Squawk Technology

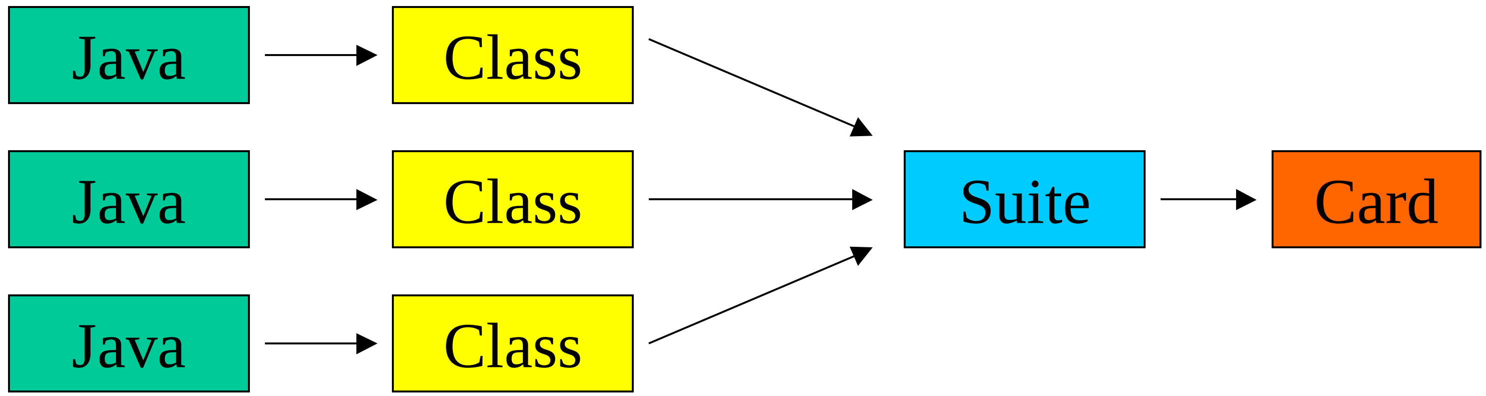
Nik Shaylor
Doug Simon
Bill Bush

Overview

- Application architecture
- Translation
- Semantics of Execution
- Memory model
- Interpreter core engineering
- Native code
- The demo

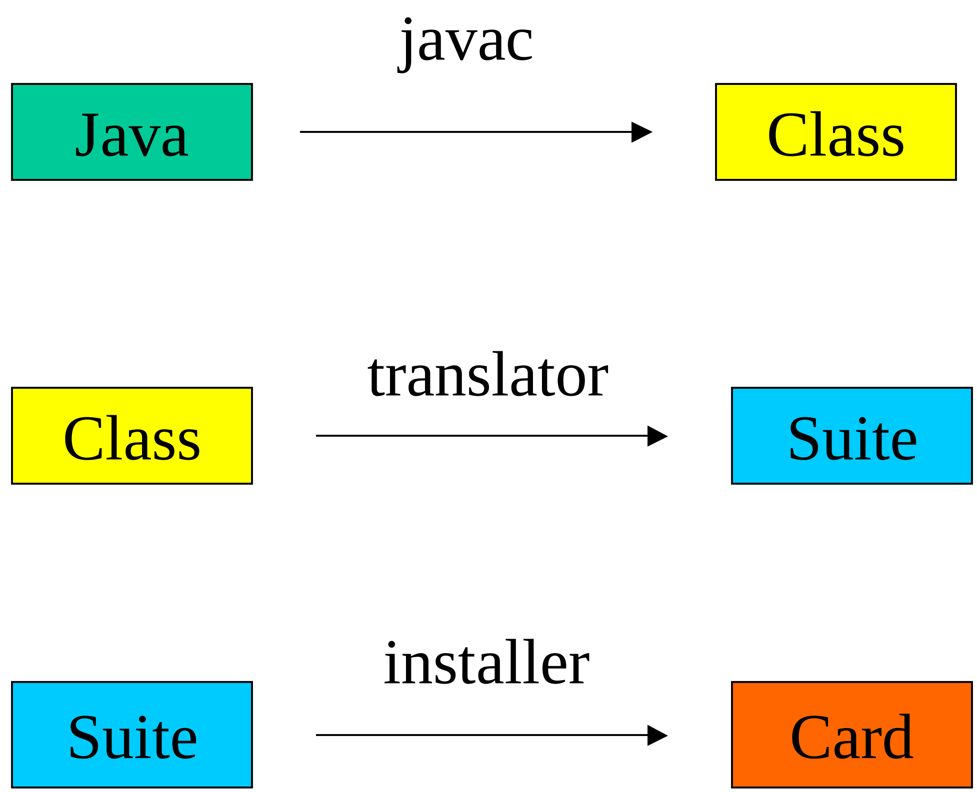
Application architecture

Squawk Technology

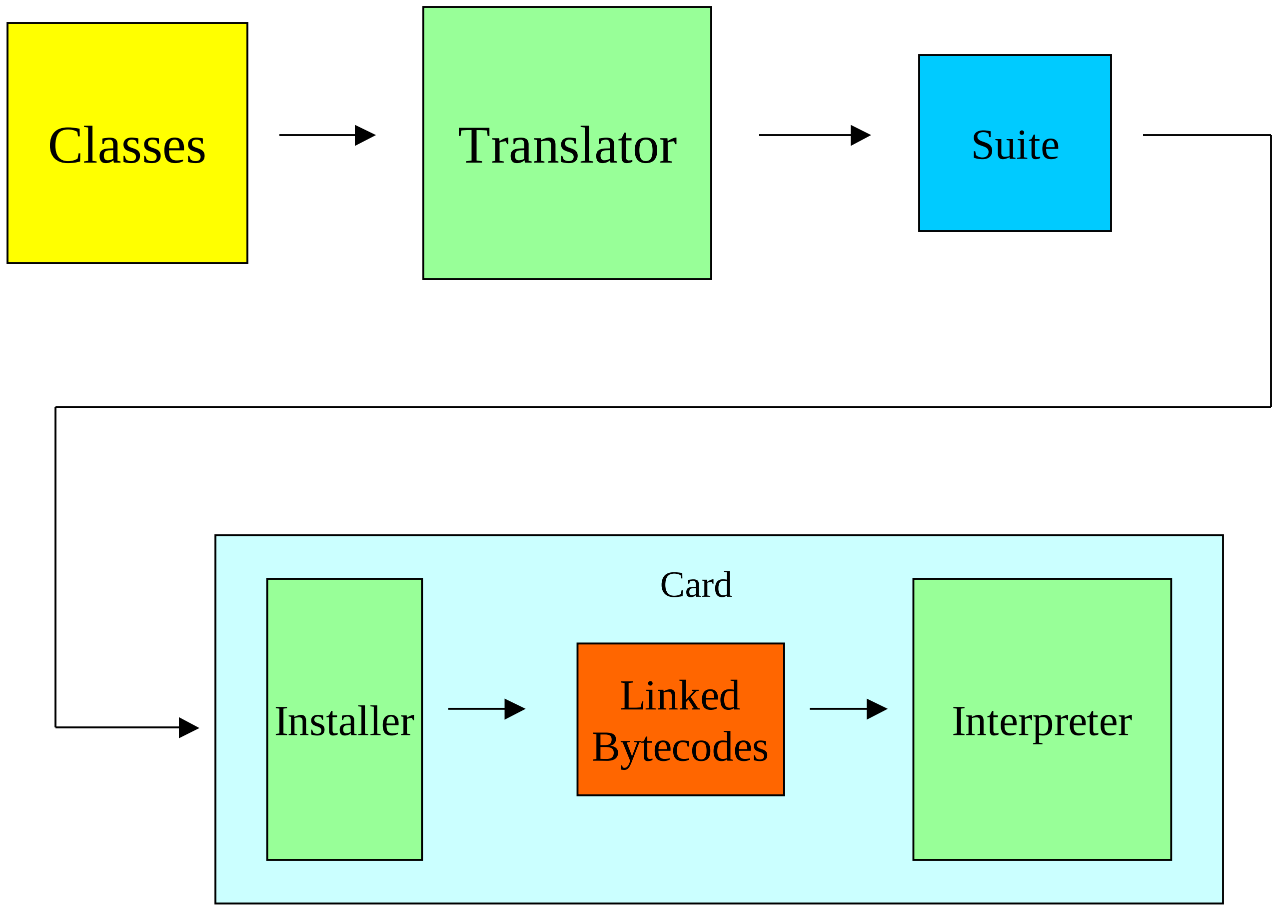


- Java source is converted into class files
- Class files are converted into a suite
- Suites are installed onto the Java card

