# Creating Objects

- Most of the time, we create objects with `alloc` and `init`...

```
NSMutableArray *cards = [[NSMutableArray alloc] init];
CardMatchingGame *game = [[CardMatchingGame alloc] initWithCardCount:12 usingDeck:d];
```

- Or with class methods

```
NSString's + (id)stringWithFormat:(NSString *)format, ...
NSString *moltuae = [NSString stringWithFormat:@"%d", 42];
UIButton's + (id)buttonWithType:(UIButtonType)buttonType;
NSMutableArray's + (id)arrayWithCapacity:(int)count;
NSArray's + (id)arrayWithObject:(id)anObject;
```

- Sometimes both a class creator method and `init` method exist

  `[NSString stringWithFormat:...]` same as `[[NSString alloc] initWithFormat:...]`

  Don't be disturbed by this. Using either version is fine.

  iOS seems to be moving more toward the `alloc/init` versions with new API, but is mostly neutral.

# Creating Objects

- You can also ask other objects to create new objects for you

  NSString's `- (NSString *)stringByAppendingString:(NSString *)otherString;`
  NSArray's `- (NSString *)componentsJoinedByString:(NSString *)separator;`
  NSString's & NSArray's `- (id)mutableCopy;`

- But not all objects given out by other objects are newly created

  NSArray's `- (id)lastObject;`
  NSArray's `- (id)objectAtIndex:(int)index;`
  Unless the method has the word "copy" in it, if the object already exists, you get a pointer to it.
  If the object does not already exist (like the first 2 examples at the top), then you're creating.

# nil

- Sending messages to `nil` is (mostly) okay. No code executed.

  If the method returns a value, it will return zero.

  `int i = [obj methodWhichReturnsAnInt];` // `i` will be zero if `obj` is `nil`

  It is absolutely fine to depend on this and write code that uses this (don't get too cute, though).

  But be careful if the method returns a C struct. Return value is undefined.

  `CGPoint p = [obj getLocation];` // `p` will have an undefined value if `obj` is `nil`

## Dynamic Binding

- Objective-C has an important type called id

  It means "pointer to an object of unknown/unspecified" type.

  `id myObject;`

  Really all object pointers (e.g. `NSString *`) are treated like `id` at runtime.

  But at compile time, if you type something `NSString *` instead of `id`, the compiler can help you.

  It can find bugs and suggest what methods would be appropriate to send to it, etc.

  If you type something using `id`, the compiler can't help very much because it doesn't know much.

  Figuring out the code to execute when a message is sent at runtime is called "dynamic binding."

- Is it safe?

  Treating all object pointers as "pointer to unknown type" at runtime seems dangerous, right?

  What stops you from sending a message to an object that it doesn't understand?

  Nothing. And your program crashes if you do so. Oh my, Objective-C programs must crash a lot!

  Not really.

  Because we mostly use static typing (e.g. `NSString *`) and the compiler is really smart.

## Dynamic Binding

- Static typing

```
NSString *s = @“x”;   // “statically” typed (compiler will warn if s is sent non-NSString messges).
id obj = s;           // not statically typed, but perfectly legal; compiler can’t catch [obj rank]
NSArray *a = obj;     // also legal, but obviously could lead to some big trouble!
```

Compiler will not complain about assignments between an id and a statically typed variable.

Sometimes you are silently doing this. You have already done so!

```
- (int)match:(NSArray *)otherCards
{
    …
    PlayingCard *otherCard = [otherCards firstObject]; // firstObject returns id!
    …
}
```

Never use “id *” by the way (that would mean “a pointer to a pointer to an object”).

# Object Typing

```
@interface Vehicle
- (void)move;
@end
@interface Ship : Vehicle
- (void)shoot;
@end

Ship *s = [[Ship alloc] init];
[s shoot];
[s move];
```

No compiler warning.
Perfectly legal since `s` "isa" `Vehicle`.
Normal object-oriented stuff here.

# Object Typing

```
@interface Vehicle
- (void)move;
@end
@interface Ship : Vehicle
- (void)shoot;
@end

Ship *s = [[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v = s;
```

No compiler warning.
Perfectly legal since s "isa" Vehicle.

