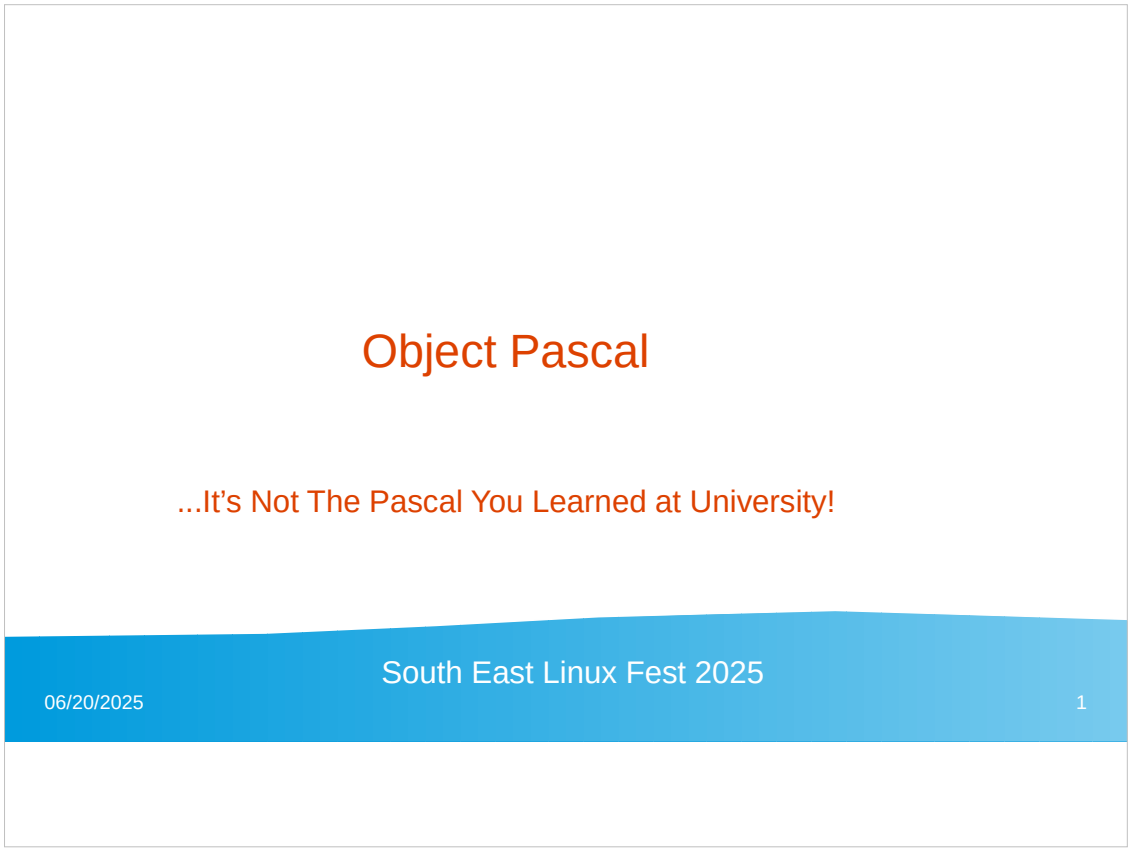


Good afternoon and welcome to the South East Linux Fest!

Before I get started, I'd like to thank the many volunteers that give their time so we can have the South East Linux Fest and so that I can have this chance to tell you about Object Pascal.

If you had Pascal in high school or college, Object Pascal is a different beast, but it has a familiar syntax.

Introduction

- Who is Brad Whitehead?
- Chief Scientist for Formularity, an electronic forms company
- Formerly, he was a Partner and Master Technology Architect with Accenture
- Has been an Object Pascal developer since 1995
- Brad holds a BS from Carnegie Mellon University and an MS from the University of Liverpool
- He can be reached at brad.whitehead@formularity.com.
- Slides are available from https://formularity.com

Let me introduce myself. I'm Brad Whitehead and I'm the Chief Scientist for Formularity, a secure electronic enrollment company.

Before that I was a Partner and technology architect with the global consulting firm of Accenture.

I've been using Pascal since 1975 and Object Pascal since it originated in 1995

The slides for today's talk will be available for download from my company's website: Formularity.com, so no need to take pictures or notes. If you do want to take pictures, this is my better side ;-)

History

- Pascal was originally developed by Niklaus Wirth in 1970, based on Algol-W and Simula 67 (same inspiration for C++)
  - It was not object oriented
- Extremely popular in the 80's and 90's with Turbo-Pascal (Anders Hejlsberg) and Macintosh Pascal Workbench (MPW)
  - Anders later wrote both Delphi and C#
- Object orientation added by Apple (MPW) and Borland (Delphi)
- Free Pascal Compiler and Lazarus GUI IDE released in 1997
- There are now at least eight (8) multi-platform implementations

Pascal is the product of Niklaus Wirth, written in 1970. He based it on both Algol and Simula. While Simula was also the inspiration for "C with Classes", the language now known as C++, the original Pascal was not object or class-oriented.

History

- Pascal was originally developed by Niklaus Wirth in 1970, based on Algol-W and Simula 67 (same inspiration for C++)
  - It was not object oriented
- Extremely popular in the 80's and 90's with Turbo-Pascal (Anders Hejlsberg) and Macintosh Pascal Workbench (MPW)
  - Anders later wrote both Delphi and C#
- Object orientation added by Apple (MPW) and Borland (Delphi)
- Free Pascal Compiler and Lazarus GUI IDE released in 1997
- There are now at least eight (8) multi-platform implementations

While Wirth wrote Pascal as a teching language, it became very popular during the 1980s and '90s. The original Macintosh was programmed in Pascal.

Probably the most famous instance of Pascal was Borland's TurboPascal. At the time that most PC compilers cost $200 or more, Borland's TurboPascal cost $39.95 and as the name suggests, it was very fast!

Interesting enough, TurboPascal was written by Anders Hejlsberg, who later went to Microsoft and wrote C#.

History

- Pascal was originally developed by Niklaus Wirth in 1970, based on Algol-W and Simula 67 (same inspiration for C++)
  - It was not object oriented
- Extremely popular in the 80's and 90's with Turbo-Pascal (Anders Hejlsberg) and Macintosh Pascal Workbench (MPW)
  - Anders later wrote both Delphi and C#
- Object orientation added by Apple (MPW) and Borland (Delphi)
- Free Pascal Compiler and Lazarus GUI IDE released in 1997
- There are now at least eight (8) multi-platform implementations

In 1995, Apple and Borland collaborated to develop the object-oriented version of Pascal. Apple released their product as the Macintosh Programming Workbench, and Borland's product was called Delphi.

History

- Pascal was originally developed by Niklaus Wirth in 1970, based on Algol-W and Simula 67 (same inspiration for C++)
  - It was not object oriented
- Extremely popular in the 80's and 90's with Turbo-Pascal (Anders Hejlsberg) and Macintosh Pascal Workbench (MPW)
  - Anders later wrote both Delphi and C#
- Object orientation added by Apple (MPW) and Borland (Delphi)
- Free Pascal Compiler and Lazarus GUI IDE released in 1997
- There are now at least eight (8) multi-platform implementations

One of the problems with Delphi was that it only ran on Windows. In 1997, the open source community wrote a cross platform version of Object Pascal called appropriately enough, the Free Pascal Compiler.

History

- Pascal was originally developed by Niklaus Wirth in 1970, based on Algol-W and Simula 67 (same inspiration for C++)
  - It was not object oriented
- Extremely popular in the 80's and 90's with Turbo-Pascal (Anders Hejlsberg) and Macintosh Pascal Workbench (MPW)
  - Anders later wrote both Delphi and C#
- Object orientation added by Apple (MPW) and Borland (Delphi)
- Free Pascal Compiler and Lazarus GUI IDE released in 1997
- There are now at least eight (8) multi-platform implementations

The last time I checked, there were at least 8 cross-platform Object Pascal implementations, a mixture of both Free and Open Source software and closed source software.

Why Should You Care?

- Strong Typing – Prevents unintentional conversions, memory errors, and spelling errors
- Separate Interface and Implementation Sections – One set of Developers can code for a Unit/Library that is still being written
- Memory Safety – Object memory is reclaimed after it passes out of scope (no question of who frees memory) and all types can be set to Nil, avoiding “use after free” errors
- Compiles to machine code – executes as fast as C
- Syntax is very similar to Python
- “System” language with embeddable assembler

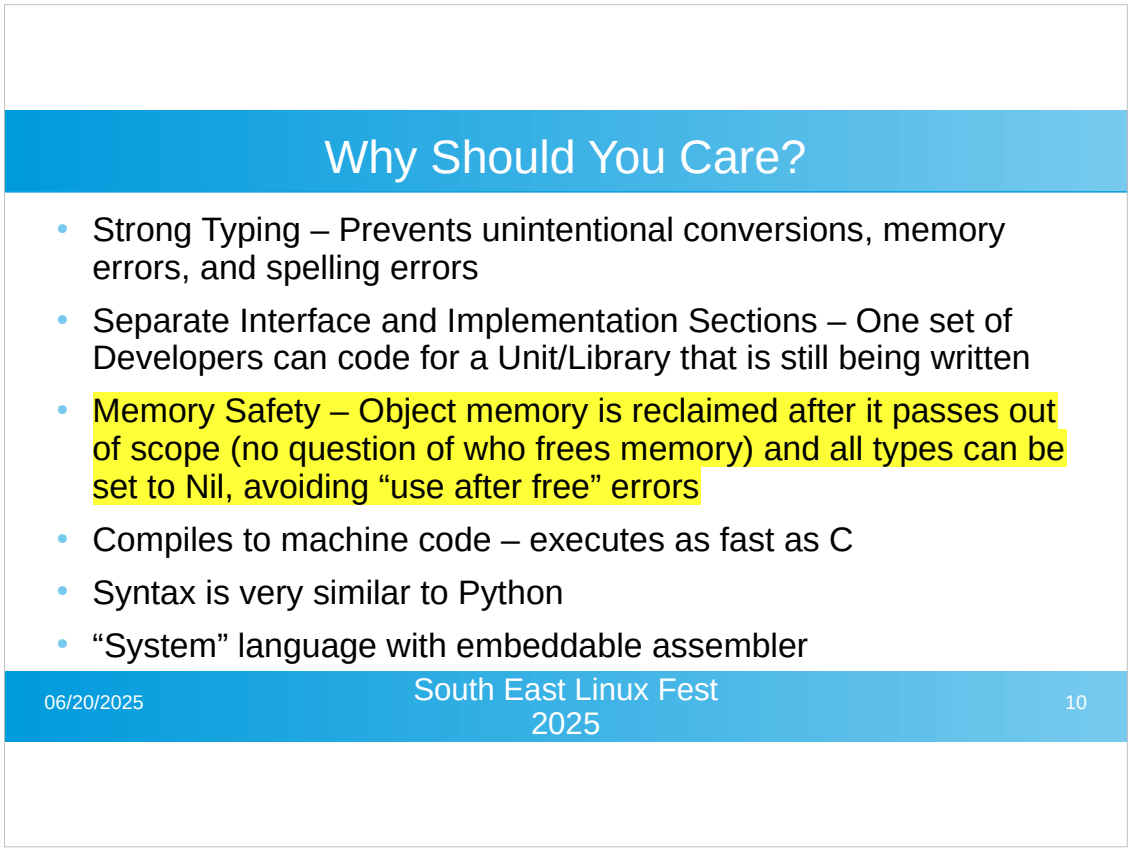
So, what does Object Pascal have to offer?

