Mobile Application Development

Lecture 13
Introduction to Objective-C

Part II

Lecture Summary

- Object creation
- Memory management
- Automatic Reference Counting
- Protocols
- Categories

Object creation

- The procedure for creating an object in Objective-C requires two steps:
  1. memory allocation
  2. object instantiation and initialization
- Memory allocation is performed by invoking the `alloc` method of `NSObject`
- Object initialization occurs through an initializer method, which always begins with `init`

Memory allocation

- The `alloc` method asks the operating system to reserve a memory area to keep the object
- `alloc` is inherited by all classes that extend the `NSObject` class
- `alloc` also takes care of setting all instance variables to `nil`

Object initialization

- The `alloc` method returns allocates an object with `nil` instance variables
- Instance variables are properly set using an **initializer** method
- The name of an initializer always begins with `init`
- Every class has a so-called **designated initializer**, which is responsible for setting all the instance variables in such a way that the returned object is in a consistent state
- Any initializer must first invoke the designated initializer of the superclass and check that it did not return a `nil` object
- Initializers can take arguments, typically those that must be passed to properly set instance variables
- It is possible to have more than one initializer, but all other initializers should always call the designated initializer

Structure of initializers

```
- (id) initWithName:(NSString *)name latitude:(double)latitude longitude:(double)longitude;
```

MDPoi.h

```
- (id) initWithName:(NSString *)name latitude:(double)latitude longitude:(double)longitude{
  if(self = [super init]){
     _name = [name copy];
     _latitude = latitude;
     _longitude = longitude;
  }
  return self;
}
```

MDPoi.m

Declaration of initializers

```
- (id) initWithName:(NSString *)name latitude:(double)latitude longitude:(double)longitude;
```

in iOS6 and before, initializers should always return `id`

in iOS7 a new special keyword `instancetype` has been introduced to explicitly say that the method returns an object of the same type of the object it has been sent the message

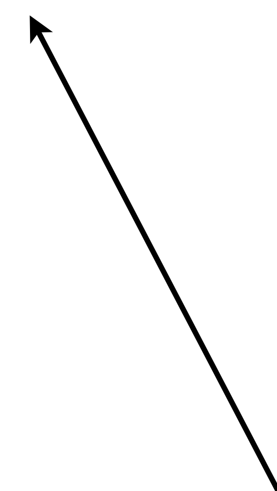
Structure of initializers

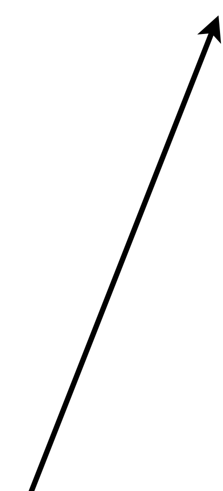
```
- (id) initWithName:(NSString *)name latitude:(double)latitude longitude:(double)longitude;
```

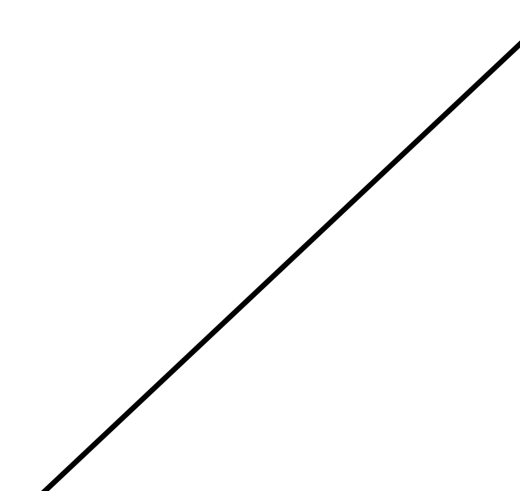
the name of an initializer starts with "`init`" by convention

Structure of initializers

```
- (id) initWithName:(NSString *)name latitude:(double)latitude longitude:(double)longitude;
```







initializer method's arguments

Implementation of initializers

```
- (id) initWithName:(NSString *)name latitude:(double)latitude longitude:(double)longitude{
  if(self = [super init]){
     _name = [name copy];
     _latitude = latitude;
     _longitude = longitude;
  }
  return self;
}
```

Implementation of initializers

```
- (id) initWithName:(NSString *)name latitude:(double)latitude longitude:(double)longitude{
  if(self = [super init]){
     _name = [name copy];
     _latitude = latitude;
     _longitude = longitude;
  }
  return self;
}
```

call the designated initializer of the superclass

Implementation of initializers

```
- (id) initWithName:(NSString *)name latitude:(double)latitude longitude:(double)longitude{
  if(self = [super init]){
     _name = [name copy];
     _latitude = latitude;
     _longitude = longitude;
  }
  return self;
}
```

assign the reference returned by the designated initializer of the superclass to `self`

Implementation of initializers

```
- (id) initWithName:(NSString *)name latitude:(double)latitude longitude:(double)longitude{
  if(self = [super init]){
      _name = [name copy];
      _latitude = latitude;
      _longitude = longitude;
  }
  return self;
}
```

check if the reference to the returned object is not `nil`

Implementation of initializers

```
- (id) initWithName:(NSString *)name latitude:(double)latitude longitude:(double)longitude{
  if(self = [super init]){
    _name = [name copy];
    _latitude = latitude;
    _longitude = longitude;
  }
  return self;
}
```

set all the instance variables

Implementation of initializers

```
- (id) initWithName:(NSString *)name latitude:(double)latitude longitude:(double)longitude{
  if(self = [super init]){
     _name = [name copy];
     _latitude = latitude;
     _longitude = longitude;
  }
  return self;
}
```

set all the instance variables with properties! **Not a good practice!!!!**

```
- (id) initWithName:(NSString *)name latitude:(double)latitude longitude:(double)longitude{
  if(self = [super init]){
     self.name = name;
     self.latitude = latitude;
     self.longitude = longitude;
  }
  return self;
}
```

Implementation of initializers

```
- (id) initWithName:(NSString *)name latitude:(double)latitude longitude:(double)longitude{
  if(self = [super init]){
     _name = [name copy];
     _latitude = latitude;
     _longitude = longitude;
  }
  return self;
}
```

return a reference to the initialized object

Creating an object

```
MDPoi *poi = [[MDPoi alloc] initWithName:@"MyPoi" latitude:45.2 longitude:10.12];
```

Creating an object

```
MDPoi *poi = [[MDPoi alloc] initWithName:@"MyPoi" latitude:45.2 longitude:10.12];
```

call `alloc`, returns a reference to an instance of `MDPoi`

Creating an object

```
MDPoi *poi = [[MDPoi alloc] initWithName:@"MyPoi" latitude:45.2 longitude:10.12];
```

call `init`ializer to set instance variables

Creating an object

```
MDPoi *poi = [[MDPoi alloc] initWithName:@"MyPoi" latitude:45.2 longitude:10.12];
```

init ialializer returns a reference to the initialized instance which is assigned to a pointer for later use

Getting an object

- There are many other ways in which you can get an object, other than calling `alloc`/`init`

```
NSString *str = [alphabet objectAtIndex:0];
```

- **How do we handle memory in this case?**

Memory management: stack and heap

- In Objective-C, all objects are allocated in the heap, so to access them we always use a pointer to the object
- There are two areas in which memory is managed: the stack and the heap
- Roughly speaking, the **stack** is a memory where data are stored with a last-in first-out policy
- The stack is where local (function) variables are stored
- The **heap** is where dynamic memory is allocated (e.g., when we use `new` in C++)
- Anytime an object is instantiated, we are asking to allocate and return a pointer to an area of the heap of the proper size
- But when we do not need the object anymore, we must free the memory so that it can be used again (`free()` in C or `delete` in C++)

The stack

- Suppose we have the code on the right side of this slide
- What does it happen in the stack?

```
- (void) countdown:(int)n{
  int j;
  if([self isEven:n] == YES) j = n/2;
  else j = n + 1;
  for(j; j > 0; j--){
     NSLog(@"%d", j);
  }
}

- (BOOL) isEven:(int)m{
  BOOL even = (m % 2 == 0);
  if(even == YES) return YES;
  else return NO;
}
```

The stack

- Each time a function gets called (or in OOP-terms, a method gets invoked), an activation record gets created and put in the stack, by copying the values and references in the stack as defined by the method’s signature

```
- (void) countdown:(int)n{
  int j;
  if([self isEven:n] == YES) j = n/2;
  else j = n + 1;
  for(j; j > 0; j--){
     NSLog(@"%d", j);
  }
}

- (BOOL) isEven:(int)m{
  BOOL even = (m % 2 == 0);
  if(even == YES) return YES;
  else return NO;
}
```

