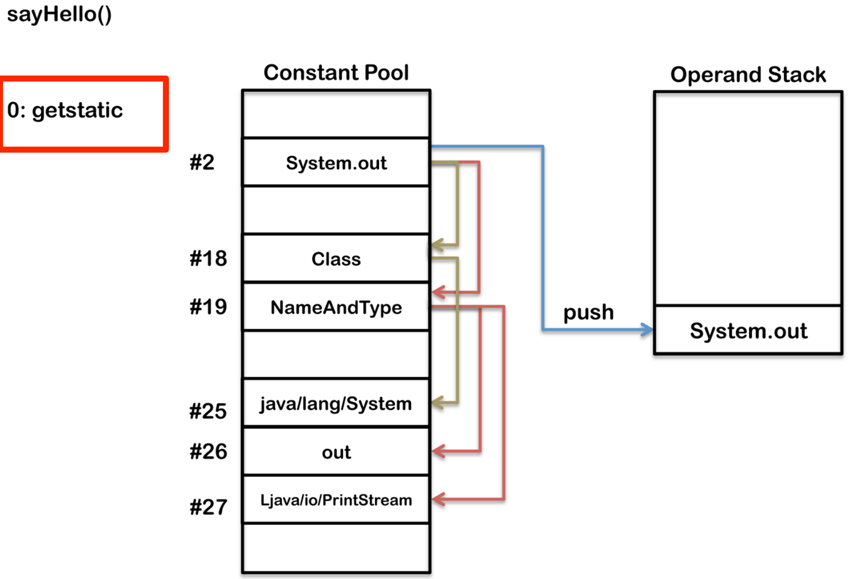
SimpleClass.class: the Constant pool

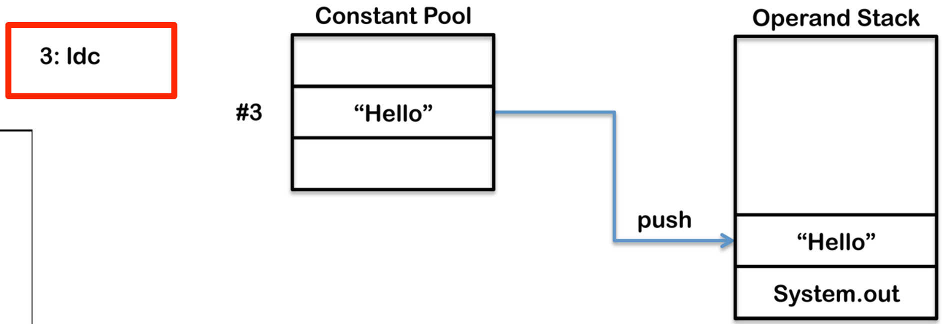
```
public class SimpleClass {
    public void sayHello() {
        System.out.println("Hello");
}}
```

