

# Data Abstraction

## Captain Abstraction Meets Private Data

Friday, December 7, 2007

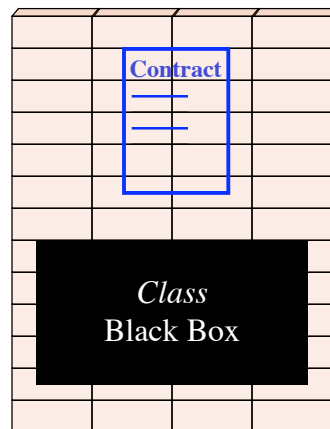


**CS111 Computer Programming**

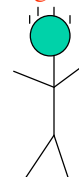
Department of Computer Science  
Wellesley College

## Revisiting A Big Idea: Abstraction

User / Client



Implementer /  
Designer



**ABSTRACTION BARRIER**

Data Abstraction 24-2

## Two Kinds of Abstraction in CS111

### 1. Procedural Abstraction:

Methods abstract over computational behavior

### 2. Data Abstraction:

Classes abstract over state and behavior of objects

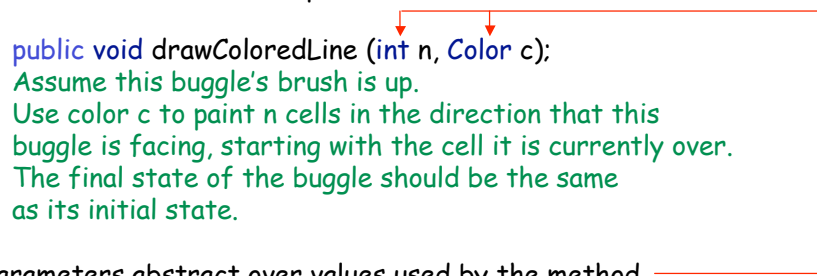
Today, we'll briefly review procedural abstraction and then focus on data abstraction.

Data Abstraction 24-3

## Procedural Abstraction

Methods capture procedural patterns, abstracting over behaviors.

The contract of a method specifies its behavior.



```
public void drawColoredLine (int n, Color c);
```

Assume this bugle's brush is up.  
Use color *c* to paint *n* cells in the direction that this bugle is facing, starting with the cell it is currently over.  
The final state of the bugle should be the same as its initial state.

Parameters abstract over values used by the method.

Data Abstraction 24-4

## Many Ways to Skin a Cat

The same method contract can often be implemented in many different ways.

Implementers are free to experiment with different implementations as long as the contract is satisfied. Often, they seek implementations that are "efficient" - ones that make effective use of time, space, and other resources.

Clients don't care what goes on "under the hood" as long as the contract is satisfied. \*

\*As long as the method is reasonably efficient

```
// Recursive solution
public void drawColoredLine (int n, Color c) {
    if (n > 0) { // n <= 0 is base case: do nothing
        paintCell(c); // paint cell under bugle;
        forward(); // move to paint rest of cells
        drawColoredLine(n-1, c); // paint rest of
cells
        backward(); // return to initial position
    }
}
```

```
// Iterative solution
public void drawColoredLine (int n, Color c) {
    Color savedColor = getColor();
    setColor(c);
    brushDown();
    for (int i = 1; i <= n; i++)
        {forward;} // paint n cells with brush
    // restore original state:
    backward(n);
    brushUp();
    setColor(savedColor);
}
```

Data Abstraction 24-5

## Data Abstraction

Classes capture object notions, abstracting over related stateful values and their associated behaviors.

The contract of a class specifies:

- The state of objects in the class (**instance variables**)
- How to construct objects in the class (**constructor methods**)
- The behavior of objects in the class (**instance methods**)
- Other related state and behavior associated with the class (**class variables** and **class methods**).

Data Abstraction 24-6

## Contract Example: java.awt.Point

// An instance of the Point class has **mutable** (changeable)  
// x and y integer coordinates.