1

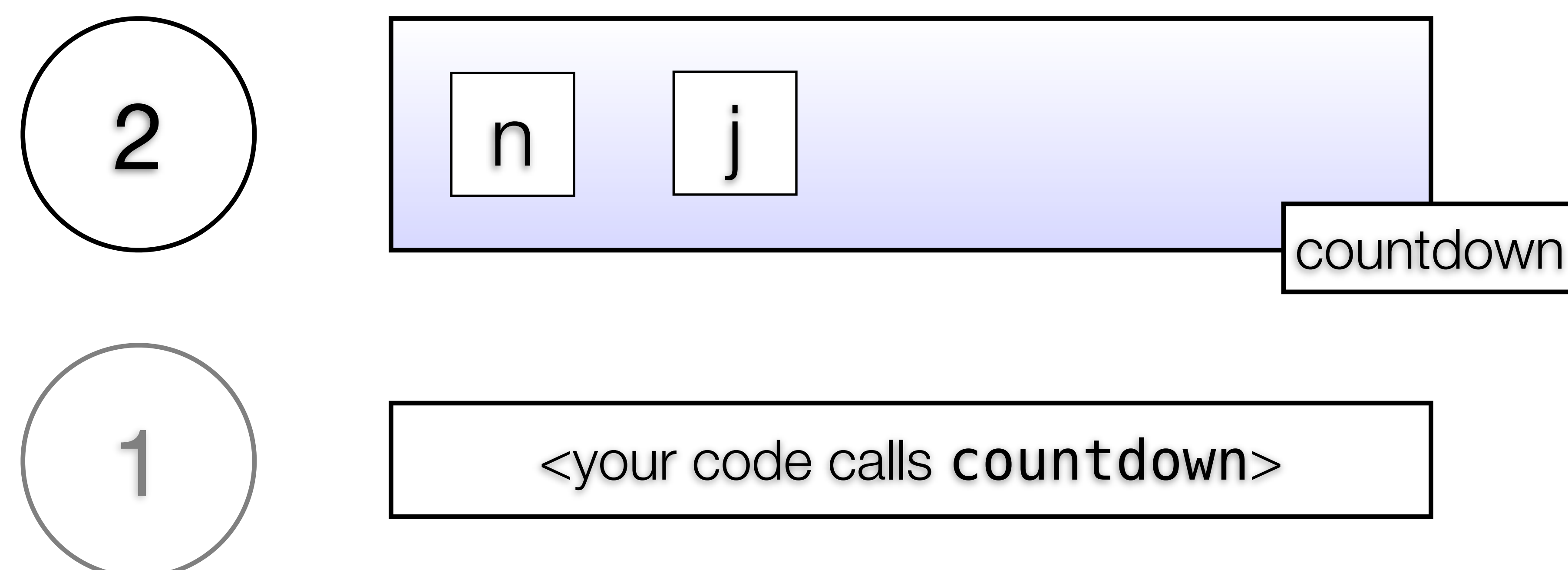
<your code calls `countdown`>

Stack

The stack

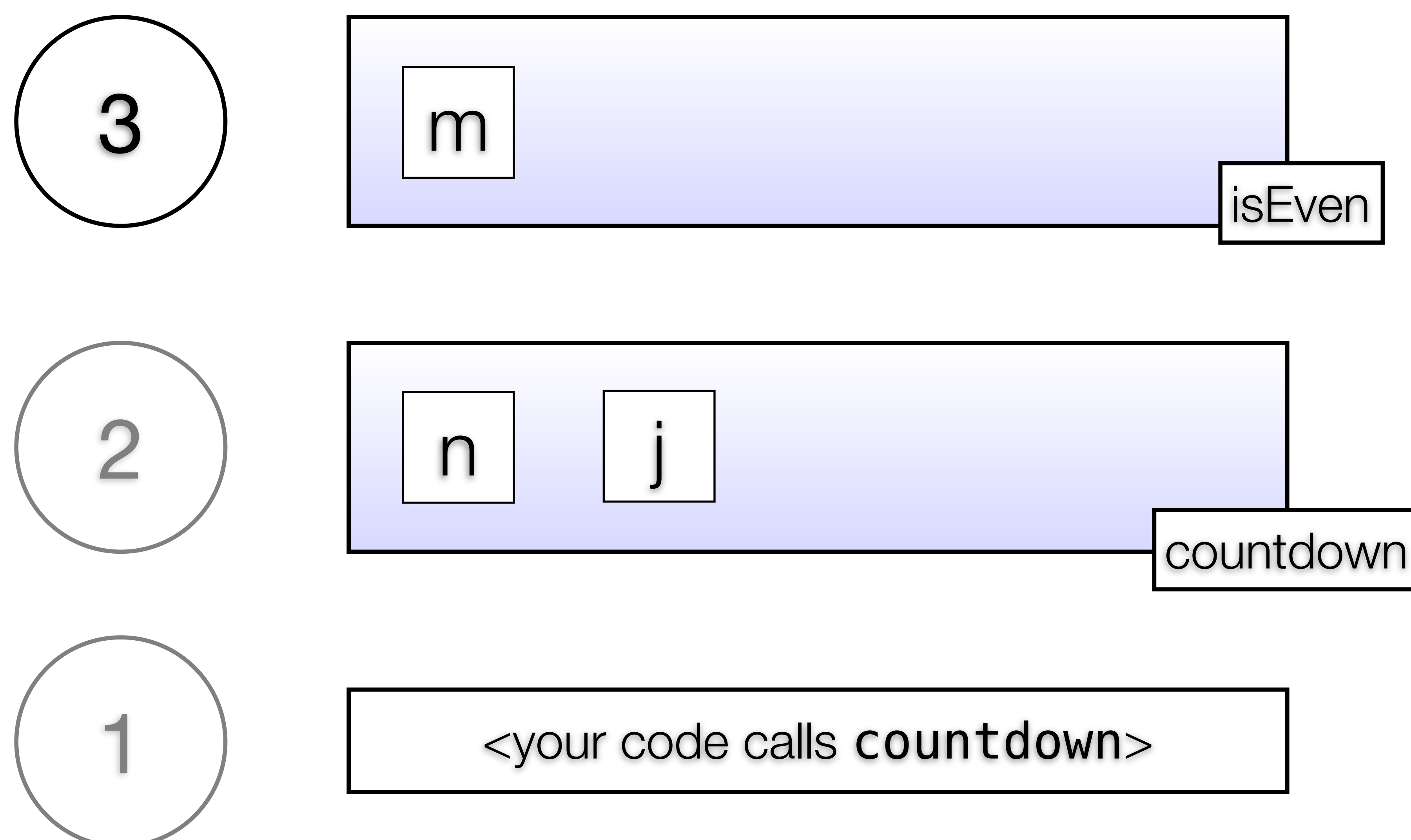
- Each time a function gets called (or in OOP-terms, a method gets invoked), an activation record gets created and put in the stack, by copying the values and references in the stack as defined by the method's signature

```
- (void) countdown:(int)n{
  int j;
  if([self isEven:n] == YES) j = n/2;
  else j = n + 1;
  for(j; j > 0; j--){
     NSLog(@"%d", j);
  }
}

- (BOOL) isEven:(int)m{
  BOOL even = (m % 2 == 0);
  if(even == YES) return YES;
  else return NO;
}
```



The stack

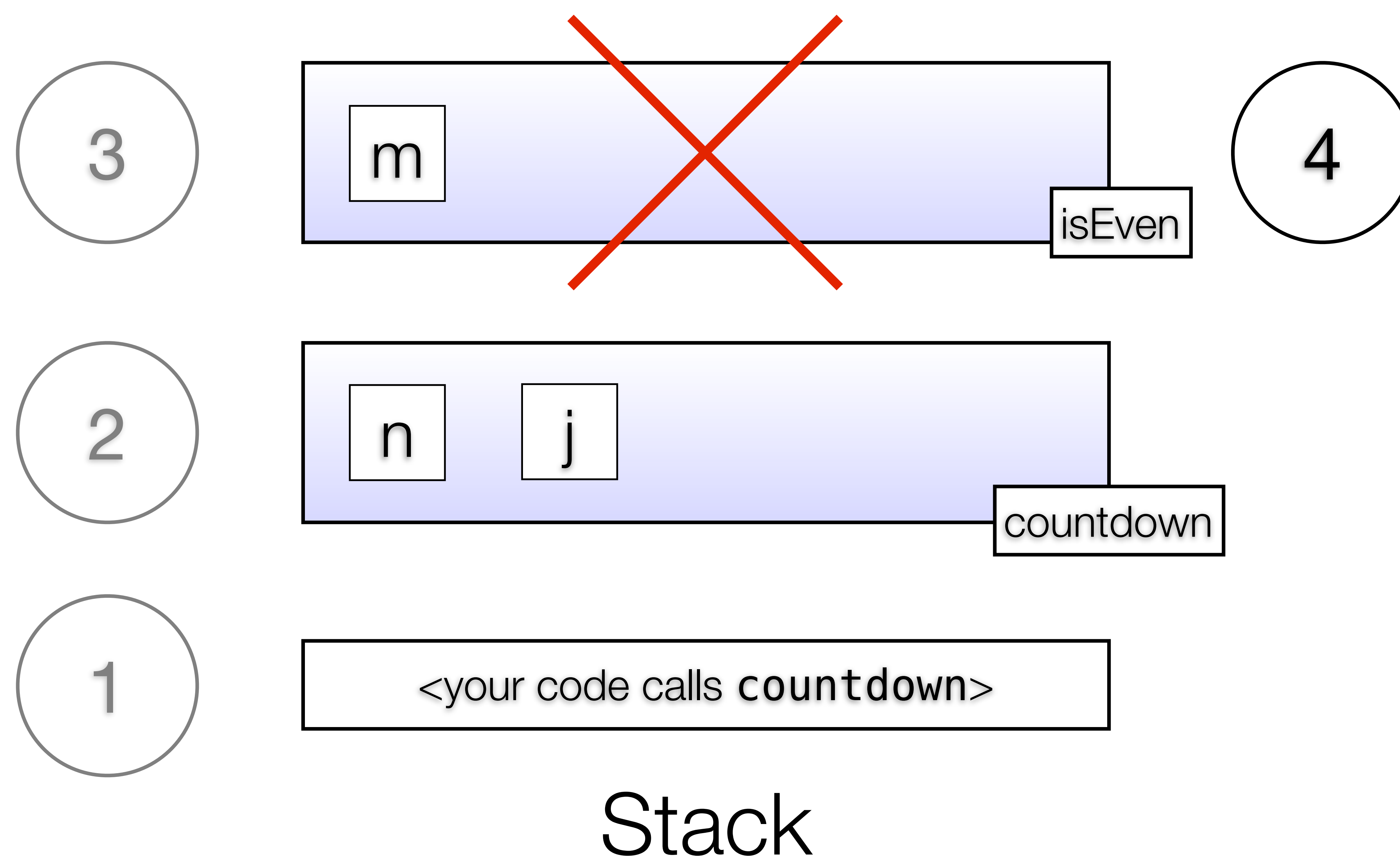
- Each time a function gets called (or in OOP-terms, a method gets invoked), an activation record gets created and put in the stack, by copying the values and references in the stack as defined by the method's signature




```
- (void) countdown:(int)n{
  int j;
  if([self isEven:n] == YES) j = n/2;
  else j = n + 1;
  for(j; j > 0; j--){
     NSLog(@"%d", j);
  }
}

- (BOOL) isEven:(int)m{
  BOOL even = (m % 2 == 0);
  if(even == YES) return YES;
  else return NO;
}
```

