Today in CS 42

"Stack Pointer" by Brett B.
There once was a doctor named Racket.
He could tell a paren from a bracket.
But his stack overflowed
with a spam overload;
128MB couldn't stack it."

Limerick by Jandro A.
A problem working in Racket:
Sometimes your code just won't hack it!
It ends up as spam;
"What stack pointer, man?"
And parsing? I want to smack it?

Haiku by Rojesh B.
Spam kills pigs each day
Racket kills the innocent
I killed stack pointer

Haiku by Henry H.
For function calling
Racket use parenthesis
Hmmm spam stack pointer

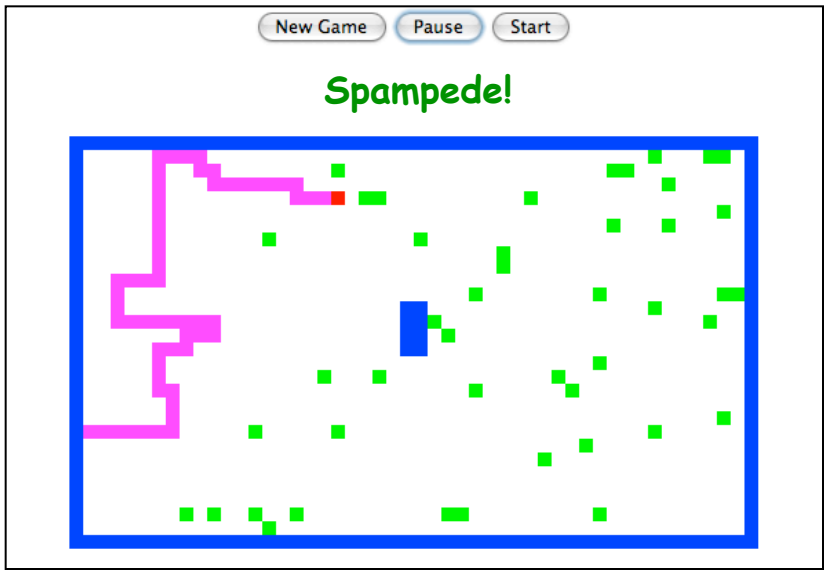
Software Engineering

strategies for software reuse

We support Green programming…

HW9: Last Python. But Beware!

Searching for SPAM



Objects and References

```
class Person(object):

    def __init__(self, name):
        self.name = name
        self.siblings = []

    def addSibling(self, sib):
        self.siblings.append(sib)

    def __repr__(self):
        return self.name
```

```
def doStuff(p):
    p.name = "Susan"
    p.addSibling(Person("Jaime"))

def doStuff2(name):
    name = "Aimee"

def main():
    p1 = Person("Harry")
    p2 = Person("Sally")
    p3 = p1

    p3.name = "George"

    print "p1 is", p1
```

This code will print:

A. p1 is George
B. p1 is Harry
C. Nothing, it will cause an error

Objects and References

```
class Person(object):

    def __init__(self, name):
        self.name = name
        self.siblings = []

    def addSibling(self, sib):
        self.siblings.append(sib)

    def __repr__(self):
        return self.name
```

```
def doStuff(p):
    p.name = "Susan"
    p.addSibling(Person("Jaime"))

def doStuff2(name):
    name = "Aimee"

def main():
    p1 = Person("Harry")
    p2 = Person("Sally")
    p3 = p1

    p3.name = "George"

    doStuff(p1)
    print "p3 is", p3, "with
         siblings", p3.siblings
```

This code will print:

A. p3 is George with siblings []
B. p3 is Susan with siblings [Jaime]
C. p3 is Susan with siblings []
D. Nothing, it will cause an error

Objects and References

```
class Person(object):

    def __init__(self, name):
        self.name = name
        self.siblings = []

    def addSibling(self, sib):
        self.siblings.append(sib)

    def __repr__(self):
        return self.name
```

```
def doStuff(p):
    p.name = "Susan"
    p.addSibling(Person("Jaime"))

def doStuff2(name):
    name = "Aimee"

def main():
    p1 = Person("Harry")
    p2 = Person("Sally")
    p3 = p1

    p3.name = "George"

    doStuff(p1)
    p1.siblings = []

    print "p3 is", p3, "with
         siblings", p3.siblings
```

This code will print:

A. p3 is George with siblings []
B. p3 is Susan with siblings [Jaime]
C. p3 is Susan with siblings []
D. Nothing, it will cause an error

Objects and References

```
class Person(object):

    def __init__(self, name):
        self.name = name
        self.siblings = []

    def addSibling(self, sib):
        self.siblings.append(sib)

    def __repr__(self):
        return self.name
```

```
def doStuff(p):
    p.name = "Susan"
    p.addSibling(Person("Jaime"))

def doStuff2(name):
    name = "Aimee"

def main():
    p1 = Person("Harry")
    p2 = Person("Sally")
    p3 = p1

    p3.name = "George"

    doStuff(p1)
    p1.siblings = []

    doStuff2(p2.name)
    print "p2 is", p2
```

This code will print:

A. p2 is Sally
B. p2 is Aimee
C. Nothing, it will cause an error

Objects and References

```
class Person(object):

    def __init__(self, name):
        self.name = name
        self.siblings = []

    def addSibling(self, sib):
        self.siblings.append(sib)

    def __repr__(self):
        return self.name
```

```
def doStuff(p):
    p.name = "Susan"
    p.addSibling(Person("Jaime"))

def doStuff2(name):
    name = "Aimee"

def main():
    p1 = Person("Harry")
    p2 = Person("Sally")
    p3 = p1

    p3.name = "George"

    doStuff(p1)
    p1.siblings = []

    doStuff2(p2.name)
    doStuff2(p1)
    print "p3 is", p3, "with
          siblings", p3.siblings
    print "p1 is", p1, "with
          siblings", p1.siblings
```

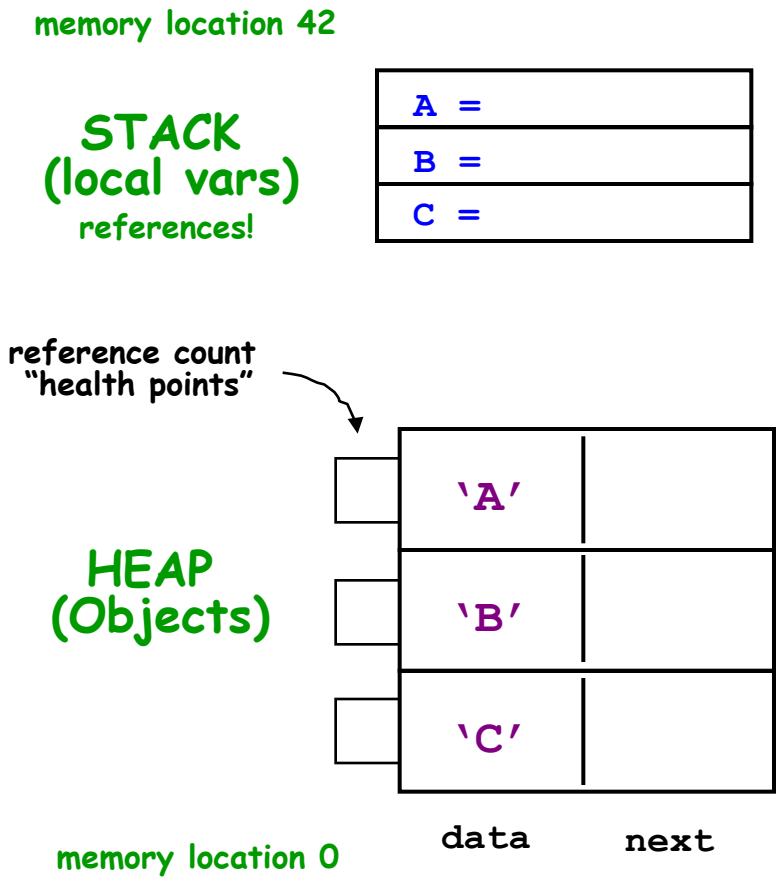
This code will print:

A. p3 is George with siblings []
   p1 is Aimee with siblings []
B. p3 is Susan with siblings [Jaime]
   p1 is Aimee with siblings []
C. p3 is Susan with siblings []
   p1 is Susan with siblings []
D. p3 is Susan with siblings [Jaime]
   p1 is Susan with siblings []
E. Nothing, it will cause an error

Talkin' Trash

Reference counting

```
def madEEmajor()
    A = QCell("A")
    if True:
        B = QCell("B")
        C = QCell("C")
        A.next = C
        B.next = C
    A.next = A
```

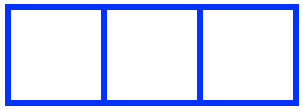


More of a good thing…

```
class QCell: # in a linked list
```


```
   def __init__(self, o):
      self.data = o
      self.next = None

   # repr also defined
```

```
class TCell:  # tree or doubly-LL
```




```
   def __init__(self, o):
      self.data = o
      self.next = None
      self.prev = None

   # repr also defined
```

twice the recursion, twice as fast ?

