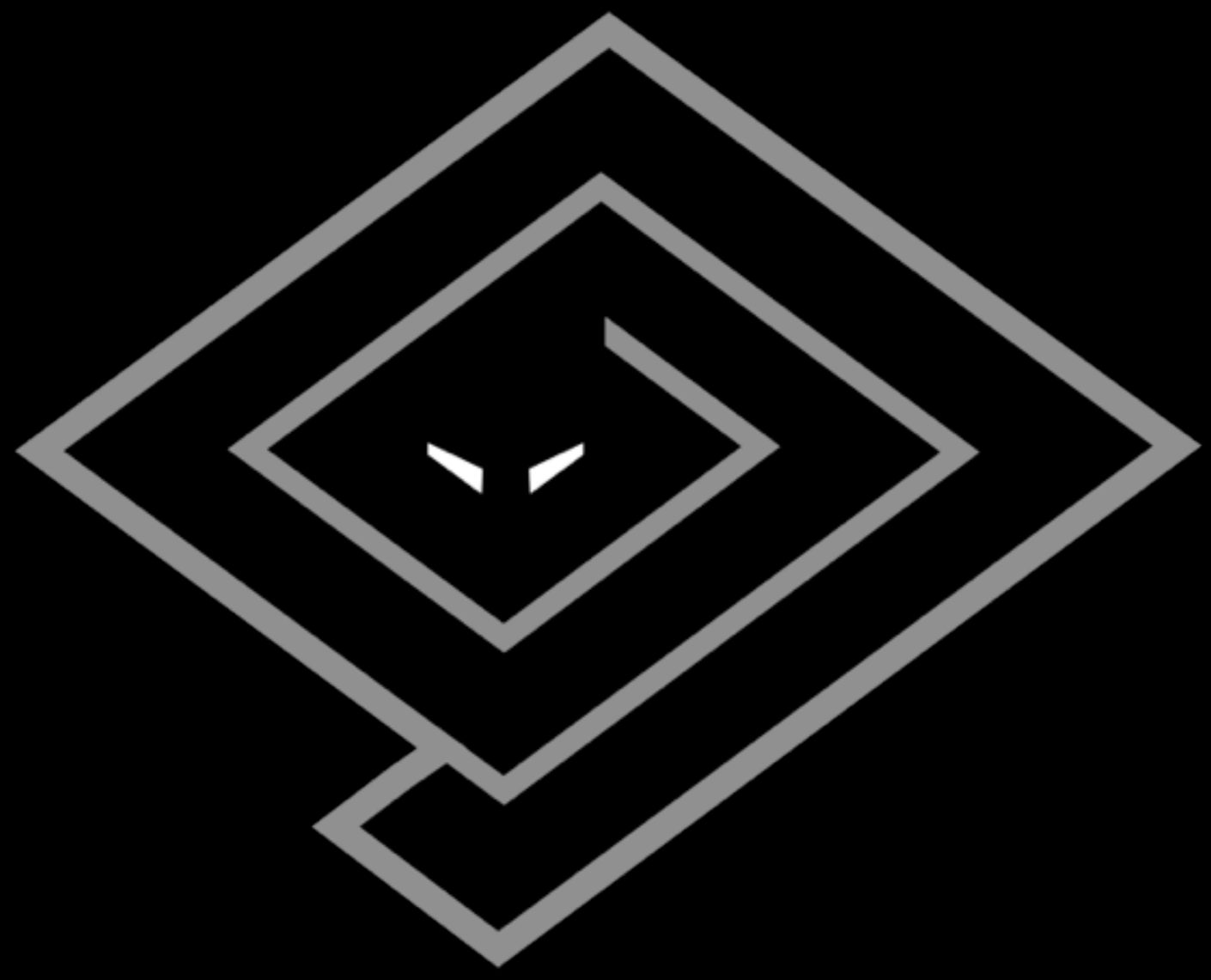
Turtles
All The Way Down

Demystifying Deferreds, Decorators, and Declarations

Hello!

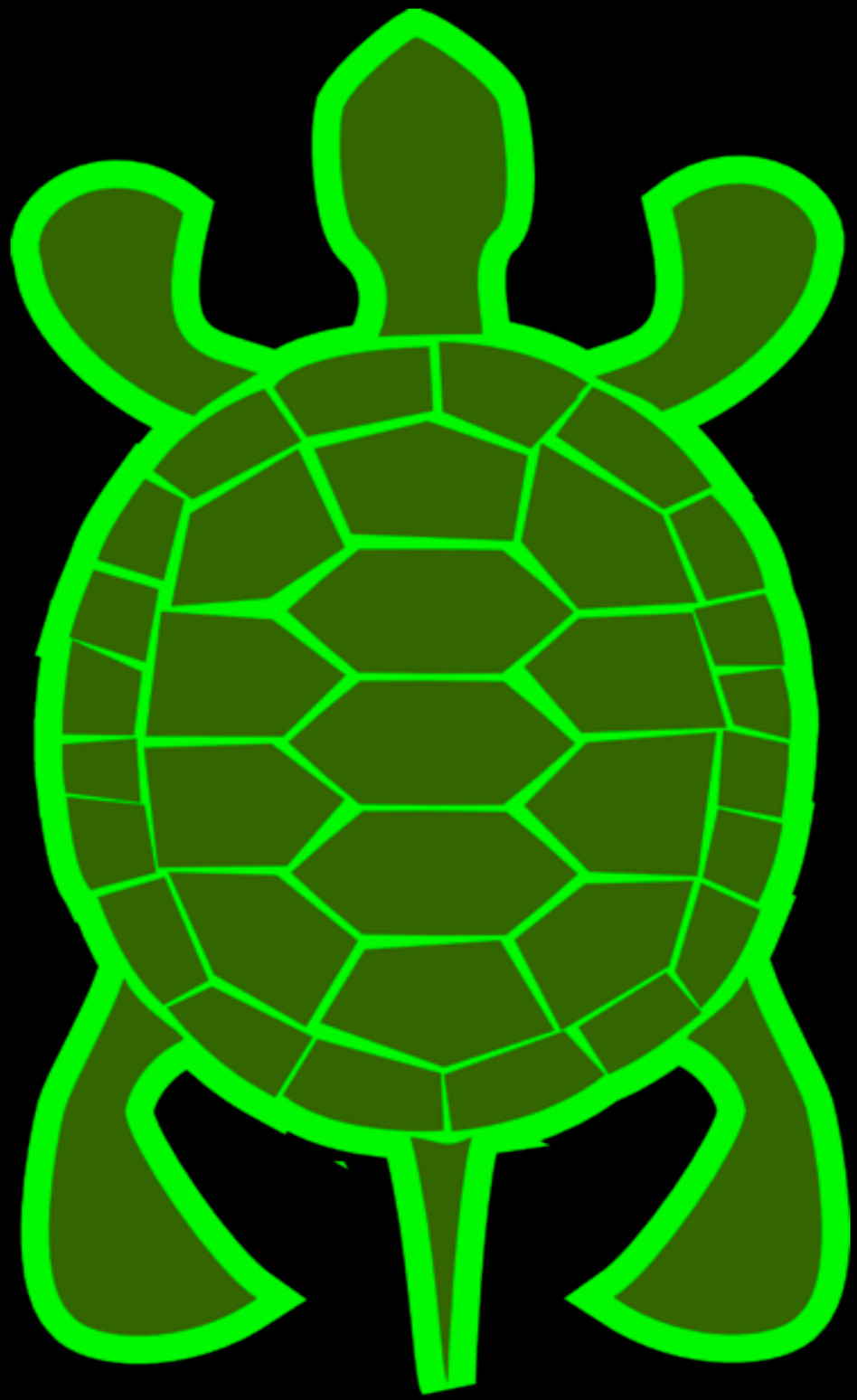
glyph@twistedmatrix.com

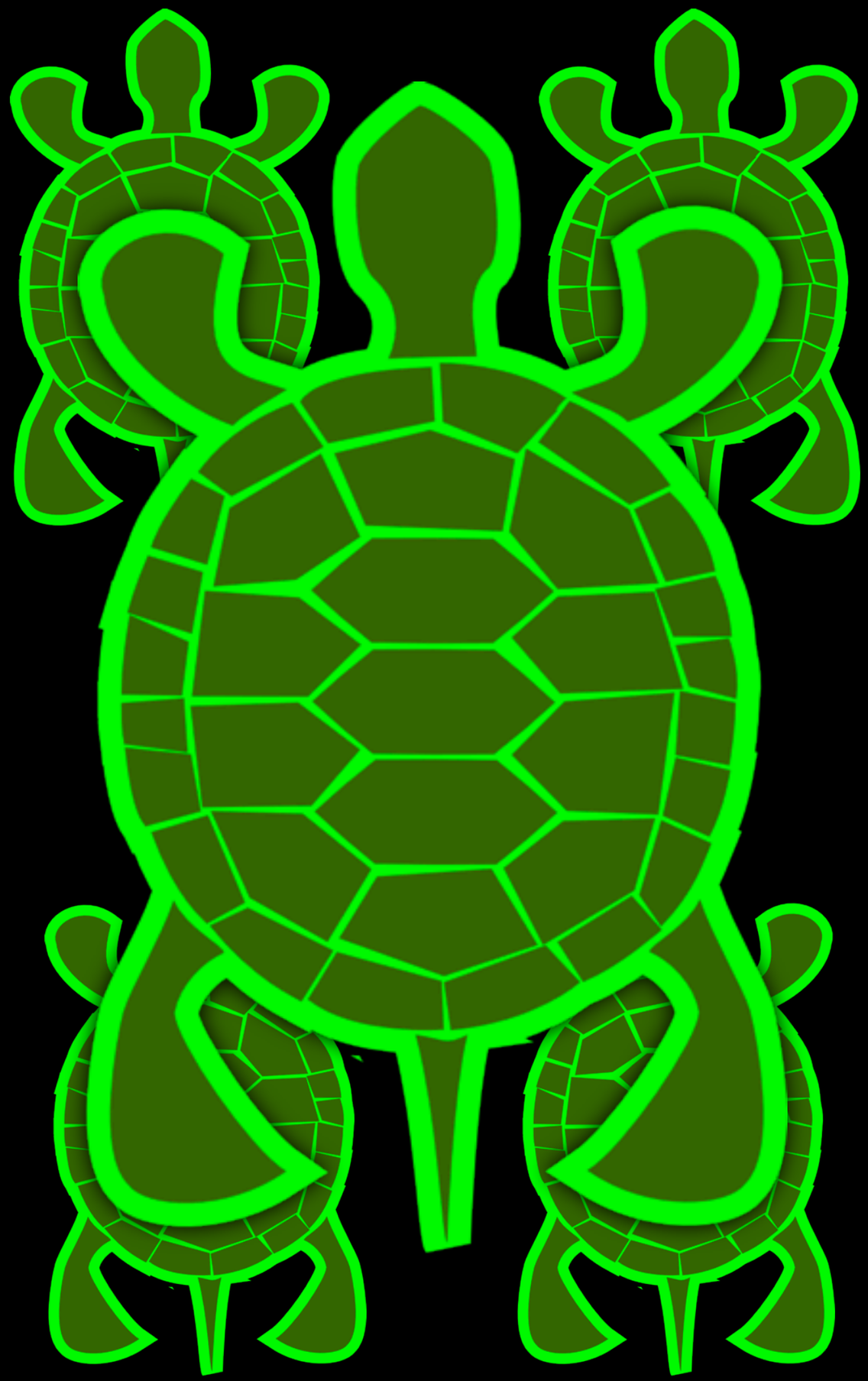
(glyph)