The stack

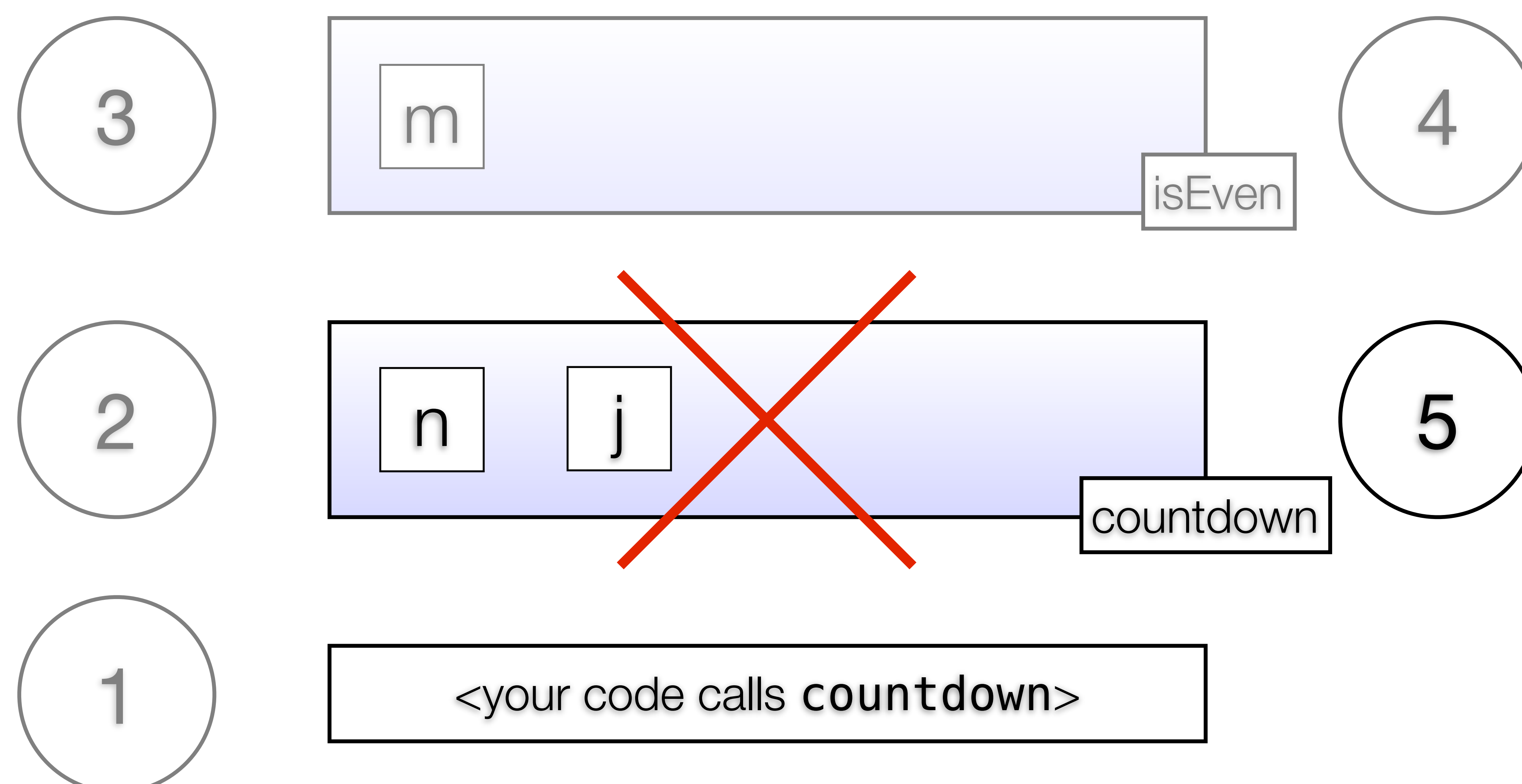
- When the method returns, its stack area gets destroyed




```
- (void) countdown:(int)n{
  int j;
  if([self isEven:n] == YES) j = n/2;
  else j = n + 1;
  for(j; j > 0; j--){
     NSLog(@"%d", j);
  }
}

- (BOOL) isEven:(int)m{
  BOOL even = (m % 2 == 0);
  if(even == YES) return YES;
  else return NO;
}
```

The stack

- When the method returns, its stack area gets destroyed

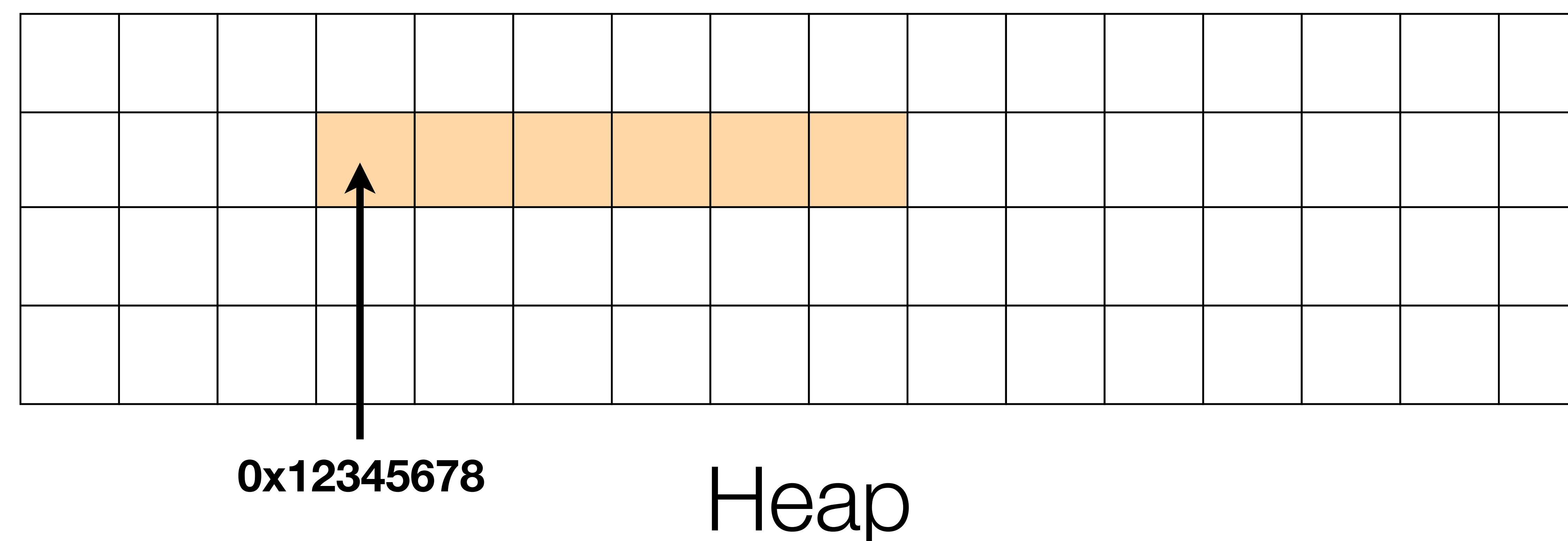



```
- (void) countdown:(int)n{
  int j;
  if([self isEven:n] == YES) j = n/2;
  else j = n + 1;
  for(j; j > 0; j--){
     NSLog(@"%d", j);
  }
}

- (BOOL) isEven:(int)m{
  BOOL even = (m % 2 == 0);
  if(even == YES) return YES;
  else return NO;
}
```

The heap

- Dynamic memory allocation allows to create and destroy objects on-the-fly as you need in your code
- As said before, dynamic memory allocation is performed in the heap, which is an area of memory managed by the operating system to be used as a common resource among processes
- When calling `malloc()` in C, or `new` in C++, the operating system will try to allocate the needed amount of space in the heap and return a pointer to the allocated area
- Once you get a pointer to the allocated area, it is your responsibility to use it and keep track of it, so that it can be freed once you are done using it: failing to free allocated memory in the heap results in a *memory leak*, which eventually might end up consuming all the memory resources of your OS