Talkin' Trash … Still

Doubly-linked lists can foil reference counters!

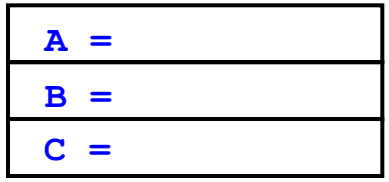
(*DLL*s that make sense!)

Mark-and-sweep

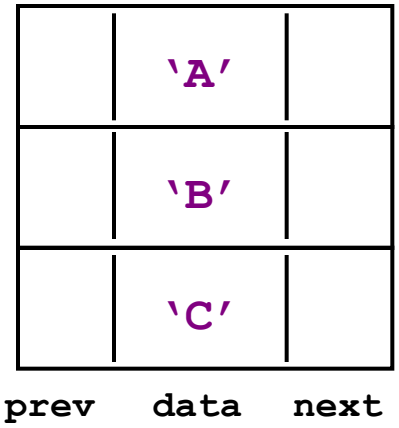
```
def madEEmajor():
    A = TCell('A')
    B = TCell('B')
    if True:
        C = TCell('C')
        A.prev = B
        A.next = C
        B.prev = C
        B.next = A
        C.prev = A
        C.next = B
```

memory location 42

STACK
(local vars)
references!



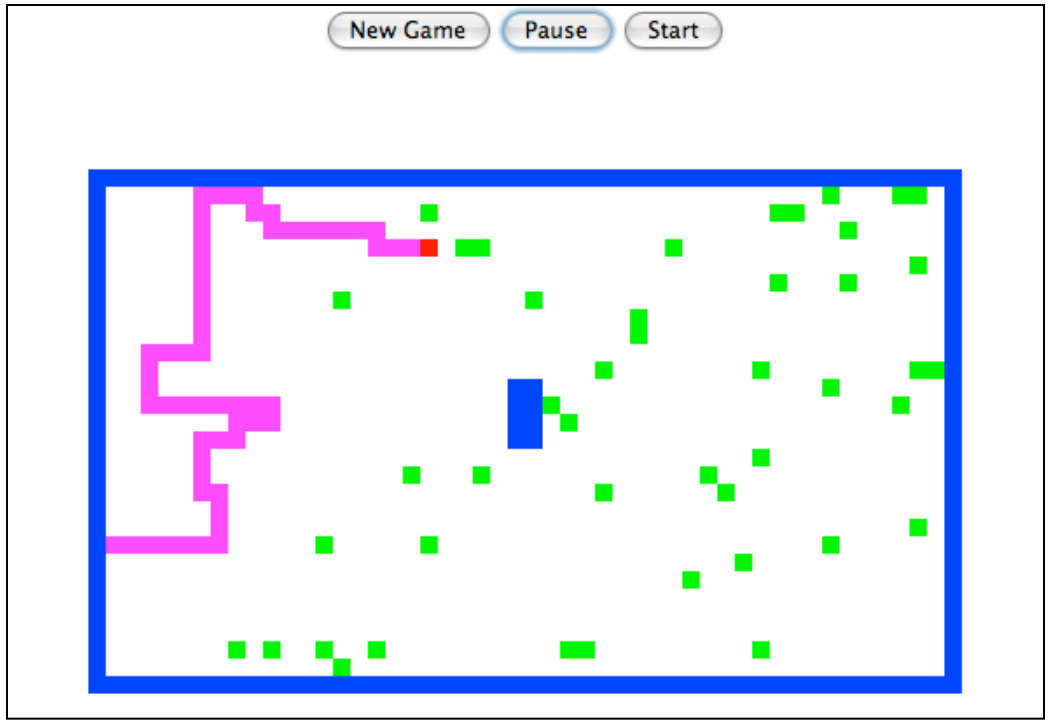
HEAP
(Objects)



memory location 0

Assignment 9! (Bye Bye Python)

- Due Monday, November 8
- Please start early!



Parts 1 and 2:
You are ready after today

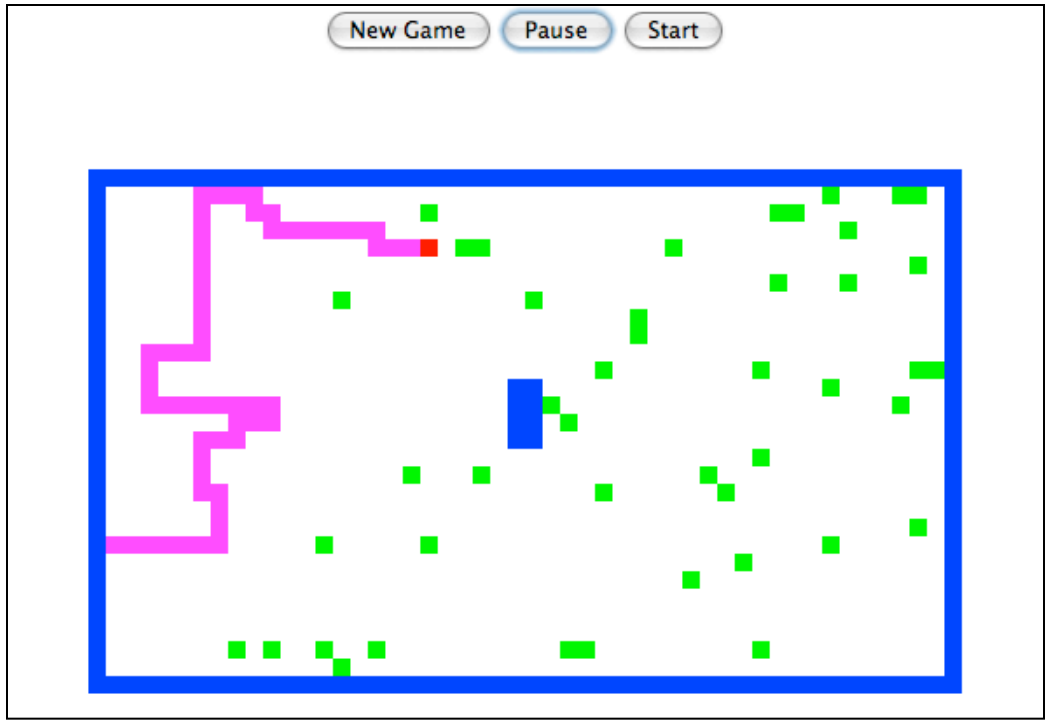
Part 3:
You'll be ready after Thursday

Top-down design

- Getting a clear view of the requirements of the project…
  - breaking it up into component pieces
  - assembling those pieces into a coherent whole…

Top-down design

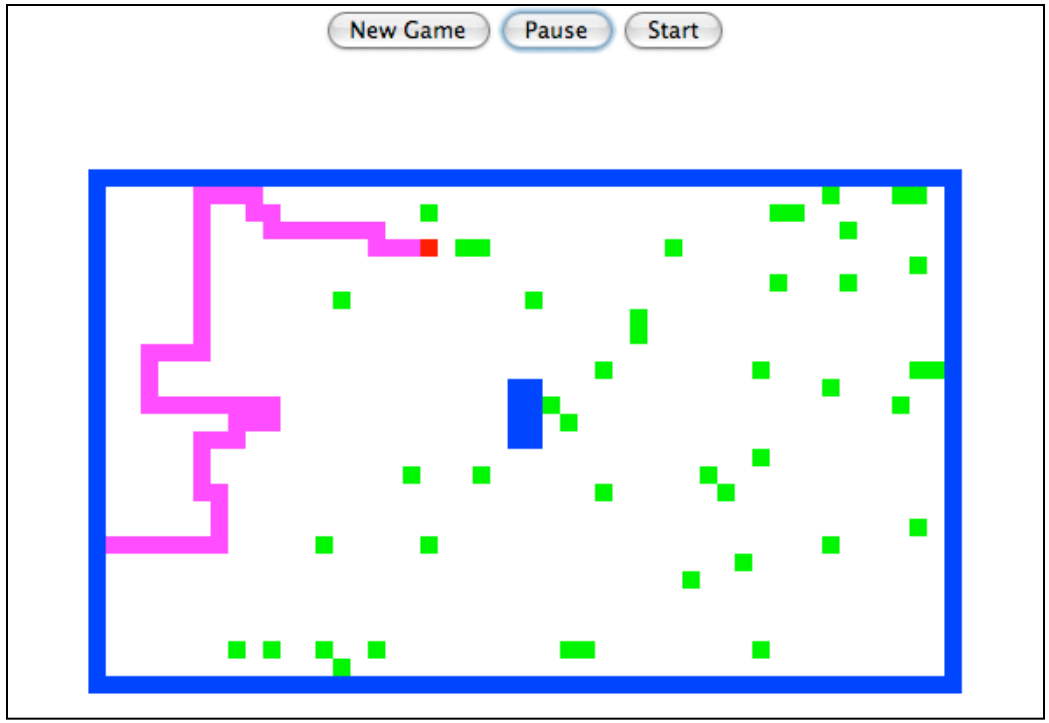
- Getting a clear view of the requirements of the project…
  - breaking it up into component pieces
  - ***reuse or recurse!***
  - assembling those pieces into a coherent whole…



What will help with Spampede?