// Instance variables

public int x; // x coordinate of point  
public int y; // y coordinate of point

Because they're public, anyone  
can **access** and **change** the x and y  
instance variables

// Constructor methods

public Point (); // construct the point (0,0);  
public Point (int x, int y); // construct the point (x,y);  
public Point (Point p); // construct a new point (p.x, p.y);

// Instance methods

public void move (int x, int y); // move this point to (x,y)  
public void translate (int dx, int dy); // move this point to (x+dx, y+dy)  
public void toString(); // return a String representation of this point  
// ... many other Point instance methods ...

Data Abstraction 24-7

## public final Variables Are Immutable

We often want instance and class variables that can be publicly accessed but are **immutable** - i.e., they cannot be changed.

The **final** keyword is used for this purpose. Once a final variable has been initialized, it cannot be changed.

// Instance variables of the Location class, whose  
// instances are immutable 2D integer points

public final int x;  
public final int y;

Can write loc.x but not loc.x = 3;

// Class variables of the Color class

public static final Color red;  
public static final Color blue;  
public static final Color magenta;

Don't want anyone to be able  
to redefine these colors!

Data Abstraction 24-8

## public vs. private Variables

Most classes that we have used this semester do **not** have **public** instance variables; they have **private** instance variables instead.

E.g.: Buggle, Color, Direction, Graphics, IntList,  
Picture, StringList, Sprite, Turtle

Why?

- A contract shouldn't be cluttered with details clients don't need to know.
- Private variables can be accessed/changed only via methods, giving the implementer fine-grained control over their use. E.g., using `setPosition()` to change a Buggle's position allows redrawing the Buggle in the grid.
- Keeping variables hidden gives implementers the freedom to use clever implementations and change implementations.

*Moral: Make your instance variables private!*

Data Abstraction 24-9

## Tinkering Under the Hood

To get some hands-on experience with private data, we'll spend the rest of today tinkering with the implementations of several classes that we know and love:

- Pic
- Buggle
- IntList
- Direction



Data Abstraction 24-10

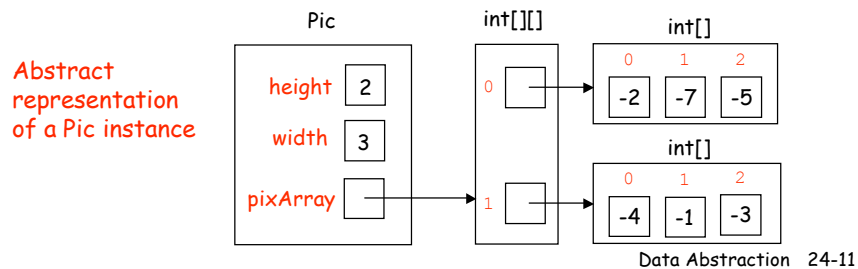
## Abstract Representation

When instance variables are private, we often infer the **abstract representation** of an instance from the method contracts.

Consider the `Pic` class from PS9, whose methods include:

```
public int getHeight();
public int getWidth();
public int getPixel(int row, int col);
public void setPixel(int row, int col, int value);
public int[][] getPixArray();
```

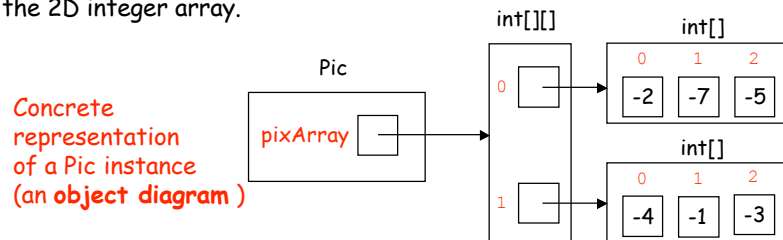
Based on the `Pic` contract, we can imagine that `Pic` instances look like:



## Concrete Pic Representation

The **concrete representation** specified by the actual instance variables can differ from the abstract representation inferred from the contract.

