Inside a JVM

With Cliff Click

WHAT IS IN A JVM?

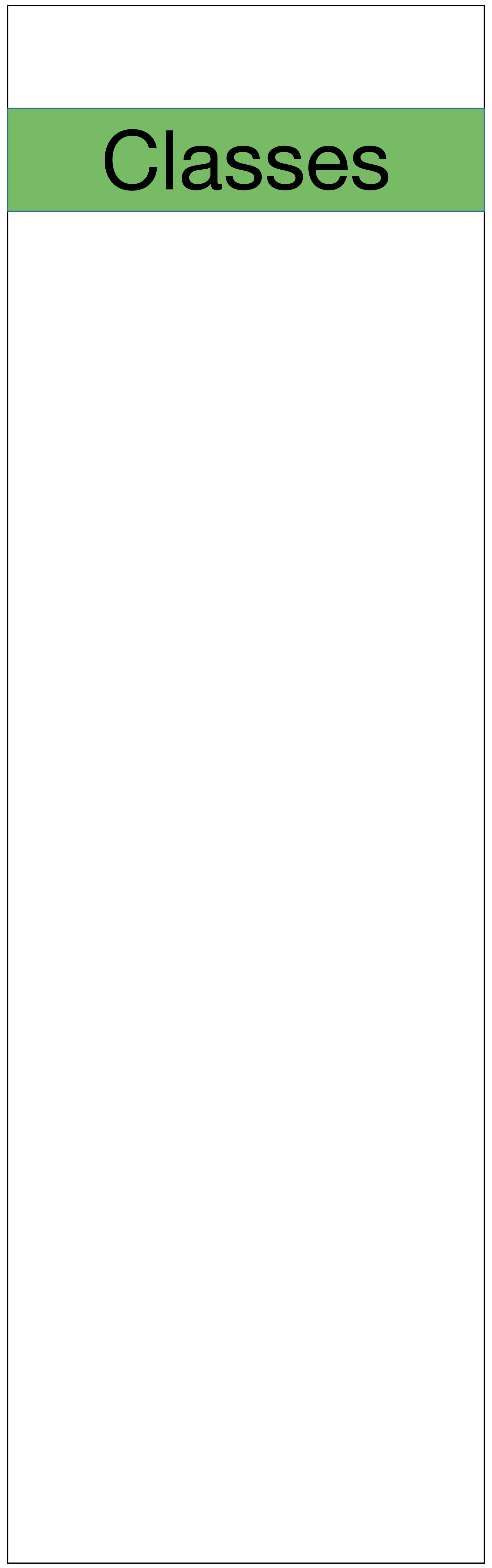
WHAT IS IN A JVM?

Well… lots! Let's break down the big pieces:

WHAT IS IN A JVM?

Well… lots! Let’s break down the big pieces:

Classes, Meta-data, things that describe your program structures.

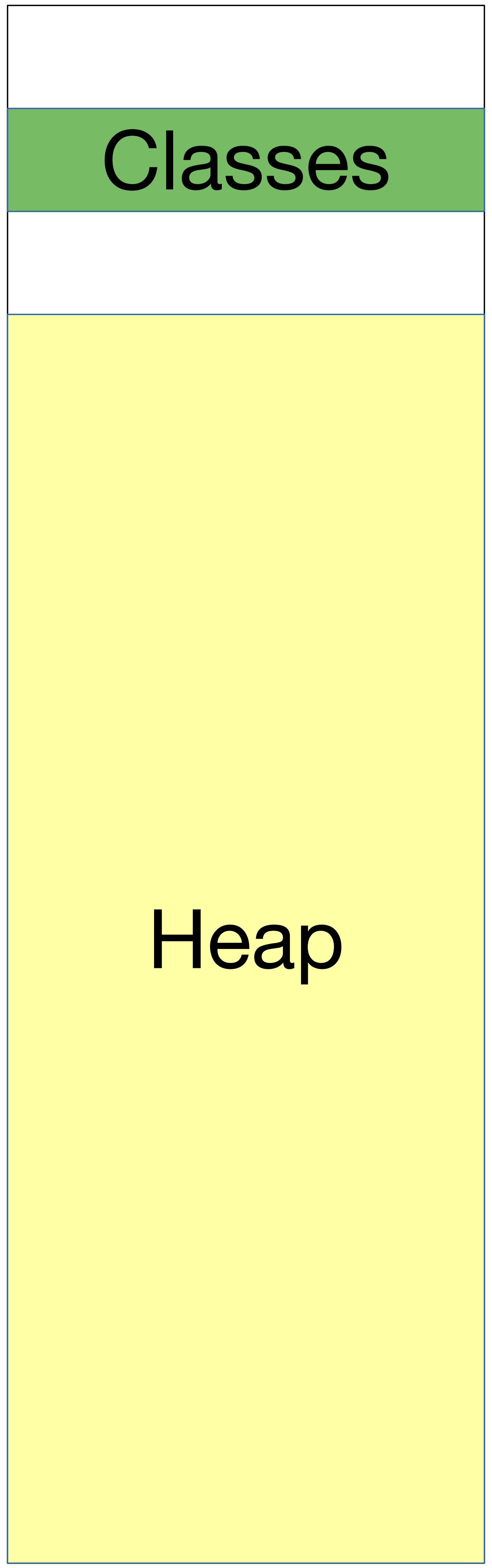


WHAT IS IN A JVM?

Well… lots! Let’s break down the big pieces:

Classes, Meta-data, things that describe your program structures.

Data, the Heap, and Garbage Collection to manage it.



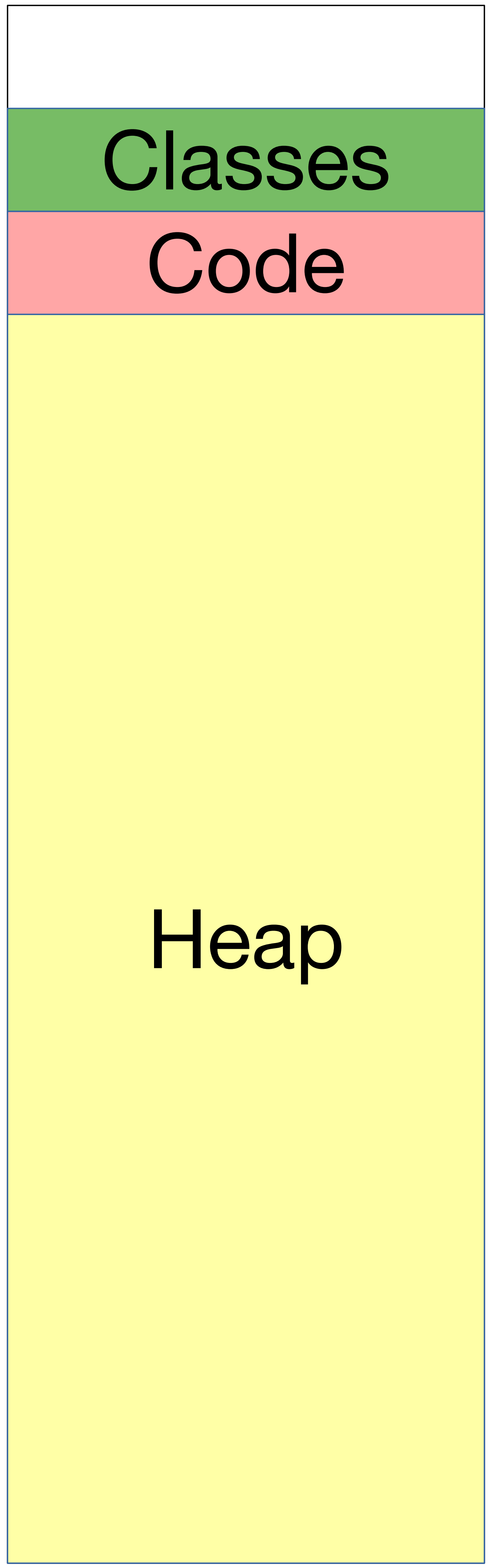
WHAT IS IN A JVM?

Well… lots! Let's break down the big pieces:

Classes, Meta-data, things that describe your program structures.

Data, the Heap, and Garbage Collection to manage it.

Bytecodes, an Execution Model; an Interpreter and JIT(s) to run fast.



WHAT IS IN A JVM?

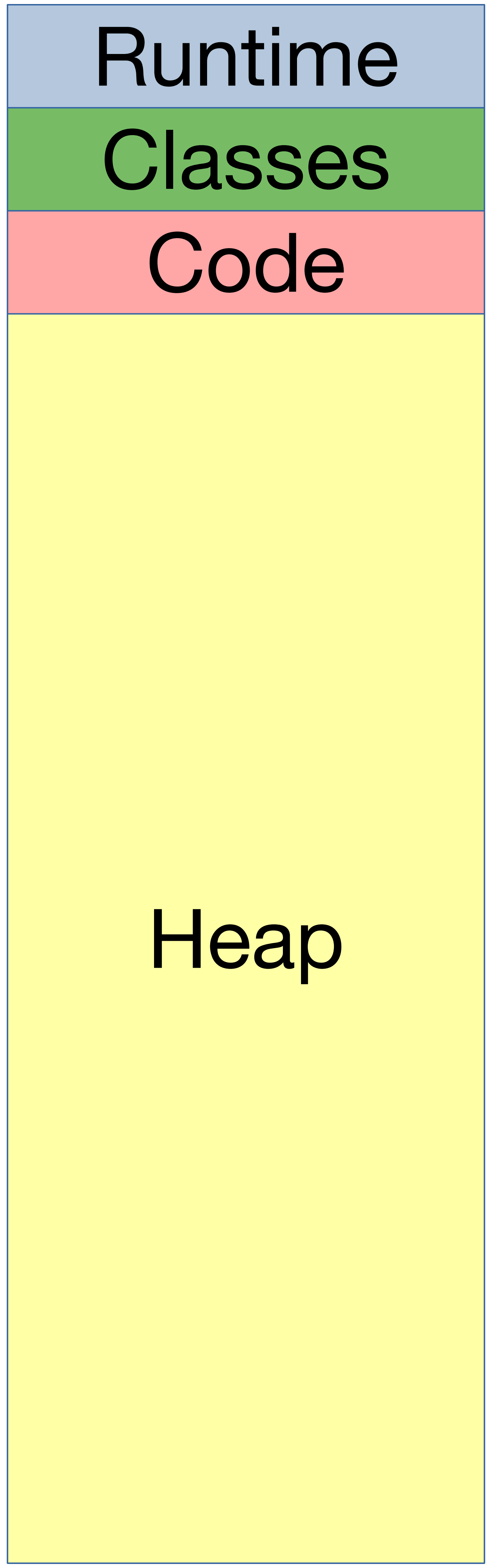
Well… lots! Let's break down the big pieces:

Classes, Meta-data, things that describe your program structures.

Data, the Heap, and Garbage Collection to manage it.

Bytecodes, an Execution Model; an Interpreter and JIT(s) to run fast.

Runtime, Locks & Threads, OS access (files,JNI,Time), Debugging



Classes & Meta-data

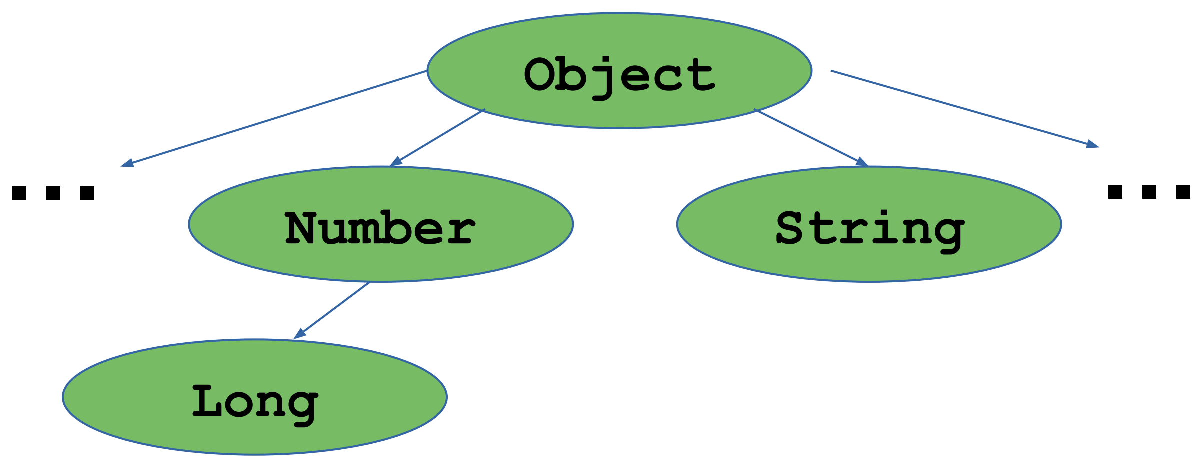
Classes! Java has classes, this is easy!

Classes & Meta-data

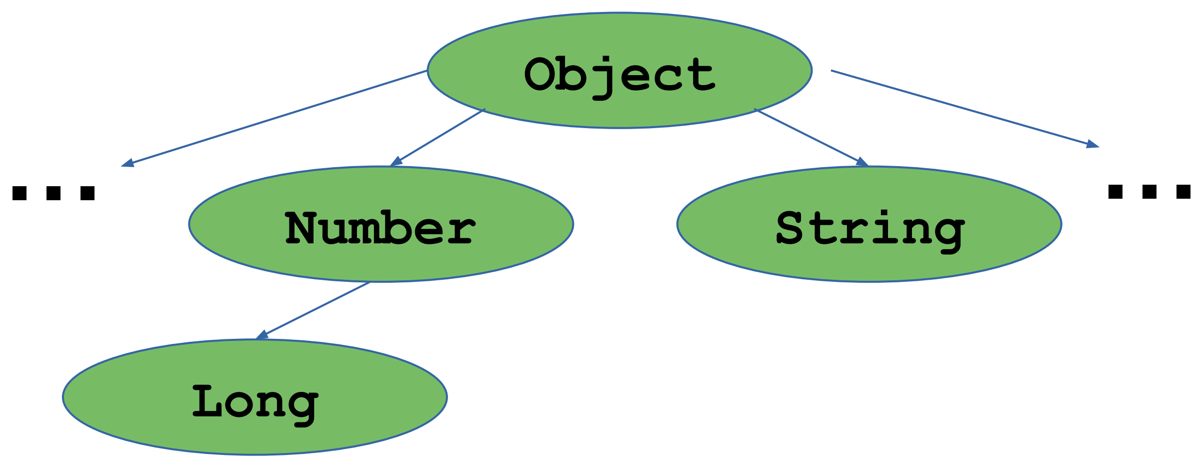
Classes! Java has classes, this is easy!

Classes form a tree, rooted at Object.

Classes



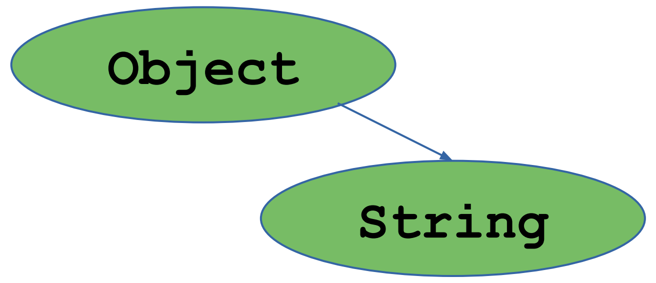
Classes & Meta-data



Classes! Java has classes, this is easy!

Classes form a tree, rooted at Object. They have a super-class and some subclasses (and interfaces).

Classes & Meta-data



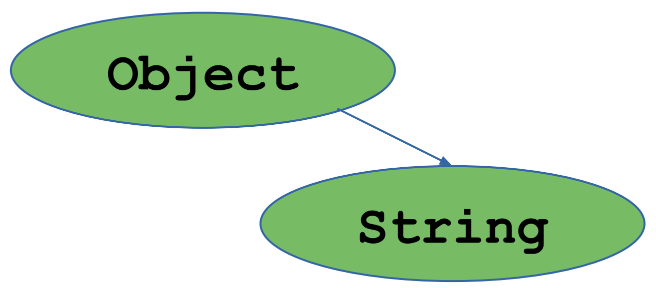
Classes! Java has classes, this is easy!

Classes form a tree, rooted at Object. They have a super-class and some subclasses (and interfaces). They have fields and methods.

```
public final class String implements ... {
   /** The value is used for character storage. */
   private final char value[];

   /** Cache the hash code for the string */
   private int hash; // Default to 0

...
```

```
/**
 * Returns a hash code for this string.
 * @return  a hash code value for this object.
 */
public int hashCode() {
   int h = hash;
   if (h == 0 && value.length > 0) {
      char val[] = value;

      for (int i = 0; i < value.length; i++) {
         h = 31 * h + val[i];
      }
      hash = h;
   }
   return h;
}
```

Classes & Meta-data

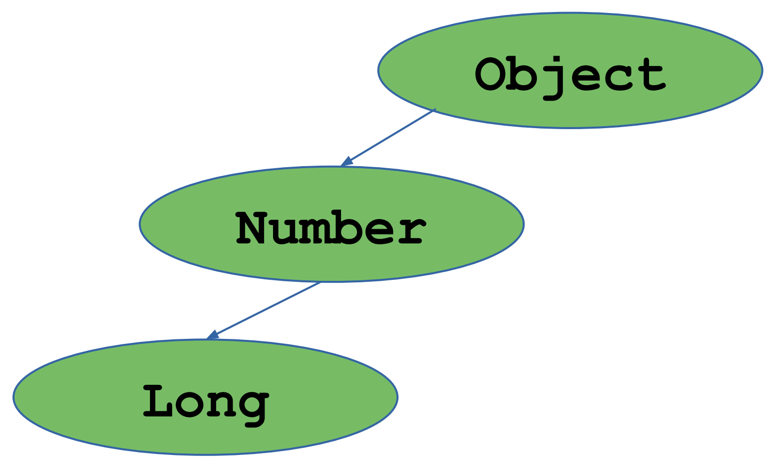


Classes! Java has classes, this is easy!

Classes form a tree, rooted at Object. They have a super-class and some subclasses (and interfaces). They have fields and methods. They are initialized (or not), and the <clinit> has run (or not).

```
/** use serialVersionUID from JDK 1.0.2 for interoperability */
private static final long serialVersionUID = -6849794470754667710L;

/**
 * Class String is special cased within the Serialization Stream Protocol.
 *
 * A String instance is written into an ObjectOutputStream according to
 * <a href="{@docRoot}/../platform/serialization/spec/output.html">
 * Object Serialization Specification, Section 6.2, "Stream Elements"</a>
 */
private static final ObjectStreamField[] serialPersistentFields =
    new ObjectStreamField[0];
```

Classes & Meta-data

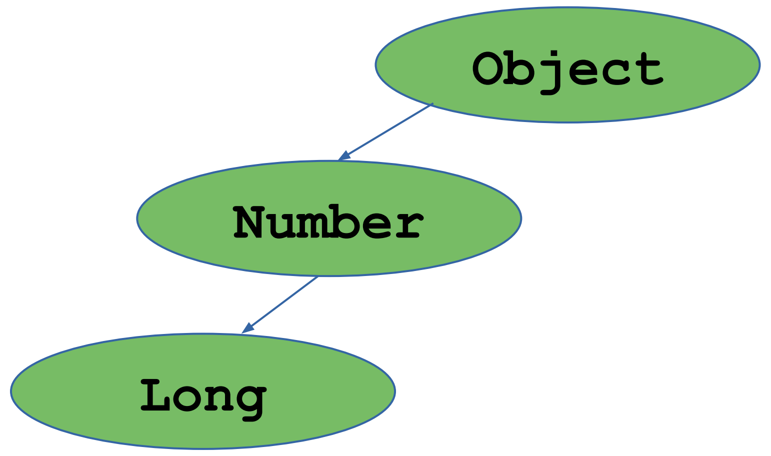


Classes! Java has classes, this is easy!