Top-down design

- Getting a clear view of the requirements of the project…
  - breaking it up into component pieces
  - ***reuse or recurse!***
  - assembling those pieces into a coherent whole…



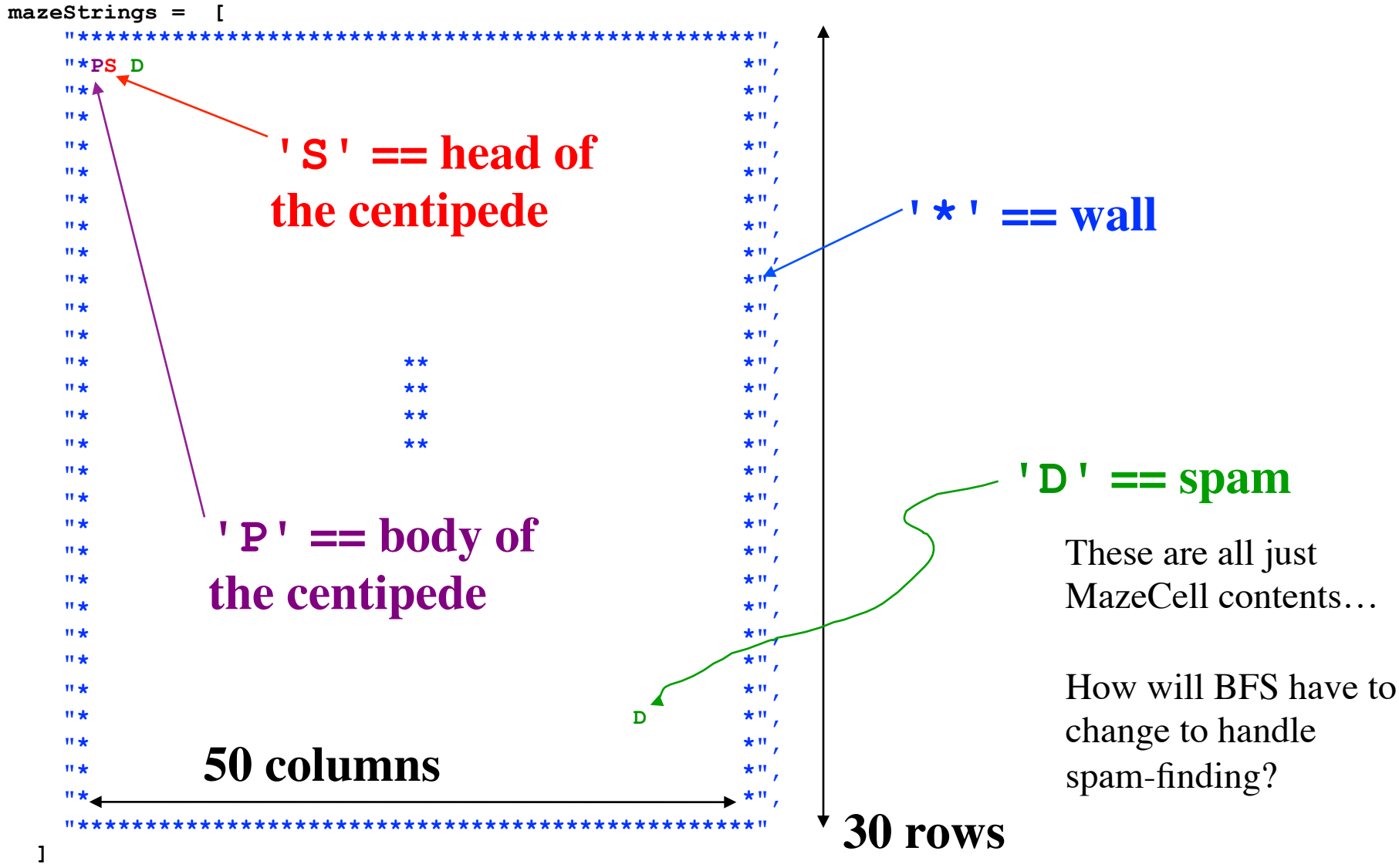
What can we reuse for Spampede?

`Maze`, v. 2.0

More SPAM than ever before… we'll need a new strategy (or class)

mazeStrings

Specifying the maze
within the code itself…



These are all just MazeCell contents…

How will BFS have to change to handle spam-finding?

Avoid "magic numbers" !

even Prof. Benjamin agrees...

data members used to give better names to values:

```
SPAM  = 'D';
START = 'S';
WALL  = '*';
PEDE  = 'P';
```

values for use in functions, etc. Preface with class name, if elsewhere!

These constants make code easier to read and modify

The “AI” Pede!

```
"**************************************************",
"*PSooooooo                                       *"
"*        o                                       *",
"*        o                                       *",
"*        o                                       *",
"*        o                                       *",
"*        D                                       *",
"*                                                *",
"*                                                *",
"*                                                *",
"*                                                *",
"*                                                *",
"*                      **                        *",
"*                      **                        *",
"*              D       **                        *",
"*                      **                        *",
"*                                                *",
"*                                                *",
"*                                                *",
"*                                                *",
"*                                                *",
"*                                                *",
"*                                                *",
"*                                                *",
"*                                                *",
"*                                                *",
"*                                                *",
"*                                                *",
"*                                                *",
"*                                                *",
"*                                                *",
"*                                                *",
"**************************************************"
```

Go east, smartie-pede!

But just *one* step, then check again!

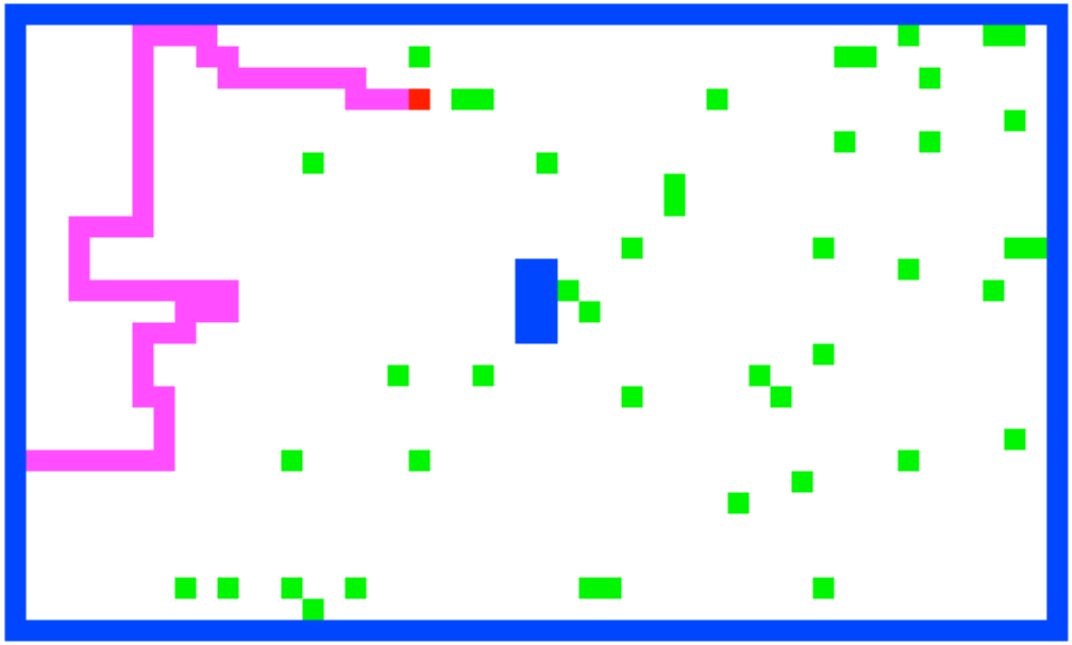
PSooooooo
o
o
o

New maze methods

(1) **`multiBFS`** finding the nearest spam *without changing the maze!*

```
(MazeCell) multiBFS( (MazeCell) start, (char) dest)
```

Why return a **`MazeCell`**?



for testing:

print the maze with the path, then *remove the path* before returning…

New `Maze` methods

(1) **multiBFS** should find the nearest spam *without changing the maze!*

```
multiBFS(start, dest)
```