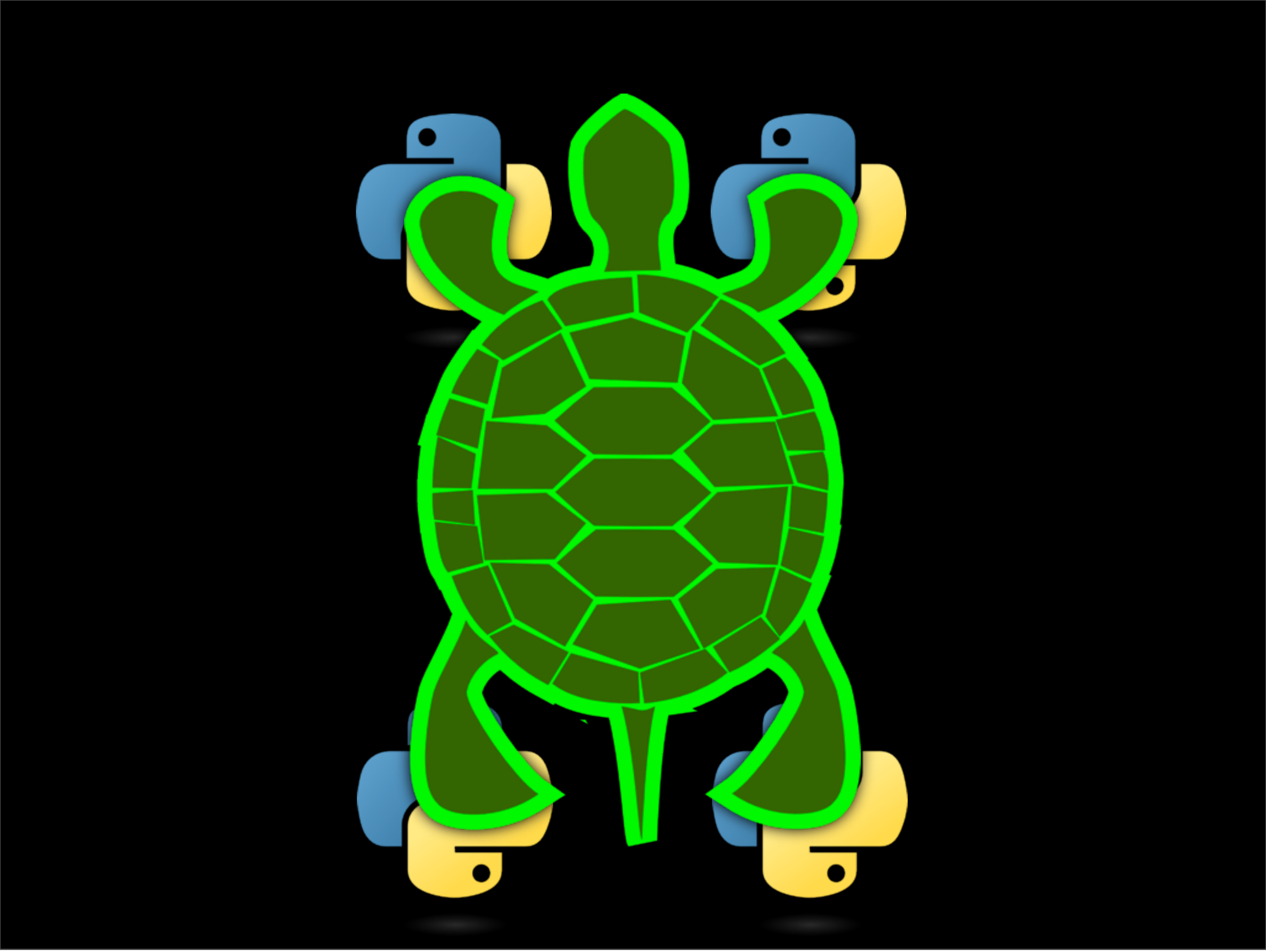
(twisted)

(twisted)





Saturday, February 20, 2010



EXPECTATIONS

MANAGEMENT

!

What you **won't** learn:

What you **won't** learn:

- In-Depth Usage of:
  - Twisted
  - Interfaces
  - Deferreds
  - Decorators
  - Metaclasses
  - ...

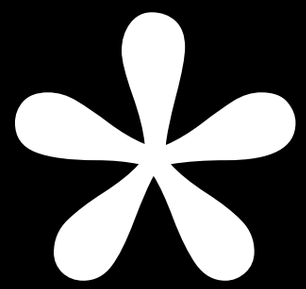
What you **will** learn:

What you **will** learn:

- How to think about:
  - what an “object” is,
  - what “def” and “class” really mean,
  - and most of all:

How to conquer your fear of “weird” Python.

How to conquer your fear of “weird” Python.*



Easy

Consistent

Simple

(as long as you think of it the right way)

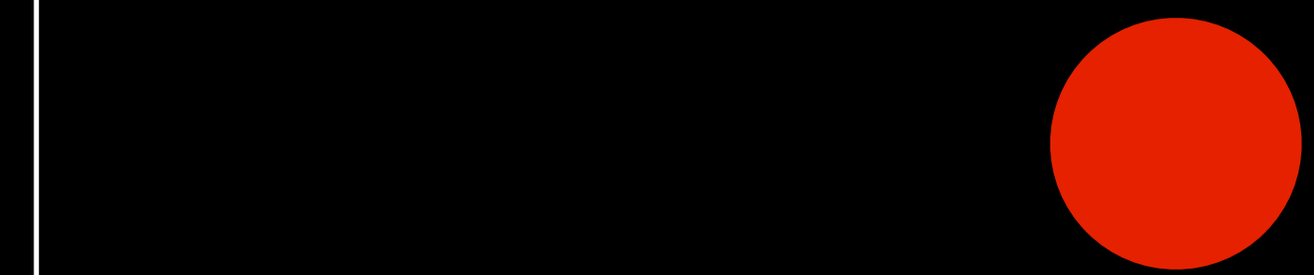
X

$$X =$$

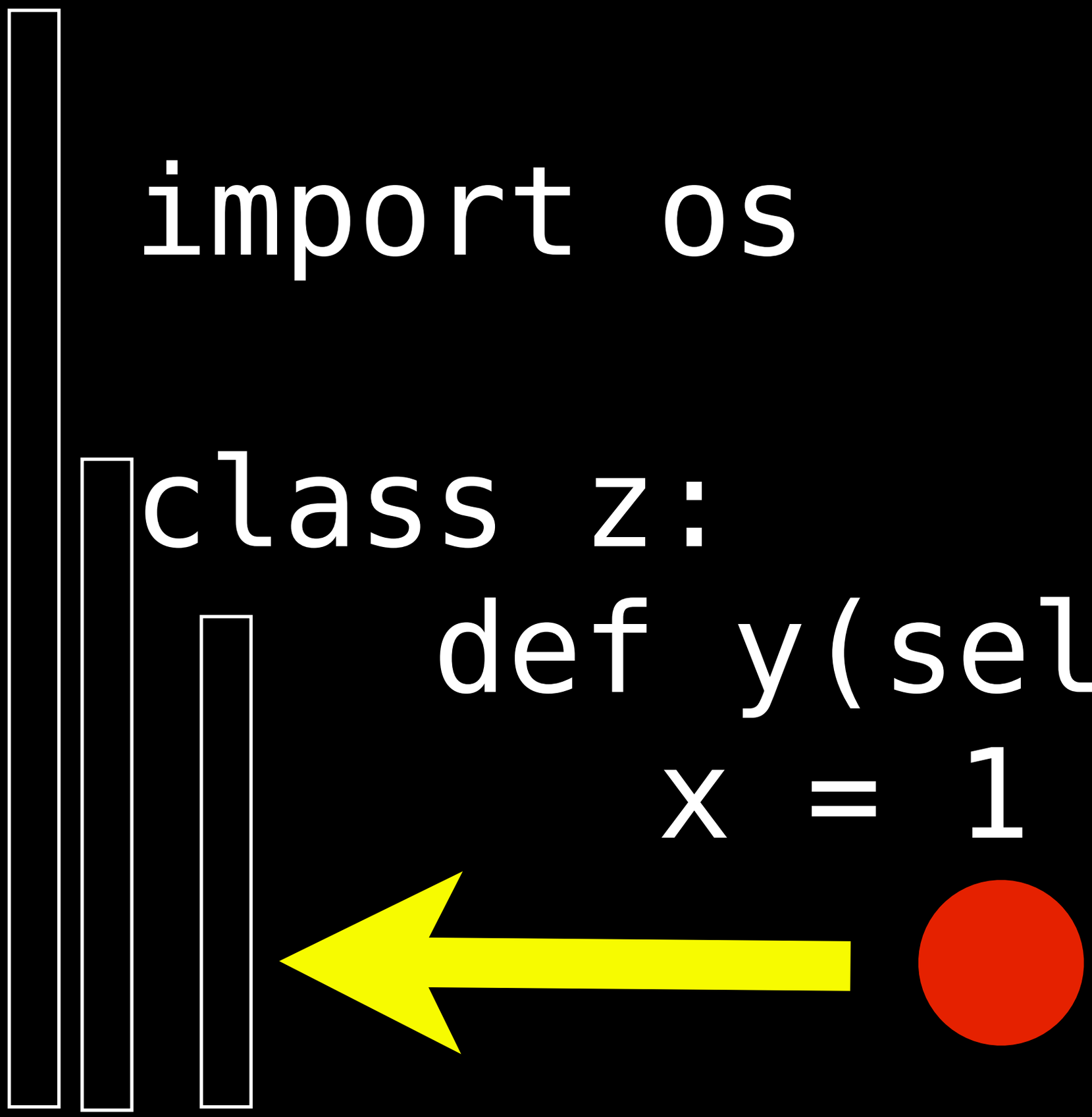
$$x = 1$$

```
import os

class z:
    def y(self):
        x = 1
```

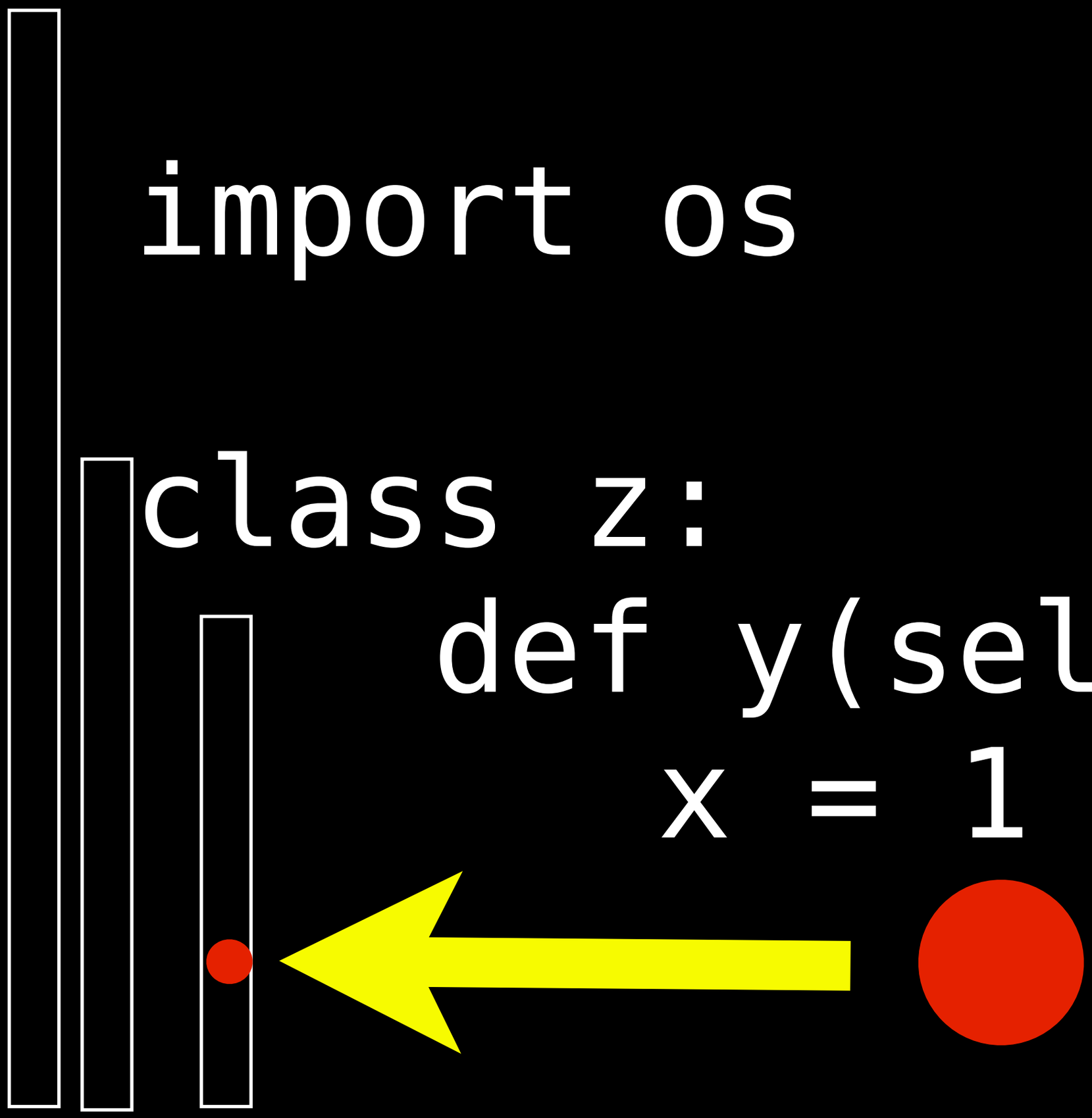
```
import os

class z:
    def y(self):
        x = 1
```