Classes form a tree, rooted at Object. They have a super-class and some subclasses (and interfaces). They have fields and methods. They are initialized (or not), and the <clinit> has run (or not). They might have inner classes, or be “Single Abstract Method”.

```
public final class Long extends Number implements Comparable<Long> {
...
  private static class LongCache {
      private LongCache(){}

      static final Long cache[] = new Long[-(-128) + 127 + 1];

      static {
          for(int i = 0; i < cache.length; i++)
              cache[i] = new Long(i - 128);
      }
  }
```

Classes & Meta-data

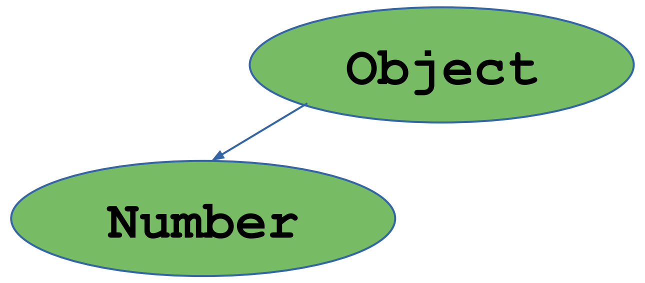


Classes! Java has classes, this is easy!

Classes form a tree, rooted at Object. They have a super-class and some subclasses (and interfaces). They have fields and methods. They are initialized (or not), and the <clinit> has run (or not). They might have inner classes, or be "Single Abstract Method". They have instances.

```
public final class Long extends Number implements Comparable<Long> {
...
  private static class LongCache {
      private LongCache(){}

      static final Long cache[] = new Long[-(-128) + 127 + 1];

      static {
          for(int i = 0; i < cache.length; i++)
              cache[i] = new Long(i - 128);
      }
  }
```

Classes & Meta-data



Classes! Java has classes, this is easy!

Classes form a tree, rooted at Object. They have a super-class and some subclasses (and interfaces). They have fields and methods. They are initialized (or not), and the <clinit> has run (or not). They might have inner classes, or be “Single Abstract Method”. They have instances. They can be abstract or have custom Loaders or Security Domains or annotations or…

```
 * @jls 5.1.2 Widening Primitive Conversions
 * @jls 5.1.3 Narrowing Primitive Conversions
 * @since   JDK1.0
 */
public abstract class Number implements java.io.Serializable {
    /**
     * Returns the value of the specified number as an {@code int},
     * which may involve rounding or truncation.
```

Classes & Meta-data

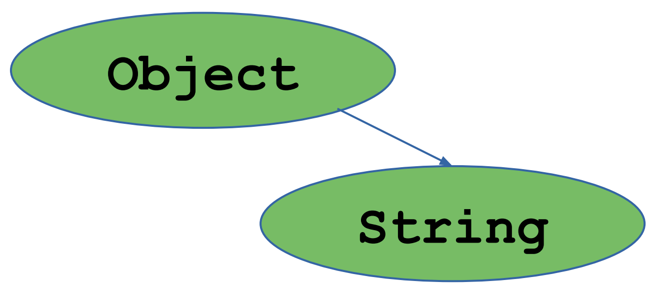
Classes! Java has classes, this is easy!

Classes form a tree, rooted at Object. They have a super-class and some subclasses (and interfaces). They have fields and methods. They are initialized (or not), and the <clinit> has run (or not). They might have inner classes, or be “Single Abstract Method”. They have instances. They can be abstract or have custom Loaders or Security Domains or annotations or…

Meta-data! There is also internal **meta-data,** not available via reflection.

Classes

Classes & Meta-data

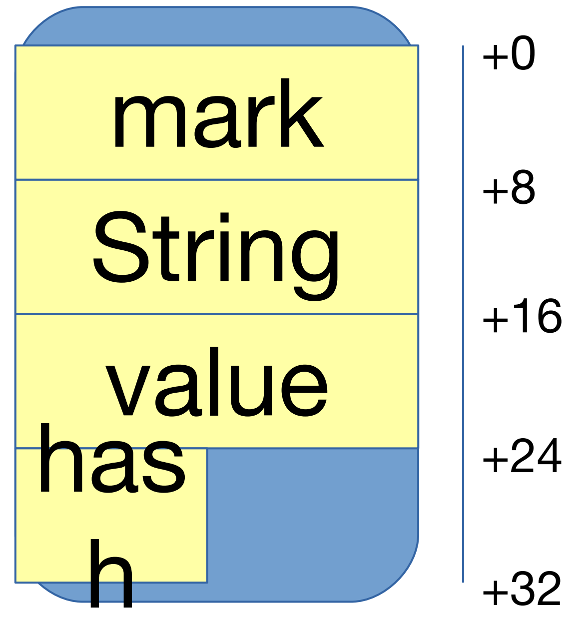


Classes! Java has classes, this is easy!

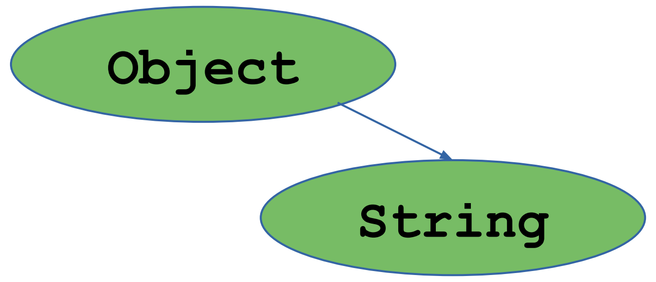
Classes form a tree, rooted at Object. They have a super-class and some subclasses (and interfaces). They have fields and methods. They are initialized (or not), and the <clinit> has run (or not). They might have inner classes, or be "Single Abstract Method". They have instances. They can be abstract or have custom Loaders or Security Domains or annotations or…

Meta-data! There is also internal **meta-data,** not available via reflection.

Object field layout: offset & size.



Classes & Meta-data

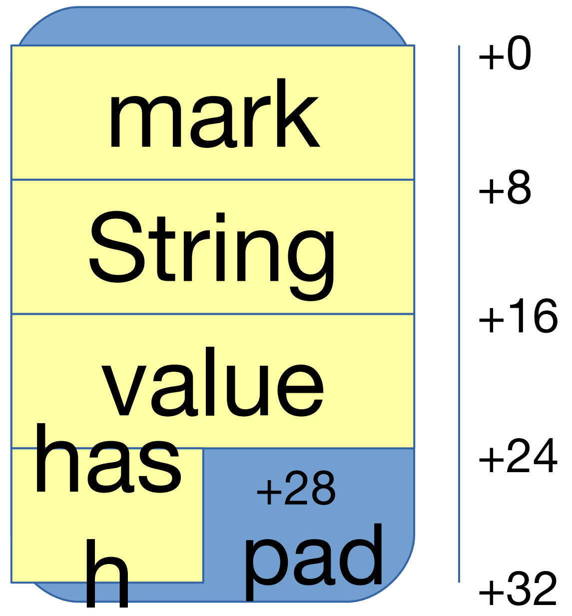


Classes! Java has classes, this is easy!

Classes form a tree, rooted at Object. They have a super-class and some subclasses (and interfaces). They have fields and methods. They are initialized (or not), and the <clinit> has run (or not). They might have inner classes, or be "Single Abstract Method". They have instances. They can be abstract or have custom Loaders or Security Domains or annotations or…

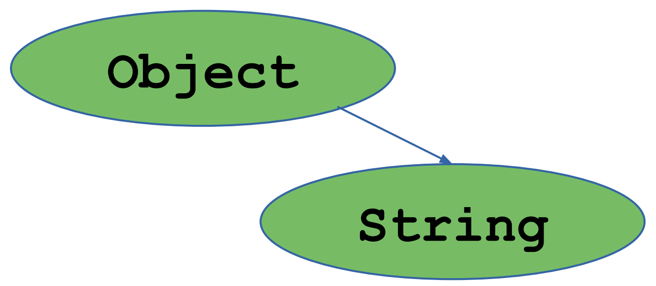
Meta-data! There is also internal **meta-data,** not available via reflection.

Object field layout: offset & size. Padding.



Classes

Classes & Meta-data



Classes! Java has classes, this is easy!

Classes form a tree, rooted at Object. They have a super-class and some subclasses (and interfaces). They have fields and methods. They are initialized (or not), and the <clinit> has run (or not). They might have inner classes, or be "Single Abstract Method". They have instances. They can be abstract or have custom Loaders or Security Domains or annotations or…

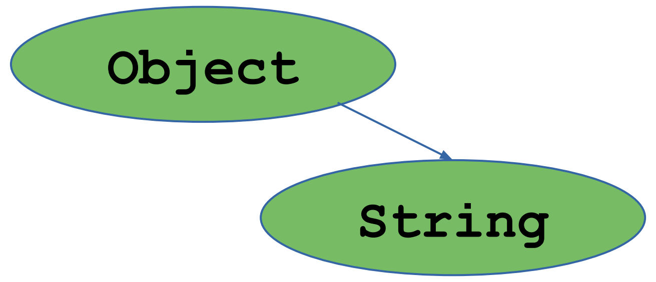
Meta-data! There is also internal **meta-data,** not available via reflection.

Object field layout: offset & size. Padding. Profiling data on methods,

```
String.hashCode:
+0: 100
+2: 1000
+4: 1000
+8:...
```

Classes

Classes & Meta-data



Classes! Java has classes, this is easy!

Classes form a tree, rooted at Object. They have a super-class and some subclasses (and interfaces). They have fields and methods. They are initialized (or not), and the <clinit> has run (or not). They might have inner classes, or be “Single Abstract Method”. They have instances. They can be abstract or have custom Loaders or Security Domains or annotations or…

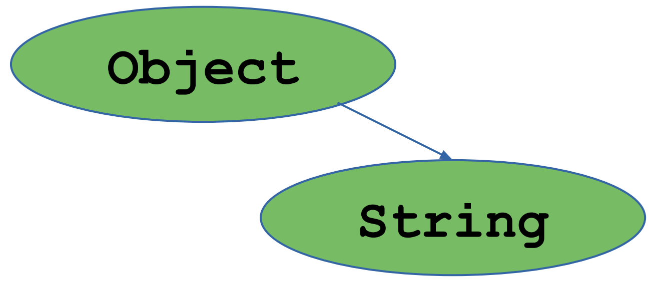
Meta-data! There is also internal **meta-data,** not available via reflection.

Object field layout: offset & size. Padding. Profiling data on methods, but also on fields, locks, exceptions.

```
String.hashCode:
+0: 100
+2: 1000
+4: 1000
+8:...
```

Classes

Classes & Meta-data



Classes! Java has classes, this is easy!

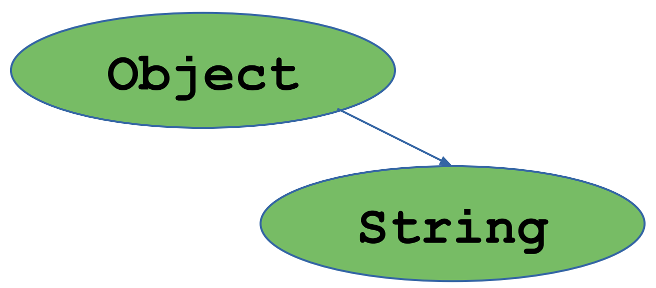
Classes form a tree, rooted at Object. They have a super-class and some subclasses (and interfaces). They have fields and methods. They are initialized (or not), and the <clinit> has run (or not). They might have inner classes, or be "Single Abstract Method". They have instances. They can be abstract or have custom Loaders or Security Domains or annotations or…

Meta-data! There is also internal **meta-data,** not available via reflection.

Object field layout: offset & size. Padding. Profiling data on methods, but also on fields, locks, exceptions. JIT'd code.

```
String.hashCode:
  add esp,16
  mov ebx,ecx
  xor eax,eax
...
```

Classes & Meta-data



Classes! Java has classes, this is easy!

Classes form a tree, rooted at Object. They have a super-class and some subclasses (and interfaces). They have fields and methods. They are initialized (or not), and the <clinit> has run (or not). They might have inner classes, or be "Single Abstract Method". They have instances. They can be abstract or have custom Loaders or Security Domains or annotations or…

Meta-data! There is also internal **meta-data,** not available via reflection.

Object field layout: offset & size. Padding. Profiling data on methods, but also on fields, locks, exceptions. JIT'd code. Safepoints and OopMaps.

```
String.hashCode:
  add esp,16
  mov ebx,ecx
  xor eax,eax
...
```

Classes & Meta-data

Classes! Java has classes, this is easy!

Classes form a tree, rooted at Object. They have a super-class and some subclasses (and interfaces). They have fields and methods. They are initialized (or not), and the <clinit> has run (or not). They might have inner classes, or be "Single Abstract Method". They have instances. They can be abstract or have custom Loaders or Security Domains or annotations or…

Meta-data! There is also internal **meta-data,** not available via reflection.

Object field layout: offset & size. Padding. Profiling data on methods, but also on fields, locks, exceptions. JIT'd code. Safepoints and OopMaps. GC and heap-walking support, and much more.

Classes & Meta-data

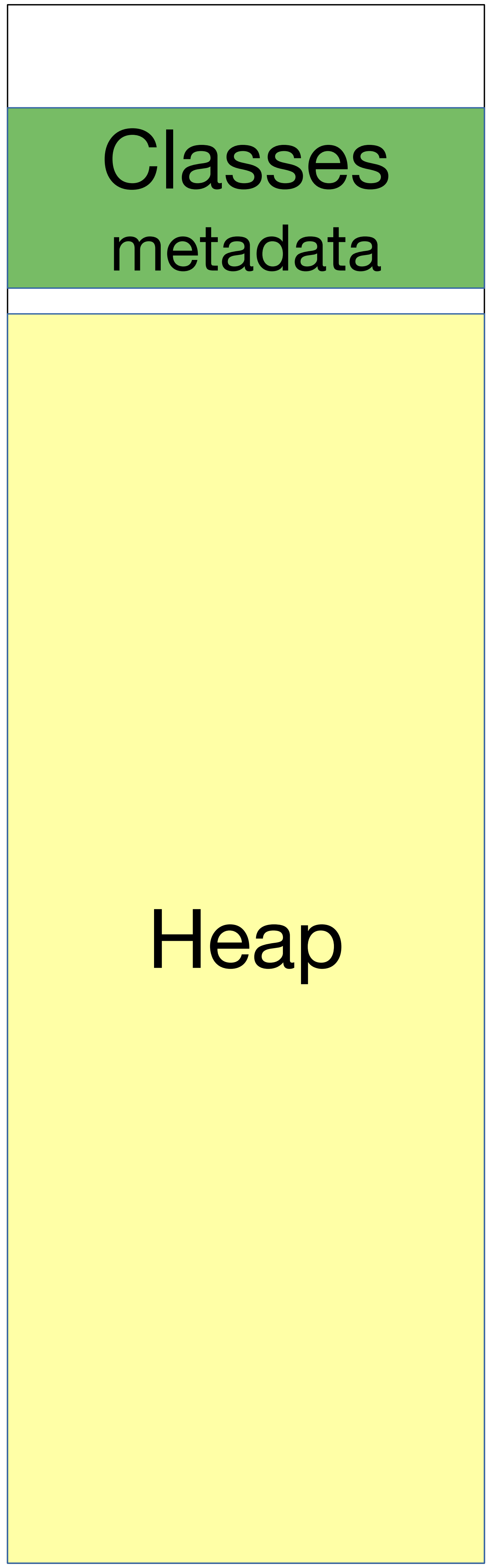
Classes! There is a lot here, generally well covered elsewhere.

Meta-data is harder to see, but we'll start looking at it piece-meal as part of other topics. Just be aware that it exists, and takes up space as part of a Class

Classes & Meta-data

Classes! There is a lot here, generally well covered elsewhere.

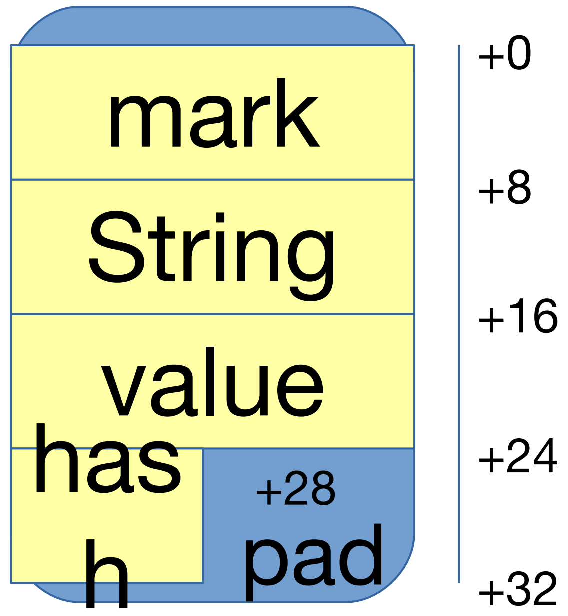
Meta-data is harder to see, but we'll start looking at it piece-meal as part of other topics. Just be aware that it exists, and takes up space as part of a Class but perhaps not part of the JVM Heap.

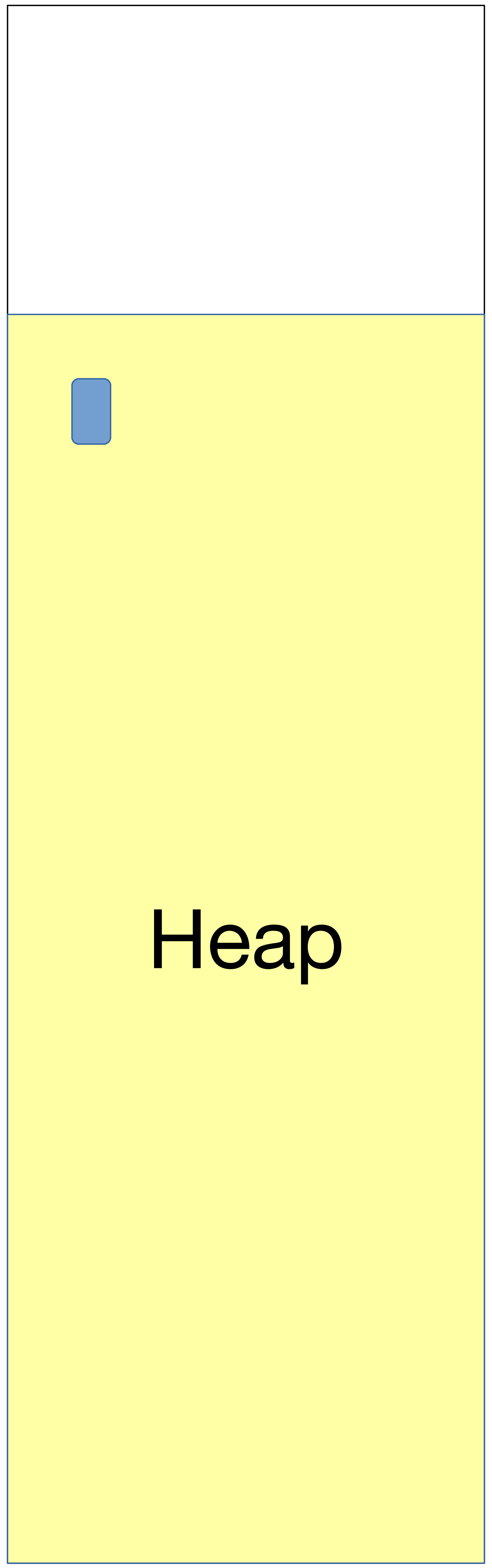


Data and the Heap

Data is stored in Objects, and Objects are stored in the Heap.

