JPizza

Release 2.1.0

Jackson Smith

Feb 22, 2022

CONTENTS

CHAPTER

ONE

USAGE

1.1 Installation

To install JPizza, first download the latest archive from GitHub. Extract the archive, and add the bin folder of the archive to your path. You can then run `jpizza help` for more information on how to use JPizza.

1.2 File Extensions

- `.devp` : A raw JPizza source file.
- `.jbox` : A compiled JPizza bytecode file.

1.3 Running JPizza

You can run JPizza code by simply typing `jpizza <filepath>` into the command line. This will run both the source and bytecode files. To compile a file to bytecode,, simply run `jpizza <filepath> --compile`.

If you want to run JPizza in the repl, simply run `jpizza`.

CHAPTER

TWO

MAIN DOCUMENTATION

2.1 Comments

Comments are statements that the programming language ignores. You can declare comments in JPizza using two angle brackets facing away from each other.

You can also make multiline comments using two left facing angle brackets as the opening and closing.

```
<> This is a comment.

<<
This is a multiline comment.
Isn't it cool?
<<
```

2.2 Statements

Code in JPizza is made up of statements, which can be any form of expression. To end a statement, you must end it with a semicolon.

```
"This is a statement";
2 + 2;
3 + 4
<> No semicolon! This will cause an error!
```

2.3 Operators

JPizza has many operators, a list can be found below.

2.3.1 Binary Operations

Binary operations take in two operands.

- `+` - Adds the operands together.
- `-` - Subtracts the right operand from the left.
- `*` - Multiplies the operands together.
- `/` - Divides the left operand by the right.
- `^` - Raises the left operand to the power of the right operand.
- `%` - Gets the left operand modulo the right operand.
- `x=` (*where x is any binary operation above*) - Assigns the left operand, assuming it is a variable, to the result of the binary operation `x`.
- `:` - Returns the left value unless it is `null` or causes an error, in which case it returns the right value. If the right value causes an error, it returns `null`.

2.3.2 Unary Operations

Unary Operations only take in one operand.

- `x++` - Increments `x` and assigns the value to `x`, assuming it is a variable.
- `x--` - Decrements `x` and assigns the value to `x`, assuming it is a variable.
- `--x` - Returns the decremented value of `x`.
- `++x` - Returns the incremented value of `x`.
- `-x` - Gets the negative value of `x`.
- `!x` - Inverts `x`.
- `@x` - Converts `x` into a byte array.
- `$x` - Turns byte array `x` into an object.
- `~x` - Gets the binary complement of integer `x`.

2.3.3 Conditional Operations

- `&` - Checks if both the left operand, and the right operand are true.
- `|` - Checks if either the left operand, or the right operand is true.
- `>` - Checks if the left operand is greater than the right operand.
- `>=` - Checks if the left operand is greater than or equal to the right operand.
- `<` - Checks if the left operand is less than the right operand.
- `<=` - Checks if the left operand is less than or equal to the right operand.
- `==` - Checks if the left operand is equal to the right operand.
- `!=` - Checks if the left operand is not equal to the right operand.

2.3.4 Bit Operations

- ~& - Gets the bitwise and of both operands.
- ~| - Gets the bitwise or of both operands.
- ~^ - Gets the bitwise xor of both operands.
- <~ - Bit shifts the left operand by the right operand.
- ~~ - Performs an unsigned bit shift on the left operand by the right operand.
- ~> - Performs a signed bit shift on the left operand by the right operand.

2.4 Advanced Numbers

JPizza has some additional number properties.

2.4.1 Hex Numbers

You can create a hex number by typing `0x` followed by your hex number. These have the type of `hex`, and are represented as `0xHEXNUMBER` when printed. Adding a hex number to a normal number results in a normal number, but adding a normal number to a hex number results in a hex number.

`1 + 0x1 = 2` `0x1 + 1 = 0x2`

2.4.2 Algebraic Multiplication

Similar to algebra, you can enter commands such as `3x` as a shorthand for `3 * x`. However, you cannot do this with two variables, only a variable in a number. Attempting to do `xy` results in an error, since it treats `xy` as one word.

`var x => 4; 3x -> 12`

2.5 Variable Assignment

Variables are defined using the keywords '**var**' and '**bake**'.

Bake permanently sets a variables value, to where it cannot be changed. You can also use **const** as an alternative to this keyword.

