
Kenya Reference Guide

Tristan Allwood

Matthew Sackman

The Kenya Reference Guide

This is a guide to the usage the Kenya System and Language.

Table of Contents

Advanced Kenya Features Examples

List of Tables

List of Examples

Next

Chapter 1. Kenya Syntax

Chapter 1. Kenya Syntax

Tristan Allwood

Table of Contents

Comments

Comments in any programming language are important, as they allow the programmer to annotate the program code, to make it clearer and more easily maintainable.

There are two types of comments in Kenya, *Block Comments* and *Line Comments* (these are the same as in Java).

Block comments can be used to mark several lines, or just a few characters as "comments" (i.e. not Kenya program code).

Block comments start on a /* symbol, and then finish on the first */ (this property means they cannot be nested).

Line comments are denoted by //, and make everything to the right of them on a line "commented out".

Example 1.1. Comments Example

```
        /* This is a block comment
         * that spreads across multiple lines.
         * The *'s on the left are a style that helps to see
         * what is a comment and what is not in source code.
         */

        void main(){
                int a = 3;

//              This line is a comment
                int a = a + 2; // A now equals 5

                /* these lines are commented out - i.e. they are not seen
                 *  when Kenya compiles / executes / translates them.
                 * a = a + 1;
                 * a = a / 2;
                 * a = ??? <- What should I put here?
                */

                myFunction(a);
        }

        /* on the line below I have commented out the second argument to the function
         * so it is really a 1 argument function; note, just because you can do this,
         * DOESNT mean you should do this :) */

        void myFunction( int a /*, int b */ ) {
//              println(a + b); This line is commented out thanks to line comments.
                println(a);
        }
```

Methods

Method Declarations

Methods in Kenya have the following structure:

```
returnType methodName( arguments... ){
        statements...
}
```

The return type for a function can be any valid type (basic, array, class or user defined), if you don't wish to return something from a function, use the keyword *void*.

The arguments are a list of variable declarations which are passed in from the calling method, see the example below:

Returning from a Method

Returning from a method is usually directed by the keyword *return*.

If a method is declared to return a type (i.e. isn't declared void), then all possible routes through the method that could run to completion must feature a return statement, with an expression which is what-to-return.

If a method is declared void, then return statements with no expression may be used to prematurely return from the method, or the method can just "run off the end" where an implicit return is assumed.

For more, see the example below.

Method Invoking

To invoke a method, use its name, and pass in arguments of the correct type in the same order as declared.

If a method has a non-void return type, you can also use the returned value from the function as part of an expression.

Methods Example

Example 1.2. Methods Example

```
void main(){
        int seven = addTwoNumbers( 3, 4);

        printANumber( seven );

        int[] nums = { 1,2,3,4,5,6 };
        printANumber ( getLargest( nums ));
}

int addTwoNumbers(int a, int b){
        return a + b;
}

void printANumber( int number ){
        println(number);
}

int getLargest( int[] array ){

        if( array.length == 0 ){
                return -1;
        }

        int cLargest = array[0];

        for(int i = 1 ; i < array.length ; i++ ){
                if( array[i] > cLargest ){
                        cLargest = array[i];
                }
        }

        return cLargest;

}
```

Control Flow

Statements in Kenya do not all have to be assignments, method calls and returns. The flow through a program can be manipulated using loops and conditionals.

For Loops

The for-loop construct is very powerful and flexible. Its syntax is:

```
for( loop_start ; loop_condition ; loop_update ) {
        statements...
}
```

What happens is that loop_start is executed just before any loops, loop_condition is tested each time round the loop, and if and only if that returns true are the statements executed. Then after the statements are executed, loop_update is executed, then the loop_condition is tested again, and thus the loop loops.

The most common use for a for loop is to do something to every element of an array; e.g:

```
void main(String[] args){
        //prints out the provided arguments to the program.
        for( int i = 0 ; i < args.length ; i++ ){
                println( args[i] );
        }
}
```

However there are many other things you can use them for aswell, as loop_start and loop_update can be (almost) any appropriate statement, loop_update is also optional (i.e. you don't have to put it there).

If you are in the middle of a for loop and wish to exit it prematurely, you can use the keyword *break* to immediatly leave the current loop.

Example 1.3. For Loop Example

```
void main(){

        int[] x = { 1, 3, 2, 5, 4 };


        for( int i = minArray(x) ; i <= maxArray(x) ; i++ ){
                for( int j = 0 ; j < x.length ; j++ ){
                        if( x[j] == i ) {
                                println("The number " + i + " is in the array at index " + j );
                                break;  // this breaks out of the first for loop.
                        }
```

```
                }
        }

}


int minArray( int[] a ){
        int minVal = a[0];

        for( int i = 0 ; i < a.length ;i++){
                if( a[i] < minVal ){
                        minVal = a[i];
                }
        }
        return minVal;
}

int maxArray( int[] a ){
        int maxVal = a[0];

        for( int i = 0 ; i < a.length ;i++){
                if( a[i] > maxVal ){
                        maxVal = a[i];
                }
        }
        return maxVal;
}
```

While Loops

The syntax of a while loop is below. Its general meaining is "while the condition is true, execute the loop".

```
vhile ( condition ) {
        statements ...
}
```

It tests a condition before each iteration of the loop, if it is true the loop is executed, if it false, we leave the loop.

As with for loops, the *break* keyword can be used to prematurely exit from the loop.

Example 1.4. While Loops Example

```
void main(){

        println("Whatever you say, I can say too :)");

        while( !isEOF() ){
                String s = readString();

                if( s == "stop" ){
                        break;
                }

                println( s );
        }

}
```

If Statements

if statements can come in three very similar varieties, as shown below:

```
if( boolean_condition ) {
        statements executed only if condition is true;
}


if( boolean_condition ) {
        statements executed only if condition is true;
} else {
        statements executed only if condition is false;
}


if( condition1 ) {
        statements executed only if condition is true;
} else if ( condition2 ) {
        statements executed only if condition1 is false and condition 2 is true.
} //else//else if here as appropriate
```

With chained if-then-else if-then-else if statements, it is important to remember that the first true block is executed, and none of the rest.