First off, Strong Typing. What this may seem like extra work, strong typing actually helps prevent conversion errors, memory errors, and typing errors. And, Pascal’s type system isn’t as annoying as Java’s or C++’s.

Why Should You Care?

- Strong Typing – Prevents unintentional conversions, memory errors, and spelling errors
- Separate Interface and Implementation Sections – One set of Developers can code for a Unit/Library that is still being written
- Memory Safety – Object memory is reclaimed after it passes out of scope (no question of who frees memory) and all types can be set to Nil, avoiding “use after free” errors
- Compiles to machine code – executes as fast as C
- Syntax is very similar to Python
- “System” language with embeddable assembler

Pascal programs are written in two parts, the interface section and the implementation section. The interface section corresponds to the export statements of some other languages. It also means that you can give the interface section to other developers and they can write code without waiting for you to finish the implementation. It also helps hide the “What” from the “How”.



One of the hot topics today is the Rust language and its memory safety. Memory management in C and C++ is like juggling razor blades. One slight slipup and … you leak memory or you have a “use after free” vulnerability.

Object Pascal uses both reference counting and null pointer detection to avoid memory errors. So, it’s memory safety with a familiar syntax, unlike Rust.

Why Should You Care?

- Strong Typing – Prevents unintentional conversions, memory errors, and spelling errors
- Separate Interface and Implementation Sections – One set of Developers can code for a Unit/Library that is still being written
- Memory Safety – Object memory is reclaimed after it passes out of scope (no question of who frees memory) and all types can be set to Nil, avoiding “use after free” errors
- Compiles to machine code – executes as fast as C
- Syntax is very similar to Python
- “System” language with embeddable assembler

The Free Pascal Compiler, as well as most other implementations are true compilers, generating the native machine code of the platform. So, no byte-code virtual machines, and full C-like speeds

Why Should You Care?

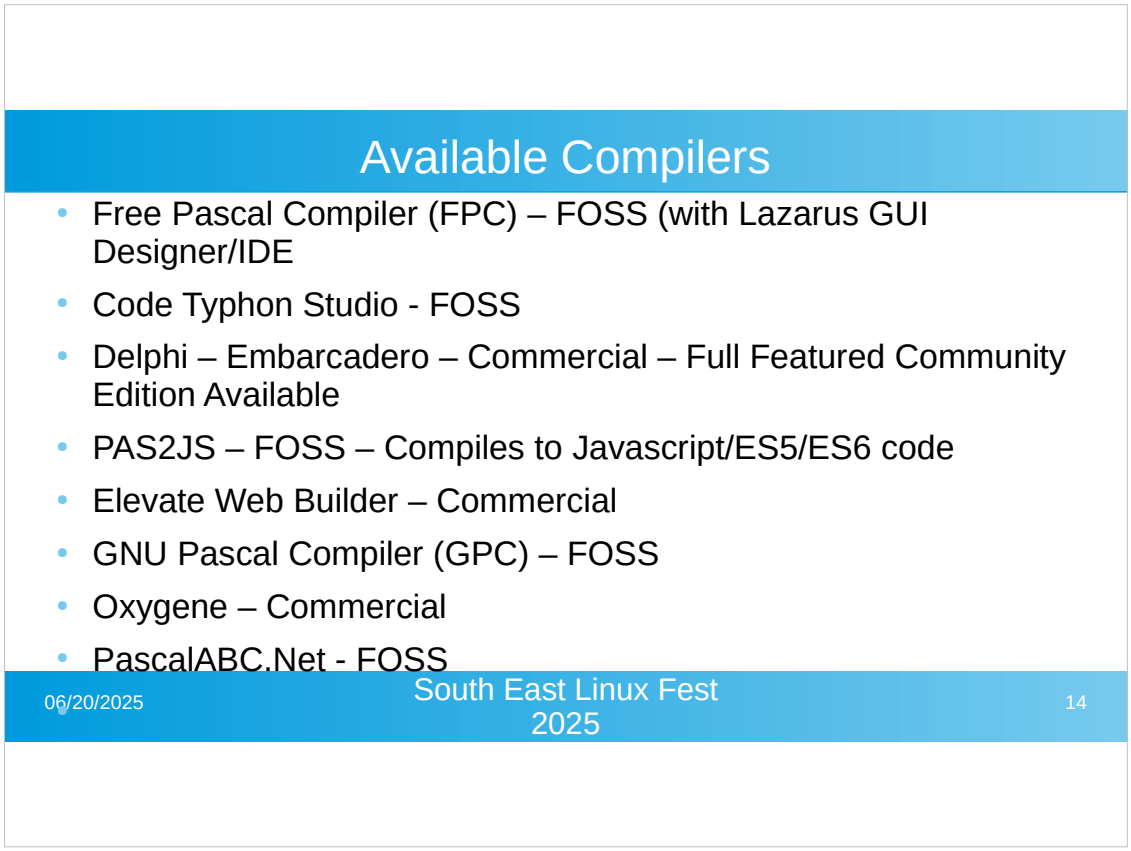
- Strong Typing – Prevents unintentional conversions, memory errors, and spelling errors
- Separate Interface and Implementation Sections – One set of Developers can code for a Unit/Library that is still being written
- Memory Safety – Object memory is reclaimed after it passes out of scope (no question of who frees memory) and all types can be set to Nil, avoiding “use after free” errors
- Compiles to machine code – executes as fast as C
- Syntax is very similar to Python
- “System” language with embeddable assembler

How many Python programmers do we have here today? One of the things you’ll notice when I show you some Object Pascal code later is the similarity to Python code. It’s very easy for a Python programmer to pick up Object Pascal and use its benefits of speed and true multi-threading. No Global Interpreter Locks!

Why Should You Care?

- Strong Typing – Prevents unintentional conversions, memory errors, and spelling errors
- Separate Interface and Implementation Sections – One set of Developers can code for a Unit/Library that is still being written
- Memory Safety – Object memory is reclaimed after it passes out of scope (no question of who frees memory) and all types can be set to Nil, avoiding “use after free” errors
- Compiles to machine code – executes as fast as C
- Syntax is very similar to Python
- “System” language with embeddable assembler

As I noted, Pascal was used as the system programming language for the original MacOS. Object Pascal can and is used to write operating system code. If you can’t do it in Pascal, it supports dropping down into an integrated assembler.



Here are some of the eight implementations of Object Pascal I'm familiar with. As you can see, over half are open source.

While the Delphi implementation isn't open source, they do have a free full-featured, time-unlimited Community Edition. Basically, you can use this Community Edition and sell the resulting software until you have annual sales of $5000. This is of interest because while the Free Pascal Compiler can cross compile to both Android and IOS, the Delphi version is much easier to use for cross-compilation.

Interpreters

- https://www.onlinegdb.com/online_pascal_compiler - Online
- https://www.tutorialspoint.com/compilers/online-pascal-compiler.htm - Online
- https://onecompiler.com/pascal - Online
- https://www.startpage.com/sp/search – Online
- DWScript - FOSS

There are also Object Pascal interpreters. The first 4 on this list are online development environments. The last is an interpreted version designed to be embedded in a Pascal program as a scripting language. No need use a different language like Lua for scripting. ;-)

Platforms

- AMD64/x86_64
- i386
- PowerPC
- PowerPC64
- SPARC
- SPARC64
- ARM
- AArch64
- MIPS, Motorola 68k
- AVR
- JVM
- RISC-V
- Extensa
- Z80
- ESP32

Here are some of the hardware platforms that Object Pascal runs on.

Operating Systems

- LINUX
- MacOS/iOS/Darwin
- FreeBSD and other BSD flavors
- Web Assembly (WASM)
- Windows (16/32/64 bit, CE, and native NT)
- DOS (16 bit, or 32 bit DPMI), OS/2
- AIX/VAX/HP-PA/zOS
- Android
- Raspian
- Haiku
- Nintendo GBA/DS/Wii/Switch
- AmigaOS
- MorphOS
- AROS
- Atari TOS
- various embedded platforms

And here are the operating systems.

Rich Flow Control

- IF – THEN - ELSE
- CASE – OF – ELSE (no “fall through”)
- FOR - DO
- WHILE – DO
- REPEAT – UNTIL
- FOR <Object> IN <Iteratable Object> (Arrays, Lists, Collections, Sets, Enumerations)
- Exception – Try – Except - Finally

Object Pascal has a rich set of flow controls.

Rich Flow Control

- IF – THEN - ELSE
- CASE – OF – ELSE (no “fall through”)
- FOR - DO
- WHILE – DO
- REPEAT – UNTIL
- FOR <Object> IN <Iteratable Object> (Arrays, Lists, Collections, Sets, Enumerations)
- Exception – Try – Except - Finally

Of particular note, the Case statement automatically does not fall through to subsequent cases, this eliminates a common error with some switch statements. It also has a “catch-all” Else condition.

Rich Flow Control

- IF – THEN - ELSE
- CASE – OF – ELSE (no “fall through”)
- FOR - DO
- WHILE – DO
- REPEAT – UNTIL
- FOR <Object> IN <Iteratable Object> (Arrays, Lists, Collections, Sets, Enumerations)
- Exception – Try – Except - Finally

Pascal has fixed iteration looping

Rich Flow Control

- IF – THEN - ELSE
- CASE – OF – ELSE (no “fall through”)
- FOR - DO
- WHILE – DO
- REPEAT – UNTIL
- FOR <Object> IN <Iteratable Object> (Arrays, Lists, Collections, Sets, Enumerations)
- Exception – Try – Except - Finally

Conditional looping at the start of the loop

Rich Flow Control

- IF – THEN - ELSE
- CASE – OF – ELSE (no “fall through”)
- FOR - DO
- WHILE – DO
- REPEAT – UNTIL
- FOR <Object> IN <Iteratable Object> (Arrays, Lists, Collections, Sets, Enumerations)
- Exception – Try – Except - Finally

And conditional looping at the end of the loop

Rich Flow Control

- IF – THEN - ELSE
- CASE – OF – ELSE (no “fall through”)
- FOR - DO
- WHILE – DO
- REPEAT – UNTIL
- FOR <Object> IN <Iteratable Object> (Arrays, Lists, Collections, Sets, Enumerations)
- Exception – Try – Except - Finally

Object Pascal data structures are iterable and can be used with a For-In statement.

Rich Flow Control

- IF – THEN - ELSE
- CASE – OF – ELSE (no “fall through”)
- FOR - DO
- WHILE – DO
- REPEAT – UNTIL
- FOR <Object> IN <Iteratable Object> (Arrays, Lists, Collections, Sets, Enumerations)
- Exception – Try – Except - Finally

Finally (and the pun is intended), Object Pascal uses exceptions for error handling and it has a Finally clause that is guaranteed to be executed. This is where you can close files and databases safely, even if your program crashes.

Primitive and Structured Data Types

- Integer, Word, and LongInt,
- Real, Single, Double, Extended (Exponential), Currency, and 80-bit Real
- Char, WideChar, ShortString, WideString, and ANSIString
- Boolean
- Static Array and DynamicArray
- Records
- Sets
- Dictionary
- List, and ClassList
- Pointers
- Variants
- Enumerations and SubRange
- Objects and Classes

Pascal has two types of data; the basic data structures with the usual various string and numeric types, as well as boolean.