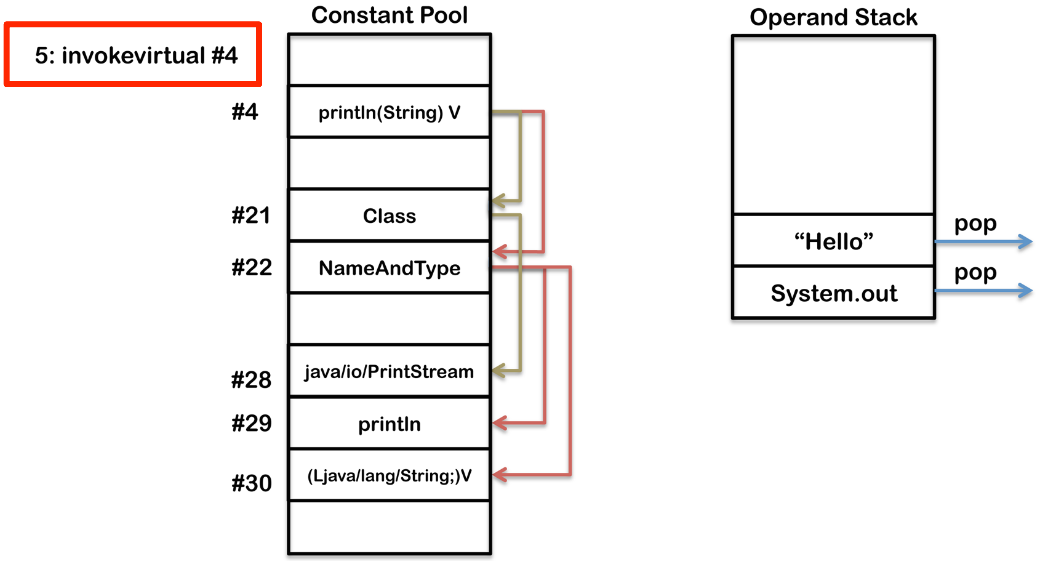
```
Compiled from "SimpleClass.java"
public class SimpleClass
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Methodref          #6.#14         // java/lang/Object."<init>":()V
   #2 = Fieldref           #15.#16        // java/lang/System.out:Ljava/io/PrintStream;
   #3 = String             #17            // Hello
   #4 = Methodref          #18.#19        // java/io/PrintStream.println:(Ljava/lang/String;)V
   #5 = Class              #20            // SimpleClass
   #6 = Class              #21            // java/lang/Object
   #7 = Utf8               <init>
   #8 = Utf8               ()V
   #9 = Utf8               Code
  #10 = Utf8               LineNumberTable
  #11 = Utf8               sayHello
  #12 = Utf8               SourceFile
  #13 = Utf8               SimpleClass.java
  #14 = NameAndType        #7:#8          // "<init>":()V
  #15 = Class              #22            // java/lang/System
  #16 = NameAndType        #23:#24        // out:Ljava/io/PrintStream;
  #17 = Utf8               Hello
  #18 = Class              #25            // java/io/PrintStream
  #19 = NameAndType        #26:#27        // println:(Ljava/lang/String;)V
  #20 = Utf8               SimpleClass
  #21 = Utf8               java/lang/Object
  #22 = Utf8               java/lang/System
  #23 = Utf8               out
  #24 = Utf8               Ljava/io/PrintStream;
  #25 = Utf8               java/io/PrintStream
  #26 = Utf8               println
  #27 = Utf8               (Ljava/lang/String;)V
```

```
public void sayHello();
   descriptor: ()V
   Code:
     stack=2, locals=1, args_size=1
        0: getstatic     #2
        3: ldc           #3
        5: invokevirtual #4
        8: return
```

```
public void sayHello();
   descriptor: ()V
   Code:
     stack=2, locals=1, args_size=1
        0: getstatic     #2
        3: ldc           #3
        5: invokevirtual #4
        8: return
```







Loading, Linking, and Initializing

- **Loading**: finding the binary representation of a class or interface type with a given name and creating a class or interface from it
- **Linking**: taking a class or interface and combining it into the run-time state of the Java Virtual Machine so that it can be executed
- **Initialization**: executing the class or interface initialization method `<clinit>`

JVM Startup

- The JVM starts up by loading an initial class using the **bootstrap classloader**
- The class is linked and initialized
- **`public static void main(String[])`** is invoked.
- This will trigger loading, linking and initialization of additional classes and interfaces…

Loading

- Class or Interface *C* creation is triggered
  - by other class or interface referencing *C*
  - by certain methods (eg. reflection)
- Array classes are generated by the JVM
- Check whether already loaded
- If not, invoke the appropriate loader.loadClass
- Each class is tagged with the *initiating loader*
- *Loading constraints* are checked during loading
  - to ensure that the same name denotes the same type in different loaders