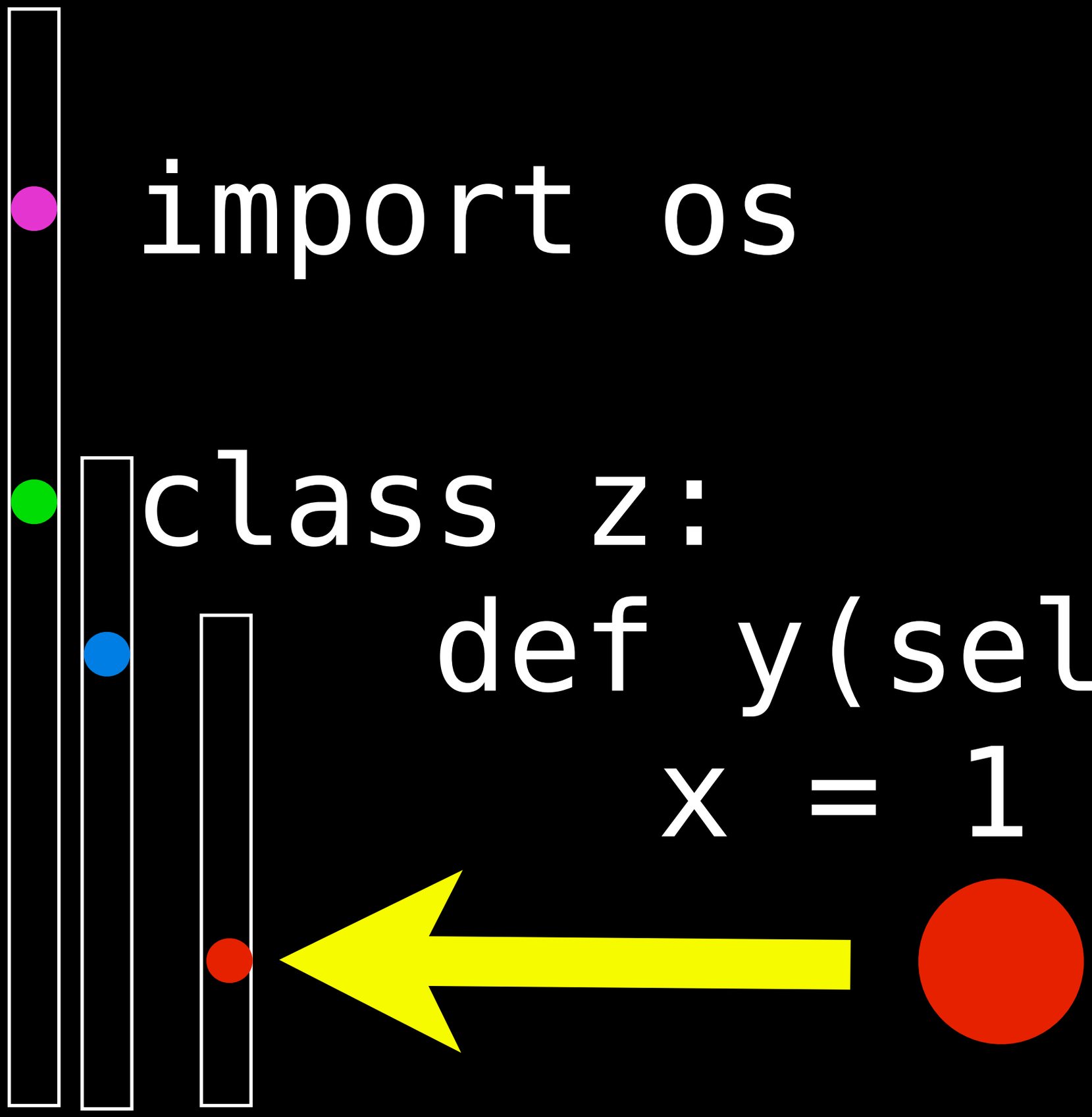
```
import os

class z:
    def y(self):
        x = 1
```



```
import os

class z:
    def y(self):
        x = 1
```



```
import os

class z:
    def y(self):
        x = 1
```



$$x = 1$$

```
globals()['x'] = 1
```

Or,

```
def value(f):
    return f()
@value
def x():
    return 1
```

Or,

```
def one(name, bases, attrs):
    return 1

class x:
    __metaclass__ = one
```

```
def one(name, bases, attrs):
    return 1
```

```
class x(metaclass=one):
    pass
```

```
def one(name, bases, attrs):
    return 1
```

```
class x(metaclass=one):
    pass
```

Woo, python 3!

Free your mind!

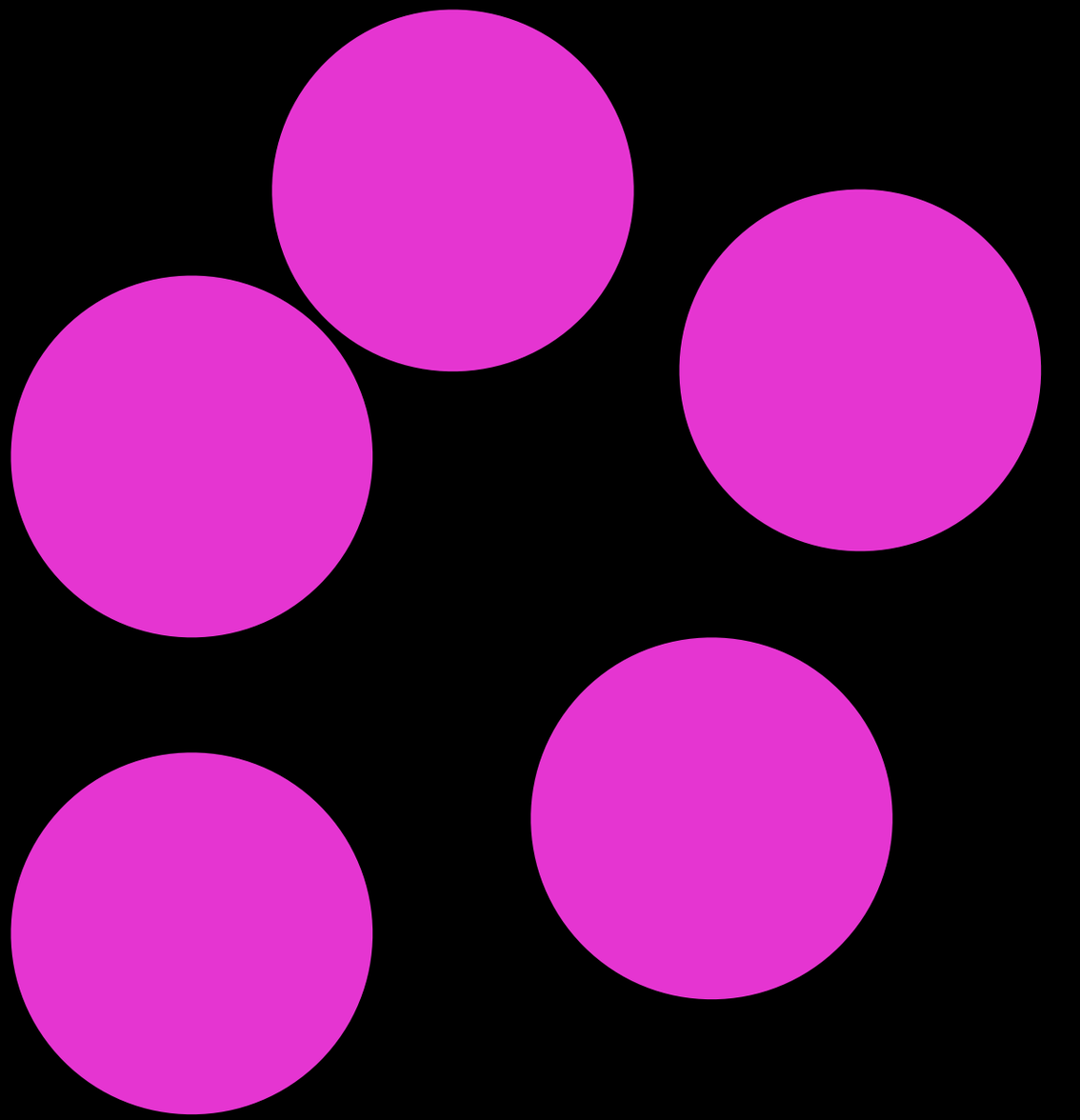
Python is special!

Java:

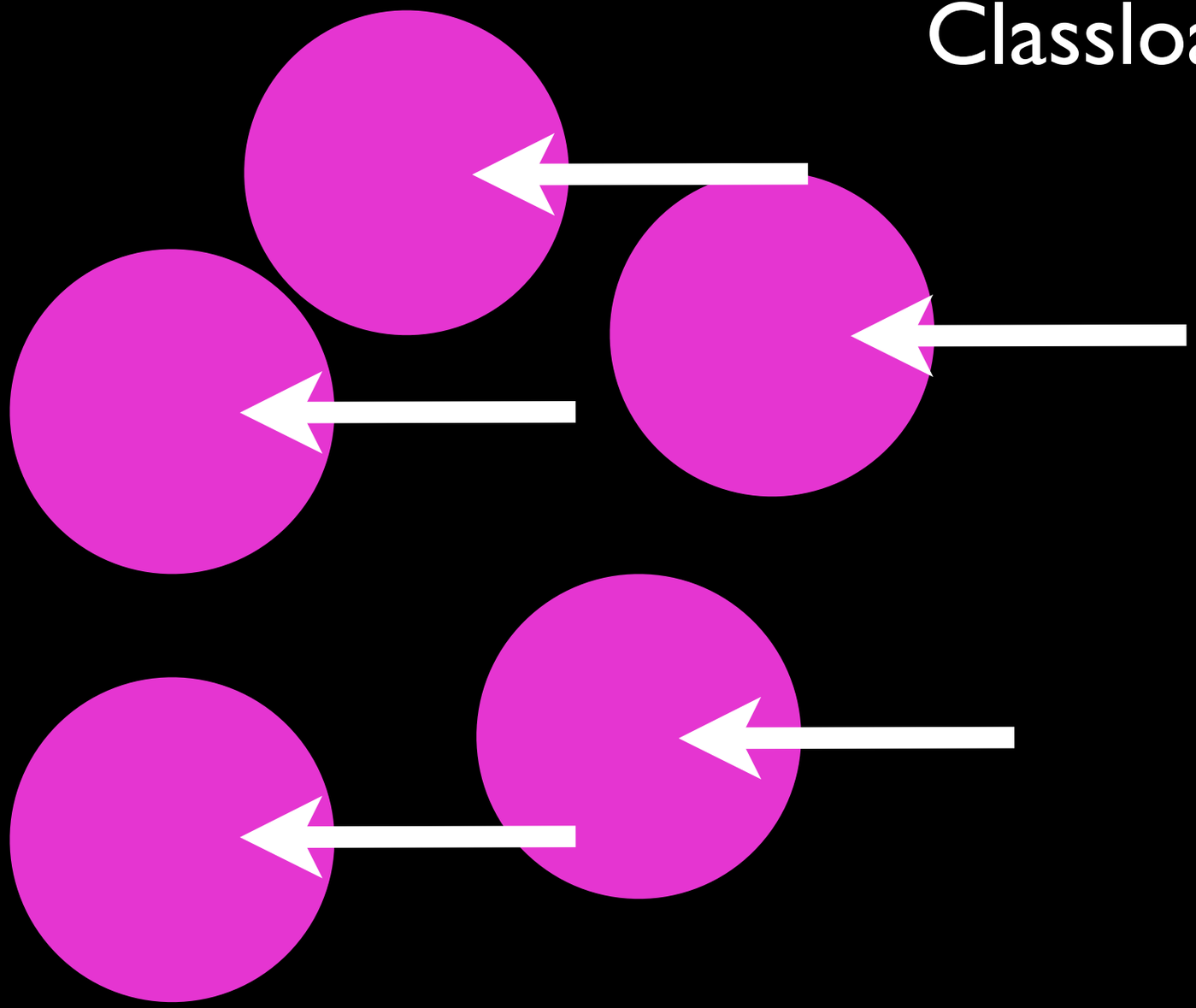
```
class Foo {
    @Annotation
    public static final int bar (String [])
    {
        ...
    }
}
```

?