Primitive and Structured Data Types

- Integer, Word, and LongInt,
- Real, Single, Double, Extended (Exponential), Currency, and 80-bit Real
- Char, WideChar, ShortString, WideString, and ANSIString
- Boolean
- Static Array and DynamicArray
- Records
- Sets
- Dictionary
- List, and ClassList
- Pointers
- Variants
- Enumerations and SubRange
- Objects and Classes

It also has a rich set of collective data types

Class/Object Notation

- Class TWall = Class(TObject); – Single Inheritance
  - Private
    - FBeers : array of Tbeer;
    - FnumberOfBeers : Integer;
  - Public
    - Function TellCount(): integer;
    - Procedure RemoveBeer(NoOfBeers : integer);
    - Constructor Create(NoOfBottles : integer);
  - Protected (Accessible Only to Derived Classes)
  - Property NumberOfBeers : Integer Read FNumberOfBeers Write TellCount();

The heart of Object Pascal is Classes. Here is an example of a Class in Object Pascal. In the religious war of multiple versus single inheritance, Pascal comes down on the side of single inheritance Here we are defining the Wall type and saying that it is derived from the TObject class. TObject is the most fundamental base class.

Class/Object Notation

- Class TWall = Class(TObject); – Single Inheritance
  - Private
    - FBeers : array of Tbeer;
    - FnumberOfBeers : Integer;
  - Public
    - Function TellCount(): integer;
    - Procedure RemoveBeer(NoOfBeers : integer);
    - Constructor Create(NoOfBottles : integer);
  - Protected (Accessible Only to Derived Classes)
  - Property NumberOfBeers : Integer Read FNumberOfBeers Write TellCount();

Pascal Classes have three types of data access restrictions. Anything under Private can only be accessed the Class itself. It's "hidden" from the rest of the program.

Class/Object Notation

- Class TWall = Class(TObject); – Single Inheritance
  - Private
    - FBeers : array of Tbeer;
    - FnumberOfBeers : Integer;
  - Public
    - Function TellCount(): integer;
    - Procedure RemoveBeer(NoOfBeers : integer);
    - Constructor Create(NoOfBottles : integer);
  - Protected (Accessible Only to Derived Classes)
  - Property NumberOfBeers : Integer Read FNumberOfBeers Write TellCount();

Here we have two data fields. The first is an array of Beers objects. Pascal arrays can hold any type of data. The second field is an integer. Both of these are under Private so they can only be read or written to by the Object itself.

Class/Object Notation

- Class TWall = Class(TObject); – Single Inheritance
  - Private
    - FBeers : array of Tbeer;
    - FnumberOfBeers : Integer;
  - Public
    - Function TellCount(): integer;
    - Procedure RemoveBeer(NoOfBeers : integer);
    - Constructor Create(NoOfBottles : integer);
  - Protected (Accessible Only to Derived Classes)
  - Property NumberOfBeers : Integer Read FNumberOfBeers Write TellCount();

The Public access control is just that. Anything under Public can be read, written, or called by any other part of the program.

Class/Object Notation

- Class TWall = Class(TObject); – Single Inheritance
  - Private
    - FBeers : array of Tbeer;
    - FnumberOfBeers : Integer;
  - Public
    - Function TellCount(): integer;
    - Procedure RemoveBeer(NoOfBeers: integer);
    - Constructor Create(NoOfBottles : integer);
  - Protected (Accessible Only to Derived Classes)
  - Property NumberOfBeers : Integer Read FNumberOfBeers Write TellCount();

Here's a public Function that tells us how many bottles of beer are on the wall. It returns an integer, the number of beers.

Class/Object Notation

- Class TWall = Class(TObject); – Single Inheritance
  - Private
    - FBeers : array of Tbeer;
    - FnumberOfBeers : Integer;
  - Public
    - Function TellCount(): integer;
    - Procedure RemoveBeer(NoOfBeers : integer);
    - Constructor Create(NoOfBottles : integer);
  - Protected (Accessible Only to Derived Classes)
  - Property NumberOfBeers : Integer Read FNumberOfBeers Write TellCount();

This is a Procedure. Functions and Procedures are virtually identical, but Procedures do not return anything. Here we are talking about a public procedure that can remove X number of beers from the Wall at a time.

Class/Object Notation

- Class TWall = Class(TObject); – Single Inheritance
  - Private
    - FBeers : array of Tbeer;
    - FnumberOfBeers : Integer;
  - Public
    - Function TellCount(): integer;
    - Procedure RemoveBeer(NoOfBeers : integer);
    - Constructor Create(NoOfBottles : integer);
  - Protected (Accessible Only to Derived Classes)
  - Property NumberOfBeers : Integer Read FNumberOfBeers Write TellCount();

Every Class has a constructor. This is a special procedure that's called when the Class is instantiated into an Object. It's responsible for setting all the initial conditions of the Object.

Class/Object Notation

- Class TWall = Class(TObject); – Single Inheritance
  - Private
    - FBeers : array of Tbeer;
    - FnumberOfBeers : Integer;
  - Public
    - Function TellCount(): integer;
    - Procedure RemoveBeer(NoOfBeers : integer);
    - Constructor Create(NoOfBottles : integer);
  - Protected (Accessible Only to Derived Classes)
  - Property NumberOfBeers : Integer Read FNumberOfBeers Write TellCount();

Here's the Protected access designator. Items under Protected can only be read, written, or called by a child Class derived from this parent Class.

Class/Object Notation

- Class TWall = Class(TObject); – Single Inheritance
  - Private
    - FBeers : array of Tbeer;
    - FnumberOfBeers : Integer;
  - Public
    - Function TellCount(): integer;
    - Procedure RemoveBeer(NoOfBeers : integer);
    - Constructor Create(NoOfBottles : integer);
  - Protected (Accessible Only to Derived Classes)
  - Property NumberOfBeers : Integer Read FNumberOfBeers Write TellCount();

Finally, Pascal Classes can have Properties. Properties are what's called "syntactic sugar". It's a short-hand way of making a Function or Procedure look like a Field. There can be a Function for Reading, or a Procedure for Writing to a Private Field. You can see, in this particular case, reading reads the private field directly, while writing uses the TellCount function.

Other Features

- Interfaces (COM and CORBA)
- Direct calls to and from C and C++ (without name-mangling)
- Exceptions for error handling, including ‘Finally’
- Concrete and Virtual Methods (VMT)
- Generics
- Nested and Anonymous Functions
- Pointers (including callbacks)
- Class Helpers
- Case Insensitive

Some other important features of Object Pascal:

Objects can simulate multiple inheritance through Interfaces. These Interfaces can conform to either Microsoft’s Common Object Model (COM) interface or the old Common Object Request-Broker Architecture (CORBA).

Other Features

- Interfaces (COM and CORBA)
- Direct calls to and from C and C++ (without name-mangling)
- Exceptions for error handling, including ‘Finally’
- Concrete and Virtual Methods (VMT)
- Generics
- Nested and Anonymous Functions
- Pointers (including callbacks)
- Class Helpers
- Case Insensitive

Pascal code can call into C functions or be called by C functions. When Name Mangling is turned off in C++, the same execution exchange is possible. So, you can use C and C++ libraries and Pascal libraries can be used by C and C++ programs.

Other Features

- Interfaces (COM and CORBA)
- Direct calls to and from C and C++ (without name-mangling)
- Exceptions for error handling, including ‘Finally’
- Concrete and Virtual Methods (VMT)
- Generics
- Nested and Anonymous Functions
- Pointers (including callbacks)
- Class Helpers
- Case Insensitive

I already covered Object Pascal’s exceptions for error handling

Other Features

- Interfaces (COM and CORBA)
- Direct calls to and from C and C++ (without name-mangling)
- Exceptions for error handling, including ‘Finally’
- Concrete and Virtual Methods (VMT)
- Generics
- Nested and Anonymous Functions
- Pointers (including callbacks)
- Class Helpers
- Case Insensitive

For the most part, Pascal methods are concrete and therefore very fast. However, if runtime polymorphism is required, the methods can be called through a Virtual Method Table.

Other Features

- Interfaces (COM and CORBA)
- Direct calls to and from C and C++ (without name-mangling)
- Exceptions for error handling, including ‘Finally’
- Concrete and Virtual Methods (VMT)
- Generics
- Nested and Anonymous Functions
- Pointers (including callbacks)
- Class Helpers
- Case Insensitive

I’m not a big fan of Generics, but if you want create collective data types through templates, Object Pascal has Generics for your use.

Other Features

- Interfaces (COM and CORBA)
- Direct calls to and from C and C++ (without name-mangling)
- Exceptions for error handling, including ‘Finally’
- Concrete and Virtual Methods (VMT)
- Generics
- Nested and Anonymous Functions
- Pointers (including callbacks)
- Class Helpers
- Case Insensitive

Object Pascal methods can be nested to any depth and you can define a method without associating a name with it. Again, I’m not sure that is as important to Pascal as it is to say Javascript.

Other Features

- Interfaces (COM and CORBA)
- Direct calls to and from C and C++ (without name-mangling)
- Exceptions for error handling, including ‘Finally’
- Concrete and Virtual Methods (VMT)
- Generics
- Nested and Anonymous Functions
- Pointers (including callbacks)
- Class Helpers
- Case Insensitive

Object Pascal has pointers to memory and to methods. The only thing it doesn’t permit is pointer arithmetic. This is to prevent memory access vulnerabilities.

Other Features

- Interfaces (COM and CORBA)
- Direct calls to and from C and C++ (without name-mangling)
- Exceptions for error handling, including ‘Finally’
- Concrete and Virtual Methods (VMT)
- Generics
- Nested and Anonymous Functions
- Pointers (including callbacks)
- Class Helpers
- Case Insensitive

Class Helpers are methods that can be added to an existing Class without having to have access to the original source code of the Class. Basically, extend an existing class you didn’t write.

Other Features

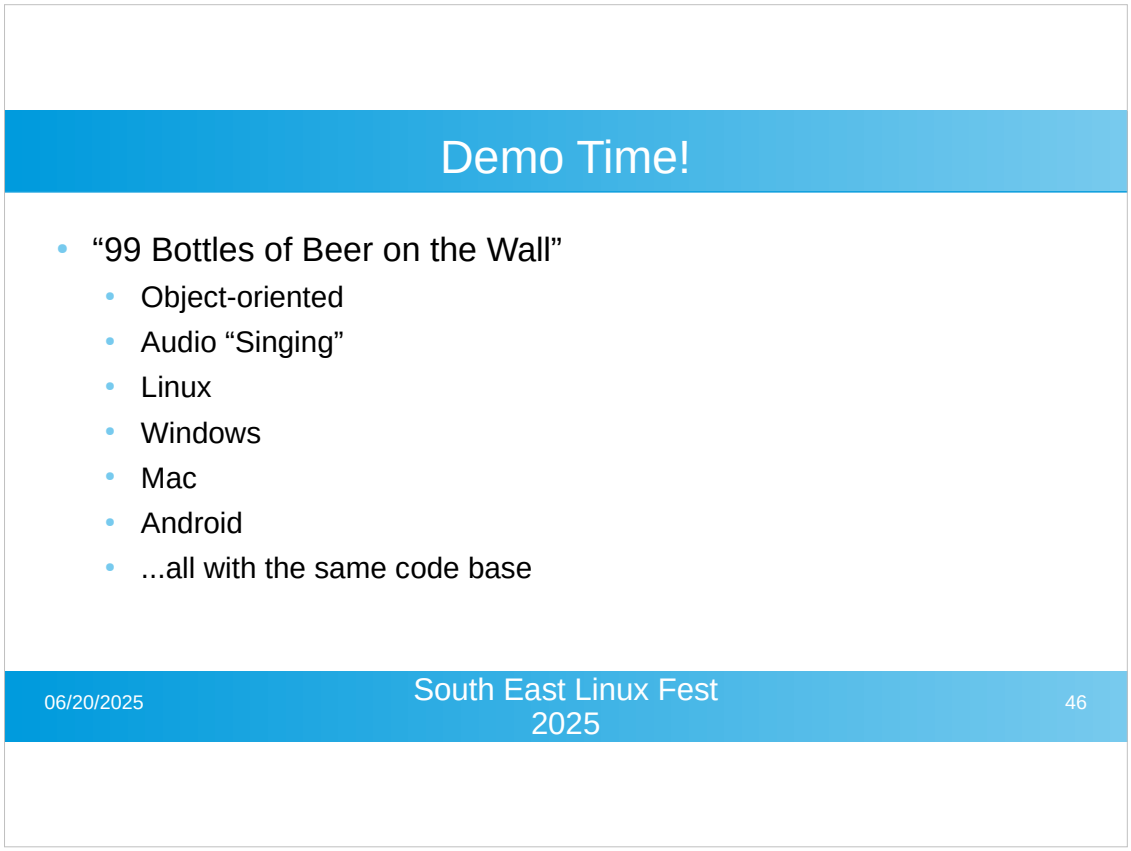
- Interfaces (COM and CORBA)
- Direct calls to and from C and C++ (without name-mangling)
- Exceptions for error handling, including ‘Finally’
- Concrete and Virtual Methods (VMT)
- Generics
- Nested and Anonymous Functions
- Pointers (including callbacks)
- Class Helpers
- Case Insensitive

Finally, Object Pascal is case insensitive. Some people might consider that a negative, but I strive to write error-free code. Having the same variables or methods differentiated only by the shift key seems to me like an error waiting to happen.

