WTF is [Ljava/lang/String; ?!?

A deep dive into the shallow end of the JVM

A Word From Our Sponsor

The Compiler (javac)

Translates Java source code into .class files

- Requires access to all referenced classes

OpenJDK (Sun/Oracle) version is intentionally simple, relies on Hotspot for optimization

The Classfile

Every class has its own .class file

- Including nested/inner classes

Contains compiled bytecode, along with metadata

- Method signatures, field definitions
- Names for all referenced classes/methods
- Debugging information

Classloading

Classes are loaded by a ClassLoader

- Classloaders form a hierarchy
- Files loaded by different loaders are different classes

Classes are loaded as needed

- Can be slow if network involved
- “Commonly used” classes are preloaded

Classloading, continued

Each class is verified as it’s loaded

- Bytecode is valid
- No invalid memory accesses
- No attempt to override access control

After verification, static initializers run

- Can trigger loading of additional classes

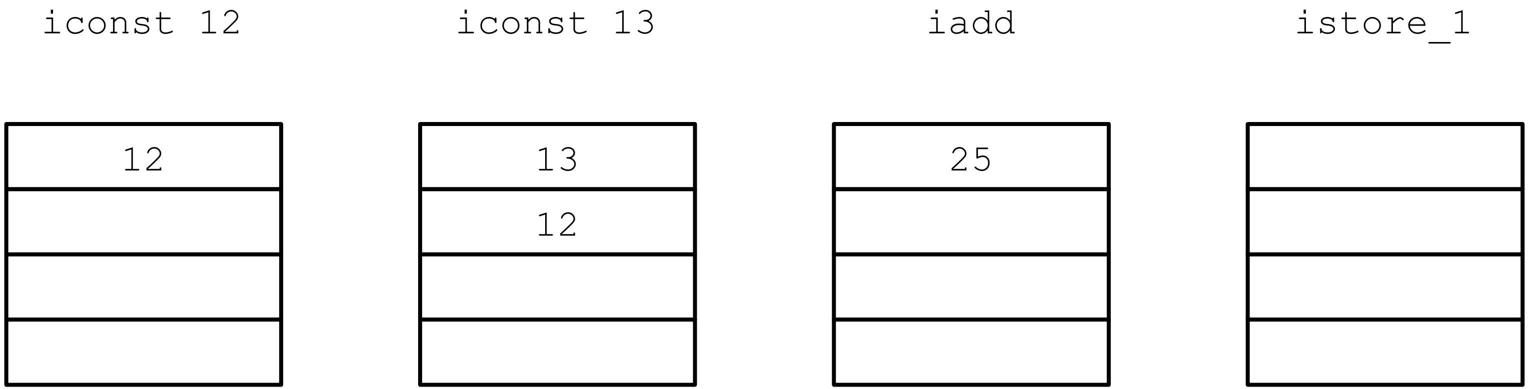
The JVM

A RISC emulator running on a CISC processor

- Stack-based
- Limited data types
- Each operation specified by 1-byte code

Supported operations driven by Java language

A Stack-Based Processor



iconst 12

12

iconst 13

13

12

iadd

25

istore_1

Two Types of Stacks

Operand Stack

- Values for arithmetic operations
- References for method invocations

Call Stack

- Local Variables and Method Parameters
- 32-bit-wide slots, numbered 0 .. N
- Instance methods put reference to object in slot 0

Limited Data Types

Each “slot” in stack is 32-bits wide

Fully supported: `int`, `long`, `float`, `double`

Promoted: `byte`, `short`, `char`, `boolean`

Arrays stored at “native” size

Object field size implementation-dependent

Types of JVM Operations

Load/store local variable

Load/store field (static or instance)

Arithmetic

Test/Branch

New

Monitor entry/exit (synchronization)