# Object Typing

```
@interface Vehicle
- (void)move;
@end
@interface Ship : Vehicle
- (void)shoot;
@end

Ship *s = [[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v = s;
[v shoot];
```

Compiler warning!
Would not crash at runtime though.
But only because we know v is a Ship.
Compiler only knows v is a Vehicle.

# Object Typing

```
@interface Vehicle
- (void)move;
@end
@interface Ship : Vehicle
- (void)shoot;
@end

Ship *s = [[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v = s;
[v shoot];

id obj = ...;
[obj shoot];
```

No compiler warning.
The compiler knows that the method `shoot` exists,
so it's not impossible that `obj` might respond to it.
But we have not typed `obj` enough for the compiler to be sure it's wrong.
So no warning.
Might crash at runtime if `obj` is not a `Ship`
(or an object of some other class that implements a `shoot` method).

# Object Typing

```
@interface Vehicle
- (void)move;
@end
@interface Ship : Vehicle
- (void)shoot;
@end

Ship *s = [[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v = s;
[v shoot];

id obj = ...;
[obj shoot];
[obj someMethodNameThatNoObjectAnywhereRespondsTo];
```

Compiler warning!
Compiler has never heard of this method.
Therefore it's pretty sure obj will not respond to it.

# Object Typing

```
@interface Vehicle
- (void)move;
@end
@interface Ship : Vehicle
- (void)shoot;
@end

Ship *s = [[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v = s;
[v shoot];

id obj = ...;
[obj shoot];
[obj someMethodNameThatNoObjectAnywhereRespondsTo];

NSString *hello = @"hello";
[hello shoot];
```

Compiler warning.
The compiler knows that `NSString` objects do not respond to `shoot`.
Guaranteed crash at runtime.

# Object Typing

```
@interface Vehicle
- (void)move;
@end
@interface Ship : Vehicle
- (void)shoot;
@end

Ship *s = [[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v = s;
[v shoot];

id obj = ...;
[obj shoot];
[obj someMethodNameThatNoObjectAnywhereRespondsTo];

NSString *hello = @"hello";
[hello shoot];
Ship *helloShip = (Ship *)hello;
```

No compiler warning.
We are "casting" here.
The compiler thinks we know what we're doing.

# Object Typing

```
@interface Vehicle
- (void)move;
@end
@interface Ship : Vehicle
- (void)shoot;
@end

Ship *s = [[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v = s;
[v shoot];

id obj = ...;
[obj shoot];
[obj someMethodNameThatNoObjectAnywhereRespondsTo];

NSString *hello = @"hello";
[hello shoot];
Ship *helloShip = (Ship *)hello;
[helloShip shoot];
```

No compiler warning!
We've forced the compiler to think that the NSString is a Ship.
"All's well," the compiler thinks.
Guaranteed crash at runtime.

# Object Typing

```
@interface Vehicle
- (void)move;
@end
@interface Ship : Vehicle
- (void)shoot;
@end

Ship *s = [[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v = s;
[v shoot];

id obj = ...;
[obj shoot];
[obj someMethodNameThatNoObjectAnywhereRespondsTo];

NSString *hello = @"hello";
[hello shoot];
Ship *helloShip = (Ship *)hello;
[helloShip shoot];
[(id)hello shoot];
```

No compiler warning!
We've forced the compiler to ignore the object type by "casting" in line.
"All's well," the compiler thinks.
Guaranteed crash at runtime.

## Dynamic Binding

- So when would we ever intentionally use this dangerous thing!
  When we want to mix objects of different classes in a collection (e.g. in an `NSArray`).
  When we want to support the "blind, structured" communication in MVC (i.e. delegation).
  And there are other generic or blind communication needs.
  But to make these things safer, we're going to use two things: Introspection and Protocols.
- Introspection
  Asking at runtime what class an object is or what messages can be sent to it.
- Protocols
  A syntax that is "in between" `id` and static typing.
  Does not specify the class of an object pointed to, but does specify what methods it implements.
  Example …
  `id <UIScrollViewDelegate> scrollViewDelegate;`
  We'll cover how to declare and use protocols next week.