After variables are defined, they can be set with the **var** keyword omitted. This will set the variable of the same name in the most recent scope.

```
var x  => 3;
bake y => "Hello!"; <> const y => "Hello!"; works also
x => 4;
y => "Goodbye!"; <> Throws an error since y is a baked variable.
```

You can declare multiple variables as null in the same statement by seperating each by a comma.

```
var x, y, z;
<> Declares x, y, and z! All are set to null.

x => 1; y => 2; z => 3;
<> Works!
```

If you want to remove a variable from the scope, you can use the `free` keyword followed by the variable name.

```
var x => 2;
println(x); <> 2

free x;
println(x); <> Error!
```

There is also a unique type of variable known as a callback variable. This type of variable does not store a value, but instead stores its expression. Whenver it is referenced, it returns the computed expression.

This can be compared to macros in other languages, like C.

```
var x => 4;
cal y -> x ^ 2;

println(y); <> Prints 16, since x^2 is currently 16.

x => 8;
println(y); <> Prints 64, since x^2 is now 64.
```

An additional feature for variables is static typing.

Although JPizza is dynamically typed, meaning that variables can store any type of data, you can statically type them using a colon followed by the variable type.

```
var x: num => 4;
<> Sets the type of variable x to be a number.

x => 8; <> Ok!
x => "Hello!"; <> Error!
```

To statically type variables quickly, you can use the **let** keyword, by doing `let x => value;`. This will automatically statically type `x` to the type of `value`.

```
let x => 4;
<> Sets x to 4 and statically types x to the type of 4, which is a number.

x => 8; <> Ok!
x => "Hello!"; <> Error!
```

One final special feature of variables is range limitations. In JPizza, you can use square brackets following a variable name to limit the range that a numeric value may lie in under that variable. By default, the value you enter will be the maximum, and the minimum will be set to 0. To add a custom minimum along with the maximum, you can have the minimum, followed by a pipe, (`|`), and then the maximum.

```
var sbyte [ -127 | 127 ] => 56;
<> Limits the variable sbyte to a range of -127 <= n <= 127.
var usbyte [ 255 ] => 12;
```

(continues on next page)

(continued from previous page)

```
<> Limits the variable usbyte to a range of 0 <= n <= 255.

<> Assigning either variable to a non-numeric or a number outside of the range causes a␣
↪runtime error!
```

2.6 Format Strings

Format strings give you the ability to directly inline variables and expressions into strings! To do this, create a string using backticks, (`), and inline any variables inside of brackets prefixed with a dollar sign. To escape that, you can prefix the dollar sign with an exclamation mark.

```
var x => 4;

println(`The value of x is ${x}`); <> Prints "The value of x is 4".
println(`To write ${x} you can do !${x}`); <> Prints "To write 4 you can do ${x}".
```

2.7 Lists

Lists are, simply put, a list or array of values.

Lists are defined using square brackets. Inside the brackets are each element, seperated by a comma.

List elements can be accessed by enclosing the index you want to access in square brackets following the list.

If you use a negative number to access a list index, it counts backwards.

Items can be appended to the list using the modulo operator, and removed using the division operator.

You can extend lists by other lists using the plus operator.

```
var list => [1, 2];

println( list[0] );
<> Prints the first element of the list, which is 1 in this case.

list += [3, 4];
<> Extends the list with the elements 3 and 4.
println( list );
<> Prints [1, 2, 3, 4]

list /= 3;
<> Removes 3 from the list.
list %= 6;
<> Adds 6 to the list.
```

2.8 Dictionaries

Dictionaries are a set of keys and values, like a word dictionary. In the case of the English dictionary, the keys would be words, and the values would be the definitions.

Dictionaries can be declared using braces. Inside these braces each key and value should be defined using the key followed by a colon and then the value. Each pair should be seperated by a comma.

You can get the stored value of a key by enclosing the key in square brackets following the dictionary.

You can set key value pairs by using the set function, and you can delete keys using the delete function.

```
var coolDict => {
    "abc": 123,
    "Hello": "world."
};

set(coolDict, "Hello", "world!");
<> Sets the value of the key "Hello" from "world." to "world!".

delete(coolDict, "abc");
<> Removes the key value pair "abc": 123 from the dictionary.

println("Hello" + " " + coolDict["Hello"]);
<> Prints "Hello world!".
```

2.9 Conditional Statements

2.9.1 If Statements

If statements are declared using the '**if**' keyword. The if keyword should then be followed with a true/false condition in parentheses, and then the body.

The body is the code that is run *if* the true/false condition is true. You can use curly braces, ({}), to write multiple lines, but if you only want to write one line braces are not necessary.

You can add additional branches to the statement using the '**elif**' and '**else**' keywords.