Class Loader Hierarchy

- **Bootstrap Classloader** loads basic Java APIs, including for example `rt.jar`. It may skip much of the validation that gets done for normal classes.
- **Extension Classloader** loads classes from standard Java extension APIs such as security extension functions.
- **System Classloader** is the default application classloader, which loads application classes from the classpath
- **User Defined Classloaders** can be used to load application classes:
  - for runtime reloading of classes
  - for loading from different sources, eg. from network
  - for supporting separation between different groups of loaded classes as required by web servers
- Class loader hooks: `findClass` (builds a byte array), `defineClass` (turns an array of bytes into a class object), `resolveClass` (links a class)

Runtime Constant Pool

- The constant_pool table in the `.class` file is used to construct the *run-time constant pool* upon class or interface creation.
- All references in the run-time constant pool are initially symbolic.
- Symbolic references are derived from the`.class` file in the expected way
- Class names are those returned by **`Class.getName()`**
- Field and method references are made of name, descriptor and class name

Linking

- Link = verification, preparation, resolution
- **Verification**: see below
- **Preparation**: allocation of storage (method tables)
- **Resolution** (optional): resolve symbol references by loading referred classes/ interfaces
  - Otherwise postponed till first use by an instruction

Verification

- When?
  - Mainly during the load and link process
- Why?
  - No guarantee that the class file was generated by a Java compiler
  - Enhance runtime performance
- Examples
  - There are no operand stack overflows or underflows.
  - All local variable uses and stores are valid.
  - The arguments to all the JVM instructions are of valid types.

Verification Process

- Pass 1 – when the class file is loaded
  - The file is properly formatted, and all its data is recognized by the JVM
- Pass 2 – when the class file is linked
  - All checks that do not involve instructions
    - `final` classes are not subclassed, `final` methods are not overridden.
    - Every class (except `Object`) has a superclass.
    - All field references and method references in the constant pool have valid names, valid classes, and a valid type descriptor.

Verification Process – cont.

- Pass 3 – still during linking
  - **Data-flow analysis** on each method.
  - Ensure that at any given point in the program, no matter what code path is taken to reach that point:
    - The operand stack is always the same size and contains the same types of objects.
    - No local variable is accessed unless it is known to contain a value of an appropriate type.
    - Methods are invoked with the appropriate arguments.
    - Fields are assigned only using values of appropriate types.
    - All opcodes have appropriate type arguments on the operand stack and in the local variables
    - A method must not throw more exceptions than it admits
    - A method must end with a return value or throw instruction
    - Method must not use one half of a two word value

Verification Process – cont.

- Pass 4 - the first time a method is actually invoked
  - a virtual pass whose checking is done by JVM instructions
    - The referenced method or field exists in the given class.
    - The currently executing method has access to the referenced method or field.
  - Each cell has one, and only one type
    - Primitive / reference.

Initialization

- <clinit> initialization method is invoked on classes and interfaces to initialize class variables
- static initializers are executed
- direct superclass need to be initialized prior
- happens on direct use: method invocation, construction, field access
- synchronized initializations: state in Class object
- <init>: initialization method for instances
  - **invokespecial** instruction
  - can be invoked only on uninitialized instances

Initialization example (1)

```
class Super {
    static { System.out.print("Super ");}
}
class One {
    static { System.out.print("One ");}
}
class Two extends Super {
    static { System.out.print("Two ");}
}
class Test {
  public static void main(String[] args) {
   One o = null;
   Two t = new Two();
   System.out.println((Object)o == (Object)t);
  }
}
```

What doers **`java Test`** print?

Super Two False

Initialization example (2)

```
class Super { static int taxi = 1729;}
}
class Sub extends Super {
    static { System.out.print("Sub ");}
}
class Test {
    public static void main(String[] args) {
   System.out.println(Sub.taxi);
}}
```

What does **`java Test`** print?

