The Nice user's manual

Daniel Bonniot

Bryn Keller

Francis Barber

The Nice user's manual

Daniel Bonniot
Bryn Keller
Francis Barber

Table of Contents

List of Tables

List of Examples

Foreword

This manual describes the Nice programming language. It is currently under redaction, which means that many aspects of the language are absent from it, or that some sections are mostly empty. During this time, it is recommended to read also the Nice tutorial [http://nice.sf.net/language.html], which contains lots of additional information. Both documents currently assume some knowledge of Java, or at least of an object oriented language.

The authors of this manual are Bryn Keller and Daniel Bonniot, with contributions from Francis Barber.

Chapter 1. Philosophy

> A language that doesn't affect the way you think about programming, is not worth knowing.
>
> —Alan J. Perlis

> The art of progress is to preserve order amid change and to preserve change amid order.
>
> —Alfred North Whitehead

The Nice programming language is a new object-oriented programming language based on Java. It incorporates features from functional programming, and puts into practice state-of-the-art results from academic research. This results in more expressivity, modularity and safety.

Safety
: Nice detects more errors during compilation than existing object-oriented languages (null pointer accesses, cast exceptions). This means that programs written in Nice never throw the infamous `NullPointerException` nor `ClassCastException`. This aspect is developed in more detail in this article [http://nice.sf.net/safety.html].

Modularity
: In object-oriented languages, it is possible to add a new class to an existing class hierarchy. In Nice, it is also possible to add *methods* to existing classes without modifying their source file. This is a special case of *multi-methods*.

Expressivity
: Many repetitive programming tasks can be avoided by using Nice's advanced features. Ever got bored of writing tons of loops, casts, overloaded methods with default values, ... ?

Chapter 2. Packages

A *package* is a group of related classes, methods, and variables. You can declare that all the code in a particular `.nice` file belongs to a certain package with the declaration:

```
package package-name;
```

To make use of all the public entities from another Nice package, use the `import` statement:

```
import package-name;
```

Note that this is somewhat different from Java packages. In Java, a class can be used independently of its declaring package. In Nice methods can be declared outside classes, so importing the whole package is important to know the methods available for a given class. This probably implies that Nice projects should be designed with smaller, self-contained packages that Java projects.

Therefore, only whole packages can be imported. It is not possible to import a single class. Likewise, there is no need to include a `.*` after the package name as in Java. In fact, using `.*` indicates that you wish to import the contents of a Java package rather than a Nice package. See the section on Java imports for details.

The `main` method

If a package contains a method whose name is `main` and the method has a `void` return type and takes a single `String[]` as its argument, this method receives special treatment, in that when the program is run, execution will begin with this `main` method. The runtime system will pass the command line arguments used to invoke the program as the argument to `main`.

Note that since the main unit of code in Nice is the package, and not the class, the `main` method should be implemented outside of any class declaration.

Chapter 3. Classes and Interfaces

Declaring a class

Fields

The main component of a class is the list of its fields. A field is a variable that is attached to each instance of the class. It has a type, a name, and optionally a default initial value. The syntax for field declaration is:

```
type field-name[ = initial-value];
```

If no default value is given, then every call to the constructor must specify the value for this field. If it is given, a call to the constructor call still override it with a different value, in which case the default value is not computed (this fact is only important if it has side effects).

Constructors

Classes in Nice have a default (or "automatic") constructor which is generated automatically by the compiler. This constructor allows all the fields of the class to be specified explicitly, though if the field has a default value in the class definition, it can be omitted from the constructor call and the default value will be used. In many cases this default constructor is all that is needed, and it saves the programmer from having to write tedious code that simply accepts values and assigns them to fields.

Example 3.1. Class definition and creations using the automatic constructor

```
class Car
{
  String brand;
  String model;
  int numberOfWheels = 4;
  int numberOfDrivingWheels = 2;
}

void test()
{
  Car renault5 = new Car(brand: "Renault", model: "Cinq");
  Car jeep = new Car(brand: "Jeep", model: "Some jeep", numberOfDrivingWheels: 4);
}
```

It is required to include the names of the fields in the call to the constructor. This is important for two reasons. First, it is easy to understand what the arguments represent, without looking at the class definition. Second, it does not require some arbitrary ordering of the fields [1] . Because the names of the fields

[1] A problem happens in Java when the order of the parameters of a constructor should be changed. This requires modifying all the call sites, which is at best tedious and error-prone, at worse impossible (when writing a library used by others). When the order is changed and some caller are not modified, the following happens: if the exchanged parameters have incompatible types, the compilation of the caller will fail; otherwise the code might even compile and produce wrong results at runtime. There is no simple way to solve this issue in Java. Using names in the call to the constructor in Nice is the solution.

are used, they can be given in any order.

When more control over construction is desired, it is possible to write new constructors that can be used in place of the automatic constructor. These are written much like any other method, but they use a slightly different declaration syntax:

```
new class-name(param-typeparam-name[ = initial-value], ...)
{
  method-body;
  this(argument, ...);
}
```

This syntax requires that the last statement in any custom constructor calls some other constructor for the same type. In most cases, the automatic constructor will be called.