Objects are made with ‘**new**’ and reclaimed with **GC**.





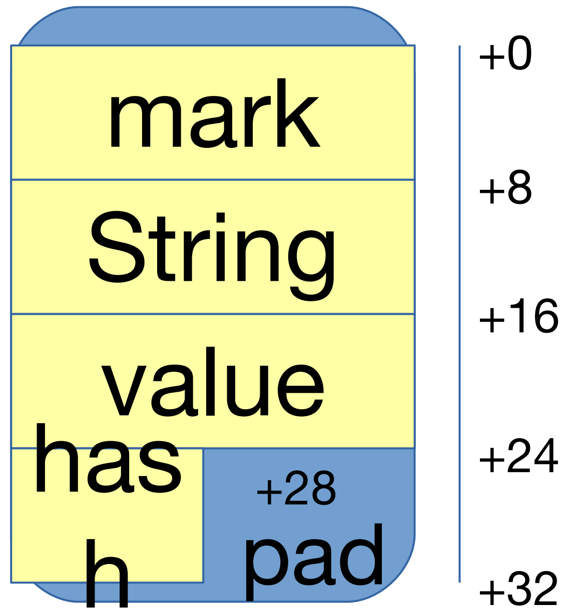
Data and the Heap

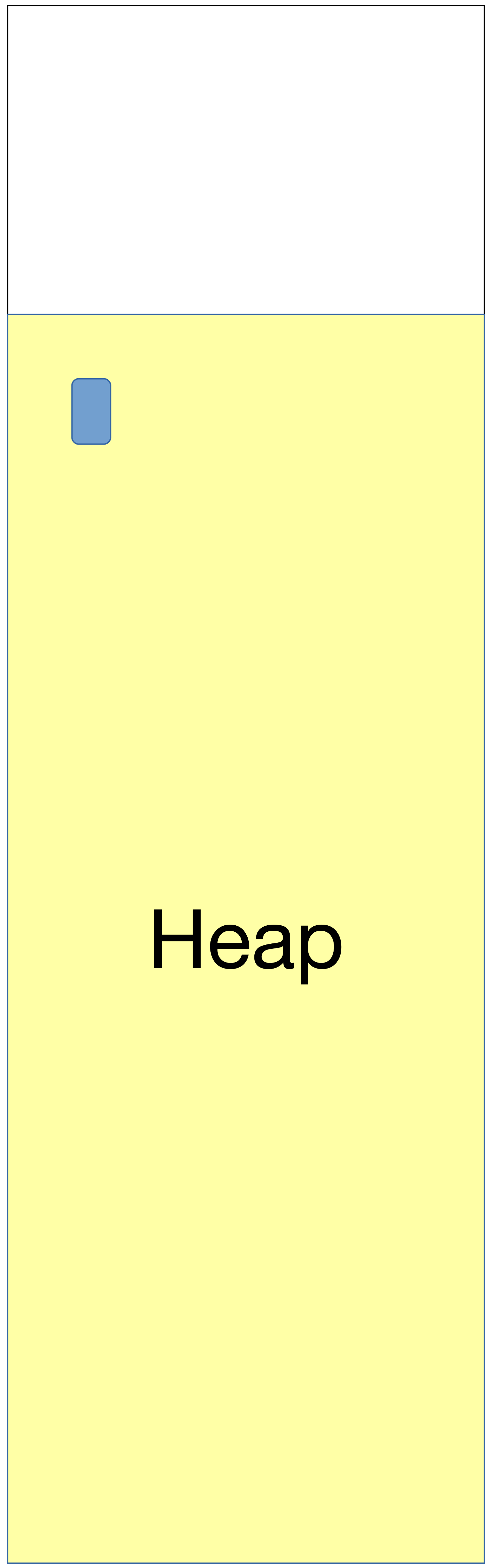
Data is stored in Objects, and Objects are stored in the Heap.

Objects are made with ‘**new**’ and reclaimed with **GC**.

The **Heap** probably is the largest user of your machine resources.

Working with those Objects probably uses most of you dev cycles.





Data and the Heap

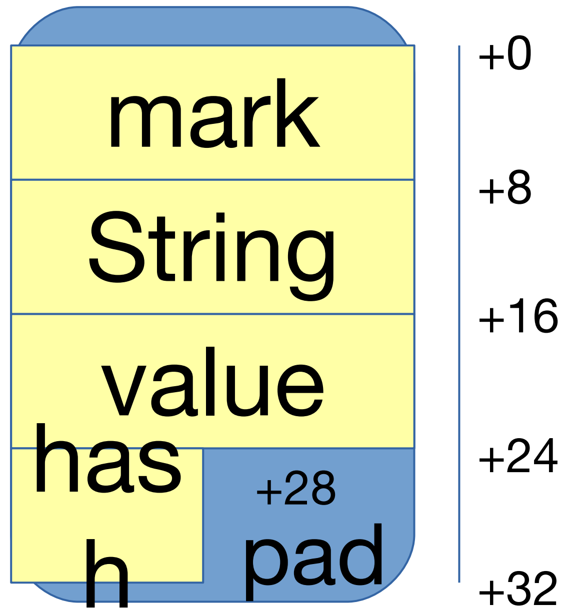
Data is stored in Objects, and Objects are stored in the Heap.

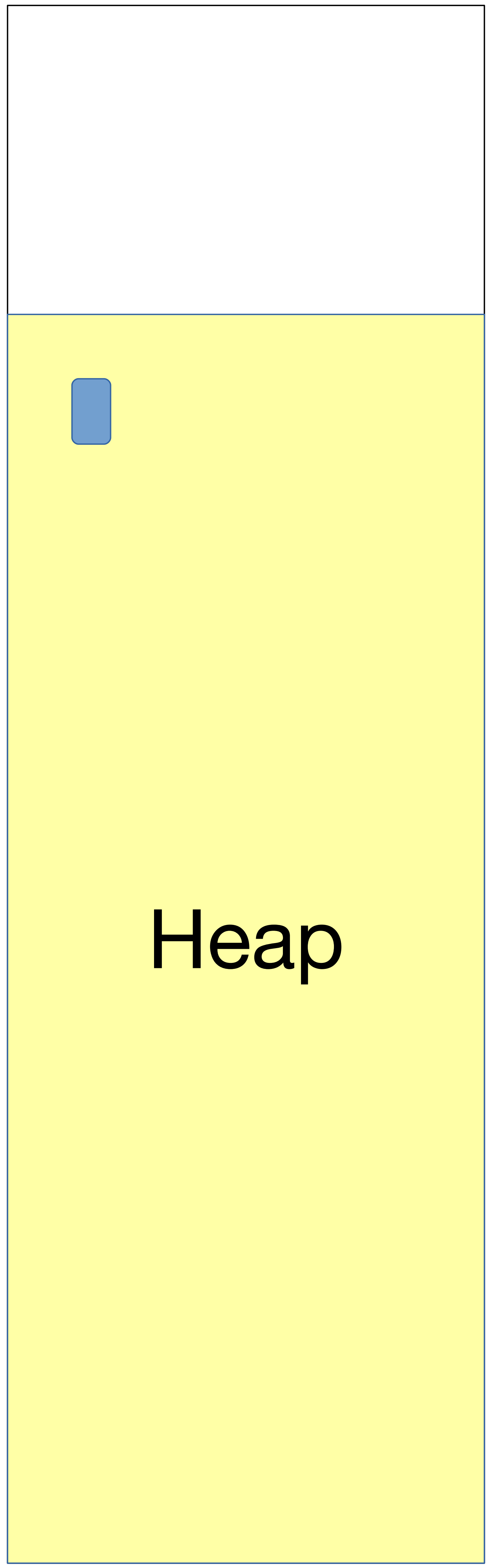
Objects are made with ‘**new**’ and reclaimed with **GC**.

The **Heap** probably is the largest user of your machine resources.

Working with those Objects probably uses most of you dev cycles.

GC tuning is often a huge issue, and can make or break production performance. **GC** tuning is well covered elsewhere…





Data and the Heap

Data is stored in Objects, and Objects are stored in the **Heap**.

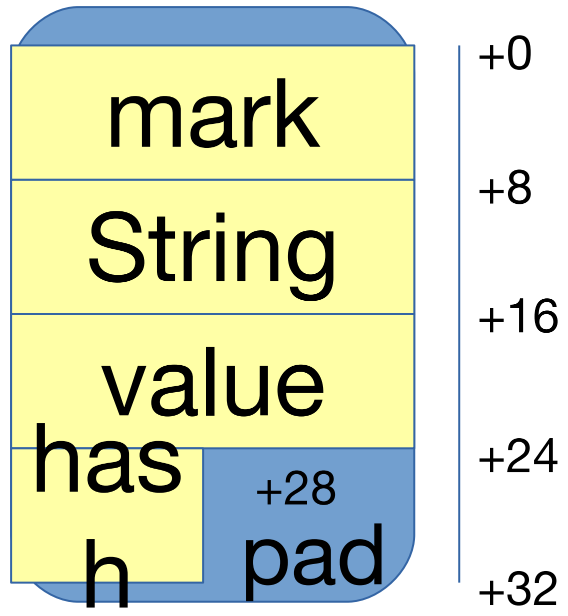
Objects are made with ‘**new**’ and reclaimed with **GC**.

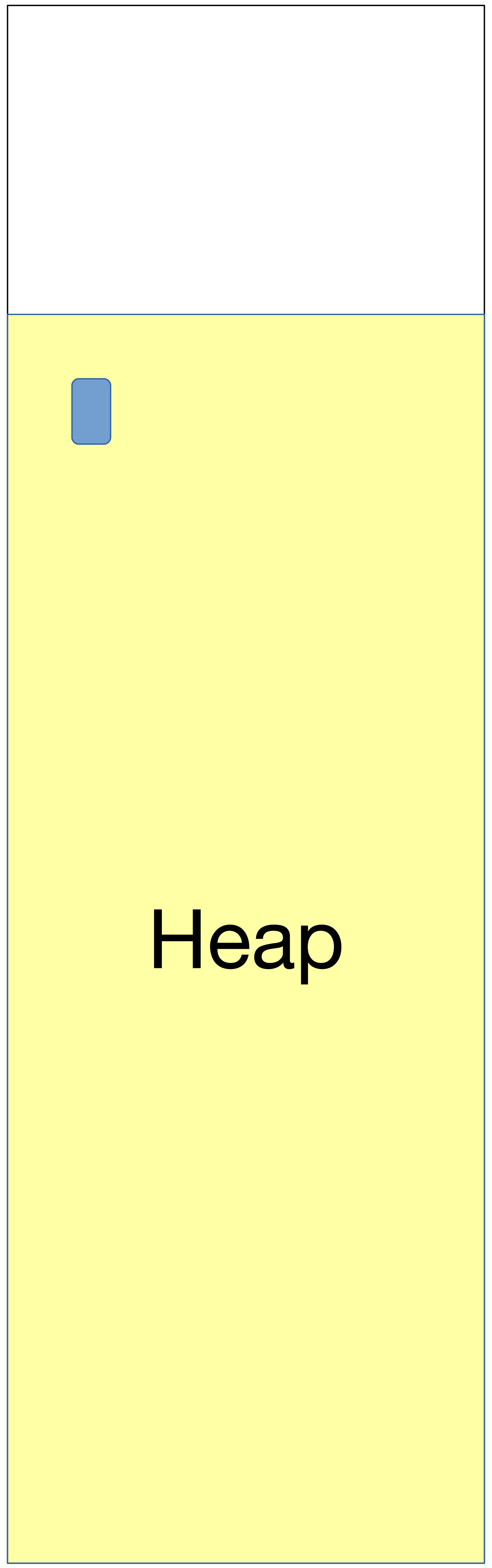
The **Heap** probably is the largest user of your machine resources.

Working with those Objects probably uses most of you dev cycles.

GC tuning is often a huge issue, and can make or break production performance. **GC** tuning is well covered elsewhere…

… so here I’ll just present a high-level view of the interaction between your program and **GC**.

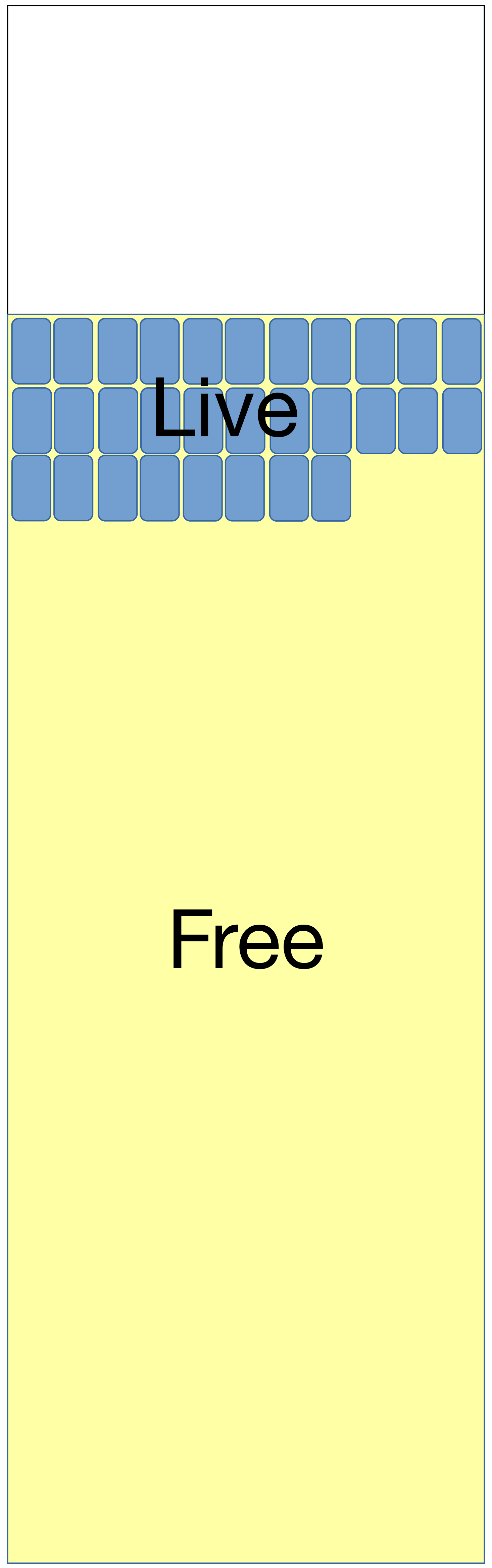




Data and the Heap

The **Heap** is split into Live and Free regions.

Generally Live is further split into generations, not pictured here.

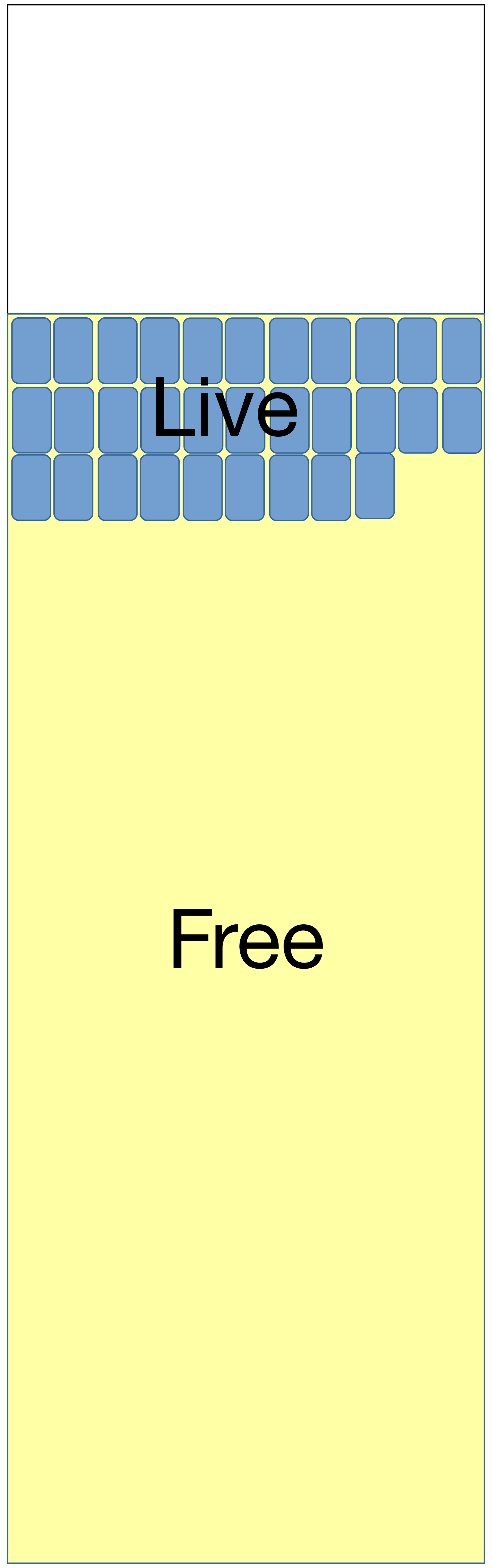


Data and the Heap

The **Heap** is split into Live and Free regions.

Generally Live is further split into generations, not pictured here.

As

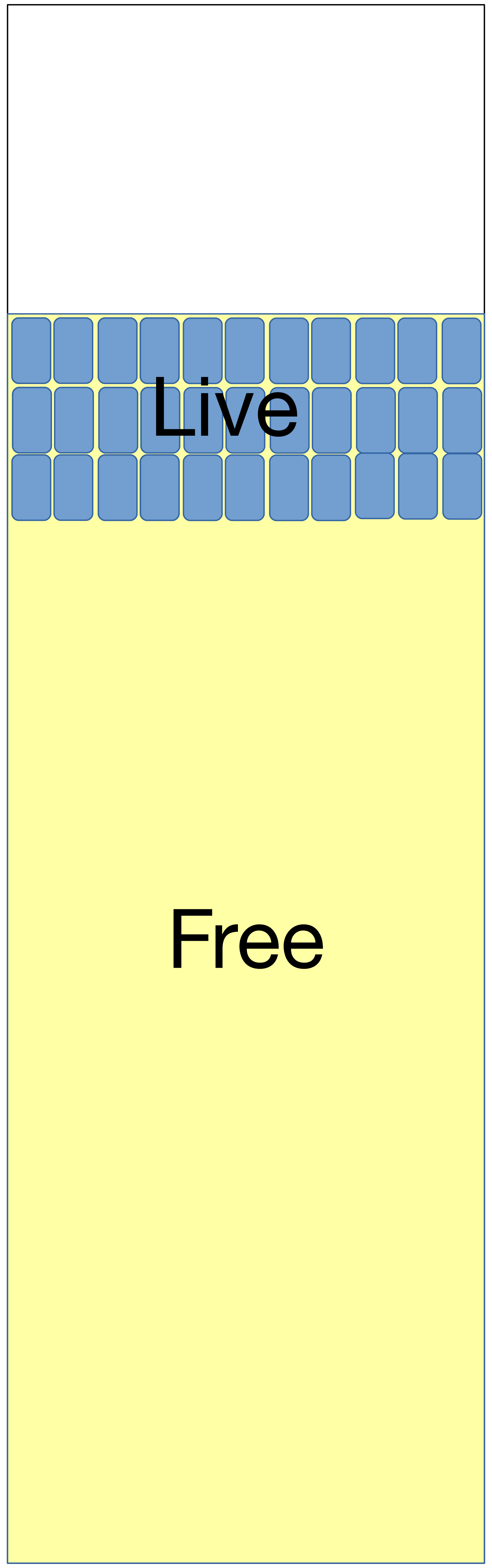


Data and the Heap

The **Heap** is split into Live and Free regions.

