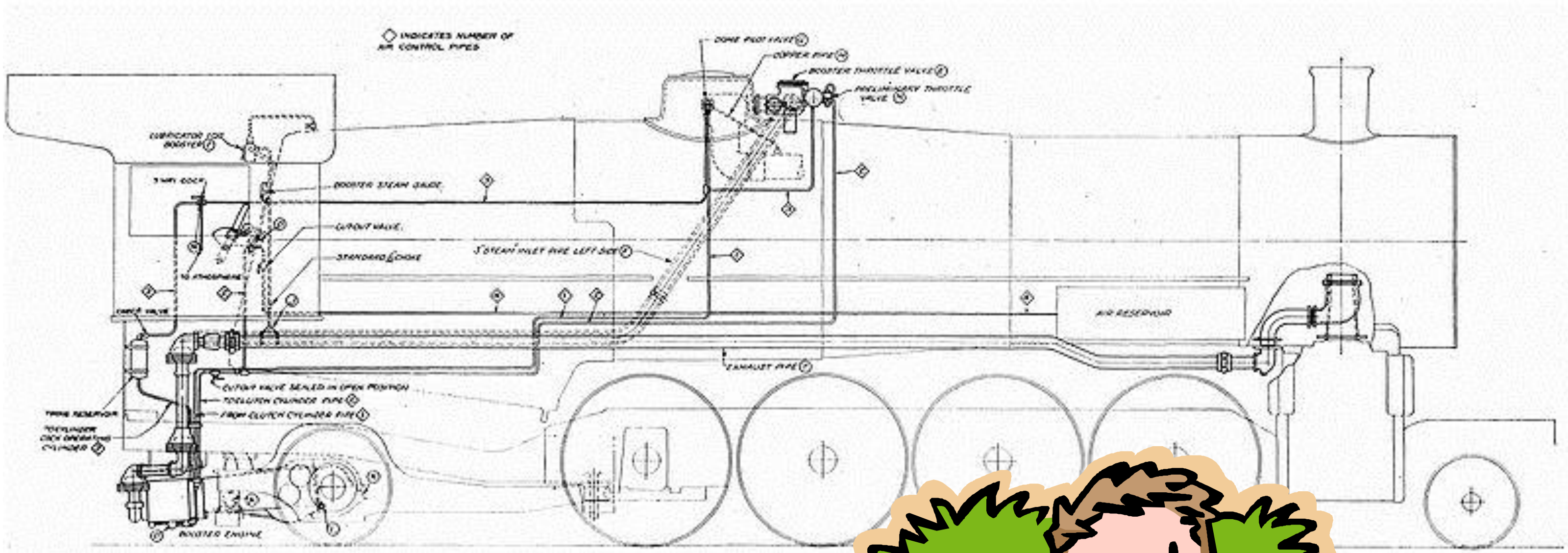
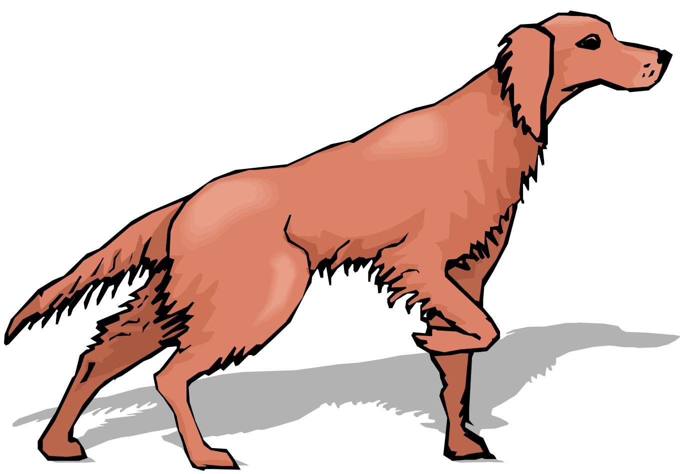
Designing data types