Elif statements are declared the same as if statements except with the '**elif**' keyword and should follow an (el)if statements body. These will run if the previous (el)if statement is false, and their condition is true.

Else statements are simply the '**else**' keyword followed by the bodyz. Else statements should come after an (el)if statement, and they only execute if the previous (el)if statement is false.

```
if (true) {
    println("This will always run!");
} elif (false) {
    println("This will never run. :(");
} else {
    println("If this ever runs, seek shelter immediately,
             the world is ending.");
}
```

2.9.2 Queries

Queries are a unique JPizza expression that is similar to a ternary in other languages, but cooler.

Query statements start with a question mark followed by a true/false condition. This condition should then be followed with a colon and an expression. This expression will be returned if the condition is true.

You can add additional branches to queries using the dollar sign followed by a condition. Then simply follow the dollar sign with a colon and an expression.

To add a default branch, you can use a dollar sign and an underscore, (`$_`).

```
var input => nfield("Please enter a number: ");
<> Assigns a user chosen number to the variable input.

var query => ? input > 10 : "Number is awesome!"
             $ input <  5 : "Number is cool."
             $_           : "Number is meh...";

<> Assigns the value of the query to the variable query.
<> If the input is greater than 10, the value will be "Number is awesome!"
<> If the input is less than 5, the value will be "Number is cool."
<> If none of the above are true, the value will be "Number is meh..."
```

2.9.3 Case Trees

Case trees are special statements that can quickly run code if two values are equal. JPizza has two types of case trees, the famous switch statement and the special match statement.

Switch

Switch statements allow you to pass in a value and run code based on if a given value is equal to one of the cases. What makes switch statements special is their ability to fall-through, meaning that once one case is matched, the code of all cases below it run until it reaches the '**break**' keyword or runs out of cases.

To declare a switch statement, start with the '**switch**' keyword followed by the value you want to compare in parentheses. Then follow it with curly braces, (`{}`).

Inside the braces are the cases. To declare a new case, start with the keyword '**case**' followed by the value you want to check. Follow the value with a colon. Any code written afterwards up until the next case statement will be run if the values are equal.

To add a default case, you can use the '**default**' keyword followed by a colon, and the code you want to execute. Defaults cases are immune to fall-through.

```
switch (4) {
    case 1:
    case 3:
    case 5:
    case 7:
    case 9:
        println("Odd number!");
        break;
    <> Fall through makes it so that all the cases up until case 9 execute this code.
```

(continues on next page)

(continued from previous page)

```
    case 2:
    case 4:
    case 6:
    case 8:
        println("Even number!");
        break;

    default:
        println("Unhandled number...");
    <> This will execute if the number is not in any of the above cases.
};
```

Match

Match statements are similar to switch statements in structure, but different in execution. Match statements return a value when their case is matched, and they are immune to fall-through. Match statements also have the ability to use the pattern matching feature, which will be elaborated on momentarily.

To declare a match statement, start with the '**match**' keyword followed by the value you want to compare in parentheses. Then follow it with curly braces, ({}).

Inside the braces are the cases. To declare a new case, start with the keyword '**case**' followed by the value you want to check. Follow the value with a temporary assignment arrow, (->) followed by the value you want to return and then a semicolon.

To add a default case, you can use the '**default**' keyword followed by an arrow, and the value you want to return.

```
println(match (4) {
    case 2 -> 4;
    case 4 -> 8;
    case 8 -> 12;

    default -> 0;
});

<> Prints the matching value, which is 8 in this case.
```

You can also use pattern matching, which allows you to declare a special statement called a pattern and use that to extract or test specific attributes against an object. The general syntax for pattern matching is the object name followed by parenthesis. Inside of the parenthesis should be the attribute name followed by a colon and then the expected value, each separated by a comma. You can make the expected value a variable that does not exist, and it will instead assign the value of the attribute to that variable if the rest of the pattern matches. You can also omit the colon and expected value and it will automatically set the expected value to be a variable that is the same name as the attribute.

```
class Box {
  pub value;
  ingredients<x> {
    value => x;
  }
}

var example => Box(123);
```

(continues on next page)

(continued from previous page)

```
match (example) {
  Box(value: 456) -> println("Is Box(456)");
  <> Tests if its a Box where the attribute 'value' is 456.

  Box(value: x) -> println(`Is Box(${x}`);
  <> Tests if its a Box, and assigns the attribute 'value' to the variable x.

  Box(value) -> println(`Is Box(${x})`);
  <> Tests if its a Box, and assigns the attribute 'value' to the variable value.
  <> Same thing as Box(value: value)
};
```

2.10 Loops

JPizza has 5 types of loops, standard loops, **for** loops, **iterative** loops, **while** loops, and **do-while** loops.

