Basic GNUstep Base Library Classes

Nicola Pero n.pero@mi.flashnet.it

June 2000 AD

1 NSString

1.1 What is it

Instances of the `NSString` class represent strings composed of unicode characters. Unicode characters are described by a 16-bit integer; this type is called `unichar` in GNUstep, but you will very rarely (if ever) need to work with it directly. Unicode is a standard which supports the character sets of nearly all human languages.

1.2 Static Instances

The easiest way to create a `NSString` is by creating a 'static' instance of it using the `@"..."` construct. For example,

```
NSString *name = @"Nicola";
```

`name` will point then to a 'static' instance of `NSString` containing a unicode representation of the ASCII string `Nicola`. A 'static' instance basically means an instance which is allocated at compile time, whose instance variables are fixed, and which can never be released.

1.3 stringWithFormat

The other main way of creating `NSString`s is by means of the class method `+stringWithFormat:`. This is a method of the class `NSString`; it accepts a list of arguments which are processed in a way very similar to how the `printf` function of the standard C library processes its arguments; the difference is that the result of the method is not sent to the standard output, but rather it is put in a `NSString` which is then returned as return value of the method.

Here is an example:

```
int age = 25;
NSString *message;

message = [NSString stringWithFormat: @"Your age is %d", age];
```

`message` will be a `NSString` containing "Your age is 25".

A special feature of `stringWithFormat` is that it recognises the `%@` conversion specification. You can use this to specify another `NSString` (in the same way as you would use `%s` to specify another C string):

```
NSString *first;
NSString *second;

first = @"Nicola";
second = [NSString stringWithFormat: @"My name is %@", first];
```

this code will cause the `second` variable to be set to the string `My name is Nicola`.

More generally, you can use the `%@` specification to output a description of an object (as returned by the `NSObject`'s `-description`). This is often useful in debugging, as in:

```
NSObject *obj = [anObject someMethod];

NSLog (@"The method returned: %@", obj);
```

1.4 Converting to and from C strings

It is often useful to be able to create a `NSString` from a standard ASCII C string (not fixed at compile time). Say for example that our program needs to call a C library function

```
char *function (void);
```

which returns some useful information in a C string. We want to create a `NSString` using the contents of the C string. The simplest way to do it is by using the `NSString`'s class method `+stringWithCString:`, as follows:

```
char *result;
NSString *string;

result = function ();
string = [NSString stringWithCString: result];
```

Sometimes we need to do the reverse, i.e. to convert a `NSString` to a standard C ASCII string. We can do it using the `-cString` method of the `NSString` class, as in the following example:

```
char *result;
NSString *string;

string = @"Hello";
result = [string cString];
```

1.5 NSMutableString

`NSString`s are immutable objects, that is, once you create an `NSString`, you can not modify it. This allows the GNUstep libraries to optimise the `NSString` code. If you need to be able to modify a string, you should use a special subclass of `NSString`, called `NSMutableString`. Since `NSMutableString` is a subclass of `NSString`, you can use a `NSMutableString` wherever a `NSString` could be used. But, a `NSMutableString` responds to methods which allow you to modify the string directly, a thing you can't do with a generic `NSString`.

To create a NSMutableString, you can use +stringWithFormat:, as in the following example:

```
NSString *name = @"Nicola";
NSMutableString *str;

str = [NSMutableString stringWithFormat: @"Hello, %@", name];
```

While NSString's implementation of +stringWithFormat: returns a NSString, NSMutableString's implementation returns a NSMutableString. Static strings created with the @"..." construct are always immutable.

In practice, NSMutableStrings are not used very often, because usually if you want to modify a string you just create a new string derived from the one you already have.

The most interesting method of the NSMutableString class is possibly the method -appendString:. It takes as argument a NSString, and appends it to the receiver.

For example, the following code:

```
NSString *name = @"Nicola";
NSString *greeting = @"Hello";
NSMutableString *s;

s = AUTORELEASE ([NSMutableString new]);
[s appendString: greeting];
[s appendString: @", "];
[s appendString: name];
```

(where we used new to create a new empty NSMutableString) produces the same result as the following one:

```
NSString *name = @"Nicola";
NSString *greeting = @"Hello";
NSMutableString *s;

s = [NSMutableString stringWithFormat: @"%@, %@",
                                        greeting, name];
```

1.6 Reading and Saving Strings to/from Files

We don't have the time to describe all the string-related features of the GNUstep base library; but it's useful to have at least a quick look at how easy is writing/reading strings to/from files.

