Object-Oriented Programming with Objective-C

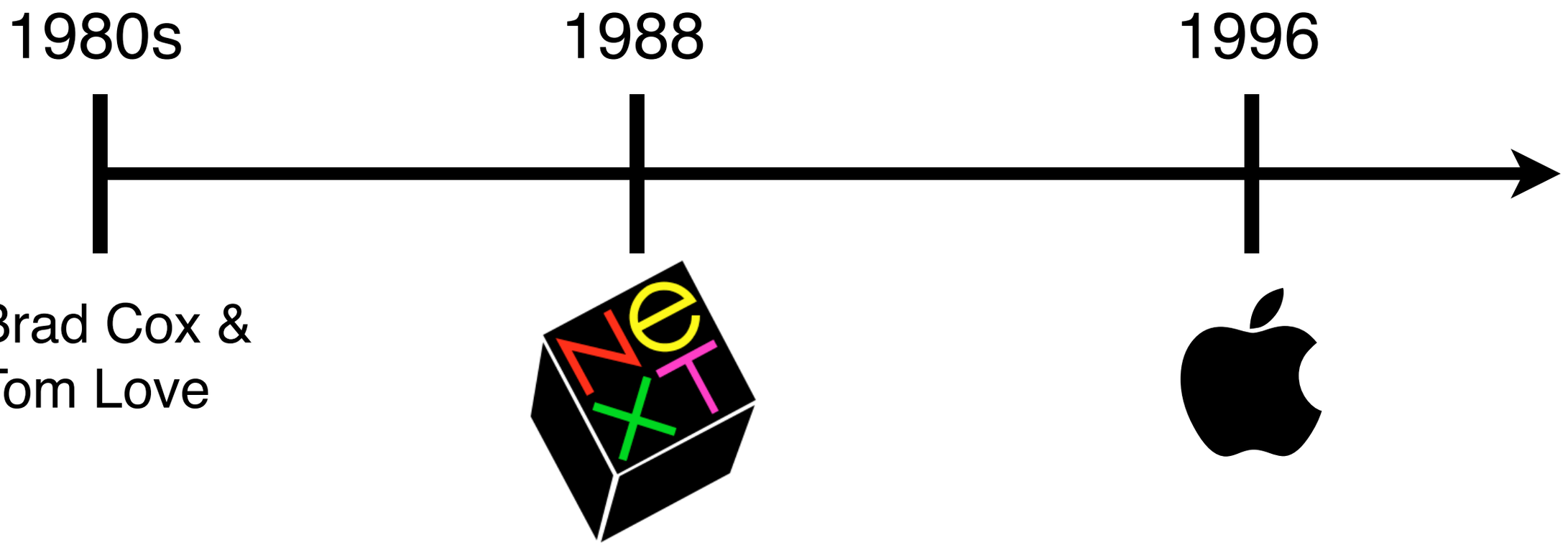
Lecture 2

Objective-C

A Little History

- Originally designed in the 1980s as a fusion of Smalltalk and C
- Popularized by NeXTSTEP in 1988 (hence the ubiquitous “NS”)
- Apple bought NeXT in 1996, acquiring their operating system along with Steve Jobs

A Little History



Features

- Simple extension to C
- Low level efficiency and interoperability
- Dynamic runtime, like Smalltalk
- Optional static typing