E.g., a `Pic` instance need not have actual `height` and `width` instance variables, because this information is already encoded in the 2D integer array.

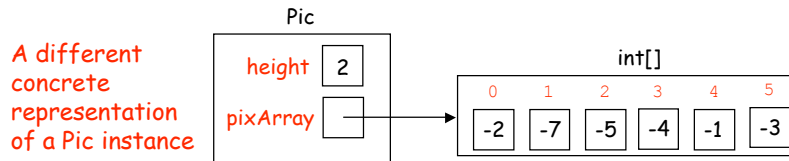


```
public int getHeight() { return pixArray.length; }
public int getWidth() { if (getHeight() == 0) return 0;
                       else return pixArray[0].length; }
```

Data Abstraction 24-12

## A Different Concrete Pic Representation

Sometimes a concrete representation can differ dramatically from the abstract one. E.g., a Pic instance might consist of a `height` variable and a 1D array of pixels.



```
public int getHeight() { return height; }
public int getWidth() { if (height == 0) return 0;
                        else return pixArray.length/height; }
public int getPixel (int row, int col) {
    return pixArray(row*getWidth() + col);
}
```

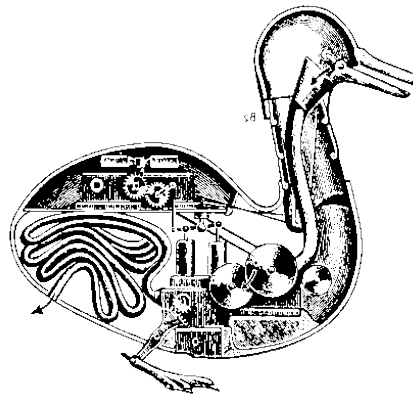
How would you write `getPixArray()` using this representation?

Data Abstraction 24-13

## Like A Duck

The Pic examples illustrate a key aspect of data abstraction: clients care about the abstract representations and behavior of objects, not their concrete structure.

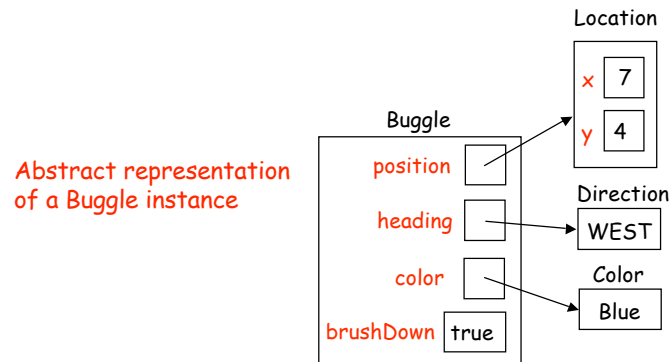
If an object walks like a duck, quacks like a duck, and behaves like a duck every other respect, data abstraction says that it's a duck --- even if it's a mechanical duck or a wolf in a duck suit!



Data Abstraction 24-14

## Abstract Buggle Representation

All semester long we've drawn abstract representations of Buggles:



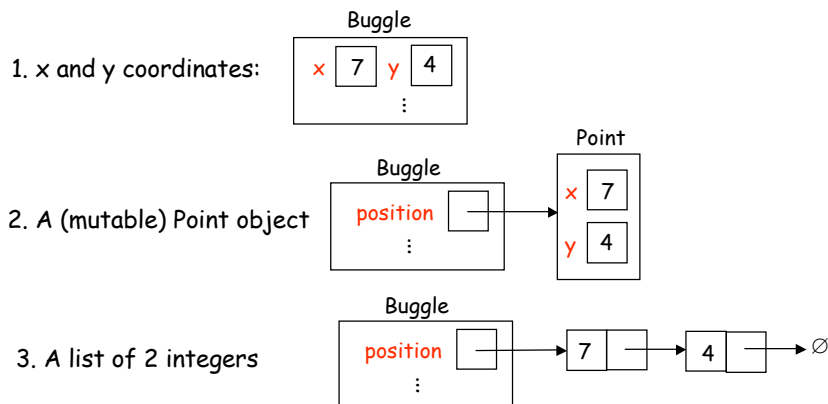
(We choose to represent Direction and Color instances with names that indicate the value.)