Learning References

- “Modern Object Pascal – Introduction for Programmers” (castle-engine.io/modern_pascal)
- Free Pascal Wiki (https://wiki.freepascal.org)
- Tech & Swords Youtube videos – Marcus Fernstrom

Here are three excellent sources for learning Object Pascal. I highlighted the first because it’s written for existing programmers that want to know how to use Object Pascal and don’t need “this is what a variable is” type lesson.

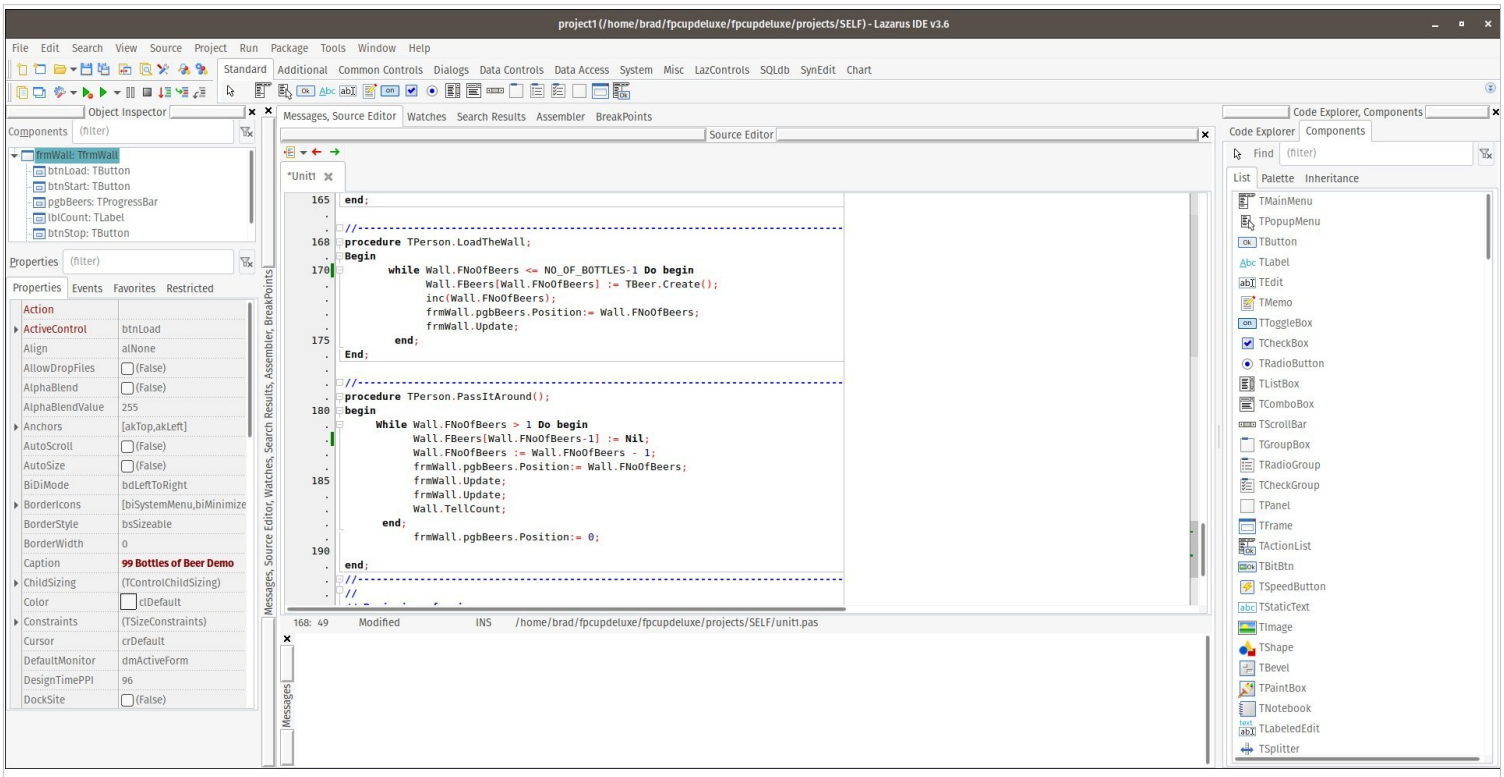


OK, time for our demo section.

I wanted to use something other than the usual “Hello World!” program, because, frankly, that only shows a simple I/O function. So I picked the “99 Bottles of Beer on the Wall” example program. Again, you can do this procedurally with about 6 lines of Pascal code. But I wanted to demonstrate Object Pascal’s object-orientation. So I created a Beer Class, a Wall Class, and a Person Class.

Then to add to the demo, I incorporated the open source Text-To-Speech program ‘espeak-ng’. So we’ll be able to hear the program sing the song.

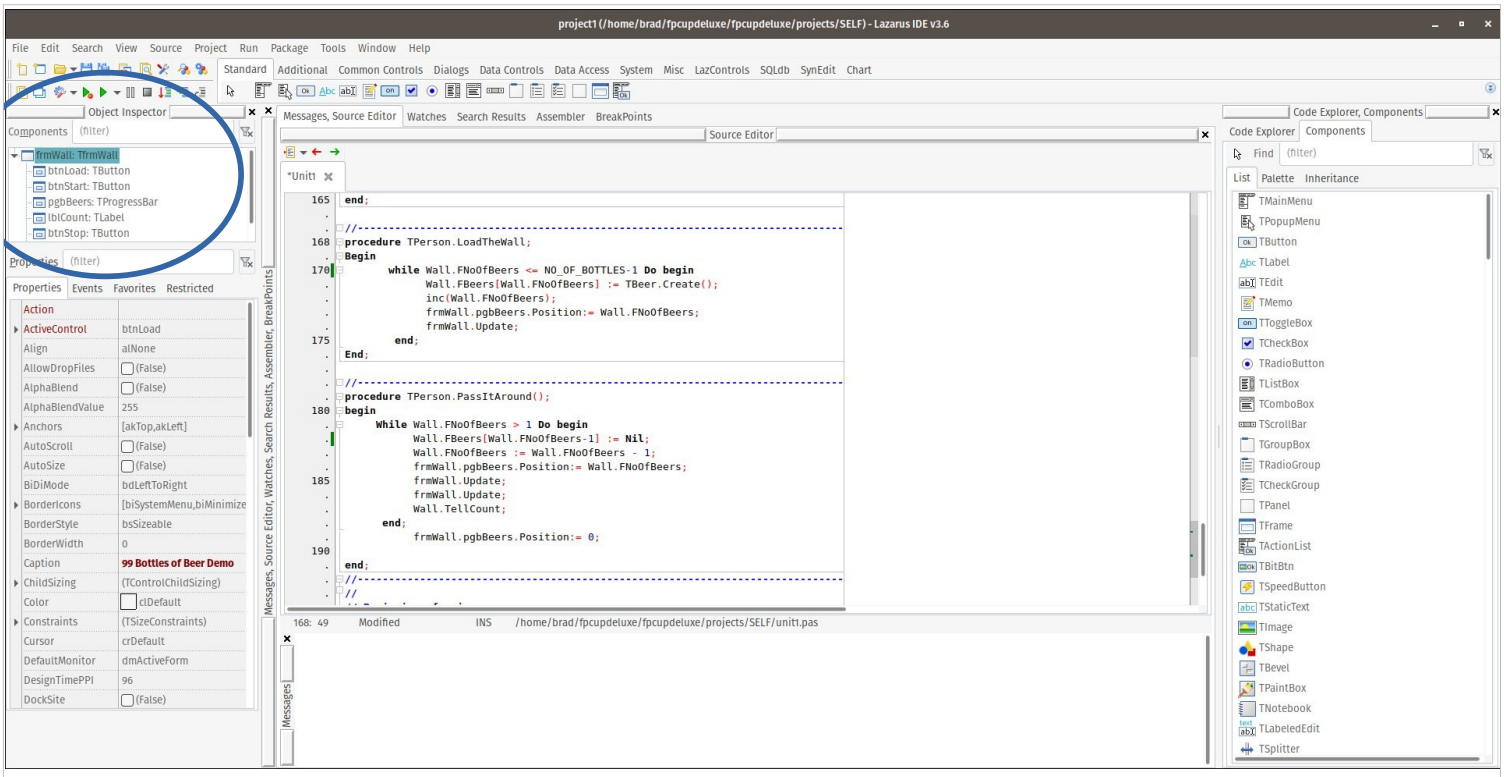
If time permits, I’ll show this same Object Pascal program running on Linux, Windows, Mac, and Android, using the same code.



Free Pascal Compiler's Lazarus IDE

I had intended on showing all the code live, but then realized it was, for the most part, too small. So I'll introduce you to the IDE and the code through screen shot slides.