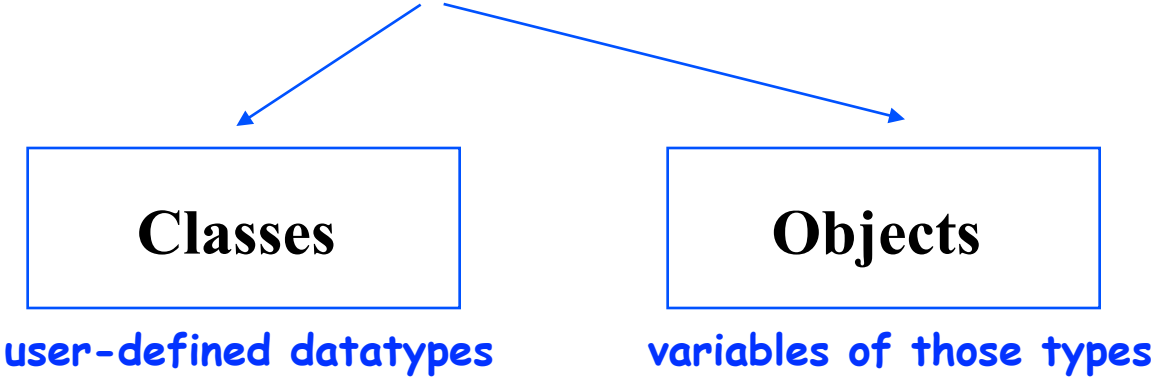
(2) a new constructor (for **mazeStrings**)

```
def __init__(self, filename=None):
    ''' Initialize a new Maze from a File or
        from the string above '''
    if filename is None:
        self.rows = len(mazeStrings)
        self.columns = len(mazeStrings[0])
        self.maze = []
        for r in range(self.rows):
            self.maze.append([])
            for c in range(self.columns):
                self.maze[r].append(MazeCell(r,c,mazeStrings[r][c]))
    else:
        #same as before
```

```
mazeStrings =
  ["************************",
   "*PS      D             *",
   "*                      *",
   "*                      *",
   "*                      *",
   "************************"]
```

Beyond copy-and-paste

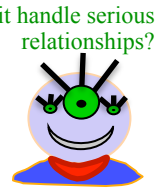
Python is an *object-oriented* programming language:



The primary goal of OOP is to create a good abstraction

- one that models relationships accurately
- without forcing the user to keep track of more than necessary

There are ***two*** relationship types that Python can model:



Object-oriented programming

There are ***two*** basic relationship types that Python models:

Part-of

- a QCell is "part of" a Queue
- a MazeCell is part of a Maze
- a char (contents) is part of a MazeCell

containment or embedding

Is-a

- a String is an Object
- a MazeCell is an Object
- a SpamMaze is a Maze

specializations and extensions

Big Software Strategies

Reuse!

Embedding	Inheritance
using existing classes as data members	extending classes that are already written
Part-of	Is-a

Embedding vs. Inheritance

What does each do?

- Models the "part of" relationship among classes
- Models the "is a" relationship among classes

Why would I want to?

- Code reuse
- Lets new code call old code
- Code reuse
- Lets old code call new code

Code Reuse in Spampede

Embedding

- `Queue` is used in `Maze`
- `MazeCell` is used in `Maze`
- Python's `deque` class is used in `SpamMaze`