Generally Live is further split into generations, not pictured here.

As your

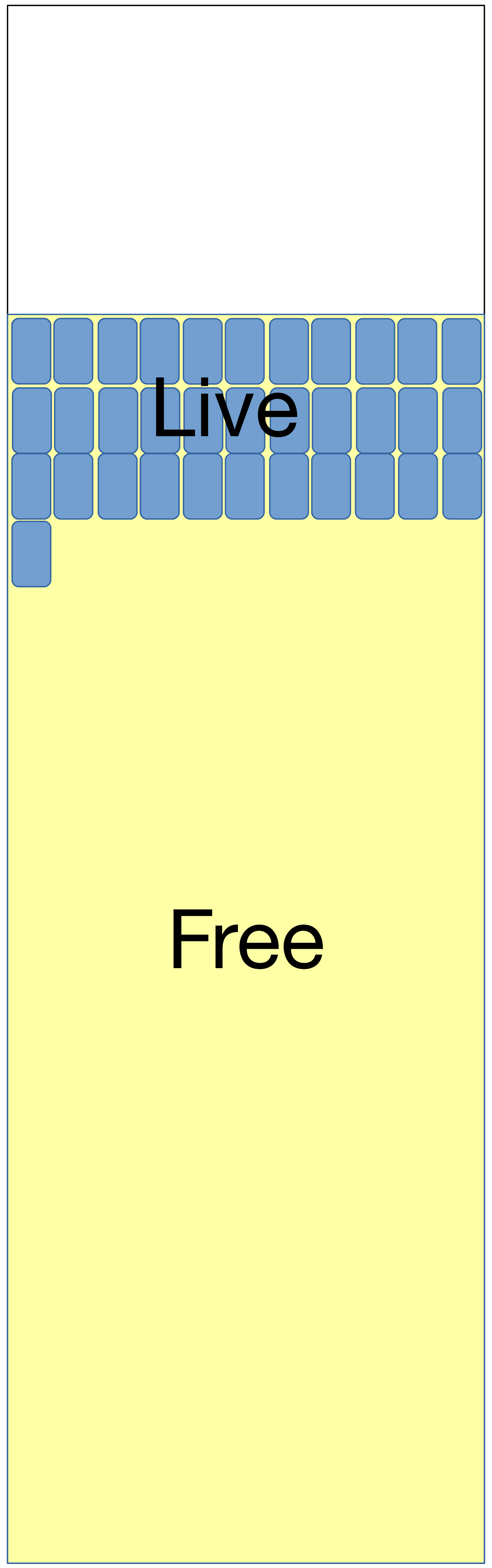


Data and the Heap

The **Heap** is split into Live and Free regions.

Generally Live is further split into generations, not pictured here.

As your program

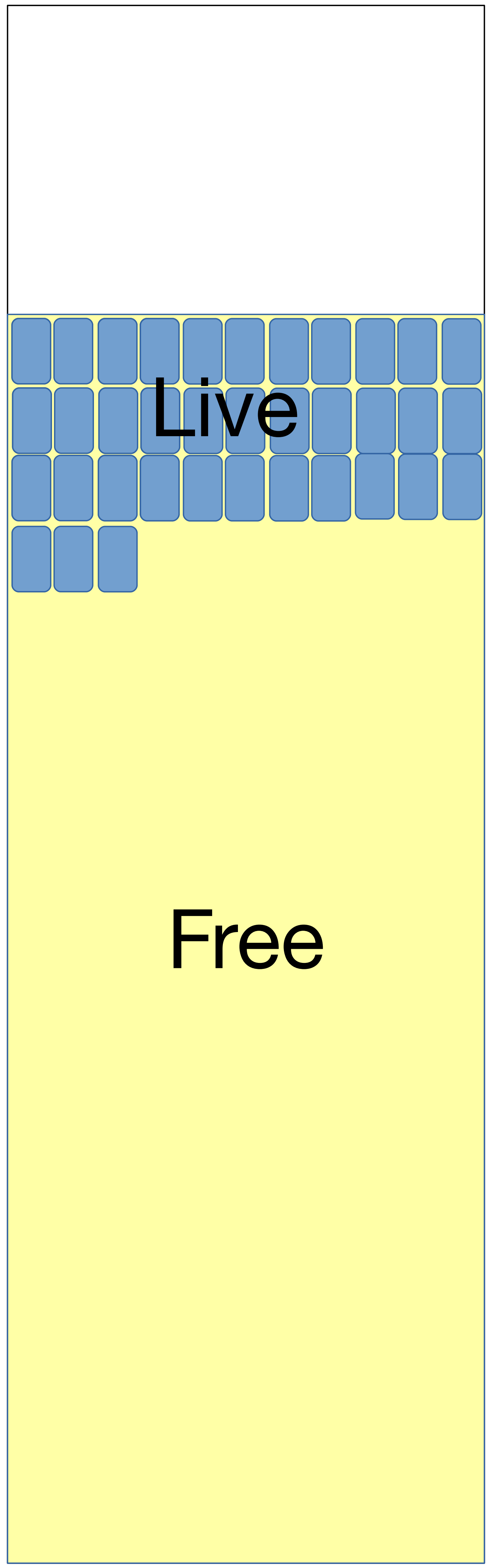


Data and the Heap

The **Heap** is split into Live and Free regions.

Generally Live is further split into generations, not pictured here.

As your program runs,

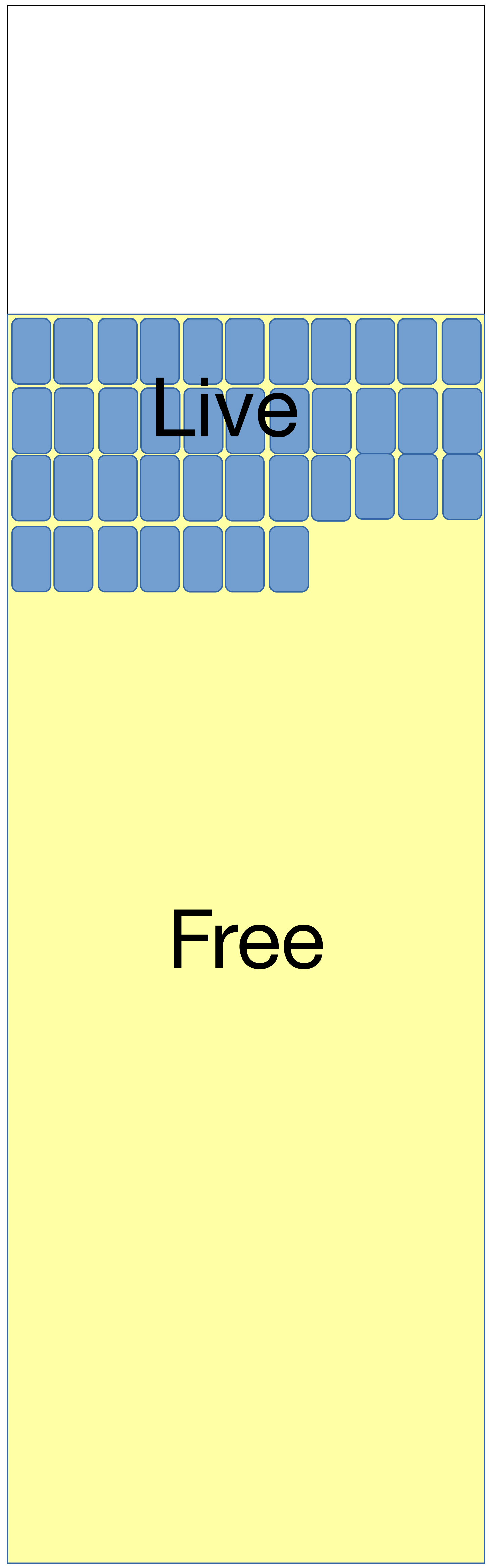


Data and the Heap

The **Heap** is split into Live and Free regions.

Generally Live is further split into generations, not pictured here.

As your program runs, the border shifts between Free and Live.

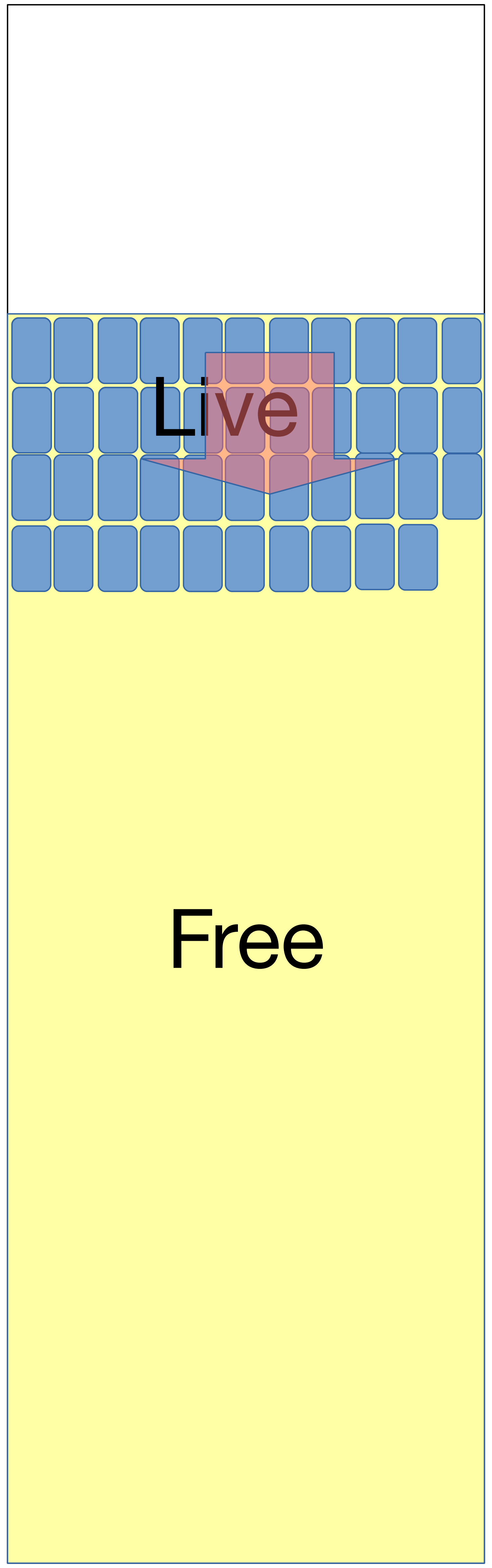


Data and the Heap

The **Heap** is split into Live and Free regions.

Generally Live is further split into generations, not pictured here.

As your program runs, the border shifts between Free and Live.

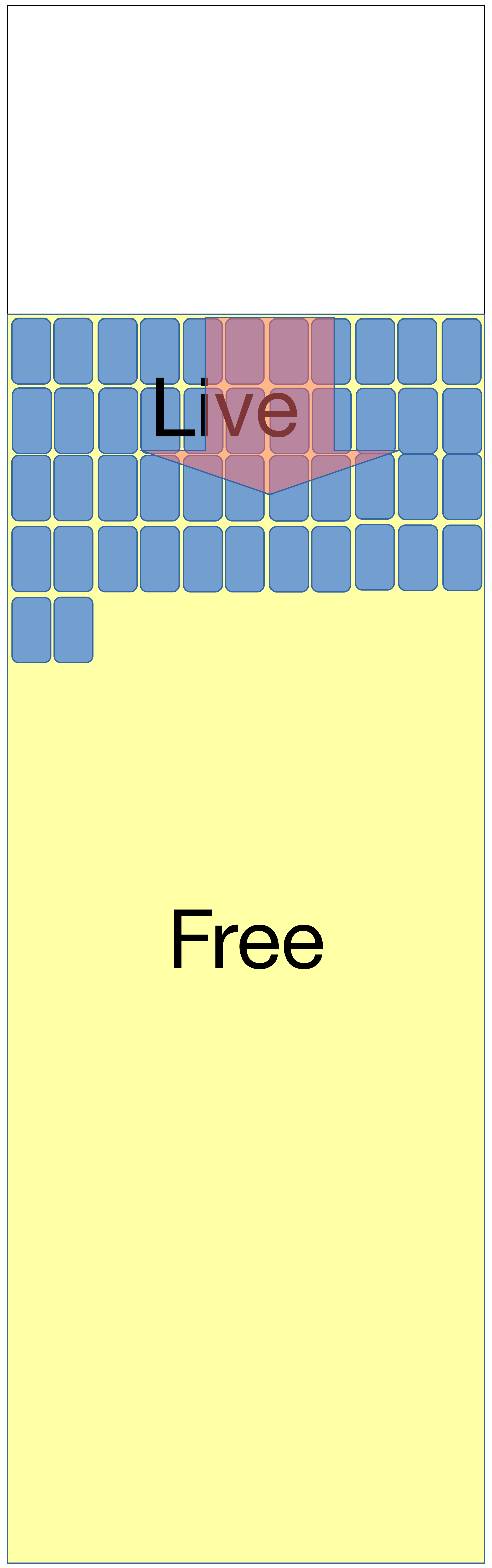


Data and the Heap

The **Heap** is split into Live and Free regions.

Generally Live is further split into generations, not pictured here.

As your program runs, the border shifts between Free and Live.



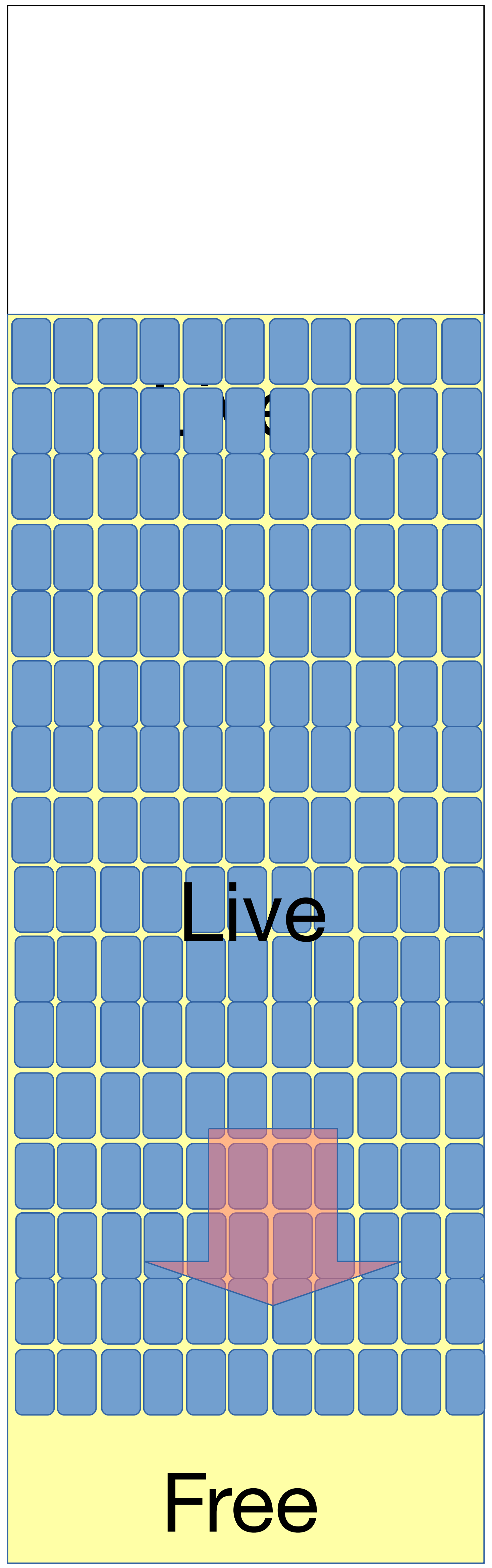
Data and the Heap

The **Heap** is split into Live and Free regions.

Generally Live is further split into generations, not pictured here.

As your program runs, the border shifts between Free and Live.

Until Free runs out.



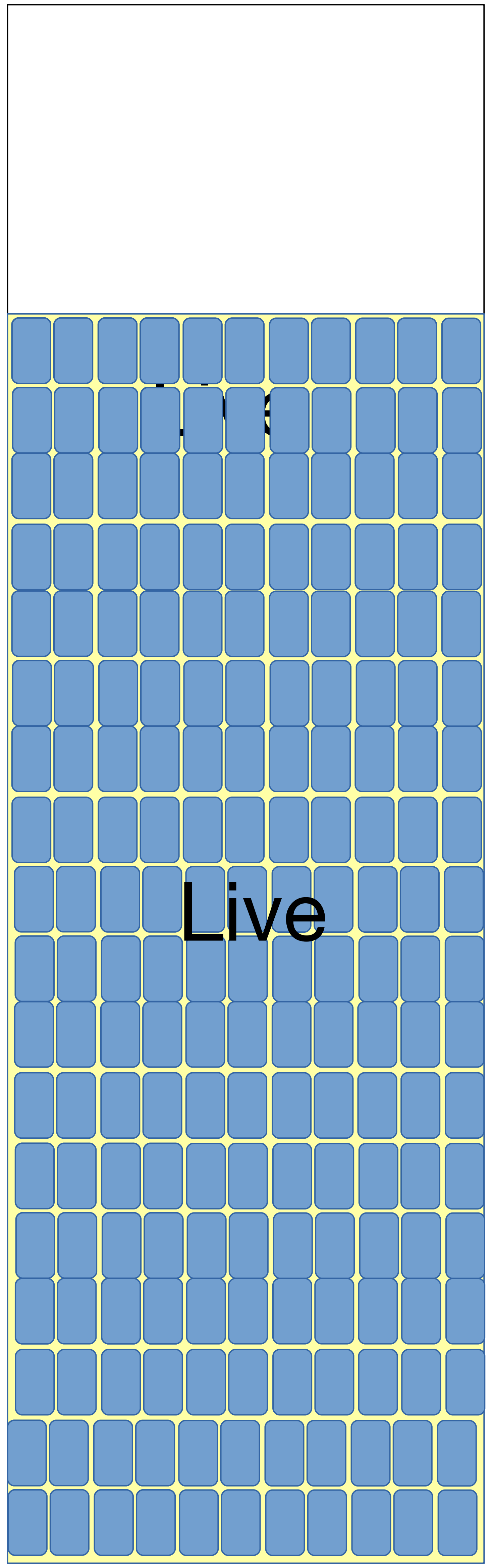
Data and the Heap

The **Heap** is split into Live and Free regions.