Example 1.5. If Statement Example

```
void main(){

        String s = "hi";

        int j = 3;

        if ( j == 3 ) {
                println("j == 3");
        } else {
                println("j != 3");
        }

        if( s == "hi" ){
                println(s);
        }


        if( j != 3 ){
                println("j not 3");
        }else if( s == "hi" ){
                println("s = hi");
                j = 4;
        }else if( j == 4 ){
                //even tho j gets set to 4 above,
                //because this is in an else, this
                // line is not executed.
                println("j == 4");
        }
```

```
        if( j == 4 ){
                println("but j == 4");
        }

}
```

Switch Statements

Switch statements allow you to select code to execute depending on the value of an int (or char).

The basic syntax of a switch is as follows:

```
switch( int_expression ) {

        case int_constant_expression:{
                statements;
                break;
        }
        case int_constant_expression:{
                statments;
        }
        default:{
                statements;
        }
}
```

The program jumps to the case that matches the value of the int, and executes the statements for that case, and all the cases directly below it in order.

To stop this behaviour (which is known as falling-through), a break statement can be used to leave the switch statement.

If none of the cases match, the default case is executed (if present), or (if there is no default case) the entire switch is skipped.

Example 1.6. Switch Example

```
void main(){
        for( int i = 0 ; i < 6 ; i++ ){
                printSomething(i);
                println();
        }
}


void printSomething(int i){

        switch( i ) {
                case 0:{
                        println("Case 0");
                }
```

```
                        case 2:{
                                println("Case 2");
                                break;
                        }
                        case 4:{
                                println("Case 4");
                        }
                        default:{
                                println("Default");
                                break;
                        }
                        case 1:{
                                println("Case One");
                        }
                        case 3:{
                                println("Case Three");
                        }
                }
        }
```

Assertions

An assertion is a way of checking what you think should be true at a program point is true at that point. The syntax for an assertion comes in two forms (with and without a message):

```
assert boolean_condtion;

assert boolean_condition : string_message_to_print;
```

Assertions can, for example, be used to check pre-conditions during development of code, e.g:

Example 1.7. Assertion Example

```
void main(){
        printNumber(3);
        printNumber(8);
        printNumber(-1);
}


void printNumber(int a){
        assert ( a >= 0 && a <= 10 ) : "Invalid argument";

        println(a);
}
```

Variables

The syntax for declaring a variable comes in two varieties, depending on whether you initialise the variable aswell.

```
//to declare a variable
type name;

//to declare a variable an initialise it to the given value.
type name = expression;
```

Now that you have a variable, you can use it in expression inbetween operators, or pass it to other methods, or assign it new values.

If you try and use a variable that has been declared, but not assigned a value, in general you will get an error since it hasn't been initialised.

With variables of a user-defined class type, you do not need to use an initialiser, as they are implicitly created for you when you declare them. (Note: in Java this is not the case)

Operators

Operators are things that are used to build expressions between variables, literals and functioncalls, below is the reference for all the operators in Kenya and what the types they operate upon are.

Table 1.1. Kenya Boolean Operators

Operator	Name	Function	Example
&&	conditional and	boolean and, but only evaluates the second operand if the first one is true	`void main(){ boolean b = false; if( func1() && b ){ /*definatly does not run*/ } }`
||	conditional or	boolean or, but only evaluates the second operand if the first one is false	`void main(){ boolean a = true; if( a || func2() ){ /*definatly runs*/ } }`
^	binary xor	the result is binary xor (exclusive-or) of its two boolean arguments	`void main(){ boolean a = true; boolean b = true; if( a ^ b ){ /* definatly does not run */ } }`
!	binary negation	The result is the bianary negation of its argument	`void main(){ boolean a = true; if( !a ){ /* definatly does not run */ } }`

Table 1.2. Kenya Numerical Operators

Operator	Name	Function	Example

+	Plus	Adds two numbers, and returns the result	`void main(){`
			`  //Prints out 5.`
			`  int x = 3 + 2;`
			`  println( x );`
			`}`

-	Subtract	Subtracts two numbers, and returns the result	`void main(){`
			`  //Prints out 3`
			`  int x = 5 - 2;`
			`  println( x );`
			`}`

*	Multiply	Multiplies two numbers, and returns the result	`void main(){`
			`  //Prints out 6`
			`  int x = 3 * 2;`
			`  println( x );`
			`}`

/	Divide	Divides two numbers. If both numbers are int's, this does an integer division (i.e. rounds the result), if either is a double, it does a double-precision division	`void main(){`
			`  //Prints out 1`
			`  int x = 3 / 2;`
			`  println(x);`
			
			`  //Prints out 1.5`
			`  double d = 3.0 / 2.0;`
			
			`  println(d);`
			`}`

%	Modulo	Returns the remainder from dividing two numbers	`void main(){`
			`  //Prints out 1`
			`  int x = 3 % 2;`
			
			`  println( x );`
			`}`

Table 1.3. Kenya Relational Operators

Operator	Name	Function	Example
<	less than	compares two numeric arguments (char, int, double) and returns binary true if the left-hand argument is *less than* the right hand argument, or binary false otherwise	`void main(){ int a = 3; double d = 1.0; if( d < a ){ /* this definatly runs */ } }`
<=	less than or equals	compares two numeric arguments (char, int, double) and returns binary true if the left-hand argument is *less than or equal to* the right hand argument, or binary false otherwise	`void main(){ int a = 3; double d = 1.0; if( d <= a ){ /* this definatly runs */ } }`
>	greater than	compares two numeric arguments (char, int, double) and returns binary true if the left-hand argument is *greater than* the right hand argument, or binary false otherwise	`void main(){ int a = 3; double d = 1.0; if( d > a ){ /* this definatly does not run */ } }`
>=	greater than or equals	compares two numeric arguments (char, int, double) and returns binary true if the left-hand argument is *greater than or equal to* the right hand argument, or binary false otherwise	`void main(){ int a = 3; double d = 1.0; if( d >= a ){ /* this definatly does not run */ } }`

Notes:

In previous versions of Kenya there where friendly versions of '&&', '||' and '!' (being called 'and', 'or' and 'not'). These have now been dropped as they are not in java.