Squawk Technology



Squawk Technology



Translation

Source example

A method from java.lang.Object

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

After javac

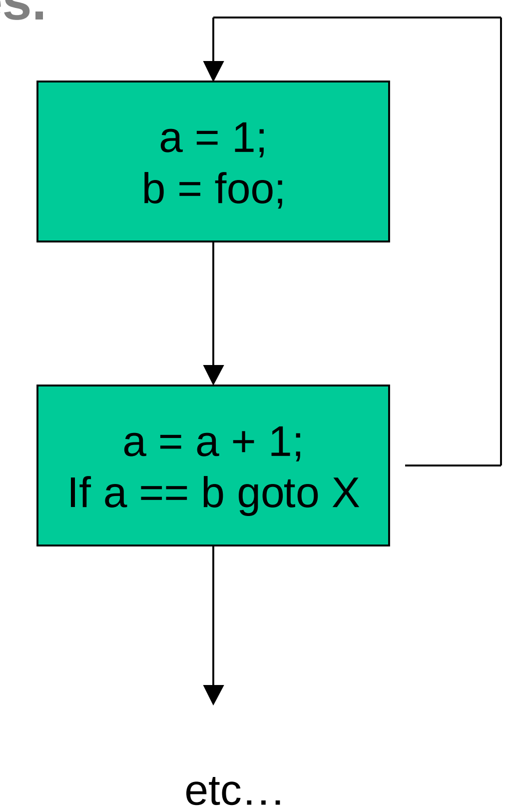
```
Method boolean equals(java.lang.Object)
   0 aload_0
   1 aload_1
   2 if_acmpne 9
   5 iconst_1
   6 goto 10
   9 iconst_0
  10 ireturn
```

After translation (XML output)

```
<method_body>
  <type>1</type>
  <entry>7</entry>
  <locals>
    <type>1</type>
    <type>1</type>
  </locals>
  <stack>2</stack>
  <code>
    <load_0/>
    <load_1/>
    <if_icmpne/><byte>2</byte>
    <const_1/>
    <return/>
    <const_0/>
    <return/>
  </code>
</method_body>
```

Bytecode manipulation

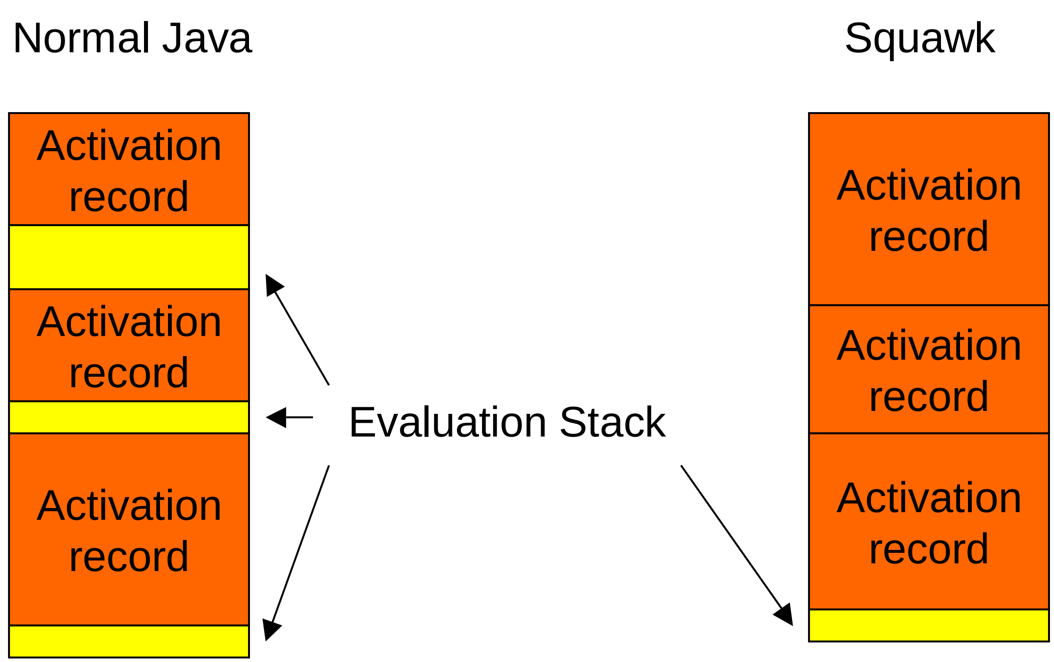
- **The stack must be empty at basic block boundaries.**



This is an issue that simplifies verification.

Bytecode manipulation

- The stack must only contain the operands for certain operations such as invoke, getstatic etc.



This is an issue that greatly simplifies GC.

Bytecode manipulation

- **Local variables can only be used for one data type.**

Normal Java
Header
int/Object
float/byte[]

Squawk
Header
Object
byte[]
int
float

This is an issue that simplifies verification and GC.

Bytecode manipulation

- The 'new' bytecode is replaced with 'clinit' and constructor methods are responsible for object creation.

Normal Java

```
new
dup
invokespecial
store n
```

Squawk

```
clinit
const_null
invokeinit
Store n
```

This is an issue that simplifies verification.

Bytecode manipulation

- All ‘dup’ bytecodes are replaced with increased use of local variables.
- The constant pool is replaced by inlining constants, and a per-class object and a class reference pool.
- The size of most bytecodes are reduced.
- Substantial size reduction over class files

Cost are minimal:

~5% increase in local variables.

~3% increase in code size.

Semantics of Execution

Java semantics

The semantics of Squawk Java at the source code level is the same as the CLDC 1.0 Java platform.

```
public CubeCanvas() {
    int distance, factor, tmp, index = 4;

    screen_x = new int[250];
    screen_y = new int[250];
    x = new int[12];
    y = new int[12];

    rnd = new Random();

    objectList = new TObject[13];

    sun = objectList[0] = new TObject(getCubeData());
    sun.setColorIndex(ColorTable.getIndexForColor(ColorTable.red));
    sun.scaleDown(2);

    mercury = objectList[1] = new TObject(getCubeData());
    mercury.translate(0, 0, -300);
    mercury.setColorIndex(ColorTable.getIndexForColor(ColorTable.gray));
    mercury.scaleDown(2);
    mercury.rotationOrigin = sun.origLocation;
    …etc…
```

Java semantics

- This means Squawk supports
  - Threads
  - Object synchronization
  - Exceptions
  - Objects and arrays in RAM
  - Multidimensional arrays
  - Dynamic class loading and unloading
  - Long, double, and floating point data types
  - Class initialization
  - Class verification
  - Exact garbage collection
  - The standard execution lifecycle

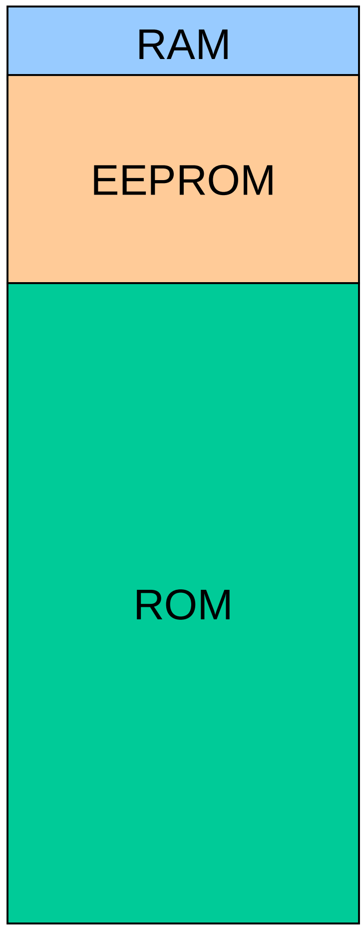
However, all of these are optional features that can easily be customized for Java card.