Example 3.2. Class definition and creations using custom constructors

```
/**
 * Class which encapsulates a strategy for
 * way of translating a character.
 */
class Translator
{
  //The function that actually performs the translation
  char->char transFunction;

  //convenience method
  char translate(char input)
    {
      return (this.transFunction)(input);
    }
}

/**
 * Constructor which takes a map of characters, and
 * returns a Translator which looks up the input
 * character in the map, or just returns the input
 * character if it's not in the map.
 */
new Translator(Map<char,char> characters)
{
  this(transFunction: char c => characters.get(c) || c);
}



//A translator which provides rot13 ciphering.
//Uses automatic constructor.
var Translator rot13 = new Translator(transFunction:
                                      char c => char(int(c) + 13));

//A translator which just changes 's' or 'S' to '$'.
//Uses custom constructor.
var Translator sToDollar = new Translator(characters: listToMap([
                                                       ('s', '$'),
                                                       ('S', '$')
                                                       ]));
```

It is also possible to define *initializers* for your classes. An initializer is simply a block of code that executes after construction is finished. To define an initializer, simply include code directly in the class definition, inside a block:

```
class class-name
{
  {
    initializer
  }
  fields-and-methods
}
```

Here is an example. Notice that since the initializer runs after any constructors, the hidden counter field will always be set to zero, even if the caller had tried to set it to some other value when the object was created.

Example 3.3. Class initializers

```
class Counter
{

    {
      this._internal_counter = 0;
    }

  int _internal_counter = 0;
}
```

Parametric classes

A powerful feature of Nice is the ability to define *parametric classes*. Parametric classes are like templates in C++, or similar constructs in various functional languages. Programming with parametric classes is sometimes called *generic programming*.

A parametric (or parameterized) class is simply a class which has a parameter. In this case the parameter is a *type* rather than a value. You can consider a parametric class as a family of related classes, all of which have the same behavior and structure except for the part that is parameterized. A common case where this is useful is in data structures.

Example 3.4. Simple Java Collection

```
class Stack
{
  List contents = new LinkedList();
  void push(Object o)
  {
    contents.add(o);
```

```
  }

  //... omitted methods

  public static void main(String[] args)
  {
    Stack st = new Stack();
    st.push("Test");
    Integer num = (Integer)st.pop(); // Runtime error
  }
}
```

There is a big type safety problem here. We pushed a String on to the stack, and then tried to pop it off into an Integer, resulting in an exception at runtime. Parametric classes can solve this problem.

Example 3.5. Simple Nice Collection

```
class Stack<T>
{
  List<T> contents = new LinkedList();
  void push(T t)
  {
    contents.add(t);
  }

  //... omitted methods

}

void main(String[] args)
{
  Stack<String> st = new Stack();
  st.push("Test");
  Integer num = st.pop(); // Compile time error!
}
```

In the Nice version, we have parameterized `Stack` by a type T. Essentially, Stack<T> is a recipe for the compiler that tells it how to create a `Stack` that works for any given type. So now the compiler knows that we only mean for Strings to go into our stack, and it reports an error when we write code that expects an Integer to come out of our Stack<String>.

Declaring an Interface

Nice has single inheritance, like Java or C#, which means that each class can have at most one superclass which it extends. Sometimes, it would be nice to have a class that "extends" two (or more) different classes, taking some of its behavior and data from each. In Nice, as in Java, this can be accomplished via *interfaces*.

Interfaces are declared just like classes, except that they may not contain data fields, only methods. Unlike in Java, they may also contain default implementations of the methods, making interfaces more convenient and useful than Java interfaces.

To say even this much is still to think of interfaces in Java terms, however. In Nice, an interface doesn't really "contain" anything at all, it's just a marker. Just as `java.io.Serializable` is just a tag to

tell Java that it's okay to use serialization on instances of a class, all Nice interfaces are really tags.

This is because Nice has multi-methods, which are defined by themselves and not contained in a class or interface. It is always possible to define new methods for an interface, just as it is always possible to define new methods for a class. Another consequence of the fact Nice is based on multi-methods is that interface definitions can "contain" not only method signatures, but also default implementations.

Nice does accept a style of interface definition similar to Java's, as in the following example:

Example 3.6. Declaring an Interface

```
interface Component
{

  String getID();

  (int,int) getPosition();

  (int,int) getDimensions();

  int getArea()
  {
    (int width, int height) = this.getDimensions();
    return width * height;
  }

}
```

However, it's equally possible to define the same interface in this style:

Example 3.7. Declaring an Interface with External Methods

```
interface Component {}

String getID(Component comp);

(int,int) getPosition(Component comp);

(int,int) getDimensions(Component comp);

int getArea(Component comp)
{
  (int width, int height) = comp.getDimensions();
  return width * height;
}
```

and in fact, it's fine to mix the two styles, declaring some of the methods inside the `interface` block, and some outside. One good practice might be to declare those methods which have no default implementation inside the `interface` block, and those with default implementations outside of it. That way someone reading the code will have a clear idea of which methods must be implemented when implementing an interface. Of course, the compiler will ensure that all necessary methods are implemented, so this is only a suggestion.

Enumerations

Enumerations or simply (enums) are a group of related constants. Many languages support defining simple constants, and of course Nice does also. Many programs are written to use certain numeric constants with special meanings. For instance, one might write a method for a vending machine:

```
let int NICKEL = 5;
let int DIME = 10;
let int QUARTER = 25;

class VendingMachine
{
  int change = 0;
}

void addCoin(VendingMachine machine, int cents)
{
  machine.change += cents;
}
```

but this method isn't very safe! It will accept any amount of change, including nonsensical amounts like 3, or 234320. One way to address this problem would be to do runtime checks to ensure that the value is acceptable, and throw an exception if it is not. However, there is an easier solution in Nice: the enum.

```
enum class-name[(parameter-type parameter-name, ...)]
{
  option,...
}
```

Enums can be simple symbols like

```
enum Color { Red, Orange, Yellow, Blue, Indigo, Violet }
```

or they can contain integer (or other) values:

```
enum VendingCoin(int value)
{
  nickel(5), dime(10), quarter(25);
}

class VendingMachine
{
  int change = 0;
}

void addCoin(VendingMachine machine, VendingCoin cents)
{
  machine.change += cents.value;
}
```

Of course, a realistic vending machine would have to keep track of the actual number of each coin rather than a total value!

