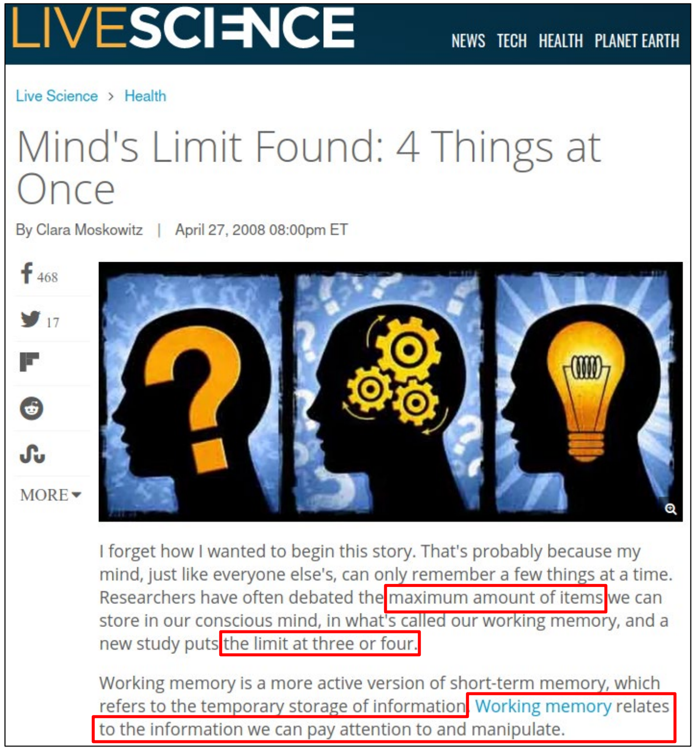
Abstract Data Types (I)
(and Object-Oriented Programming)

Jordi Cortadella and Jordi Petit
Department of Computer Science

How many horses can you distinguish?



LIVESCIENCE NEWS TECH HEALTH PLANET EARTH

Live Science > Health

Mind's Limit Found: 4 Things at Once

By Clara Moskowitz | April 27, 2008 08:00pm ET

468 17 MORE

I forget how I wanted to begin this story. That's probably because my mind, just like everyone else's, can only remember a few things at a time. Researchers have often debated the maximum amount of items we can store in our conscious mind, in what's called our working memory, and a new study puts the limit at three or four.

Working memory is a more active version of short-term memory, which refers to the temporary storage of information. Working memory relates to the information we can pay attention to and manipulate.

Two examples

```
# Main loop of binary search
while left <= right:
    i = (left + right)/2
    if x < A[i]: right = i-1
    elif x > A[i]: left = i+1
    else: return i
```

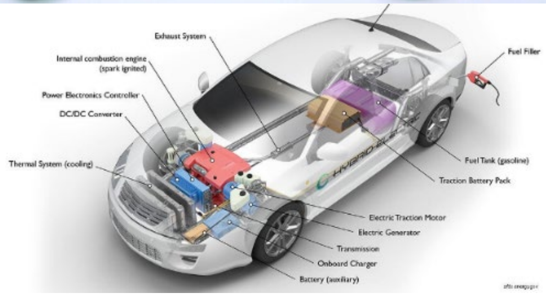
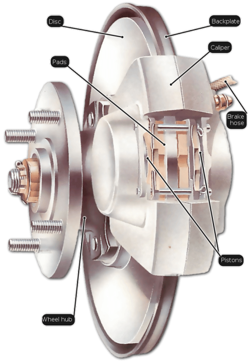
Variables used (5):
`A, x, left, right, i`
(only 3 modified)

```
# Main loop of insertion sort
for i in range(1, len(A)):
    x = A[i]
    j = i
    while j > 0 and A[j-1] > x:
       A[j] = A[j-1]
       j -= 1
    A[j] = x
```

