A look at the internals of the Self VM

The Self Language and VM

- We’ll take a look at some of the internal structure of the Self VM. Much of what we see here will be useful in understanding the HotSpot JVM and/or V8.

- The main difference is that the Self VM was built for 32-bit uniprocessors.

- The primary references for the storage organization are [Lee 88,Chambers91] although some details were altered in later versions. Source (in C++) for the current version can be found on github via selflanguage.org.

Language recap

- Objects, slots, assignment
- Methods and messages
- Parent slots and inheritance
- Blocks, control flow
- Vectors

Objects and slots

```
3

4.2

(| x. y. |)

(| x = 3. y <- 4.2. |)

(| x = 3. y <- 4.2. |) y: 5

(| origin = (| x=3. y=4. |). corner = (| w=5. h=7. |) |)
```

Methods and messages

```
p: (| x. y. rho = ((x square + y square) squareRoot) |).

p x: 3. p y: 4. p rho  “5.0”

p _Clone
```

Parent slots and inheritance

```
q = (| parent* = p. x <- 9.
         rho = (resend.rho printString) |)

q x: 10

q y: 10

q rho “returns ’14.1421’”
```

Blocks

```
p rho < q rho ifTrue: [closest: p] False: [closest: q]

[p rho < q rho] whileTrue: [p x: p x + 1]

[p rho < q rho] whileTrue:
   [p x: p x square.
    p x > 1000 ifTrue: [^p]]
```

Control flow primitives

method invocation (i.e., sending a message)

block invocation: [p x: p x + 1] value

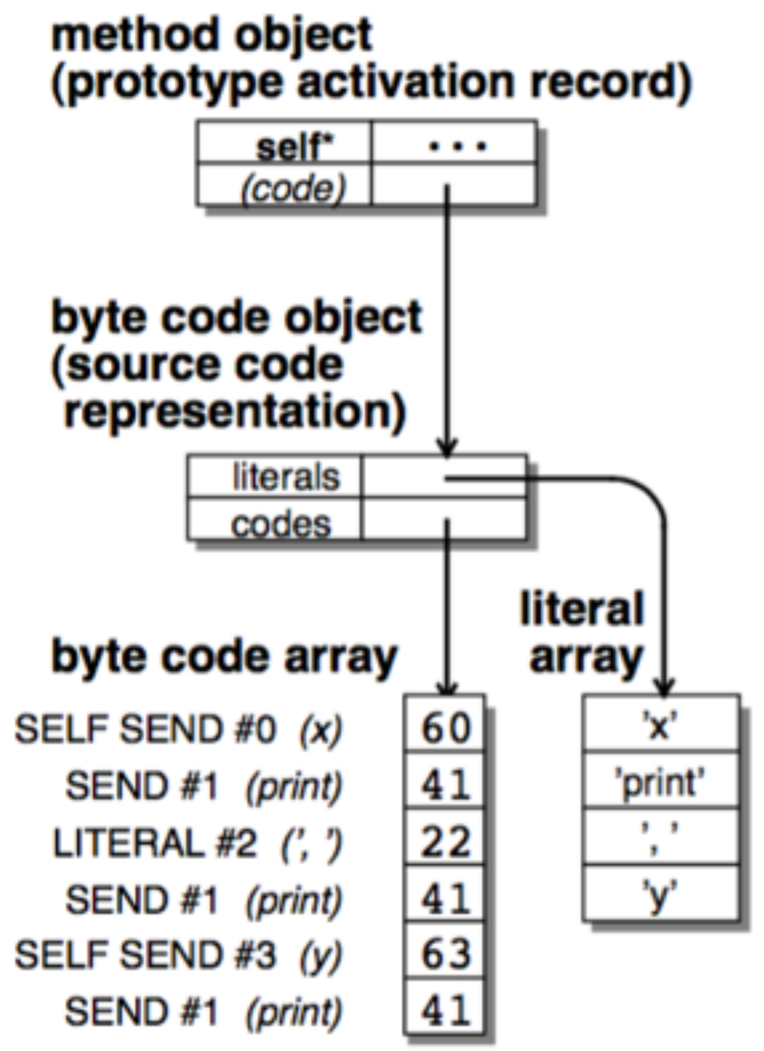
method restart:
    loop = (value. _Restart)

non-local return (^ at the end of a block)

Vectors

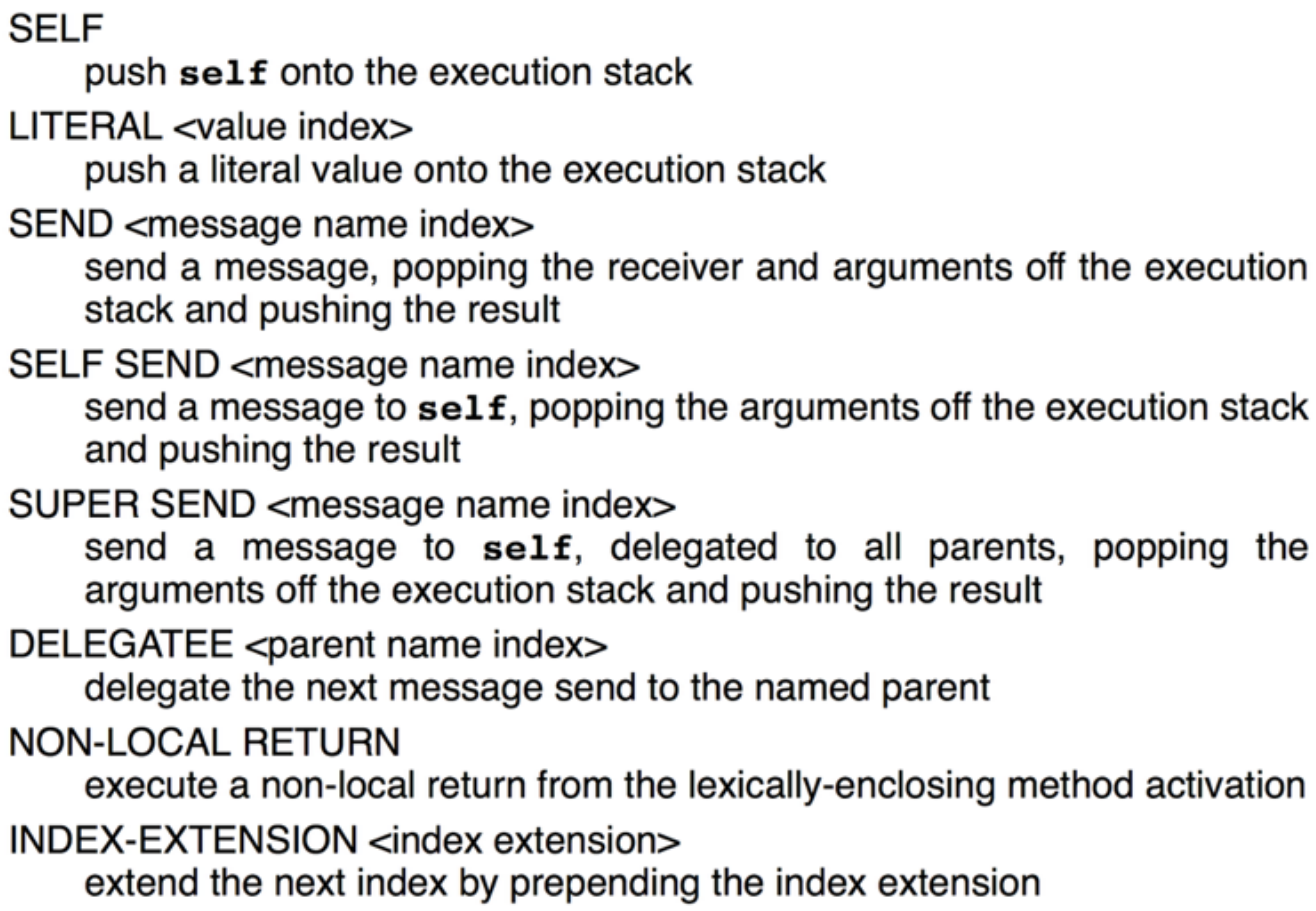
```
v: vector copy

v: v copyAddLast: 7

bv: byteVector copySize: (v at: 1) FillingWith: 0
```