Throw

Types of Method Invocations

Static

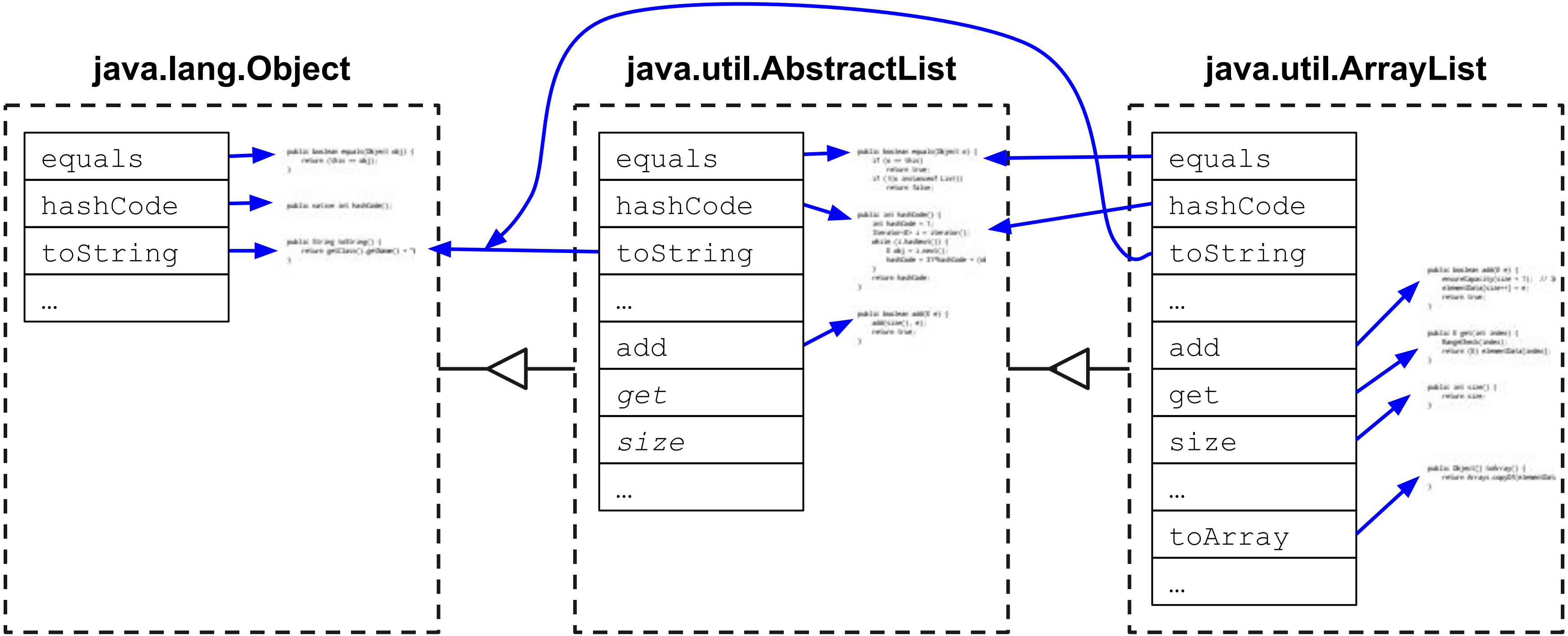
Special (private, constructor, super)

Virtual (protected, package, public)

Interface

Dynamic

Virtual Method Dispatch



java.lang.Object

- equals
- hashCode
- toString
- …

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

```
public native int hashCode();
```

```
public String toString() {
    return getClass().getName() + "
}
```

java.util.AbstractList

- equals
- hashCode
- toString
- …
- add
- *get*
- *size*
- …

```
public boolean equals(Object o) {
    if (o == this)
        return true;
    if (!(o instanceof List))
        return false;
```

```
public int hashCode() {
    int hashCode = 1;
    Iterator<E> i = iterator();
    while (i.hasNext()) {
        E obj = i.next();
        hashCode = 31*hashCode + (ob
    }
    return hashCode;
}
```

```
public boolean add(E e) {
    add(size(), e);
    return true;
}
```

java.util.ArrayList

- equals
- hashCode
- toString
- …
- add
- get
- size
- …
- toArray
- …

```
public boolean add(E e) {
    ensureCapacity(size + 1);  // In
    elementData[size++] = e;
    return true;
}
```