Semantic extensions

- Squawk has been designed to support a number of semantic extensions to the standard platform
  - Persistent read-only & read-write object memories.
  - The Isolate API
  - Other issues like the firewall, transactions etc.

Memory model

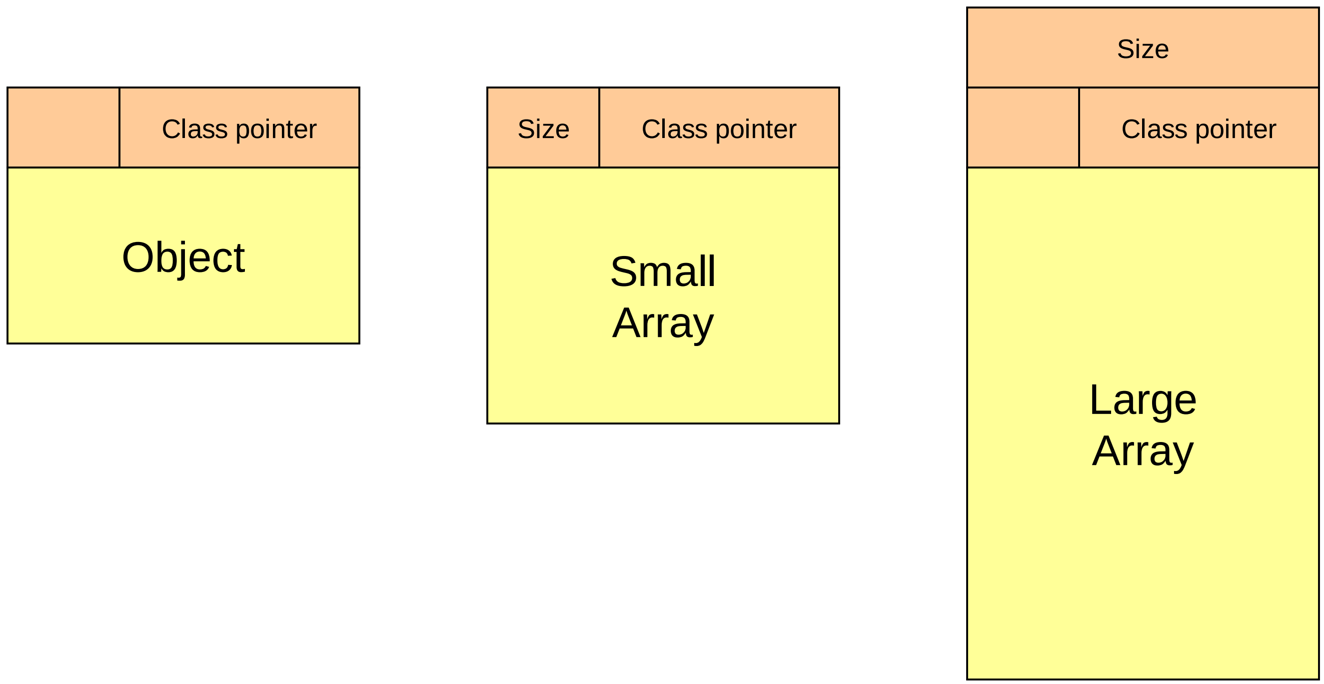
Memory model



- Squawk uses three object memories.
- All Squawk data structures are Java objects.
- The same garbage collector is used for the RAM and EEPROM.

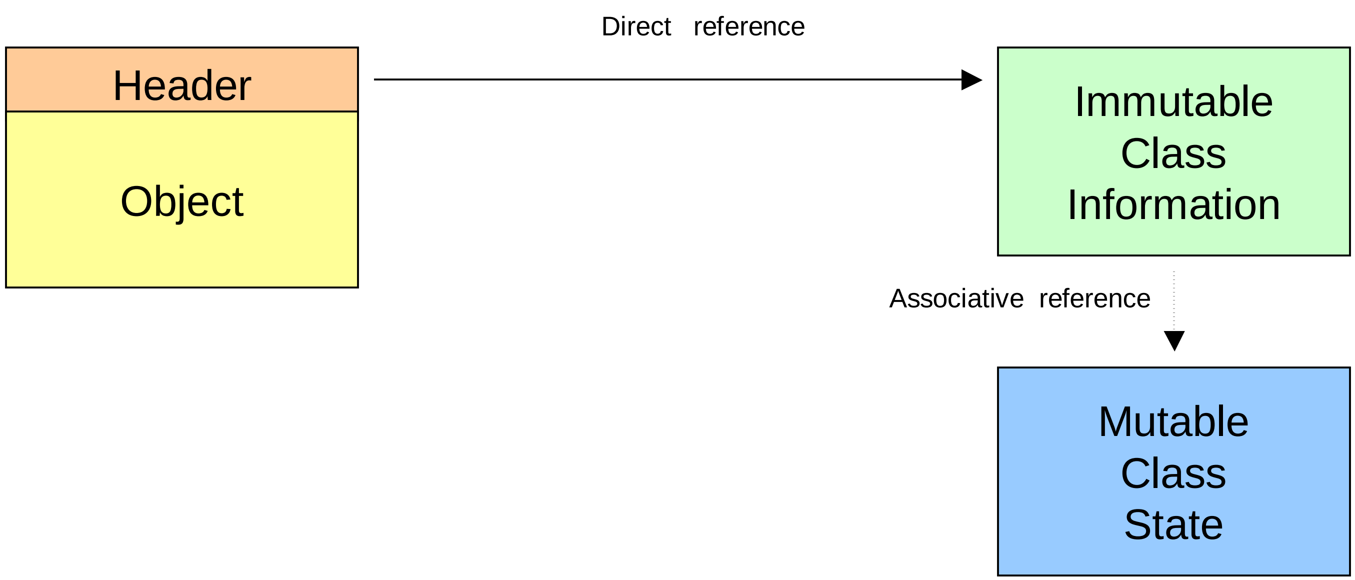
Object format

- Most objects have a 32 bit header.
- 8 bits are used for the size of small arrays.
- Arrays larger than 254 elements require an extra word.