Method objects and activation



The representation of the point **print** method. The top object is the prototype activation record, containing placeholders for the local slots of the method (in this case, just the **self** slot) plus a reference to the byte code object representing the source code (actually stored in the method's map). The byte code object contains a byte array for the byte codes themselves, and a separate object array for the constants and message names used in the source code.

Bytecodes



SELF
: push **`self`** onto the execution stack

LITERAL <value index>
: push a literal value onto the execution stack

SEND <message name index>
: send a message, popping the receiver and arguments off the execution stack and pushing the result

SELF SEND <message name index>
: send a message to **`self`**, popping the arguments off the execution stack and pushing the result

SUPER SEND <message name index>
: send a message to **`self`**, delegated to all parents, popping the arguments off the execution stack and pushing the result

DELEGATEE <parent name index>
: delegate the next message send to the named parent

NON-LOCAL RETURN
: execute a non-local return from the lexically-enclosing method activation

INDEX-EXTENSION <index extension>
: extend the next index by prepending the index extension

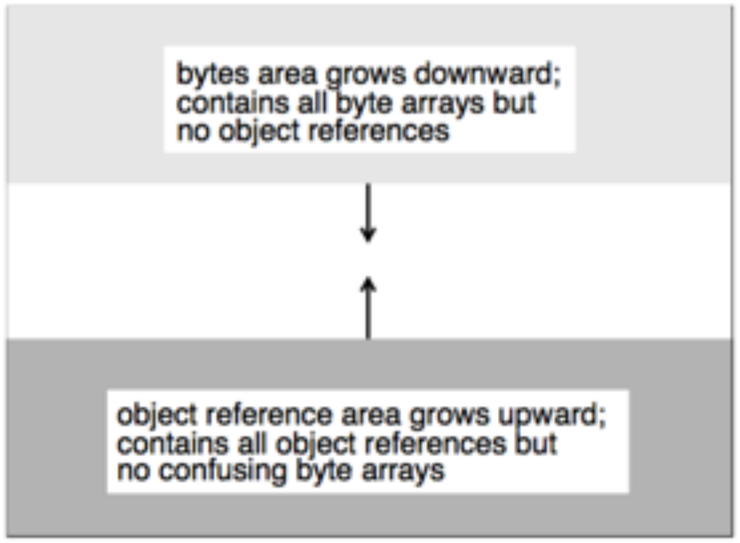
VM internals: memory organization

In the source tree, paths mentioned are relative to self/vm/src/any

Heap structure

- Memory is organized into two generations, with Young space containing Eden, From and To, and Old space being segmented. (`memory/{universe.hh,generation.hh,space.hh}`)

- Each space is filled from the ends in: slots and reference arrays grow from low-to-high addressed; byte arrays have their indexable slots in the high-to-low region. This saves scanning time.


bytes area grows downward; contains all byte arrays but no object references
object reference area grows upward; contains all object references but no confusing byte arrays

A SELF Memory Space

Tag structure

- Each 32-bit $\xrightarrow{}$ 2 bits reserved. object reference (known as an *oop*) in the reference area (and in registers and on the stack) has two lower-order tag bits (`objects/{tag.hh,oop.hh}`):

  00 - integer (`objects/smiOop.hh`)

  01 - object (`objects/memOop.hh`)

  10 - float (2 bits of exponent missing) (`objects/floatOop.hh`)