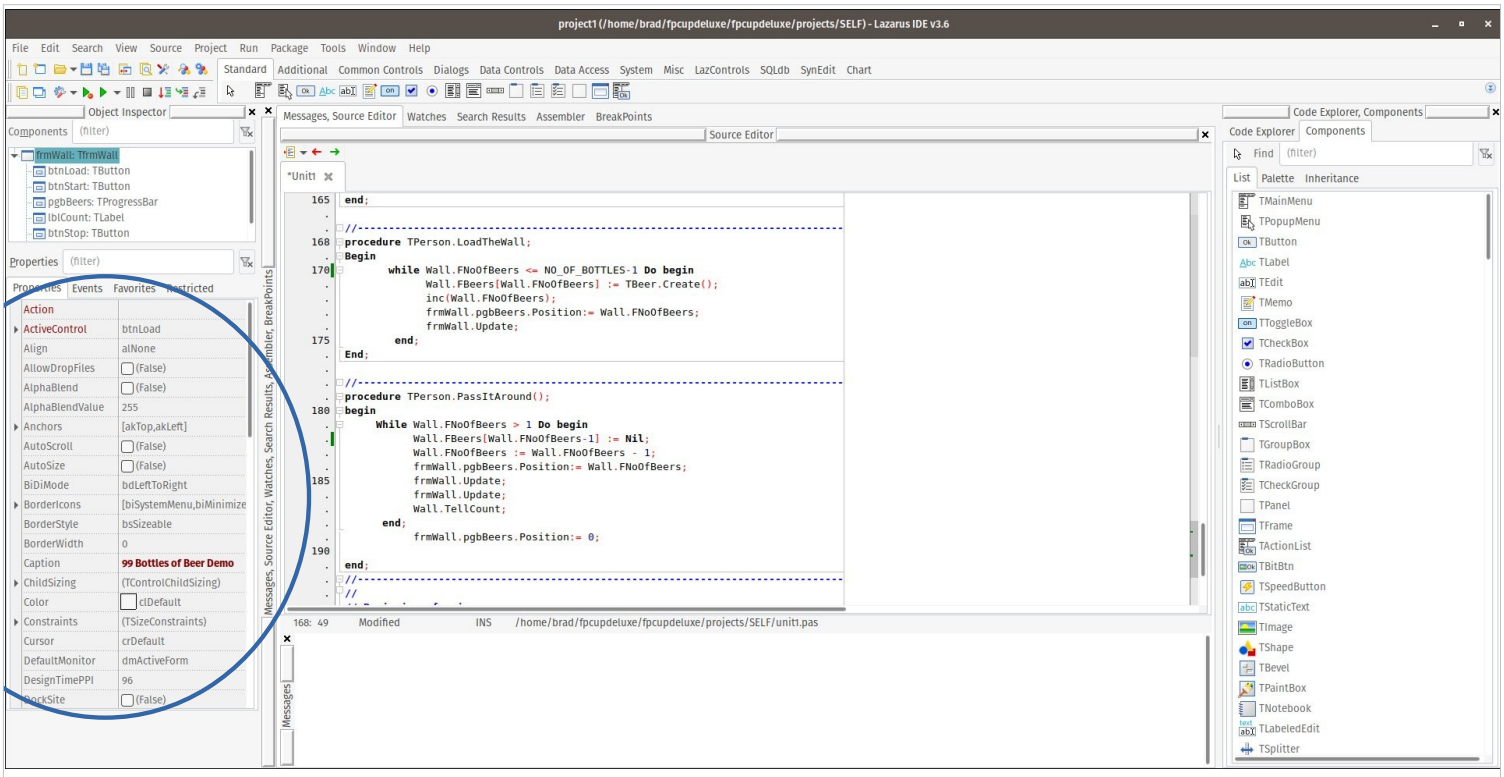
Here is the IDE. It appears somewhat complex, it's pretty simple.



List of Widgets in the Form

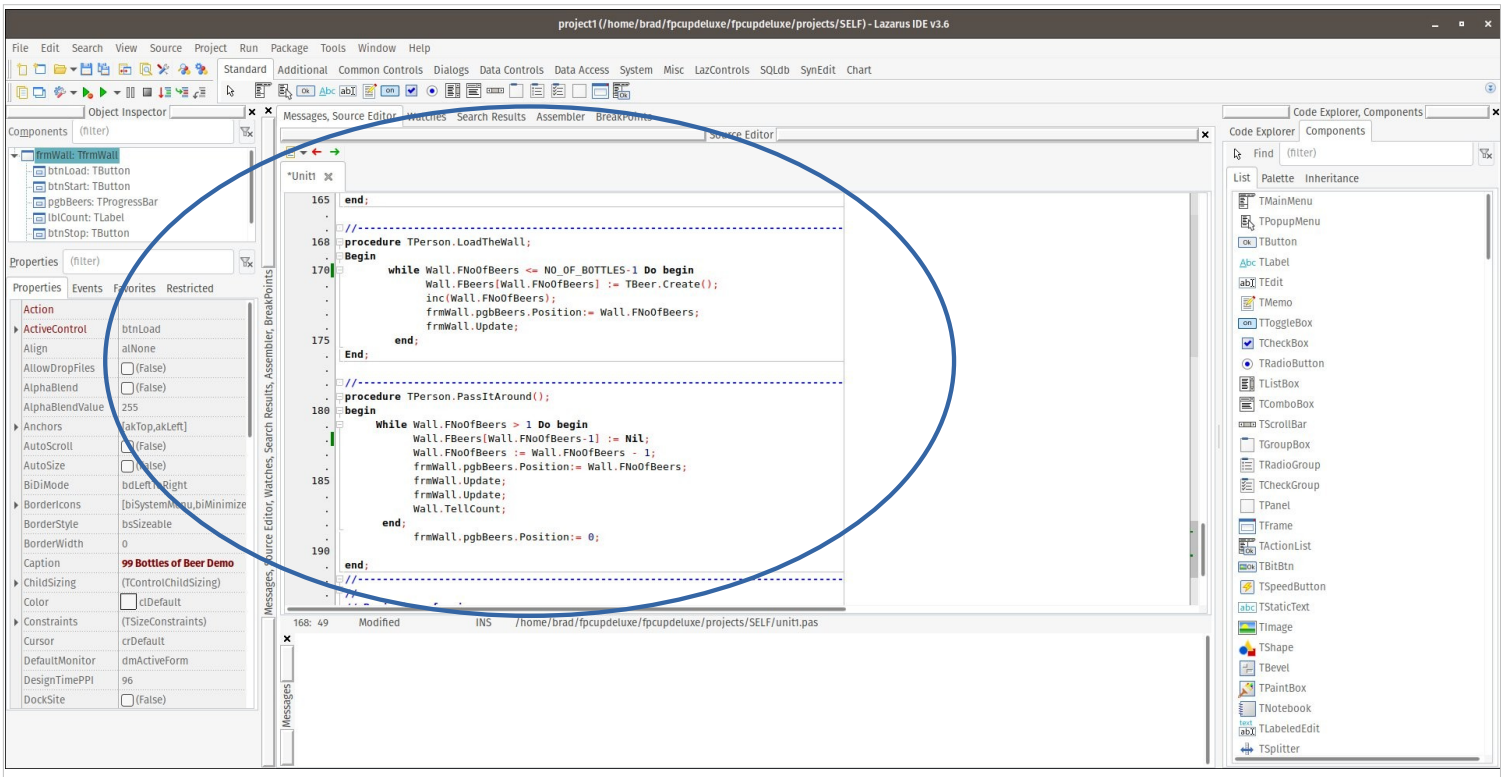
Here is a list of all the widgets we'll use on our Form.

The form is our graphical display screen



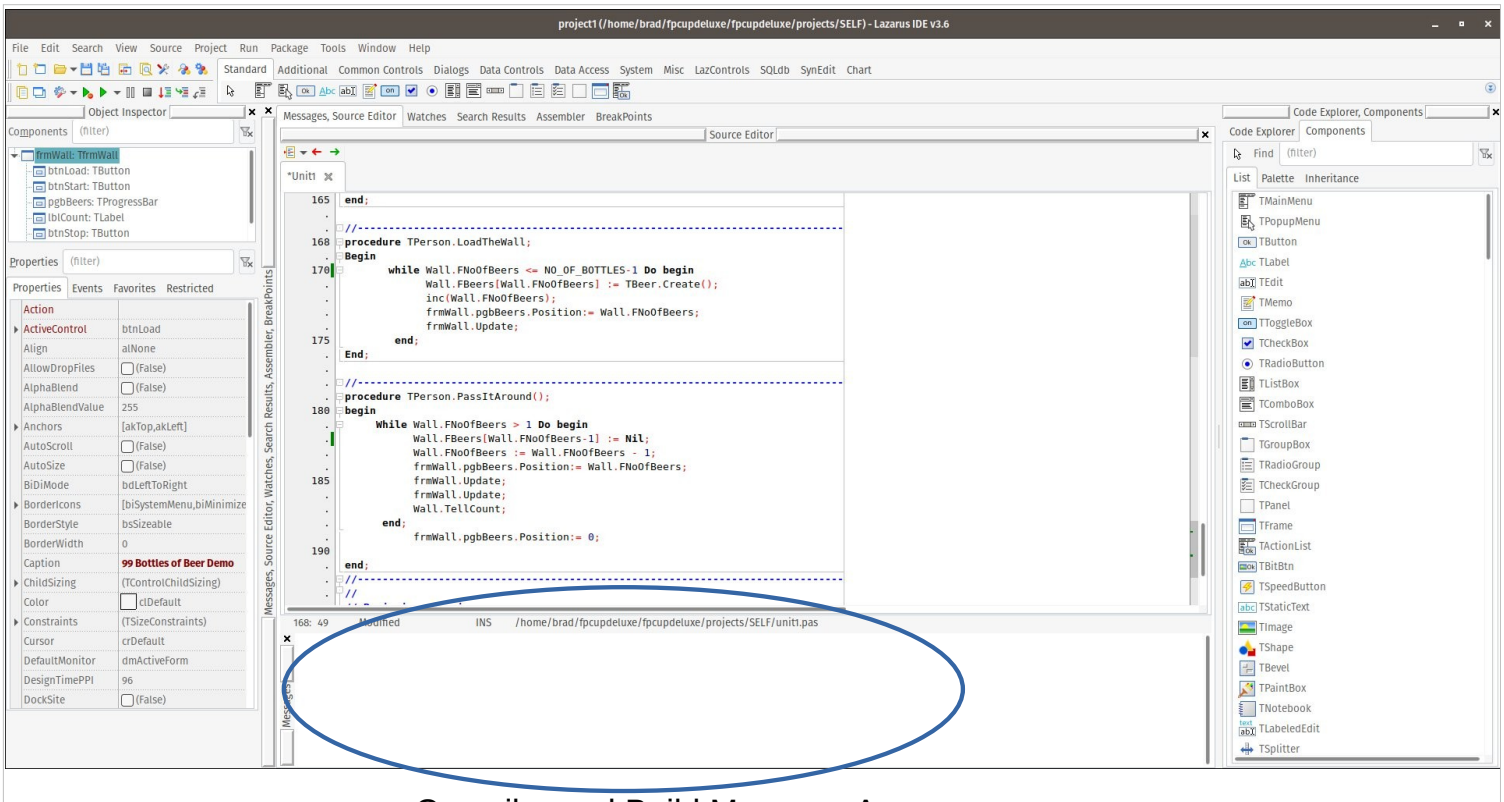
Properties of Each Widget

When you click on a widget in the list, you can set various compile time attributes, such as color, caption, contents, etc.



