Objective-C Reference Card

(for Java Programmers)

Basic Syntax

All of C syntax is inherited by Objective C with the following additions:

Declare objectx to be a pointer to an object of type MyClass, allocate and initialize it:

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

Invoke *methodf* of objectx, no args `[objectx methodf];`

Invoke *methodg* of objectx, passing arg1

```
[objectx methodg: arg1];
```

Methods with more than one argument:

```
[objectx initWithData: myData andParent: myParent];
```

Header File

MyClass.h:

```
@interface MyClass : MySuperClass
{
   int instanceVar1;
   NSString *instanceVar2;
   MyClass *nextOneOfMe;
}
- (void) methodf;
- (void) methodg: (ClassA *) argname;
- (void) initWithData: (Data *) data
   andParent: (MyClass *) parent;
+ (void) classMethod;
@end
```

Implementation File

MyClass.m:

```
#import"MyClass.h"
@implementation MyClass
- (void) methodf
{
   // do something good
}
- (void) methodg: (ClassA *) arg
{
   // do something good with arg
}
- (void) initWithData: (Data *) data
   andParent: (MyClass *) parent;
{
   // do something good with data and parent
}
@end
```

Protocols (like Java Interfaces)

MyProtocol.h:

```
@protocol MyProtocol
- (void) aProtocolMethod;
- (void) anotherProcotolMethod;
@end
```

A class that adopts a protocol would do the following:

```
#import "MyProtocol.h"
@interface ClassName : ItsSuperclass < MyProtocol,
                 AnotherProtocol >
  // method declarations
@end
```

Categories (extend any class)

If you want to extend any class in the system, in this example `ClassName`, first define a category in a header file. *Note: a category cannot define any instance variables, just new methods.*

CategoryName.h:

```
#import "ClassName.h"
@interface ClassName ( CategoryName )
  // method declarations
@end
```

Here is the implementation file

CategoryName.m:

```
#import "CategoryName.h"
@implementation ClassName ( CategoryName )
  // method definitions
@end
```

Note: Categories that extend the class `NSObject` are called *informal protocols*, and behave much like a protocol: they specify a set of behaviors that a particular object has. See *Misc Hints* for testing whether an object has a behavior.

Note$_2$: Categories can be declared *within* an implementation file, which is a common way to create "private" methods.

NSString constants

```
NSString aString = @"the value of aString";
```

aString is a fully valid instance of `NSString`, so you can send it messages like this:

```
int length = [aString length];
NSString upper = [@"Some String" uppercaseString];
```

Memory Management 101

If an object calls `alloc`, `copy`, or `retain`, it must also eventually `release` the object (perhaps in another method, or at least in its `-dealloc` method.

`autorelease` performs a `release` some time after the *calling* method has exited. It allows the calling method to use (or store and `retain`) the object before the `release` happens.

If any method stores a pointer to an object internally, it must `retain` that object until that pointer is cleared. Care must be taken when objects have circular references.

Accessor methods pattern:

```
-(NSString *) getAttr {
   return attr;
}
-(void) setAttr:(NSString *)newAttr {
   id oldAttr = attr;
   attr = [newAttr retain];
   [oldAttr release];
}
```

Note: it's a good idea to have accessors for all instance variables. It helps with memory management as well as Key/Value encoding. Thus, even though Objective C lets you declare `@public`, `@private` *and* `@protected` *instance variables, external classes should always use accessors instead of directly modifying them. Oddly, there's no way to declare private methods. See* **Categories'** *Note$_2$ to see how this is handled.*

Exception Handling

As of MacOS X 10.3, exceptions are very similar to Java's. Here's an example:

```
 @throw myException;
...
@try {
   [cup fill];
} @catch (NSException *exception) {
   NSLog(@"main: Caught %@: %@",
      [exception name], [exception reason]);
} @finally {
   ...
}
```

Misc Hints

Here's how you do the equivalent of Java's `instanceof`:

`[anObject isKindOfClass:[NSPopUpButton class]]`

Works for a class, and

`[anObject conformsToProtocol: @protocol(MyProtocol)]`

checks whether an object implements a protocol, and

`[anObject respondsToSelector: @selector(methodWithArg:)]`is pretty much self-explanatory.

`@synchronized(anObject){...}` ensures that only one thread runs the enclosed code, using anObject as the lock.*10.3 feature.*