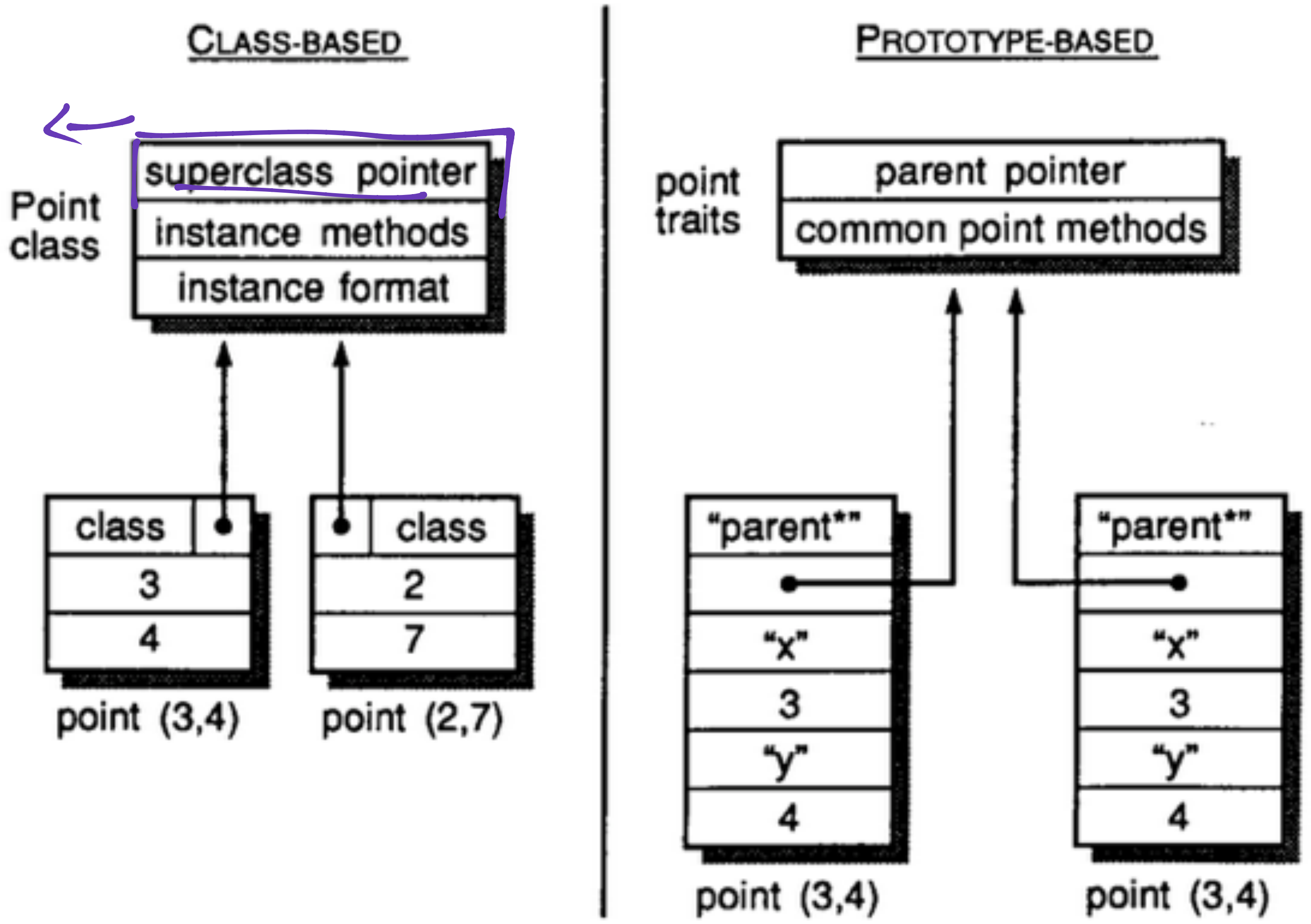
  11 - mark — only used in an object's header word (`objects/markOop.hh`)

- The use of a mark word allows for fast scanning in either direction to find an object header.

Maps

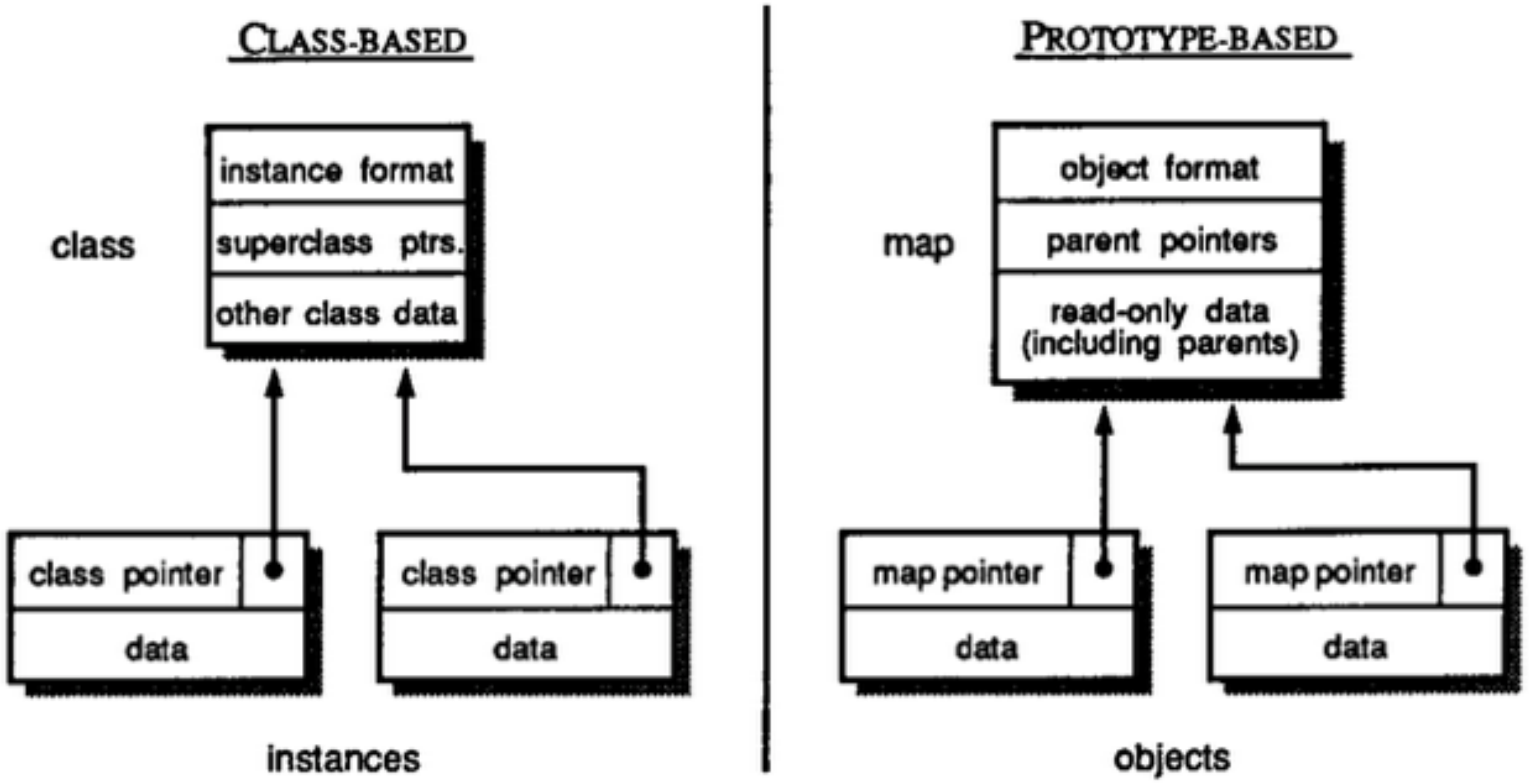
- Prototypes, implemented naively, would waste a lot of space. ✓

may not be able to access parent instance's slots.