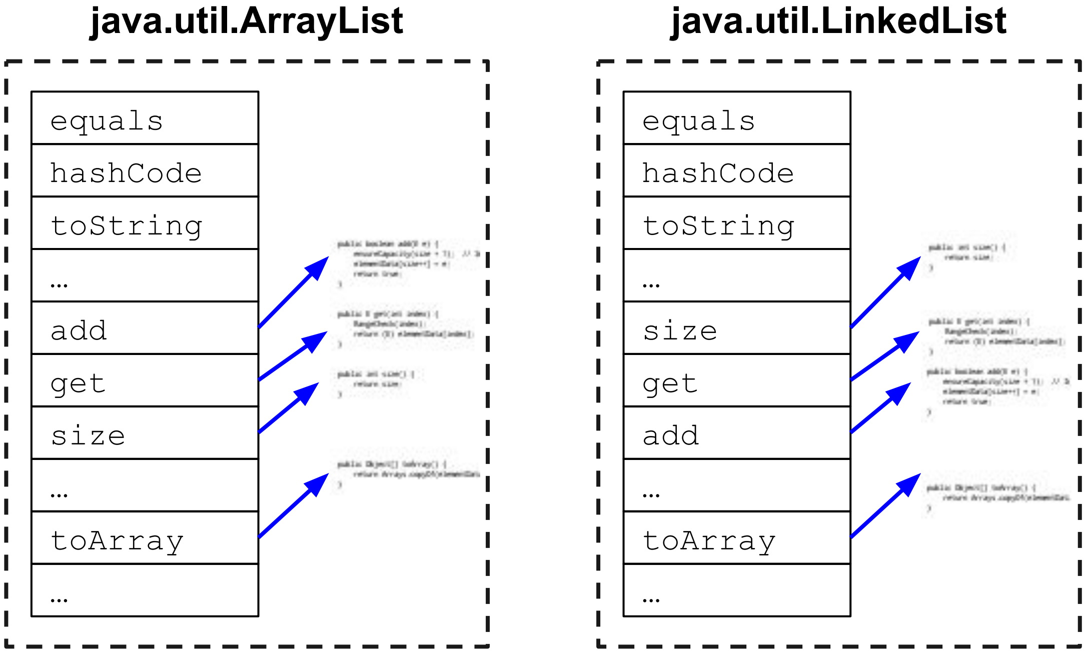
```
public E get(int index) {
    RangeCheck(index);
    return (E) elementData[index];
}
```

```
public int size() {
    return size;
}
```

```
public Object[] toArray() {
    return Arrays.copyOf(elementDat
}
```

Interface Dispatch

```
List<String> myList = // …
String first = myList.get(0)
```



java.util.ArrayList

```
equals
hashCode
toString
…
add
get
size
…
toArray
…
```

```
public boolean add(E e) {
    ensureCapacity(size + 1);  // In
    elementData[size++] = e;
    return true;
}
```

```
public E get(int index) {
    RangeCheck(index);
    return (E) elementData[index];
}
```

```
public int size() {
    return size;
}
```

```
public Object[] toArray() {
    return Arrays.copyOf(elementData
}
```

java.util.LinkedList

```
equals
hashCode
toString
…
size
get
add
…
toArray
…
```

```
public int size() {
    return size;
}
```

```
public E get(int index) {
    RangeCheck(index);
    return (E) elementData[index];
}
```

```
public boolean add(E e) {
    ensureCapacity(size + 1);  // In
    elementData[size++] = e;
    return true;
}
```

```
public Object[] toArray() {
    return Arrays.copyOf(elementData
}
```