But the concrete representation could be rather different.  
Let's explore some possibilities!

Data Abstraction 24-15

## Buggle Positions

A Buggle position need not be represented as a Location object. It could be:



... and many other representations (e.g. a 2-element integer array in PS10).

How would the implementation change to use these representations?

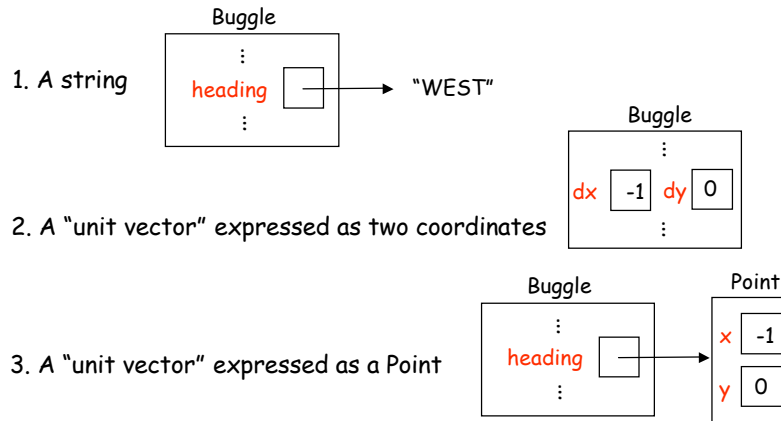
What are their advantages and disadvantages?

Data Abstraction 24-16



## Bugle Directions

A Bugle heading need not be represented as a Direction object. It could be:



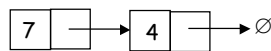
... and many other representations (e.g. a 2-element integer array in PS10).

How would the implementation change to use these representations?

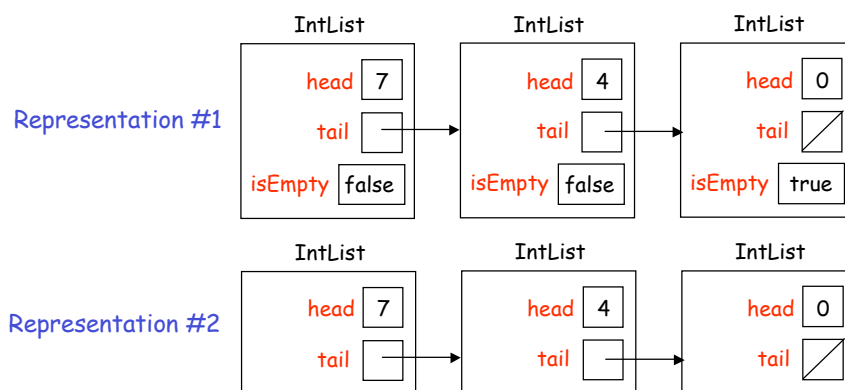
What are their advantages and disadvantages?

Data Abstraction 24-17

## IntList Representations



How would you implement the core `IntList` operations using the following concrete representations?



Data Abstraction 24-18

## On To CS230!

Although the examples we've shown illustrate the key ideas of data abstraction, they're not particularly compelling.

The Data Structures course (CS230) is chock full of compelling examples of standard data abstractions that every practicing programmer needs to know:

- vectors (extensible arrays)
- stacks
- queues
- priority queues
- sets
- bags
- tables
- trees
- graphs

Data Abstraction 24-19