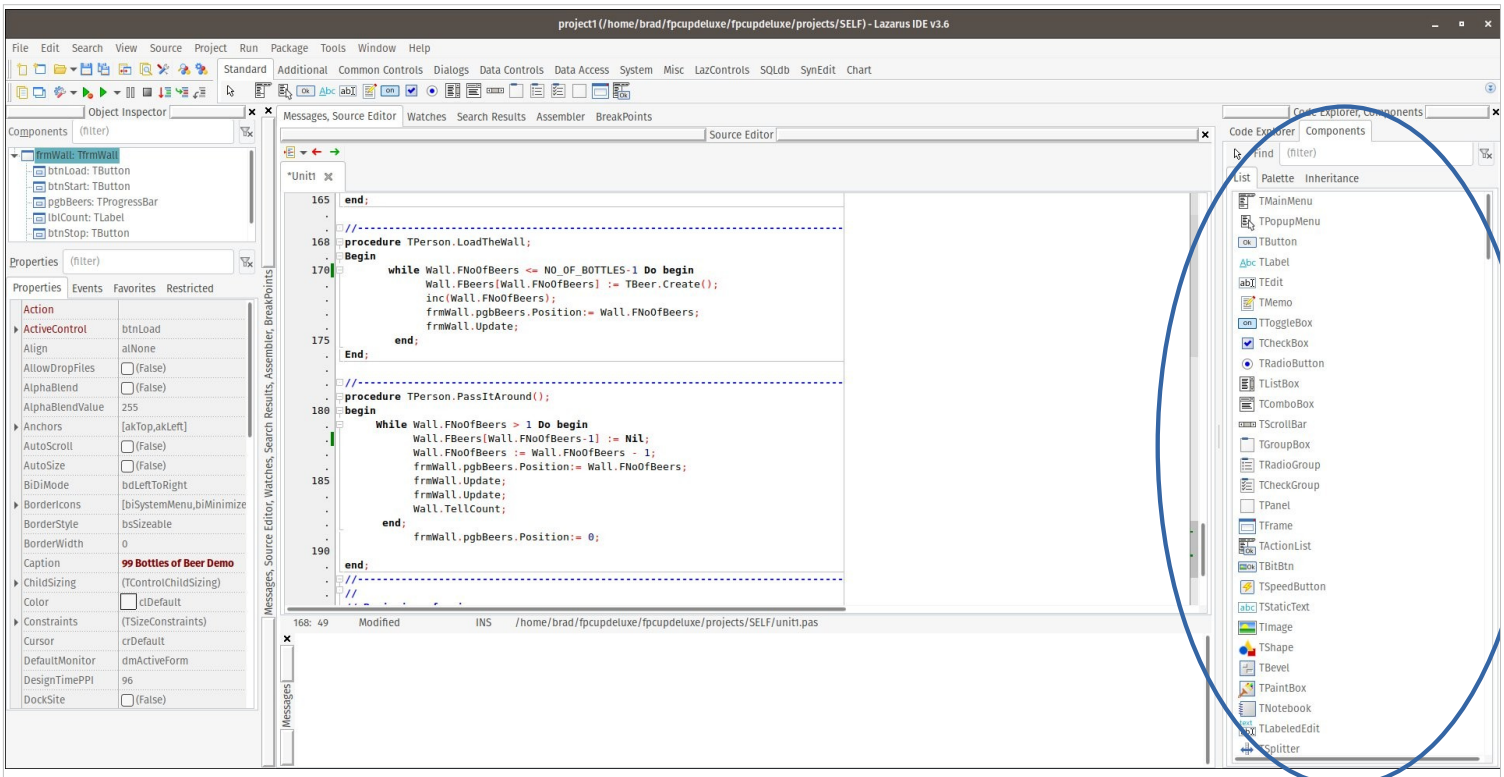
Code Editing Window (w/IntelliSense and Breakpoints)

Here's the some editor. It has IntelliSense, showing you possible choices and method signatures. This is also where you can set breakpoints for the debugger.



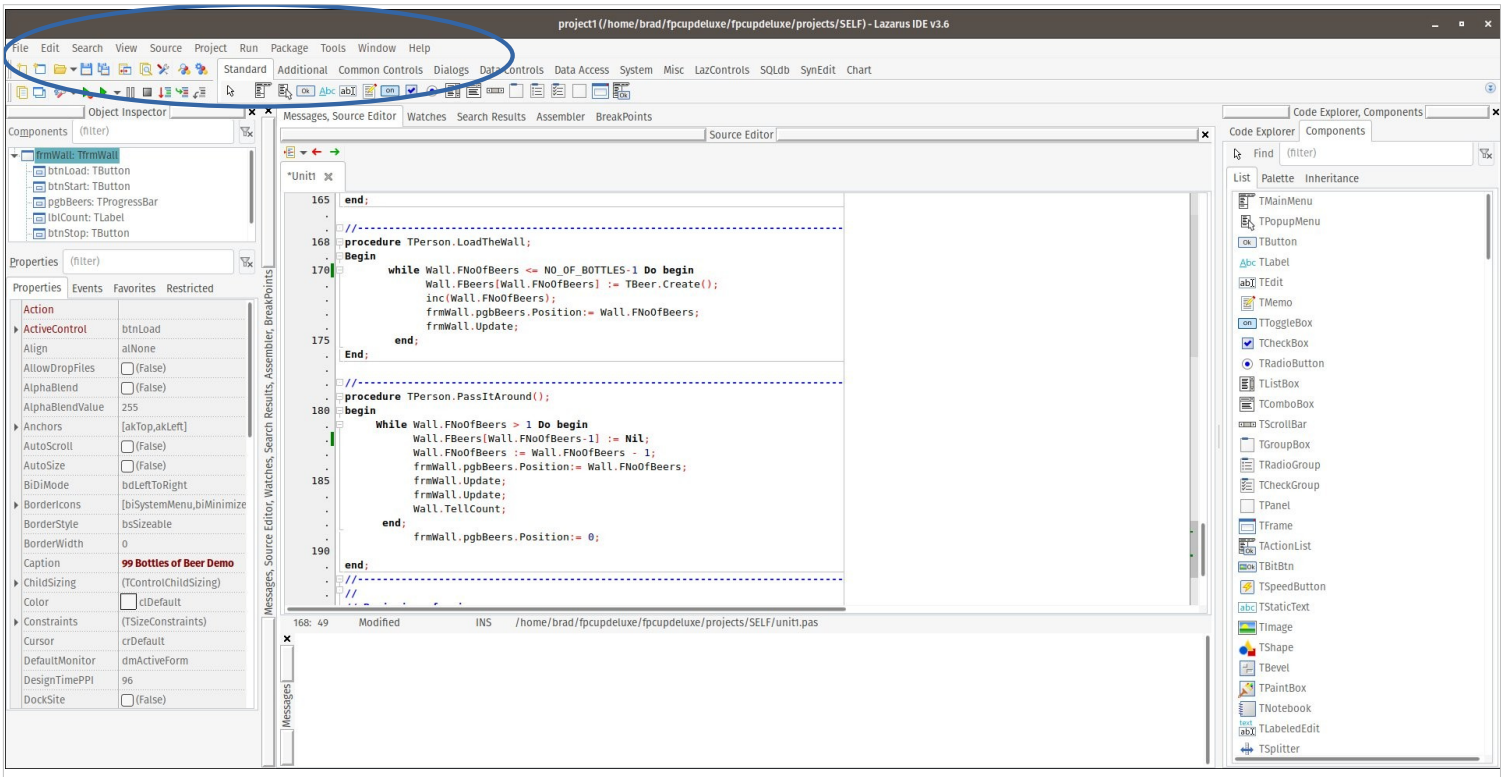
Compiler and Build Message Area

Here are where messages from the compilation and build are displayed. We hope for the green “Successful” message! :-)



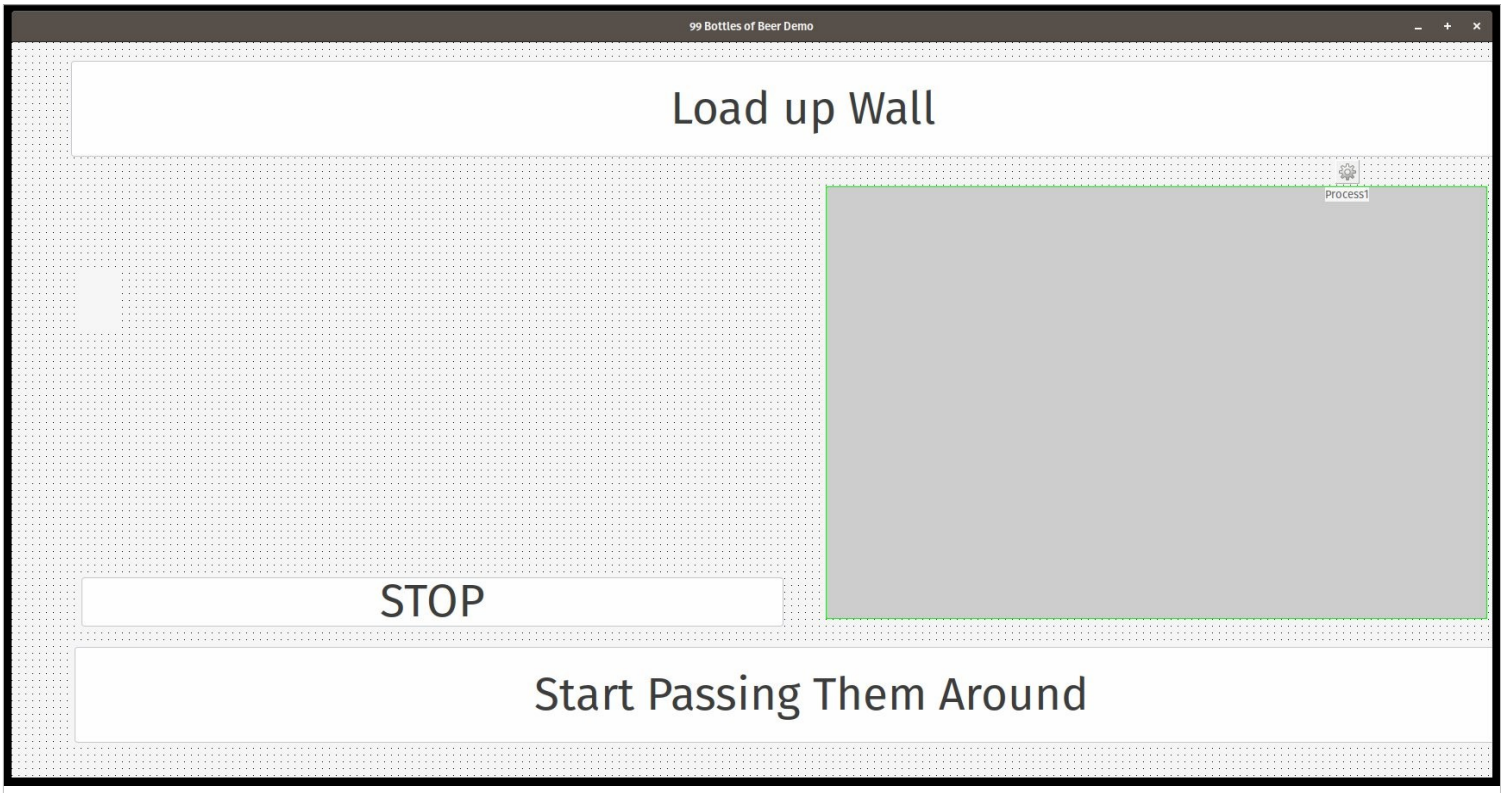
Available Widgets Library

This is the palette of all the widgets that can be dragged onto the Form.