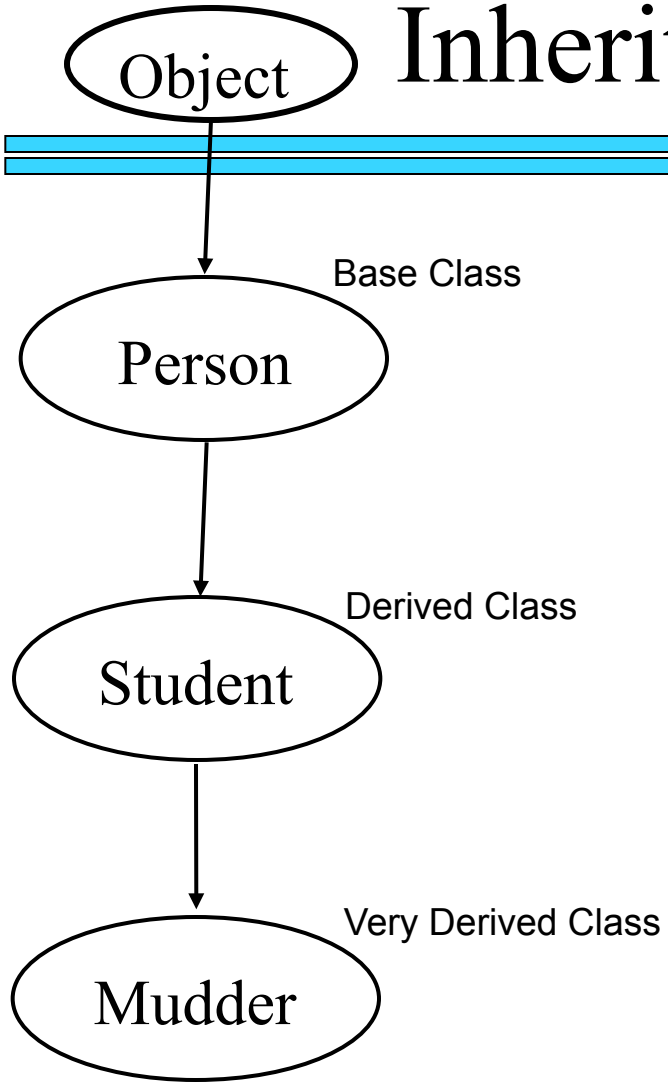
Part-of

Inheritance

- `SpamMaze` inherits from `Maze`

Is-a

Inheritance Hierarchy



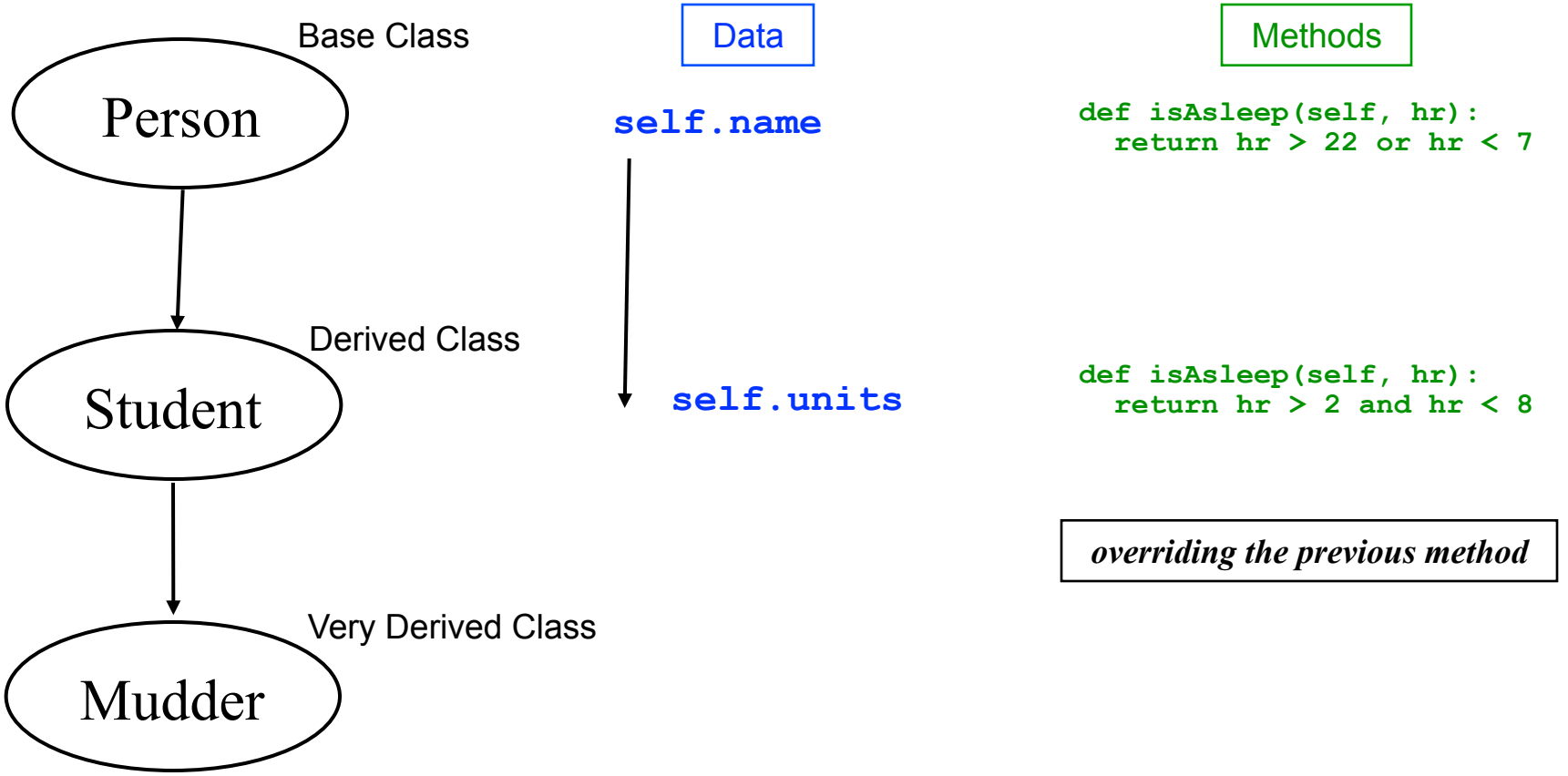
Data

```
self.name
```

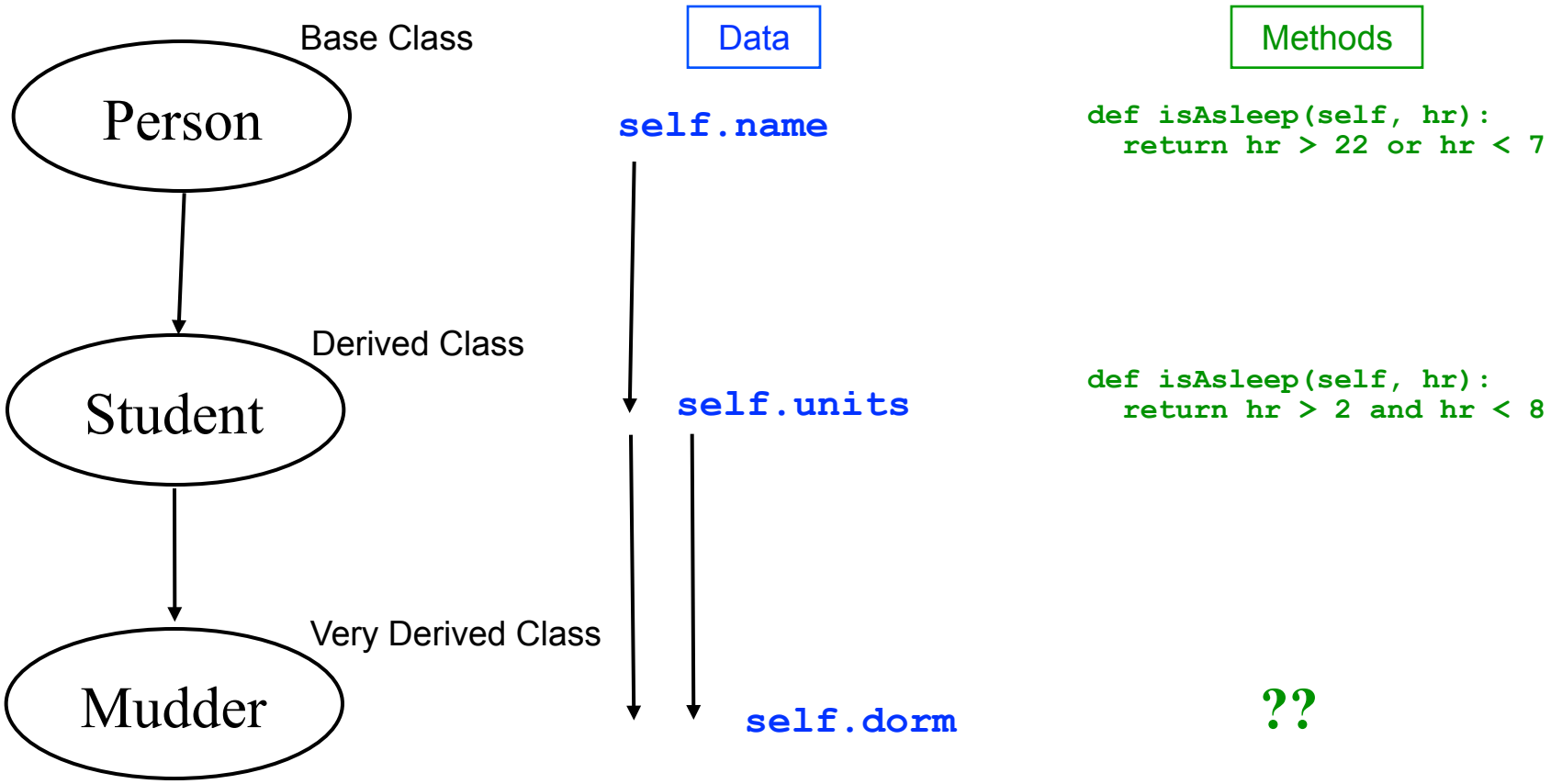
Methods

```
def isAsleep(self,hr):
  return hr > 22 or hr < 7
```

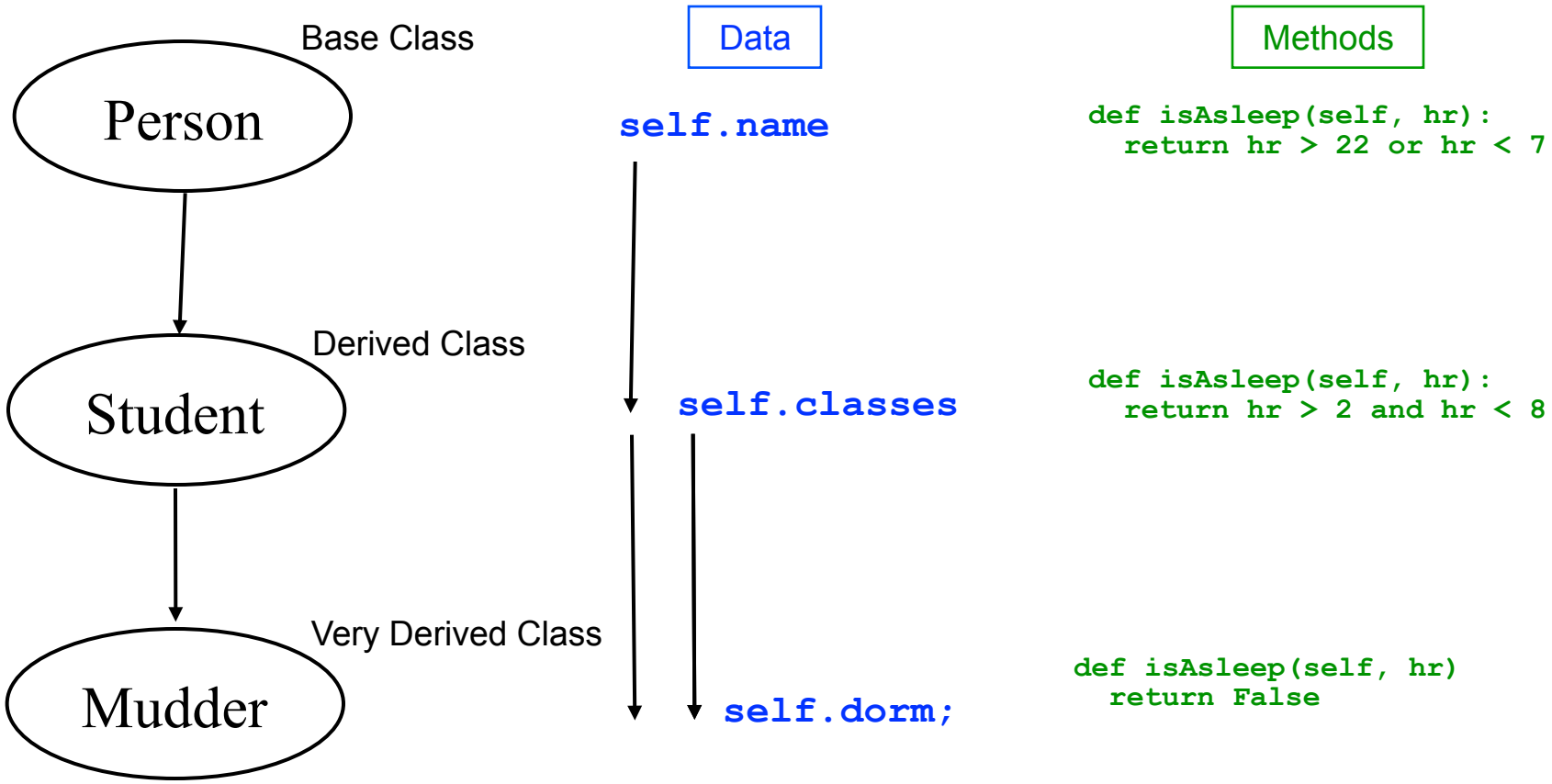
Inheritance Hierarchy



Inheritance Hierarchy



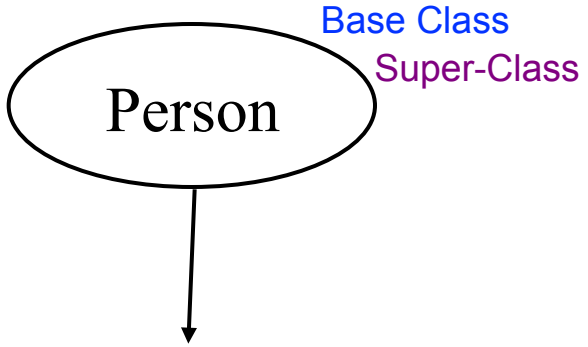
Inheritance Hierarchy



Any further?

```
class Person(object):

    def __init__(self, name):
        self.name = name

    def isAsleep( self, hr ):
        return 22 < hr or 7 > hr

    def __repr__(self):
        return self.name

    def status( self, hr ):
        if self.isAsleep( hr ):
            print "Now offline: " + self
        else:
            print "Now online: " + this
```

