Objective-C Protocols, Categories, Error Handling, and Parsing

2501ICT/7421ICTNathan

René Hexel

School of Information and Communication Technology
Griffith University

Semester 1, 2012

Outline

Review

- Lists and Arrays as part of the Standard API
  - → `NSArray` and `NSMutableArray` in Objective-C
  - → `vector` and `list` in C++
- Objective-C allows run-time reflection of Collection objects
  - → allows objects of multiple classes within a single collection
  - → container classes for primitive types
  - → `NSNumber`, `NSValue`, `NSNull`, and `NSData`
- C++ requires compile time templates
  - → allows only one type of element per collection
  - → e.g. `vector<int>`, `list<string>`, etc.
- C++ Namespaces and Operator Overloading
  - → save typing
  - → can make source code more readable
  - → need to be used with care!

Objective-C Protocols

Protocols

- Protocols work like interfaces in Java
    - they specify a number of methods a class must implement

Example (protocol example)

```
@protocol Printing
- (void) print;          // conforming classes must have a 'print' method
@end


@interface MyClass:  NSObject <Printing>         // MyClass conforms to Printing
{
        int a, b;
}
- init;
- setA: (int) newA  b:  (int) newB;
// - (void) print;                            // must exist, but not in interface!
@end
```

Example for using Protocols

Example (NSCopying and Printing protocols)

```
/*
* statically indicate that an object conforms to a protocol
*/
id<Printing> aPrintingObject = [obj someMethod];

id<Printing, NSCopying> other = [obj someOtherMethod];

[aPrintingObject print];                 // we know this conforms to Printing

aPrintingObject = [other copy];          // 'other' conforms to NSCopying as well

/*
* we can also test conformance dynamically via conformsToProtocol:
*/
id obj = other;

if ([obj conformsToProtocol:  @protocol(Printing)])
        [obj print];                     // only invoke print if obj conforms
```

Introspection

Introspection

Checking for individual Methods

- Objective-C allows to check for individual Methods
  - does not require a full protocol
  - useful if only one method needs to be checked dynamically

Example (`-respondsToSelector:` example)

```
id obj = [anArray objectAtIndex:  5];     // whatever object is found in the array

/*
* check if "obj" has a "print" method before invoking it:
*/
if ([obj respondsToSelector:  @selector(print)])
        [obj print];                          // only invoke print if method exists
```

Determining an Object's type

- Objective-C also allows to check which class an object belongs to
  - → `isMemberOfClass:` tests for a specific class only
  - → `isKindOfClass:` tests for a class or any of its subclasses

Example (dynamically determining class membership)

```
id obj = [anArray objectAtIndex: 6];     // whatever object is found in the array

/*
* check if "obj" is a mutable string
*/
if ([obj isMemberOfClass: [NSMutableString class]])
       [obj appendString: @","];         // append a comma

/*
* check if "obj" is any kind of string (including NSMutableString) or number
*/
if ([obj isKindOfClass: [NSString class]])
       printf("%s", [obj UTF8String]);          // print as a string
else if ([obj isKindOfClass: [NSNumber class]])
       printf("%lg", [obj doubleValue]);        // print as a double
```

Categories

Using and Extending Classes

- When should a class be subclassed?
  - if you just want to use a class, make it a member variable of your class
    - → a `Zoo` class should just have `Animal` members
  - → for more specific concepts, use a subclass
    - → a `Cat` class should be derived from an `Animal` class
- Objective-C offers a third option: Categories
  - a category allows you to add methods to an existing class
    - → these methods become available immediately to any code using the existing class!
  - useful if you believe a method is missing from a class!