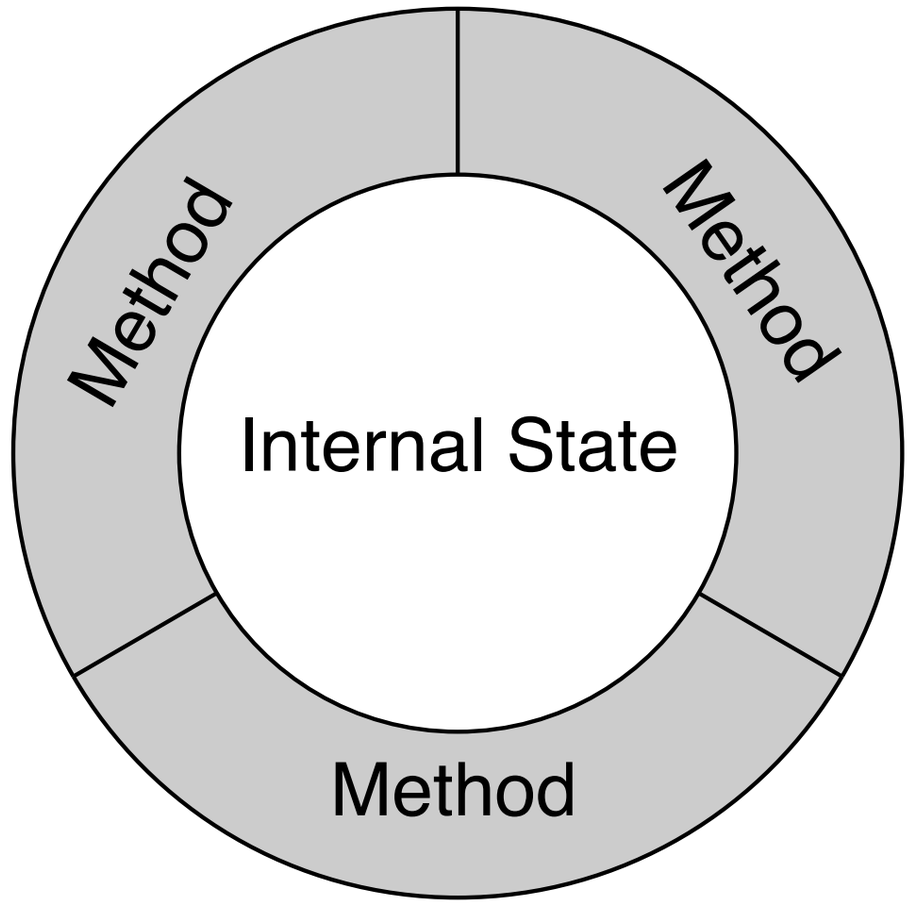
“Object-Oriented”

“I thought of objects being like biological cells and/or individual computers on a network, only able to communicate with messages”