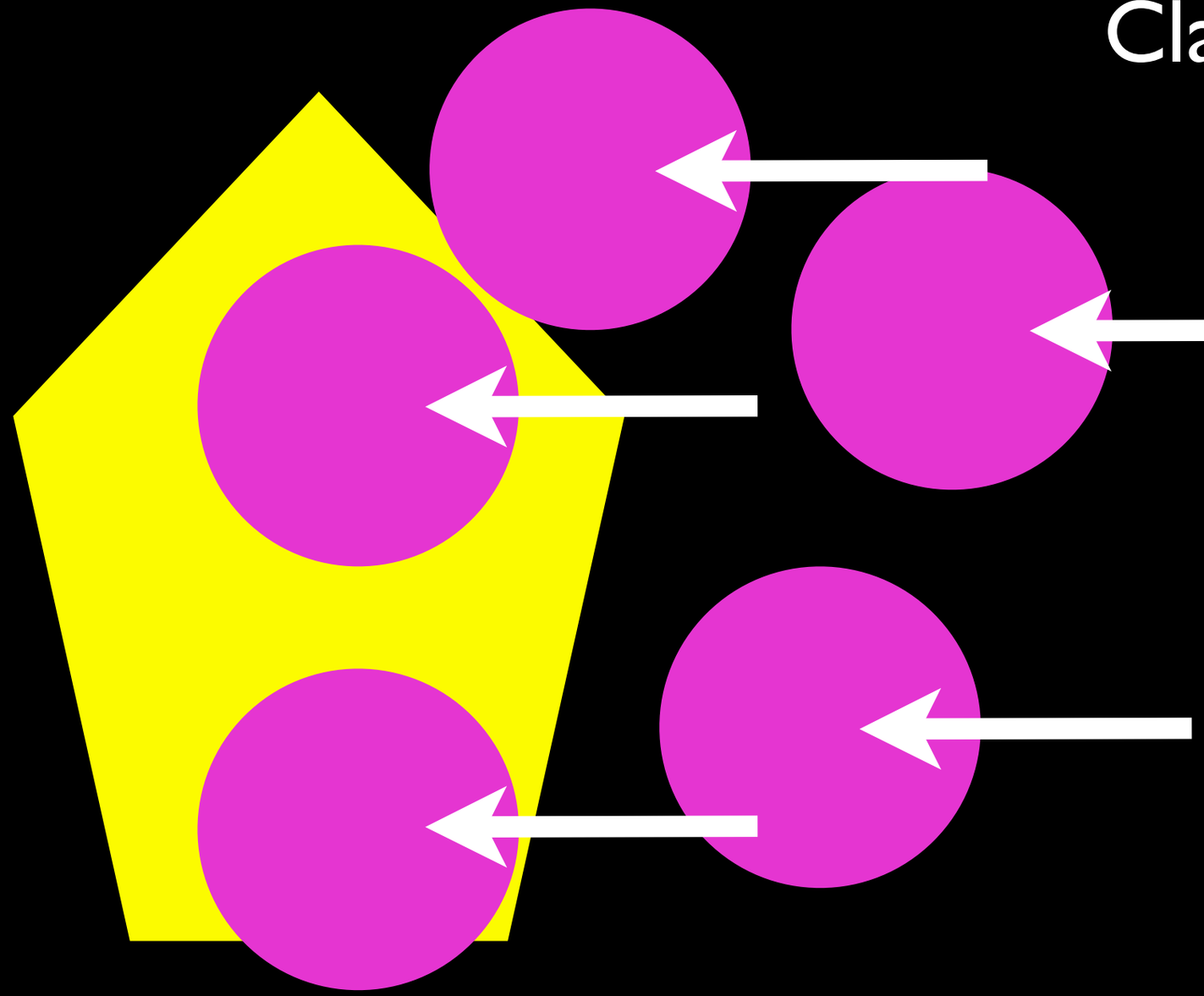
```
class Foo {
    @Annotation
    public static final int bar (String [])
    {
        ...
    }
}
```