Variables used (4):
`A, x, i, j`

Hiding details: abstractions

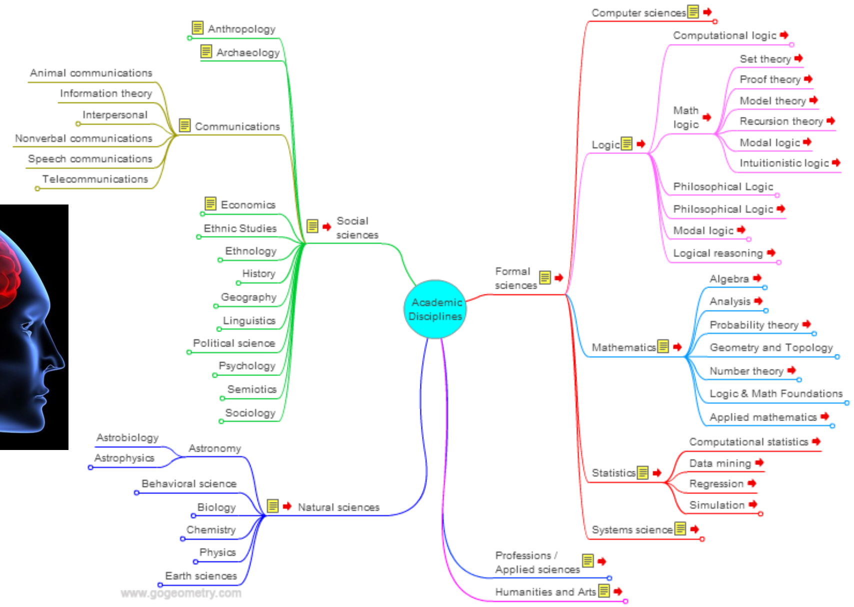


Different types of abstractions



Concept maps are hierarchical: why?

Each level has few items

The computer systems stack

- Application
- Algorithm
- Programming Language
- Operating System
- Instruction Set Architecture
- Microarchitecture
- Register-Transfer Level
- Gate Level
- Circuits
- Devices
- Technology

Image Credit: Christopher Batten, Cornell University

The computer systems stack

- Application
- Algorithm
- Programming Language
- Operating System
- Instruction Set Architecture
- Microarchitecture
- Register-Transfer Level
- Gate Level
- Circuits
- Devices
- Technology

How data flows through system

Boolean logic gates and functions

Combining devices to do useful work

Transistors and wires

Silicon process technology

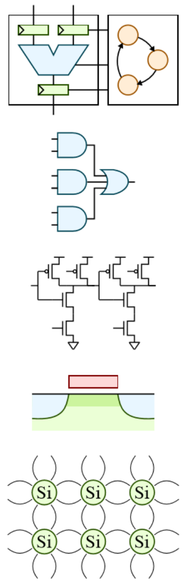


Image Credit: Christopher Batten, Cornell University

The computer systems stack

- Application
- Algorithm
- Programming Language
- Operating System
- Instruction Set Architecture
- Microarchitecture
- Register-Transfer Level
- Gate Level
- Circuits
- Devices
- Technology

Mac OS X, Windows, Linux
Handles low-level hardware management

MIPS32 Instruction Set
Instructions that machine executes

```
blez  $a2, done
move  $a7, $zero
li    $t4, 99
move  $a4, $a1
move  $v1, $zero
li    $a3, 99
lw    $a5, 0($a4)
addiu $a4, $a4, 4
slt   $a6, $a5, $a3
movn  $v0, $v1, $a6
addiu $v1, $v1, 1
movn  $a3, $a5, $a6
```

Image Credit: Christopher Batten, Cornell University

The computer systems stack

- Application
- Algorithm
- Programming Language
- Operating System
- Instruction Set Architecture
- Microarchitecture
- Register-Transfer Level
- Gate Level
- Circuits
- Devices
- Technology

Sort an array of numbers
2,6,3,8,4,5 -> 2,3,4,5,6,8

Insertion sort algorithm

1. Find minimum number in input array
2. Move minimum number into output array
3. Repeat steps 1 and 2 until finished

C implementation of insertion sort

```
void isort( int b[], int a[], int n ) {
  for ( int idx, k = 0; k < n; k++ ) {
    int min = 100;
    for ( int i = 0; i < n; i++ ) {
      if ( a[i] < min ) {
        min = a[i];
        idx = i;
      }
    }
    b[k]   = min;
    a[idx] = 100;
  }
}
```