Generally Live is further split into generations, not pictured here.

As your program runs, the border shifts between Free and Live.

Until Free runs out.



Data and the Heap

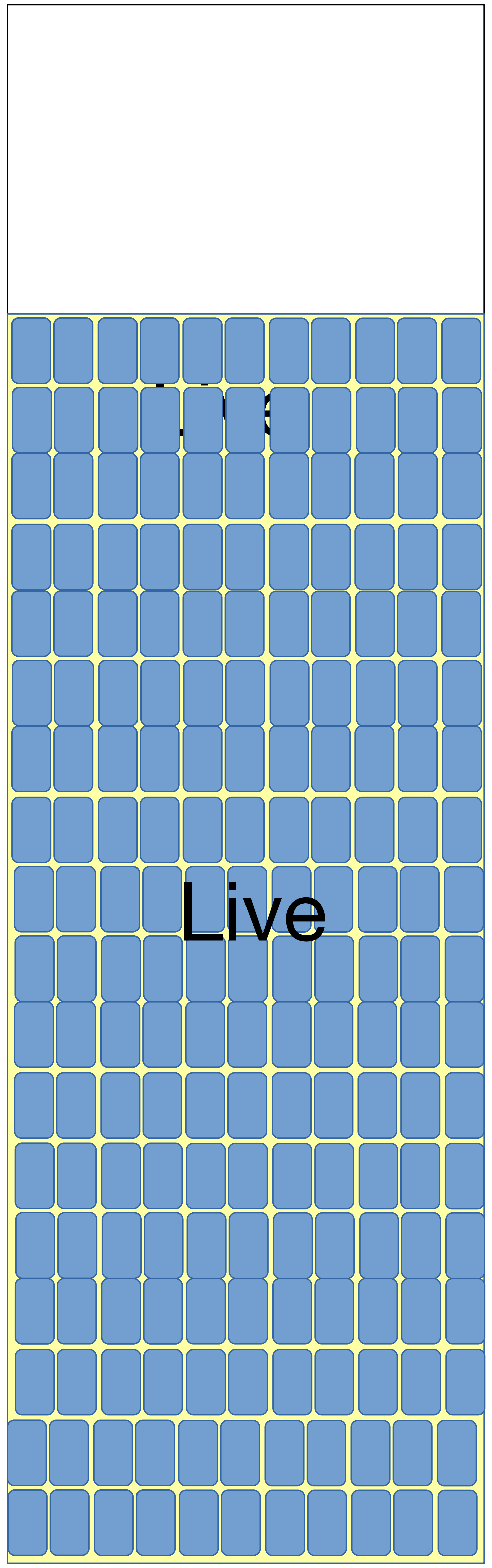
The **Heap** is split into Live and Free regions.

Generally Live is further split into generations, not pictured here.

As your program runs, the border shifts between Free and Live.

Until Free runs out.

GC “reverses” this process,



Data and the Heap

The **Heap** is split into Live and Free regions.

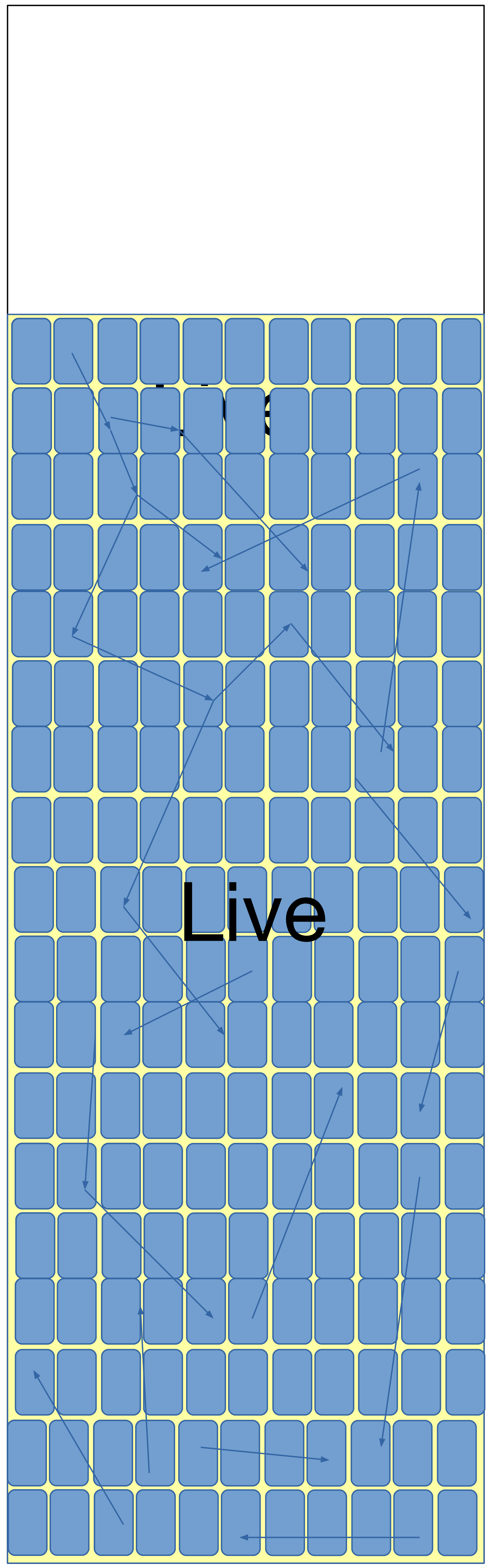
Generally Live is further split into generations, not pictured here.

As your program runs, the border shifts between Free and Live.

Until Free runs out.

GC “reverses” this process,

by following all the pointers to see what can be reached.



Data and the Heap

The **Heap** is split into Live and Free regions.

Generally Live is further split into generations, not pictured here.

As your program runs, the border shifts between Free and Live.

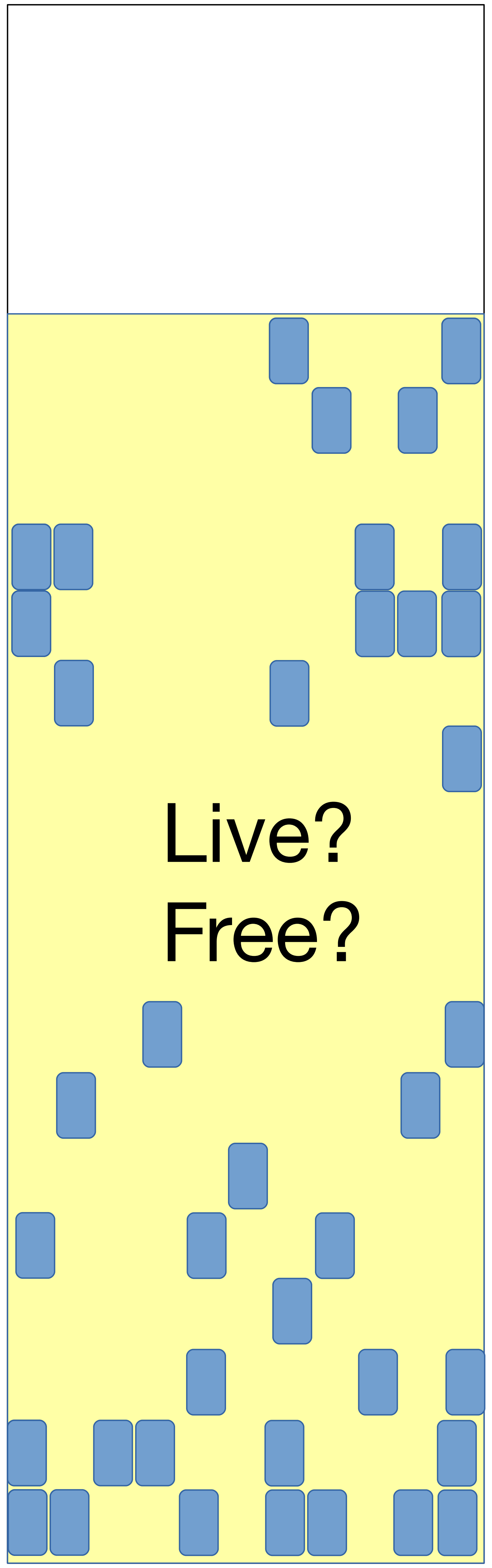
Until Free runs out.

GC “reverses” this process,

by following all the pointers to see what can be reached,

and freeing the unreachable.

This **fragments** memory.



Data and the Heap

The **Heap** is split into Live and Free regions.

Generally Live is further split into generations, not pictured here.

As your program runs, the border shifts between Free and Live.

Until Free runs out.

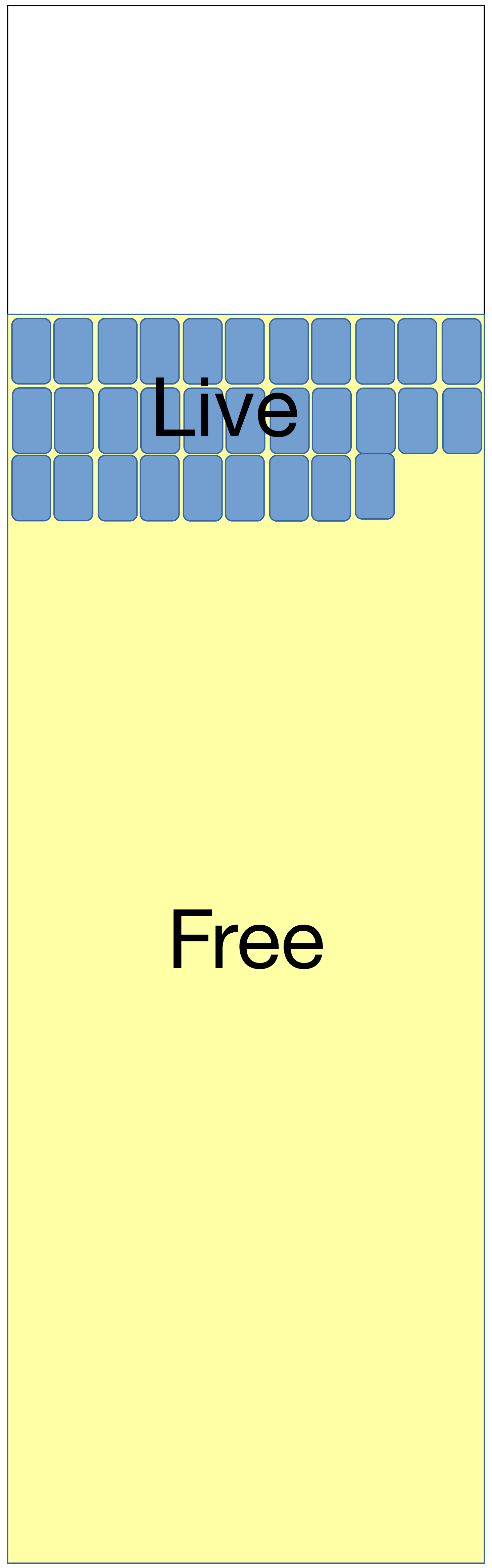
GC “reverses” this process,

by following all the pointers to see what can be reached,

and freeing the unreachable.

This **fragments** memory.

So **GC** generally moves objects to compact them,



Data and the Heap

The **Heap** is split into Live and Free regions.

Generally Live is further split into generations, not pictured here.

As your program runs, the border shifts between Free and Live.

Until Free runs out.

GC “reverses” this process,

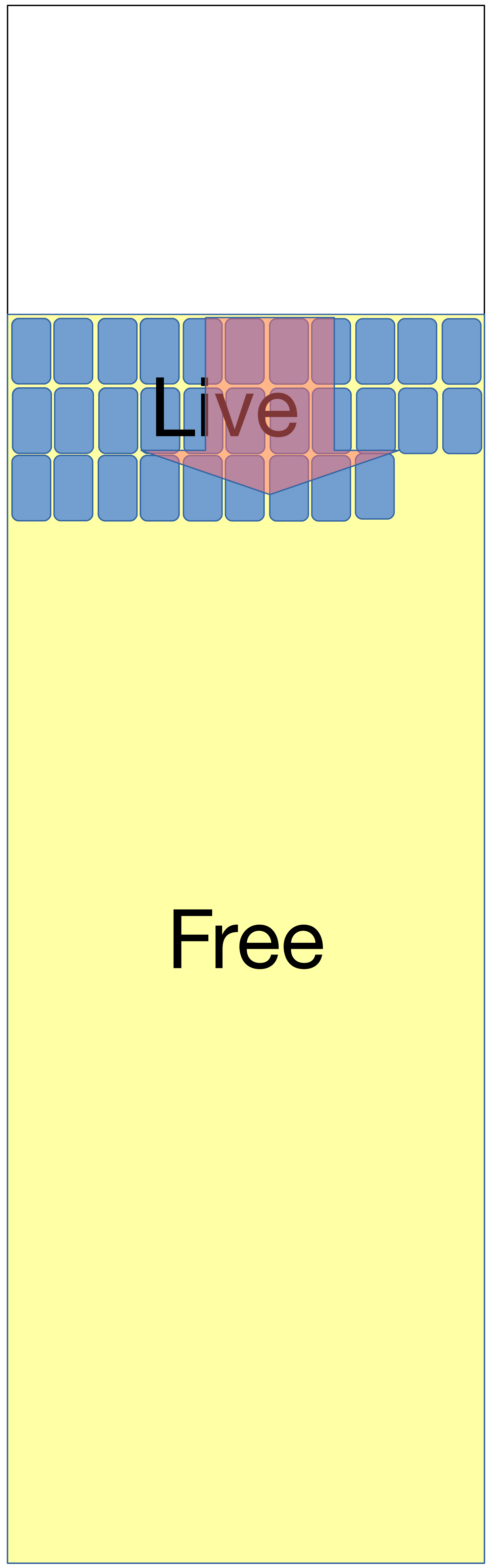
by following all the pointers to see what can be reached,

and freeing the unreachable.

This **fragments** memory.

So **GC** generally moves objects to compact them,

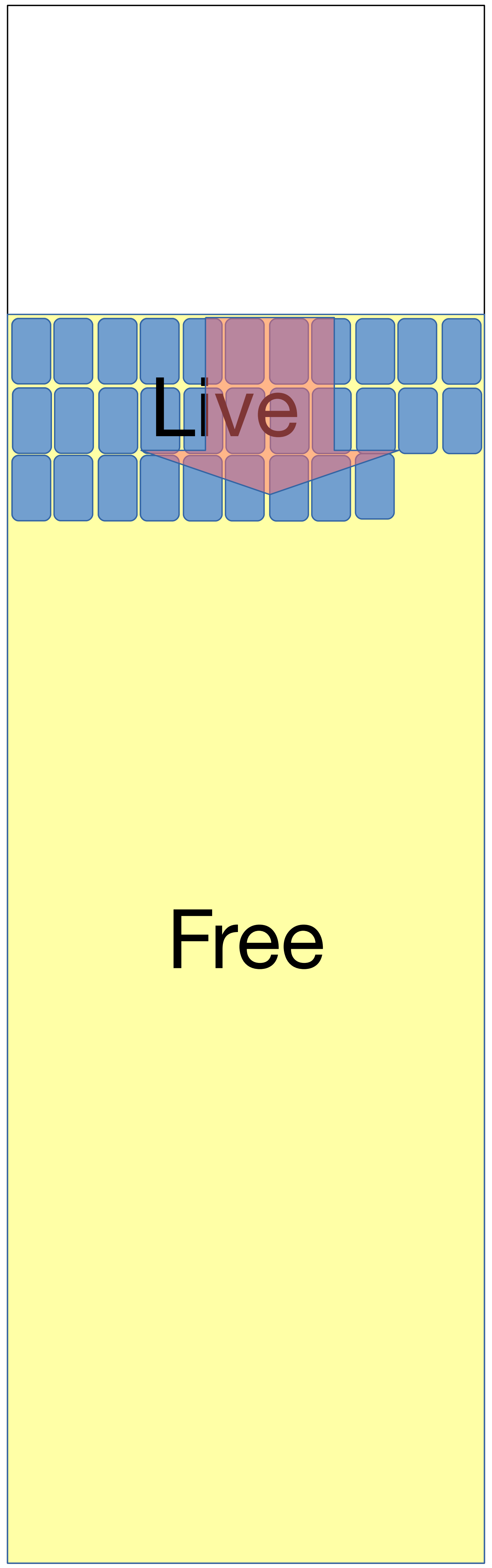
And the cycle repeats.



Data and the Heap

This process is discrete in detail, but statistical in bulk.

Similar to fluid flow, we can **up-level the discussion** from individual allocations to looking at the **allocation rate**.

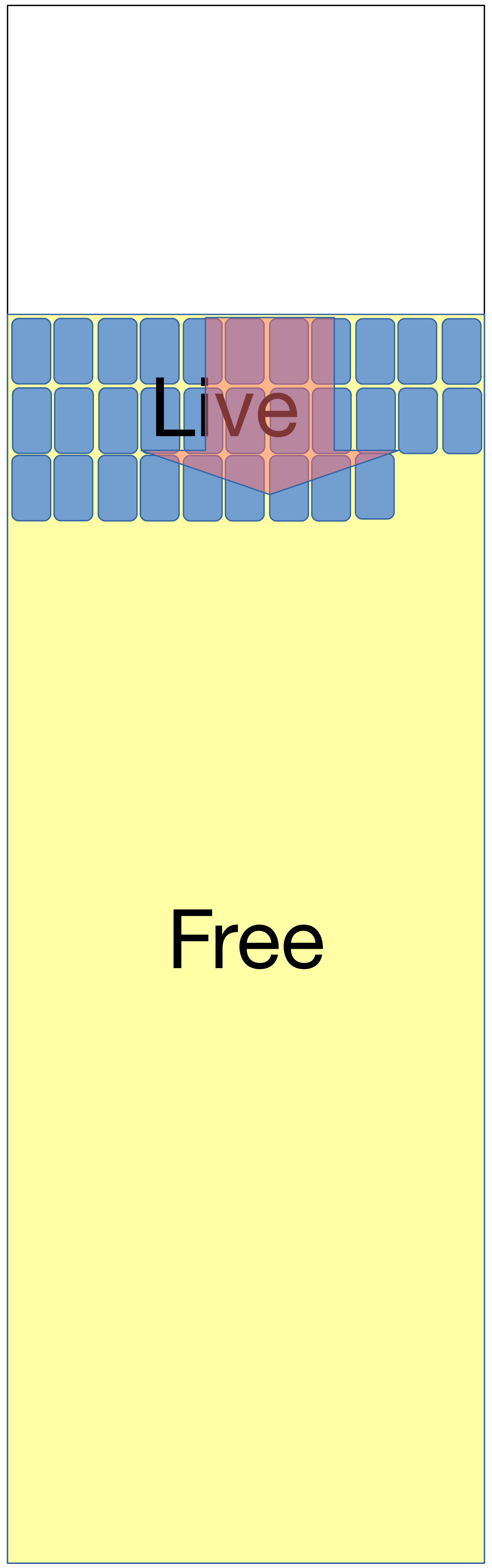


Data and the Heap

This process is discrete in detail, but statistical in bulk.

Similar to fluid flow, we can **up-level the discussion** from individual allocations to looking at the **allocation rate**.

The mutator (Java program) “pumps” memory from Free to Live.