Classloaders

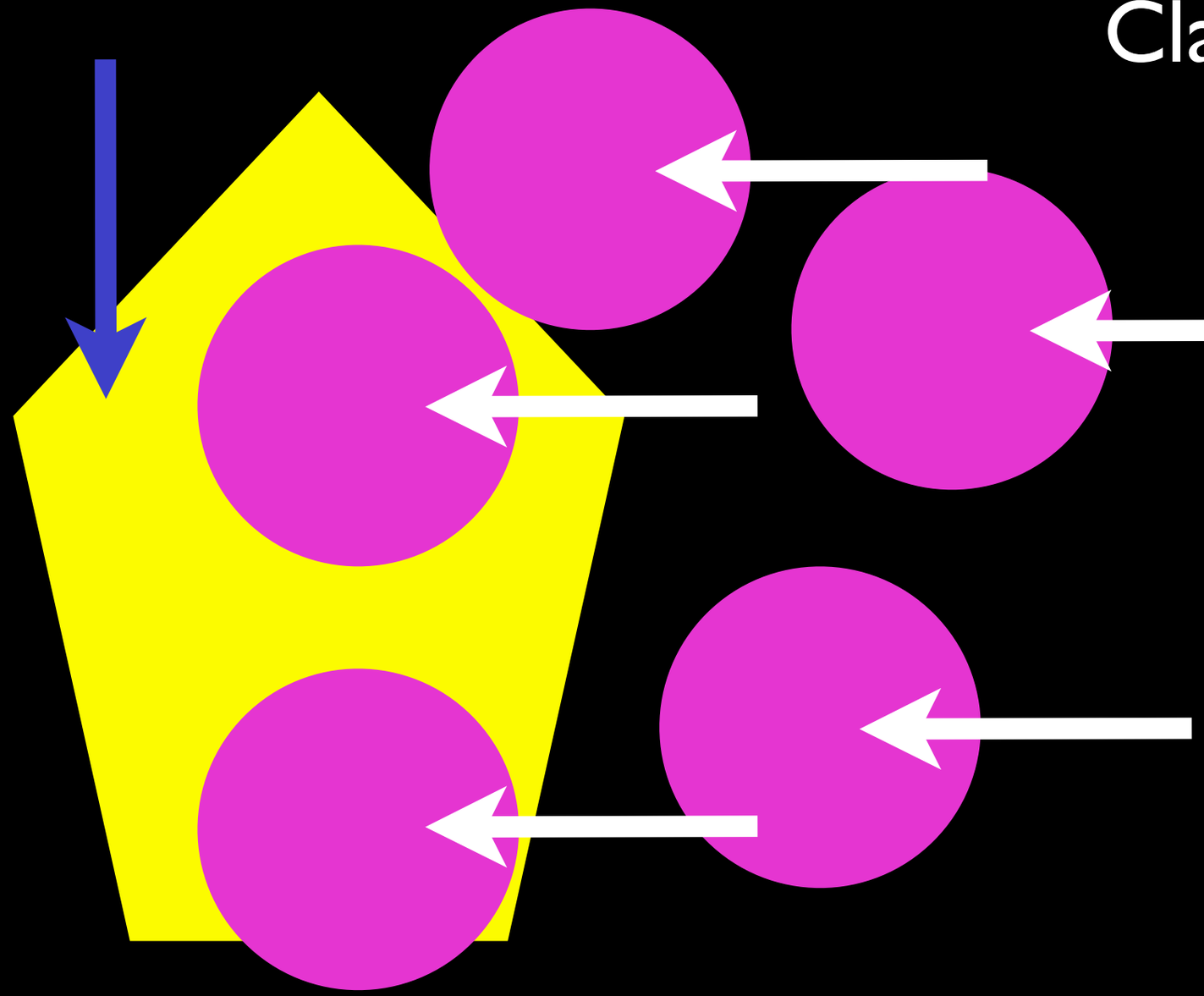


Classloaders



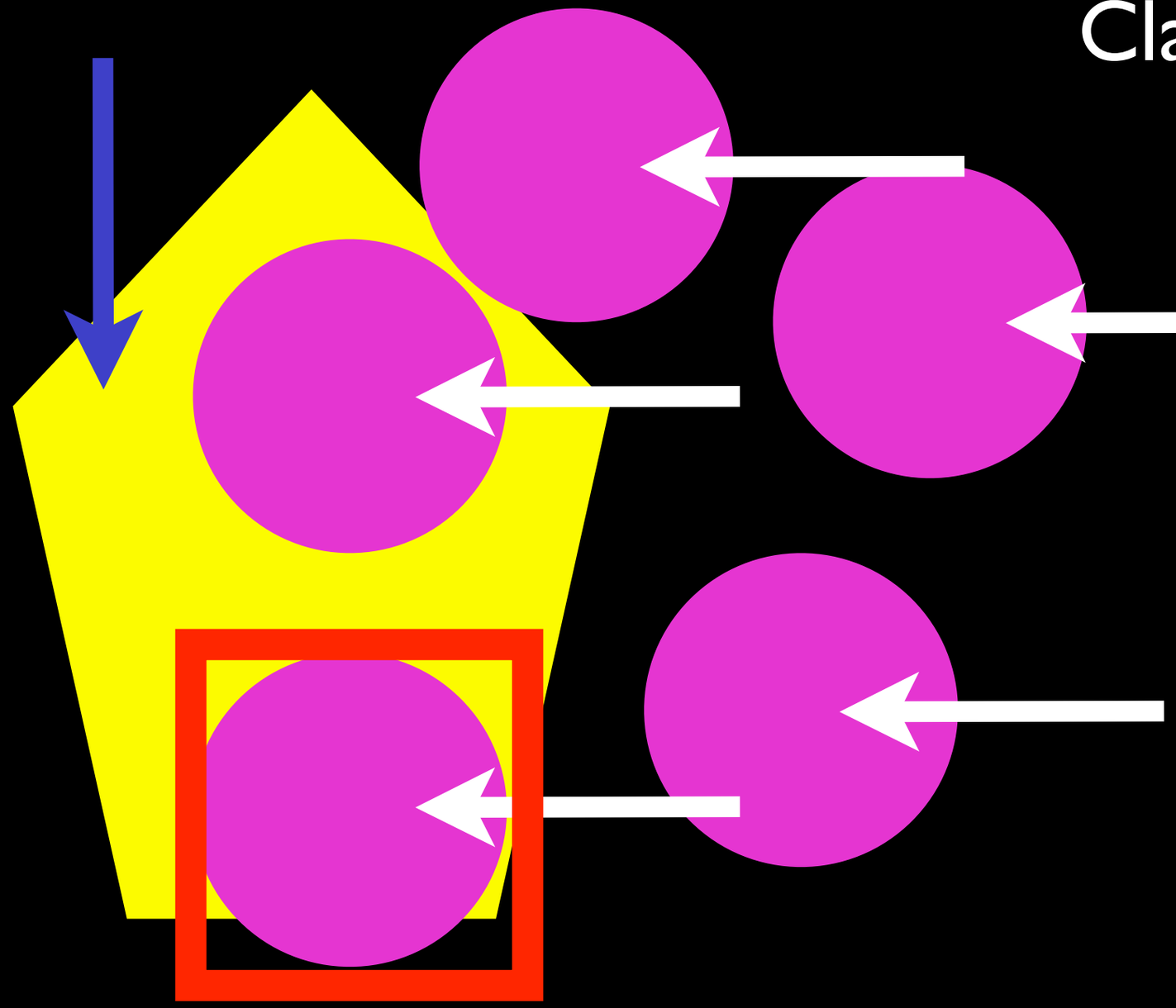
Retention Policies

Classloaders



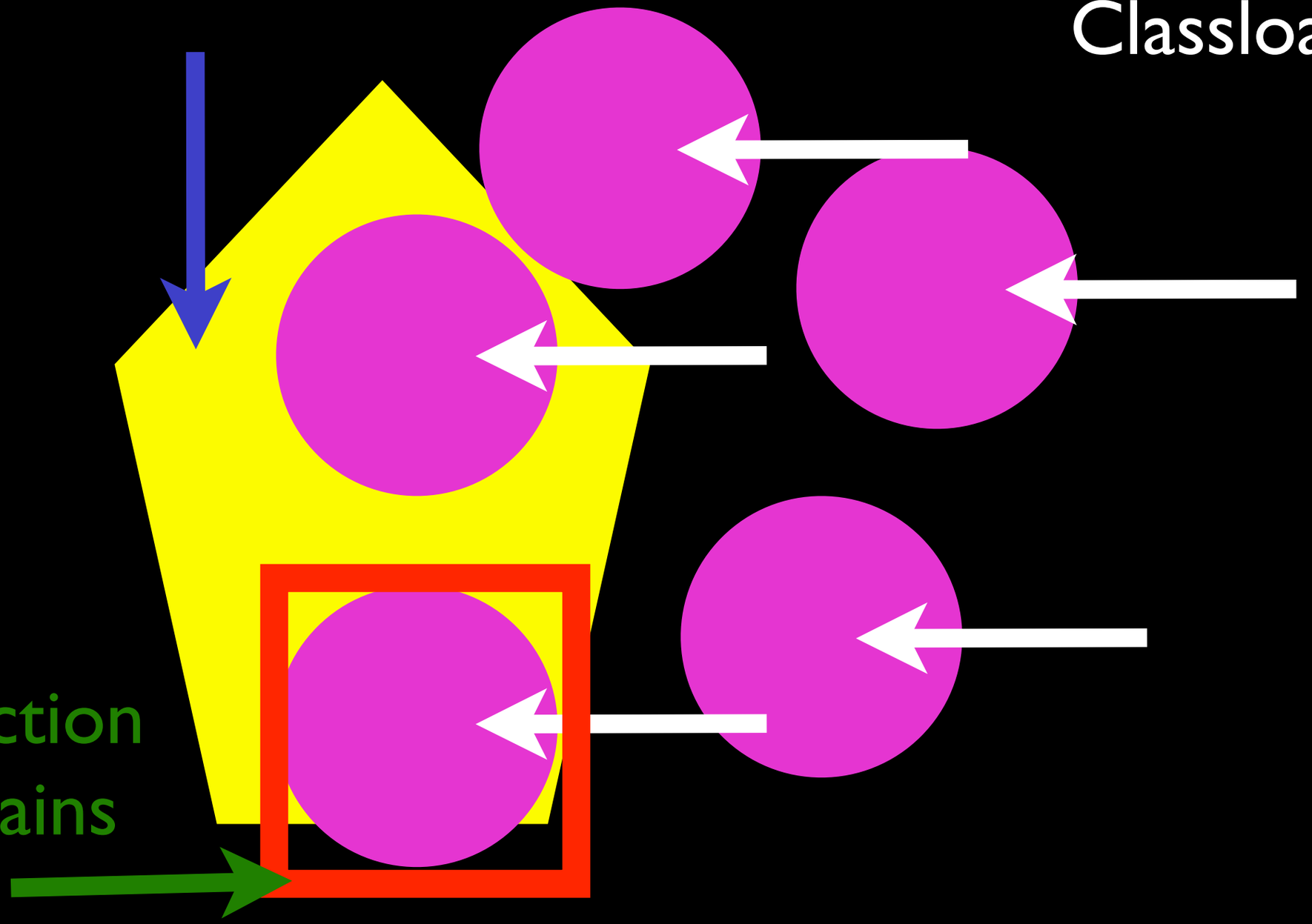
Retention Policies

Classloaders



Retention Policies

Classloaders

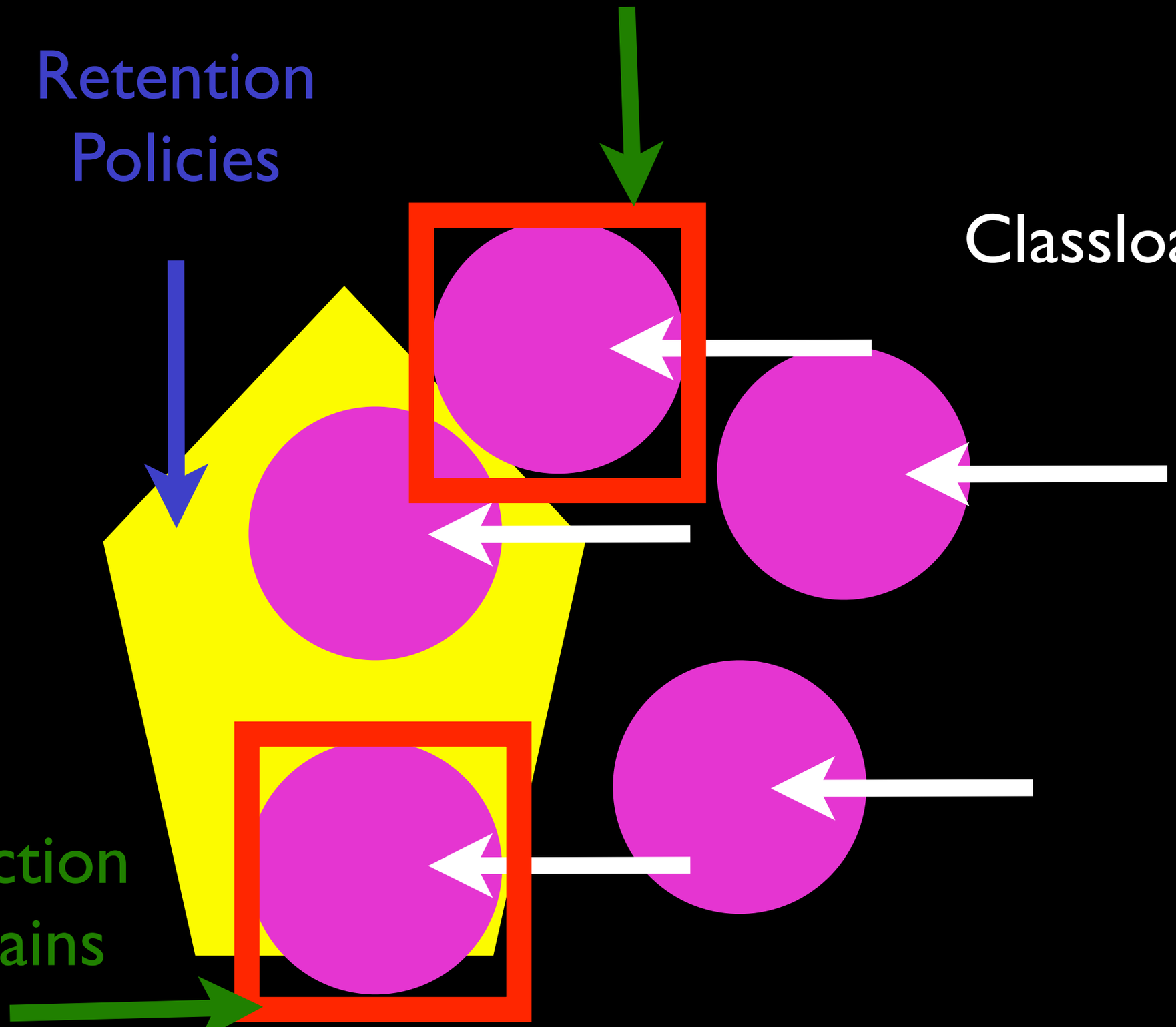


Protection Domains

Retention Policies

Classloaders

Protection Domains



Retention Policies

Classloaders

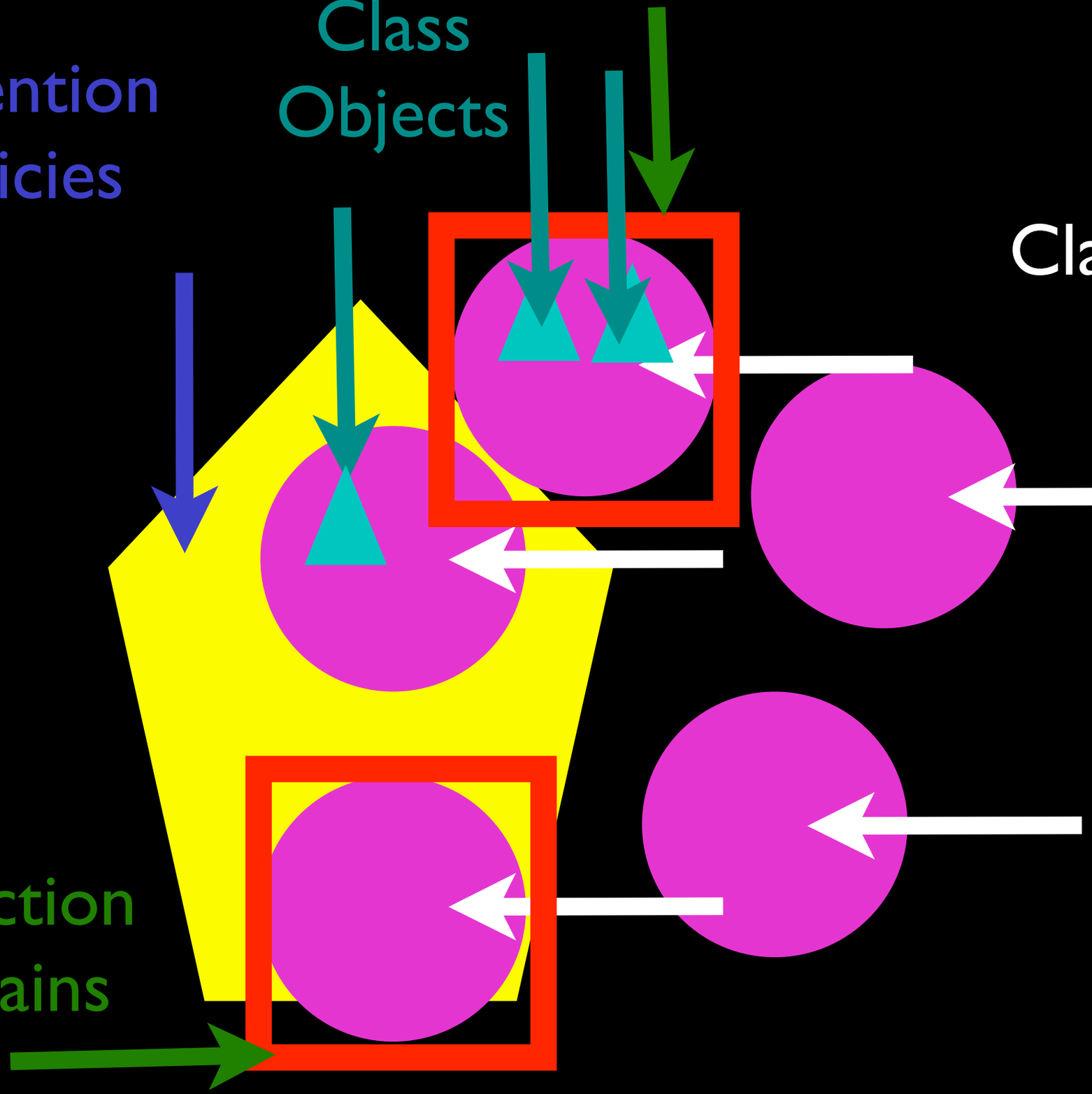
Protection Domains

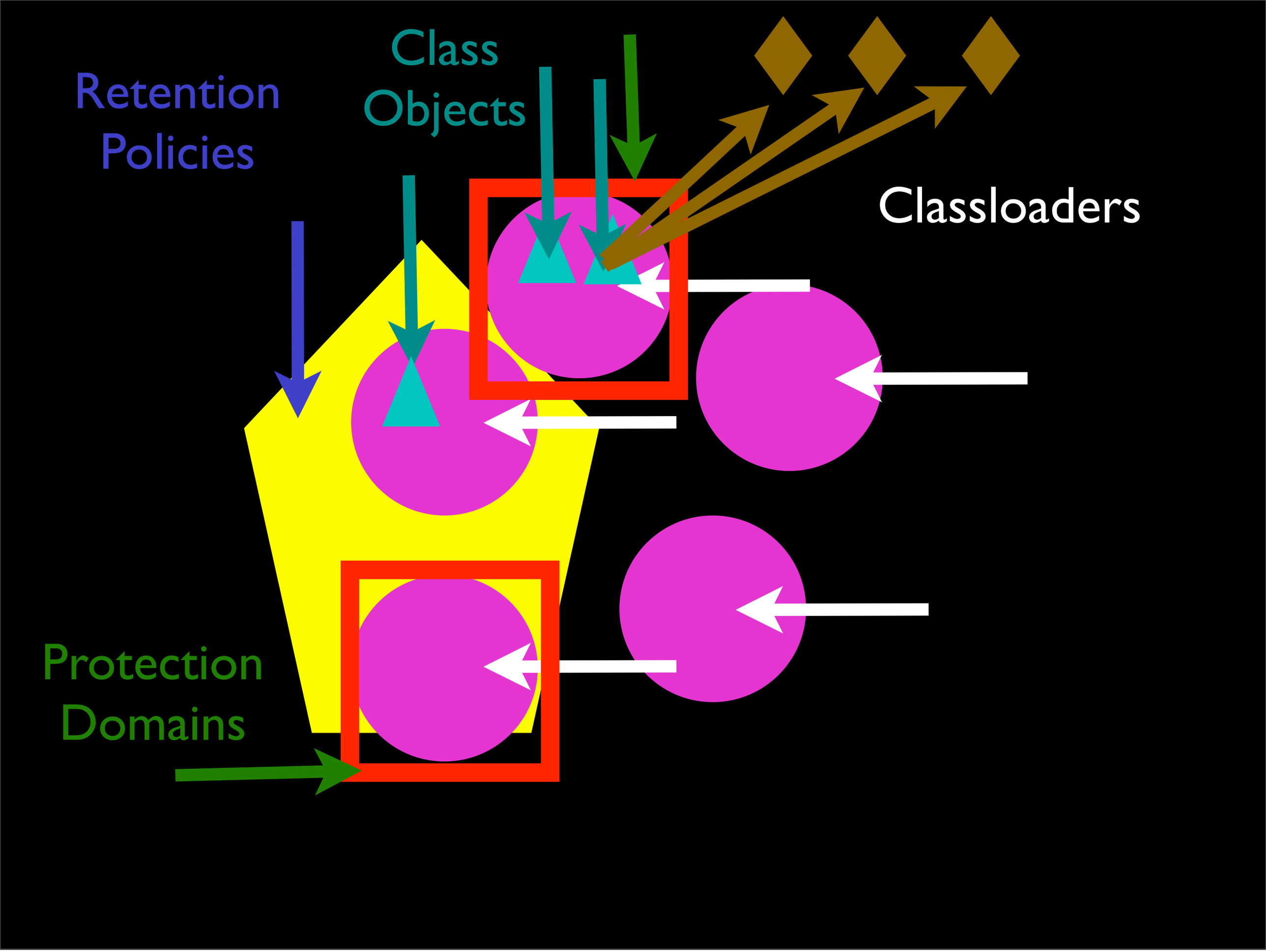
Retention Policies

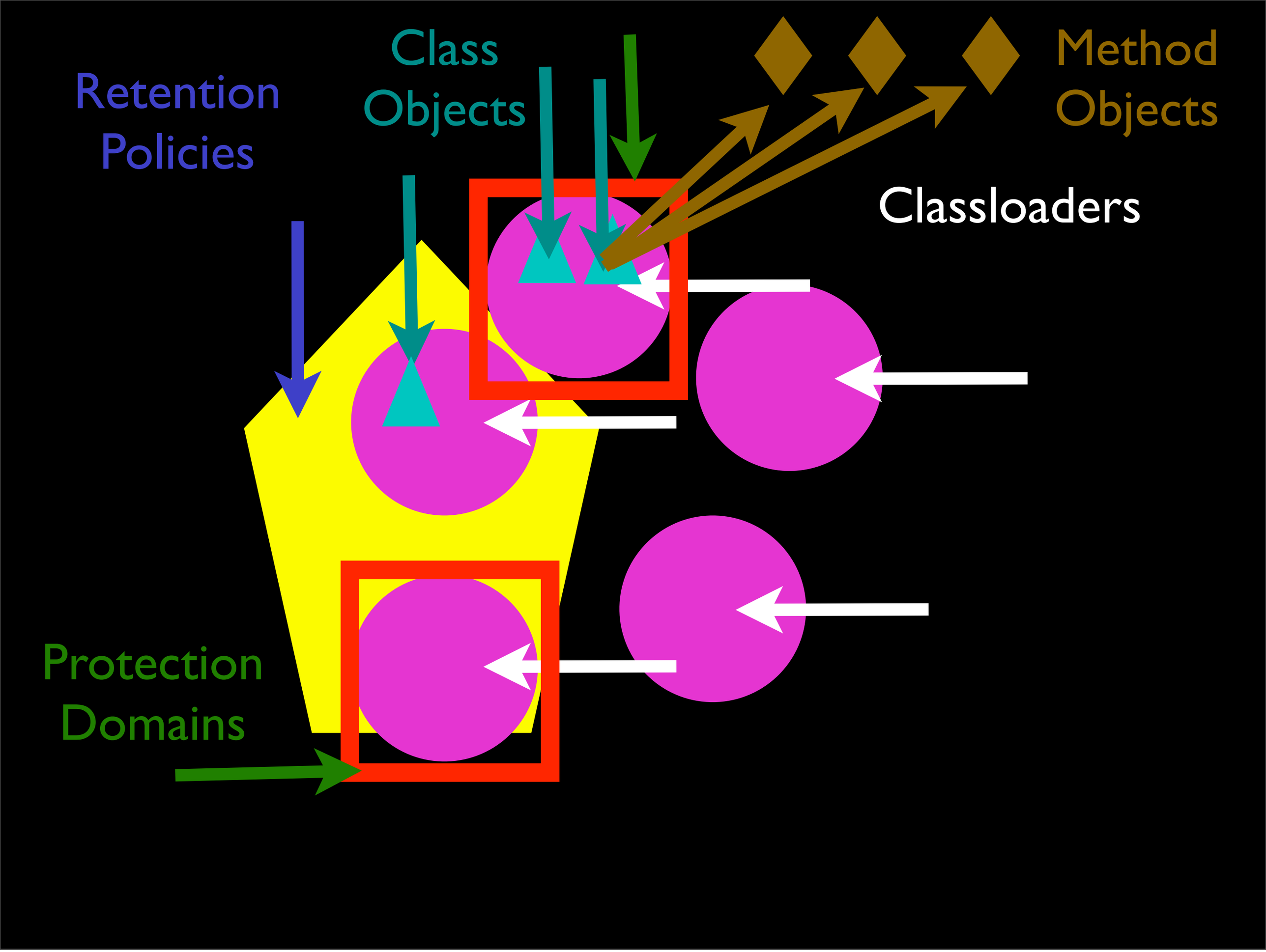