Overview

- Object Oriented Programming (OOP)
- Data encapsulation
  - Important consideration when designing a class
  - Access modifiers
    - Getters and setters
  - Immutability, preventing change to a variable
- Checking for equality
  - Not always as simple as you might think!
    - floating-point variables
    - reference variables

Object Oriented Programming

- Procedural programming [verb-oriented]
  - Tell the computer to do this
  - Tell the computer to do that
- OOP philosophy
  - Software simulation of real world
  - We know (approximately) how the real world works
  - Design software to model the real world
- Objected oriented programming (OOP) [noun-oriented]
  - Programming paradigm based on data types
  - Identify: objects that are part of problem domain or solution
    - Objects are distinguishable from each other (references)
  - State: objects know things (attributes)
  - Behavior: objects do things (methods)

Alan Kay

- Alan Kay [Xerox PARC 1970s]
  - Invented Smalltalk programming language
  - Conceived portable computer
  - Ideas led to: laptop, modern GUI, OOP

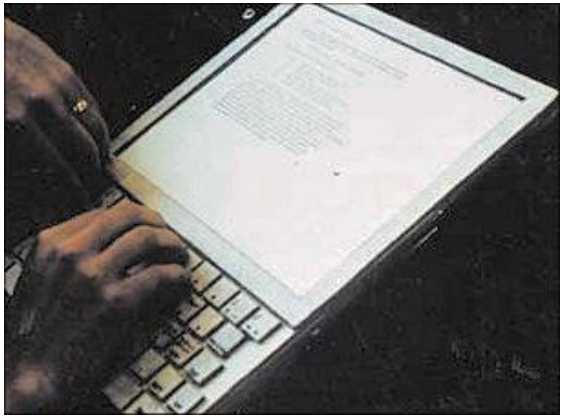
Alan Kay
2003 Turing Award

"The computer revolution hasn't started yet."

"The best way to predict the future is to invent it."

"If you don't fail at least 90 per cent of the time, you're not aiming high enough."

— Alan Kay



Dynabook: A Personal Computer for Children of All Ages, 1968.

Data encapsulation

- Data type (aka class)
  - "Set of values and operations on those values"
  - e.g. `int`, `String`, `Room`, `Fraction`, `Circle`, `Balloon`
- Encapsulated data type
  - Hide internal representation of data type.
- Separate implementation from design specification
  - Class provides data representation & code for operations
  - Client uses data type as black box
  - API specifies contract between client and class
- Bottom line:
  - You don't need to know how a data type is implemented in order to use it

Intuition

Client

Client needs to know how to use API

API
- volume
- change channel
- adjust picture
- decode NTSC signal

Implementation
- cathode ray tube
- electron gun
- Sony Wega 36XBR250
- 241 pounds

Implementation needs to know what API to implement

Implementation and client need to agree on API ahead of time.

Intuition

Client

API
- volume
- change channel
- adjust picture
- decode NTSC signal

Implementation
- gas plasma monitor
- Samsung FPT-6374
- wall mountable
- 4 inches deep

Client needs to know how to use API

Implementation needs to know what API to implement

Can substitute better implementation without changing the client.


WEEKLY WORLD NEWS

THE COMPUTER CRASH OF THE MILLENNIUM!

JANUARY 1, 2000

THE DAY THE EARTH WILL STAND STILL!

ALL BANKS WILL FAIL!

FOOD SUPPLIES WILL BE DEPLETED!