All the operators above have the same function in java (except for String equality as noted), however there are a few extra things you can do in java that you can't do in Kenya:

- '&' and '|' are not in kenya (binary operators that definatly evaluate both the left and right hand side of the expression).
- The bitwise operators/versions of '&', '|', '~', '^' are not in Kenya.
- The instanceof operator is not in Kenya.
- The integer-bitwise shift operators <<, >> and >>> are not in Kenya.

Chapter 2. Kenya Types

Tristan Allwood

Table of Contents

Kenya Primitive Types

Table 2.1. Kenya Primitive Types

Name	Description	Range of values	Default value
boolean	Represents an elementary truth value.	One of the literals *true* or *false*	The literal *false*
char	Represents a single character literal. This is stored as a 16bit number.	Numerically: *0* to *65535* (but see notes below)	Numerical *0* or as a literal (see below) *'\0'*
int	Represents an integer (whole) number. This is stored as a 32bit 2's complement number.	*-2147483648* to *2147483647*	*0*
double	Represents a floating point number to "double precision". This is stored as a 64bit number.	*4.9E-324* to *1.7976931348623157E308*	*0*

Since the primitive types are built into Kenya, there is special syntax for declaring their 'values' in your programs.

A boolean has only one of two values, *true* or *false*, and so you can write *true* or *false* directly in your source code.

A character is technically a number, but usually you use it to store a single character, like *A*, *B* etc. To facilitate this, you can include a single character (in the ASCII range) directly in your source code by surrounding it in ' marks. You may also wish to include some special characters (like a newline or tab), and this is done using special *escape sequences*, these are explained in the table below:

Table 2.2. Kenya Character Escape Sequences

Escape Sequence	Description
’\0’	The character with the numeric value of 0
’\b’	Represents backspace
’\t’	Represents a tab character
’\n’	Represents a linefeed character (newline)
’\f’	Represents a formfeed character
’\r’	Represents a carriage return character
’\"’	Represents a double quote symbol (")
’\’’	Represents a single quote symbol (’)
’\\’	Represents a backslash symbol (\)

Because char’s, int’s and double’s all represent numbers, it is possible in certain circumstances to use one to represent another. Without any extra syntax, you can use a char anywhere where you would need an int or double, or an int anywhere where you would need a double. This is called a widening conversion and happens automatically. To go the other way, use the round, ceil, floor, and toChar functions that are built into Kenya.

Kenya Class Types

Kenya has built in one main class type, *String*. A String is used to represent a sequence of characters, and as such you may write a sequence of characters (including the escape sequences above) for a string directly in your source code, between quote (") marks.

Class Types can also be user defined, the syntax for doing this is:

```
class class_name {
        class_variable_declaration;
        class_variable_declaration = default_value;
}
```

A more grounded example would be a class to store an X,Y co-ordinate, that defaults at 10,3.

Example 2.1. Class Type Example

```
void main(){

        //here we create a new Coordinate object.
        Coordinate c;

        printCoordinate(c);

        //here we change the values of the variables
        // the Coordinate object holds.
        c.x = 3;
        c.y = 2;

        printCoordinate(c);
}

void printCoordinate( Coordinate c ){
        println( c.x + " " + c.y );
}

class Coordinate {
        int x = 0;
        int y = 3;
}
```

All Class types may also be assigned to the special value *null* (which is also the Class Type's default value when they are initialised in arrays or as elements of other classes). The literal *null* may therefore be written directly in your source code in conditionals, assignments, etc.

Note, Class Types in Kenya are fairly restrictive compared to those in Java. Mainly they cannot extend each other in a Type Heirarchy, and they cannot contain methods.

Kenya Built In Types Example

Below is an example program demonstrating the special syntax for Kenya's built-in types.

Example 2.2. Types Example

```
void main(){

        String hi = "Hello World";
        char c = '\t';
        boolean bob = false;
        int x = 30;

        if( hi == "Hello World" ){
                print( "The result was: " + true + '\t' + "\n\t\t ok.");
        }
}
```

Kenya Array Types

Arrays are special cases of the Class Types.

An array allows you to store multiple objects of the same type in an indexable, fixed size list.

To declare an array, place square bracket pairs *"[]"* after the type of the variable you wish to declare, for example:

```
void main(){
        int[] x;                // this declares an array of integers
        String[][] y;   // this declares an array of arrays of String.
}
```

with an array declared, it can then be assigned to either null, another array of the same type, or to a new array of a given size. When you assign an array to a new array of a given size, the value of each element of the array is the default for that type (as described above - the default value for an array is null). If you are declaring an array variable, you can also initialise it with known values;

Example 2.3. Array Initialisation

```
void main(){

        int[] x = null;
                // initialise the array to null.


        String[] y = new String[10];
                /* initialise the array with 10
                String objects each with the
                value null. */


        boolean[][] b = new boolean[3][3];
                /* initialise the array with
                   3 arrays each containing 3
                   booleans starting with the
                   default value of false. */


        String[] initStrings = { "Happy", "Flowers" };
                /* initialise the array with two
                   elements, with the values "Happy"
                   and "Flowers" */


        int[][] initInts = { { 0, 1 } , { 0, 1 } };
                /* initialise the array with two
```

```
                    arrays that both have 0 as the
                    first element, and 1 as the second
                */
        }
```

To be able to interrogate an array to get or set the values of what it containts, you index the array with a mathematical expression.