Class Objects

Classloaders

Protection Domains




Retention Policies
Class Objects
Classloaders
Protection Domains


Retention Policies
Class Objects
Method Objects
Classloaders
Protection Domains

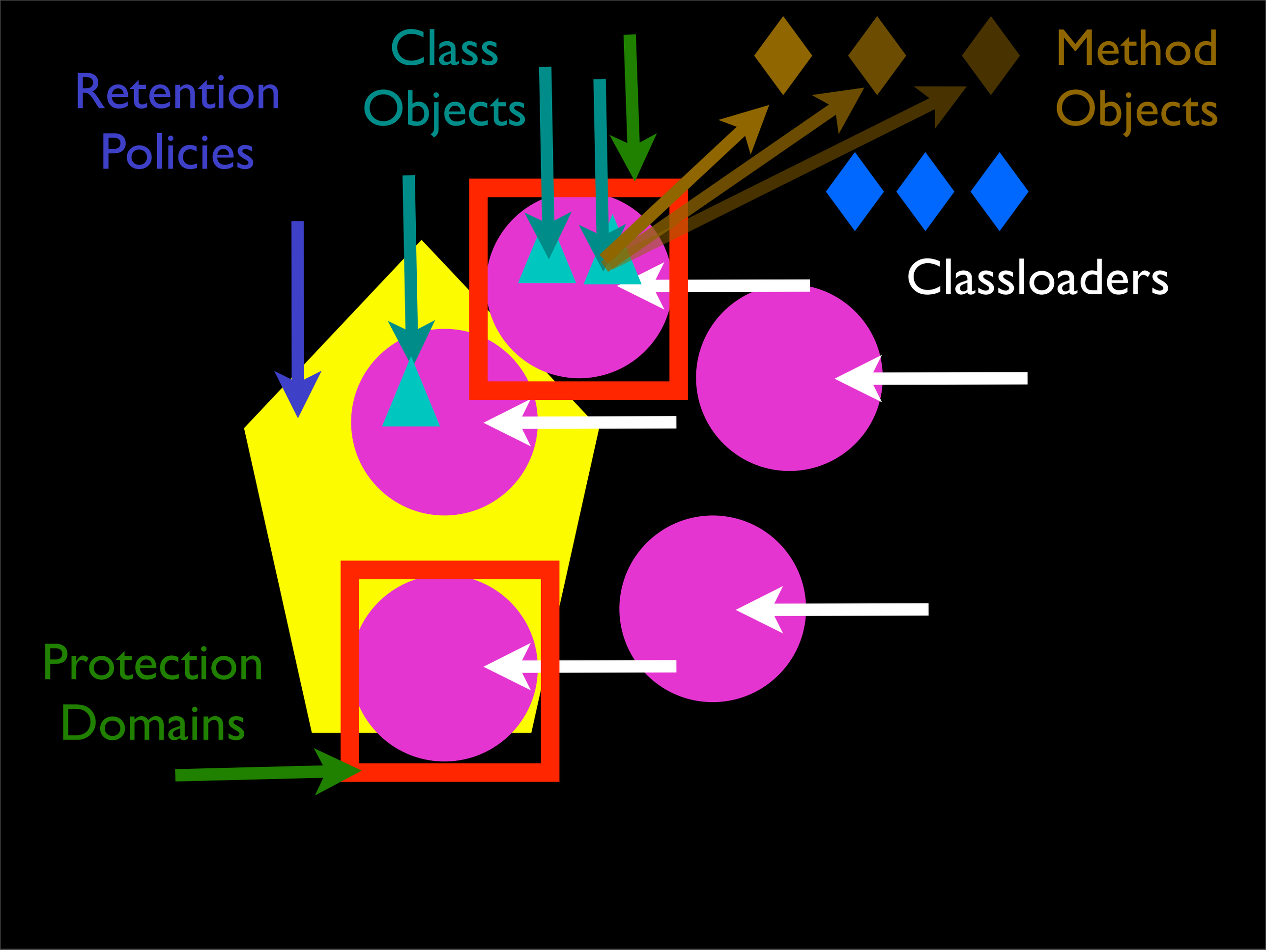
Retention Policies

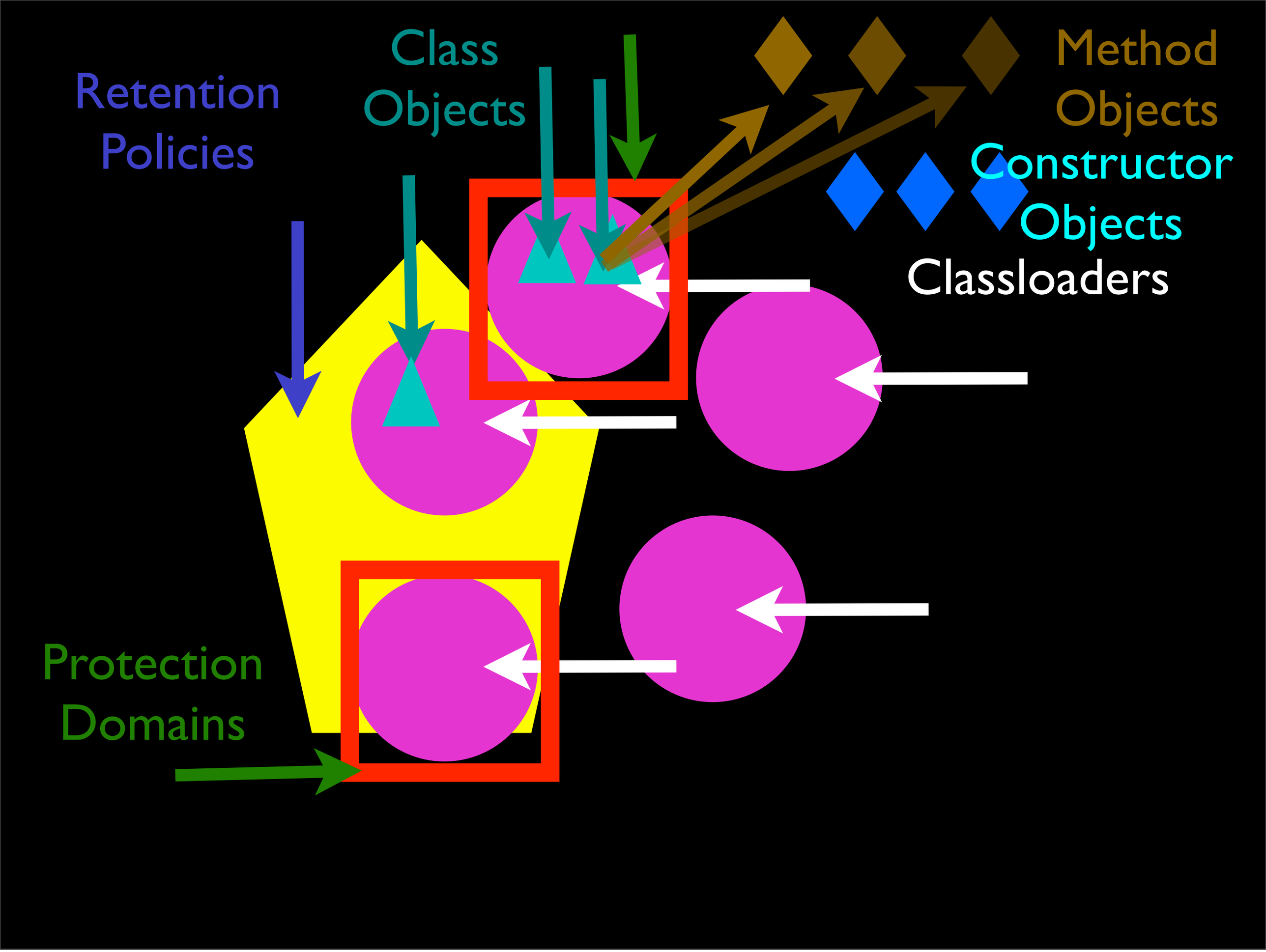
Class Objects

Method Objects

Classloaders

Protection Domains


Retention Policies
Class Objects
Method Objects
Classloaders
Protection Domains


Retention Policies
Class Objects
Method Objects
Constructor Objects
Classloaders
Protection Domains