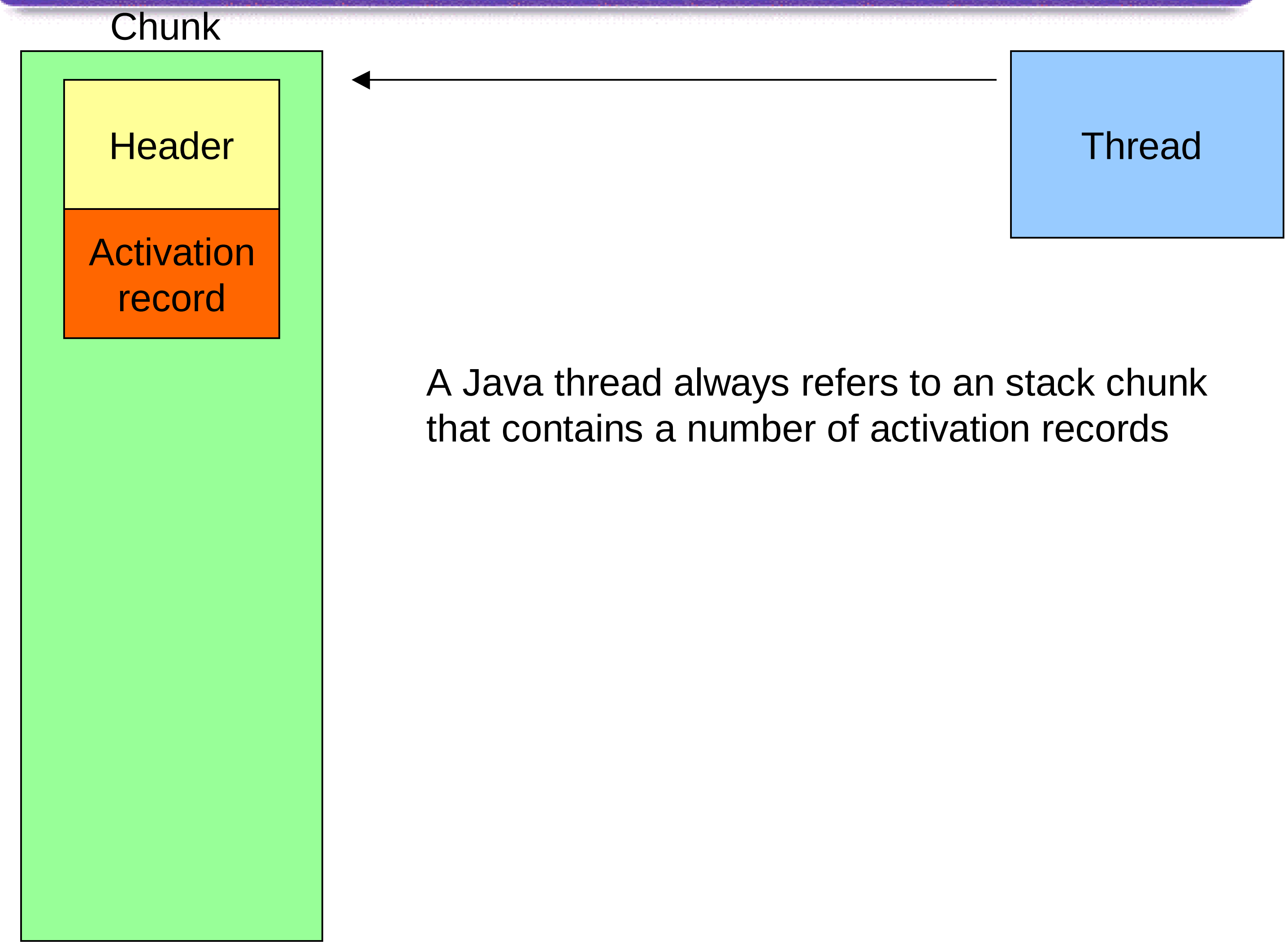


Class references

- Objects directly refer to the immutable information of their class. This can therefore be kept in ROM.
- The mutable class state is held separately in an associatively mapped location.

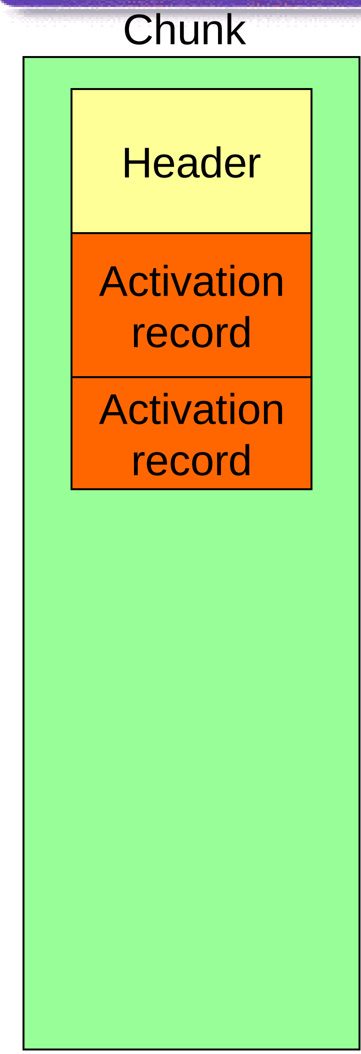


Activation records, Stack chunks, and Threads



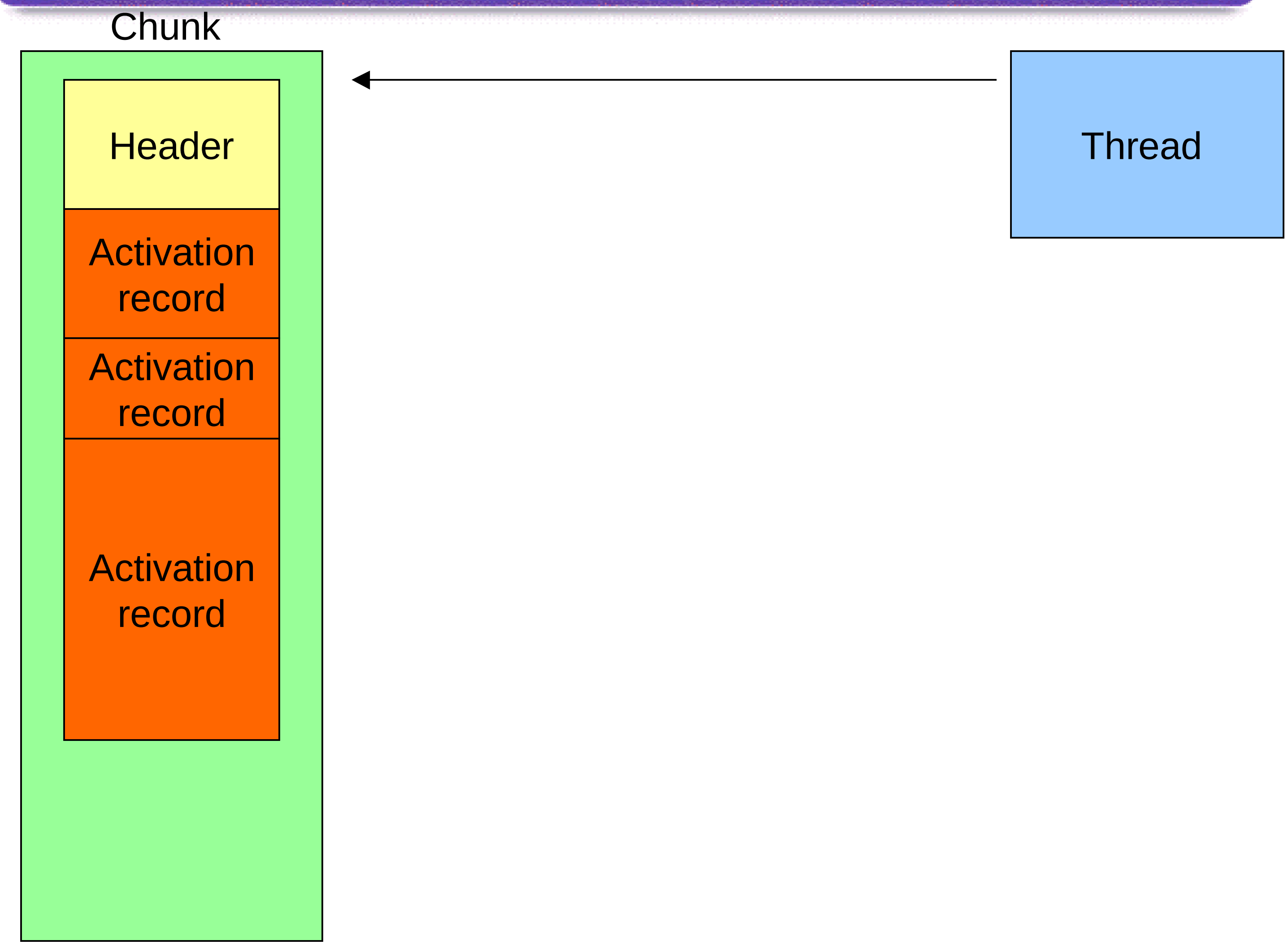
A Java thread always refers to an stack chunk that contains a number of activation records