Only prints "1729"

A reference to a static field (§8.3.1.1) causes initialization of only the class or interface that actually declares it, even though it might be referred to through the name of a subclass, a subinterface, or a class that implements an interface. (page 385 of [JLS-8])

Finalization

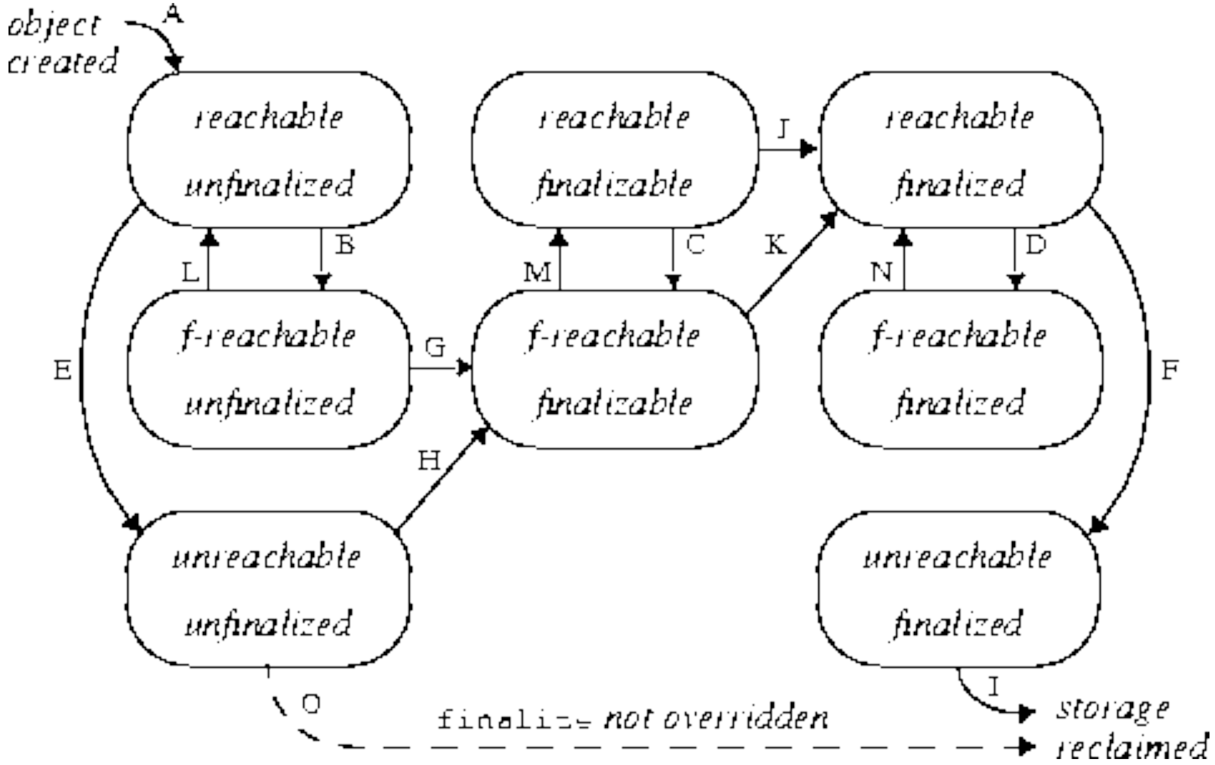
- Invoked just before garbage collection
- JLS does not specify when it is invoked
- Also does not specify which thread
- No automatic invocation of super's finalizers
- Very tricky!

```
void finalize() {
      classVariable = this; // the object is reachable again
}
```

- Each object can be
  - Reachable, finalizer-reachable, unreachable
  - Unfinalized, finalizable, finalized

Finalization State Diagram

`https://notendur.hi.is/snorri/SDK-docs/lang/lang083.htm`



`finalize()` is never called a second time on the same object, but it can be invoked as any other method!

JVM Exit

- `classFinalize` similar to object finalization
- A class can be unloaded when
  - no instances exist
  - class object is unreachable
- JVM exits when:
  - all its non-daemon threads terminate
  - `Runtime.exit` or `System.exit` assuming it is secure
- finalizers can be optionally invoked on all objects just before exit