Retention Policies

Class Objects

Method Objects

Constructor Objects

Classloaders

Protection Domains

Retention Policies

Class Objects

Method Objects

Constructor Objects

Classloaders

Protection Domains

Field Objects

Retention Policies

Class Objects

Method Objects

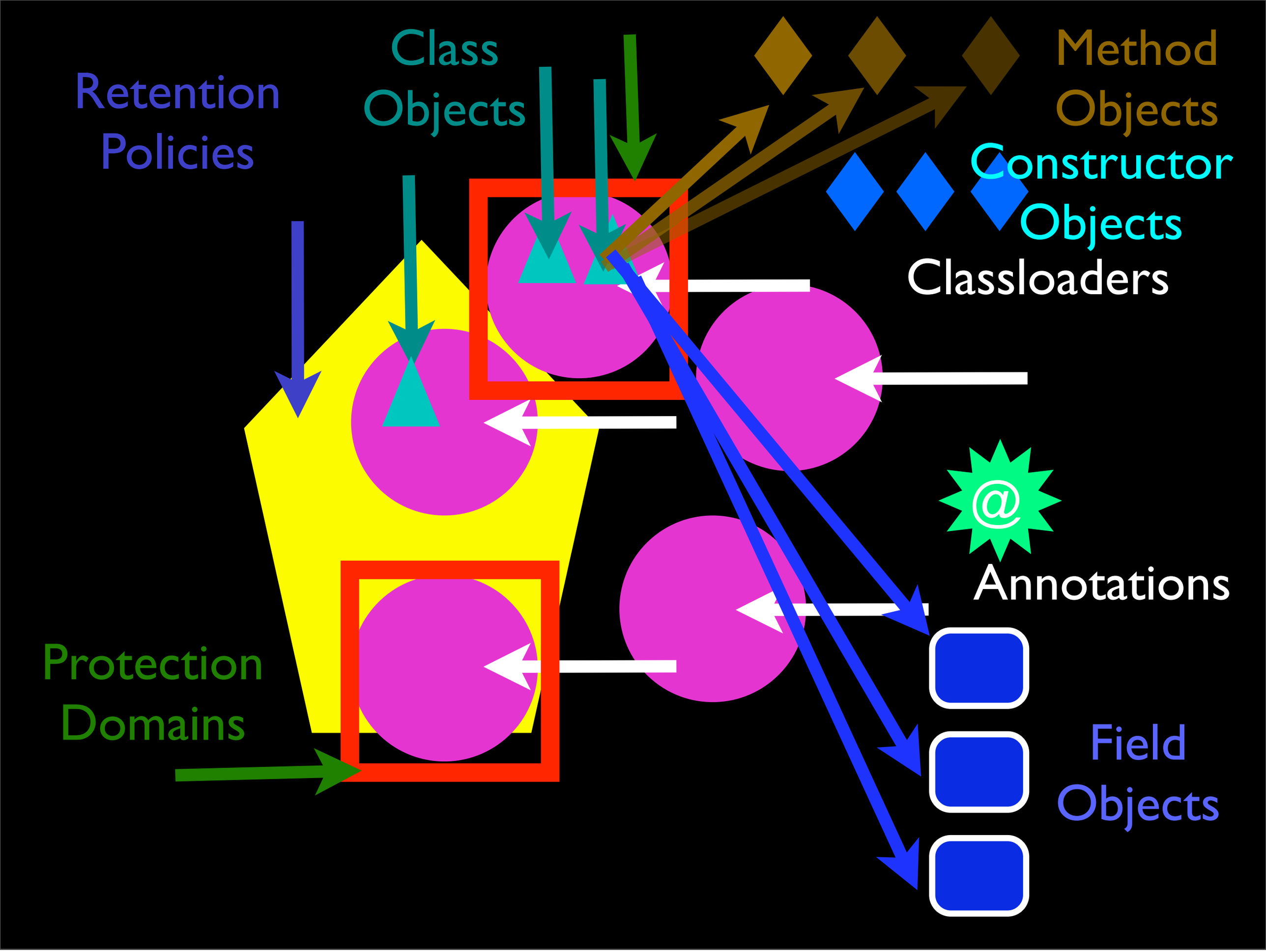
Constructor Objects

Classloaders

@

Protection Domains

Field Objects


Retention Policies
Class Objects
Method Objects
Constructor Objects
Classloaders
@
Annotations
Protection Domains
Field Objects

? ? ?

Why so many ways?

Syntactic sugar makes the medicine go down.

3 + 4

(3).__add__(4)

(or, actually...)

```
m = getattr(type(3), '__add__', None)
if m is not None:
    return m(3, 4)
else:
    m = getattr(type(4), '__radd__', None)
    return m(4, 3)
```

Why so much flexibility?

Anything Python does,
you can do too.

Decimal

```
>>> from decimal import Decimal
>>> a = Decimal("3")
>>> b = Decimal("4")
>>> a + b
Decimal('7')
```

Fraction

```
>>> from fractions import Fraction
>>> a = Fraction(3, 1)
>>> b = Fraction(4, 1)
>>> a / b
Fraction(3, 4)
```

3 basic types
of “thing”

Noun

Being Verb

Action Verb

Noun:

object

Nouns:

X

Nouns:

Instance

Nouns:

Class

Nouns:

Function

Nouns:

Method

Nouns:

Module

Nouns:

Built-In Types

Action verb:
method call

Action Verb:

```
...(): __call__
```

Action Verb:

```
+: .__add__()
-: .__sub__()
*: .__mul__()
/: .__div__()
[]: .__getitem__()
```

Being Verb: Assignment

IT'S

Being Verb:

```
x = 1
```

↓

$x = 1$

Being Verb:

```
def b():
    return 1
```

↓

```
b = types.FunctionType(
        types.CodeType(...),
        globals(),
        “b”)
```

Being Verb:

```
class X(A, B):
    y = 1
```

↓

```
d = {“y”: 1}
metaclass = ...
X = metaclass(“X”, (A, B), d)
```

Being Verb:

```
class X(A, B):
    y = 1
```

↓

```
d = {“y”: 1}
metaclass = ...
X = metaclass(“X”, (A, B), d)
```

Woo, Python 3!

Being Verb:

```
class X(A, B):
    y = 1
```

↓

```
d = {“y”: 1}
metaclass = ...
X = metaclass(“X”, (A, B), d)
```

Being Verb:

```
import x
```

↓

```
x = __import__(“x”, ...)
```

Being Verb:

```
@a
def b():
    pass
```

↓

```
def b():
    pass
b = a(b)
```

What happens when you ferment syntactic sugar?

Syntactic Alcohol!

(drink responsibly)

Decimal & Fraction (Numbers Flambé)

(this slide intentionally left blank)

How does it work?

```
class Number(object):
    def __init__(self, value):
        self.value = value

    def __add__(self, other):
        return Number(str(int(self.value) +
                          int(other.value)))

    def __repr__(self):
        return "Number({value})".format(
            value=repr(self.value))

print Number("3") + Number("4")
```

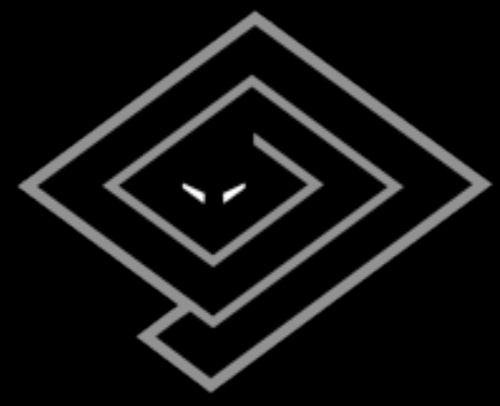
```
class Number(object):
    def __init__(self, value):
        self.value = value

    def __add__(self, other):
        return Number(str(int(self.value) +
                          int(other.value)))

    def __repr__(self):
        return "Number({value})".format(
            value=repr(self.value))

print(Number("3") + Number("4")
```

Higher-Order Functions

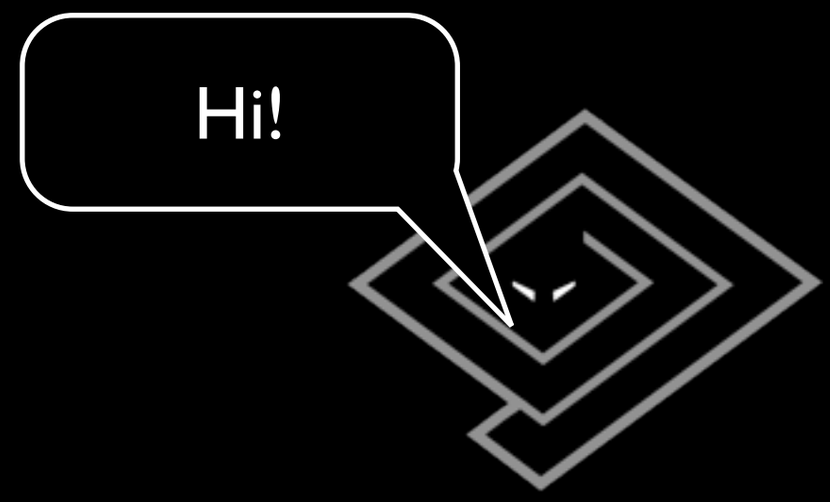
(functions that call functions)

Deferred
(Events on the Rocks)

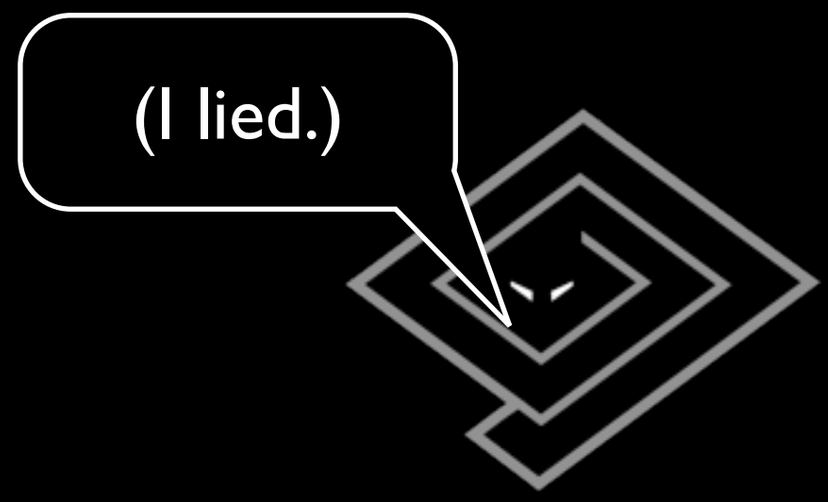
Deferred
(Events on the Rocks)



Deferred (Events on the Rocks)



Deferred (Events on the Rocks)



```
>>> from twisted.internet.defer import Deferred
>>> d = Deferred()
>>> def a(result):
...     print 'OK!', result
...     return result + 1
...
>>> d.addCallback(a)
<Deferred at 0x100509bd8>
>>> d.callback(3)
OK! 3
>>> d.addCallback(a)
OK! 4
<Deferred at 0x100509bd8  current result: 5>
```