Activation records, Stack chunks, and Threads

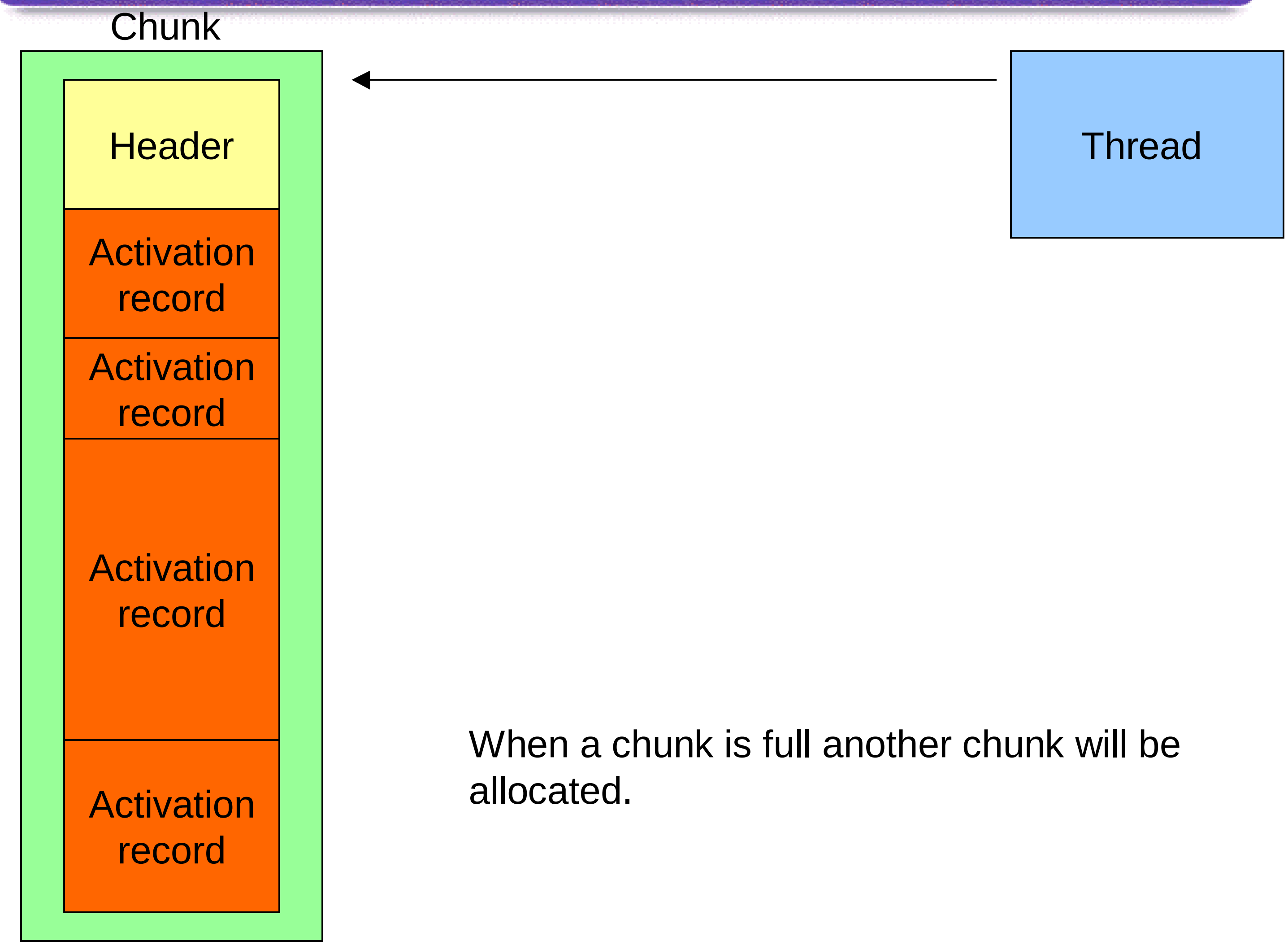


As the method call depth increases so does the list of activation records.