Remember, an array has a fixed length once it has been declared, and you cannot assign or read outside of that length. Also, *array indexes start at 0*, i.e. an array of length 10 has indexes 0,1,2,3,4,5 ... 9, but index 10 is out of range. To know how long an array is, you can interrogate it for its length; as shown below:

Example 2.4. Array Indexing and Length Interrogation

```
        void main(){

                int[] x = new int[5];
                // x.length returns the length of the array as an int

                for( int i = 0 ; i < x.length ; i++ ){
                        x[i] = i * i;
                }

                println("Squares");
                for( int i = 0 ; i < x.length ; i++ ){
                        println( "The number: " + i + " squared is: " + x[i] );
                }


                println("Powers");
                double[][] powers = new double[10][3];
                for( int i = 0 ; i < powers.length ; i++ ){
                        for(int j = 0 ; j < powers[i].length ; j++ ){
                                powers[i][j] = pow( i, j );
                        }
                }

                double[][] powers = new double[10][3];
                for( int i = 0 ; i < powers.length ; i++ ){
                        for(int j = 0 ; j < powers[i].length ; j++ ){
                                powers[i][j] = pow( i, j );
                        }
                }

        }
```

Chapter 3. Kenya Built In Functions

Tristan Allwood

Table 3.1. Kenya Built In Numerical Functions

Prototype(s)	Description	Example

`int abs(int param);`
`double abs(double param);`
	Returns the absolute value of a number, e.g. -1 becomes 1	`void main(){`
		`  int i = -1;`
		`  // prints out 1;`
		`  println( abs( i ) );`
		
		`  double d = 0.3;`
		`  // prints out 0.3;`
		`  println( abs( d ) );`
		`}`

`int ceil(double param);`	Returns the ceiling of a number (i.e. rounds it *up*)	`void main(){`
		`  // prints out 3;`
		`  println( ceil( 2.4 ) );`
		
		`  // prints out -2;`
		`  println( ceil( -2.4) );`
		`}`

`double exp(double param);`	Returns Euler's number e raised to the power of a double value	`void main(){`
		`  // prints out e`
		`  // to-the-power-of 3`
		`  println( exp( 3 ) );`
		`}`

`int floor(double param);`	Returns the floor of a number (i.e. rounds it *down*)	`void main(){`
		
		`  // prints out 2;`
		`  println( floor( 2.4 ) );`
		
		`  // prints out -3;`
		`  println( floor( -2.4) );`
		`}`

`double log(double param);`	Returns the natural logarithm (base e) of a double value	`void main(){`
		`  //prints out log base e of 3`
		`  println( log( 3 ) );`
		`}`

Prototype(s)	Description	Example

`double pow(double a, double b);`	Returns the *first* argument raised to the power of the *second* argument	`void main(){`
		`  //prints out 8.0`
		`  println( pow( 2, 3) );`
		`}`

`double random();`	Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0. Returned values are chosen pseudorandomly with (approximately) uniform distribution from that range.	`void main(){`
		`  //prints out 10 random numbers`
		`  //between 0 and 10`
		`  for(int i=0;i<10;i++){`
		`  println( random() * 10 );`
		`  }`
		`}`

`int round(double param);`	Returns the closes int value to the argument	`void main(){`
		`  //prints out 2`
		`  println( round( 2.1 ) );`
		
		`  //prints out 3`
		`  println( round( 2.5 ) );`
		`}`

`double sqrt(double param);`	Returns the correctly rounded positive square root of a double value.	`void main(){`
		`  //prints out 10.0`
		`  println( sqrt( 100 ) );`
		`}`

Table 3.2. Kenya Built In Trigonometric Functions

Prototype(s)	Description	Example

`double sin(double angle);`	Returns the trigonometric sine of an angle (given in *radians*).	`const double PI = 3.141592653589793;`
		
		`void main(){`
		`  //computes roughly 0.5;`
		
		`  double angleInRadians = 30 * PI / 180;`
		`  println( sin( angleInRadians ) );`
		`}`

`double cos(double angle);`	Returns the trigonometric cosine of an angle (given in *radians*).	`const double PI = 3.141592653589793;`
		
		`void main(){`
		`  //computes roughly 0.5;`
		
		`  double angleInRadians = 60 * PI / 180;`
		`  println( cos( angleInRadians ) );`
		`}`

Prototype(s)	Description	Example

`double tan(double angle);`	Returns the trigonometric tangent of an angle (given in *radians*).	`const double PI = 3.141592653589793;`
		`const double ANGLE = 30.96375653;`
		`void main(){`
		
		`  //computes roughly 0.6;`
		
		`  double angleInRadians = ANGLE * PI / 180;`
		`  println( tan( angleInRadians ) );`
		`}`

`double asin(double a);`	Returns the arc sine of the parameter a (given in *radians*).	`const double PI = 3.141592653589793;`
		
		`void main(){`
		` //computes roughtly 30`
		
		` double angleInRadians = asin( 0.5 );`
		` println( angleInRadians * 180 / PI );`
		`}`

`double acos(double a);`	Returns the arc cosine of the parameter a (given in *radians*).	`const double PI = 3.141592653589793;`
		
		`void main(){`
		` //computes roughtly 60`
		
		` double angleInRadians = acos( 0.5 );`
		` println( angleInRadians * 180 / PI );`
		`}`

`double atan(double a);`	Returns the arc tangent of the parameter a (given in *radians*).	`const double PI = 3.141592653589793;`
		