Maps

- Instead, we factor out the shareable part into a *map.* In later VMs this is called a *hidden class.*



Map internals

- Each map (`objects/map.hh`) contains a list of slot descriptors (`objects/slotDesc.hh`), each of which names and categorizes the corresponding slot (`objects/slotType.hh`):

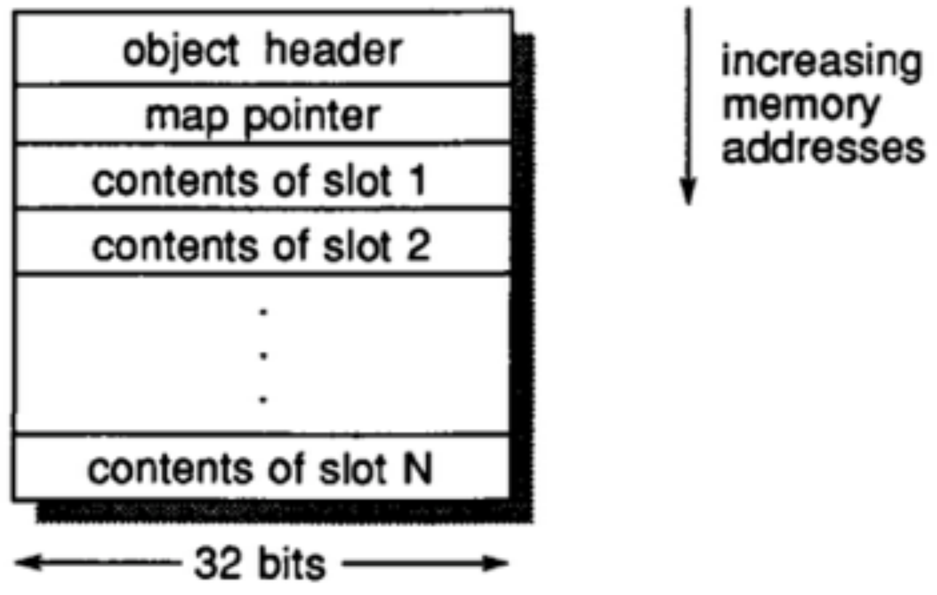
  - *assignable* data slot (has corresponding word in object as indicated offset), or *constant* data slot (value is in slotDesc), or *argument* slot (methods only)

  - is it a parent slot?

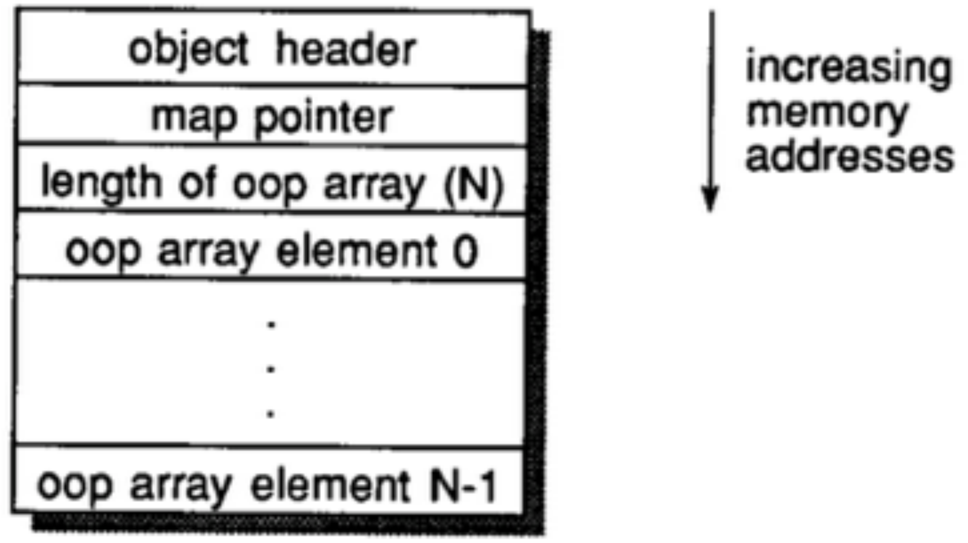
The map table

- When an object is cloned, it shares its map with its clone.
- When objects are altered using the *programming primitives* (which can, e.g., add a new slot to an object), a new map is created, but checked against a canonical map table (`memory/mapTable.hh`) to ensure that all maps are structurally unique.

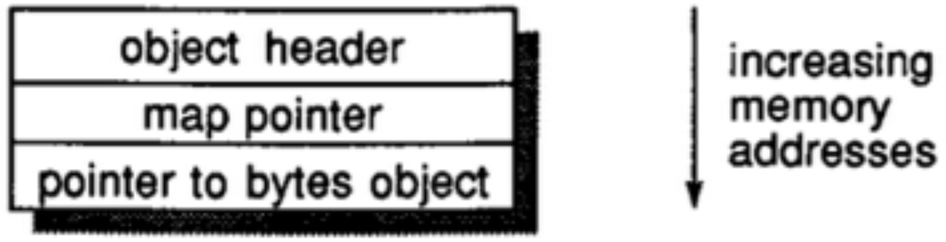
Object layouts



Named slots



Indexed slots



Indexed bytes

Objects in the VM

- As you’ve discovered for yourselves by now, programming objects in C is rather tedious.
- The Self VM uses C++ tricks to make Self objects look like C++ objects from within the VM, so that the VM itself can be written in an object-oriented style.

Objects in C++

- C structs and C++ simple classes are semantically very similar: they lay out fields in memory, and can be used as overlays to interpret memory layout, by casting an address to a pointer-to-struct.

- Methods in classes have the same cost as C functions — they are statically bound, based on static type. Inheritance does not affect this.

- If `p->f()` is statically bound, the value of `p` is irrelevant — only its type matters in binding to `f`.

- However, making a single method *virtual* causes the creation of a vtable, and adds a hidden slot in the object for the vtable pointer. Now, `p->f()` does a lookup via `p`’s vtbl slot (if `f` is `virtual`).

Virtuals only in maps

- The solution in Self is to use virtuals only in maps, and to access them via the map pointer. This avoids the memory cost of an extra word in every object.
    - Example: `oop->map()->scavenge()`
- In HotSpot, see `oops/klass.hpp` (maps are replaced by `klasses` in HS; words are not tagged because Java is statically typed).
- V8 uses visitor objects instead of virtuals `(HeapObject::map, Map)`