Activation records, Stack chunks, and Threads

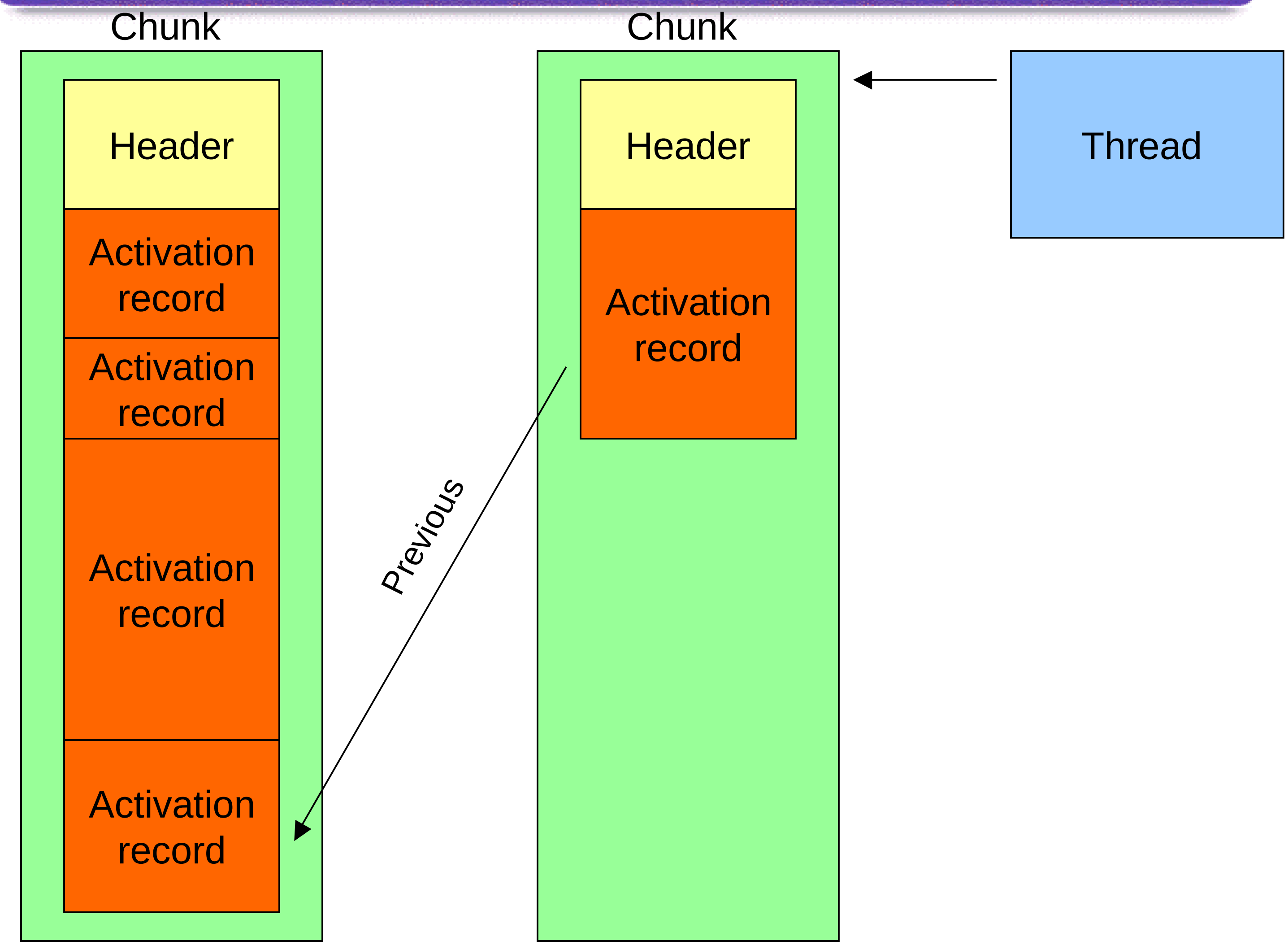


Activation records, Stack chunks, and Threads

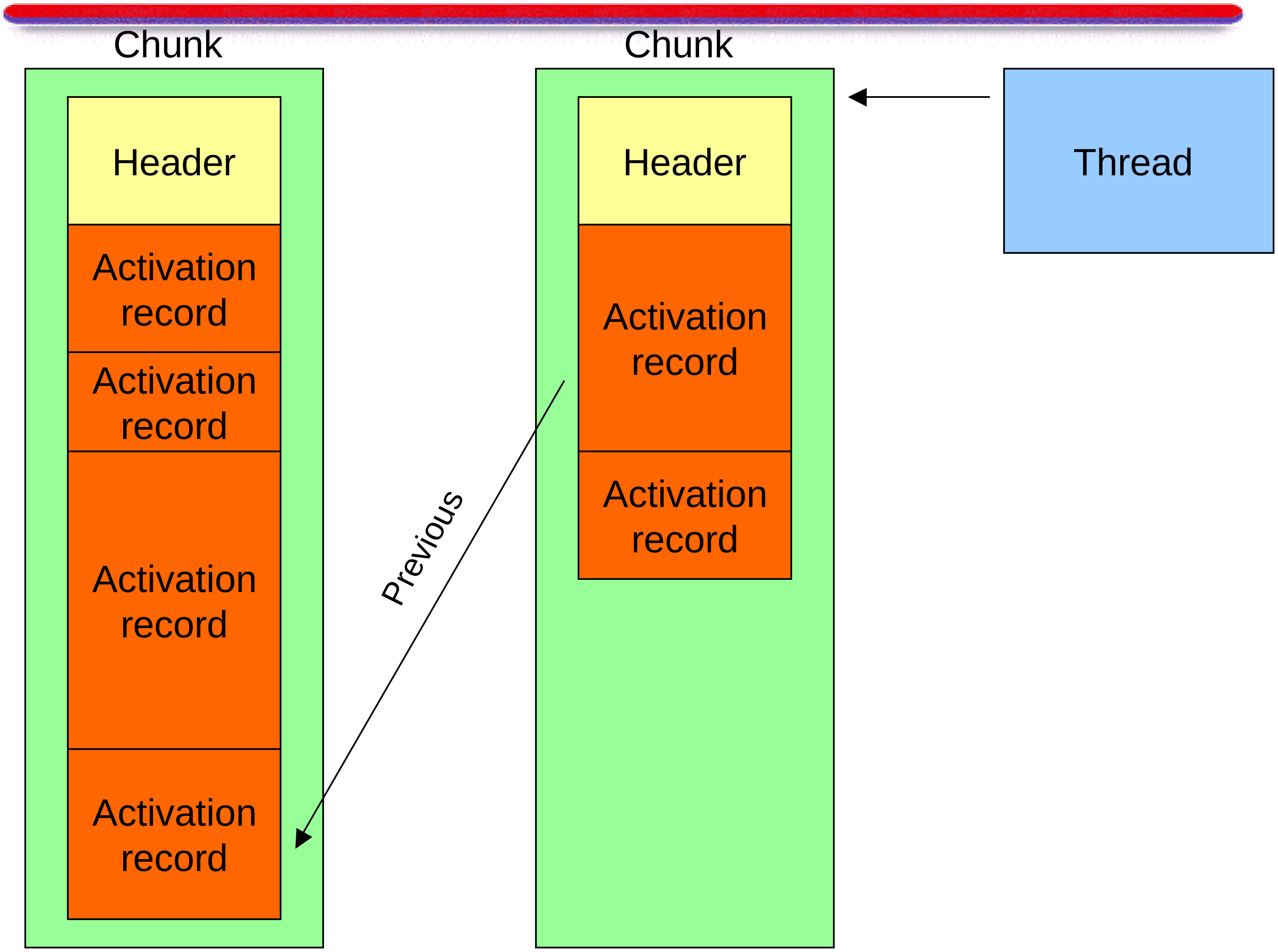


When a chunk is full another chunk will be allocated.

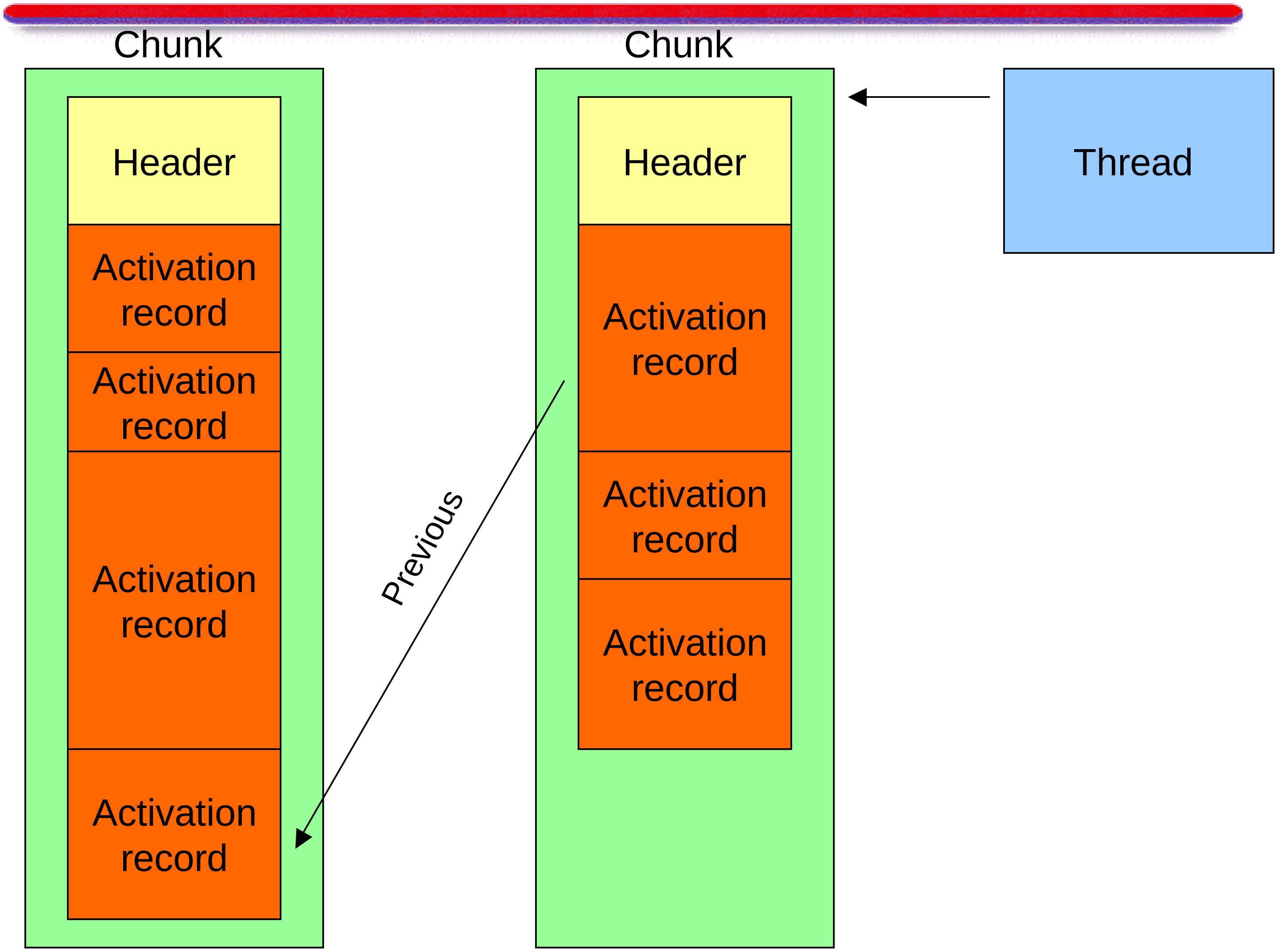
Activation records, Stack chunks, and Threads