		`void main(){`
		` //computes roughtly 30.96375653;`
		
		` double angleInRadians = atan( 0.6 );`
		` println( angleInRadians * 180 / PI );`
		`}`

Table 3.3. Kenya Built In Input-Output Functions

Prototype(s)	Description	Example

`void print(boolean a);`
`void print(char c);`
`void print(int i);`
`void print(double d);`
`void print(<CLASS_TYPE> act);`
	Prints out a (usually) sensible String representation of the provided argument to the standard output stream.	`void main(){`
		
		`  //prints out`
		`  // "the trueth hurts!"`
		
		`  String s = "the";`
		
		`  char c = '\n';`
		
		`  boolean b = true;`
		
		`  print(s);`
		`  print(' ');`
		`  print(b);`
		`  print("th hurts!");`
		`}`

`void println();`
`void println(boolean a);`
`void println(char c);`
`void println(int i);`
`void println(double d);`
`void println(<CLASS_TYPE> act);`
	Prints out a (usually) sensible string represntation of the provided object to the standard output stream, followed by a newline symbol.
	
	The no-argument form of this method simply prints out a newline character.
		`void main(){`
		
		`  /* prints out:`
		`the`
		`true`
		
		`th hurts!`
		`  */`
		
		`  String s = "the";`
		
		`  char c = '\n';`
		
		`  boolean b = true;`
		
		`  println(s);`
		`  println(b);`
		`  println();`
		`  println("th hurts!");`
		`}`

`char read();`	Returns the next non-whitespace (i.e. non space,tab, or newline) char on the input stream.	`void main(){`
		
		`//prints a non-whitespace`
		`// char in a box`
		`  println("Enter a character");`
		
		`  char c = read();`
		
		`  println("###");`
		`  println("#" + c + "#");`
		`  println("###");`
		`}`

Prototype(s)	Description	Example

`char readChar();`	Returns the next char on the input stream.	`void main(){`
		
		`//prints a char in a box.`
		
		`  println("Enter a character");`
		
		`  char c = readChar();`
		
		`  println("###");`
		`  println("#" + c + "#");`
		`  println("###");`
		`}`

`int readInt();`	Skipping any whitespace on the input stream, attempts to parse the next characters as an integer, and returns that value.	`void main(){`
		
		`//asks for two numbers`
		`//and prints them out`
		`//in descending order.`
		
		`  println("Enter 2 integers");`
		
		`  int a = readInt();`
		
		`  int b = readInt();`
		
		`  if(a > b){`
		`    print(a + "," + b);`
		`  }else{`
		`    print(b + "," + a);`
		`  }`
		
		`}`

`double readDouble();`	Skipping any whitespace on the input stream, attempts to parse the next characters as a double, and returns that value.	`void main(){`
		
		`//asks for two numbers`
		`//and prints them out`
		`//in descending order.`
		
		`  println("Enter 2 numbers");`
		
		`  double a = readDouble();`
		
		`  double b = readDouble();`
		
		`  if(a > b){`
		`    print(a + "," + b);`
		`  }else{`
		`    print(b + "," + a);`
		`  }`
		
		`}`

Prototype(s)	Description	Example

`String readString();`	Skipping whitespace (and using whitespace as a delimeter), reads the next String from standard input.	`void main(){`
		` println("Please enter your name");`
		
		` String s = readString();`
		
		` println("Hello " + s );`
		`}`

`boolean isEOF();`	Returns true iff the end of file has been reached on standard input.	`void main(){`
		` println("Please enter your" +`
		`  " full name" +`
		`  ", then press EOF");`
		
		` String s = "";`
		` while(!isEOF()){`
		`  s = s + " " + readString();`
		` }`
		
		` println("Hello" + s);`
		
		`}`

To help illustrate the I/O Functions better, here is a larger example involving several of them together:

Example 3.1. Input-Output Built In Functions

```
void main(){

    println("Please enter your name: ");

    String name = readString();

    println("Hello " + name);
    println("Would you like to play a game? [y/n]");

    char c = read();

    if( c == 'y' || c == 'Y' ){
        playGame();
    }

        println("I'm now going to say what you say " +
                        "until you stop saying what you say.");
        while(!isEOF()){
                println(readString());
        }
}

void playGame(){
         int val = round( random() * 10);

        println("Guess the number between 0 and 10 ");
```

```
        int guess = readInt();

        if( guess == val ){
            println("Correct");
        }else{
            println("Wrong, the answer was " + val );
    }
}
```

Table 3.4. Kenya Utility Functions

Prototype	Description	Example(s)

`void arraycopy(boolean[] source, boolean[] dest);`
`void arraycopy(char[] source, char[] dest);`
`void arraycopy(int[] source, int[] dest);`
`void arraycopy(double[] source, double[] dest);`
`void arraycopy(String[] source, String[] dest);`
`void arraycopy(<CLASS_TYPE>[] source, <CLASS_TYPE>[] dest);`
	Performs a shallow copy of the contents of the source array into the dest array. If the arrays are not the same length, the elements are copied from the source to the dest at the same respective indexes, with not copying elements from the end of source (if source is longer than dest), or by padding dest with nulls/0/false (if dest is longer).	`void main() {`
		`  int[] a = { 1,2, 3, 4, 5 };`
		`  int[] b = new int[5];`
		`  arraycopy(a,b);`
		`// prints out 5`
		`  println(b[4]);`
		`  boolean[] c = { true, true };`
		`  boolean[] d = new boolean[3];`
		`  arraycopy(c,d);`
		`// prints out false`
		`  println(d[2]);`
		`}`