Data and the Heap

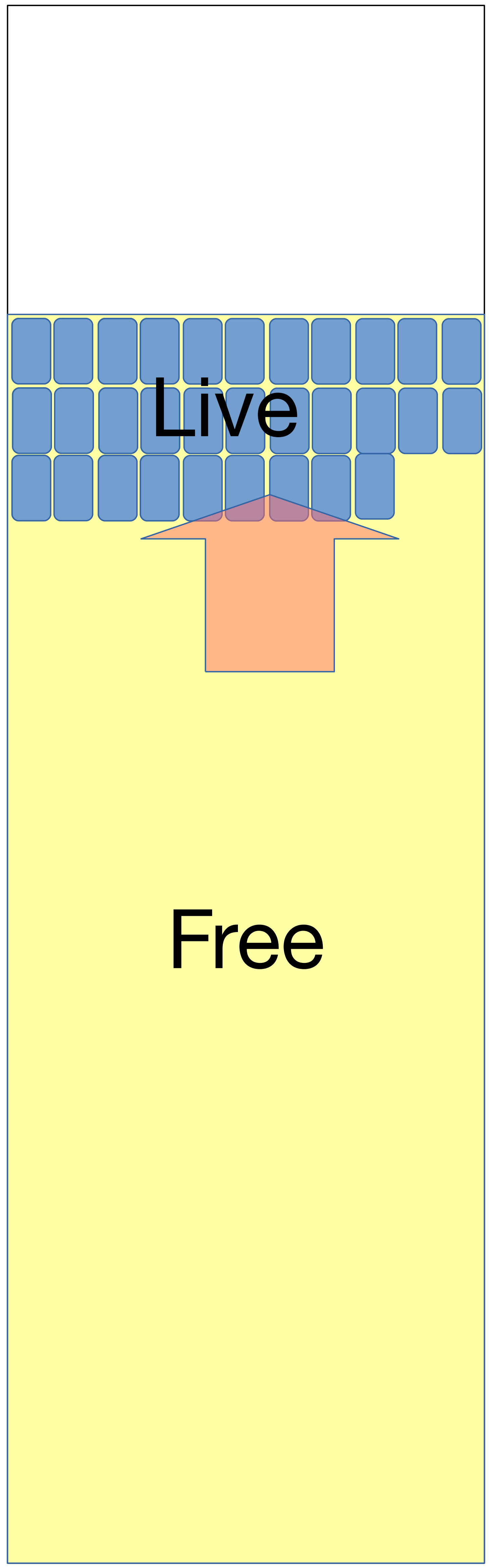
This process is discrete in detail, but statistical in bulk.

Similar to fluid flow, we can **up-level the discussion** from individual allocations to looking at the **allocation rate**.

The mutator (Java program) “pumps” memory from Free to Live.

GC “pumps” memory from Live to Free.

These two forces **must** balance out (or you die Out of Memory),



Data and the Heap

This process is discrete in detail, but statistical in bulk.

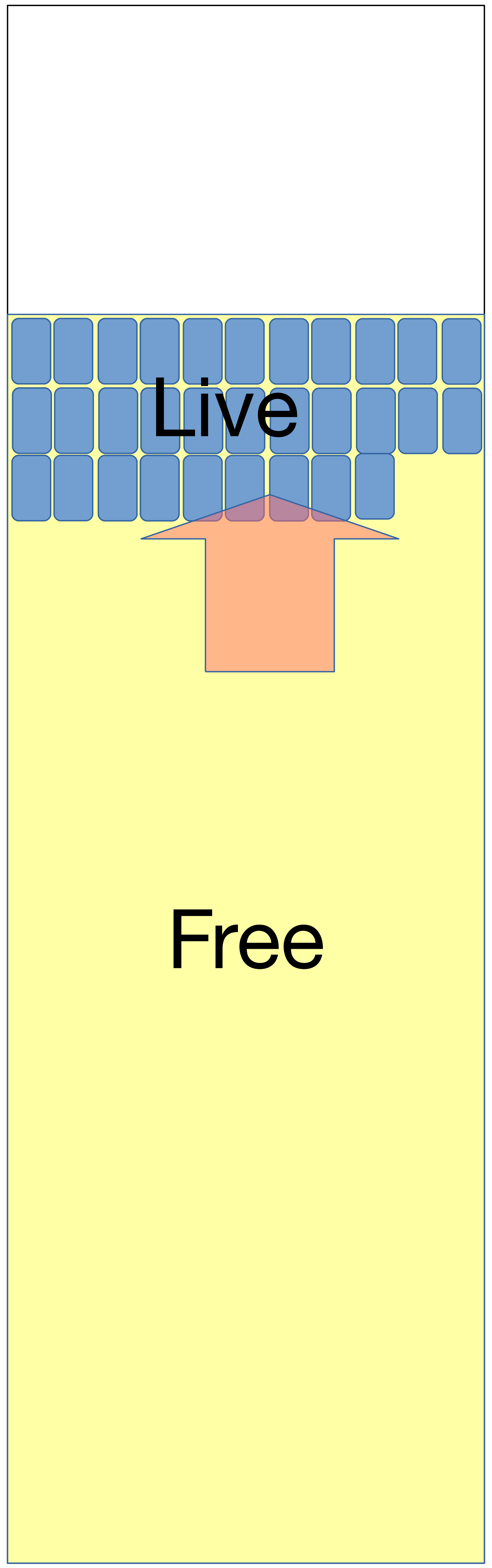
Similar to fluid flow, we can **up-level the discussion** from individual allocations to looking at the **allocation rate**.

The mutator (Java program) “pumps” memory from Free to Live.

GC “pumps” memory from Live to Free.

These two forces **must** balance out (or you die Out of Memory),

But they have different CPU costs.



Data and the Heap

This process is discrete in detail, but statistical in bulk.

Similar to fluid flow, we can **up-level the discussion** from individual allocations to looking at the **allocation rate**.

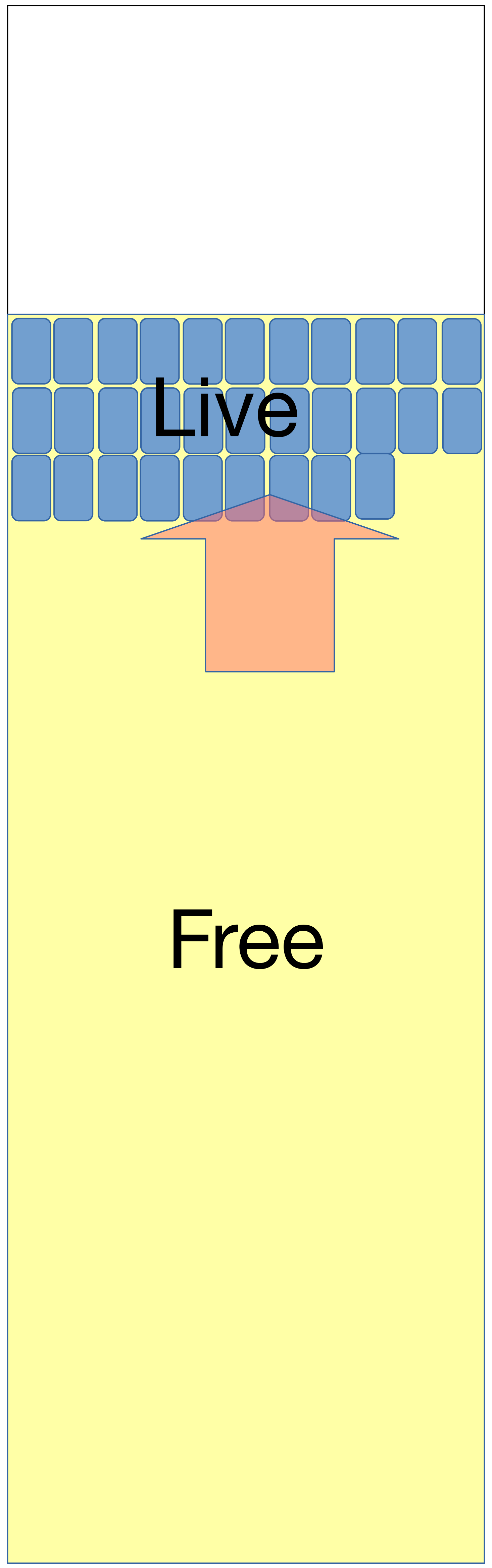
The mutator (Java program) “pumps” memory from Free to Live.

GC “pumps” memory from Live to Free.

These two forces **must** balance out (or you die Out of Memory),

But they have different CPU costs.

The allocation rate (and CPU cost) is up to the developer.



Data and the Heap

This process is discrete in detail, but statistical in bulk.

Similar to fluid flow, we can **up-level the discussion** from individual allocations to looking at the **allocation rate**.

The mutator (Java program) “pumps” memory from Free to Live.

GC “pumps” memory from Live to Free.

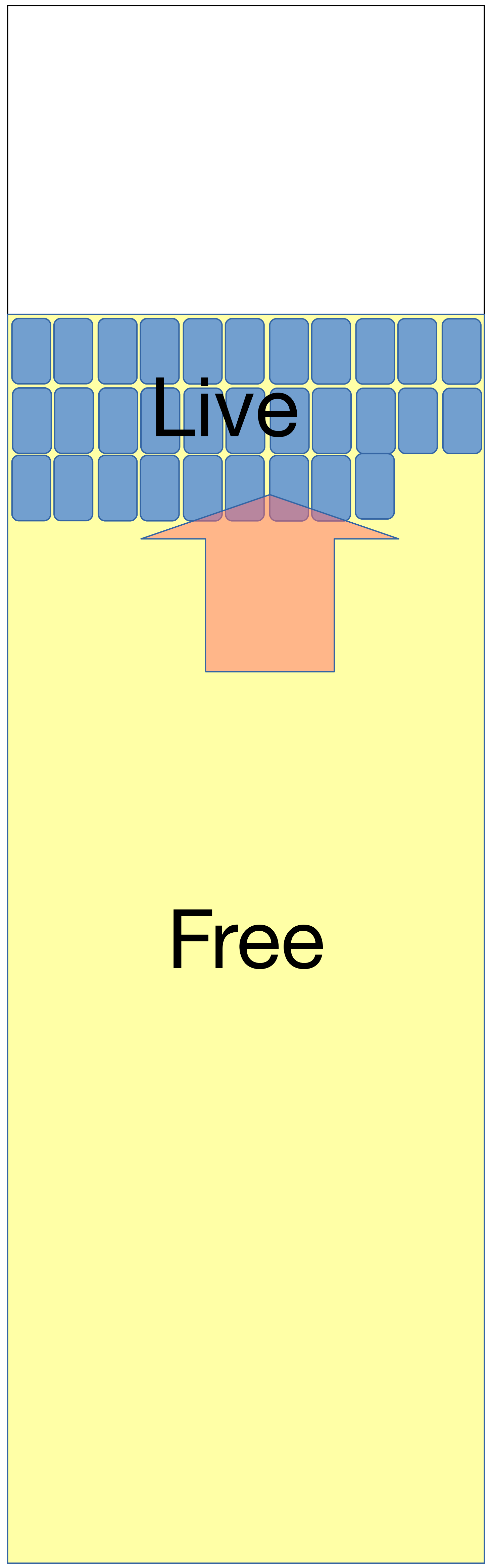
These two forces **must** balance out (or you die Out of Memory),

But they have different CPU costs.

The allocation rate (and CPU cost) is up to the developer.

The GC costs depend on the **structure of the heap**,

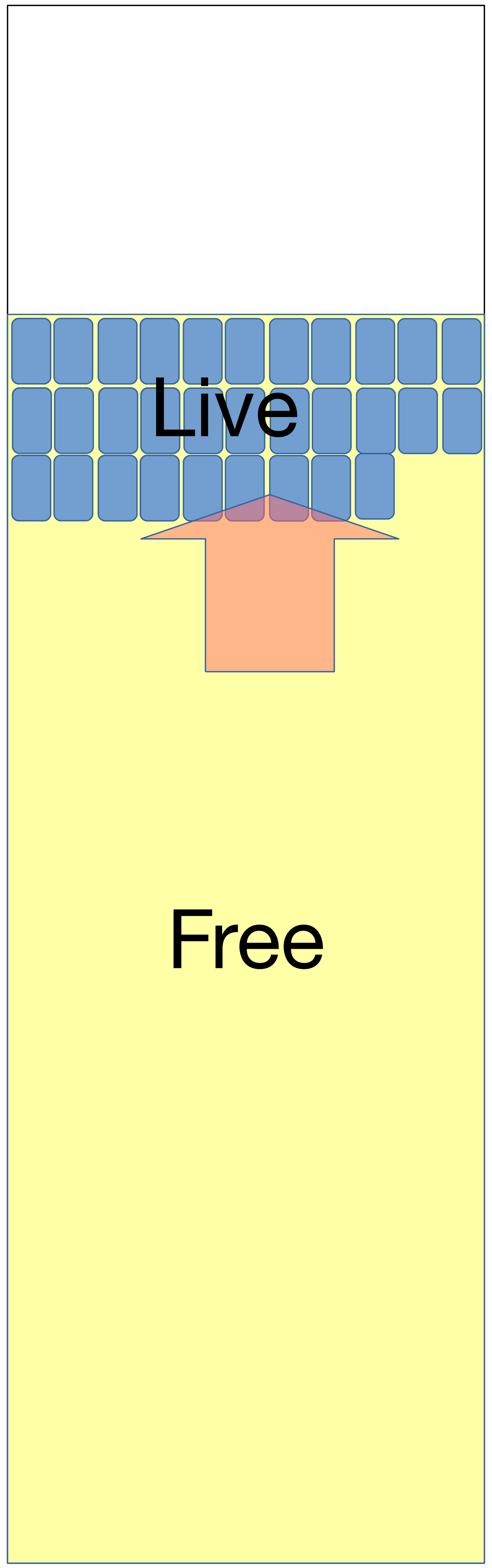
and that structure is also up to the developer.



Data and the Heap – GC Costs

All GC algorithms share some costs in common:

- A per-live-object cost
- A per-live-pointer cost
- A per-byte cost



Data and the Heap – GC Costs

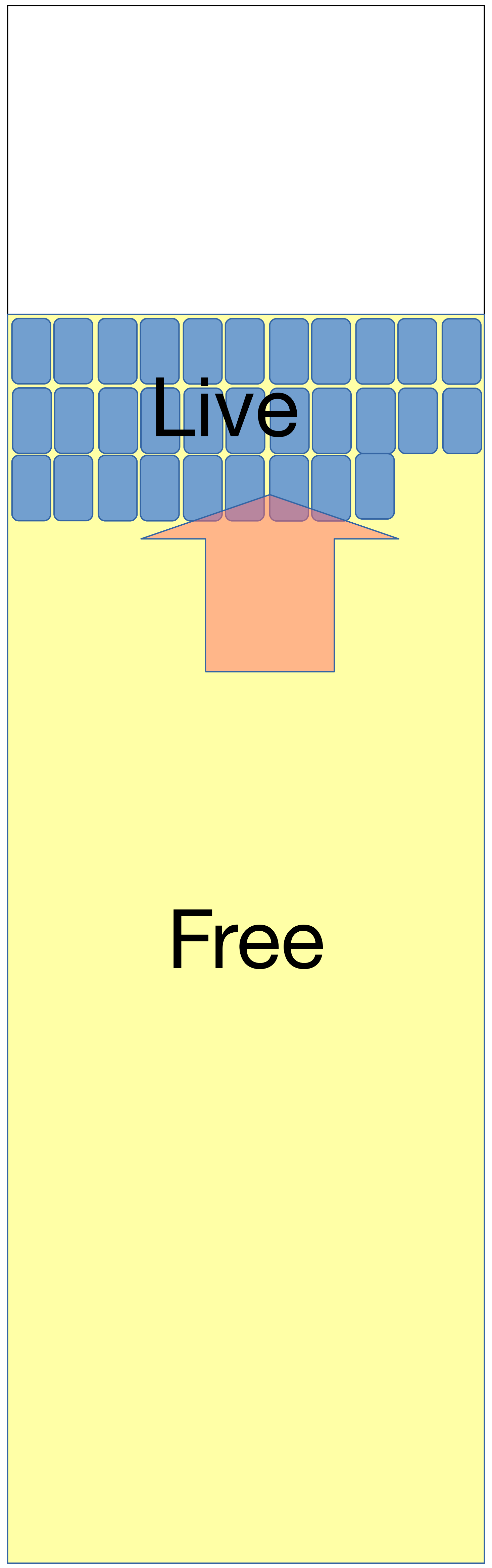
All GC algorithms share some costs in common:

- A per-live-object cost
- A per-live-pointer cost
- A per-byte cost

Mostly the per-byte costs can be very low, but not so the other costs.

A large count of objects or a large count of pointers is more expensive for **ALL** GC algorithms.

Parallel, incremental or concurrent GCs spread the cost around in different ways, with different constant factors.



Data and the Heap – GC Costs

All GC algorithms share some costs in common:

- A per-live-object cost
- A per-live-pointer cost
- A per-byte cost

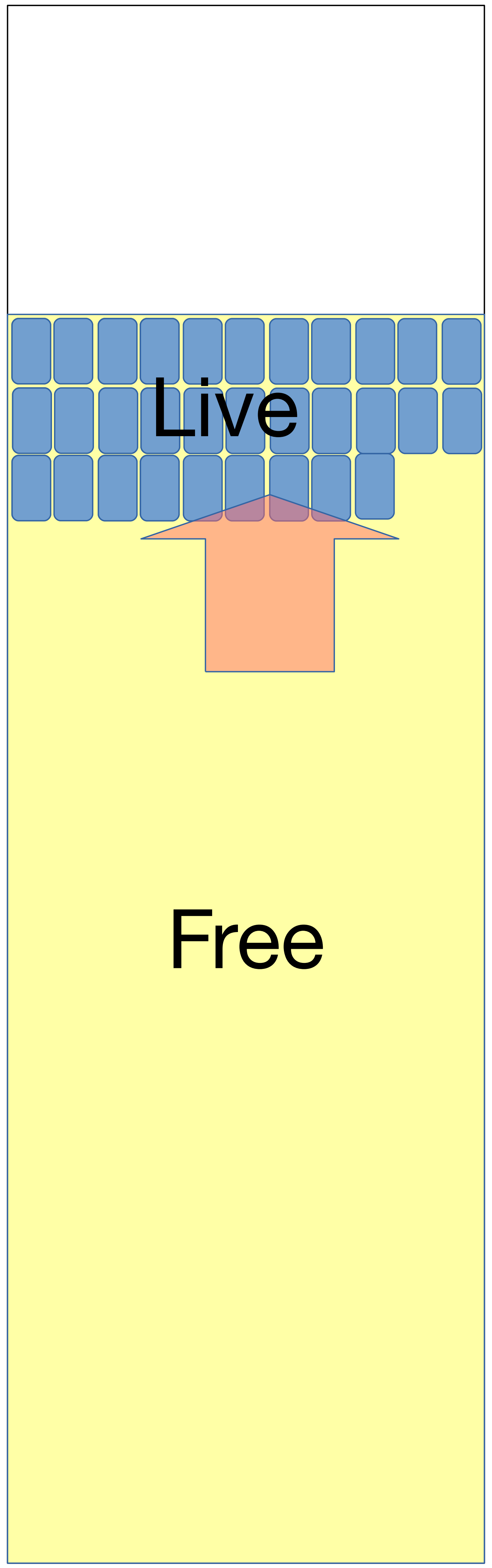
Mostly the per-byte costs can be very low, but not so the other costs.