Category Example: extending NSArray

Example (a `firstObject` method for NSArray)

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

@interface NSArray (AddFirstObject)                  // a category for NSArray
- firstObject;                                       // adds a firstObject method
@end

@implementation NSArray (AddFirstObject)             // category implementation
- firstObject                                        // firstObject implementation
{
        return [self objectAtIndex:  0];              // get first object
}
@end

int main(int argc, char *argv[])
{
        NSAutoreleasePool *pool = [NSAutoreleasePool new];
        NSArray *list = [NSArray arrayWithObjects:  @"one", @"two", nil];

        printf("%s", [[list firstObject] UTF8String]);  // print first object

        [pool release];

        return EXIT_SUCCESS;
}
```

Error Handling

Objective-C Error Handling

- Most error handling in Objective-C is in-band
  - → return value indicates failure
    - e.g. `nil` instead of a returned object, a boolean set to `NO`, an int set to `-1`, etc.
    - needs to be documented in the API
  - → requires explicit error checking
    - e.g. `if (object == nil)` …statements
- In Objective-C, messages can be sent to `nil` objects
  - method invocations on `nil` are safe!
    - → allows collating error handing
    - → better to use access methods than accessing member variables directly!

Objective-C Error Handling Example

Example (printf() may crash if there is no error handling)

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

int main(int argc, char *argv[])
{
  NSAutoreleasePool *pool = [NSAutoreleasePool new];
  NSArray *args = [[NSProcessInfo processInfo] arguments];      // cmd line args
  NSEnumerator *iterator = [args objectEnumerator];              // go through args
  NSString *arg, *s = @"Arguments are:";                         // some string
  int status = EXIT_SUCCESS;

  while (arg = [iterator nextObject])                            // next argument
          s = [s stringByAppendingFormat:  @" %@", arg];          // append arg

  if (s == nil)          // error handling can be deferred until the very end
  {
          NSLog(@"it seems this program has run out of memory");
          status = EXIT_FAILURE;
  }
  else printf("%s\n", [s UTF8String]);                           // print args

  [pool release];

  return status;
}
```

Exception Handling

- Some errors are “out of band”
  - → a network connection that closes unexpectedly
  - → a file reading error
  - → accessing elements outside of array boundaries
  - etc.
- `NS_DURING`
  - starts an exception handling domain
    - → like `try` in Java
    - → exceptions that occur will be caught
- `NS_HANDLER`
  - the actual exception handler
  - catches exceptions that occur in the handling domain
    - → like `catch` in Java
  - → `localException` refers to the exception that was thrown
- `NS_ENDHANDLER`
  - follows both normal and abnormal termination

Objective-C Exception Handling Example

Example (NSRangeException)

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

int main(int argc, char *argv[])
{
  NSAutoreleasePool *pool = [NSAutoreleasePool new];
  NSArray *array = [NSArray array];                            // an empty array

  NS_DURING
  {
    id object = [array objectAtIndex: 0];                       // will this work?
    printf("%s", [object UTF8String]);                         // never reached
  }
  NS_HANDLER
  {
    printf("%s: %s", [[localException name] UTF8String],        // print exception
                     [[localException reason] UTF8String]);    // and reason
  }
  NS_ENDHANDLER

  printf(", count = %d\n", [array count]);

  [pool release];

  return EXIT_SUCCESS;
}
```

Throwing Exceptions

- `NSException` class supports throwing exceptions
  - → can be subclassed, but often unnecessary
  - → each exception contains a name and a reason
- `raise...` methods
  - → `-raise` raises an object of type `NSException`
  - → `+raise:format:,...` creates and raises an exception in one go
- `-name` method
  - returns the name of an exception (an `NSString`)
- `-reason` method
  - returns the reason for an exception (also an `NSString`)
    - → should be a human readable reason

Objective-C Exception Throwing Example

Example (prints: `MyException:  reason 42`)

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

void some_function(void)
{
  [NSException raise:  @"MyException"              // raise 'MyException'
              format:  @"reason %d", 42];          // a not very readable reason!
}

int main(int argc, char *argv[])
{
  NSAutoreleasePool *pool = [NSAutoreleasePool new];

  NS_DURING
    some_function();                               // call some function

  NS_HANDLER
    printf("%s:  %s\n", [[localException name] UTF8String],
                         [[localException reason] UTF8String]);
  NS_ENDHANDLER

  [pool release];

  return EXIT_SUCCESS;
}
```

Object-Oriented Input Parsing

Input Parsing Introduction

- We have seen how to print output in a formatted way
  - → `printf()` and `sprintf()` in C
  - → `NSLog()` and `+stringWithFormat:` in Objective-C
  - → `std::cout` in C++
- Parsing formatted input in C
  - → `scanf()` and `sscanf()`

→ How can formatted input be parsed in Objective-C and C++?