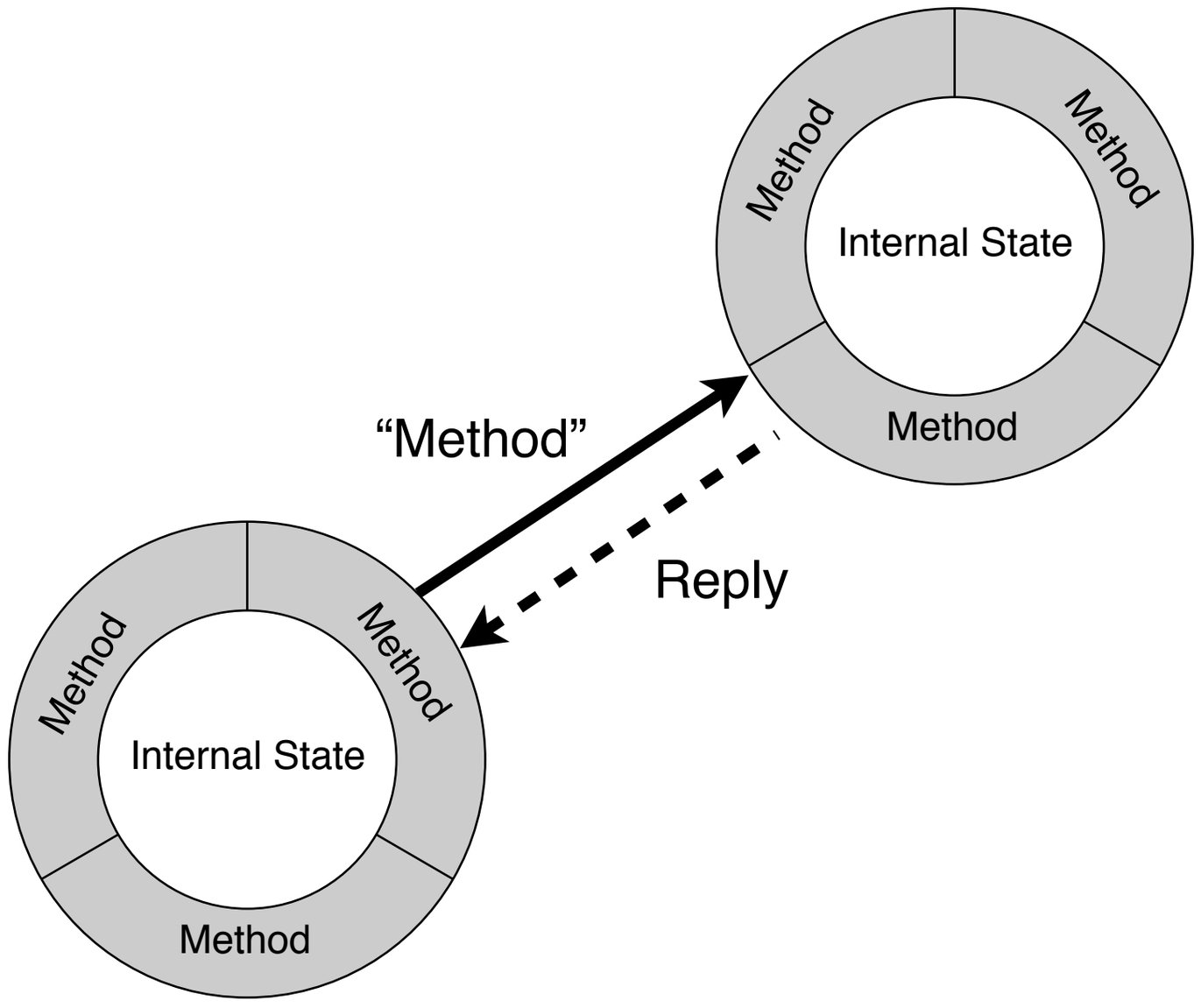
– Alan Kay, one of the fathers of OOP

- Procedural: *functions* operating on *data*
- Object-oriented: *objects* communicating with *messages*

Objects



Objects


Method
Method
Internal State
Method
"Method"
Reply
Method
Method
Internal State
Method

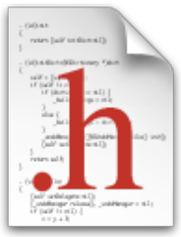
So what does Objective-C even look like?

Objective-C Files

Header file
(public)

Implementation file
(private)

Header file



MyImage.h

```
#import <Cocoa/Cocoa.h>

@interface MyImage : NSObject
{
  NSString *name;
  char pixels[16][16];
}

- (id)initWithName:(NSString *)name;

- (void)setName:(NSString *)name;
- (NSString *)name;

- (char)getPixelAtX:(int)x y:(int)y;
- (void)setPixel:(char)value atX:(int)x y:(int)y;

- (NSData *)imageData;

@end
```

Header file

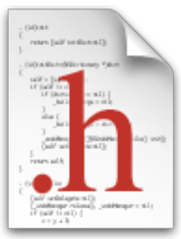
MyImage.h

```
#import <Cocoa/Cocoa.h>

@interface MyImage : NSObject
{
  NSString *name;
  char pixels[16][16];
}
```

Import Cocoa libraries so we can use Cocoa classes

```
- (id)initWithName:(NSString *)name;

- (void)setName:(NSString *)name;
- (NSString *)name;

- (char)getPixelAtX:(int)x y:(int)y;
- (void)setPixel:(char)value atX:(int)x y:(int)y;

- (NSData *)imageData;

@end
```

Header file

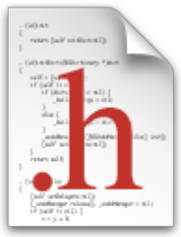


MyImage.h

```
#import <Cocoa/Cocoa.h>

@interface MyImage : NSObject
{
  NSString *name;
  char pixels[16][16];
}

- (id)initWithName:(NSString *)name;

- (void)setName:(NSString *)name;
- (NSString *)name;

- (char)getPixelAtX:(int)x y:(int)y;
- (void)setPixel:(char)value atX:(int)x y:(int)y;

- (NSData *)imageData;

@end
```

Create a new *class* called MyImage, a subclass of NSObject

Header file



MyImage.h

```
#import <Cocoa/Cocoa.h>

@interface MyImage : NSObject
{
  NSString *name;
  char pixels[16][16];
}

- (id)initWithName:(NSString *)name;

- (void)setName:(NSString *)name;
- (NSString *)name;

- (char)getPixelAtX:(int)x y:(int)y;
- (void)setPixel:(char)value atX:(int)x y:(int)y;

- (NSData *)imageData;

@end
```