Image Credit: Christopher Batten, Cornell University

Our challenge

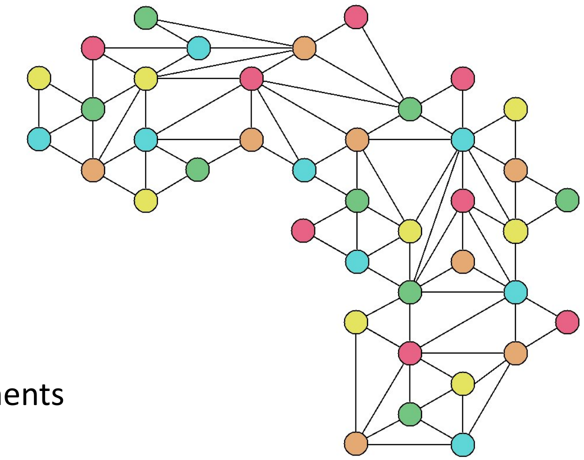
- We need to design large systems and reason about complex algorithms.
- Our working memory can only manipulate 4 things at once.
- We need to interact with computers using programming languages.
- Solution: abstraction
  - Abstract reasoning.
  - Programming languages that support abstraction.
- We already use a certain level of abstraction: functions. But it is not sufficient. We need much more.

Data types

- Programming languages have a set of primitive data types (e.g., int, bool, float, str, …).
- Each data type has a set of associated operations:
  - We can add two integers.
  - We can concatenate two strings.
  - We can divide two floats.
  - But we cannot divide two strings!
- Programmers can add new operations to the primitive data types:
  - gcd(a,b), match(string1, string2), …
- The programming languages provide primitives to group data items and create structured collections of data:
  - C: array, struct.
  - Python: list, tuple, dictionary.

Abstract Data Types (ADTs)

A set of objects and a set of operations to manipulate them



Operations:
- Number of vertices
- Number of edges
- Shortest path
- Connected components

Data type: Graph

Abstract Data Types (ADTs)

A set of objects and a set of operations to manipulate them:

$$\boldsymbol{P(x) = x^3 - 4x^2 + 5}$$

Data type: Polynomial

Operations:
- $P + Q$
- $P \times Q$
- P / Q
- $\gcd(P, Q)$
- $P(x)$
- $\mathrm{degree}(P)$

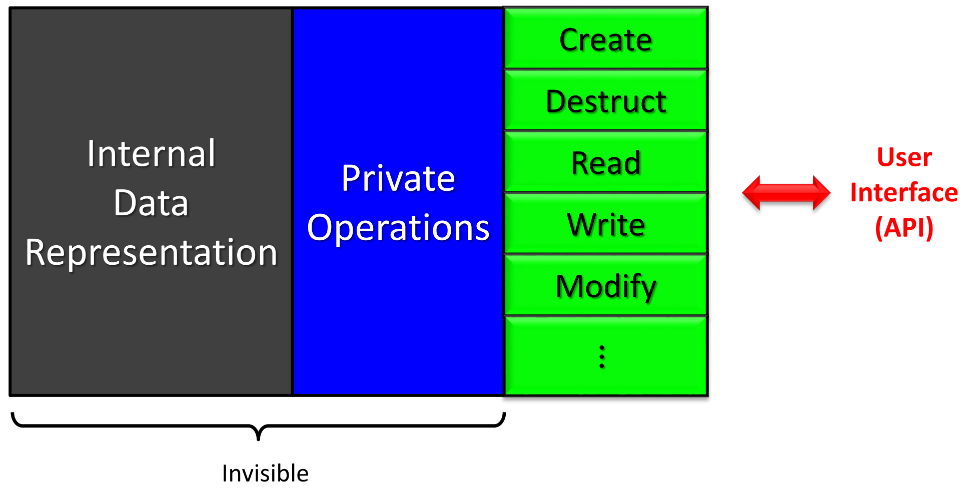
Abstract Data Types (ADTs)

- Separate the notions of specification and implementation:
  - Specification: “what does an operation do?”
  - Implementation: “how is it done?”
- Benefits:
  - Simplicity: code is easier to understand
  - Encapsulation: details are hidden
  - Modularity: an ADT can be changed without modifying the programs that use it
  - Reuse: it can be used by other programs

Abstract Data Types (ADTs)

- An ADT has two parts:
  - Public or external: abstract view of the data and operations (methods) that the user can use.
  - Private or internal: the actual implementation of the data structures and operations.
- Operations:
  - Creation/Destruction
  - Access
  - Modification