If you need to write the contents of a string into a file, you can use the method -writeToFile:atomically:, as shown in the following example:

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

int
main (void)
{
  NSAutoreleasePool *pool = [NSAutoreleasePool new];
```

```
  NSString *name = @"This string was created by GNUstep";

  if ([name writeToFile: @"/home/nico/testing" atomically: YES])
    {
      NSLog (@"Success");
    }
  else
    {
      NSLog (@"Failure");
    }
  return 0;
}
```

`writeToFile:atomically` returns `YES` upon success, and `NO` upon failure. If the atomically flag is `YES`, then the library first writes the string into a file with a temporary name, and, when the writing has been successfully done, renames the file to the specified filename. This prevents erasing the previous version of filename unless writing has been successful. Usually, this is a very good feature, which you want to enable.

Reading the contents of a file into a string is easy too. You can simply use `+stringWithContentsOfFile:`, as in the following example, which reads `@"/home/nicola/test"`:

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

int
main (void)
{
  NSAutoreleasePool *pool = [NSAutoreleasePool new];
  NSString *string;
  NSString *filename = @"/home/nico/test";

  string = [NSString stringWithContentsOfFile: filename];
  if (string == nil)
    {
      NSLog (@"Problem reading file %@", filename);
      // <missing code: do something to manage the error...>
      // <exit perhaps ?>
    }

  /*
   * <missing code: do something with string...>
   */

  return 0;
}
```

2 NSArray

2.1 What is it

Instances of `NSArray` represent arrays of objects. That is, a `NSArray` is used to store an ordered set of objects.

2.2 Comparison with Pure C Arrays

In a GNUstep tool or application, you can use pure C arrays as well, exactly as you do in C. `NSArray`s have some advantages and disadvantages over pure C arrays. The first advantage is that the programmer interface of `NSArray` is slightly easier, which makes your code simpler to read, maintain and debug. In particular, if you use arrays which can be modified (`NSMutableArray`s), the GNUstep library shrinks or expands the array automatically for you as needed when you add or remove objects, without you having to manually allocate or resize the memory needed for the array. The second advantage of `NSArray`s is that they provide facilities to do things which are not necessarily straightforward to do with pure C arrays. In the last section of this tutorial we will be learning about one of this facilities, the ability of saving an array of strings (and other simple objects) into a plain text file, and automatically recreating the array by reading the information from the file. The main disadvantage is that a `NSArray` is slower than a pure C array, but you should not overestimate this problem, which becomes important only when you need really fast code and have to iterate over really big arrays. In most cases, using a C array or a `NSArray` does not make any real difference on the performance; but of course there are cases in which it does.

2.3 NSArrays are immutable

As in the case of strings, `NSArray`s are immutable; you can't change them after creation (this allows the GNUstep `NSArray` code to use some optimisations which would be impossible otherwise). If you need an array that you can modify, you can use a subclass of `NSArray`, called `NSMutableArray`. This happens quite often, so we'll discuss it in some details later on.

2.4 arrayWithObjects

To create an `NSArray`, you usually use the class method `+arrayWithObjects:`, which takes as arguments a `nil`-terminated list of objects to put in the array:

```
NSArray *names;

names = [NSArray arrayWithObjects: @"Nicola", @"Margherita",
                                   @"Luciano", @"Silvia", nil];
```

The last element in the list must be `nil`, to mark the end of the list. You can put in the same array objects of different classes:

```
NSArray *objects;
NSButton *buttonOne;
NSButton *buttonTwo;
```

```
NSTextField *textField;

buttonOne = AUTORELEASE ([NSButton new]);
buttonTwo = AUTORELEASE ([NSButton new]);
textField = AUTORELEASE ([NSTextField new]);

objects = [NSArray arrayWithObjects: buttonOne, buttonTwo,
                                     textField, nil];
```

but you can't put `nil` inside an array. All objects in an array must be valid, not-`nil` objects.

When an object is put in an array, the `NSArray` sends to it a `-retain` message, to prevent it from being deallocated while it is still in the array. When an object is removed from the array (because the array is a `NSMutableArray` so objects can be removed from it, or because the array itself is deallocated), it is sent a `-release` message.