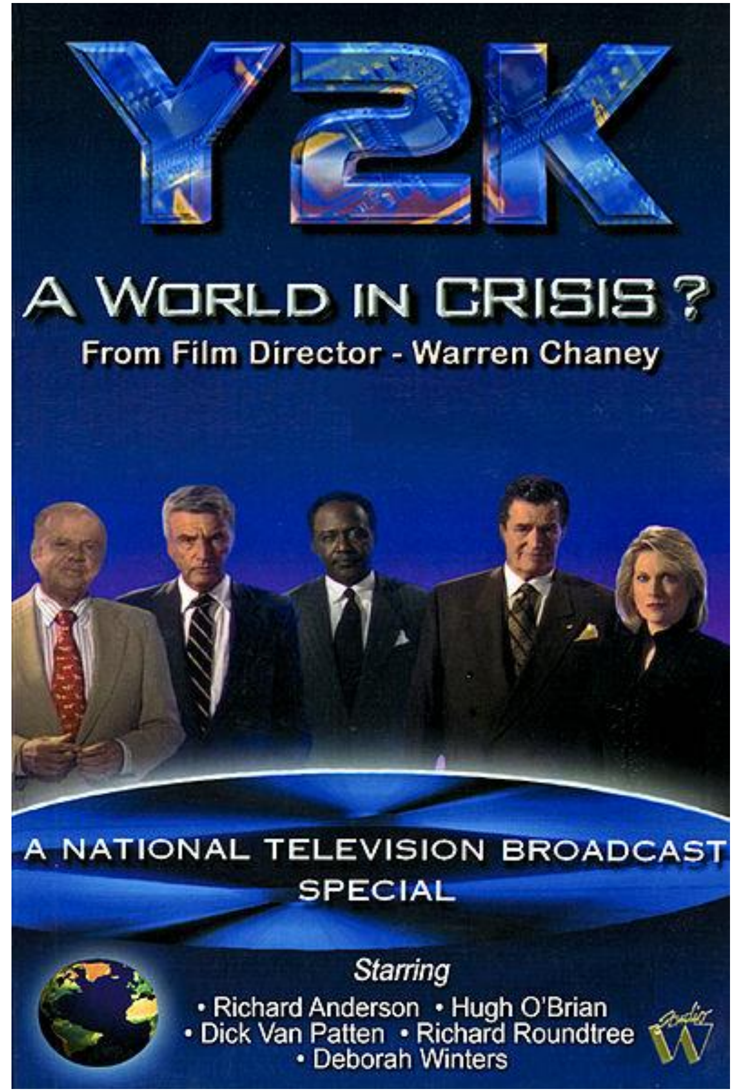
ELECTRICITY WILL BE CUT OFF!

THE STOCK MARKET WILL CRASH!

VEHICLES USING COMPUTER CHIPS WILL STOP DEAD!

TELEPHONES WILL CEASE TO FUNCTION!

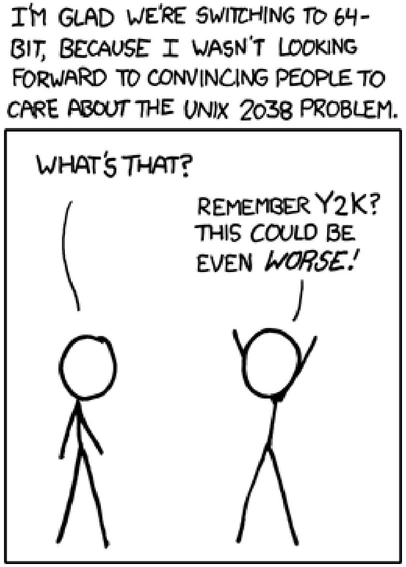
DOMINO EFFECT WILL CAUSE A WORLDWIDE DEPRESSION!



"When someone says to you, Y2K is not a problem. Inform them that it already is... one trillion dollars - and rising." --Richard Anderson

Time Bombs

- Internal representation changes
  - [Y2K] Two digit years: Jan 1, 2000
  - [Y2038] 32-bit seconds since 1970: Jan 19, 2038



http://xkcd.com/607/

- Lesson
  - By exposing data representation to client, may need to sift through millions of lines of code to update

Data encapsulation example

- Person class
  - Originally stored first & last name in one instance variable
  - Now we want them separated → change instance vars

```
class Person:

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

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

Original version, combined names

```
class Person:

    def __init__(self, name, score):
        self.first = name.split()[0]
        self.last  = name.split()[1]
        self.score = score

    def toString(self):
        result = self.first
        result += " "
        result += self.last
        return result
...
```

New version, names separated.

Non-encapsulated example

- What if we advertise attributes?
  - Client program might use them directly instead of methods

```
class Person:

    def __init__(self, name, score):
        self.first = name.split()[0]
        self.last  = name.split()[1]
        self.score = score

    def toString(self):
        result = self.first
        result += " "
        result += self.last
        return result
```

Non-encapsulated version, instance variables are public.