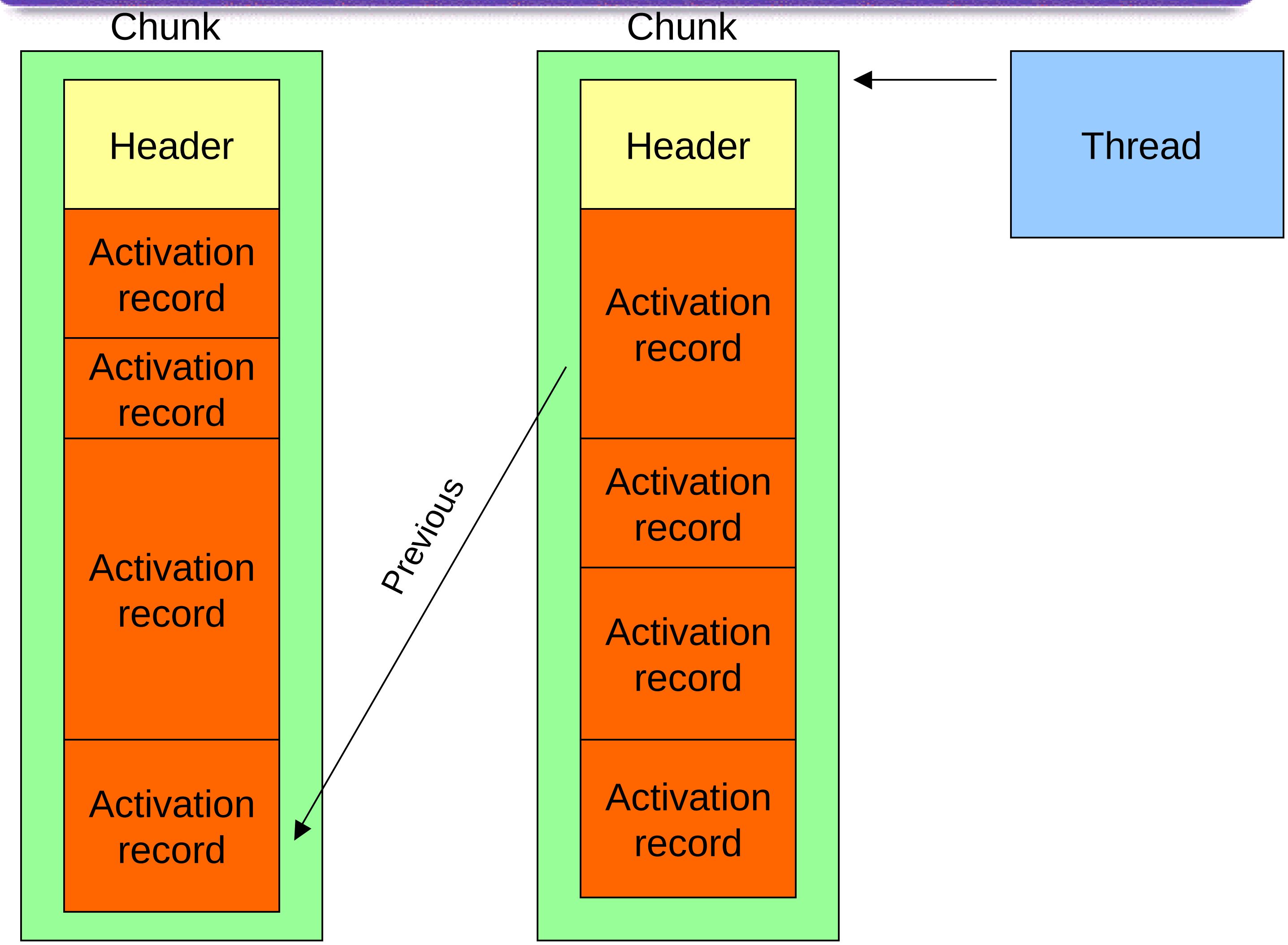
Activation records, Stack chunks, and Threads



Activation records, Stack chunks, and Threads



Activation records, Stack chunks, and Threads



Interpreter core engineering

Java to C conversion

All of the Squawk project is written in Java. The core interpreter is converted into C.

Interpreter.java → squawk.c

```
$ls -l *.java

     17519 Dec 20 13:28 Interpreter.java
    114021 Dec 18 17:30 Interpret.java
     14871 Dec 18 17:30 Memory.java
     99364 Dec 20 10:59 ObjectMemory.java
     47358 Dec 23 14:12 PlatformAbstraction.java
```

```
$ls -l *.c

     16972 Dec 23 14:39 squawk.c
    113779 Dec 23 14:39 interp.c
     14341 Dec 23 14:39 memory.c
     93520 Dec 23 14:39 object.c
     46381 Dec 23 14:39 platform.c
```

Example code

```
case OPC_IADD:  { int r = pop() ; int l = pop() ; push(l + r); continue; }
case OPC_ISUB:  { int r = pop() ; int l = pop() ; push(l - r); continue; }
case OPC_IAND:  { int r = pop() ; int l = pop() ; push(l & r); continue; }
case OPC_IOR:   { int r = pop() ; int l = pop() ; push(l | r); continue; }
case OPC_IXOR:  { int r = pop() ; int l = pop() ; push(l ^ r); continue; }
etc…
```

Most of the interpreter is written in a subset or Java and C.

Language differences

Original Java

```
    void copyBytes(int src, int dst, int num) {
        if (num < 0) {
            fatalVMError("Negative range");
        }
/*IFJ*/ System.arraycopy(memory, src, memory, dst, num);
//IFC// memmove(memory+dst, memory+src, num);
    }
```

Derived C

```
    void copyBytes(int src, int dst, int num) {
        if (num < 0) {
            fatalVMError("Negative range");
        }
/**** Line deleted by Squawk builder ****/
        memmove(memory+dst, memory+src, num);
    }
```

Language differences are solved using a special form of conditional compilation

Macro generation

Original Java

```
/*MAC*/int Frame_getLocal(int frame, int n) { return getWord(frame, n); }
```

Derived C

```
#define Frame_getLocal(frame, n) (getWord(frame, n))
```

Example of use

```
case OPC_LOAD: {
    push(Frame_getLocal(lp, fetchUnsignedByte(ip++)));
    continue;
}
```

C macros are used for efficiency and code size

Feature elimination

```
/*if[FLOATS]*/

    public final void writeFloat(float v) throws IOException {
        writeInt(Float.floatToIntBits(v));
    }

/*end[FLOATS]*/
```

Features can be excluded from the interpreter core and the Java runtime libraries.