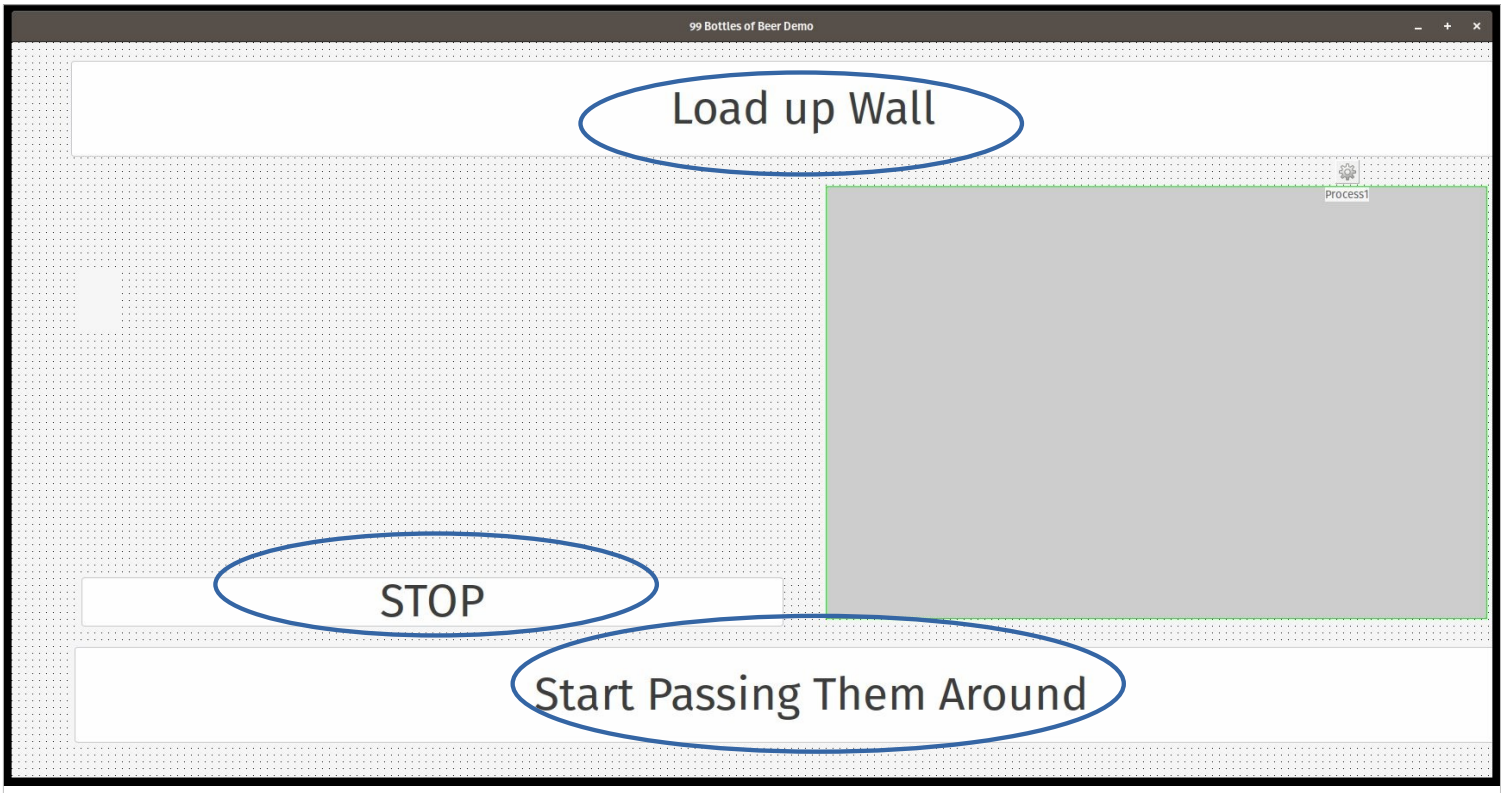
Standard Editing and Project Options Menus

Finally, here are the menus for normal editing functions and project options.



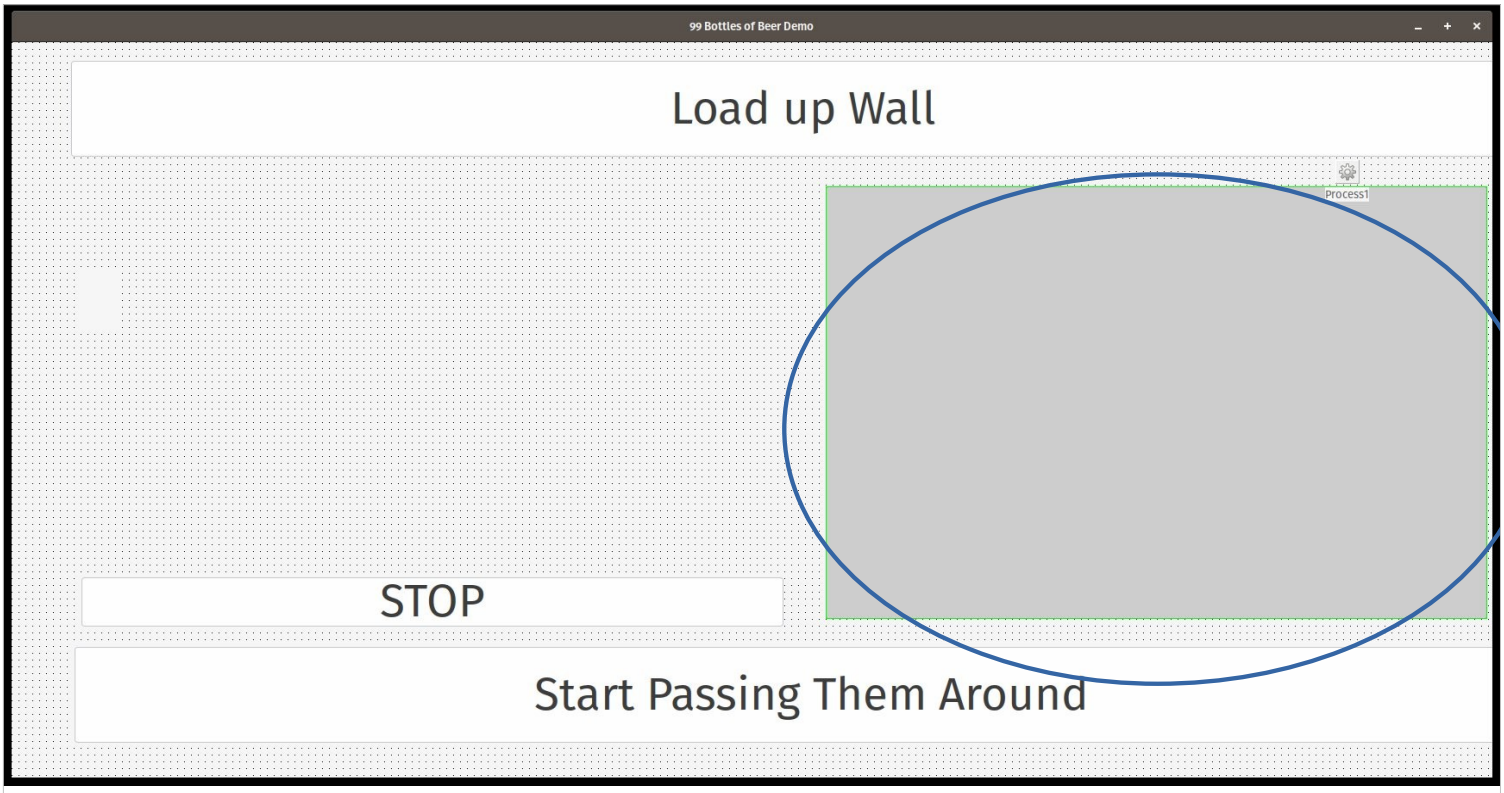
Free Pascal Compiler’s Lazarus Form Designer

Here’s our “99 Bottles of Beer” user interface Form



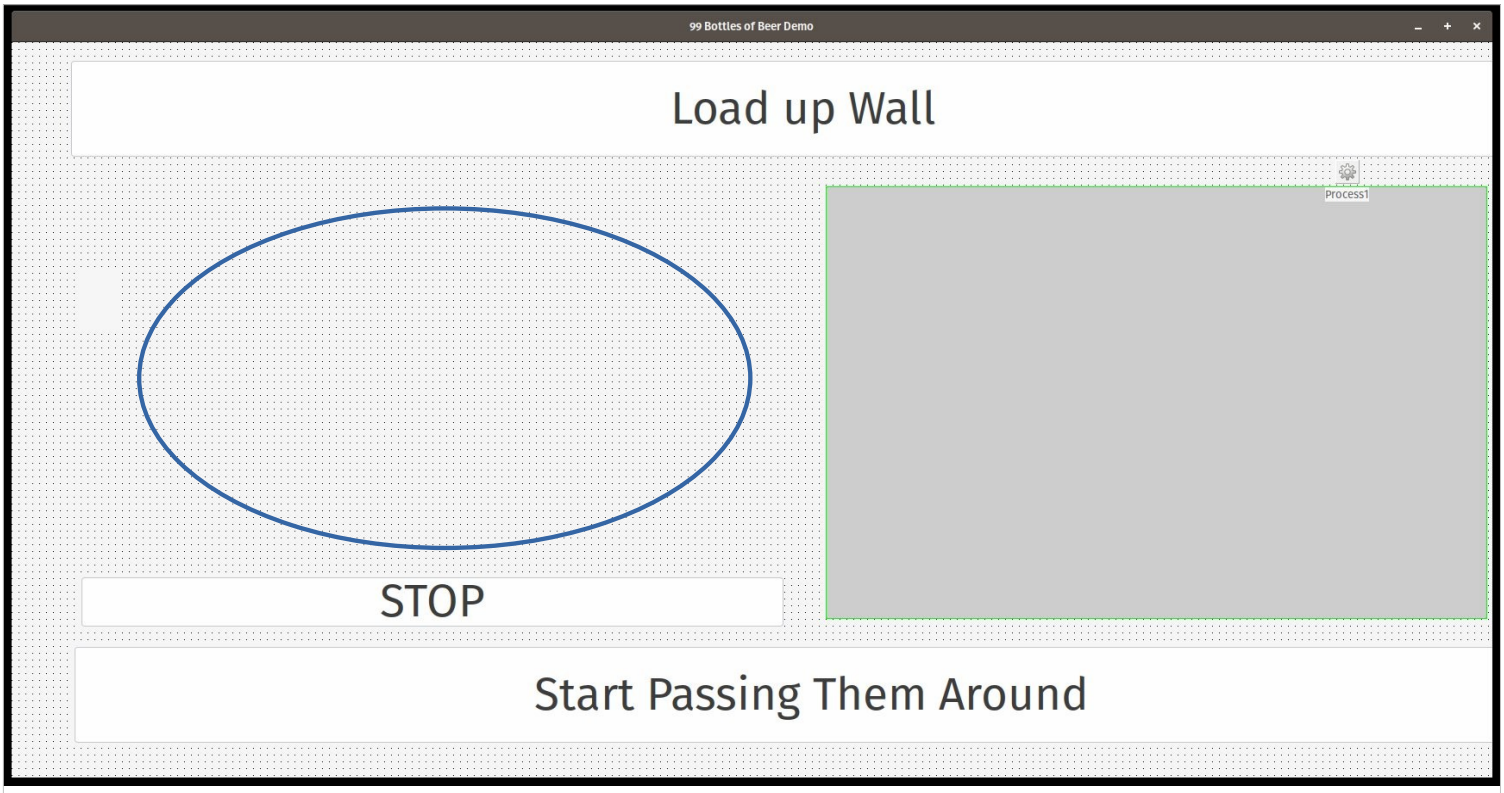
Button Widgets with Caption Property Set

I’ve dragged out three buttons and set their captions.