```
from Person import Person
...
p = Person("Bob Dole")

print(p.name + " " + p.score)

...
```

Client program.
Changing instance variables causes compile error. Client should have been using `toString()` but used instance variable because they were publically available. Code like this might be in many client programs!

Getters and setters

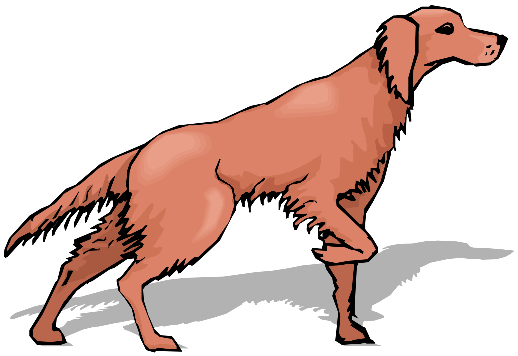
- Encapsulation does have a price
  - If clients need access to attribute, must create:
    - getter methods - "get" value of an attribute(accessor)
    - setter methods - "set" value of an attribute(mutator)

```
def getPosX(self) :
    return self.posX
```

Getter method.
Also know as an ***accessor*** method.

```
def setPosX(self, x):
    self.posX = x
```

Setter method.
Also know as a ***mutator*** method.



Immutability

- Immutable data type
  - Object's value cannot change once constructed

Class	Description	Immutable?
bool	Boolean value	✓
int	integer (arbitrary magnitude)	✓
float	floating-point number	✓
list	mutable sequence of objects	
tuple	immutable sequence of objects	✓
str	character string	✓
set	unordered set of distinct objects	
frozenset	immutable form of set class	✓
dict	associative mapping (aka dictionary)	

Immutability: Pros and Cons

- Immutable data type
  - Object's value cannot change once constructed
- Advantages
  - Avoid aliasing bugs
  - Makes program easier to debug
  - Limits scope of code that can change values
  - Pass objects around without worrying about modification
- Disadvantage
  - New object must be created for every value

String immutability: consequences

```
s = "Hello world!"
print("before : " + s)
s.upper()
print("after  : " + s)
```

```
before : Hello world!
after  : Hello world!
```

Since `String` is immutable, this method call *cannot* change the variable s!

```
s = "Hello world!"
print("before : " + s)
s = s.upper ()
print("after  : " + s)
```

```
before : Hello world!
after  : HELLO WORLD!
```

Equality: integer primitives

- Boolean operator ==
  - See if two variables are exactly equal
    - i.e. they have identical bit patterns
- Boolean operator !=
  - See if two variables are NOT equal
    - i.e. they have different bit patterns

```
a = 5

if a == 5:
    print("yep it's 5!")

while a != 0:
  a -= 1
```

This is a safe comparison since we are using an integer type.

Equality: floating-point primitives

- Floating-point primitives
  - i.e. `float`
  - Only an approximation of the number
  - Use == and != at your own peril

```
a = 0.1 + 0.1 + 0.1
b = 0.1 + 0.1
c = 0.0

if a == 0.3:
    print("a is 0.3!")

if b == 0.2:
    print("b is 0.2!")

if c == 0.0:
    print("c is 0.0!")
```

```
b is 0.2!
c is 0.0!
```

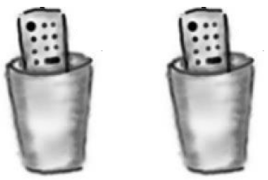
Equality: floating-point primitives

- Floating-point primitives
  - i.e. `double` and `float`
  - Only an approximation of the number
  - Use == and != at your own peril