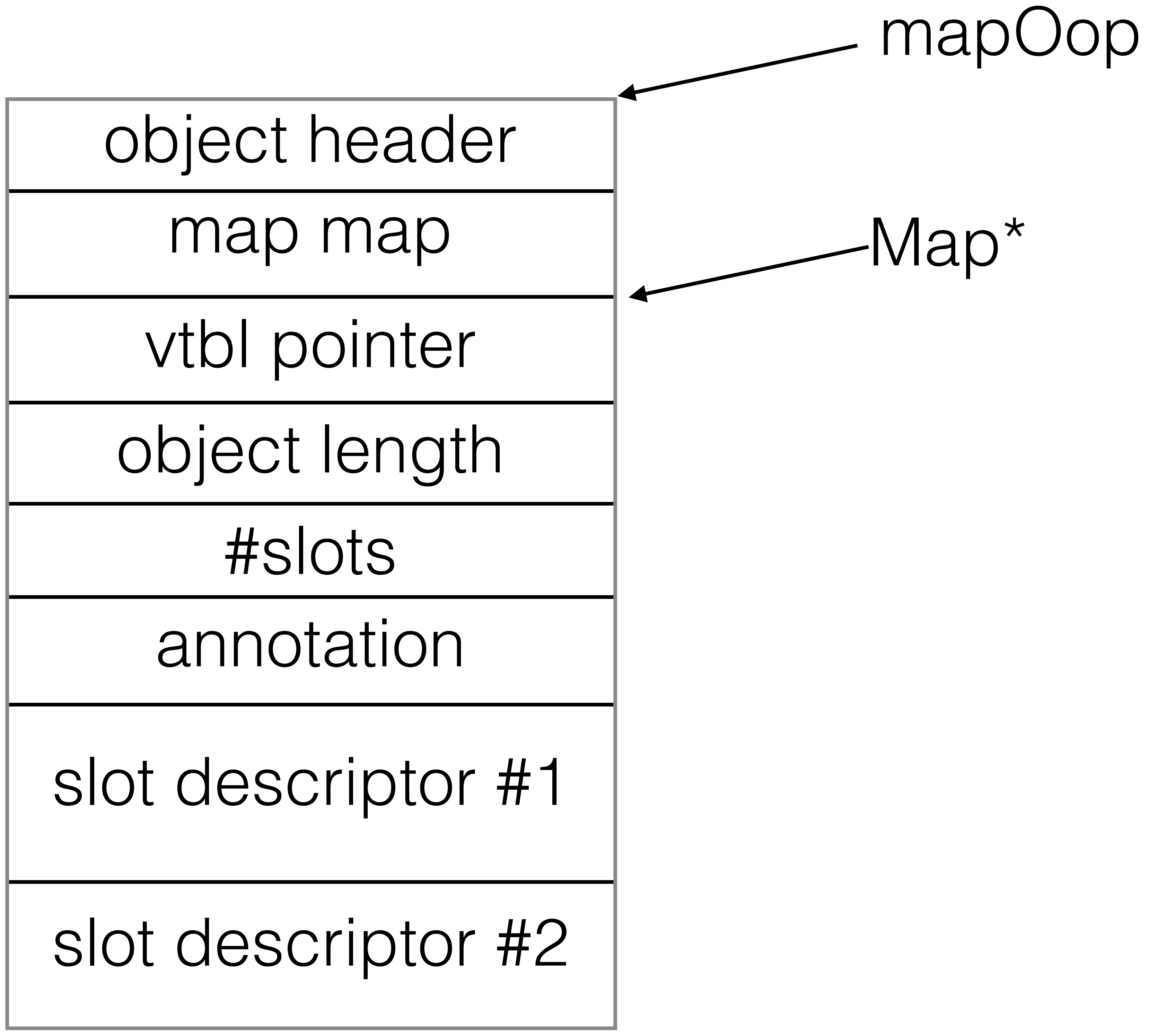
From oop to map

```
inline Map* oopClass::map() {
  fint t = tag();
  if (t == Int_Tag) {
    return smiOop(this)->map();
  } else if (t == Mem_Tag) {
    return memOop(this)->map();
  } else if (t == Float_Tag) {
    return floatOop(this)->map();
  } else {
    assert(t == Mark_Tag, "unknown tag");
    return markOop(this)->map();
  }
}
```

```
Map* memOop::map() {
 return (Map*) memOop(addr()->_map)->addr(); }
```

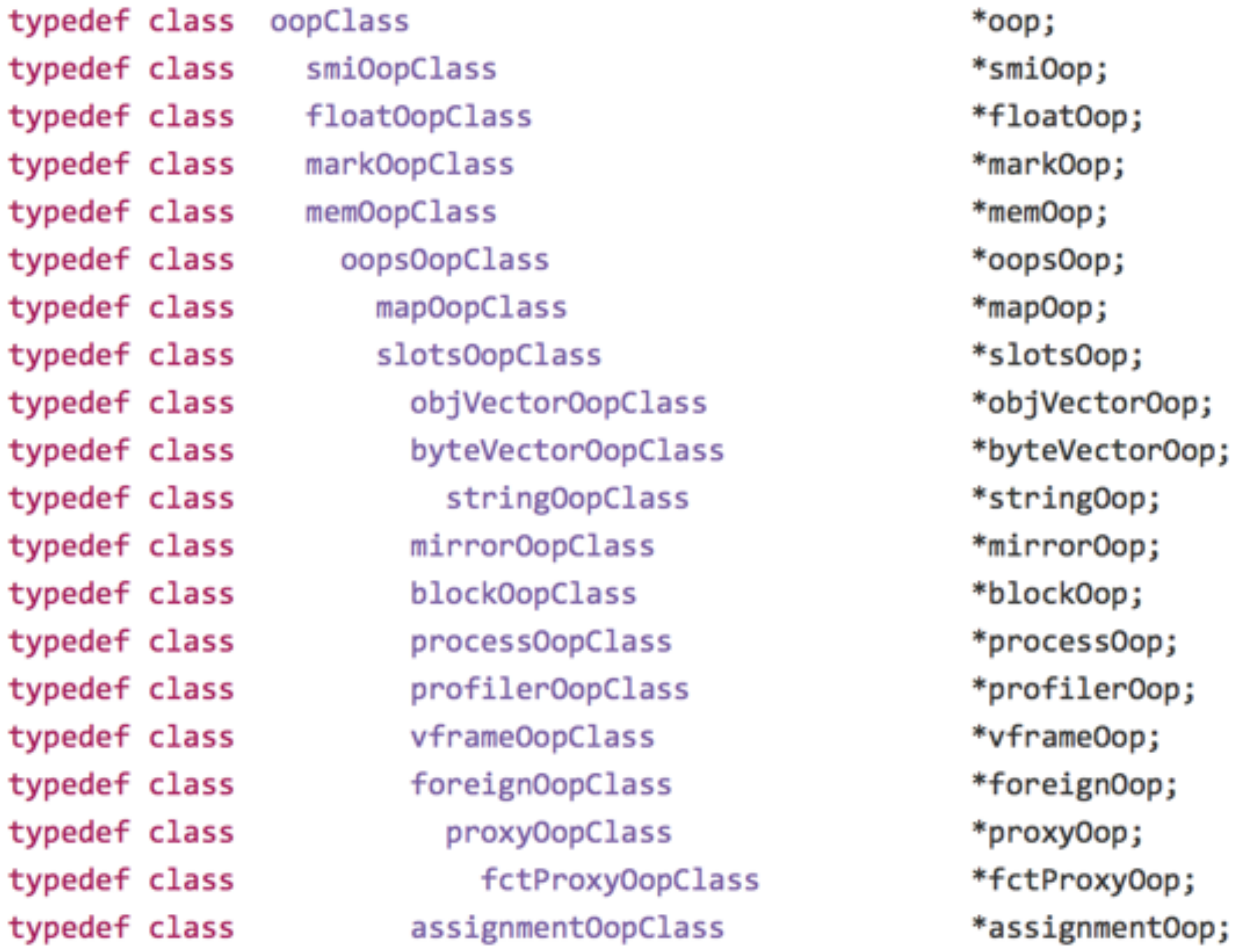
`addr()` converts a tagged `oop` to a raw pointer.