Chapter 4. Methods

Declaring methods

Method declaration takes the following form:

```
[type-parameters] return-type method-name ([parameters]);
```

Note that in Nice, methods can be defined within the body of a class definition, or outside of it - both are identical for the compiler's purposes.

It is also possible to define a default implementation at the same time that the method signature is defined. This form looks like:

```
[type-parameters] return-type method-name ([parameters]) { code-body }
```

Implementing methods

In comparison to their declarations, method implementations can be quite terse:

```
method-name ([arguments]) { body }
```

There is also a special form for methods which consist of a single expression:

```
method-name ([arguments]) = expression;
```

The [*arguments*] consist of names, and optionally types, of the form

```
name
```

or

```
class-name name
```

The *class-name* is used to specialize the method on that argument. An example should make the difference clear. Also note the use of both the block- and expression-style methods.

Example 4.1. Method Implementation

```
class Vehicle {}
class Motorcycle extends Vehicle {}
class Car extends Vehicle {}

//Declare method
int numberOfWheels(Vehicle vehicle);

//Default implementation
numberOfWheels(vehicle)
{
    throw new Exception("Unknown number of wheels!");
}

//Specialize for Cars
numberOfWheels(Car car)
```

```
{
    return 4;
}

//Specialize for Motorcycles
numberOfWheels(Motorcycle mc) = 2;
```

It is also possible to specialize a method on the *exact* class of the argument. Then the method will apply to arguments which match the class exactly, and *not* to an argument which is a subclass of the class specified. The syntax for exact matching is:

```
#class-name name
```

Specializing on the exact class of an argument is useful in situations where an implementation is correct for a certain class, but you know that each subclass will require a different implementation.

There is also an example where exact matching is required to type-check a method with a precise polymorphic type.

When specializing a method with type parameters, it is not necessary or possible to restate the type parameters in the method implementation. However, if you need access to the type parameters as in the example below, use the syntax:

```
<type-parameters> method-name ([arguments]) { body }
```

Example 4.2. Using Type Parameters in Method Implementations

```
//Method definition
<T> T lastItem(Collection<T> coll);


/* This version is incorrect because the type parameter T is not in scope:

lastItem(coll)
{
  Iterator<T> it = coll.iterator();
  //...
}
*/

// This one will work correctly:
<T> lastItem(coll)
{
  Iterator<T> = coll.iterator();
  //...
}
```

Note that it is not possible to dispatch a method call on an array type, nor to specialize a method on the subtype of a type parameter. This means that method implementations of the form:

```
foo(Collection<String> string){}
```

or

```
foo(String[] array){}
```

are not valid. You should use respectively the specializers `Collection` and `Array` instead.

Value Dispatch

In addition to choosing a method based on the classes of the arguments, it's even possible to override a method based on the actual values of the arguments. Currently this feature works with integers, booleans, strings, characters, enums, and class instances which are used as package level constants. It is also possible to override a method for the special case of null.

This feature makes it convenient to code things that might otherwise have required switch statements or nested if/else statements. Using value dispatch can be more flexible than switch or if/else statements, because you can always add new alternatives conveniently by just adding a new method implementation.

Example 4.3. Value Dispatch

```
String digitToString(int digit, String language)
  requires 0 <= digit < 10;

digitToString(digit, language)
{
  throw new Exception("Couldn't convert "digit" to language "language);
}

digitToString(1, "english") = "one";
digitToString(2, "english") = "two";
digitToString(3, "english") = "three";

digitToString(1, "french") = "un";
digitToString(2, "french") = "deux";
digitToString(3, "french") = "trois";

String booleanToYesNo(boolean bool);

booleanToYesNo(true) = "yes";
booleanToYesNo(false) = "no";

enum Grade {
  A, B, C, D, F
}

Grade charToGrade(char input);

charToGrade('a') = A;
charToGrade('b') = B;
charToGrade('c') = C;
charToGrade('d') = D;
charToGrade('f') = F;
charToGrade(input) {
  throw new IllegalArgumentException("Not a grade letter: " + input);
}

char gradeToChar(Grade grade);
gradeToChar(A) = 'a';
gradeToChar(B) = 'b';
```

```
gradeToChar(C) = 'c';
gradeToChar(D) = 'd';
gradeToChar(F) = 'f';

class Person
{
}

let Person BOB = new Person();

void greet(Person p)
{
  println("Hello, anonymous person!");
}

greet(BOB)
{
  println("Hi Bob!");
}

<T> int containsHowMany(?List<T> list);

<T> containsHowMany(List list) = list.size();
containsHowMany(null) = 0;
```

Named parameters

When calling a method it is possible to specify the name of a parameter, followed by : before the value given to that parameter. Named parameters can be given in any order. This is useful when writing the call, because one does not need to remember the order of the arguments, but only their names. This makes the code much easier to understand what each parameter represents, provided that parameter names are chosen carefully.

Example 4.4. Using named parameters

```
void copy(File from, File to) { ... }
...
copy(from: f1, to: f2);
copy(to: f3, from: f4);
```

It is of course still possible to omit the names of the parameters, in which case the arguments must be given in the order of the declaration.