# Introspection

- All objects that inherit from NSObject know these methods …

  isKindOfClass: returns whether an object is that kind of class (inheritance included)
  isMemberOfClass: returns whether an object is that kind of class (no inheritance)
  respondsToSelector: returns whether an object responds to a given method
  It calculates the answer to these questions at runtime (i.e. at the instant you send them).

- Arguments to these methods are a little tricky

  Class testing methods take a Class
  You get a Class by sending the class method class to a class (not the instance method class).

```
if ([obj isKindOfClass:[NSString class]]) {
    NSString *s = [(NSString *)obj stringByAppendingString:@"xyzzy"];
}
```

# Introspection

- Method testing methods take a selector (SEL)

  Special @selector() directive turns the name of a method into a selector

```
if ([obj respondsToSelector:@selector(shoot)]) {
    [obj shoot];
} else if ([obj respondsToSelector:@selector(shootAt:)]) {
    [obj shootAt:target];
}
```

- SEL is the Objective-C "type" for a selector

```
SEL shootSelector = @selector(shoot);
SEL shootAtSelector = @selector(shootAt:);
SEL moveToSelector = @selector(moveTo:withPenColor:);
```

# Introspection

- If you have a SEL, you can also ask an object to perform it …

Using the performSelector: or performSelector:withObject: methods in NSObject

```
[obj performSelector:shootSelector];
[obj performSelector:shootAtSelector withObject:coordinate];
```

Using makeObjectsPerformSelector: methods in NSArray

```
[array makeObjectsPerformSelector:shootSelector]; // cool, huh?
[array makeObjectsPerformSelector:shootAtSelector withObject:target]; // target is an id
```

In UIButton, `- (void)addTarget:(id)anObject action:(SEL)action …;`

```
[button addTarget:self action:@selector(digitPressed:) …];
```

# Demo

- How match: might be improved with introspection

# Foundation Framework

- NSObject

  Base class for pretty much every object in the iOS SDK

  Implements introspection methods discussed earlier.

  `- (NSString *)description` is a useful method to override (it's `%@` in `NSLog()`).
  Example … `NSLog(@"array contents are %@", myArray);`
  The `%@` is replaced with the results of invoking `[myArray description]`.

  Copying objects. This is an important concept to understand (why mutable vs. immutable?).

  ```
  - (id)copy;           // not all objects implement mechanism (raises exception if not)
  - (id)mutableCopy;    // not all objects implement mechanism (raises exception if not)
  ```

  It's not uncommon to have an array or dictionary and make a `mutableCopy` and modify that.
  Or to have a mutable array or dictionary and `copy` it to "freeze it" and make it immutable.
  Making copies of collection classes is very efficient, so don't sweat doing so.

# Foundation Framework

- NSArray

Ordered collection of objects.
Immutable. That's right, once you create the array, you cannot add or remove objects.
All objects in the array are held onto strongly.
Usually created by manipulating other arrays or with @[].

You already know these key methods ...
```
- (NSUInteger)count;
- (id)objectAtIndex:(NSUInteger)index; // crashes if index is out of bounds; returns id!
- (id)lastObject;  // returns nil (doesn't crash) if there are no objects in the array
- (id)firstObject;  // returns nil (doesn't crash) if there are no objects in the array
```

But there are a lot of very interesting methods in this class. Examples ...
```
- (NSArray *)sortedArrayUsingSelector:(SEL)aSelector;
- (void)makeObjectsPerformSelector:(SEL)aSelector withObject:(id)selectorArgument;
- (NSString *)componentsJoinedByString:(NSString *)separator;
```

# Foundation Framework

- NSMutableArray

  Mutable version of NSArray.

  Create with alloc/init or …
  + (id)arrayWithCapacity:(NSUInteger)numItems; // numItems is a performance hint only
  + (id)array; // [NSMutableArray array] is just like [[NSMutableArray alloc] init]

  NSMutableArray inherits all of NSArray's methods.
  Not just count, objectAtIndex:, etc., but also the more interesting ones mentioned last slide.