Inheritance Details



in main:

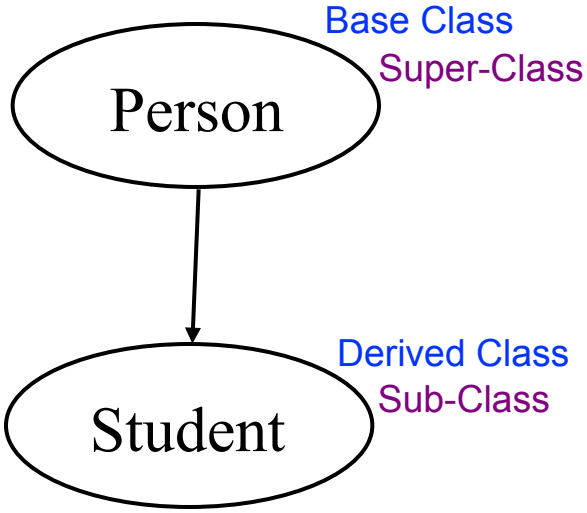
```
P = Person( "Wally" )
```

Consider a picture…

Inheritance Details

```
class Person(object):

    def __init__(self, name):
        self.name = name

    def isAsleep( self, hr ):
        return 22 < hr or 7 > hr

    def __str__(self):
        return self.name

    def status( self, hr ):
        if self.isAsleep( hr ):
            print "Now offline: " + self
        else:
            print "Now online: " + self
```

```
class Student(Person):

    def __init__(self, name, units):
        super(Student, self).__init__(name)
        self.units = units

    def isAsleep( self, hr ):
        return 2 < hr and 8 > hr

    def __str__(self):
        result = super(Student, self).__str__()
        return result + " units: " + str(self.units)
```



in main:

```
S = Student("Wally", 18)
```

Extend the picture…

Name:

```
class Mudder(Student):

  # add a dorm data member

  def isAsleep( self, hr ):
    return False

  def __init__( self, name, units, dorm ):
```

Write the constructor and **`__str__`** methods for this **`Mudder`** class.

```
  def __str__(self):
```

A **`Mudder`** should print out as

Wally units: 42 dorm: Hilton

```
  def main():
    W = Student( "Wally", 16 )


    P = Mudder( "Zach", 42, "Olin" )

    W = P;

    print W
```

What will this code print?

A. It causes an error

B. "Wally units: 16"

C. "Zach units: 42 dorm: Olin"

D. "Zach units: 42"

"Quiz"

Hw 9, Part 2: the **`SpamMaze`** class

… inheriting from the **`Maze`** class

```
class SpamMaze(Maze)
```

data members

`maze`

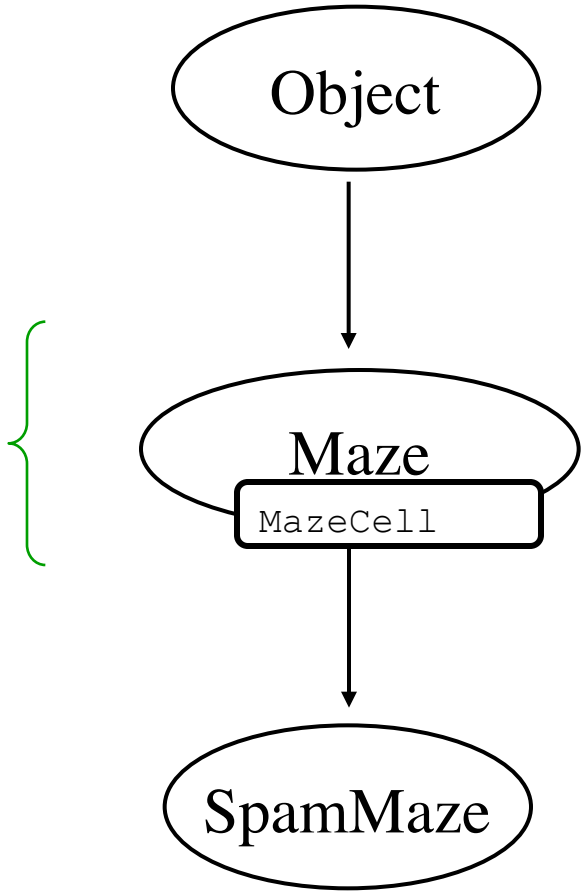
```
MazeCell[][] maze;
```

methods

`multiBFS`

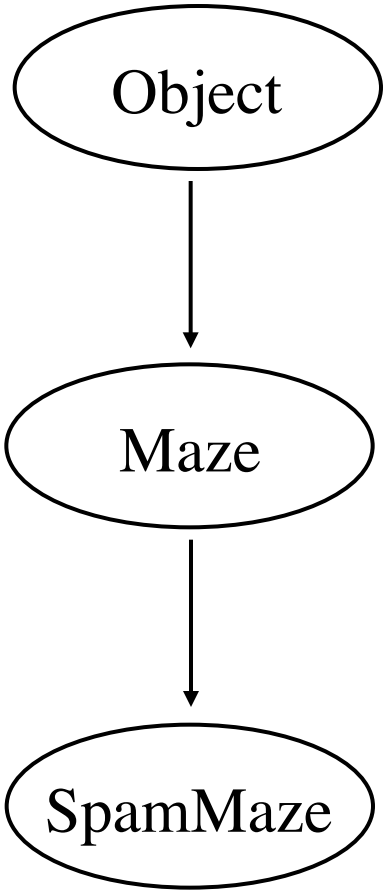
`__str__`

constructors…



Hw 9, Part 2: the **`SpamMaze`** class

… inheriting from the **`Maze`** class



data members

`maze`

`spamCells`

`pedeCells`

methods

`multiBFS`

`__str__`

constructors…

`addSpam`

`removeSpam`

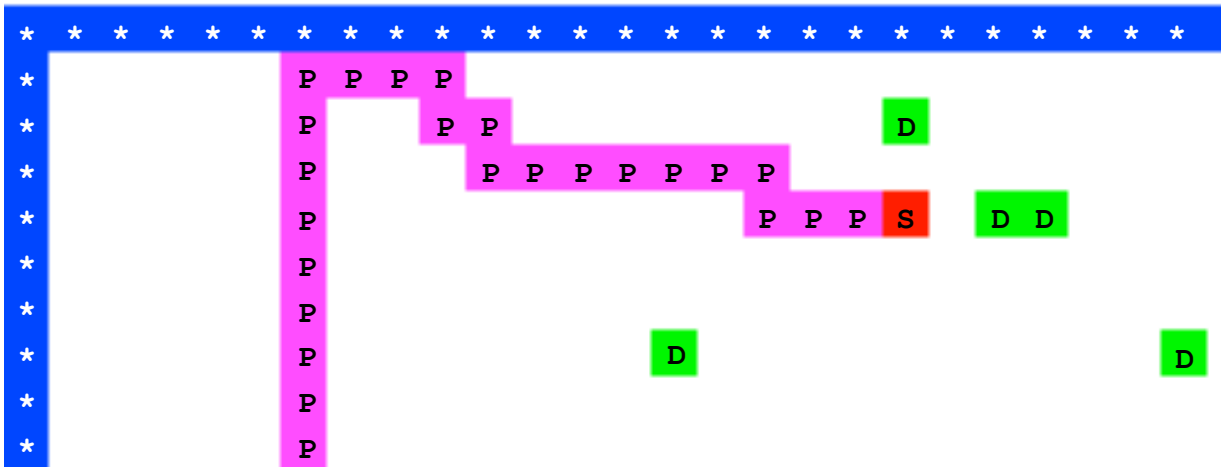
`advancePede`

`reversePede`

What types should spamCells and pedeCells be?