Optional parameters

Many methods have some parameters that are used most of the time with the same value. In Nice, a parameter can be given a default value. That argument can then be omitted when calling the method, and the default value will be used.

Example 4.5. Using named and optional parameters

```
void copy(File from, File to, int bufferSize = 1000) { ... }
...
copy(from: f1, to: f2); // with a buffer size of 1000
copy(from: f1, to: f2, bufferSize: 4000);
```

In the method definition, the optional parameters should be listed after the required parameters. This is needed to allow calls that do not name the arguments and want to use the default values for the optional parameters.

Note that the optional values of parameters can be based on other parameters, for example:

```
T[] slice(T[] array, int from = 0, int to = array.length - 1);
```

Method implementations must still bind all parameters, including the optional ones, and can dispatch on them.

Chapter 5. Assertions and contracts

Syntax

The syntax for assertions is the same as in Java (since 1.4). Assertions are statements of the following form: `assert <boolean expression> : <error message>`. The <error message> is optional.

Preconditions are introduced with the `requires` keyword, and postconditions with the `ensures` keyword. If there are several conditions in either section, they must be separated by commas. For convenience, an optional trailing comma is accepted.

In a method returning a value, the special variable name `result` can be used to refer to the result value of the method in the postcondition. The `old` to refer to values before the execution of the method is not supported yet.

For example, we can define the contract of the add method of a Buffer interface. It is guaranteed that `isEmpty` returns `true` if, and only if, `size` returns `0`. The `add` method can be called only when the buffer is not full. It is guaranteed to make the buffer non-empty, and to increase the size by one.

Example 5.1. Contracts

```
interface Buffer<Elem>
{
  int size();

  boolean isFull();
  boolean isEmpty() ensures result == (size() == 0);

  void add(Elem element)
    requires
         !isfull() : "buffer must not be not full"  // A comma here is optional
    ensures
         !isEmpty() : "buffer must not be empty",   // Note the comma
         size() == old(size()) + 1 : "count inc";
}
```

Enabling assertions and contract checking

By default, assertions and contracts are not used at runtime. They are discarded by the just-in-time compiler, and should cause no slow-down. They can be turned on when starting the JVM.

JDK 1.4 and later

The mechanism is the same as for Java assertions. Checking can be enabled at runtime with **`java -ea ...`**.

JDK 1.1, 1.2 and 1.3

Contrarily to Java, Nice produces programs with assertions that can be run on earlier JDKs. Therefore

there is no problem to distribute Nice programs using assertions and contracts. Since java will not know the `-ea` command line option, they are disabled by default. You can enable them with **`java -Dassertions=true ...`**.

Chapter 6. Statements

Local variables and constants

Local variables may be defined with the `var` keyword. Here is the syntax:

```
var [type] variable-name [= initial-value] ;
```

For local variables (not package variables), the Nice also accepts the Java style of declaration:

```
type variable-name [= initial-value] ;
```

Constants are declared with "let":

```
let [type] variable-name [= initial-value] ;
```

If the variable or constant is of a simple type (i.e., not a parameterized type), then it is not necessary to specify the type yourself, the compiler can infer the correct type automatically.

Package variables and constants

Since Nice uses packages as its largest conceptual unit, variable and constant declarations may appear outside class definitions, and are useful in the same way that static variables in Java are. Package variables are introduced with `var`, and package constants with `let`. The type must be specified, since it is a useful documentation.

Extended `for` statement

Nice supports the traditional `for` loop, with the same syntax as in Java. In addition, it provides a form that allows iterating over the items in a sequence like C#'s `foreach`. Here is the syntax:

```
for ( item-type variable-name : container ) { body }
```

Currently, this version of the `for` statement can be used to iterate over `Collection`s, arrays, `Range`s, `String`s, and `StringBuffer`s.

Example 6.1. Extended `for` statement

```
let String[] strings = ["one", "two", "three"];
for(String s : strings)
{
  println(s);
}
```

Since Nice also has syntax for generating ranges of numbers, another convenient way to use the `for` statement is as follows:

Example 6.2. Extended `for` with Ranges

```
//Print numbers from 1 to 10, inclusive
for(int i : 1..10)
{
  println(i);
}
```

Local methods

Local methods are similar to regular methods (defined at the package or class level), except that they are defined inside another method, and therefore only usable in that scope. They can refer to local variables from their context. Like anonymous methods, local methods cannot be overridden.

Example 6.3. Local method

```
Window createMyWindow()
{
  Window w = new Window();

  void addMyButton(String text)
  {
    w.addButton(new Button(text));
  }

  addMyButton("One");
  addMyButton("Two");
  addMyButton("Three");
}
```

Chapter 7. Expressions

Method calls

A call is usually of the form `f(e1, ..., en)`. `f` can be a method name, or more generally any expression which has a method type (for instance, a local parameter). `e1, ..., en` is the list of arguments to the method. Arguments can optionally be named.

In many object-oriented languages, methods are called with the syntax `e1.f(e2, ..., en)`. This syntax is also accepted in Nice, and is completely equivalent to the previous one. In particular, `e.f` can be used to retrieve the value of the field `f` in the object `e` [1] .

It is possible that a name refers to several unrelated methods. In that case, the types (and if applicable names) of the arguments are used to try to disambiguate which method must be called. If only one method is applicable, it is called. Moreover, if several methods are applicable, but one has a more restricted domain than all the others, it is chosen [2] . Otherwise, the compiler reports an ambiguity error.