How does it work?

```
# -*- test-case-name:
twisted.test.test_defer,twisted.test.test_de
fgen,twisted.internet.test.test_inlinecb -*-
# Copyright (c) 2001-2010 Twisted Matrix
Laboratories.
# See LICENSE for details.

"""
Support for results that aren't immediately
available.

Maintainer: Glyph Lefkowitz
"""

import traceback
import warnings
from sys import exc_info

# Twisted imports
from twisted.python import log, failure,
lockfile
from twisted.python.util import unsignedID,
mergeFunctionMetadata

class AlreadyCalledError(Exception):
    pass

class TimeoutError(Exception):
    pass

def logError(err):
    log.err(err)
    return err

def succeed(result):
    """
    Return a Deferred that has already had
'.callback(result)' called.

    This is useful when you're writing
synchronous code to an
    asynchronous interface: i.e., some code
is calling you expecting a
    Deferred result, but you don't actually
need to do anything
    asynchronous. Just return defer.succeed
(theResult).

    See L{fail} for a version of this
function that uses a failing
    Deferred rather than a successful one.

    @param result: The result to give to the
Deferred's 'callback'
           method.

    @rtype: L{Deferred}
    """
    d = Deferred()
    d.callback(result)
    return d


def fail(result=None):
    """
    Return a Deferred that has already had
'.errback(result)' called.

    See L{succeed}'s docstring for
rationale.

    @param result: The same argument that L
{Deferred.errback} takes.

    @raise NoCurrentExceptionError: If C
{result} is C{None} but there is no
        current exception state.

    @rtype: L{Deferred}
    """
    d = Deferred()
    d.errback(result)
    return d

def execute(callable, *args, **kw):
    """Create a deferred from a callable and
arguments.

    Call the given function with the given
arguments.  Return a deferred which
    has been fired with its callback as the
result of that invocation or its
    errback with a Failure for the exception
thrown.
    """
    try:
        result = callable(*args, **kw)
    except:
        return fail()
    else:
        return succeed(result)

def maybeDeferred(f, *args, **kw):
    """Invoke a function that may or may not
return a deferred.

    Call the given function with the given
arguments.  If the returned
    object is a C{Deferred}, return it.  If
the returned object is a C{Failure},
    wrap it with C{fail} and return it.
Otherwise, wrap it in C{succeed} and
    return it.  If an exception is raised,
convert it to a C{Failure}, wrap it
    in C{fail}, and then return it.

    @type f: Any callable
    @param f: The callable to invoke

    @param args: The arguments to pass to C
{f}
    @param kw: The keyword arguments to pass
to C{f}

    @rtype: C{Deferred}
    @return: The result of the function
call, wrapped in a C{Deferred} if
    necessary.
    """
    try:
        result = f(*args, **kw)
    except:
        return fail(failure.Failure())

    if isinstance(result, Deferred):
        return result
    elif isinstance(result,
failure.Failure):
        return fail(result)
    else:
        return succeed(result)

def timeout(deferred):
    deferred.errback(failure.Failure
(TimeoutError("Callback timed out")))

def passthru(arg):
    return arg

def setDebugging(on):
    """Enable or disable Deferred debugging.

    When debugging is on, the call stacks
from creation and invocation are
    recorded, and added to any
AlreadyCalledErrors we raise.
    """
    Deferred.debug=bool(on)

def getDebugging():
    """Determine whether Deferred debugging
is enabled.
    """
    return Deferred.debug

class Deferred:
    """This is a callback which will be put
off until later.

    Why do we want this? Well, in cases
where a function in a threaded
    program would block until it gets a
result, for Twisted it should
    not block. Instead, it should return a
Deferred.

    This can be implemented for protocols
that run over the network by
    writing an asynchronous protocol for
twisted.internet. For methods
    that come from outside packages that are
not under our control, we use
    threads (see for example L
{twisted.enterprise.adbapi}).

    For more information about Deferreds,
see doc/howto/defer.html or
    U{http://twistedmatrix.com/projects/
core/documentation/howto/defer.html}
    """
    called = 0
    paused = 0
    timeoutCall = None
    _debugInfo = None

    # Are we currently running a user-
installed callback?  Meant to prevent
    # recursive running of callbacks when a
reentrant call to add a callback is
    # used.
    _runningCallbacks = False

    # Keep this class attribute for now, for
compatibility with code that
    # sets it directly.
    debug = False

    def __init__(self):
        self.callbacks = []
        if self.debug:
            self._debugInfo = DebugInfo()
            self._debugInfo.creator =
traceback.format_stack()[:-1]

    def addCallbacks(self, callback,
errback=None,
                     callbackArgs=None,
callbackKeywords=None,
                     errbackArgs=None,
errbackKeywords=None):
        """Add a pair of callbacks (success
and error) to this Deferred.

        These will be executed when the
'master' callback is run.
        """
        assert callable(callback)
        assert errback == None or callable
(errback)
        cbs = ((callback, callbackArgs,
callbackKeywords),
               (errback or (passthru),
errbackArgs, errbackKeywords))
        self.callbacks.append(cbs)

        if self.called:
            self._runCallbacks()
        return self

    def addCallback(self, callback, *args,
**kw):
        """Convenience method for adding
just a callback.

        See L{addCallbacks}.
        """
        return self.addCallbacks(callback,
callbackArgs=args,
callbackKeywords=kw)

    def addErrback(self, errback, *args,
**kw):
        """Convenience method for adding
just an errback.

        See L{addCallbacks}.
        """
        return self.addCallbacks(passthru,
errback,

errbackArgs=args,

errbackKeywords=kw)

    def addBoth(self, callback, *args,
**kw):
        """Convenience method for adding a
single callable as both a callback
        and an errback.

        See L{addCallbacks}.
        """
        return self.addCallbacks(callback,
callback,

callbackArgs=args, errbackArgs=args,

callbackKeywords=kw, errbackKeywords=kw)

    def chainDeferred(self, d):
        """Chain another Deferred to this
Deferred.

        This method adds callbacks to this
Deferred to call d's callback or
        errback, as appropriate. It is
merely a shorthand way of performing
        the following::

            self.addCallbacks(d.callback,
d.errback)

        When you chain a deferred d2 to
another deferred d1 with
        d1.chainDeferred(d2), you are making
d2 participate in the callback
        chain of d1. Thus any event that
fires d1 will also fire d2.
        However, the converse is B{not}
true; if d2 is fired d1 will not be
        affected.
        """
        return self.addCallbacks(d.callback,
d.errback)

    def callback(self, result):
        """Run all success callbacks that
have been added to this Deferred.

        Each callback will have its result
passed as the first
        argument to the next; this way, the
callbacks act as a
        'processing chain'. Also, if the
success-callback returns a Failure
        or raises an Exception, processing
will continue on the *error*-
        callback chain.
        """
        assert not isinstance(result,
Deferred)
        self._startRunCallbacks(result)


    def errback(self, fail=None):
        """
        Run all error callbacks that have
been added to this Deferred.

        Each callback will have its result
```

Oops! I mean...

```
class Event(object):
    def __init__(self):
        self.functions = []

    def whenDone(self, somethingToDo):
        self.functions.append(somethingToDo)

    def done(self, result):
        for function in self.functions:
            result = function(result)
```

```
class Event(object):
    def __init__(self):
        self.functions = []

    def whenDone(self, somethingToDo):
        self.functions.append(somethingToDo)

    def done(self, result):
        for function in self.functions:
            result = function(result)
```

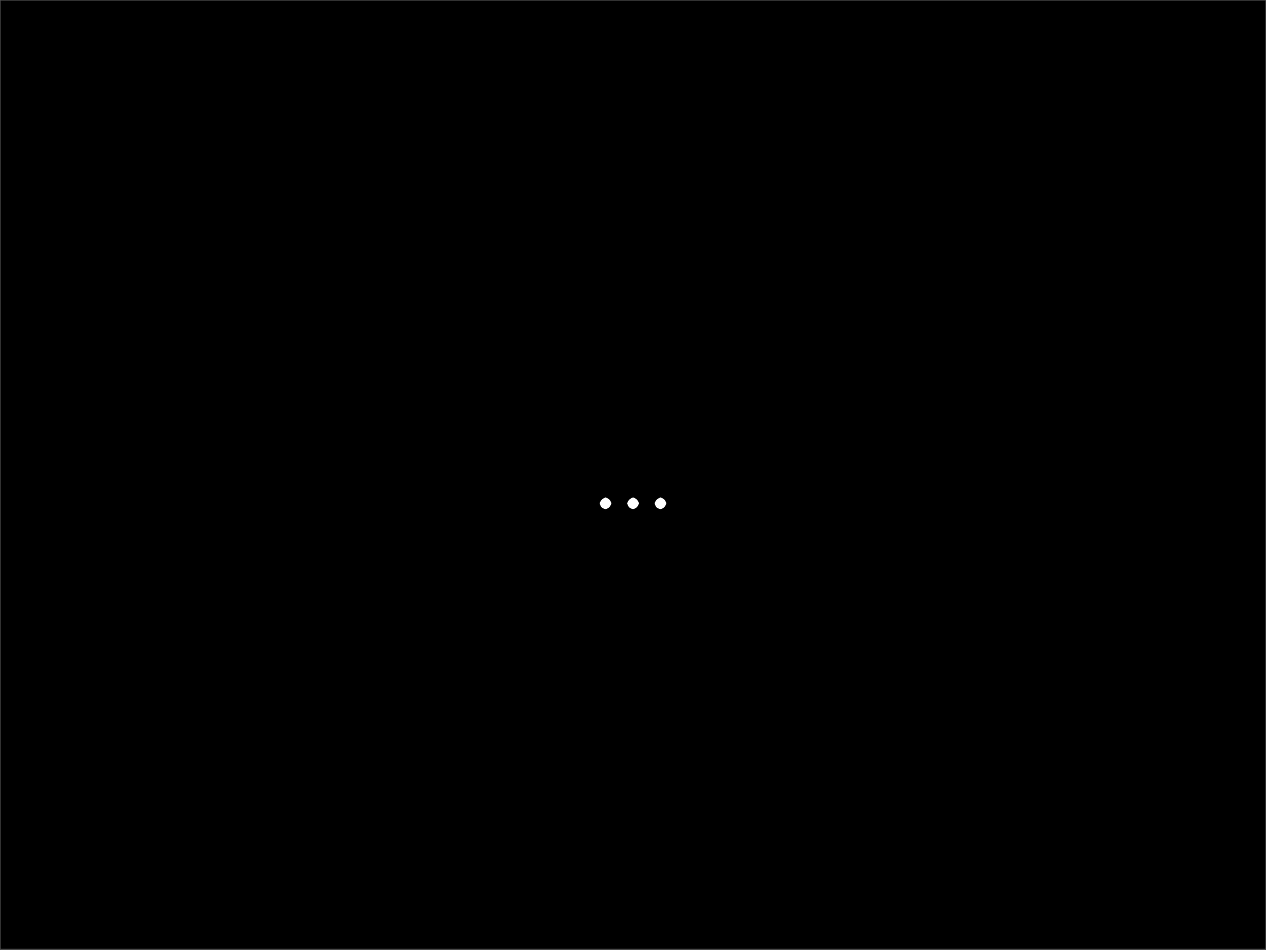
```
e = Event()
def a(result):
    print 'OK!', result
    return result + 1
e.whenDone(a)
e.whenDone(a)
e.done(3)

# prints...
OK! 3
OK! 4
```

Zope Interface: (Class Daquiri)

```
>>> from zope.interface import Interface
>>> class ISomething(Interface):
...     def something():
...         "Something."
...
>>> ISomething
<InterfaceClass __main__.ISomething>
```

```
>>> list(ISomething)
['something']
>>> hasattr(ISomething, 'something')
False
>>> ISomething['something']
<zope.interface.interface.Method object at 0x100597390>
```

How does it work?



```
import types

class DescriptionType(object):
    def __init__(self, names):
        self.names = names
    def __repr__(self):
        return 'DescriptionType({names})'.format(
            names=repr(self.names))

class MetaDescription(type):
    def __new__(cls, name, bases, namespace):
        if name == 'Description':
            return super(MetaDescription, cls).__new__(
                cls, name, bases, namespace)
        names = []
        for name in namespace:
            if isinstance(namespace[name],
                          types.FunctionType):
                names.append(name)
        return DescriptionType(names)

class Description(object):
    __metaclass__ = MetaDescription
```

```
import types

class DescriptionType(object):
    def __init__(self, names):
        self.names = names
    def __repr__(self):
        return 'DescriptionType({names})'.format(
            names=repr(self.names))

class MetaDescription(type):
    def __new__(cls, name, bases, namespace):
        if name == 'Description':
            return super(MetaDescription, cls).__new__(
                cls, name, bases, namespace)
        names = []
        for name in namespace:
            if isinstance(namespace[name],
                          types.FunctionType):
                names.append(name)
        return DescriptionType(names)

class Description(object):
    __metaclass__ = MetaDescription
```

```
import types

class DescriptionType(object):
    def __init__(self, names):
        self.names = names
    def __repr__(self):
        return 'DescriptionType({names})'.format(
            names=repr(self.names))

class MetaDescription(type):
    def __new__(cls, name, bases, namespace):
        if name == 'Description':
            return super(MetaDescription, cls).__new__(
                cls, name, bases, namespace)
        names = []
        for name in namespace:
            if isinstance(namespace[name],
                          types.FunctionType):
                names.append(name)
        return DescriptionType(names)

class Description(metaclass=MetaDescription):
    pass
```

```
import types

class DescriptionType(object):
    def __init__(self, names):
        self.names = names
    def __repr__(self):
        return 'DescriptionType({names})'.format(
            names=repr(self.names))

class MetaDescription(type):
    def __new__(cls, name, bases, namespace):
        if name == 'Description':
            return super(MetaDescription, cls).__new__(
                cls, name, bases, namespace)
        names = []
        for name in namespace:
            if isinstance(namespace[name],
                          types.FunctionType):
                names.append(name)
        return DescriptionType(names)

class Description(metaclass=MetaDescription):
    pass
```

Woo, python 3!

```
class DescribeSomething(Description):
    def something(self):
        pass
    def somethingElse(self):
        pass
    notSomething = 4321

print DescribeSomething

# ...

DescriptionType(['somethingElse', 'something'])
```

Nevow's "Stan"
(HTML, Served neat)

```
>>> from nevow.flat import flatten
>>> from nevow.tags import p, b, i
>>> p[b["hello"], ", ", i["world!"]]
>>> flatten(p(id="paragraph")[b(id="bold")
    ["hello"], ", ", i["world!"]])
'<p id="paragraph"><b id="bold">hello</b>,
<i>world!</i></p>'
```

How does it work?

```
class Outline(object):
    def __init__(self, name):
        self.name = name
        self.nodes = []

    def __getitem__(self, files):
        self.nodes.extend(files)
        return self

    def show(self, indent=0):
        print((" " * indent * 4) +
              "* " + self.name)
        for node in self.nodes:
            node.show(indent+1)
```

```
class Outline(object):
    def __init__(self, name):
        self.name = name
        self.nodes = []

    def __getitem__(self, files):
        self.nodes.extend(files)
        return self

    def show(self, indent=0):
        print((" " * indent * 4) +
              "* " + self.name)
        for node in self.nodes:
            node.show(indent+1)
```

```
Outline("Top")[
    Outline("1"),
    Outline("2")[
        Outline("a"),
        Outline("b"),
        Outline("c"),
    ],
    Outline("3")
].show()
```

* Top
    * 1
    * 2
        * a
        * b
        * c
* 3

[twisted matrix]