2.5 Accessing Objects in a NSArray

To access an object in an `NSArray`, you use the `-objectAtIndex:` method, as in the following example:

```
NSArray *numbers;
NSString *string;

numbers = [NSArray arrayWithObjects: @"One", @"Two", @"Three",
                                     nil];
string = [numbers objectAtIndex: 2];   // @"Three"
```

Of course, you have to be careful not to ask for an object at an index which is negative or bigger than the size of the array; if you do, an `NSRangeException` is raised (we'll learn more about exceptions in another tutorial).

To get the length of an array, you use the method `-count`, as in:

```
NSArray *numbers;
int i;

numbers = [NSArray arrayWithObjects: @"One", @"Two", @"Three",
                                     nil];
i = [numbers count];   // 3
```

2.6 Describing an Array

It is sometime useful for debugging to have a look at what is stored inside an `NSArray`. Nothing easier with the GNUstep base library: all methods and functions taking format arguments (in the way `+stringWithFormat:` does) recognise the conversion specification `%@`, which describes an object. So, to describe the `NSArray` called `array`, you can just do:

```
NSLog (@"Array: %@", array);
```

For example, an array created with

```
NSArray *array;

array = [NSArray arrayWithObjects: @"Hi", @"Hello",
                                   AUTORELEASE ([NSLock new]),
                                   nil];
```

will be described by the previous `NSLog` call as:

```
Jun 08 08:50:38 Test[16808] Array: (Hi, Hello, <NSLock: 8081f98>)
```

You may note that it is not as easy to get a full description of a pure C array.

2.7 Iterating over Array Elements

2.7.1 First Way - Using objectAtIndex

To iterate over the elements of an array, you may simply use a C-like approach, as in the following debugging routine which prints out the description of all the elements in an `NSArray`:

```
void
describeArray (NSArray *array)
{
  int i, count;

  count = [array count];
  for (i = 0; i < count; i++)
  {
    NSLog (@"Object at index %d is: %@",
           i, [array objectAtIndex: i]);
  }
}
```

Of course, this code is in a certain sense useless, because you can just get the complete description of the array in a single `NSLog` call, as shown in the previous section; but it fulfils its purpose, which is to show a concrete example of how to iterate on array elements.

2.7.2 Second Way - Using objectEnumerator

There is another very important way to iterate over the elements of an array, and it is by using the `-objectEnumerator` method. This method returns an object of class `NSEnumerator`, which can be used to enumerate the objects in the array. The only real thing you need to know about `NSEnumerator` is that it has a method called `-nextObject`. The first time you invoke it, it returns the first object in the array. The second time you invoke it, it returns the second element on the array, and so on till there are no more objects in the array; at this point, the `NSEnumerator` returns `nil`. In the following example, the code to describe an array is rewritten in this second way:

```
void
describeArray (NSArray *array)
```

```
{
  NSEnumerator *enumerator;
  NSObject *obj;

  enumerator = [array objectEnumerator];

  while ((obj = [enumerator nextObject]) != nil)
    {
      NSLog (@"Next Object is: %@", obj);
    }
  }
}
```

This second way is generally slightly faster than the first one but has a very important restriction: you should not modify the array (if it is a `NSMutableArray`) while enumerating its elements in this way. Be careful about this problem, because it is easy to forget this condition - this would introduce subtle bugs.

2.8 Searching for an Object

If you want to check whether an `NSArray` contains a certain object or not, you should use the `-containsObject:` method, as in the following example:

```
NSArray *array;

array = [NSArray arrayWithObjects: @"Nicola", @"Margherita",
                                   @"Luciano", @"Silvia", nil];

if ([array containsObject: @"Nicola"]) // YES
  {
    // Do something
  }
```

`-containsObject:` compares the objects using `-isEqual:`, which is usually what you want: eg, two `NSString` objects containing the same UNICODE characters would be considered equal, even if they are not the same object.