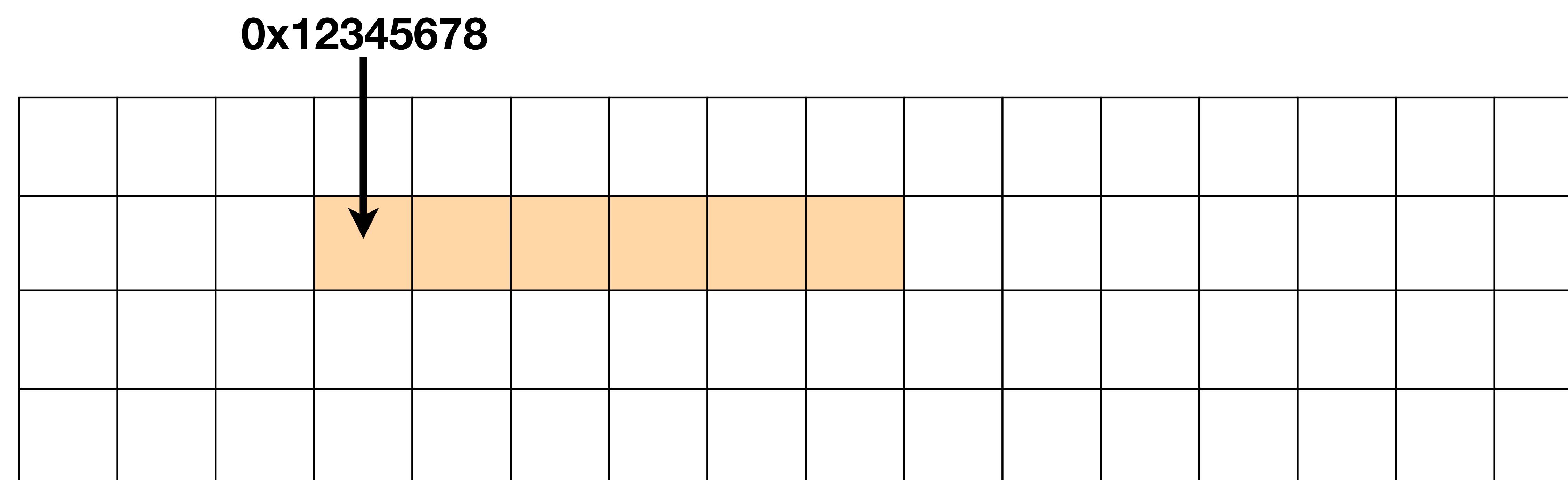
```
char* cstr = malloc(sizeof(char)*6);
for(int i = 0; i < 5; i++)
  cstr[i] = ('a' + i);
cstr[6] = '\0';
NSLog(@"%s",cstr);
free(cstr);
```



The heap

```
char* cstr = malloc(sizeof(char)*6);

for(int i = 0; i < 5; i++)
    cstr[i] = ('a' + i);

cstr[6] = '\0';

NSLog(@"%s",cstr);

free(cstr);
```

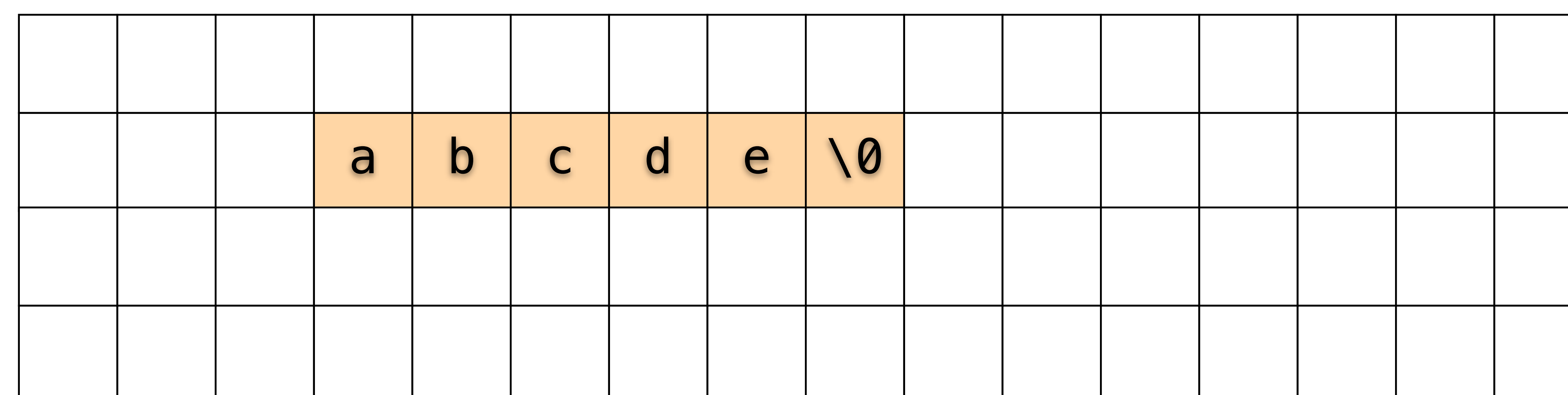
`malloc` returns the address of an area in the heap of the right size which is assigned to a pointer



Heap

The heap

```
char* cstr = malloc(sizeof(char)*6);

for(int i = 0; i < 5; i++)
    cstr[i] = ('a' + i);

cstr[6] = '\0';

NSLog(@"%s",cstr);

free(cstr);
```



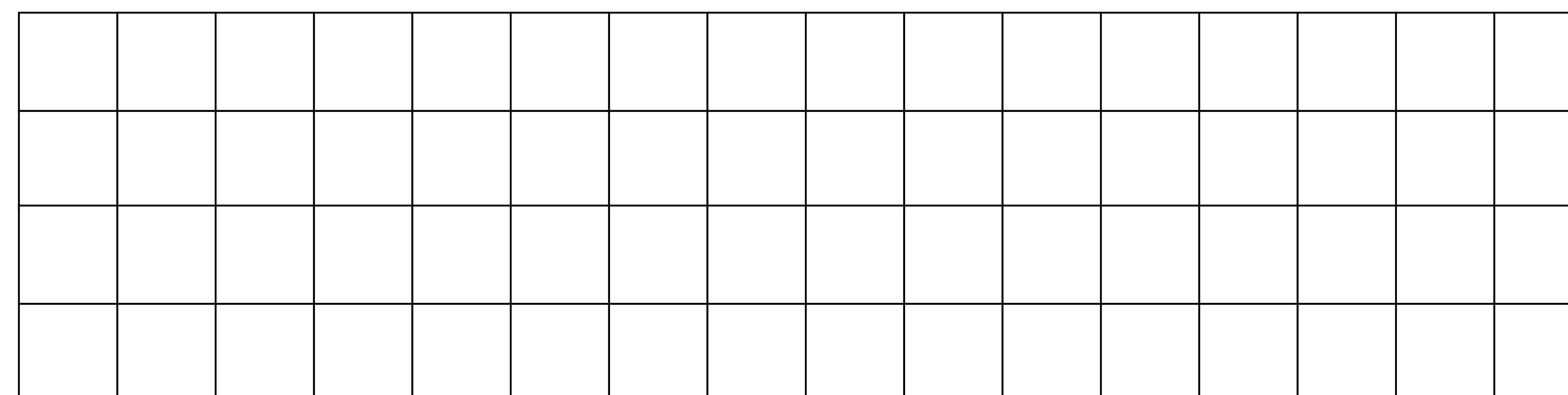
Heap

it is possible to access
the allocated memory
through the pointer

The heap

```
char* cstr = malloc(sizeof(char)*6);

for(int i = 0; i < 5; i++)
    cstr[i] = ('a' + i);

cstr[6] = '\0';

NSLog(@"%s",cstr);

free(cstr);
```

memory is deallocated by calling `free()`



Heap

Memory management

- Originally, Objective-C did not have anything like Java's garbage collector
- The technique used in Objective-C to achieve this is called **reference counting**
- Reference counting means that:

1. we keep a count of the number of times that we point to an object
2. when we need to get a reference to the object, we increment the count
3. when we are done with it, we decrease the count
4. when the counter goes to 0, the memory is freed (accessing a freed object will crash application): the method `dealloc` (inherited by `NSObject`) gets invoked

Manual Reference Counting

- An object returned by `alloc/init` has a reference count of 1
- NSObject defines two methods used to increment and decrement the reference count:
  - `retain`: increase the reference count by 1
  - `release`: decrease the reference count by 1
- We retain an object when we need to use it; the object is retained as long as needed in order to avoid that it will be destroyed while we are using it
- We release the object when we are done using it, so that the reference count can decrease and eventually reach 0 to be freed
- The method `retainCount` (inherited by `NSObject`) can be used to get the current reference count of an object

Manual Reference Counting

```
MDPoi *poi = [[MDPoi alloc] initWithName:@"MyPoi" latitude:45.2 longitude:10.12];
NSLog(@"retain count = %d", [poi retainCount]);
[poi retain];
NSLog(@"retain count = %d", [poi retainCount]);
[poi retain];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
```

retain count = 1

Manual Reference Counting

```
MDPoi *poi = [[MDPoi alloc] initWithName:@"MyPoi" latitude:45.2 longitude:10.12];
NSLog(@"retain count = %d", [poi retainCount]);
[poi retain];
NSLog(@"retain count = %d", [poi retainCount]);
[poi retain];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
```

retain count = 2

Manual Reference Counting

```
MDPoi *poi = [[MDPoi alloc] initWithName:@"MyPoi" latitude:45.2 longitude:10.12];
NSLog(@"retain count = %d", [poi retainCount]);
[poi retain];
NSLog(@"retain count = %d", [poi retainCount]);
[poi retain];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
```

retain count = 3 (arrow pointing to the second `[poi retain];`)

