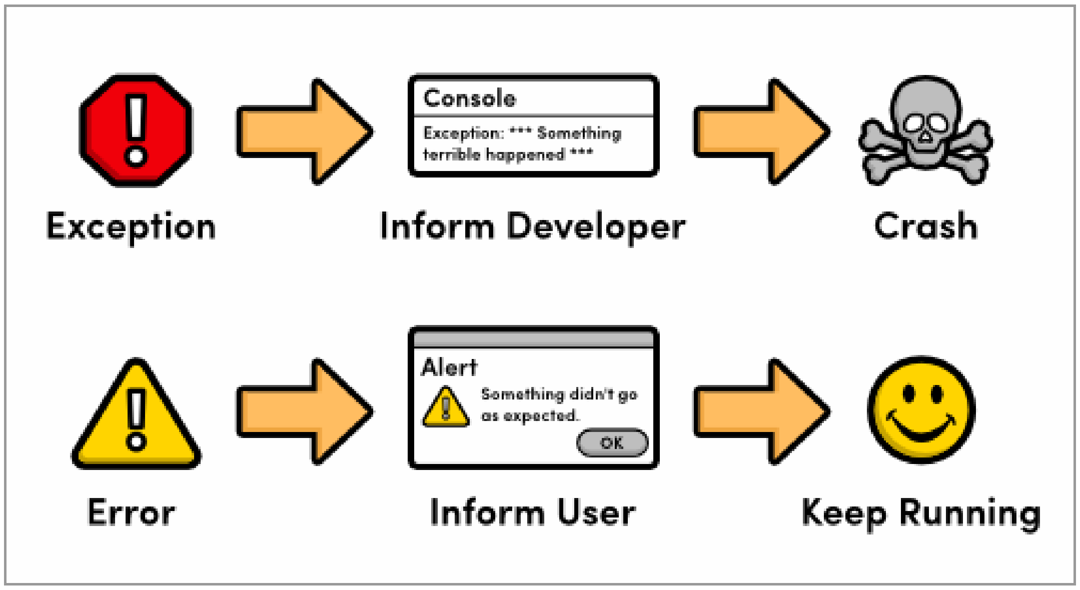
Exceptions & Errors

Two distinct types of problems can arise while an iOS or OS X application is running. **Exceptions** represent programmer-level bugs like trying to access an array element that doesn't exist. They are designed to inform the developer that an *unexpected* condition occurred. Since they usually result in the program crashing, exceptions should rarely occur in your production code.

On the other hand, **errors** are user-level issues like trying load a file that doesn't exist. Because errors are *expected* during the normal execution of a program, you should manually check for these kinds of conditions and inform the user when they occur. Most of the time, they should not cause your application to crash.


Exception
Console
Exception: *** Something terrible happened ***
Inform Developer
Crash
Error
Alert
Something didn't go as expected.
OK
Inform User
Keep Running

Exceptions vs. errors

This module provides a thorough introduction to exceptions and errors. Conceptually, working with exceptions is very similar to working with errors. First you need to detect the problem, and then you need to handle it. But, as we're about to find out, the underlying mechanics are slightly different

Exceptions

Exceptions are represented by the `NSException` class. It's designed to be a universal way to encapsulate exception data, so you should rarely need to subclass it or otherwise define a custom exception object. `NSException`'s three main properties are listed below.

Property	Description
`name`	An `NSString` that uniquely identifies the exception.
`reason`	An `NSString` that contains a human-readable description of the exception.
`userInfo`	An `NSDictionary` whose key-value pairs contain extra information about the exception. This varies based on the type of exception.

It's important to understand that exceptions are only used for serious programming errors. The idea is to let you know that something has gone wrong early in the development cycle, after which you're expected to fix it so it never occurs again. If you're trying to handle a problem that's *supposed* to occur, you should be using an error object, not an exception.

Handling Exceptions

Exceptions can be handled using the standard try-catch-finally pattern found in most other high-level programming languages. First, you need to place any code that *might* result in an exception in an `@try` block. Then, if an exception is thrown, the corresponding `@catch()` block is executed to handle the problem. The `@finally` block is called afterwards, regardless of whether or not an exception occurred.