Block Syntax for Method Calls

Nice supports an alternative syntax for method calls when one of the arguments is a method of type `void-> T`:

```
method-name(argument, ...)
{
  argument-body
}
```

If an anonymous method would normally be used, it is permissible to omit the arguments and `=>` and instead write the code for the anonymous method in a block following the call. An example will be helpful:

Example 7.1. Block syntax for method calls

```
//First define the method we're going to call:
//Like an 'if' statement
void when(boolean condition, void->void action)
{
  if (condition)
    action();
}

//Now exercise our method with block syntax:
void main(String[] args)
{
  when(1 > 0)
  {
    println("Math is working correctly!");
```

[1] A consequence of these rules is that if the field `f` contains a method, it must be called with `(e.f)(e1, ..., en)`.
[2] For instance, suppose two methods `<T> void foo(Collection<T>)` and `<T> void foo(List<T>)` are declared, and `aList` is an expression of static type List. The expression `foo(aList)` will result in calling the second `foo` method, because List is a subclass of Collection.

```
  }
}
```

Note that we could just as easily have written:

```
when(1 > 0, () => {
  println("Math is working correctly!");
});
```

but with the alternative block call syntax, the code becomes less cluttered, and easier to read. It also helps to eliminate the distinctions between built-in statements and user-defined methods. For instance, the standard `using` statement is actually just a method defined in the standard library.

Tuples

A tuple is a grouping of several values in a single expression. For instance, `("Hello", 2*3, x.toString())` is a tuple whose elements are the string `"Hello"`, the number `6` and the string representation of `x`. The type of a tuple is a *tuple type*, whose elements are the types of the corresponding values. The type of our example tuple is `(String, int, String)`.

A tuple can be of any length. In english, a tuple of two elements is called a couple. For an arbitrary number n, a tuple of n elements is called a n-tuple.

Tuple types are covariant, so a couple of ints can be used where a couple of longs is expected.

A funny feature is that swapping two variables can be done without a temporary variable, using tuples: `(x, y) = (y, x)`. An important use of tuples is to allow a method to return several values. Example 7.2, “Method returning several values using a tuple” defines and uses the method `minMax`, which takes two integers, and returns the smallest first, the biggest second.

Example 7.2. Method returning several values using a tuple

```
(int, int) minMax(int x, int y) = x < y ? (x, y) : (y, x);

void printTuple((int x, int y))
{
  println("(" + x + ", " + y + ")");
}

void main(String[] args)
{
  printTuple(minMax(14, 17));
  printTuple(minMax(42, 41));
}
```

Note that `printTuple` has only one argument, which is a tuple. We give the names `x` and `y` to the two components of this tuple, so that we can use them directly in the implementation of the method.

Arrays

An array can be created by simply enclosing the elements of the new array in brackets.

```
class-name[] variable-name = [ element1, element2, ...]
```

It is also possible to create an array which is filled with nulls, or some single value, or a value which is calculated on a cell-by-cell basis, by using the `fill` method in the standard library.

Example 7.3. Initializing an array with a single value

```
//Fill every cell with the value 5000.
int[] points = new int[10];
fill(points, 5000);
```

Example 7.4. Initializing an array with a calculated value

```
//build a little table of squares
int[] squares = fill(new int[10], int i => i * i);
```

String Literals

Literal strings are surrounded with double quotes, for instance:

```
String s = "Hello";
```

The backslash character \ is used for escaping, as in Java.

Multi-line Strings

Nice also allows multi-line string literals, using syntax borrowed from the Python language. Multi-line strings begin and end with three double quotes. Within multi-line strings, double quotes do not need to be escaped, unless there are three in a row.

Example 7.5. Multi-line strings

```
let greeting  = """
    Hello, world.
    You may be thinking, "Why was I called here today?"
    Well, there was a good reason. Honest.
""";
```

String concatenation

Just as in many other languages, strings can be concatenated using the plus (+) operator.

Strings can also be combined by simple juxtaposition - that is, by placing them side by side with some other expression. For example:

Example 7.6. String concatenation

```
String name = "Nice";
int factor = 10;
println("My favorite language is " name ", which is " factor
          " times better than what they make me use at work.");
```

Anonymous methods

Since Nice allows methods to be passed as arguments, and returned as results, it is convenient to allow small anonymous methods to be defined at the same point in the program in which they are used, similar to the use of anonymous classes in Java. Anonymous methods cannot be overridden, and have exactly one implementation.

The syntax of anonymous methods is:

```
([parameters]) => { body }
```

As with named methods, there is also a single-expression format:

```
([parameters]) => expression
```

The parentheses around the parameters are optional if there is only one parameter.

The return type of an anonymous method is automatically determined by the compiler using a process called *type inference*, so it is not necessary or possible to specify it.

Example 7.7. Using Anonymous Methods

```
String concat(Collection<String> strings)
{
    StringBuffer buff = new StringBuffer();
    strings.foreach(String s =>  {
        buff.append(s);
    });
    return buff.toString()
}

var String[] numberStrings = [1,2,3,4,5].map(int num => String.valueOf(num));

//Another way of defining "concat"
String concat2(Collection<String> strings)
{
  return strings.foldLeft((String accum, String s) => accum + s, "");
}
```

Operators

Nice supports a wide range of operators which will be familiar to most programmers. The table below lists the operators, ordered from highest precedence to lowest precedence.