Manual Reference Counting

```
MDPoi *poi = [[MDPoi alloc] initWithName:@"MyPoi" latitude:45.2 longitude:10.12];
NSLog(@"retain count = %d", [poi retainCount]);
[poi retain];
NSLog(@"retain count = %d", [poi retainCount]);
[poi retain];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
```

retain count = 2

Manual Reference Counting

```
MDPoi *poi = [[MDPoi alloc] initWithName:@"MyPoi" latitude:45.2 longitude:10.12];
NSLog(@"retain count = %d", [poi retainCount]);
[poi retain];
NSLog(@"retain count = %d", [poi retainCount]);
[poi retain];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
```

retain count = 1

Manual Reference Counting

```
MDPoi *poi = [[MDPoi alloc] initWithName:@"MyPoi" latitude:45.2 longitude:10.12];
NSLog(@"retain count = %d", [poi retainCount]);
[poi retain];
NSLog(@"retain count = %d", [poi retainCount]);
[poi retain];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
```

retain count = 0

Manual Reference Counting

```
MDPoi *poi = [[MDPoi alloc] initWithName:@"MyPoi" latitude:45.2 longitude:10.12];
NSLog(@"retain count = %d", [poi retainCount]);
[poi retain];
NSLog(@"retain count = %d", [poi retainCount]);
[poi retain];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
```

`retain count = 0`

object `poi` gets deallocated!

Manual Reference Counting

```
MDPoi *poi = [[MDPoi alloc] initWithName:@"MyPoi" latitude:45.2 longitude:10.12];
NSLog(@"retain count = %d", [poi retainCount]);
[poi retain];
NSLog(@"retain count = %d", [poi retainCount]);
[poi retain];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
NSLog(@"retain count = %d", [poi retainCount]);
[poi release];
```

retain count = 0

object **poi** gets deallocated!

Object ownership

- The memory management model is based on object ownership
- Any object may have one or more owners
- As long as an object has at least one owner, it continues to exist
- Rules for object ownership:

1. You own any object you create (using `alloc`/`init`)
2. You can take ownership of an object using `retain` (since the original owner is who created it)
3. When you no longer need it, you must relinquish ownership of an object you own using `release`
4. You must not relinquish ownership of an object you do not own

- The number of `alloc`/`init` and `retain` must always match that of `release`

Temporary object ownership

- There are times when sending a release message might create a premature deallocation of an object

```
- (NSArray *)getAllPois{
  NSMutableArray *array = [[NSMutableArray alloc] init];
  ...
  [array release];
  return array;
}
```

Temporary object ownership

- There are times when sending a release message might create a premature deallocation of an object

```
- (NSArray *)getAllPois{
  NSMutableArray *array = [[NSMutableArray alloc] init];
  ...
  [array release];
  return array;
}
```

`release` brings reference count to 0 and gets deallocated, so the returned object does not exist anymore

Temporary object ownership

- There are times when sending a release message might create a premature deallocation of an object

```
- (NSArray *)getAllPois{
  NSMutableArray *array = [[NSMutableArray alloc] init];
  ...
  [array release];
  return array;
}
```

- To avoid this situation, we must instruct the object to be released later on: use `autorelease`

```
- (NSArray *)getAllPois{
  NSMutableArray *array = [[NSMutableArray alloc] init];
  ...
  [array autorelease];
  return array;
}
```

Temporary object ownership

- There are times when sending a release message might create a premature deallocation of an object

```
- (NSArray *)getAllPois{
  NSMutableArray *array = [[NSMutableArray alloc] init];
  ...
  [array release];
  return array;
}
```

- To avoid this situation, we must instruct the object to be released later on: use `autorelease`

```
- (NSArray *)getAllPois{
  NSMutableArray *array = [[NSMutableArray alloc] init];
  ...
  [array autorelease];
  return array;
}
```

`autorelease` sends a deferred `release` message, so the object gets deallocated after it is returned

Temporary object ownership

- It is possible to avoid the use of `alloc/init` and `autorelease`

```
- (NSArray *)getAllPois{
  NSMutableArray *array = [[NSMutableArray alloc] init];
  ...
  [array autorelease];
  return array;
}
```

- We can get an `autorelease`d object so we don't need to worry about (auto)releasing it

```
- (NSArray *)getAllPois{
  NSMutableArray *array = [NSMutableArray array];
  ...
  return array;
}
```

the `array` method returns an autorelease object, which can be returned

Temporary object ownership

- Many classes provide both
  - initializers to be used to get an object you own
  - methods that return an `autoreleased` object, which you do not own
- Common examples of methods returning `autoreleased` objects:

```
NSString *str = [NSString stringWithFormat:@"item %d",i];

NSArray *array = [NSArray arrayWithObjects:@"one", @"two", @"three", nil];

NSDictionary *dict = [NSDictionary dictionaryWithObjectsAndKeys:
                        @"key1", @"value1",
                        @"key2", @"value2", nil];
```

Deallocating memory

- When the reference count of an object reaches 0, its **dealloc** method is invoked
- **dealloc** is inherited from **NSObject**, so if you do not implement it the default method is used
- <u>**dealloc** must never be called directly</u>, as it is automatically called when the reference count reaches 0
- **dealloc** must be overridden if your class has instance variables of a non-primitive type
- The last instruction of **dealloc** must be a call to the superclass implementation of **dealloc** (only time you are authorized to call **dealloc** directly)

```
- (void) dealloc{
  [_name release];
  [super dealloc];
}
```

Memory management with properties

- What about ownership of objects returned by getters or passed in to setters?
- Typically, getters return an object that is going to live long enough to be retained (similarly to an autoreleased object)
- Setters should retain the object being set (or we would have an object that might be deallocated)
- The problem is: what happens when properties are `@synthesize`d?
- There are three options to tell how the setters should be implemented: `retain`, `copy`, and `assign`

```
@property (retain) NSString *name;

@property (copy) NSString *name;

@property (assign) NSString *name;
```

Properties: retain

```
@property (retain) NSString *name;
```

`retain` means that the instance variable backing the property is retained

```
@synthesize name;
```

```
- (void) setName:(NSString *)name{
  [_name release];
  _name = [name retain];
}
```

Properties: retain

```
@property (retain) NSString *name;
```

```
@synthesize name;
```

```
- (void) setName:(NSString *)name{
  [_name release];
  _name = [name retain];
}
```

1

first, the current object is sent a `release` message, since it is no longer used (ok if it is `nil`!)

Properties: retain

```
@property (retain) NSString *name;
```

```
@synthesize name;
```

→

```
- (void) setName:(NSString *)name{
  [_name release];
  _name = [name retain];
}
```

2

next, the new object is sent a `retain` message, since we are going to use it

Properties: copy

```
@property (copy) NSString *name;
```

copy means that the instance variable backing the property is copied

```
@synthesize name;
```

```
- (void) setName:(NSString *)name{
 [_name release];
 _name = [name copy];
}
```

Properties: assign

```
@property (assign) NSString *name;
```

`assign` means that the instance variable backing the property is not retained

```
@synthesize name;
```

```
- (void) setName:(NSString *)name{
  _name = name;
}
```

Good practice

Do not use accessor methods in initializers!

- Using accessor methods (both by sending a message or by using the property, which are equivalent) is bad because accessor methods might rely on some state of the object, while in the initializer we have no guarantee about the state since it is being initialized

```
- (id) initWithName:(NSString *)name latitude:(double)latitude longitude:(double)longitude{
  if(self = [super init]){
     self.name = name;
     self.latitude = latitude;
     self.longitude = longitude;
  }
  return self;
}
```

Good practice

Do not use accessor methods in initializers!

- Using accessor methods (both by sending a message or by using the property, which are equivalent) is bad because accessor methods might rely on some state of the object, while in the initializer we have no guarantee about the state since it is being initialized

```
- (id) initWithName:(NSString *)name latitude:(double)latitude longitude:(double)longitude{
  if(self = [super init]){
     self.name = name;
     self.latitude = latitude;
     self.longitude = longitude;
  }
  return self;
}
```

Good practice

Do not use accessor methods in initializers!

- Using accessor methods (both by sending a message or by using the property, which are equivalent) is bad because accessor methods might rely on some state of the object, while in the initializer we have no guarantee about the state since it is being initialized

```
- (id) initWithName:(NSString *)name latitude:(double)latitude longitude:(double)longitude{
  if(self = [super init]){
     _name = [name copy];
     _latitude = latitude;
     _longitude = longitude;
  }
  return self;
}
```

Good practice

Do not use accessor methods in `dealloc`!