Map layout

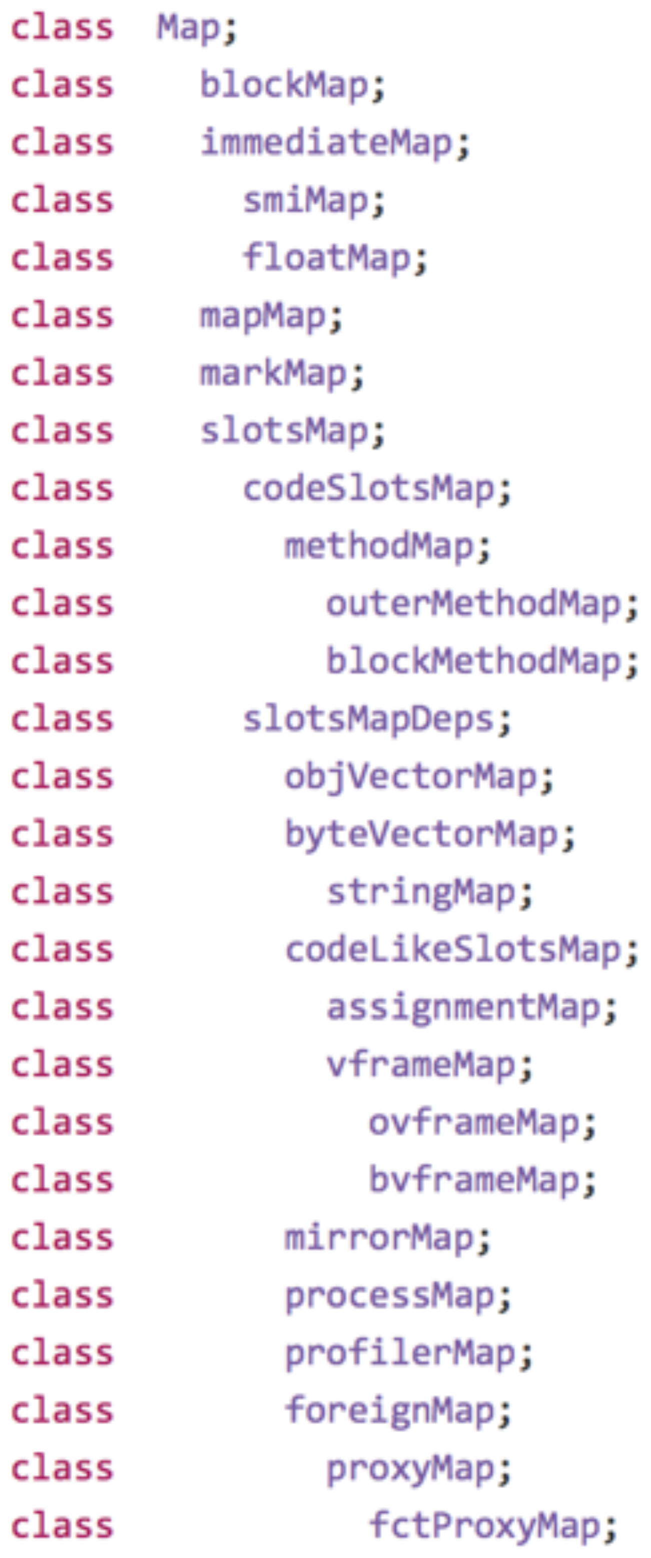

mapOop
Map*
object header
map map
vtbl pointer
object length
#slots
annotation
slot descriptor #1
slot descriptor #2

Oop and Map hierarchy

(runtime/types.hh)



```
typedef class  oopClass                           *oop;
typedef class    smiOopClass                      *smiOop;
typedef class    floatOopClass                    *floatOop;
typedef class    markOopClass                     *markOop;
typedef class    memOopClass                      *memOop;
typedef class      oopsOopClass                   *oopsOop;
typedef class        mapOopClass                  *mapOop;
typedef class        slotsOopClass                *slotsOop;
typedef class          objVectorOopClass          *objVectorOop;
typedef class          byteVectorOopClass         *byteVectorOop;
typedef class            stringOopClass           *stringOop;
typedef class          mirrorOopClass             *mirrorOop;
typedef class          blockOopClass              *blockOop;
typedef class          processOopClass            *processOop;
typedef class          profilerOopClass           *profilerOop;
typedef class          vframeOopClass             *vframeOop;
typedef class          foreignOopClass            *foreignOop;
typedef class            proxyOopClass            *proxyOop;
typedef class              fctProxyOopClass       *fctProxyOop;
typedef class          assignmentOopClass         *assignmentOop;
```



```
class  Map;
class    blockMap;
class    immediateMap;
class      smiMap;
class      floatMap;
class    mapMap;
class    markMap;
class    slotsMap;
class      codeSlotsMap;
class        methodMap;
class          outerMethodMap;
class          blockMethodMap;
class      slotsMapDeps;
class        objVectorMap;
class        byteVectorMap;
class          stringMap;
class        codeLikeSlotsMap;
class          assignmentMap;
class          vframeMap;
class            ovframeMap;
class            bvframeMap;
class        mirrorMap;
class        processMap;
class        profilerMap;
class        foreignMap;
class          proxyMap;
class            fctProxyMap;
```

The oop hierarchy is a representation hierarchy;
the Map hierarchy is behavioral.