The following `main.m` file raises an exception by trying to access an array element that doesn't exist. In the `@catch()` block, we simply display the exception details. The `NSException *theException` in the parentheses defines the name of the variable containing the exception object.

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

```
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        NSArray *inventory = @[@"Honda Civic",
                               @"Nissan Versa",
                               @"Ford F-150"];
        int selectedIndex = 3;
        @try {
            NSString *car = inventory[selectedIndex];
            NSLog(@"The selected car is: %@", car);
        } @catch(NSException *theException) {
            NSLog(@"An exception occurred: %@", theException.name);
            NSLog(@"Here are some details: %@", theException.reason);
        } @finally {
            NSLog(@"Executing finally block");
        }
    }
    return 0;
}
```

In the real world, you'll want your `@catch()` block to actually handle the exception by logging the problem, correcting it, or possibly even promoting the exception to an error object so it can be displayed to the user. The default behavior for uncaught exceptions is to output a message to the console and exit the program.

Objective-C's exception-handling capabilities are not the most efficient, so you should only use `@try/@catch()` blocks to test for truly exceptional circumstances. Do *not* use it in place of ordinary control flow tools. Instead, check for predictable conditions using standard `if` statements.

This means that the above snippet is actually a very poor use of exceptions. A much better route would have been to make sure that the `selectedIndex` was smaller than the `[inventory count]` using a traditional comparison:

```
if (selectedIndex < [inventory count]) {
    NSString *car = inventory[selectedIndex];
    NSLog(@"The selected car is: %@", car);
} else {
    // Handle the error
}
```

Built-In Exceptions

The standard iOS and OS X frameworks define several built-in exceptions. The complete list can be found here, but the most common ones are described below.

Exception Name	Description
`NSRangeException`	Occurs when you try to access an element that's outside the bounds of a collection.
`NSInvalidArgumentException`	Occurs when you pass an invalid argument to a method.
`NSInternalInconsistencyException`	Occurs when an unexpected condition arises internally.
`NSGenericException`	Occurs when you don't know what else to call the exception.

Note that these values are *strings*, not `NSException` subclasses. So, when you're looking for a specific type of exception, you need to check the `name` property, like so:

```
...
} @catch(NSException *theException) {
    if (theException.name == NSRangeException) {
        NSLog(@"Caught an NSRangeException");
    } else {
        NSLog(@"Ignored a %@ exception", theException);
        @throw;
    }
} ...
```

In an `@catch()` block, the `@throw` directive re-raises the caught exception. We used this in the above snippet to ignore all of the exceptions we didn't want by throwing it up to the next highest `@try` block. But again, a simple `if`-statement would be preferred.

Custom Exceptions

You can also use `@throw` to raise `NSException` objects that contain custom data. The easiest way to create an `NSException` instance is through the `exceptionWithName:reason:userInfo:` factory method. The following example throws an exception inside of a top-level function and catches it in the `main()` function.

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

NSString *getRandomCarFromInventory(NSArray *inventory) {
    int maximum = (int)[inventory count];
    if (maximum == 0) {
        NSException *e = [NSException
                          exceptionWithName:@"EmptyInventoryException"
                          reason:@"*** The inventory has no cars!"
                          userInfo:nil];
        @throw e;
    }
    int randomIndex = arc4random_uniform(maximum);
    return inventory[randomIndex];
}