`String charsToString(char[] source);`	Appends together the elements of the array and returns a String built from the elements.	`void main(){`
		`  char[] myChars = { 'h','e','l','l','o',' ','w','o','r','l','d' };`
		`  String helloWorld = charsToString(myChars);`
		`// prints "hello world"`
		`  println(helloWorld);`
		`}`

`char[] stringToChars(String argument);`	Takes the String argument and returns an array of char's representing the constituent characters in the String.	`void main(){`
		`  char[] myChars = stringToChars("hello world!");`
		`  println(myChars[myChars.length - 1] );`
		`}`

Prototype	Description	Example(s)

`char intToChar(int param);`	Converts its int param to its ascii/char equivalent. E.g. 65 is the ascii for 'A'.	`void main(){`
		`  char c = intToChar(65);`
		`// prints 'A'`
		`  println(c);`
		`}`

`int charToInt(char param);`	Converts its character param into its ascii / char representation. E.g. 'A' becomes 65.	`void main() {`
		`  int x = charToInt('A');`
		`  //prints 65`
		`  println(x);`
		`}`

`Enum enumSucc(Enum param);`	Returns the next enum element after the element supplied	`enum Language {`
		`  Erlang, Scala, Ruby, Smalltalk;`
		`}`
		`void main() {`
		`  Language lang = Language.Scala;`
		`  Language nextLang = enumSucc(lang);`
		`  // prints Ruby`
		`  println(nextLang);`
		`}`

Chapter 4. Kenya Advanced Language Features

Tristan Allwood

Table of Contents

With the introduction of Java 1.5, Kenya has now added some of the new Java 1.5 features, namely *generics*, *enumerated types* and *autoboxing*.

Enumerated Types

An enumerated type is a special class type that can only take one of a limited number of values and (in Kenya) does not have any member variables.

They syntax for declaring an enumerated type is:

```
enum TYPE_NAME {
        child1, child2, child3;
}
```

With an enumerated type defined, you can then create variables which have the value of one of the children (or null):

```
void main() {
        Cows c = Cows.Daisy;
        println(c);
}

enum Cows {
        Daisy, Maizy, Haizy;
}
```

One of the most useful things you can do with an enumerated type is use it for a switch variable. In switch cases, you must remember *not* to put the type-name as a prefix for the children.

Example 4.1. Enumerated Types and Switch Example

```
void main() {
        Dogs d = Dogs.Scoobie;

        printDetails( d );

}

void printDetails( Dogs d ){
        switch ( d ){
                case Lassie:{
                        println( d + " come home!");
                        break;
                }
                case Scoobie:{
                        println( d + "-dooby-do!" );
                        break;
                }
                case Timmy:{
                        println( d );
                        break;
                }
        }
}

enum Dogs{
        Lassie, Scoobie, Timmy;
}
```

Autoboxing

This is a feature that is mainly motivated by the use of Generics (below).

Kenya has four class types built in other than String, namely, Boolean, Character, Integer and Double.

On their own they are of very little value, they have no variables you can get out of them, and don’t do anything.

However you can assign them to values of the type of their respective primitive types (boolean, char, int, double), this means you have in Kenya a class type that can be used like a primitive type.

There are some things to beware of with this (e.g the class types can be assigned to null which causes a *NullPointerException* if you try to use it in a primitive type context.)

```
void main(){

        int i = 3;

        Integer ii = i;

        println(ii);
}
```

Generics

Generics allow you to create a type-safe way of storing a 'generic' class type (and thanks to autoboxing primitive types aswell).

On methods, you can define *template parameters* in the following way:

```
< Template_Paramter, TemplateParam2 > returnType methodName( arguments ... ) {

}
```

The template parameters can now be used anywhere in that method (including the arguments and return type) as normal class type 'types'.

To see an application of this, consider the following method that returns the third element of (almost) any array type:

```
void main(){

        String[] s = { "hi", "how", "are", "you" };

        String third = getElement3( s );

        println( third );
}

< T > T getElement3( T[] array ){
        return array[2];
}
```

I say almost any array type because this won't work on arrays of primitive types, but if you create an array of Integer (or the other class-type equivalent) then you can get the same effect:

```
void main(){
        Integer[] ints = { 1,2,3,4,5};

        int third = getElement3( ints );

        println( third );
}

< T > T getElement3( T[] array ){
        return array[2];
}
```

Template parameters can also be used in classes, below is a class which stores a pair of Type-Parameterd objects:

```
class Pair<T>{
        T a;
        T b;
}
```

To create an instance of this class, we must say what the type-parameters are bound to. Note, in Kenya if you use a primitive type as the binding type, it is internally automatically converted up to the class type for you; i.e.:

```
void main() {
        Pair<String> twoStrings;
        twoStrings.a = "hello";
        twoStrings.b = "world";

        //This is equivalent to Pair<Integer>
        Pair<int> twoInts;
        twoInts.a = 3;
        twoInts.b = 4;
}

class Pair<T>{
        T a;
        T b;
}
```

Advanced Kenya Features Examples

This is a program which acts as a reverse polish notation calculator, there are two versions, one using Linked Lists, the other uses Arrays.