To get the index of an object, you can use `-indexOfObject:`, which returns the index of the object (better, of an object equal to the argument), or the constant `NSNotFound` if no object equal to the argument can be found in the array, as in the following example:

```
NSArray *array;
int i, j;

array = [NSArray arrayWithObjects: @"Nicola", @"Margherita",
                                   @"Luciano", @"Silvia", nil];

i = [array indexOfObject: @"Margherita"]) // 1
j = [array indexOfObject: @"Luca"]) // NSNotFound
```

2.9 NSMutableArray

If you need to add, remove or replace objects in an array, then you should use a `NSMutableArray`. Generally, you create `NSMutableArray`s by simply using

```
NSMutableArray *array;

array = [NSMutableArray new];
```

(you then need to `AUTORELEASE` it if needed). This creates an `NSMutableArray` containing no elements.

2.9.1 Adding an Object

To add an element at the end of the array, you can use `addObject`, as in:

```
NSMutableArray *array;

array = [NSMutableArray new];
[array addObject: anObject];
```

Assuming `anObject` is an `NSObject` (but not `nil`, remember, you can't put a `nil` object into an `NSArray`). As usual, `anObject` is `RETAIN`ed when it is added to the array.

If you want to insert an object into an array at a certain position, you can use `insertObject:atIndex:`:

```
NSMutableArray *array;

array = [NSMutableArray new];
[array addObject: @"Michele"];
[array addObject: @"Nicola"];
[array insertObject: @"Alessia" atIndex: 1];

/* Now the array contains Michele, Alessia, Nicola. */
```

2.9.2 Removing an Object

To remove an object, you can use `removeObjectAtIndex:`, as in

```
NSMutableArray *array;

array = [NSMutableArray new];
[array addObject: @"Michele"];
[array addObject: @"Nicola"];
[array insertObject: @"Alessia" atIndex: 1];

/* Now the array contains Michele, Alessia, Nicola. */

[array removeObjectAtIndex: 0];

/* Now the array contains Alessia, Nicola. */
```

When an object is removed from the array, it receives a `release` message. This balances the `retain` which was sent to the object when it was first added to the array, and allows the object to be deallocated, if needed.

2.9.3 Replacing an Object

To replace an object, you can use `replaceObjectAtIndex:withObject:`, as in

```
NSMutableArray *array;

array = [NSMutableArray new];
[array addObject: @"Alessia"];
[array addObject: @"Michele"];

/* Now the array contains Alessia, Michele. */

[array replaceObjectAtIndex: 1  withObject: @"Nicola"];

/* Now the array contains Alessia, Nicola. */
```

The object which is removed from the array (because it is being replaced) receives a `release` message; the object which is added to the array (because it replaces the other object) receives a `retain` message.

3 NSDictionary

3.1 What is it

An instance of `NSDictionary` contains a set of *keys*, and for each key, an associate *value*. Both keys and values must be objects; the keys must all be different from each other, and must not be `nil`; the values must not be `nil`.

An example of `NSDictionary` may be represented as follows:

```
{
  Luca = "/opt/picture.png";
  "Birthday Photo" = "/home/nico/birthday.png";
  "Birthday Image" = "/home/nico/birthday.png";
  "My Sister" = "/home/marghe/pic.jpg";
}
```

In this example, for each key (`Luca`, etc), which is a string (an image name), there is associated a value (`/opt/picture.png`), which is a string (the full path of the file containing the image). Note that different keys may have the same values. In this case, the same actual graphic file can be accessed using two different names.

In this example, all the objects were strings, but that not need be always the case; keys and values may be arbitrary objects (and not necessarily of the same class). A basic difference between `NSArray` and `NSDictionary` is that the elements contained in an `NSArray` are lined up in a precise order, while the couples key/value contained in a `NSDictionary` are *not* ordered at all. You usually access an object in an array by specifying its position in the array (its index); in a dictionary instead, you rather ask for the value associated with a certain key. So, while arrays are useful to maintain an ordered list of objects, dictionaries are useful to maintain mappings between certain keys and certain values. In other contexts, dictionaries are called hash tables.

3.2 Creating a NSDictionary

To create a `NSDictionary`, you can use the method

`+dictionaryWithObjectsAndKeys:`

which takes as argument a list of objects (to be considered in couples; the first one is the value, the second is the key); the list is terminated by `nil`. The following example creates the dictionary used as example in the previous section:

```
NSDictionary *dict;

dict = [NSDictionary dictionaryWithObjectsAndKeys:
               @"/opt/picture.png", @"Luca",
               @"/home/nico/birthday.png", @"Birthday Photo",
               @"/home/nico/birthday.png", @"Birthday Image",
               @"/home/marghe/pic.jpg", @"My Sister", nil];
```

Please note the the keys follow their values rather than preceding them.