Allocation

```
universe::alloc_objs(fint size, bool mustAllocate)→
  newGeneration::alloc_objs(fint size, bool mustAllocate)→
    newSpace::alloc_objs(fint size, bool mustAllocate)→
      newSpace::alloc_objs_local(fint size) and then maybe
      newSpace::alloc_more_objs_and_bytes(fint size,
                fint bsize, char *&bytes, bool mustAllocate)

oop* alloc_objs_local(fint size) {
  oop* p = objs_top;
  oop* p1 = p + size;
  if (p1 < bytes_bottom) {
    objs_top = p1;
    return p;
  } else {
    return NULL;
  } }
```

Allocation — slow path

```
oop* newSpace::alloc_more_objs_and_bytes(fint size,
   fint bsize, char*& bytes, bool mustAllocate) {
  Memory->need_scavenge();
  return next_space
   ? next_space->alloc_objs_and_bytes(size, bsize, bytes,
                                       mustAllocate)
   : Memory->old_gen->alloc_objs_and_bytes(size, bsize,
                                       bytes, mustAllocate);
}
```

Scavenging

After the roots have been scavenged (in `universe::scavenge()`):

```
{FOR_EACH_OLD_SPACE(s) s->scavenge_recorded_stores();}
while (    old_gen->scavenge_promotions()
       || new_gen->scavenge_contents())
  ;
```

During card scanning the following macro is used in the unrolled loop to scan a marked card (`memory/rSet.cpp`):

```
# define VISIT(w)                                                           \
    x = *(w);                                                               \
    if (x->is_mem()) {                                                      \
      memOop mx = memOop(x);                                                \
      if (Memory->new_gen->is_new(mx, bound)) {                             \
        *(w) = mx = memOop(mx->scavenge());                                 \
        if (new_byte != 0 && Memory->new_gen->is_new(mx, bound)) new_byte = 0;\
      }                                                                     \
    }
```

Scavenger loop

(memory/space.cpp)

```
bool oldSpace::scavenge_promotions()
{
  bool flag= false;
  for (oop* p = objs_scavenge_point;  p < objs_top;  ++p) {
    oop x = *p;
    if (x->is_new()) {
      flag = true;
      x = x->scavenge();
      // haven't check-stored newly tenured objects; do it here
      Memory->store(p, x);
    }
  }
  objs_scavenge_point = objs_top;
  return flag;
}

bool newSpace::scavenge_contents() {
  if (objs_top == objs_scavenge_point) return false;
  oop *so_far = objs_scavenge_point;
  do {
    *so_far= (*so_far)->scavenge();
  } while (++so_far < objs_top);
  objs_scavenge_point = so_far;
  return true;
}
```

Finding the roots: VM roots

(memory/universe.more.cpp)

- Every component of the VM that records oops has to be able to enumerate them during a scavenge or GC.
- All VM oop variables are enumerated in a macro (universe.hpp) which can apply a second-level macro

```
# define APPLY_TO_VM_OOPS(template)                    \
    template(&Memory->lobbyObj)                        \
    template(&Memory->nilObj)                          \
    template(&Memory->trueObj)                         \
    …
# define SCAVENGE_TEMPLATE(p)                          \
    *((oop*) p) = oop(*p)->scavenge();

APPLY_TO_VM_OOPS(SCAVENGE_TEMPLATE); (in universe::scavenge())
```

Old space management

- Old space allocations monitor free space against a yellow line and a red line.

- When the yellow line is crossed, a Self thread is signalled to wake up. It can force a GC or and/or allocate another segment. It tries to adapt to changing behavior and maintain good performance.

- When the red line is crossed a GC is forced by the VM.

Programming primitives

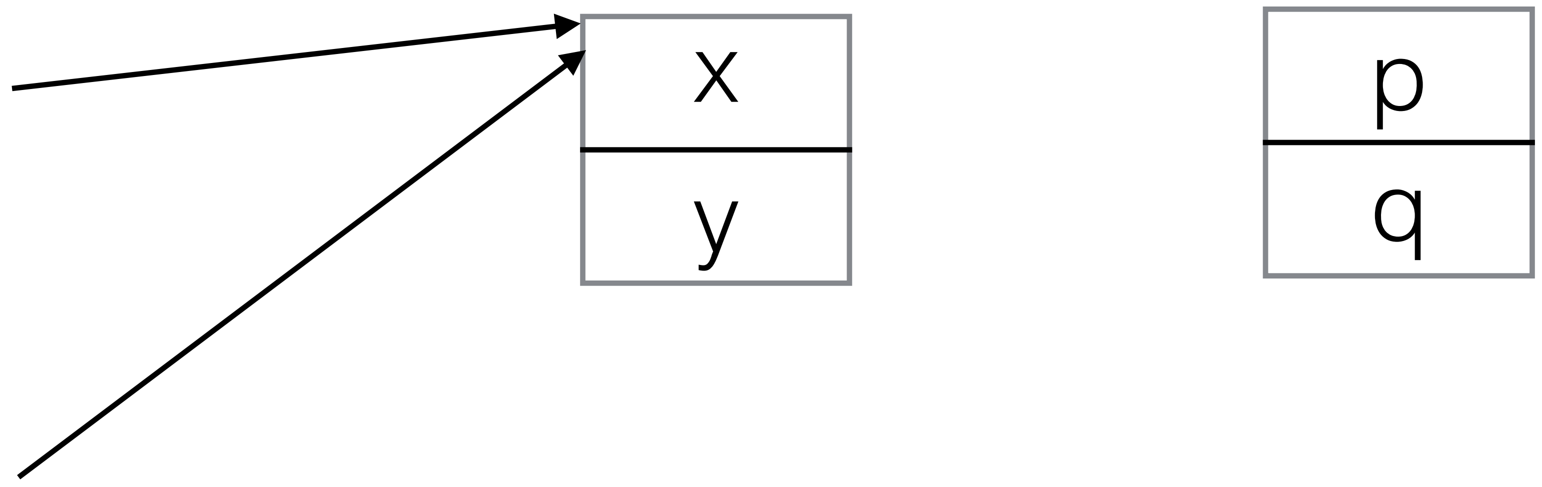
- Self *programming primitives* are invoked to modify the structure and behavior of objects (e.g., to add a slot to an object). They are also called by the parser when reading in source and constructing the corresponding objects.

- The most common idiom in the VM is to instantiate a new object from the new map, making a new map en route.

- Then, the _Define: primitive is used to substitute the original object with a clone of the new one.