Objective-C File and Standard Input

- `[NSString stringWithContentsOfFile: filename]`
  - reads the whole content of a file into a string
- `NSFileHandle`
  - more fine grained handling of files
    - → `+fileHandleWithStandardInput`
      handle for reading from `stdin`
    - → `+fileHandleForReadingAtPath: filename`
      read from the given file
    - → `-availableData`
      read all the available data from a file handle
      (e.g. one line of user input followed by return)
    - → `-readDataOfLength: length`
      read a given number of bytes
    - → `-readDataToEndOfFile`
      read all file data
    - → `-offsetInFile`
      return the current position within the file

NSFileHandle Example

Example (using NSFileHandle)

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

int main(int argc, char *argv[])
{
  NSAutoreleasePool *pool = [NSAutoreleasePool new];
  NSFileHandle *in = [NSFileHandle fileHandleWithStandardInput];// stdin

  printf("Enter your input:  ");                                 // prompt user
  fflush(stdout);                                               // flush output

  NSData *data = [in availableData];                            // read user input
  NSString *string = [[NSString alloc] initWithData:  data       // convert to string
                                           encoding:  NSUTF8StringEncoding];
  printf("You entered:  %s\n", [string UTF8String]);             // print user input

  [string release];     // don't forget proper memory management
  [pool release];

  return EXIT_SUCCESS;
}
```

Parsing Input

- `NSScanner`
  - allows parsing input

→ `+scannerWithString:` create a scanner

→ `-scanInt:` *`intPointer`* scan an integer

→ `-scanDouble:` *`doublePointer`* scan a double

→ `-scanUpToString:` *`str`* `intoString:` *`ptr`*
scan everything up to a given string into a new string

→ `-scanString:` *`str`* `intoString:` *`ptr`*
skips a given string (*ptr* can be `NULL`)

→ `-scanCharactersFromSet:` *`s`* `intoString:` *`p`*
scans characters from set *s* into string *p*

→ `-scanUpToCharactersFromSet:` *`s`* `intoString:` *`p`*
scans into string *p* until a char from set *s* is found

Character Sets

- `NSCharacterSet`
  - class for handling character sets

  → `[NSCharacterSet whitespaceCharacterSet]`
  white space characters (space, tab, . . .)

  → `[NSCharacterSet whitespaceAndNewlineCharacterSet]`
  combination of white space and new line characters

  → `[NSCharacterSet letterCharacterSet]`
  `A` to `Z`, `a` to `z`, `Ä`, `é`, `ö`, . . .

  → `[NSCharacterSet lowercaseLetterCharacterSet]` `a` to `z`, `ä`, `é`, `ü`, . . .

  → `[NSCharacterSet decimalDigitCharacterSet]`
  `0` to `9`

  → `[NSCharacterSet alphanumericCharacterSet]`
  combination of `letterCharacterSet` and `decimalDigitCharacterSet`

Objective-C Parsing Example

Example (prints: `Einstein, Albert was born in 1879`)

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

int main(int argc, char *argv[])
{
  NSAutoreleasePool *pool = [NSAutoreleasePool new];
  NSString *input = @"Albert Einstein, 1879";                  // some input
  NSString *firstName, *lastName;                              // parsing variables
  int yearOfBirth;

  NSScanner *scanner = [NSScanner scannerWithString: input];    // create scanner
  NSCharacterSet *space = [NSCharacterSet whitespaceCharacterSet]; // white space

  if ([scanner scanUpToCharactersFromSet: space intoString: &firstName] &&
      [scanner scanUpToString: @","             intoString: &lastName]  &&
      [scanner scanString: @","                 intoString: NULL]       &&
      [scanner scanInt: &yearOfBirth])
  {
    printf("%s, %s was born in %d\n", [lastName  UTF8String],
                                      [firstName UTF8String],
                                      yearOfBirth);
  }
  else NSLog(@"Cannot scan input '%@': invalid format", input);

  [pool release];
  return EXIT_SUCCESS;
}
```