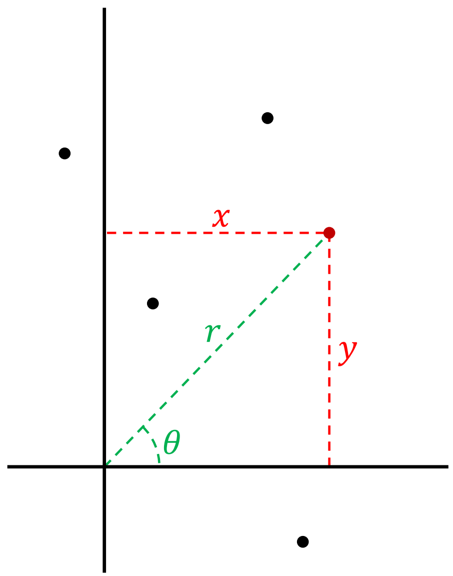
Abstract Data Types (ADTs)



API: Application Programming Interface

Example: a Point

- A point can be represented by two coordinates (x,y).
- Several operations can be envisioned:
  - Get the x and y coordinates.
  - Calculate distance between two points.
  - Calculate polar coordinates.
  - Move the point by $(\Delta x, \Delta y)$.



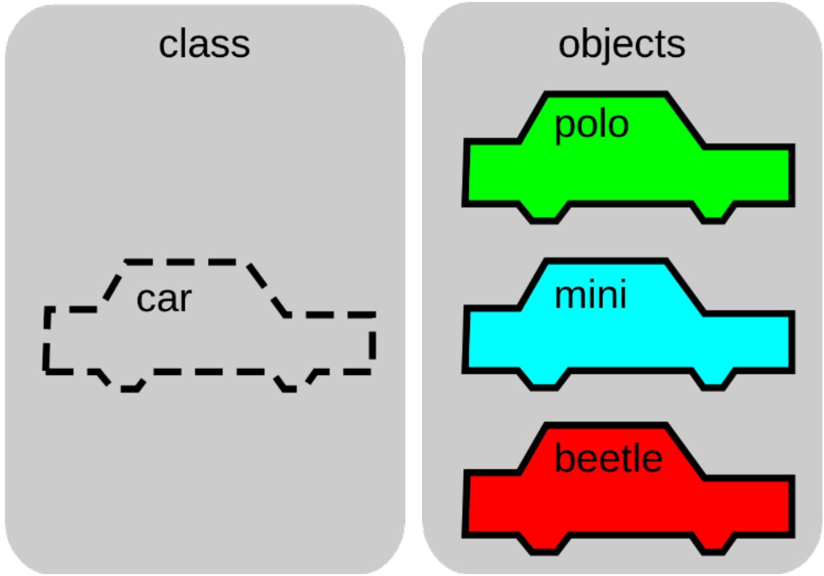
Example: a Point

```
# Things that we can do with points

p1 = Point(5.0, -3.2) # Create a point (a variable)
p2 = Point(2.8, 0)    # Create another point

# We now calculate the distance between p1 and p2
dist12 = p1.distance(p2)

# Distance to the origin
r = p1.distance()

# Create another point by adding coordinates
p3 = p1 + p2

# We get the coordinates of the new point
x = p3.x()  # x = 7.8
y = p3.y()  # y = -3.2
```

ADTs and Object-Oriented Programming

- OOP is a programming paradigm: a program is a set of objects that interact with each other.
- An object has:
  - fields (or attributes) that contain data
  - functions (or methods) that contain code
- Objects (variables) are instances of classes (types). A class is a template for all objects of a certain type.
- In OOP, a class is the natural way of implementing an ADT.

Classes and Objects



Let us design the new type for Point

```python
class Point:
    """A class to represent and operate with two-dimensional points"""

    # Declaration of attributes (recommended for type checking)
    _x: float # x coordinate
    _y: float # y coordinate

    def __init__(self, x: float = 0, y: float = 0):
        """Constructor with x and y coordinates"""
        self._x, self._y = x, y

    def x(self) -> float:
        """Returns the x coordinate"""
        return self._x

    def y(self) -> float:
        """Returns the y coordinate"""
        return self._y

    def distance(self, p: Optional['Point']) -> float:
        """Returns the distance to point p
           (or the distance to the origin if p is None)"""
        dx, dy = self.x(), self.y()
        if p is not None:
            dx -= p.x()
            dy -= p.y()
        return math.sqrt(dx*dx + dy*dy)
```