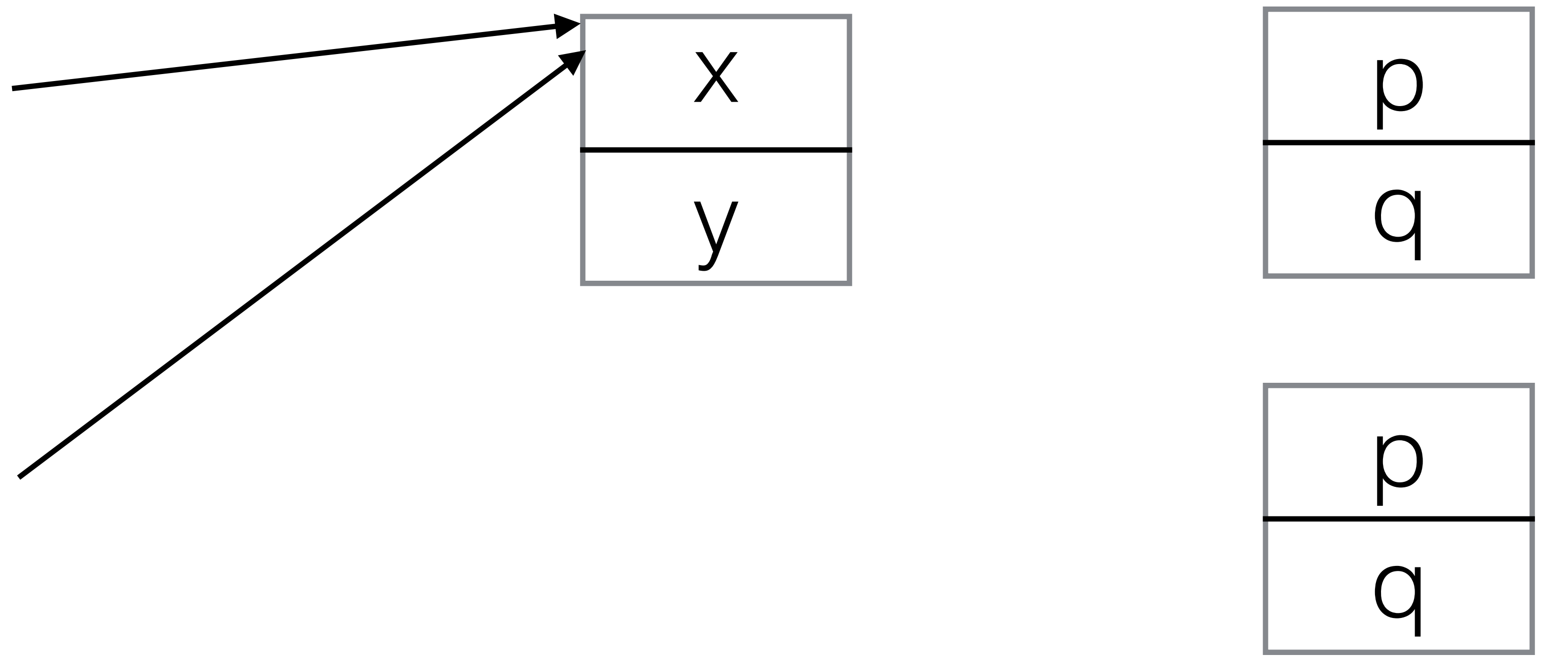
_Define

(| x. y. |) _Define: (| p. q. |)



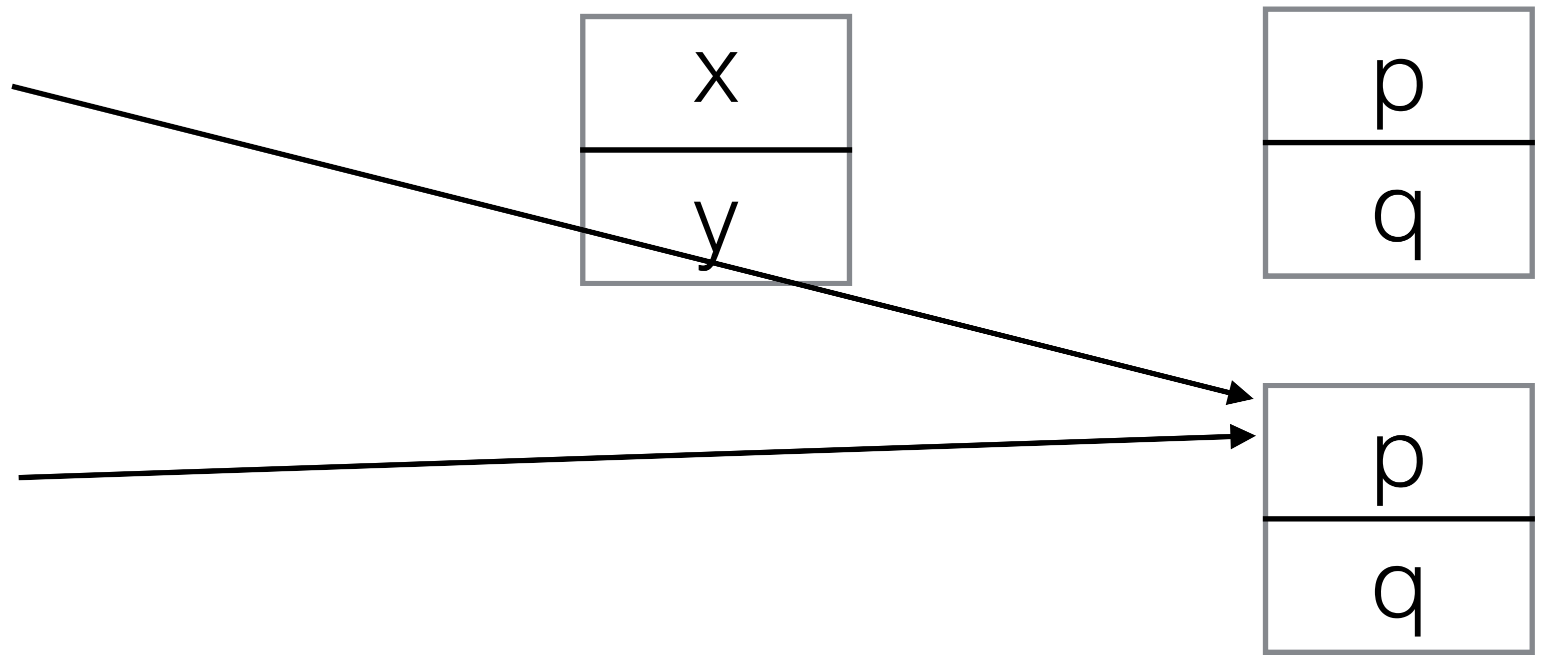
_Define

```
(| x. y. |) _Define: (| p. q. |)
```



_Define

(| x. y. |) _Define: (| p. q. |)



_Define implementation

- Most invocations of _Define are during parsing, where we are repeatedly altering object structures. Most of the objects undergoing mutation were just created — and hence are in the young generation.

- The implementation of _Define optimizes this case by using the remembered set to locate all the references to be updated, without having to scan the entire old generation.

Scanning at high speed

- For objects in the old generation undergoing mutation, we need to scan the entire heap for heap.
- This is also used in the programming environment to support exploration of behavior (“find all objects that respond to *x*”)
- The marks in object headers allow for scanning at full bandwidth; when a reference is found, the containing object can be located by scanning backward to the previous header.
- (Final twist) To elide the boundary check in the scan, a sentinel containing the desired reference is placed at the end of the space being scanned.