Example 4.2. Advanced Kenya Features Example - With Linked Lists.

```
/**
* Tristan Allwood (toa02 AT doc.ic.ac.uk)
* Reverse Polish Notation calculator example in Kenya
* Demonstrating enumerated types, generics, autoboxing
*
* This version uses Linked-List based stacks.
*
* Input on std input an RPN calculation (e.g.
* 1 2 + 3 * means 1 + 2 * 3
* 4 2 / means 4 / 2
* and on std out printed is the result ( e.g. 9 and 2 above ).
* Error cases are not handled gracefully here, esp no input = crash.
* Operators are +, -, *, /, % everything else that is not a number [0-9] is
* interpreted as ignorable whitespace.
*/

/**
* Program entry point, prints a welcome and then calls necessary methods
* to run the program
*/
void main(){

    //Print a welcome
    println("Welcome to the calculator!");
    println("Please enter a calculation to be performed in reverse polish notation");

    //Build the stacks of numbers and operators
    Pair<Stack<int>,Stack<Operator>> stacks = buildStacks();

    //Calculate the result from these
    int result = mainLoop(stacks.b,stacks.a);

    //Print out the result
    println( "The answer is: " + result );
}

/**
* An enumeration of the Operators / Elements of the processed input stack.
* Number means that that element is a number ( found in the numbers stack )
*/
enum Operator{
    Plus, Minus, Times, Divide, Modulo, Number;
```

```
}

/**
* Processes a stack of operators and a stack of numbers and uses them
* to calculate the result of the calculation.
*/
int mainLoop( Stack<Operator> operators, Stack<int> numbers ){
    Stack<int> tmpNums;


    while( hasNext(operators) ){
        Operator op = pop(operators);

        if ( op == Operator.Number ) {
            push( tmpNums, pop(numbers) );
        } else {
            int snd = pop(tmpNums);
            int fst = pop(tmpNums);

            switch( op ) {
                case Plus:{
                    push(tmpNums, fst + snd);
                    break;
                }
                case Minus:{
                    push(tmpNums, fst - snd);
                    break;
                }
                case Times:{
                    push( tmpNums, fst * snd);
                    break;
                }
                case Divide:{
                    push( tmpNums, fst / snd);
                    break;
                }
                case Modulo:{
                    push( tmpNums, fst % snd );
                    break;
                }
            }
        }
    }

    return pop(tmpNums);
}


/**
* Builds the safer Stacks of operators and ints by parsing standard input.
*/
Pair< Stack<int>, Stack<Operator> > buildStacks(){
    Stack<int> numStack;
    Stack<Operator> wholeStack;

    Stack<char> inputStack = getInputStack();
```

```
    for( char c = pop(inputStack) ; hasNext( inputStack ) ; c = pop(inputStack) ){

        if( '0' <= c & c <= '9' ){
            push( inputStack, c );
            int value = readNumber( inputStack );
            push(numStack, value );
            push(wholeStack, Operator.Number );
        } else {

            switch( c ){
                case '+':{
                    push( wholeStack, Operator.Plus );
                    break;
                }
                case '-':{
                    push( wholeStack, Operator.Minus );
                    break;
                }
                case '/':{
                    push( wholeStack, Operator.Divide );
                    break;
                }
                case '*':{
                    push( wholeStack, Operator.Times );
                    break;
                }
                case '%':{
                    push( wholeStack, Operator.Modulo );
                    break;
                }
                default:{
                    /* any other letter is just ignored */
                }
            }//switch
        } //else
    }//for

    Pair< Stack<int>, Stack<Operator> > retVal;
    reverseStack(numStack);
    reverseStack(wholeStack);
    retVal.a = numStack;
    retVal.b = wholeStack;

    return retVal;

}//method


/**
* Converts a number representation on the stack (e.g. [1,2,3] )
* into its real value ( 123 )
*/
int readNumber( Stack<char> source ){
    int number = 0;
    char c = pop(source);
    while( '0' <= c & c <= '9' ){
        number = number * 10 + ( c - '0' );
```

```
        c = pop(source);
    }
    push( source , c );

    return number;
}

/**
* Turns standard input into a stack of characters.
*/
Stack<char> getInputStack(){
    Stack<char> theStack;
    for( char c = read() ; c != toChar(-1) ; c = read() ){
        push(theStack, c);
    }

    reverseStack(theStack);

    return theStack;
}




/* ------------------------- */
/* *** Pair utility class *** */
/* ------------------------- */
class Pair<T,Q> {
    T a;
    Q b;
}

/* ---------------------------------------- */
/* *** General Stack Library Stuff Below *** */
/* ---------------------------------------- */

/**
* Reverses the given stack.
*/
<T> void reverseStack( Stack<T> stack){

    Stack<T> out;

    while( hasNext(stack) ){
        push( out, pop(stack) );
    }

    stack.head = out.head;

}

/**
* Prints out the given stack.
*/
<T> void printStack(Stack<T> stack){
    LinkedList<T> top = stack.head;
```

```
    print("[ " );
    while( top != null ) {
        print( top.node + " ");
        top = top.next;
    }
    print("]");
}

/**
* Returns true iff the given stack has a "next" element
*/
<T> boolean hasNext( Stack<T> stack ){
    return stack.head != null;
}

/**
* pushes a value onto the top of the given stack
*/
<T> void push( Stack<T> stack, T value ){
    LinkedList<T> top;
    top.node = value;
    top.next = stack.head;
    stack.head = top;
}

/**
* pops a value off the top of a given stack
*/
<T> T pop( Stack<T> stack ){
    T value = stack.head.node;
    stack.head = stack.head.next;
    return value;
}

/**
* A Stack class ( holds a pointer to a LinkedList )
*/
class Stack<T>{
    LinkedList<T> head;
}

/**
* A LinkedList class ( used to back a Stack )
*/
class LinkedList<T>{
    LinkedList<T> next;
    T node;
}
```