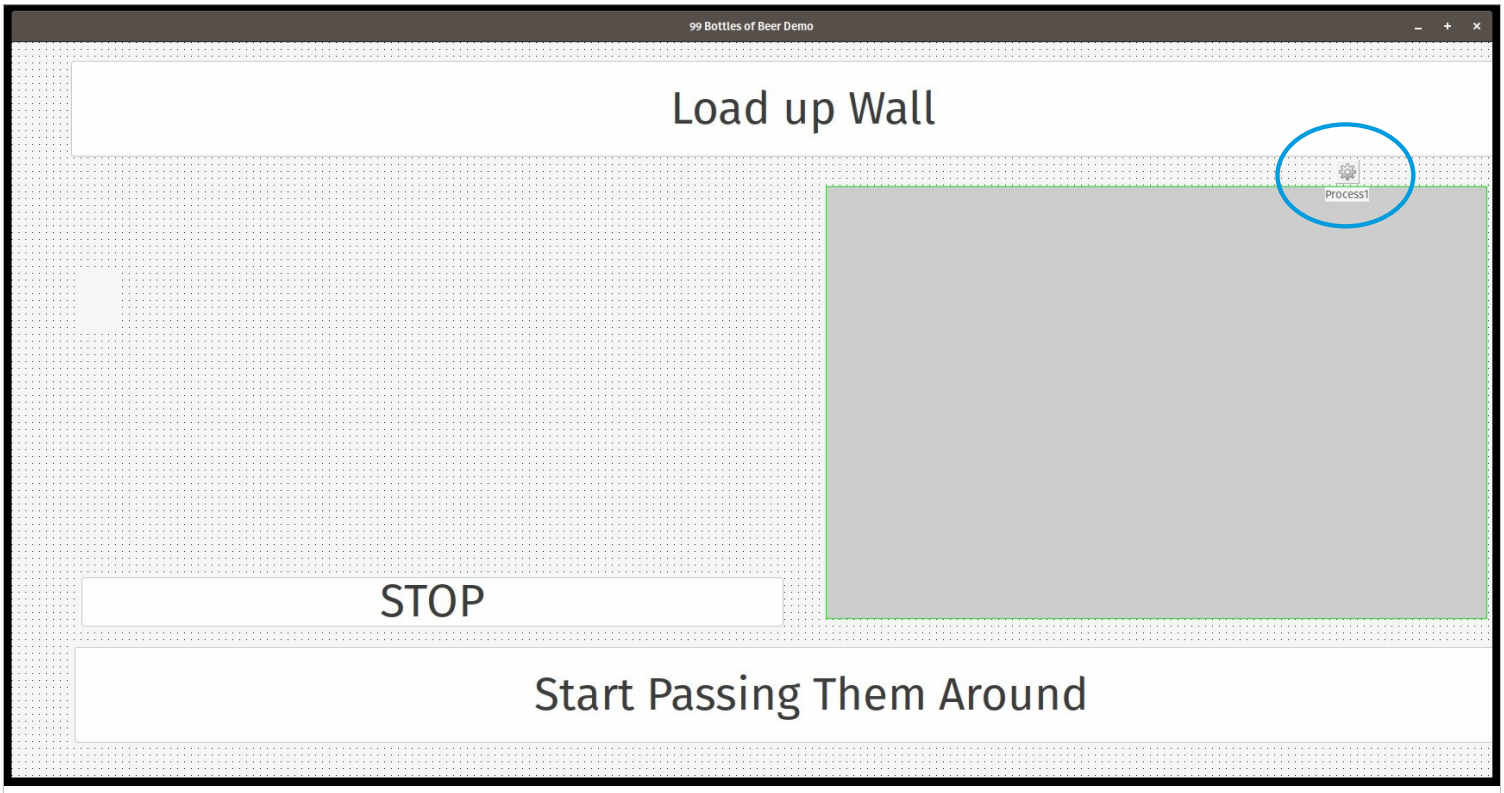
A Progress Bar Widget (This is Our “Wall”)

This is a progress bar and it represents our wall of beer. As we add beers to the wall, it will fill up and then empty as we remove bottles.



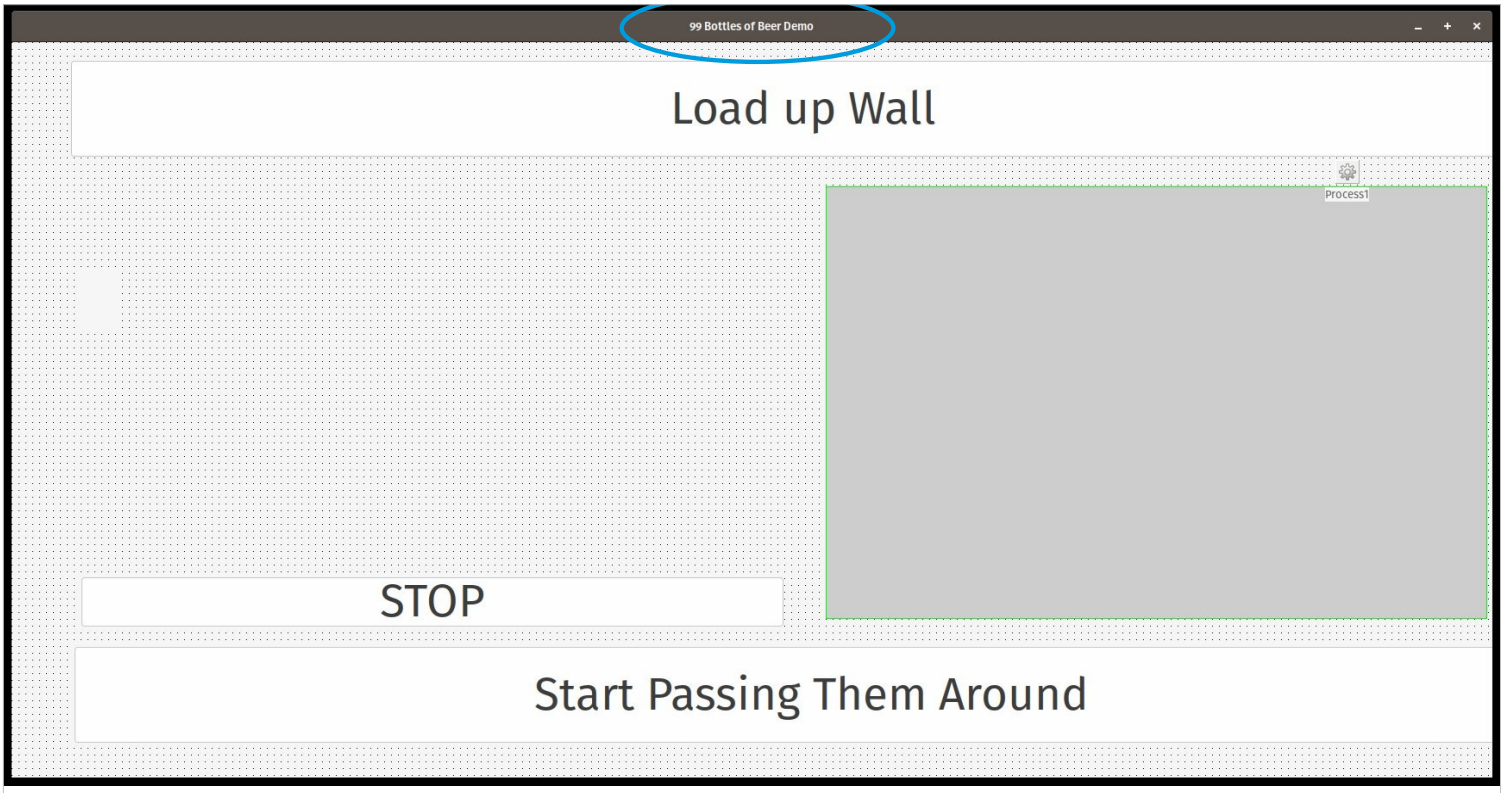
Text Label Widget (Invisible Because the Caption Property is Not Set)

You can’t see it because I removed the Caption, but this area has a text label. This is where we’ll print out the verses of the song as we go along.



"Process" Widget – For Calling External Programs

This little widget here is a Tprocess Object. It lets us run an external program, which in our case is the voice synthesizer.



Our Window Caption

Finally, we have our Caption for the Window.

```
type
//
// The form and its elements
//
  { TfrmWall }
  TfrmWall = class(TForm)
    btnLoad: TButton;
    btnStart: TButton;
    btnStop: TButton;
    lblCount: TLabel;
    pgbBeers: TProgressBar;
    Process1: TProcess;
    procedure btnLoadClick(Sender: TObject);
    procedure btnStartClick(Sender: TObject);
    procedure btnStopClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  end;
  //
  // The Beer class
  //
  TBeer = class
    constructor Create();
  end;
  //
  // The Wall class
  //
  TWall = class
    FBeers : array of TBeer;
    FNoOfBeers : integer;
    function TellCount(): integer;
    constructor Create(NoOfBottles : integer);
  end;
  //
  // the Person class
  //
  TPerson = class
     procedure LoadTheWall();
     procedure PassItAround();
     constructor Create();
  end;
```

Four Classes in Our Program

Switching back to code, here are our four Classes...

```
type
//
// The form and its elements
//
  { TfrmWall }
  TfrmWall = class(TForm)
    btnLoad: TButton;
    btnStart: TButton;
    btnStop: TButton;
    lblCount: TLabel;
    pgbBeers: TProgressBar;
    Process1: TProcess;
    procedure btnLoadClick(Sender: TObject);
    procedure btnStartClick(Sender: TObject);
    procedure btnStopClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  end;
```

Form Class Definition

The first Class is our Form. You can see it has the three Button type objects, the Label, the Progress Bar, and that Process widget. It also has 4 procedures. The first three are called in response to Button push events. The last one is a pseudo-constructor that sets the parameters of the Progress Bar.

```
end;
//
// The Beer class
//
TBeer = class
  constructor Create();
end;
//
```

“Beer” Class Definition

Now, here’s our Beer Class. It just exists. It doesn’t have any data nor does it have any methods other than the mandatory constructor.

```
end;
//
// The Wall class
//
TWall = class
  FBeers : array of TBeer;
  FNoOfBeers : integer;
  function TellCount(): integer;
  constructor Create(NoOfBottles : integer);
end;
//
```

"Wall" Class Definition

Here's the Wall Class. It has an unsized array of Beer objects. It has an integer field where it keeps track of the current number of beers available on the wall. It has one Function, where it 1) prints out the current lyric of the song, and 2) it returns the number of beers remaining. Finally, it has a constructor that sets the size of the array, based on our global constant of the number of beers.

```
//
// the Person class
//
TPerson = class
   procedure LoadTheWall();
   procedure PassItAround();
   constructor Create();
end;
```

"Person" Class Definition

Here's the Person Class. It doesn't have any data, but it has two Procedures that 1) load X number of Beers into the Wall's array, and 2) decreases the number of Beers by one and asks the Wall to tell us how many bottles on beer remain on the wall.

```
  //
  // Declaration of global variables - create our objects from their class template
  //
var
  frmWall: TfrmWall;
  Person : TPerson;
  Wall   : TWall;
  Beer   : TBeer;
//
//---------------------------------------------------------------------------
//
implementation
```

Instantiating Classes into Actual Objects

OK, Classes are templates from which Objects are constructed. Here we declare four variables that will hold an instance of each of our four Classes.

```
168 //------------------------------------------------------------------------------
    procedure TPerson.LoadTheWall;
    Begin
170         while Wall.FNoOfBeers <= NO_OF_BOTTLES-1 Do begin
                Wall.FBeers[Wall.FNoOfBeers] := TBeer.Create();
                inc(Wall.FNoOfBeers);
                frmWall.pgbBeers.Position:= Wall.FNoOfBeers;
                frmWall.Update;
175          end;
    End;

    //------------------------------------------------------------------------------
    procedure TPerson.PassItAround();
180 begin
        While Wall.FNoOfBeers > 1 Do begin
              Wall.FBeers[Wall.FNoOfBeers-1] := Nil;
              Wall.FNoOfBeers := Wall.FNoOfBeers - 1;
              frmWall.pgbBeers.Position:= Wall.FNoOfBeers;
185           frmWall.Update;
              frmWall.Update;
              Wall.TellCount;
         end;
              frmWall.pgbBeers.Position:= 0;
190
    end;
    //------------------------------------------------------------------------------
    //
```

Example Procedures from Person Object

To show you a sample of the code flow, here the Person's PassItAround procedure.

OK, let's switch over to the live IDE running on Pop! OS Linux and compile and run our program!