Example: Java

```
public static void main(String[] argv)
{
    for (int ii = 1 ; ii < 10 ; ii += 2)
    {
        System.out.println(ii);
    }
}
```

Example: Bytecode

```
public static void main(java.lang.String[]);
  Code:
   0:    iconst_1
   1:    istore_1
   2:    goto    15
   5:    getstatic    #16; //Field java/lang/System.out:Ljava/io/PrintStream;
   8:    iload_1
   9:    invokevirtual    #22; //Method java/io/PrintStream.println:(I)V
   12:    iinc    1, 2
   15:    iload_1
   16:    bipush    10
   18:    if_icmplt    5
   21:    return
}
```

Hotspot

Runtime optimizer for frequently-called code

- Replace interpreted code by native
- “Traditional” compiler optimizations
- Function inlining
- Replace interface invocation if only one impl

General JVM Performance Tweaks

- Heap management
- Intrinsics
- ...

Watching Hotspot at Work

-XX:+PrintCompilation

- Writes console messages as functions compiled

-XX:+PrintInlining

- Writes console messages as functions inlined
- Requires `-XX:+UnlockDiagnosticVMOptions`

-XX:+PrintAssembly

- Writes generated machine code
- Requires `-XX:+UnlockDiagnosticVMOptions`
- Requires disassembler agent

Myths and Misconceptions

And maybe a few uncomfortable truths

Java is Slow!

Until Hotspot kicks in, JVM is an interpreter

- And even Hotspot can’t match hand-tuned libraries

Startup loads lots of classes

- Don’t use Spring for a command-line filter app

GC can create inconvenient pauses

Java Uses Too Much Memory!

Don’t confuse virtual and resident memory

- JVM will reserve max heap from OS
- OS will assign physical memory as needed

Memory is under $15/Gb

But that isn’t a license to go wild

- Large heaps == lots of garbage when collector runs
- Over-committing can lead to big problems

We Need Obfuscation!

Simple Bytecode + Symbolic Names
= Easy to Decompile

- Java stores method/variable names in classfile, unlike “compiled” languages
- Obfuscators work by changing names
- Are names really the barrier to understanding?

We Need Obfuscation!

Simple Bytecode + Symbolic Names
= Easy to Decompile?

- Java stores method/variable names in classfile, unlike “compiled” languages
- Obfuscators work by changing names
- Are names really the barrier to understanding?

If you still want to obfuscate, use Scala

Always use StringBuilder!

```
public String concat1(
    String s1,
    String s2,
    String s3)
{
    return s1 + s2 + s3;
}
```

```
public String concat2(
    String s1,
    String s2,
    String s3)
{
    StringBuilder sb
        = new StringBuilder();
    sb.append(s1);
    sb.append(s2);
    sb.append(s3);
    return sb.toString();
}
```

The JVM Can’t Do Tail Recursion!

Definition:
tail call is last call in method

Optimization:
replace call by jump

```
public int foo(int x)
{
    // do something
    return bar(y);
}
```

tail call

```
public int bar(int x)
{
    // do something
    return bar(y);
}
```

tail-call recursion

Of course it can!

You just need `goto` and static analysis

- Scala supports tail-recursive methods

The JVM does apply some constraints

- `goto` is limited to intra-method jumps
- Can’t combine methods from different classes

Hotspot doesn’t need to play by the rules

For More Information

Generating bytecode listings

- `javap -c FULLY.QUALIFIED.CLASSNAME`

List undocumented JVM options

- `java -XX:+UnlockDiagnosticVMOptions -XX:+PrintFlagsFinal`

JVM Spec

- `http://docs.oracle.com/javase/specs/jvms/se7/html/index.html`

Hotspot Internals Wiki

- `https://wikis.oracle.com/display/HotSpotInternals/Home`