Example 4.3. Advanced Kenya Features - With Arrays

```
/**
* Tristan Allwood (toa02 AT doc.ic.ac.uk)
* Reverse Polish Notation calculator example in Kenya
* Demonstrating enumerated types, generics, autoboxing
*
* This version uses Array based stacks.
```

```
*
* Input on std input an RPN calculation (e.g.
* 1 2 + 3 * means 1 + 2 * 3
* 4 2 / means 4 / 2
* and on std out printed is the result ( e.g. 9 and 2 above ).
* Error cases are not handled gracefully here, esp no input = crash.
* Operators are +, -, *, /, % everything else that is not a number [0-9] is
* interpreted as ignorable whitespace.
*/

const int BUFFER_SIZE = 30; // how big a buffer to use for the arrays.

/**
* Program entry point, prints a welcome and then calls necessary methods
* to run the program
*/
void main(){

    //Print a welcome
    println("Welcome to the calculator!");
    println("Please enter a calculation to be performed in reverse polish notation");

    //Build the stacks of numbers and operators
    Pair<Stack<int>,Stack<Operator>> stacks = buildStacks();

    //Calculate the result from these
    int result = mainLoop(stacks.b,stacks.a);

    //Print out the result
    println( "The answer is: " + result );
}

/**
* An enumeration of the Operators / Elements of the processed input stack.
* Number means that that element is a number ( found in the numbers stack )
*/
enum Operator{
    Plus, Minus, Times, Divide, Modulo, Number;
}

/**
* Processes a stack of operators and a stack of numbers and uses them
* to calculate the result of the calculation.
*/
int mainLoop( Stack<Operator> operators, Stack<int> numbers ){
    Stack<int> tmpNums = newStack( new Integer[BUFFER_SIZE]);


    while( hasNext(operators) ){
        Operator op = pop(operators);

        if ( op == Operator.Number ) {
            push( tmpNums, pop(numbers) );
        } else {
            int snd = pop(tmpNums);
            int fst = pop(tmpNums);
```

```
            switch( op ) {
                case Plus:{
                    push(tmpNums, fst + snd);
                    break;
                }
                case Minus:{
                    push(tmpNums, fst - snd);
                    break;
                }
                case Times:{
                    push( tmpNums, fst * snd);
                    break;
                }
                case Divide:{
                    push( tmpNums, fst / snd);
                    break;
                }
                case Modulo:{
                    push( tmpNums, fst % snd );
                    break;
                }
            }
        }
    }

    return pop(tmpNums);
}


/**
* Builds the safer Stacks of operators and ints by parsing standard input.
*/
Pair< Stack<int>, Stack<Operator> > buildStacks(){
    Stack<int> numStack = newStack( new Integer[ BUFFER_SIZE] );
    Stack<Operator> wholeStack = newStack( new Operator[BUFFER_SIZE] );

    Stack<char> inputStack = getInputStack();

    for( char c = pop(inputStack) ; hasNext( inputStack ) ; c = pop(inputStack) ){

        if( '0' <= c & c <= '9' ){
            push( inputStack, c );
            int value = readNumber( inputStack );
            push(numStack, value );
            push(wholeStack, Operator.Number );
        } else {

            switch( c ){
                case '+':{
                    push( wholeStack, Operator.Plus );
                    break;
                }
                case '-':{
                    push( wholeStack, Operator.Minus );
                    break;
                }
                case '/':{
```

```
                    push( wholeStack, Operator.Divide );
                    break;
                }
                case '*':{
                    push( wholeStack, Operator.Times );
                    break;
                }
                case '%':{
                    push( wholeStack, Operator.Modulo );
                    break;
                }
                default:{
                    /* any other letter is just ignored */
                }
            }//switch
        } //else
    }//for

    Pair< Stack<int>, Stack<Operator> > retVal;
    reverseStack(numStack);
    reverseStack(wholeStack);
    retVal.a = numStack;
    retVal.b = wholeStack;

    return retVal;

}//method


/**
* Converts a number representation on the stack (e.g. [1,2,3] )
* into its real value ( 123 )
*/
int readNumber( Stack<char> source ){
    int number = 0;
    char c = pop(source);

    while( '0' <= c & c <= '9' ){
        number = number * 10 + ( c - '0' );
        c = pop(source);
    }
    push( source , c );

    return number;
}

/**
* Turns standard input into a stack of characters.
*/
Stack<char> getInputStack(){
    Stack<char> theStack = newStack( new Character[BUFFER_SIZE] );
    for( char c = read() ; c != toChar(-1) ; c = read() ){
        push(theStack, c);
    }

    reverseStack(theStack);
```

```
    return theStack;
}



/* ------------------------- */
/* *** Pair utility class *** */
/* ------------------------- */
class Pair<T,Q> {
    T a;
    Q b;
}

/* ---------------------------------------- */
/* *** General Stack Library Stuff Below *** */
/* ---------------------------------------- */

/**
* Since in generics you can't create an array of
* Template type at runtime, you have to provide
* the backing buffer when you create a new Stack.
* This is a method that emulates the constructor.
*/
<T> Stack<T> newStack( T[] backBuffer ){
    Stack<T> newStack;
    newStack.store = backBuffer;
    return newStack;
}

/**
* Reverses the given stack.
*/
<T> void reverseStack( Stack<T> stack){
    T[] array = stack.store;
    int max = stack.cPos - 1;

    for( int i = 0 ; i < max ; i++ ){
        T tmp = array[i];
        array[i] = array[max];
        array[max] = tmp;
        max--;
    }

}

/**
* Prints out the given stack.
*/
<T> void printStack(Stack<T> stack){
    print("[ ");
    for( int i = 0 ; i < stack.cPos ; i++ ){
        print( stack.store[i] + " " );
    }
    println("]");
}
```

```
/**
* Returns true iff the given stack has a "next" element
*/
<T> boolean hasNext( Stack<T> stack ){
    return stack.cPos > 0;
}

/**
* pushes a value onto the top of the given stack
*/
<T> void push( Stack<T> stack, T value ){
    stack.store[stack.cPos] = value;
    stack.cPos++;
}

/**
* pops a value off the top of a given stack
*/
<T> T pop( Stack<T> stack ){
    stack.cPos--;
    T value = stack.store[stack.cPos];
    stack.store[stack.cPos] = null; // just for safety
    return value;
}

/**
* A Stack class
*/
class Stack<T>{
    T[] store;
    int cPos;
}
```