Table 7.1. Operators

Operator	Kind	Associativity
`()`	grouping	none
`[], .`	postfix	left
`new`	prefix	right
`++, --`	postfix	left
`+, -, ++, --, ~, !`	prefix	right
`**`	binary	right
`*, /, %`	binary	left
`+, -`	binary	left
`<<, >>, >>>`	binary	left
`..`	binary	left
`<, <=, >, >=`	binary	both
`instanceof`	binary	left
`=, !`	binary	left
`&`	binary	left
`^`	binary	left
`|`	binary	left
`&&`	binary	left
`||`	binary	left
`?:`	ternary	right
`=>`	binary	right
`*=, /=, %=, +=, -=, &=, ^=, _=, <<=, >>=, >>>=, =`	binary	right

Conversion between primitive types

The numeric primitive types are, from the largest to the smallest: double, float, long, int, short, byte. Conversion from a smaller to a larger type is automatic. Conversion from a larger to a smaller type must be done explicitly, since they can lose information about the magnitude of the value.

The explicit conversion is done by calling a special method, whose name is the target type. If *`e`* is a numeric expression, and *`type`* one of the numeric types, then *`type`*`(`*`e`*`)` will convert the value of the expression, so that it is represented as a value in *`type`*. This is equivalent to the `(`*`type`*`)` `e` syntax in Java.

In most cases, characters should be treated as abstract entities and not as the numbers that happen to encode them. Therefore, the `char` type is not a numeric type in Nice. Operations on characters can be performed using the methods of class `java.lang.Character`. Additionally, it is possible to convert a character `c` to its integer value with `int(c)`, and to convert an integer to the character it encodes with `char(i)`.

For instance, here is code to read characters from a Reader, whose `read` method returns an int (`-1` meaning that the end of stream has been reached):

Example 7.8. Converting a value of a primitive type

```
let reader = new BufferedReader(new InputStreamReader(System.in));

int v;
while ((v = reader.read()) != -1) {
  char c = char(v);
  ... // Do something with the character.
}
```

Chapter 8. Interfacing with Java

Using Java from Nice

The Java libraries are automatically imported as necessary. Therefore it is straightforward to use existing Java code in Nice programs. In particular, it is possible to call Java methods, to instantiate Java classes, to create a Nice subclass of a Java class, overriding methods from the parent, ...

This section lists advanced features related to the use of Java code in Nice.

Subclassing a Java class

A Nice class can extend a Java class, simply by naming it in its `extends` clause. Similarly, it can implement Java interfaces.

When instantiating a Nice class with a Java parent, the automatic constructor comes in several versions, one for each constructor of the parent class. The call must then provide the values for one of the constructors of the parent, followed by the named values for the fields defined in the Nice class.

Importing packages

When Java classes of some package are used frequently, it is possible to make the package part of their name implicit. For instance, after a declaration `import java.io.*;` it is possible to refer to class java.io.File with the short name File. It is not possible to import only a single name, such as `import java.io.File;`, you must import the whole package.

Classes defined in the current package always have priority over imported packages. If a short name is used that refers to two classes from imported packages, the compiler will report an ambiguity, and the fully qualified name must be used.

Option types

Nice's advanced type system makes the distinction between normal (e.g. String) and option (e.g. ?String) types, which allows to prevent `NullPointerExceptions` (see option types). This poses a problem when using existing Java libraries. If my Nice program calls a Java method that returns a String (the Java type), does it mean String or ?String in Nice? Since Java allows the value to be `null`, the Nice compiler currently supposes it's ?String (it can only be sure about primitive types like int).

```
// Java code
public class MyJavaClass {
  public String getName() {
    return "foo";
  }
}
```

In Nice, if `myJavaClass` is a variable of type MyJavaClass, then `myJavaClass.getName()` has type ?String.

If the Java code might indeed return `null`, or you are not sure, this is all good. To use the value you will have to test it against `null`. However there are many cases where you know the value cannot be `null`, for instance if this fact is specified in the Javadoc comment of the method. In that cases there are currently two possibilities. The first is to use the `notNull` method. So in our example `notNull(myJavaClass.getName())` has type String. This solution is simple, but can be annoying if

you call the same method many times, or because it makes your code less readable.

The second solution is to once and for all tell the compiler the precise type of the method (this is also useful for methods with parametric types):

```
String getName(MyJavaClass) = native String MyJavaClass.getName();
```

The Nice part (on the left of the `=` character) is the header of a method definition. Note that return type is String (without the optional type marker `?`). On the other hand, the right part says that the method is already defined in class MyJavaClass, with name `getName`, the one that takes no argument and returns a String. With this declaration, `myJavaClass.getName()` has type String.

Using Nice from Java

It is possible to use libraries developed in Nice in a Java program. Before anything, make sure that the classes generated by nicec can be found on your classpath when you compile the Java program with javac.

Calling a method

For methods declared inside a class, and for those declared at the package level and whose first argument is a Nice class, you can simply use them as usual in Java.

This is not possible for methods declared at the package level and whose first argument is a Java class, a possibly null type or a primitive type. You can call a such a method `m` declared in a nice package `pkg` by naming it `pkg.dispatch.m` in the Java program. It does not matter in what package(s) `m` is implemented: `pkg` must simply be the package where the method was declared.

Calling a constructor

To instantiate in Java a class defined in Nice, do as usual: call its constructor, using the `new` keyword. Nice classes have one automatically generated constructor, with one parameter for each field of the class, including those inherited from Nice super-classes. Additionally, if some fields have default values, then a second constructor is generated. It only has parameters for the fields without a default value, and assigns to the other fields their default value.