  And you know that it implements these key methods as well …
  - (void)addObject:(id)object; // to the end of the array (note id is the type!)
  - (void)insertObject:(id)object atIndex:(NSUInteger)index;
  - (void)removeObjectAtIndex:(NSUInteger)index;

# Enumeration

- Looping through members of an array in an efficient manner

  Language support using for-in.

  Example: NSArray of NSString objects

```
NSArray *myArray = ...;
for (NSString *string in myArray) { // no way for compiler to know what myArray contains
    double value = [string doubleValue]; // crash here if string is not an NSString
}
```

  Example: NSArray of id

```
NSArray *myArray = ...;
for (id obj in myArray) {
    // do something with obj, but make sure you don't send it a message it does not respond to
    if ([obj isKindOfClass:[NSString class]]) {
        // send NSString messages to obj with no worries
    }
}
```

# Foundation Framework

- NSNumber

  Object wrapper around primitive types like `int`, `float`, `double`, `BOOL`, `enum`s, etc.

  ```
  NSNumber *n = [NSNumber numberWithInt:36];
  float f = [n floatValue]; // would return 36.0 as a float (i.e. will convert types)
  ```

  Useful when you want to put these primitive types in a collection (e.g. `NSArray` or `NSDictionary`).

  New syntax for creating an `NSNumber` in iOS 6: `@()`

  ```
  NSNumber *three = @3;
  NSNumber *underline = @(NSUnderlineStyleSingle); // enum
  NSNumber *match = @([card match:@[otherCard]]); // expression that returns a primitive type
  ```

- NSValue

  Generic object wrapper for some non-object, non-primitive data types (i.e. C structs).

  e.g. `NSValue *edgeInsetsObject = [NSValue valueWithUIEdgeInsets:UIEdgeInsetsMake(1,1,1,1)]`

  Probably don't need this in this course (maybe when we start using points, sizes and rects).

# Foundation Framework

- NSData
  "Bag of bits." Used to save/restore/transmit raw data throughout the iOS SDK.
- NSDate
  Used to find out the time right now or to store past or future times/dates.
  See also NSCalendar, NSDateFormatter, NSDateComponents.
  If you are going to display a date in your UI, make sure you study this in detail (localization).
- NSSet / NSMutableSet
  Like an array, but no ordering (no objectAtIndex: method).
  member: is an important method (returns an object if there is one in the set isEqual: to it).
  Can union and intersect other sets.
- NSOrderedSet / NSMutableOrderedSet
  Sort of a cross between NSArray and NSSet.
  Objects in an ordered set are distinct. You can't put the same object in multiple times like array.

# Foundation Framework

- NSDictionary

Immutable collection of objects looked up by a key (simple hash table).
All keys and values are held onto strongly by an NSDictionary.

Can create with this syntax: `@{ key1 : value1, key2 : value2, key3 : value3 }`

```
NSDictionary *colors = @{ @"green" : [UIColor greenColor],
                          @"blue" : [UIColor blueColor],
                          @"red" : [UIColor redColor] };
```

Lookup using "array like" notation …

```
NSString *colorString = …;
UIColor *colorObject = colors[colorString]; // works the same as objectForKey: below
```

```
- (NSUInteger)count;
- (id)objectForKey:(id)key; // key must be copyable and implement isEqual: properly
```

NSStrings make good keys because of this.
See NSCopying protocol for more about what it takes to be a key.

# Foundation Framework

- NSMutableDictionary

  Mutable version of NSDictionary.

  Create using alloc/init or one of the + (id)dictionary... class methods.

  In addition to all the methods inherited from NSDictionary, here are some important methods ...

```
- (void)setObject:(id)anObject forKey:(id)key;
- (void)removeObjectForKey:(id)key;
- (void)removeAllObjects;
- (void)addEntriesFromDictionary:(NSDictionary *)otherDictionary;
```

# Enumeration

- Looping through the keys or values of a dictionary

  Example:

```
NSDictionary *myDictionary = ...;
for (id key in myDictionary) {
    // do something with key here
    id value = [myDictionary objectForKey:key];
    // do something with value here
}
```