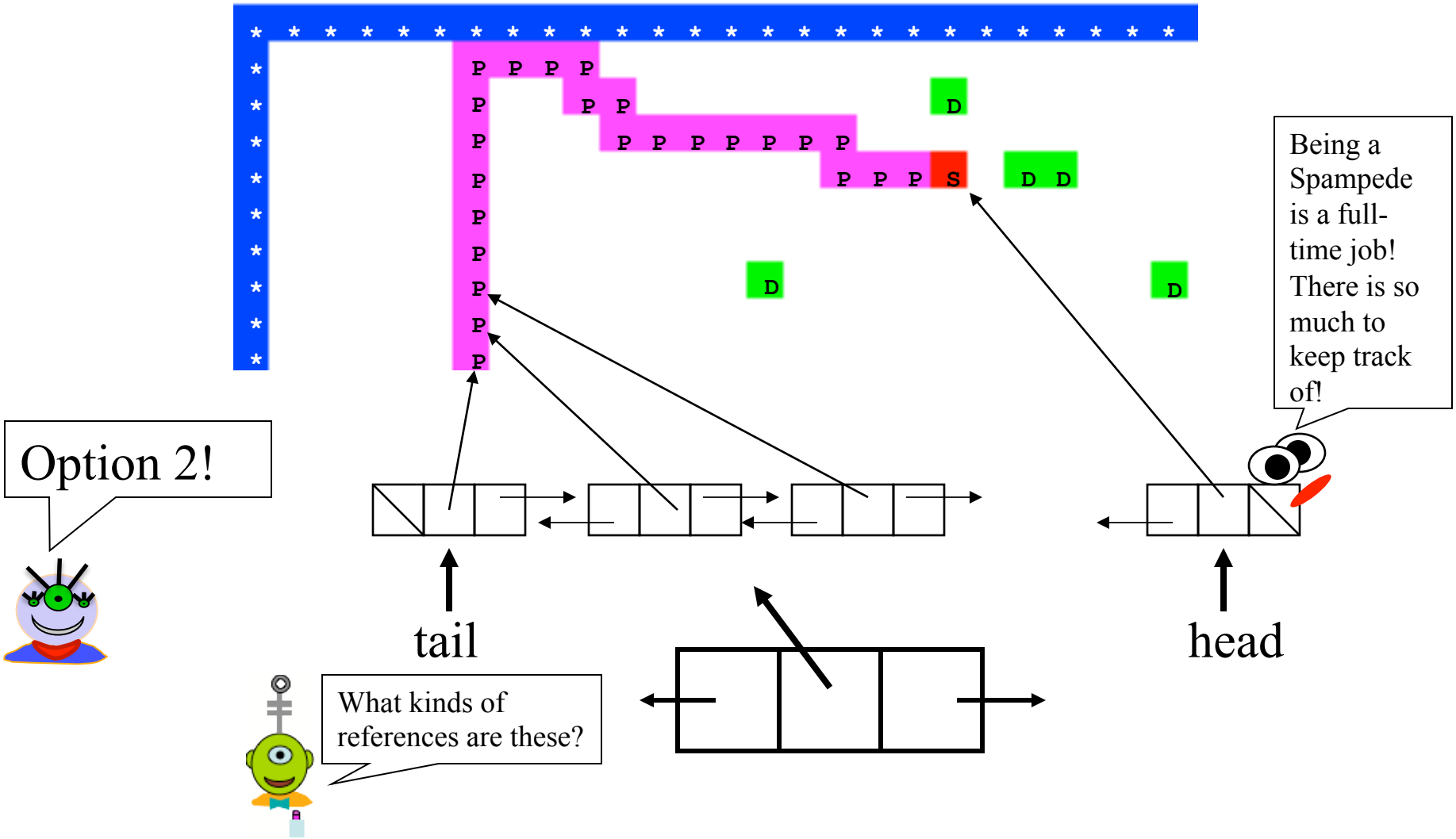
I don't plan to implement a `__str__()` method in SpamMaze!

Visualizing `spamCells` and `pedeCells`

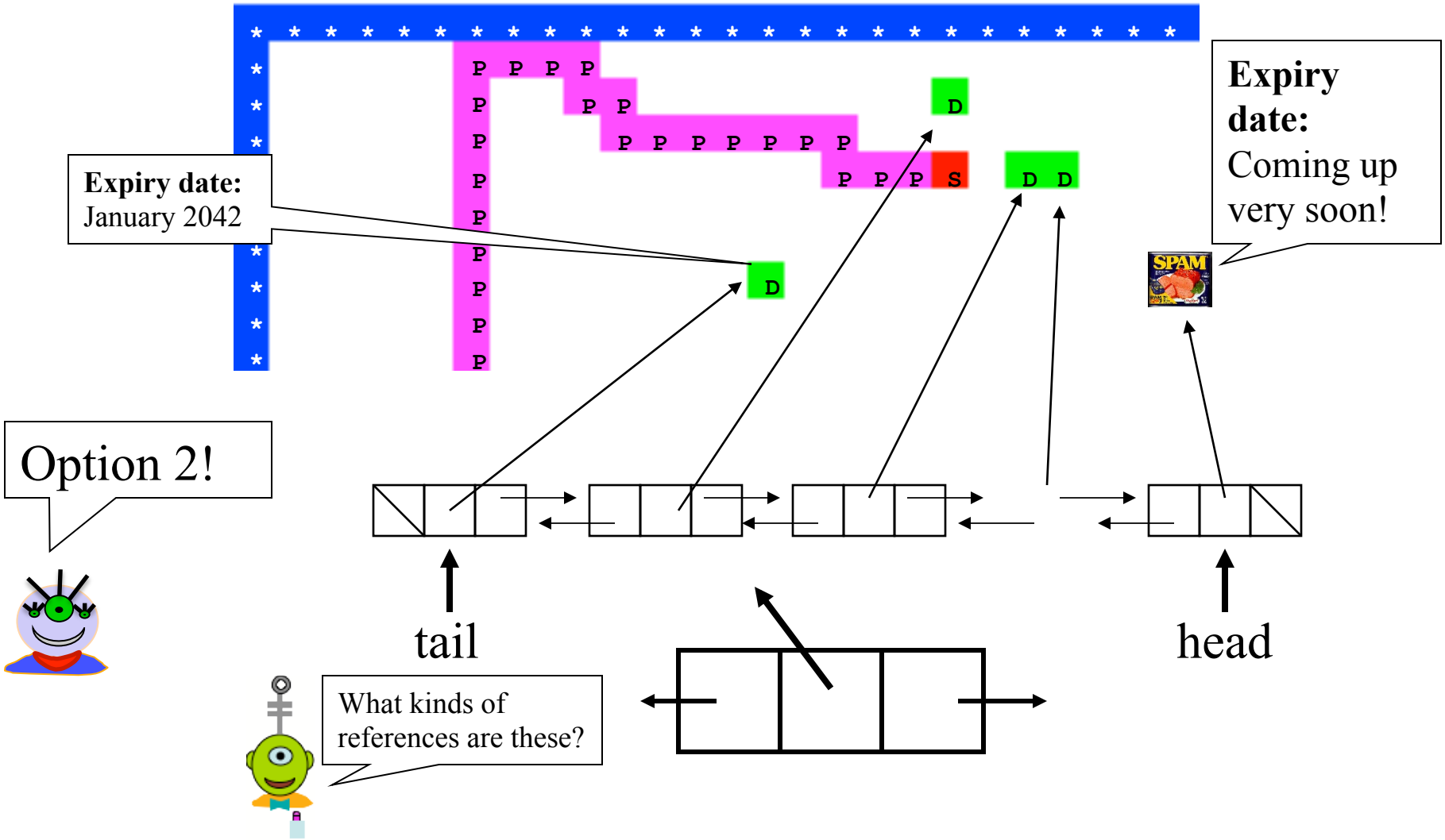


Option 1: I'll just look at the contents of the MazeCells and update the SpamMaze accordingly…

Visualizing `spamCells` and `pedeCells`



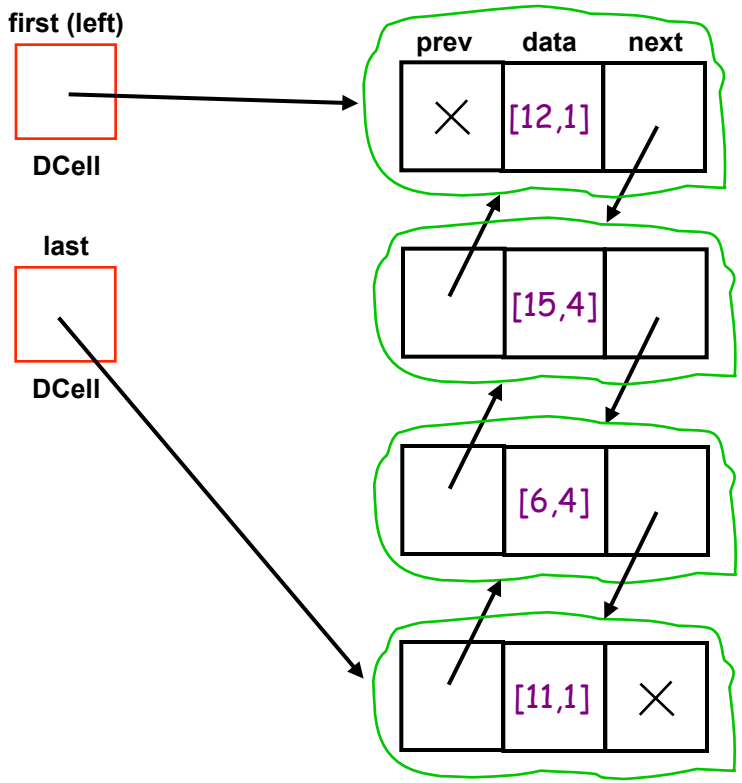
Visualizing `spamCells` and `pedeCells`

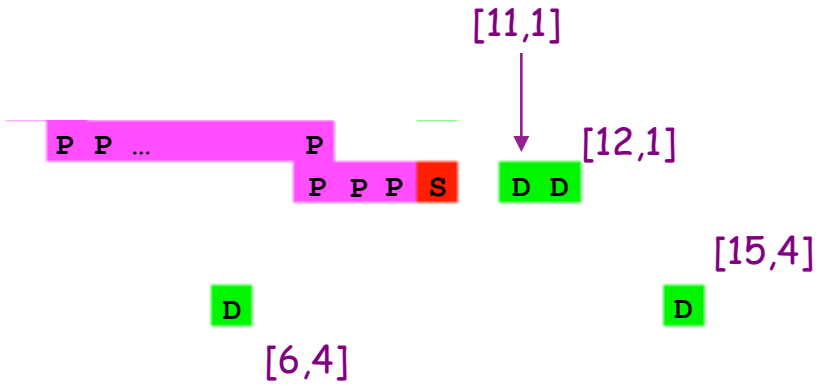


Stacking the deque

A double-ended queue can add and remove elements from the front ***OR*** the back…

```
spamCells = collections.deque()
```





What kind of list will **spamCells** emulate?