Complete example

Example 8.1. Using Nice from Java

Let's see an example that illustrates all these features. Suppose your Nice library looks like this:

```
package my.nice.pkg;

class Person
{
  String name;

  // A method:
  String display() = "Person: " + name;
}
```

```
class Worker extends Person
{
  int salary;

  display() = "Worker: " + name + " salary: " + salary;

  boolean isRich() = salary > 500;
}
```

Then your Java program can create new persons and workers:

```
package main.java.pkg;

import my.nice.pkg.*;

public class Main
{
  public static void main(String[] args)
  {
    Person p = new Person("John");
    Worker w = new Worker("Julia", 1000);

    System.out.println(p.display());
    System.out.println(w.display());
    if (w.isRich())
      System.out.println("A well paid worker!");
  }
}
```

Optional parameters

Nice allows method and parameters to be optional. Since this feature is not available in Java, you have to provide the value of all parameters, regardless of their optionality. The same applies for class fields with initializers: they are ignored in the Java program, and must be given a value in the call to the constructor.

Other aspects

Fields of Nice classes are accessed normally from Java programs. It is possible to declare a Java subclass of a Nice class. The constructor will need to call the parent constructor. It is not fully possible to override a Nice multi-method in a Java program. However, provided that the method is compiled as an instance method, as specified above, you can override that method as usual for Java. That is, only the first (implicit) argument can be used to perform the override.

Chapter 9. Types

Option types

Nice (unlike Java) makes a distinction between a type that may be `null`, and one that may not be. So, if you want to allow `null` `String`s, you write `?String` for the type. If `null`s should not be allowed, you just write `String`. It is not possible to pass a `?String` where a `String` is expected, unless the compiler can prove it's not `null`. The easiest way to prove that the variable is not `null` is to use an `if` statement:

Example 9.1. Using option types

```
void foo(String arg) {...}

void bar(?String arg) {
        if (arg != null) {
                //Here Nice knows arg is not null, so it can
                // be passed to a method which takes a String.
                foo(arg);
        }
        foo(arg); //Here arg may or may not be null,
                  //so Nice gives a compilation error.
}
```

Therefore, you never have to check that all your arguments aren't `null` again, unless you actually want to allow `null`s.

It's important to remember that nullness inference doesn't work on fields of objects, only on local variables. There's a good reason for this: the value of an instance variable may have changed (updated by another thread, perhaps) between the time the value was checked for nullness and the time it is actually used. To avoid this problem, simply take a local reference to the field first:

Example 9.2. Nullness inference with fields

```
class Person
{
  ?String name
}

//Takes a String, not a ?String
void printName(String name)
{
  println(name);
}

void main(String[] args)
{
  Person p = loadPersonFromFile(args[0]);
  let name = p.name;
  if (name == null)
    throw new Exception("Unknown person!");
```

```
    else
      //safe to call printName, we know name is a String.
      printName(name);
  }
```

Sometimes, you know that a value cannot be `null`, although it has an option type. You can then use the `notNull` method to assert this fact. If an expression *`e`* has type ?*`type`*, then `notNull(`*`e`*`)` has type !*`type`*. Note that if *`type`* is not a type variable, !*`type`* is the same as *`type`*.

The `notNull` method uses `assert` to check that the argument is not `null`. This means that the check will only happen at runtime if assertion checks are enabled. Otherwise, execution will continue, but the JVM will probably fail soon afterwards with a `NullPointerException` if the value is `null`, and it is used as if it was not `null`.

There are (rare) situations where it is necessary to allow a `null` where one shouldn't normally go, for instance if you need to declare a variable before entering a loop, but don't have a value to initialize it with. You should first try to think about an alternative way of writing the code so that this is not needed. However, if there is none, a solution is to use `cast(null)`. This expression will be accepted in any context. It is then your responsibility to make sure that this value is not used.

Note that `notNull` and `cast(null)` have completely different uses. `notNull` is used to tell the compiler that you know a certain value is not `null`, although the type system does not guarantee it. On the other hand, `cast(null)` is used to provide a non-null value which is never going to be used. You could not use `notNull(null)` for that purpose, because this would fail at runtime if assertion checks are enabled.

Additional information is available on the Wiki page about option types [http://nice.sourceforge.net/cgi-bin/twiki/view/Doc/OptionTypes].

There is related type, which is prefixed with an exclamation point (`!`). This is used to specify a non-`null` type when it is not known if the original type allowed `null` or not. This is only needed for type parameters; for all other types, !*`type`* is equivalent to *`type`*.

For instance, in the following method:

```
<T> T myMethod(Collection<T>);
```

the type parameter T can be instantiated at *any* type, including option types like ?String. If it is necessary to exclude option types from the domain of T, the type can be specified this way:

```
<!T> !T myMethod(Collection<!T>);
```

Class Casting

Class casting is not used in Nice. Similar to Option types and nullness, if the compiler can prove that the type of an object is compatible with the target context, it will allow that object to be used. This can be done using the `instanceof` operator and an `if` statement:

Example 9.3. Proving the type of an object

```
        void foo(FileOutputStream arg) {...}

        void bar(OutputStream arg) {
                if (arg instanceof FileOutputStream) {
```

```
                    //Here Nice knows arg is a FileOutputStream, so it can
                    // be passed to a method which takes a FileOutputStream.
                    foo(arg);
                }
                foo(arg); //Here arg may or may not be a FileOutputStream,
                          //so Nice gives a compilation error.
        }
```

As with Option types, type inference doesn't work on fields of objects or on method calls, only on local variables. There's a good reason for this: the value of an instance variable may have changed (updated by another thread, perhaps) between the time the value was checked for its type and the time it is actually used. Similarly, method calls can return objects of different sub-types every time they are called. To avoid this problem, simply assign the value to a local variable first.