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        @try {
            NSString *car = getRandomCarFromInventory(@[]);
            NSLog(@"The selected car is: %@", car);
        } @catch(NSException *theException) {
            if (theException.name == @"EmptyInventoryException") {
                NSLog(@"Caught an EmptyInventoryException");
            } else {
                NSLog(@"Ignored a %@ exception", theException);
                @throw;
            }
        }
    }
    return 0;
}
```

While occasionally necessary, you shouldn't really need to throw custom exceptions like this in normal applications. For one, exceptions represent programmer errors, and there are very few times when you should be planning for serious coding mistakes. Second, `@throw` is an expensive operation, so it's always better to use errors if possible.

Errors

Since errors represent *expected* problems, and there are several types of operations that can fail without causing the program to crash, they are much more common than exceptions. Unlike exceptions, this kind of error checking is a normal aspect of production-quality code.

The `NSError` class encapsulates the details surrounding a failed operation. It's main properties are similar to `NSException`.

Property	**Description**
`domain`	An `NSString` containing the error's domain. This is used to organize errors into a hierarchy and ensure that error codes don't conflict.
`code`	An `NSInteger` representing the ID of the error. Each error in the same domain must have a unique value.
`userInfo`	An `NSDictionary` whose key-value pairs contain extra information about the error. This varies based on the type of error.

The `userInfo` dictionary for `NSError` objects typically contains more information than `NSException`'s version. Some of the pre-defined keys, which are defined as named constants, are listed below.

Key	**Value**
`NSLocalizedDescriptionKey`	An `NSString` representing the full description of the error. This usually includes the failure reason, too.
`NSLocalizedFailureReasonErrorKey`	A brief `NSString` isolating the reason for the failure.

`NSUnderlyingErrorKey`	A reference to another `NSError` object that represents the error in the next-highest domain.

Depending on the error, this dictionary will also contain other domain-specific information. For example, file-loading errors will have a key called `NSFilePathErrorKey` that contains the path to the requested file.

Note that the `localizedDescription` and `localizedFailureReason` methods are an alternative way to access the first two keys, respectively. These are used in the next section's example.

Handling Errors

Errors don't require any dedicated language constructs like `@try` and `@catch()`. Instead, functions or methods that *may* fail accept an additional argument (typically called `error`) that is a reference to an `NSError` object. If the operation fails, it returns `NO` or `nil` to indicate failure and populates this argument with the error details. If it succeeds, it simply returns the requested value as normal.

Many methods are configured to accept an **indirect reference** to an `NSError` object. An indirect reference is a pointer to a pointer, and it allows the method to point the argument to a brand new `NSError` instance. You can determine if a method's `error` argument accepts a indirect reference by its double-pointer notation: `(NSError **)error`.

The following snippet demonstrates this error-handling pattern by trying to load a file that doesn't exist via `NSString`'s `stringWithContentsOfFile:encoding:error:` method. When the file loads successfully, the method returns the contents of the file as an `NSString`, but when it fails, it directly returns `nil` and *indirectly* returns the error by populating the `error` argument with a new `NSError` object.

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

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        NSString *fileToLoad = @"/path/to/non-existent-file.txt";
        NSError *error;
        NSString *content = [NSString stringWithContentsOfFile:fileToLoad
                                                      encoding:NSUTF8StringEncoding
                                                         error:&error];

        if (content == nil) {
            // Method failed
            NSLog(@"Error loading file %@!", fileToLoad);
            NSLog(@"Domain: %@", error.domain);
            NSLog(@"Error Code: %ld", error.code);
            NSLog(@"Description: %@", [error localizedDescription]);
            NSLog(@"Reason: %@", [error localizedFailureReason]);
        } else {
            // Method succeeded
            NSLog(@"Content loaded!");
            NSLog(@"%@", content);
        }
    }
    return 0;
}
```

Notice how the we had to pass the `error` variable to the method using the reference operator. This is because the `error` argument accepts a double-pointer. Also notice how we checked the return value of the method for success with an ordinary `if` statement. You should only try to access the `NSError` reference if the method directly returns `nil`, and you should never use the presence of an `NSError` object to indicate success or failure.

Of course, if you only care about the success of the operation and aren't concern with *why* it failed, you can just pass `NULL` for the `error` argument and it will be ignored.

Built-In Errors

Like `NSException`, `NSError` is designed to be a universal object for representing errors. Instead of subclassing it, the various iOS and OS X frameworks define their own constants for the `domain` and `code` fields. There are several built-in error domains, but the main four are as follows:

```
NSMachErrorDomain
NSPOSIXErrorDomain
NSOSStatusErrorDomain
NSCocoaErrorDomain
```

Most of the errors you'll be working with are in the `NSCocoaErrorDomain`, but if you drill down to the lower-level domains, you'll see some of the other ones. For example, if you add the following line to `main.m`, you'll find an error with `NSPOSIXErrorDomain` for its domain.

```
NSLog(@"Underlying Error: %@", error.userInfo[NSUnderlyingErrorKey]);
```

For most applications, you shouldn't need to do this, but it can come in handy when you need to get at the root cause of an error.