- If we are in **`dealloc`**, it means that the object has a reference count of 0 and that it is being deallocated, so it is neither safe nor appropriate to send messages

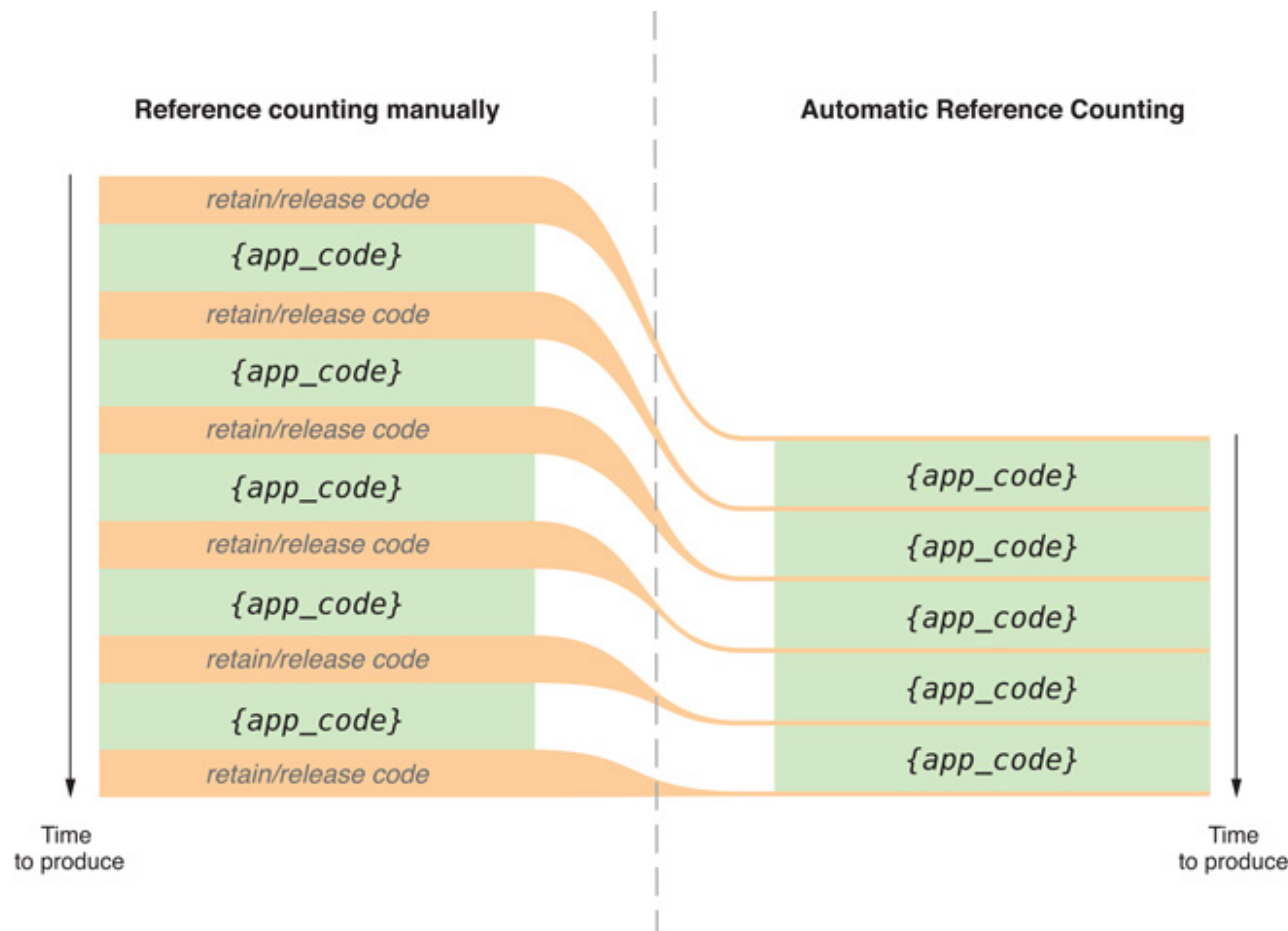
```
- (void) dealloc{
  [_name release];
  [super dealloc];
}
```

Automatic Reference Counting

- Since iOS 4, a new feature has been introduced to simplify life of Objective-C programmers for managing memory allocation: **Automatic Reference Counting (ARC)**
- ARC delegates the responsibility to perform reference counting from the programmer to the compiler
- `retain`, `release`, and `autorelease` must no longer be explicitly invoked
- ARC evaluates the lifetime requirements of objects and automatically inserts appropriate memory management calls for you at compile time
- The compiler also generates appropriate `dealloc` methods; implement a custom `dealloc` if you need to manage resources other than releasing instance variables, but in any case you must not call `[super dealloc]`, since ARC does it automatically
- Corollary: with ARC, the use of `retain`, `release`, `autorelease`, `retainCount`, and `dealloc` is forbidden
- ARC is NOT Garbage Collection, since no process is being executed to cleanup memory

Automatic Reference Counting

- How does code get updated with ARC?



Automatic Reference Counting

- How does code get updated with ARC?

MRC

```
- (NSArray *) getAllPois{
  NSMutableArray *array = [[NSMutableArray alloc] init];
  ...
  [array autorelease];
  return array;
}
```

Automatic Reference Counting

- How does code get updated with ARC?

MRC

```
- (NSArray *) getAllPois{
  NSMutableArray *array = [[NSMutableArray alloc] init];
  ...
  [array autorelease];
  return array;
}
```

`autorelease` is used to defer the release of the object

Automatic Reference Counting

- How does code get updated with ARC?

MRC

```
- (NSArray *) getAllPois{
  NSMutableArray *array = [[NSMutableArray alloc] init];
  ...
  [array autorelease];
  return array;
}
```

ARC

```
- (NSArray *) getAllPois{
  NSMutableArray *array = [[NSMutableArray alloc] init];
  ...
  ...
  return array;
}
```

`autorelease` is used to defer the release of the object

Automatic Reference Counting

- How does code get updated with ARC?

MRC

```
- (NSArray *) getAllPois{
  NSMutableArray *array = [[NSMutableArray alloc] init];
  ...
  [array autorelease];
  return array;
}
```

`autorelease` is used to defer the release of the object

ARC

```
- (NSArray *) getAllPois{
  NSMutableArray *array = [[NSMutableArray alloc] init];
  ...
  ...
  return array;
}
```

the compiler generates the appropriate call for releasing the object

Properties with ARC

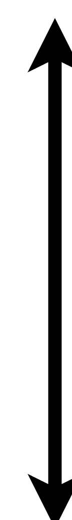
- What about the ownership of objects returned by accessor methods?
- The implementation of the accessor methods depends on the declaration of the properties
- ARC introduces the concept of **weak references**
- A weak reference does not extend the lifetime of the object it points to, and automatically becomes `nil` when there are no strong references to the object
- The keywords `weak` and `strong` are introduced to manage weak and strong references for objects

Properties with ARC: `strong`

- `strong` properties are equivalent to `retain` properties

```
@property (strong) MyClass *myObject;
```

ARC



```
@property (retain) MyClass *myObject;
```

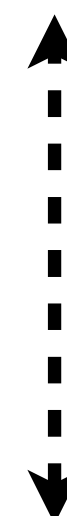
MRC

Properties with ARC: weak

- weak properties are similar to assign properties
- If the MyClass instance is deallocated, the property value is set to nil instead of remaining as a dangling pointer

```
@property (weak) MyClass *myObject;
```

ARC



```
@property (assign) MyClass *myObject;
```

MRC

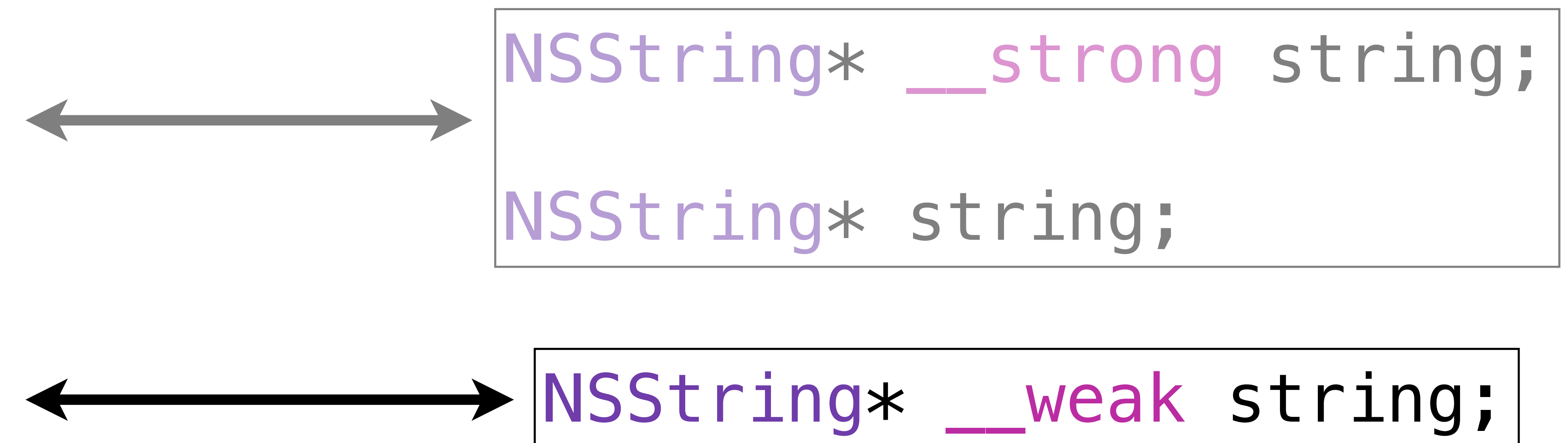
New lifetime qualifiers

- ARC also introduces new lifetime qualifiers for objects:
  - `__strong` (default): the object remains “alive” as long as there is a strong pointer to it

```
NSString* __strong string;

NSString* string;
```

New lifetime qualifiers

- ARC also introduces new lifetime qualifiers for objects:
  - `__strong` (default): the object remains “alive” as long as there is a strong pointer to it
  - `__weak` specifies a reference that does not keep the referenced object alive (a weak reference is set to `nil` when there are no strong references to the object)