Type Parameters

Classes and methods can be parameterized with type variables in Nice as is demonstrated in each of their respective chapters. In addition to the ability to introduce type variables, Nice provides the ability to constrain those types in certain ways. Option types are one such constraint.

Simple type parameters of the form

`<`*`type-variable`* [, *`type-variable`* ...]`>`

are commonly encountered in Nice, but are a simplified form of a more general syntax. <T> is actually shorthand for <Any T>, which says that T may be any type at all. It is possible to constrain T so that it must be a subtype of some other type. This example shows that `filter` may be implemented for any type T, but C must be a subclass of Collection.

```
<Collection C, T> C<T> filter(C<T>, T->boolean);
```

It's also possible to enforce some relation among the type parameters, such that one must be a subtype of the other, as in this example from the standard library, which can be read as "Any T and U, so long as U is a subtype of T".

```
<T, U | U <: T> U[] fill(T[] array, int->U value)
```

Note the notation `<T | T <: SomeType >` means the same thing as `<SomeType T>`.

There is also a notation for the same class as the declaring class of a method: alike. For instance, the following class Foo declares a method `copy`, such that `x.copy()` has the same type as `x`, `x` being of any subclass of Foo.

Example 9.4. A copy method with an exact type

```
class Foo
{
  String value;

  alike copy();
}

copy(#Foo f) = new Foo(value: f.value);
```

```
class Bar extends Foo
{
  int id;
}

copy(#Bar b) = new Bar(value: b.value, id: generateNewId());
```

Note that exact matching is required to implement the `copy` method.

Abstract Interfaces

Nice offers a powerful tool known as the *abstract interface*. Abstract interfaces are similar in some respects to regular Java- or Nice-style interfaces in that they define a set of methods that a type must implement to be considered a member of the interface.

Two things make an abstract interface different from a regular interface. First, a class can be made to implement an abstract interface *after* it's been defined, and even the source code is unnecessary. You can make the basic `String` class implement your new abstract interface, if you like. Second, an abstract interface is not actually a type, it's an annotation which can be applied to types. This means that when you declare an abstract interface, you're not creating a new type, you're saying "types which implement this abstract interface will all have these methods", but those types are not related in any other way. So it is not possible to make a `List<MyAbstractInterface>`, or to use an abstract interface as a type declaration like `var MyAbstractInterface absInter = ... .`

So what is an abstract interface *for*? Abstract interfaces can appear in type parameters, so one can write methods for them:

```
<MyAbstractInterface T> void doSomething(T thing);
```

and then apply these methods to any object whose class implements `MyAbstractInterface`. It's especially useful for situations where one wants to use a method on a group of unrelated types, as in the example below. We have a `log()` method which is parameterized for any class which implements the abstract interface `LogEntry`, and we make a number of classes implement `LogEntry`, so they can be passed to our log method.

Example 9.5. Abstract Interfaces

```
//Abstract interface for things which can be logged.
abstract interface LogEntry
{
  String toLogString();
  int severity();
}

//Some constants for severity levels
let int DEBUG = 1;
let int INFO = 2;
let int ERROR = 3;

<LogEntry E> void log(E entry, int severity = -1)
{
  if (severity < 0)
    severity = entry.severity();
```

```
  someLogWriter.println(timeStamp() + " " + severity + " " + entry.toLogString());
}

//Make strings pass straight through as DEBUG info.
class java.lang.String implements LogEntry;

toLogString(String s) = s;
severity(String s) = DEBUG;


//Make throwables print a stack trace, plus a message
class nice.lang.Throwable implements LogEntry;

toLogString(Throwable t)
{
  let writer = new StringWriter();
  let printWriter = new PrintWriter(writer);
  printWriter.println("ERROR: " + t.getClass().getName() + ": " + t.getMessage());
  t.printStackTrace(printWriter);
  printWriter.close();
  return writer.toString();
}

severity(Throwable t) = ERROR;

//We like to log requests, too
class javax.servlet.http.HttpServletRequest implements LogEntry;

toLogString(HttpServletRequest r) = "Request from: " + r.getRemoteHost();
severity(HttpServletRequest r) = INFO;
```

There are some interesting things to notice about this code. First, we only had to write `log()` once, and we left it up to the LogEntry to do the formatting. This could be done with a regular interface as well, except that we also made String and Throwable implement our abstract interface, which we couldn't have done with a regular interface. So now we can write code like

```
log(request);
log("Beginning processing");
try
  {
    1/0;
  }
catch (Exception e)
  {
    log(e);
  }
```

and our abstract interface implementations will take care of making sure that we get the right formatting and severity levels. The most interesting thing about this code is that if we write `log(5)` then the compiler will catch the error, because `byte` doesn't implement `LogEntry`. If we had defined `log` with the signature `<E> void log(E entry, int severity = -1)`, we could have achieved all the same effects, but we would have lost the type safety abstract interfaces gave us above - because `<E>` means *any* type is allowed, so we would have had to define some defaults:

```
toLogString(e)
{
  throw new Exception("No toLogString method defined!");
}

severity(e)
{
  throw new Exception("No severity method defined!");
}
```

Then we'd be no better off than in a dynamically typed language - we wouldn't find out that we'd tried to log an integer until we got an exception at runtime. With abstract interfaces, we were able to tell the compiler *exactly* which classes should be allowed as arguments to `log`, and so our program is safer as a result.

Daniel Bonniot, Nice's creator, invented abstract interfaces. You can read more about them in the Academic Research [http://nice.sourceforge.net/research.html] section of the Nice website.