After you've determined your error domain, you can check for a specific error code. The Foundation Constants Reference describes several `enum`'s that define most of the error codes in the `NSCocoaErrorDomain`. For example, the following code searches for a `NSFileReadNoSuchFileError` error.

```
...
if (content == nil) {
    if ([error.domain isEqualToString:@"NSCocoaErrorDomain"] &&
        error.code == NSFileReadNoSuchFileError) {
        NSLog(@"That file doesn't exist!");
        NSLog(@"Path: %@", [[error userInfo] objectForKey:NSFilePathErrorKey]);
    } else {
        NSLog(@"Some other kind of read occurred");
    }
} ...
```

Other frameworks should include any custom domains and error codes in their documentation.

Custom Errors

If you're working on a large project, you'll probably have at least a few functions or methods that can result in an error. This section explains how to configure them to use the canonical error-handling pattern discussed above.

As a best practice, you should define all of your errors in a dedicated header. For instance, a file called `InventoryErrors.h` might define a domain containing various error codes related to fetching items from an inventory.

```
// InventoryErrors.h

NSString *InventoryErrorDomain = @"com.RyPress.Inventory.ErrorDomain";

enum {
    InventoryNotLoadedError,
    InventoryEmptyError,
    InventoryInternalError
};
```

Technically, custom error domains can be anything you want, but the recommended form is `com.<Company>.<Framework-or-project>.ErrorDomain`, as shown in `InventoryErrorDomain`. The `enum` defines the error code constants.

The only thing that's different about a function or method that is error-enabled is the additional `error` argument. It should specify `NSError **` as its type, as shown in the following iteration of `getRandomCarFromInventory()`. When an error occurs, you point this argument to a new `NSError` object. Also note how we defined `localizedDescription` by manually adding it to the `userInfo` dictionary with `NSLocalizedDescriptionKey`.

```
// main.m
#import <Foundation/Foundation.h>
#import "InventoryErrors.h"

NSString *getRandomCarFromInventory(NSArray *inventory, NSError **error) {
    int maximum = (int)[inventory count];
    if (maximum == 0) {
        if (error != NULL) {
            NSDictionary *userInfo = @{NSLocalizedDescriptionKey: @"Could not"
            " select a car because there are no cars in the inventory."};

            *error = [NSError errorWithDomain:InventoryErrorDomain
                                         code:InventoryEmptyError
                                     userInfo:userInfo];
        }
        return nil;
    }
    int randomIndex = arc4random_uniform(maximum);
    return inventory[randomIndex];
}

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        NSArray *inventory = @[];
        NSError *error;
        NSString *car = getRandomCarFromInventory(inventory, &error);

        if (car == nil) {
            // Failed
            NSLog(@"Car could not be selected");
            NSLog(@"Domain: %@", error.domain);
            NSLog(@"Error Code: %ld", error.code);
            NSLog(@"Description: %@", [error localizedDescription]);

        } else {
            // Succeeded
            NSLog(@"Car selected!");
            NSLog(@"%@", car);
        }
    }
    return 0;
}
```

Since it techinically is an error and not an exception, this version of `getRandomCarFromInventory()` is the “proper” way to handle it (opposed to Custom Exceptions).

Summary

Errors represent a failed operation in an iOS or OS X application. It's a standardized way to record the relevant information at the point of detection and pass it off to the handling code. Exceptions are similar, but are designed as more of a development aid. They generally should not be used in your production-ready programs.

How you handle an error or exception is largely dependent on the type of problem, as well as your application. But, most of the time you'll want to inform the user with something like `UIAlertView` (iOS). or `NSAlert` (OS X). After that, you'll probably want to figure out what went wrong by inspecting the `NSError` or `NSException` object so that you can try to recover from it.

The next module explores some of the more conceptual aspects of the Objective-C runtime. We'll learn about how the memory behind our objects is managed by experimenting with the (now obsolete) Manual Retain Release system, as well as the practical implications of the newer Automatic Reference Counting system.

Continue to *Memory Management* ›