New lifetime qualifiers

- ARC also introduces new lifetime qualifiers for objects:
  - `__strong` (default): the object remains “alive” as long as there is a strong pointer to it

```
NSString* __strong string;

NSString* string;
```

  - `__weak` specifies a reference that does not keep the referenced object alive (a weak reference is set to `nil` when there are no strong references to the object)

```
NSString* __weak string;
```

  - `__unsafe_unretained` specifies a reference that does not keep the referenced object alive and is not set to `nil` when there are no strong references to the object (if the object it references is deallocated, the pointer is left dangling, as with `assign` properties)

```
NSString* __unsafe_unretained string;
```

New lifetime qualifiers

- ARC also introduces new lifetime qualifiers for objects:
  - `__strong` (default): the object remains “alive” as long as there is a strong pointer to it

```
NSString* __strong string;

NSString* string;
```

  - `__weak` specifies a reference that does not keep the referenced object alive (a weak reference is set to `nil` when there are no strong references to the object)

```
NSString* __weak string;
```

  - `__unsafe_unretained` specifies a reference that does not keep the referenced object alive and is not set to `nil` when there are no strong references to the object (if the object it references is deallocated, the pointer is left dangling, as with `assign` properties)

```
NSString* __unsafe_unretained string;
```

  - `__autoreleasing` is used to denote arguments that are passed by reference and are `autoreleased` on return

```
NSString* __autoreleasing string;
```

New lifetime qualifiers

- ARC also introduces new lifetime qualifiers for objects:
  - `__strong` (default): the object remains “alive” as long as there is a strong pointer to it

```
NSString* __strong string;
NSString* string;
```

  - `__weak` specifies a reference that does not keep the referenced object alive (a weak reference is set to `nil` when there are no strong references to the object)

```
NSString* __weak string;
```

  - `__unsafe_unretained` specifies a reference that does not keep the referenced object alive and is not set to `nil` when there are no strong references to the object (if the object it references is deallocated, the pointer is left dangling, as with old `assign` properties)

```
NSString* __unsafe_unretained string;
```

  - `__autoreleasing` is used to denote arguments that are passed by reference and are `autoreleased` on return

```
NSString* __autoreleasing string;
```

Protocols

- Objective-C provides a way to define a set of methods similarly to Java interfaces: **protocols**
- Interfaces (.h files) are used to declare the methods and properties associated with a class
- Protocols declare the methods expected to be used for a particular situation
- Sometimes it is not important who is going to perform an action, we just need to know that someone is going to do it (e.g., table view data source)
- **Protocols define messaging contracts**
- A protocol is used to declare methods and properties that are independent of any specific class
- The usage of protocols allows to maximize the reusability of code by minimizing the interdependence among parts of your code

Defining a protocol

- Protocols are defined in the interface file, between the `@protocol` directive and the corresponding `@end`

```
@protocol MyProtocol

// definition of methods and properties
- (void)method1;
- (NSString *)method2;
- (NSArray *)method3:(NSString *)str;

@end
```

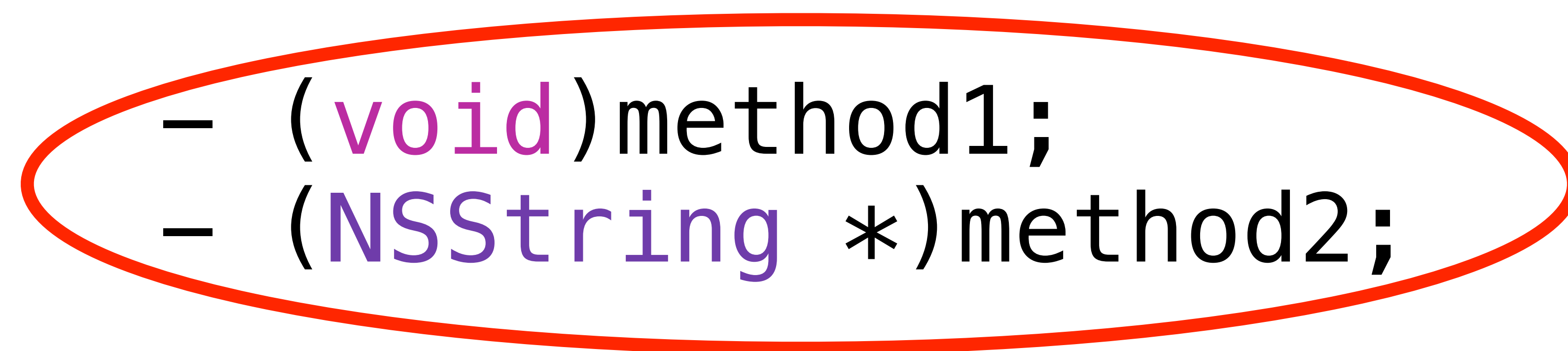
Defining a protocol

- @protocol methods can be declared as @required (default) or @optional

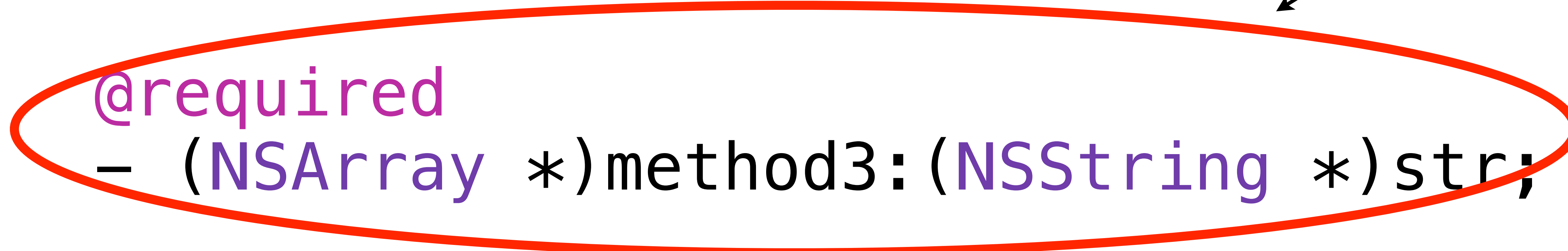
```
@protocol MyProtocol

- (void)method1;
- (NSString *)method2;

@optional
- (void)optionalMethod;

@required
- (NSArray *)method3:(NSString *)str;

@end
```

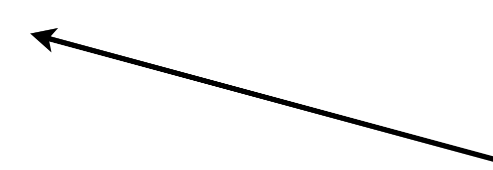
Defining a protocol

- @protocol methods can be declared as @required (default) or @optional

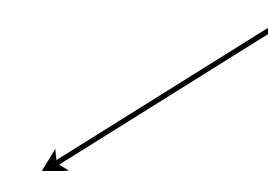


```
@protocol MyProtocol

- (void)method1;
- (NSString *)method2;

@optional
- (void)optionalMethod;

@required
- (NSArray *)method3:(NSString *)str;

@end
```