A large count of objects or a large count of pointers is more expensive for **ALL** GC algorithms.

Parallel, incremental or concurrent GCs spread the cost around in different ways, with different constant factors.

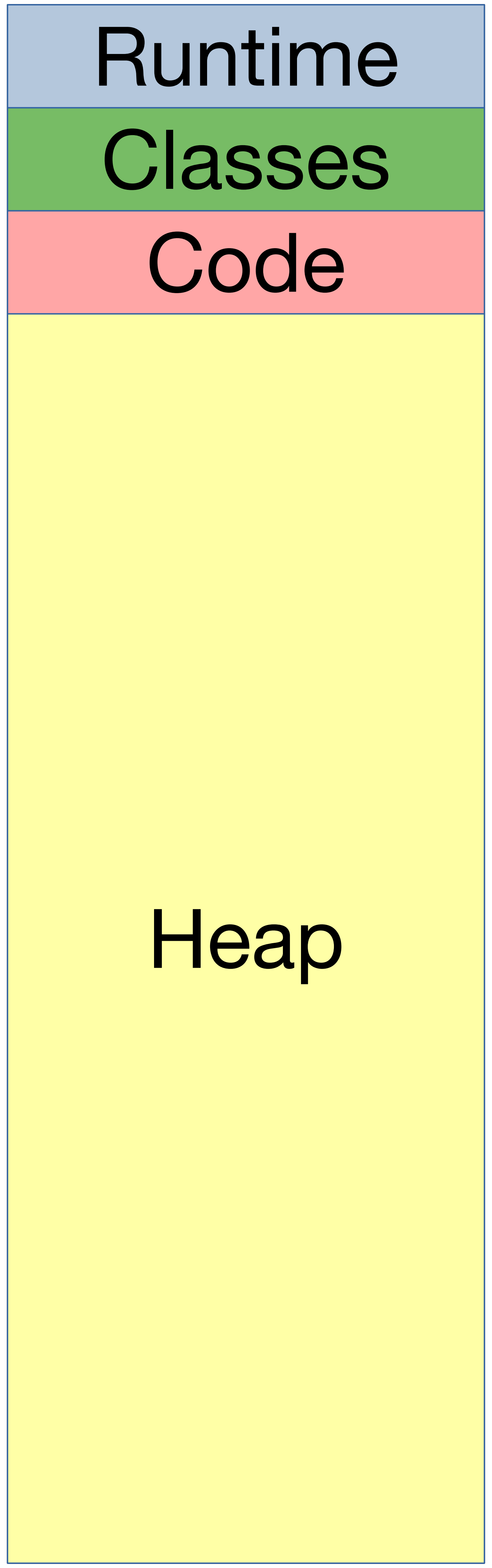
But always, fewer objects & pointers cost less than more of either.

A million **`Long`**s are hugely more expensive than a single **`long[1000000]`**



Bytecodes and an Execution Model

JVMs run JVM Bytecodes – not machine code.



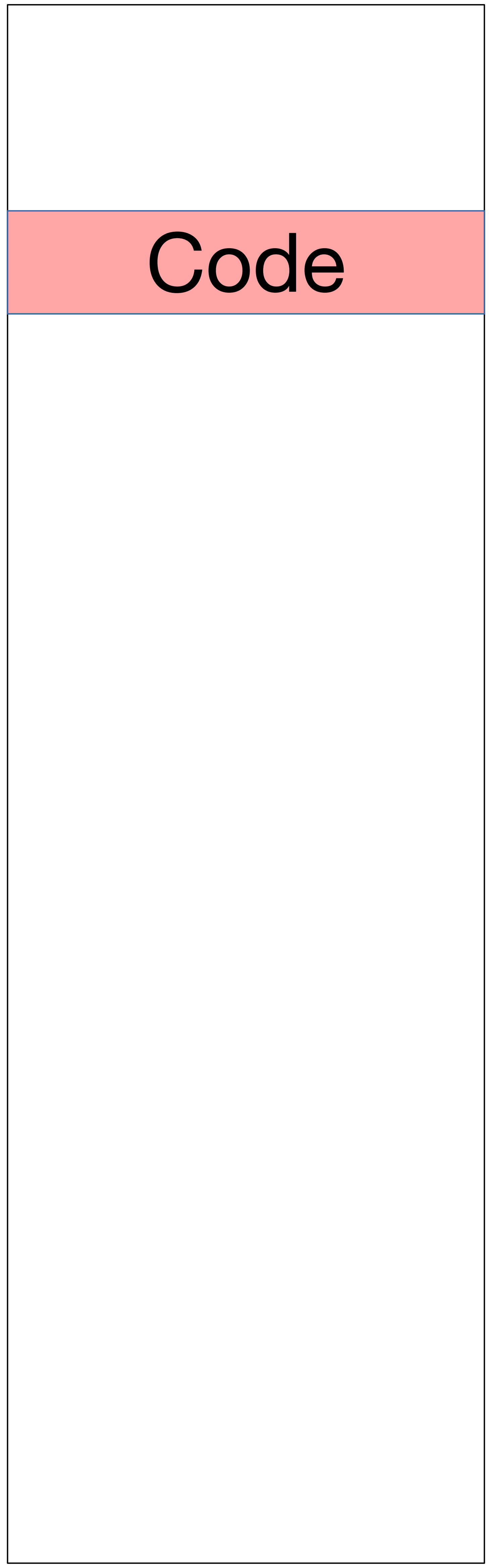
Bytecodes and an Execution Model

JVMs run JVM Bytecodes – not machine code.

No hardware runs bytecodes directly,

So they are emulated on real hardware (e.g. X86 or ARM chips).

This emulation is called an **execution model**.



Bytecodes and an Execution Model

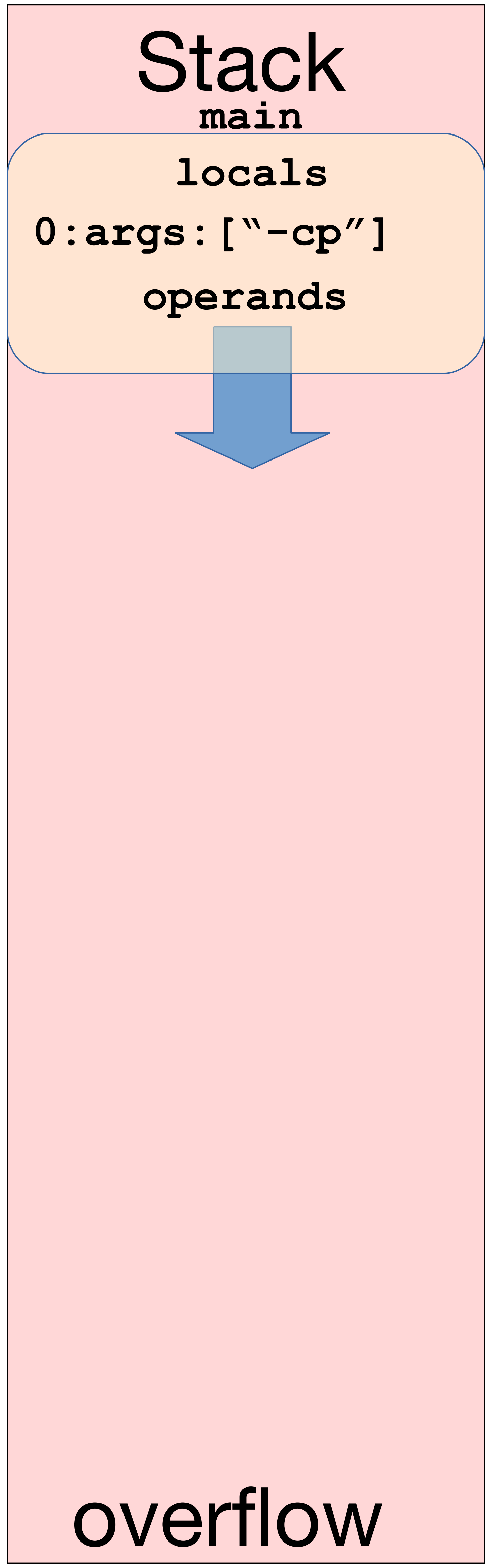
JVMs run JVM Bytecodes – not machine code.

No hardware runs bytecodes directly,

So they are emulated on real hardware (e.g. X86 or ARM chips).

This emulation is called an **execution model**.

The JVM execution model is a **stack machine** with registers.



Bytecodes and an Execution Model

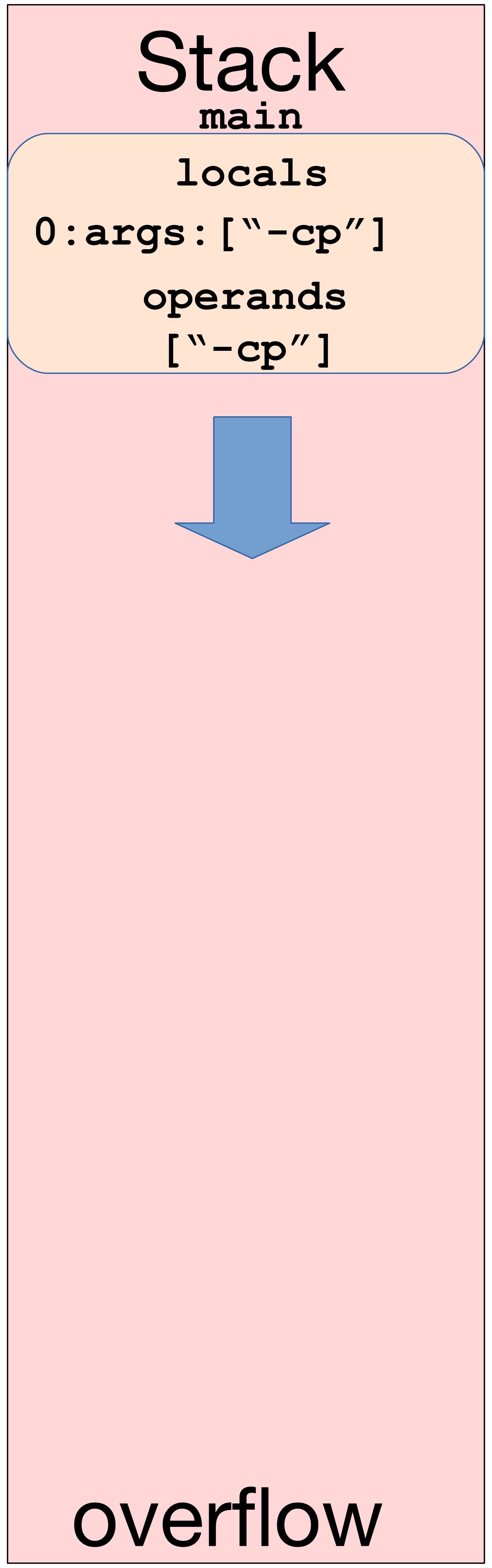
JVMs run JVM Bytecodes – not machine code.

No hardware runs bytecodes directly,

So they are emulated on real hardware (e.g. X86 or ARM chips).

This emulation is called an **execution model**.

The JVM execution model is a **stack machine** with registers.



Bytecodes and an Execution Model

JVMs run JVM Bytecodes – not machine code.

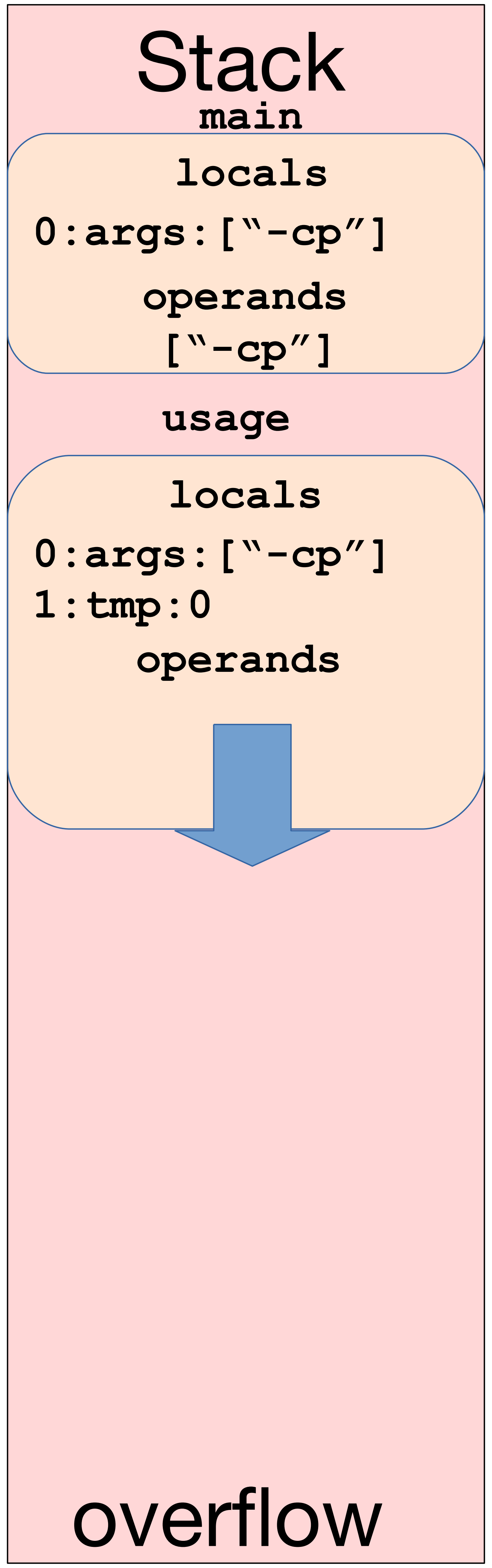
No hardware runs bytecodes directly,

So they are emulated on real hardware (e.g. X86 or ARM chips).

This emulation is called an **execution model**.

The JVM execution model is a **stack machine** with registers.

A Java function call pushes a **frame**, with some **local variables** (including call arguments) and a **operand** stack.



Bytecodes and an Execution Model

JVMs run JVM Bytecodes – not machine code.

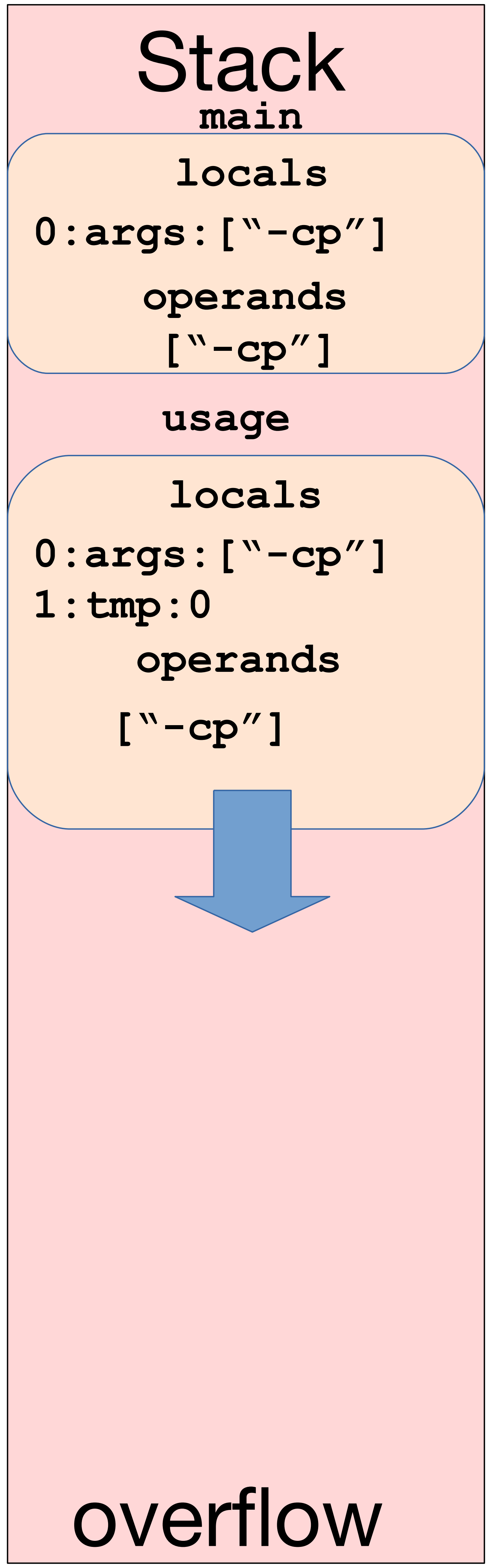
No hardware runs bytecodes directly,

So they are emulated on real hardware (e.g. X86 or ARM chips).

This emulation is called an **execution model**.

The JVM execution model is a **stack machine** with registers.

A Java function call pushes a **frame**, with some **local variables** (including call arguments) and a **operand** stack. Bytecodes push and pop values between registers and stack



Bytecodes and an Execution Model

JVMs run JVM Bytecodes – not machine code.

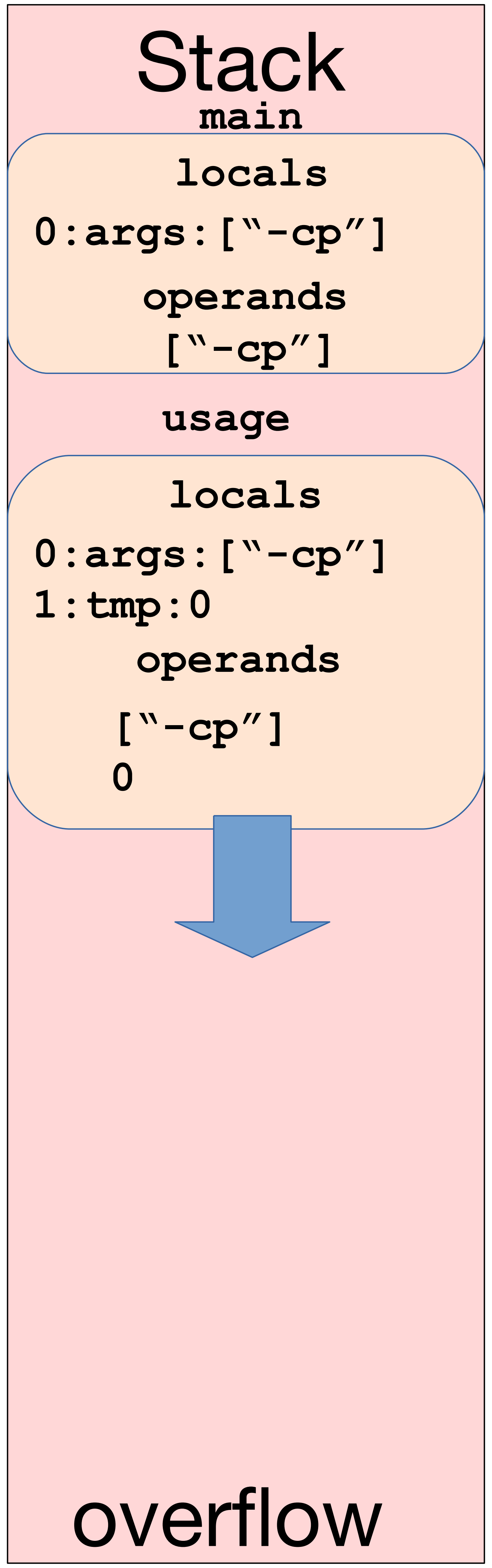
No hardware runs bytecodes directly,

So they are emulated on real hardware (e.g. X86 or ARM chips).