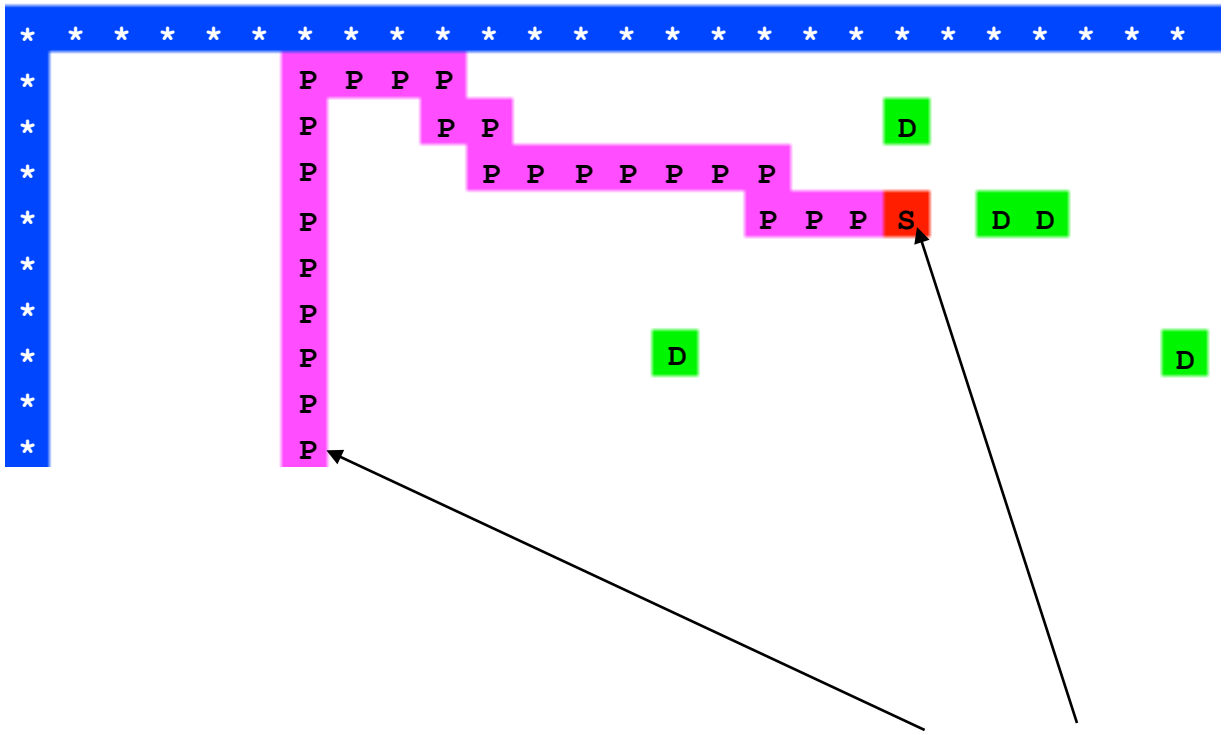
Which will be the next spam to disappear?

Implemented via a series of doubly-linked cells (**DCell**s or deque cells)

So I guess pedeCells are just queuedCells?

Moving the centipede

How might a double-ended queue update the **centipede**, too?




```
pedeCells = collections.deque()
```

Inheritance Reminders

- Models the ***kind-of*** or ***is-a*** relationship
- Pass a class name at the class declaration to indicate a base class
- Only add new data members -- the old ones are already there
- The keyword *super* calls the base class's constructor or methods
- Overriding functions have the same signature as in the base class -- *only override if a new implementation is needed*.

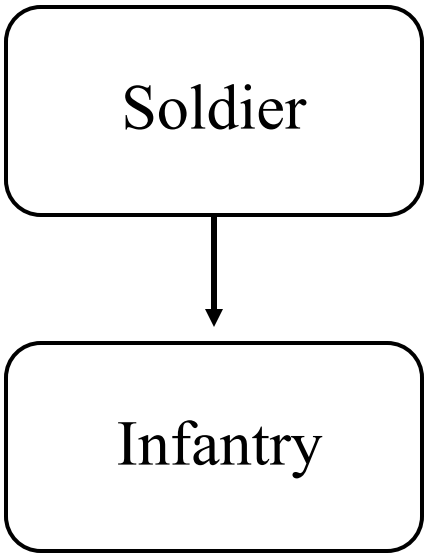
(from Joshua Bloch, *Effective Java*)

Why Inheritance isn't all it's cracked up to be

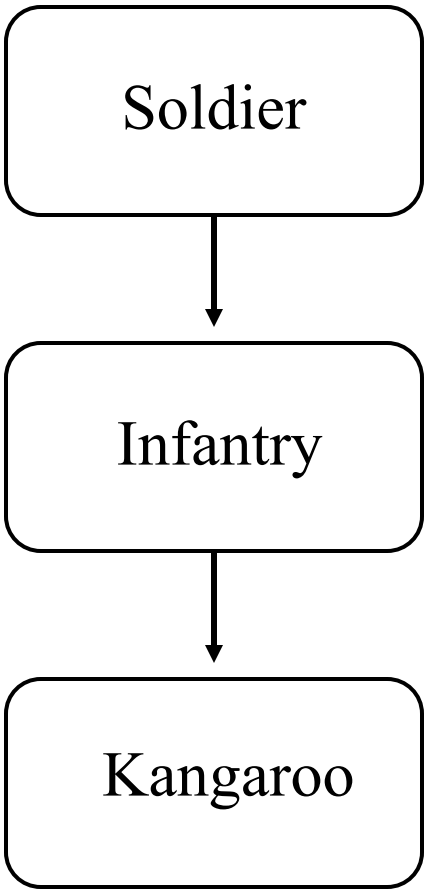
```
class InstrumentedHashSet extends HashSet {
   private int addCount = 0;  // count of elements added
   public InstrumentedHashSet() {}
   public InstrumentedHashSet(Collection c) { super(c); }
   public boolean add(Object o) {
        addcount++;
        return super.add(o);
   }
   public boolean addAll(Collection c) {
        addCount += c.size();
        return super.addAll(c);
   }
   public int getAddCount() { return addCount; }
}
...
InstrumentedHashSet ihs = new InstrumentedHashSet();
ihs.addAll(Arrays.asList(new String[]{"A","B","C"}))
```

Answer?

Other Dangers of Inheritance…



The Dangers of Inheritance…



Object-Oriented Systems Engineering
Object reuse

Careless Code Reuse Causes Killer Kangaroos

Mutant Marsupials Take Up Arms Against Australian Air Force

The reuse of some object-oriented code has caused tactical headaches for Australia's armed forces. As virtual reality simulators assume larger roles in helicopter combat training, programmers have gone to great lengths to increase the realism of their scenarios, including detailed landscapes and - in the case of the Northern Territory's Operation Phoenix- herds of kangaroos (since disturbed animals might well give away a helicopter's position). The head of the Defense Science & Technology Organization's Land Operations/Simulation division reportedly instructed developers to model the local marsupials' movements and reactions to helicopters. Being efficient programmers, they just re-appropriated some code originally used to model infantry detachment reactions under the same stimuli, changed the mapped icon from a soldier to a kangaroo, and increased the figures' speed of movement. Eager to demonstrate their flying skills for some visiting American pilots, the hotshot Aussies "buzzed" the virtual kangaroos in low flight during a simulation. The kangaroos scattered, as predicted, and the visiting Americans nodded appreciatively... then did a double-take as the kangaroos reappeared from behind a hill and launched a barrage of Stinger missiles at the hapless helicopter. Apparently the programmers had forgotten to remove that part of the infantry coding. The lesson? Objects are defined with certain attributes, and any new object defined in terms of an old one inherits all the attributes. The embarrassed programmers had learned to be careful when reusing object-oriented code, and the Yanks left with a newfound respect for Australian wildlife. Simulator supervisors report that pilots from that point onward have strictly avoided kangaroos, just as they were meant to.

From June 15, 1999, Defense Science and Technology Organization Lecture Series, Melbourne, Australia, and staff reports. Item taken from Software Testing and Quality Engineering magazine, Volume 1, Issue 6 (November/December 1999).