Required methods



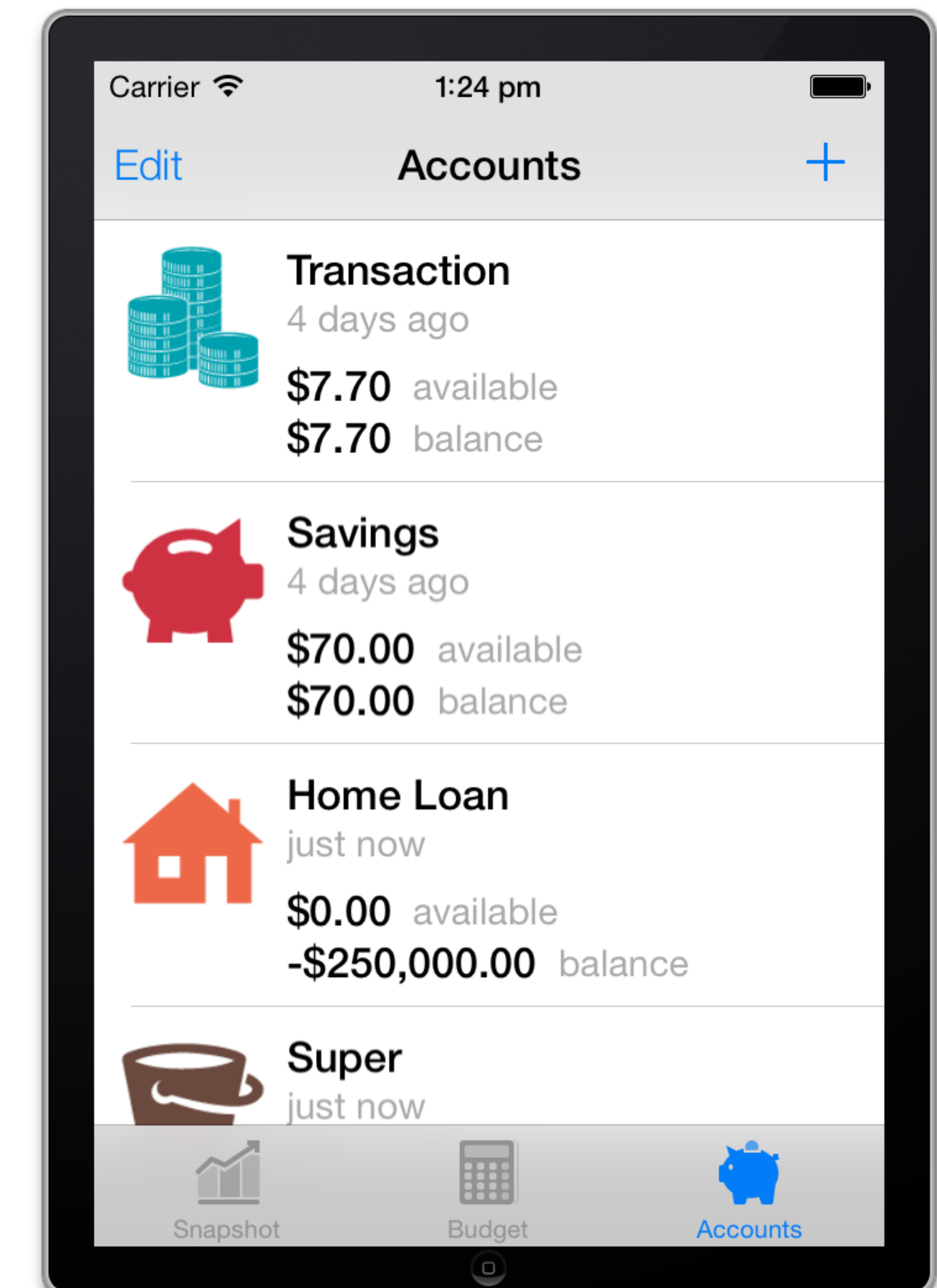
Defining a protocol

- `@protocol` methods can be declared as `@required` (default) or `@optional`

```
@protocol MyProtocol

- (void)method1;
- (NSString *)method2;

@optional
- (void)optionalMethod;

@required
- (NSArray *)method3:(NSString *)str;

@end
```

Conforming to protocols

- Classes that implement the required methods defined in a protocol are said to **conform to the protocol**
- Or the other way: if a class is willing to conform to a protocol, it must implement all the required methods defined in the protocol definition
- By conforming to a protocol, a class is allowed to be used to provide concrete behavior to parts of the program that depend on such behavior but do not depend on a specific implementation of that behavior
- For example:
  - `UITableViewDataSource` is a protocol that defines methods that a `UITableView` depends on
  - `UITableView` implementation (how it is displayed, ...) is independent however from the data it has to display
  - `UITableView` uses a `UITableViewDataSource` to fetch the data it will display
  - a class that conforms to `UITableViewDataSource` will then be passed to the `UITableView` at runtime



Defining a class that conforms to a protocol

- In order for a class to conform to a protocol, it must:

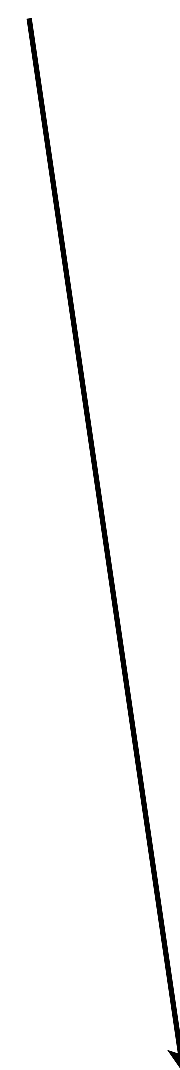
1. declare it between angle brackets

```
@interface MyClass : NSObject<MyProtocol>{

  ...

}
@end
```

Defining a class that conforms to a protocol

- In order for a class to conform to a protocol, it must:

1. declare it between angle brackets



```
@interface MyClass : NSObject<MyProtocol>{
  ...
}
@end
```

Defining a class that conforms to a protocol

- In order for a class to conform to a protocol, it must:

1. declare it between angle brackets
2. implement the required protocol methods

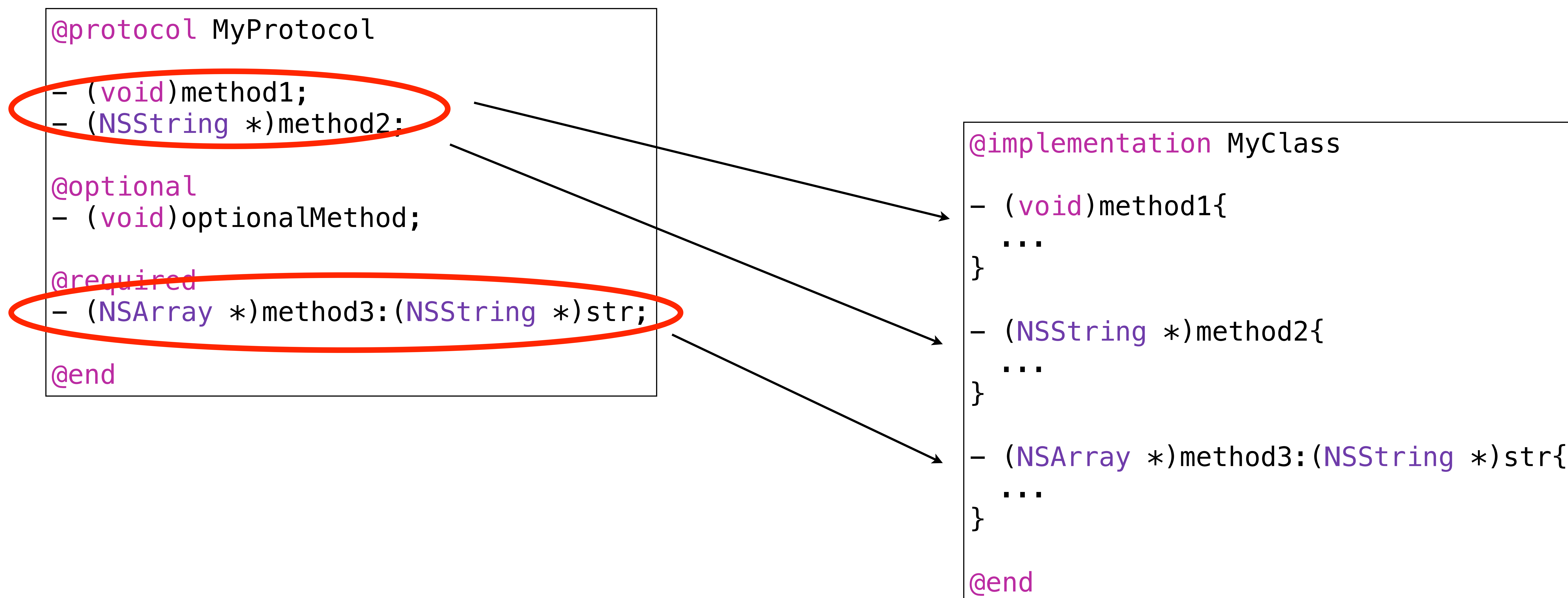
```
@protocol MyProtocol

- (void)method1;
- (NSString *)method2;

@optional
- (void)optionalMethod;

@required
- (NSArray *)method3:(NSString *)str;

@end
```

```
@implementation MyClass

- (void)method1{
  ...
}

- (NSString *)method2{
  ...
}

- (NSArray *)method3:(NSString *)str{
  ...
}

@end
```

Defining a class that conforms to a protocol

- In order for a class to conform to a protocol, it must:

1. declare it between angle brackets
2. implement the required protocol methods



Protocols

- Since some methods may have been declared as optional, it is important to check at runtime whether the target object has implemented a method before sending a message to it

```
if ([target respondsToSelector:@selector(aMethod:)]) {
  ...
}
```

Categories

- Sometimes, you may wish to extend an existing class by adding behavior that is useful only in certain situations
- Subclassing in those situations might be a “hard” approach
- Objective-C provides a solution to manage these situations in a clean and lightweight way: **categories**
- Categories add methods to existing classes, without subclassing
- If you need to add a method to an existing class, perhaps to add functionality to make it easier to do something in your own application, the easiest way is to use a category

Defining a category that enhances a class

- In order to define a category for a class, you must:

1. "re-define" the class with the `@interface` directive
2. specify the name of the category between parentheses
3. declare the new methods
4. provide an implementation for the category in the **.m** file using the `@implementation` directive

- There is a naming convention for the name of the files: `<BaseClass>+<Category>.h` and `<BaseClass>+<Category>.m`

Example: a category that enhances NSString

```
@interface NSString (MyCategory)

- (int) countOccurrences:(char)c;

@end
```

NSString+MyCategory.h

```
@implementation NSString (MyCategory)

- (int) countOccurrences:(char)c{
  ...
}

@end
```

NSString+MyCategory.m

```
#import "NSString+MyCategory.h"

...

int occurA = [self.name countOccurrences:'a'];

...
```

usage

A trick with categories: private methods and properties

- It is nice to have utility methods that you use in your class “privately” and that you do not want to expose
- We have seen that the @private modifier can be used in conjunction with attributes in order to restrict the visibility of the variables
- Unfortunately, this does not apply with methods
- <u>A little workaround to get private methods in your class is to define an anonymous category in your implementation file and declare your private methods there</u>
- Since the declaration is inside the implementation file, it will be hidden to other classes

A trick with categories: private methods and properties

```
@interface MDPoi ()

@property (strong) NSString *name;
- (void) privateMethod;

@end

@implementation MDPoi

@synthesize name = _name;

- (void) privateMethod{
  ...
}

@end
```

MDPoi.m

Mobile Application Development

Lecture 13
Introduction to Objective-C

Part II