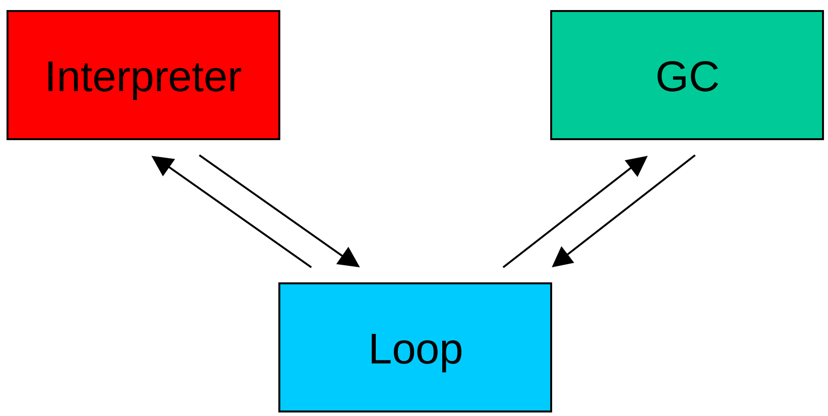
The Garbage Collector

GC Features

- The collector is exact.
  - Conservative (inexact) GC prevents memory compaction.
  - Exact collection is faster and uses less memory.
- The translator makes exact garbage collection much easier
  - Local variables are either pointers or non-pointers
  - Sections of evaluation stack are not found between activation records.
- The interpreter and the collector are never running at the same time.

The Interpreter – GC Loop

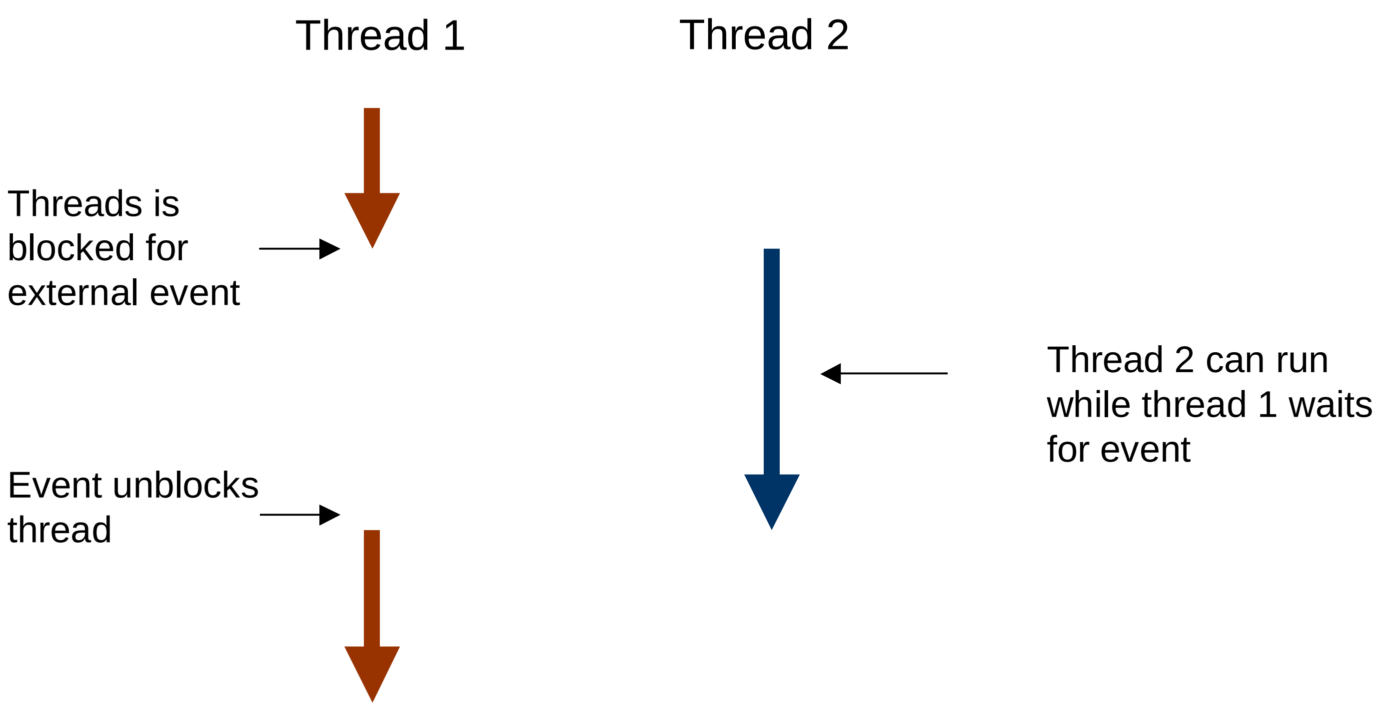
```
for (;;) {
    chunk = interpret(chunk, res);
    setCurrentStackChunk(chunk);
    res   = gc();
    chunk = getCurrentStackChunk();
}
```



Native Methods

Native methods

- Squawk supports a native method interface that can cause a Java thread to be suspended while it is waiting for an asynchronous event to take place.



Native methods

- However thread scheduling is never done in native code.
- Native methods are divided into three parts.
  - parm()   – Sets up a parameter
  - startIO() – This does the request and returns execution status.
  - endIO()   – This returns the result of the operation.

```
public int read(byte b[], int off, int len) throws IOException {

    do {
        Native.parm(Native.OP_READBUF);
        Native.parm(b);
        Native.parm(off);
        Native.parm(len);
    } while(Native.startIO(chan));

    return Native.endIO(chan);
}
```

Native methods

- Thread blocking is done in startIO() when the native method indicates that the operation cannot be completed immediately.

```
public static boolean startIO(int chan) throws IOException {

    int event = execute(chan);

    if (event > 0) {
       Thread.waitForEvent(event);     // Wait for event to be ready
       return true;                    // Tell caller to repeat request
    }

    return false;                      // Request finished
}
```

Native methods

- **The native code must have execute() and a result() functions.**

```
int execute(int *parms) {
    switch (parms[0]) {
        case OP_READBUF: {
            if (dataNotReady) {
                 return eventNumber;
            }
            . . . Body of request . . .
            result = something;
            return 0;
        }
    }
}

jlong result() {
    return result;
}
```

Native methods

- When an event occurs the event number is recorded in native code.
- Asynchronous event notification is polled from Java code.

```
void someEvent() {
    addEventToQueue(eventNumber);
}

int getEvent() {
    return getEventFromQueue();
}

while ((eventNumber = Native.getEvent()) != 0) {
    Thread thread = (Thread)events.remove(eventNumber);
    runnableThreads.add(thread);
}
```

Native methods

- This interface has the following advantages:
  - The system is fully asynchronous allowing Java threads to execute while other threads are waiting for I/O.
  - Pointers into the object memory are never retained in native code because requests that cannot be satisfied immediately are always repeated.
  - This means that when the garbage collector runs there is never an object pointer in native code that needs updating.
  - Thread scheduling is all done in Java code.

The Demo

Files

- squawk.jar contains the java version and the I/O library.
- squawk.exe is the C version.
- squawk.image is an image of the ROM, EEPROM, and RAM.
- squawk.txt is an description of how to use the demo.

```
  36864 Jan  4 13:44 squawk.exe
4784160 Jan  4 13:37 squawk.image
 204775 Jan  4 13:44 squawk.jar
   3028 Dec 31 17:15 squawk.txt
```

I/O

- The version of the demo in C uses the same Java code for I/O.