This emulation is called an **execution model**.

The JVM execution model is a **stack machine** with registers.

A Java function call pushes a **frame**, with some **local variables** (including call arguments) and a **operand** stack. Bytecodes push and pop values between registers and stack,



Bytecodes and an Execution Model

JVMs run JVM Bytecodes – not machine code.

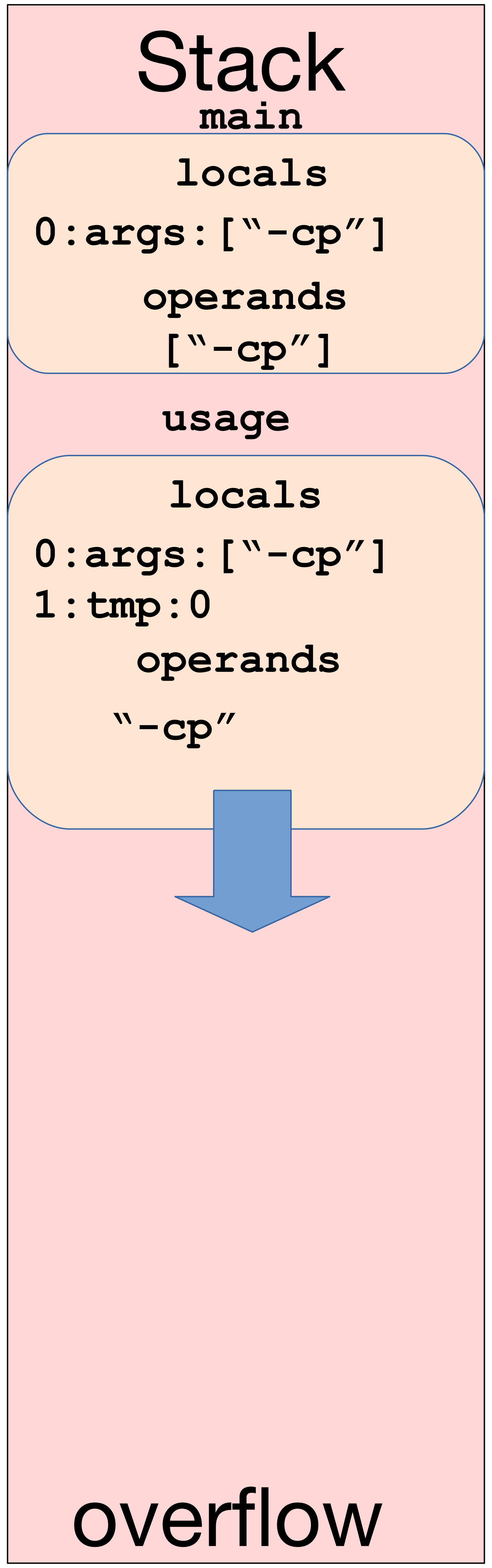
No hardware runs bytecodes directly,

So they are emulated on real hardware (e.g. X86 or ARM chips).

This emulation is called an **execution model**.

The JVM execution model is a **stack machine** with registers.

A Java function call pushes a **frame**, with some **local variables** (including call arguments) and a **operand** stack. Bytecodes push and pop values between registers and stack, or compute on the stack.



Bytecodes and an Execution Model

JVMs run JVM Bytecodes – not machine code.

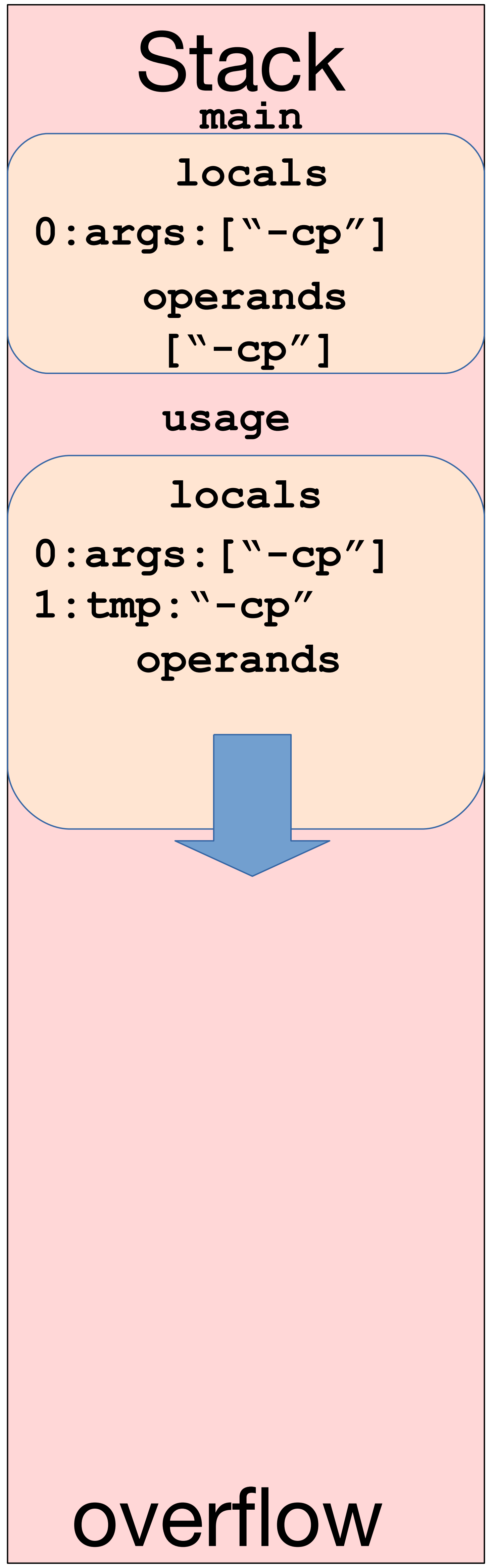
No hardware runs bytecodes directly,

So they are emulated on real hardware (e.g. X86 or ARM chips).

This emulation is called an **execution model**.

The JVM execution model is a **stack machine** with registers.

A Java function call pushes a **frame**, with some **local variables** (including call arguments) and a **operand** stack. Bytecodes push and pop values between registers and stack, or compute on the stack.



Bytecodes and an Execution Model

JVMs run JVM Bytecodes – not machine code.

No hardware runs bytecodes directly,

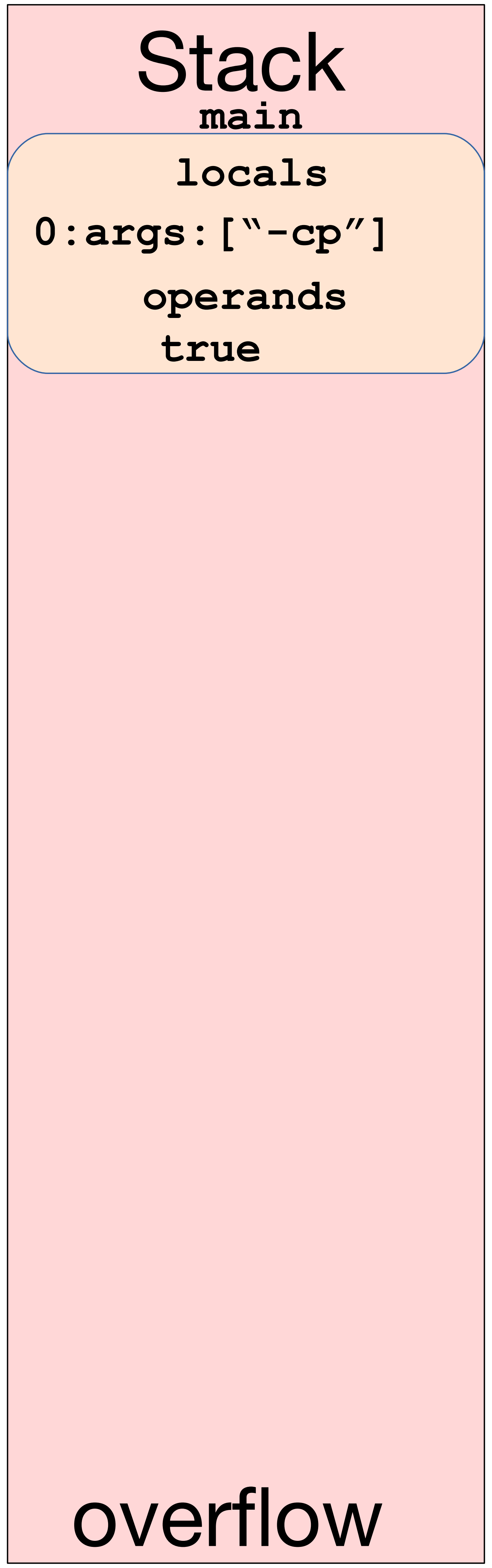
So they are emulated on real hardware (e.g. X86 or ARM chips).

This emulation is called an **execution model**.

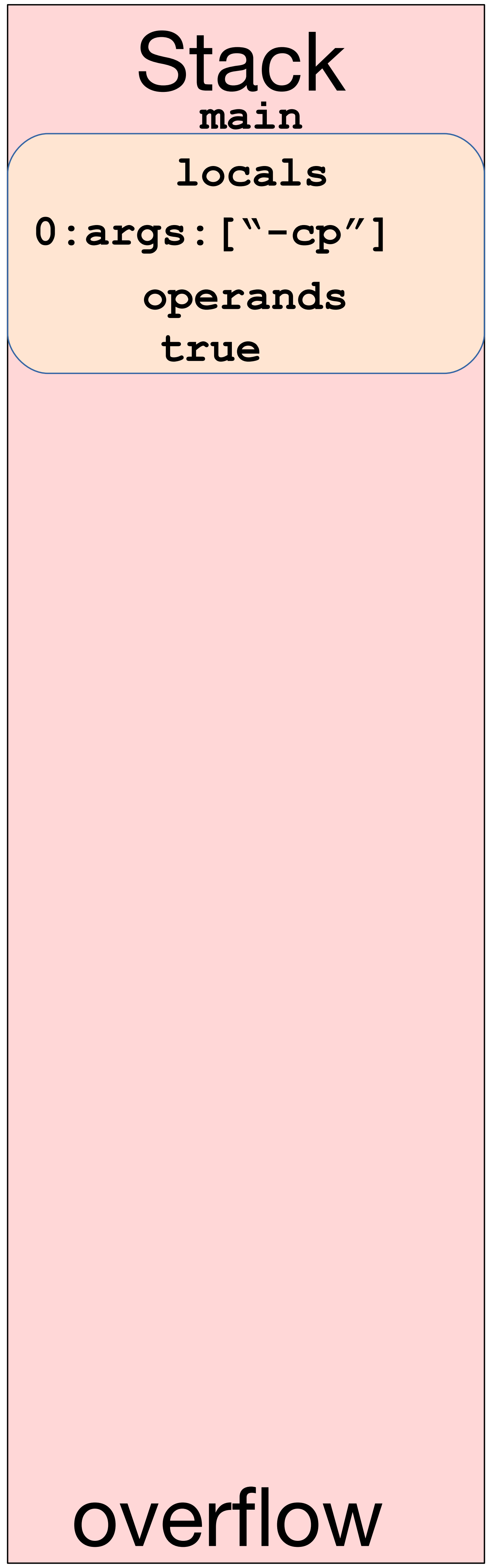
The JVM execution model is a **stack machine** with registers.

A Java function call pushes a **frame**, with some **local variables** (including call arguments) and a **operand** stack. Bytecodes push and pop values between registers and stack, or compute on the stack.

Returning removes the frame.



Bytecodes and an Execution Model



Since no **real** machine runs this, the java **virtual** machine emulates it.

The **interpreter** runs one bytecode at a time, and is fairly slow.

But it can start immediately!

The interpreter also **profiles**: it counts functions (and loops), and when the count is large enough, it triggers a compilation.

The **J**ust **I**n **T**ime compiler, or JIT, compiles hot bytecodes into real machine code. The JIT'd code runs about 10 times faster than the interpreter.

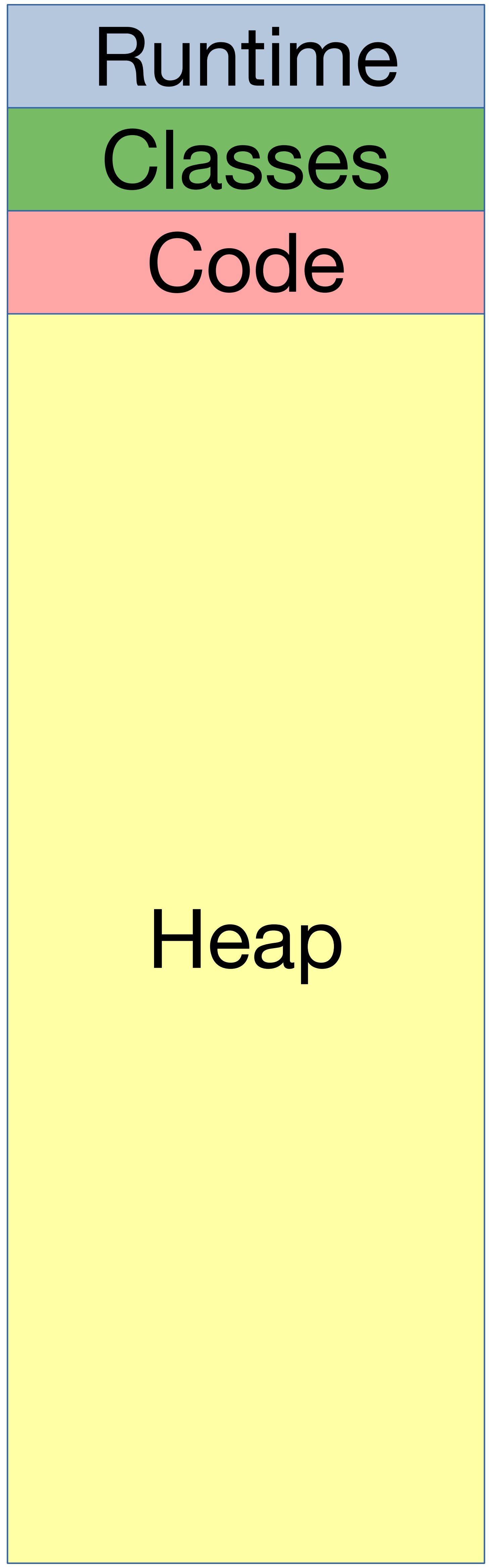
JIT'ing takes a little time, both to profile and to compile.

This means Java programs **accelerate** over time, getting faster on new code after a few seconds.

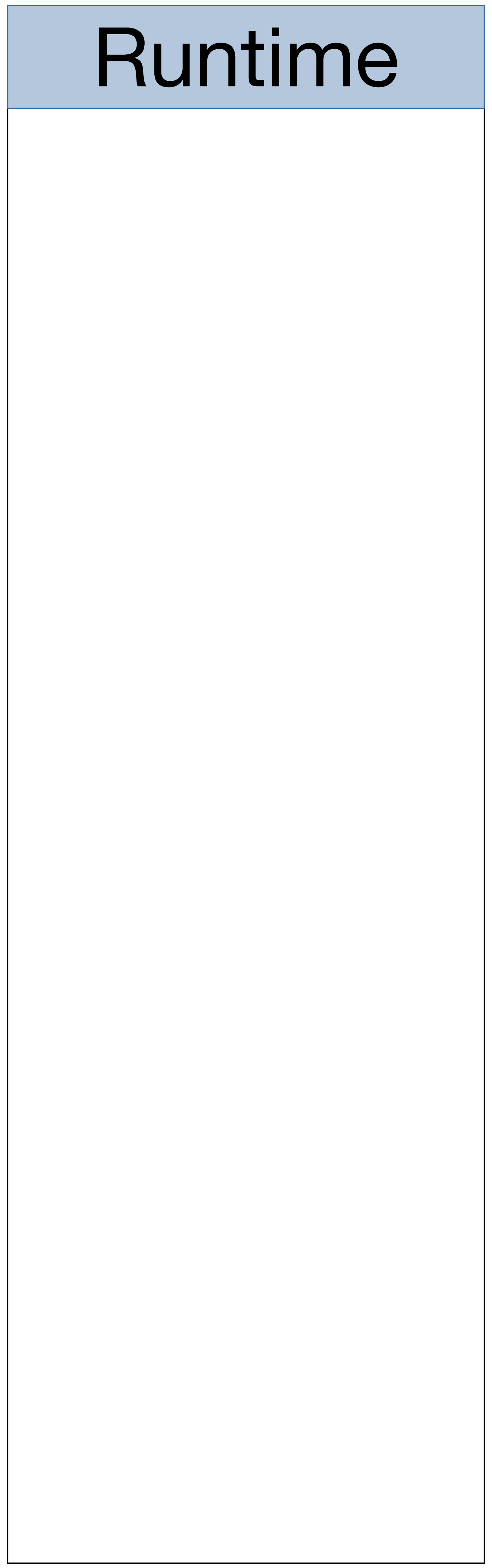
The JVM Runtime

The JVM Runtime is basically a “catch all” for everything else.

The runtime tracks:



The JVM Runtime



The JVM Runtime is basically a “catch all” for everything else.

The runtime tracks:

- Threads & thread stacks
- Classes loaded
- State of the Heap, and triggers GC as needed
- Profiled code, and triggers JIT’ing when needed
- Catches hardware exceptions and turns them into JVM exceptions
- Handles slow-path locking, blocking & waking threads
- Handles I/O and native calls
- … and much much more

Threads and Classes

Threads are newly made via a native OS call.

Threads are tracked for GC (thread stacks rapidly change are are the root of most pointer graphs).

Threads can throw exceptions, can block, sleep and awake.

Threads make objects (can trigger GC), run code (can trigger JIT'ing)

Classes are loaded from files, they have initialization states.

They describe object shapes (for GC), include code to execute.

Reflection can query them, or make objects or calls.

GC and JIT

Runtime

The GC is triggered by the runtime; triggers vary by GC type.

Obviously for being out of Free, but concurrent GCs trigger before being out, based on allocation rate.

JIT’ing is triggered by interpreter profiles, and by breaking heroic assumptions, such as class loading after assuming no-new-classes.

The JIT’d code has fast-path support for special cases such as

- uncontended locks
- allocations when not out of memory
- System.arraycopy & friends (clone, Arrays.copyOf)

Runtime

Exceptions, Locks, I/O & Native Code

JIT’d code (and other places) will use hardware to null-test “for free”.

If the test fails, a hardware exception is raised, instead of a branch.

The Runtime catches these and converts them as needed.

Fast-path locking does not use the Runtime, but a failed lock-acquire does. The thread may spin awhile before failing and sleeping. When the lock releases, the thread needs to be awakened so it can have a turn.

Native code handles most I/O requests, and the runtime handles the smooth hand-off of ByteBuffers between the JVM & OS.

WHAT IS IN A JVM?

Classes, Meta-data, things that describe your program structures.

Data, the Heap, and Garbage Collection to manage it.

Bytecodes, an Execution Model; an Interpreter and JIT(s) to run fast.

Runtime, Locks & Threads, OS access (files,JNI,Time), Debugging