3.3 Retrieving a value with objectForKey:

To retrieve the value associated with a given key, you may use the method `-objectForKey:`, as in the following example:

```
NSDictionary *dict;
NSString *path;

dict = [NSDictionary dictionaryWithObjectsAndKeys:
               @"/opt/picture.png", @"Luca",
               @"/home/nico/birthday.png", @"Birthday Photo",
               @"/home/nico/birthday.png", @"Birthday Image",
               @"/home/marghe/pic.jpg", @"My Sister", nil];

// Following will set path to /home/nico/birthday.png
path = [dict objectForKey: @"Birthday Image"];
```

If the key is not in the dictionary, `-objectForKey:` will return `nil`, for example

```
// Following will set path to nil
path = [dict objectForKey: @"My Mother"];
```

(assuming `dict` is the one created in the previous example). In real life applications, in most cases you don't know if the key is in the dictionary or not until you try retrieving the value associated with it. In these cases, you need to check the result of `objectForKey:` to make sure it's not `nil` before using it. For example, assuming that `dict` is a dictionary, you would normally do –

```
NSString *imageName = @"My Father";
NSString *path;

path = [dict objectForKey: imageName];

if (path == nil)
  {
    // This means the dictionary does not contain it
    NSLog (@"Don't know the path to the image '%@'", imageName);
  }
else
  {
    // Do something with path
  }
```

This is also the standard way to check that a key is contained in a dictionary – you call `objectForKey:` and compare the result with `nil`.

3.4 Enumerating all the Keys and Values

Sometime, you need to iterate over all the key/value pairs in a dictionary. To do this, you use the method `-allKeys` to retrieve an array of all the keys in the dictionary; this array contains all the keys, in no particular (ie random) order. You can then cycle over this array, and for each key retrieve its value. The following example prints out all the key-values in a dictionary:

```
void
describeDictionary (NSDictionary *dict)
{
  NSArray *keys;
  int i, count;
  id key, value;

  keys = [dict allKeys];
  count = [keys count];
  for (i = 0; i < count; i++)
  {
    key = [keys objectAtIndex: i];
    value = [dict objectForKey: key];
    NSLog (@"Key: %@ for value: %@", key, value);
  }
}
```

As usual, this code is just an example of how to enumerate all the entries in a dictionary; in real life, to get a description of a NSDictionary, you just do `NSLog (@"%@", myDictionary);`.

3.5 NSMutableDictionary

An `NSDictionary` is immutable. If you need a dictionary which you can change, then you should use `NSMutableDictionary`.

`NSMutableDictionary` is a subclass of `NSDictionary`, so that you can do with an `NSMutableDictionary` everything which you can do with a simple `NSDictionary` plus, you can modify it.

To set the value of a key in a dictionary, you can use `-setObject:forKey:`. If the key is not yet in the dictionary, it is added with the given value. If the key is already in the dictionary, its previous value is removed from the dictionary (which has the important side-effect that it is sent a `-release` message), and the new one is is put in its place (the new one receives a `-retain` message when it is added, as usual). You should note that, while values are retained, keys are instead copied.

So, if you use a `NSMutableDictionary`, you can create our usual dictionary as follows:

```
NSMutableDictionary *dict;

dict = [NSMutableDictionary new];
AUTORELEASE (dict);
[dict setObject: @"/opt/picture.png"
         forKey: @"Luca"];
[dict setObject: @"/home/nico/birthday.png"
         forKey: @"Birthday Photo"];
[dict setObject: @"/home/nico/birthday.png"
         forKey: @"Birthday Image"];
[dict setObject: @"/home/marghe/pic.jpg"
         forKey: @"My Sister"];
```

To remove a key and its associated value, you can just use the method `-removeObjectForKey:`:

```
[dict removeObjectforKey: @"Luca"];
```

this will remove the key `@"Luca"` and its associated value from the mutable dictionary.

4 Property Lists

As promised, in this last section we are going to have a quick look at a very interesting feature of GNUstep: the support for property lists.