Define the *instance variables* for the MyImage class

Header file

MyImage.h

```
#import <Cocoa/Cocoa.h>

@interface MyImage : NSObject
{
  NSString *name;
  char pixels[16][16];
}

- (id)initWithName:(NSString *)name;

- (void)setName:(NSString *)name;
- (NSString *)name;

- (char)getPixelAtX:(int)x y:(int)y;
- (void)setPixel:(char)value atX:(int)x y:(int)y;

- (NSData *)imageData;

@end
```

Declare the *methods* for the MyImage class

Declaring a Method

```
- (char)getPixelAtX:(int)x y:(int)y;
```

Declaring a Method

```
- (char)getPixelAtX:(int)x y:(int)y;
```

The *return type* of the method — in other words, the type of the expression `[obj getPixelAtX:x y:y]`

Declaring a Method

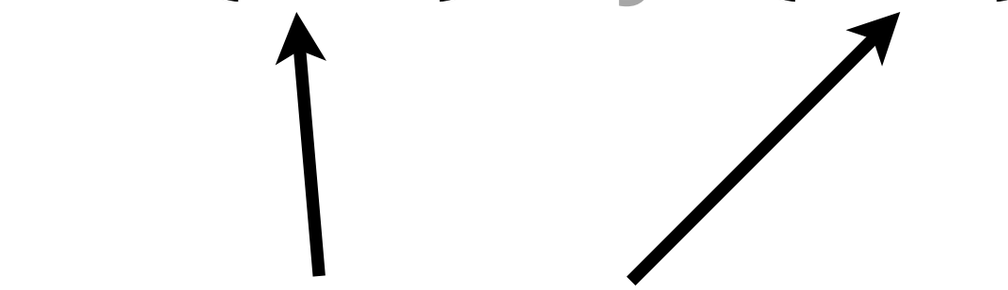
```
- (char)getPixelAtX:(int)x y:(int)y;
```

The name of the method, also called its *selector*

Objective-C uses *named parameters*, a feature from Smalltalk which improves readability

Declaring a Method

```
- (char)getPixelAtX:(int)x y:(int)y;
```



Implementation file

MyImage.m

```
#import "MyImage.h"

@implementation MyImage

- (NSString *)name {
  return name;
}
```

```
- (void)setPixel:(char)value atX:(int)x y:(int)y {
  if (x < 0 || y < 0 || x >= 16 || y >= 16) {
    [NSException raise:@"InvalidRangeException"
format:@"Pixel location outside image range"];
  }
  pixels[x][y] = value;
}

// all the other methods

@end
```

Implementation file

MyImage.m

```
#import "MyImage.h"

@implementation MyImage

- (NSString *)name {
  return name;
}
```

Import the header file so we know what we're implementing

```
- (void)setPixel:(char)value atX:(int)x y:(int)y {
  if (x < 0 || y < 0 || x >= 16 || y >= 16) {
    [NSException raise:@"InvalidRangeException"
format:@"Pixel location outside image range"];
  }
  pixels[x][y] = value;
}

// all the other methods

@end
```

Implementation file

MyImage.m

```
#import "MyImage.h"

@implementation MyImage

- (NSString *)name {
  return name;
}
```

Tell Obj-C we are defining methods for MyImage

```
- (void)setPixel:(char)value atX:(int)x y:(int)y {
  if (x < 0 || y < 0 || x >= 16 || y >= 16) {
    [NSException raise:@"InvalidRangeException"
format:@"Pixel location outside image range"];
  }
  pixels[x][y] = value;
}

// all the other methods

@end
```

Implementation file

MyImage.m

```
#import "MyImage.h"

@implementation MyImage

- (NSString *)name {
  return name;
}
```

Define the methods using Objective-C code

```
- (void)setPixel:(char)value atX:(int)x y:(int)y {
  if (x < 0 || y < 0 || x >= 16 || y >= 16) {
    [NSException raise:@"InvalidRangeException"
format:@"Pixel location outside image range"];
  }
  pixels[x][y] = value;
}

// all the other methods

@end
```

Sending a Message

```
[object method];
```

Sending a Message

```
int age1 = [person age];
int age2 = [[person father] age];
```