```
a = 0.1 + 0.1 + 0.1
b = 0.1 + 0.1
c = 0.0
EPSILON = 1e-9

if abs(a - 0.3) < EPSILON:
    print("a is 0.3!")

if abs(b - 0.2) < EPSILON:
    print("b is 0.2!")

if abs(c) < EPSILON:
    print("c is 0.0!")
```

```
a is 0.3!
b is 0.2!
c is 0.0!
```

Equality: reference variables



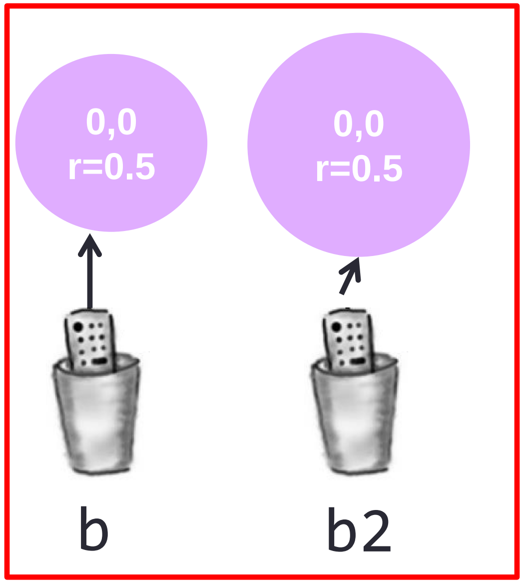
- Boolean operator ==, !=
  - Compares bit values of remote control
    - Not the values stored in object's instance variables
  - Usually not what you want

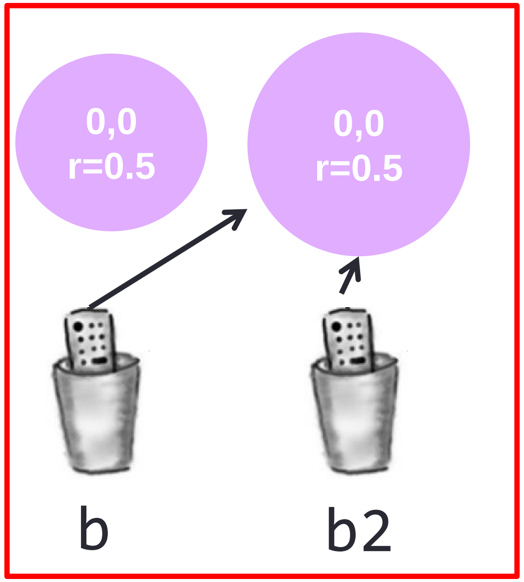
```
b  = Circle.Circle(0.0, 0.0, 0.5)
b2 = Circle.Circle(0.0, 0.0, 0.5)

if b == b2:
   print("circles equal!")

b = b2
if b == b2:
   print("circles now equal!")
```

Equality: reference variables

```
b  = Circle.Circle(0.0, 0.0, 0.5)
b2 = Circle.Circle(0.0, 0.0, 0.5)



if b == b2:
   print("circles equal!")




b = b2
if b == b2:
   print("circles now equal!")
```

```
balls now equal
```





Object equality

- Implement `equals()` method
  - Up to class designer exactly how it works
  - Client needs to call `equals()`, not == or !=

```
class Circle:

…

    def equals(self, other):
        EPSILON = 1e-9
        return (abs(self.posX   - other.posX)   < EPSILON) and
               (abs(self.posY   - other.posY)   < EPSILON) and
               (abs(self.radius - other.radius) < EPSILON)

…
```

Summary

- Object oriented programming
- Data encapsulation
  - Important consideration when designing a class
  - Access modifiers decide who can see what
    - Getters and setters
  - Immutability, preventing change to a variable
- Equality
  - Usually avoid == or != with floating-point types
  - Usually avoid == or != with reference types
    - Implement or use the `equals()` method