As we saw, it is very easy to save/read a string to/from a file. GNUstep goes well beyond this. You can save/read any object to/from a file, using the archiving/dearchiving facilities; these facilities are great, but save/read objects in a binary format. For arrays and dictionaries containing only strings, GNUstep provides other facilities (the ones we are interested on here) to save/read them from files in a human-readable (and human-modifiable) format, called a *property list*. Actually, the set of objects which you can save and read in this way is much more general:

1. Any string can be saved/read as a property list.

2. Any array or dictionary containing only objects which can be saved/read as a property list can be also saved/read as a property list.

In practice, any array or dictionary which contains other arrays and dictionaries can be saved/read provided that in the end all the 'final' objects are strings.

To save or read a dictionary or an array of the correct type, you write or read them to/from file in the same way as you do with strings, that is, you use the method `-writeToFile:atomically:` to write them to files, and the method `+dictionaryWithContentsOfFile:` (for dictionaries) or the method `+arrayWithContentsOfFile:` (for arrays) to read them from files.

For example, the following code creates a dictionary which is used to store, for each person, some info; then it saves it to a file in the form of a property list:

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

int
main (void)
{
  NSAutoreleasePool *pool = [NSAutoreleasePool new];
  NSMutableDictionary *dict;
  NSDictionary *dict2;

  dict = [NSMutableDictionary new];
  AUTORELEASE (dict);

  dict2 = [NSDictionary dictionaryWithObjectsAndKeys:
            @"Nicola", @"Name",
            @"Pero", @"Surname",
            @"n.pero@mi.flashnet.it", @"Email", nil];
  [dict setObject: dict2  forKey: @"Nicola"];

  dict2 = [NSDictionary dictionaryWithObjectsAndKeys:
            @"Ettore", @"Name",
            @"Perazzoli", @"Surname",
            @"ettore@helixcode.com", @"Email", nil];
  [dict setObject: dict2  forKey: @"Ettore"];

  dict2 = [NSDictionary dictionaryWithObjectsAndKeys:
            @"Richard", @"Name",
            @"Frith-Macdonald", @"Surname",
            @"richard@brainstorm.co.uk", @"Email", nil];
  [dict setObject: dict2  forKey: @"Richard"];

  if ([dict writeToFile: @"/home/nico/testing" atomically: YES])
    {
      NSLog (@"Success");
    }
  else
    {
      NSLog (@"Failure");
    }
```

```
  return 0;
}
```

I tried it on my machine (I encourage you to do the same on yours) and it wrote the following into my `/home/nico/testing` file:

```
{
    Ettore = {
        Email = "ettore@helixcode.com";
        Name = Ettore;
        Surname = Perazzoli;
    };
    Nicola = {
        Email = "n.pero@mi.flashnet.it";
        Name = Nicola;
        Surname = Pero;
    };
    Richard = {
        Email = "richard@brainstorm.co.uk";
        Name = Richard;
        Surname = "Frith-Macdonald";
    };
}
```

As you see, it is a very simple format, and very nice to read and edit manually. The " (speech-marks) are used whenever a string contains spaces or special characters; they are omitted for simple strings. Dictionaries are saved as in

```
{
  Email = "richard@brainstorm.co.uk";
  Name = Richard;
  Surname = "Frith-Macdonald";
}
```

(Note that they use curly brackets, and each key/value pairs is followed by a ; (semicolon), even the last one). Arrays (not shown in the previous example) are saved as in

```
(
 "Nicola Pero",
 "Ettore Perazzoli",
 "Richard Frith-Macdonald"
)
```

(they use round brackets, and each entry is followed by a comma, except the last one). You can find other examples of this format by looking inside your `/GNUstep` directory.

One of the advantages of this format is that it is very general, but still very portable and simple. It is so easy and simple (unless `XML`) that you can write a complete parser/unparser for a new platform in one or two days.

Once you have saved the data to a file in the form of a property list, you can easily retrieve it from the same file. The following code retrieves the data from the file `/home/nico/testing`, and prints out the `Email` of `Ettore`:

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

int
main (void)
{
  NSAutoreleasePool *pool = [NSAutoreleasePool new];
  NSDictionary *dict;
  NSDictionary *dict2;
  NSString *email;

  dict = [NSDictionary dictionaryWithContentsOfFile:
                         @"/home/nico/testing"];
  dict2 = [dict objectForKey: @"Ettore"];
  email = [dict2 objectForKey: @"Email"];

  NSLog (@"Ettore's Email is: %@", email);

  return 0;
}
```