Sending a Message

```
[array addObject:obj];
[array insertObject:obj atIndex:0];
```

self and super

self is a reference to the current object

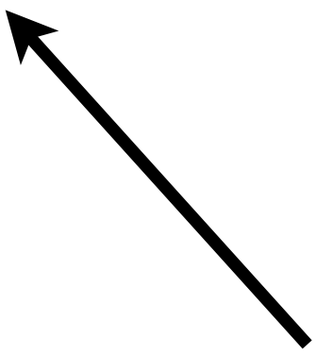
```
- (NSTimeInterval)age
{
  return [[NSDate date] timeIntervalSinceDate:[self birthday]];
}
```

super allows access to the superclass's methods

```
- (NSTimeInterval)age
{
  return [super age] + 60;
}
```

Class Methods

```
+ (BOOL)isSubclassOfClass:(Class)aClass;
```



The plus sign indicates that this is a *class method*, received by the class itself (instead of some particular instance of the class)

Class Methods

```
+ (BOOL)isSubclassOfClass:(Class)aClass;
```

Class methods are called on the class itself

```
[NSButton isSubclassOfClass:[NSControl class]]
```

Variables

- Variables are declared just like in C:

```
int i = 0, j;
float x = 1.0;
NSString *str = @"asdf";
```

- Objects are *always* pointers (`NSString *`)
- Instead of `"C strings"`, Cocoa uses `@"NSStrings"`, which can contain Unicode text

Objective-C Types

- Dynamically-typed:

```
id fido;
```

- Statically-typed:

```
MyDog *fido;
```

- Objective-C uses *dynamic binding*; method names are not resolved until run time

nil

- `nil` is the “nothing object”
- If `person == nil` then:

```
[person age] == 0
[person name] == nil
[person eat:taco] // fails silently
```

- Some methods take `nil` as a “don’t care” parameter

BOOL

- Objective-C was developed before C99, when C gained a boolean type

	ObjC	YES	NO
C++ / C99		`true`	`false`
actual value		`1`	`0`
C		`TRUE`	`FALSE`

Class

- Classes are (almost) like any other object
- Type is `Class` , no *

```
Class myClass = [MyClass class];
[myClass classMethod];
if ([someObject isKindOfClass:myClass])
  // the object is a MyClass
```

Selectors

Includes all colons

```
SEL fetchSelector = @selector(fetch:);
[fido performSelector:fetchSelector withObject:theBall];
```

is equivalent to

```
[fido fetch:theBall];
```

but you can store `SEL`s and pass them around
(you can even call `NSSelectorFromString(NSString *aSelectorName)`!)

Selectors in Cocoa

Delegates / “duck typing”

```
if ([obj respondsToSelector:@selector(fetch:)]) {
    [obj fetch:theBall];
}
```

Target / action pattern

```
[button setTarget:self];
[button setAction:@selector(buttonPressed:)];
```

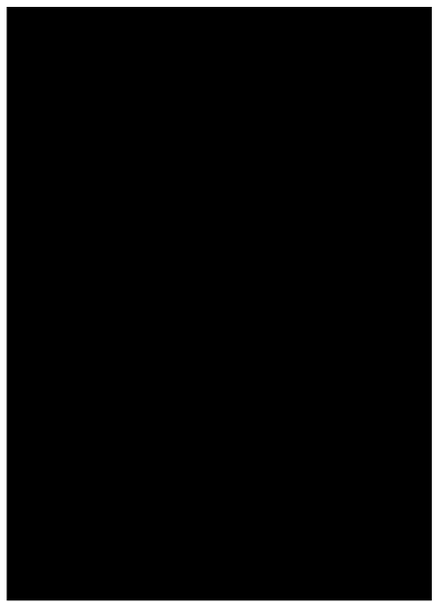
Cocoa Foundations

The Life of an Object

- Allocation
- Initialization
- …
- Deallocation

The Life of an Object

```
+ (id)alloc;
```



```
- (id)init;
```

```
MyClass *myInstance = [[MyClass alloc] init];
```

The Life of an Object

```
- (void)dealloc;
```

This is called by Cocoa — never call `dealloc` directly!

The Life of an Object

- Your objects should have:
- an `init` method which initializes all instance variables
- a `dealloc` method which cleans up any resources you have allocated
- We will discuss this in more depth when we cover *memory management*

Naming Conventions

- Variables start with a lowercase letter and use capitalization to distinguish words: e.g. `thisIsAGoodName`, but not `a_bad_name`
- Class names start with a prefix indicating the project, company, or author to avoid name collisions: e.g. `SRScroller` or `ProjMainView`, but not `mutableString` or `TEXT_FIELD`