Let us design the new type for Point

⋮

```python
def angle(self) -> float:
    """Returns the angle of the polar coordinate"""
    if self.x() == 0 and self.y() == 0:
        return 0
    return math.atan2(self.y()/self.x())

def __add__(self, p: 'Point') -> 'Point':
    """Returns a new point by adding the coordinates of two points.
       This is a method associated to the + operator"""
    return Point(self.x() + p.x(), self.y() + p.y())

def __eq__(self, p: 'Point') -> bool:
    """Checks whether two points are equal.
       This is a method associated to the == operator"""
    return self.x() == p.x() and self.y() == p.y()
```

How the class methods are invoked

```
p1 = Point(5.0, -3.2)  # __init__(p1, 5.0, -3.2)
p2 = Point(2.8)        # __init__(p2, 2.8, 0)

dist12 = p1.distance(p2)  # distance(p1, p2)
                     self

# Distance to the origin
r = p1.distance()        # distance(p1, None)

# Create another point by adding coordinates
p3 = p1 + p2  # Equivalent to p1.__add__(p2)

# We get the coordinates of the new point
x = p3.x()  # x = 7.8
y = p3.y()  # y = -3.2
```


How the class methods are invoked

```
p1 = Point(5.0, -3.2)
p2 = Point(2.8)

dist12 = p1.distance(p2)

r = p1.distance()

p3 = p1 + p2

x = p3.x()
y = p3.y()
```

```
class Point:
    def __init__(self, x: float = 0, y: float = 0):
    def x(self) -> float:
    def y(self) -> float:
    def distance(self, p: Optional['Point']) -> 
float:
    def angle(self) -> float:
    def __add__(self, p: 'Point') -> 'Point':
    def __eq__(self, p: 'Point') -> bool:
```


Python naming conventions

Type	Examples
Function	`distance, dot_product, multiply_by_two`
Variable	`x, num, num_elements`
Class	`Point, CityGraph, ParkingLot`
Public method	`distance, get_angle, shortest_path`
Private method	`_gcd, _check, _calculate_mean`
Magic method	`__init__, __add__, __eq__, __str__`
Constant	`GRAVITY, MIN_DISTANCE, MAX_NUM_PEOPLE`
Module	`point.py, city_graph.py, parking_lot.py`
Package	`geometry, citygraph`

Recommendation:

- use short names for modules and packages
- no underscores for package names

Comment: `PascalCase`, `camelCase` and `snake_case`

Magic methods

- They are invoked internally to implement certain actions.
- They are not supposed to be invoked by the user.
- Some examples:
  - Arithmetic: __add__, __mul__, __div__, __truediv__, __neg__, …
  - Relational: __eq__, __ne__, __gt__, __ge__, …
  - Representation: __str__, __repr__, …
  - Class initialization: __init__, __new__, __del__
  - and others

Class Point in C++

```
// The declaration of the class Point
class Point {

public:
  // Constructor
  Point(double x, double y);

  // Constructor for (0,0)
  Point();

  // Gets the x coordinate
  double x() const;

  // Gets the y coordinate
  double y() const;

  // Returns the distance to point p
  double distance(const Point& p) const;

  // Returns the distance to the origin
  double distance() const;

  // Returns the angle of the polar coordinate
  double angle() const;

  // Creates a new point by adding the coordinates of two points
  Point operator + (const Point& p) const;

private:
  double _x, _y;  // Coordinates of the point
};
```


Implementation of the class Point

```
// The constructor: different implementations
Point::Point(double x, double y) {
  _x = x; _y = y;
}

// or also
Point::Point(double x, double y) : _x(x), _y(y) {}
```

They are equivalent, but only one of them should be chosen.
We can have different constructors with different *signatures*.

```
// The other constructor
Point::Point() : x(0), y(0) {}
```


Implementation of the class Point

```
double Point::x() const {
  return _x;
}

double Point::y() const {
  return _y;
}

double Point::distance(const Point& p) const {
  double dx = x() – p.x(); // Better getX() than x
  double dy = y() – p.y();
  return sqrt(dx*dx + dy*dy);
}

double Point::distance() const {
  return sqrt(x()*x() + y()*y());
}
```

Note: compilers are smart. Small functions are expanded inline.

Implementation of the class Point

```
double Point::angle() const {
  if (x() == 0 and y() == 0) return 0;
  return atan(y()/x());
}

Point Point::operator + (const Point& p) const {
  return Point(x() + p.x(), y() + p.y());
}
```

Conclusions

- The human brain has limitations: 4 things at once.
- Modularity and abstraction are for designing large maintainable systems.