2.10.1 Standard Loops

What is a Standard Loop?

Standard loops are loops that do not have any sort of condition, and simply repeat code infinitely until it is stopped, either from inside the code or by the user.

How to Make a Standard Loop

To make a standard loop, we simply need to type the keyword '**loop**'. Following the **loop** keyword, we need our body. The body is the code that is executed during the loop.

JPizza offers two forms of bodies, the arrow form, and the braces form.

Arrow Form

Arrow form allows you to quickly write the loop on one line, and return a list afterward.

Using the assignment arrow, (=>), followed by an expression, we can set the body to the provided expression.

Once the loop finishes, it returns a list in order of each evaluation of the expression.

```
var index => 0;
var lp => loop => if (index > 5) break;
                  else index++;

<<

This assigns lp to the value of the loop.
The loop increments the value index until it is greater than 5.
This returns a list of [1, 2, 3, 4, 5, 6].

<<
```

Braces Form

Brace form allows you to write multiple lines of code, but with no return value.

By following the condition of the loop with curly braces, ({}), we can use brace form. Simply insert your code inside the braces to set it to the body.

```
var r;
loop {
  if ((r => random()) > 0.5)
    break;
  println(r);
}

<<

This loop prints random numbers until one is greater than 0.5.

<<
```

2.10.2 For Loops

What is a For Loop?

For loops are a type of loop that runs code over a range of numbers, passing in a variable for the current loop number.

How to Make a For Loop

For loops are denoted by the '**for**' keyword followed by parentheses. Inside the parentheses will be the condition of the for loop.

Condition

First start off with an identifier, like 'i'. This identifier will be the name of the variable that is passed into our code representing the current loop number.

Following the identifier we put a temporary assignment arrow, (->), which means that our variable will be dropped after the loop ends.

Next we have our first number, then a colon followed by our second number. This represents our range.

Finally, for the body we have the option to put a step number in, which represents how much we increase as we iterate. If we want to add this, we put the step arrow, (>>), followed by how much we want to step.

Body

After we close the parentheses, we write our body. The body is what is executed for each iteration.

JPizza offers two forms of bodies, the arrow form, and the braces form.

Arrow Form

Arrow form allows you to quickly write the loop on one line, and return a list afterward.

Using the assignment arrow, (=>), followed by an expression, we can set the body to the provided expression.

Once the loop finishes, it returns a list in order of each evaluation of the expression.

```
var lp => for (n -> 0:3) => 2 ^ n;

<<

This assigns lp to the value of the for loop.
The loop iterates n over the range 0 to 3. It then evaluates to 2^n.
This returns a list of [1, 2, 4], as 2 ^ 0 is 1, 2 ^ 1 is 2, and 2 ^ 2 is 4.

<<
```

Braces Form

Brace form allows you to write multiple lines of code, but with no return value.

By following the condition of the loop with curly braces, ({}), we can use brace form. Simply insert your code inside the braces to set it to the body.

```
for (i -> 3:5 >> 0.5) {
    println(i);
    <> Prints i for each iteration.
}

<<

This loop iterates i over the range 3 to 5.
As specified by our step arrow, it increases by 0.5 each time.
This ends up printing:
3
3.5
4
4.5

<<
```

2.10.3 Iterative Loops

What is an Iterative Loop?

An iterative loop is a loop that iterates over a list of values. It takes in an identifier and assigns a variable using that identifier to the current element.

It's the same thing as writing a for loop that iterates over each index in a list and then assigns a variable to the value at each index.

How to Make an Iterative Loop

Similarly to a for loop, it starts off with the keyword '**for**', followed by the condition.

Condition

The condition starts with an identifier, which will be passed into the body as our variable. Follow the identifier with an iterator arrow, (<-), and then the value you want to iterate over.

Body

JPizza offers two forms of bodies, the arrow form, and the braces form.

Arrow Form

Arrow form allows you to quickly write the loop on one line, and return a list afterward.

Using the assignment arrow, (=>), followed by an expression, we can set the body to the provided expression.

Once the loop finishes, it returns a list in order of each evaluation of the expression.

```
var lp => for (n <- [1, 2, 3]) => n + 2;

<<

This assigns lp to the value of the iterative loop.
The loop iterates over the list [1, 2, 3] and adds 2 to each.
This returns a value of [3, 4, 5], as 1 + 2 is 3, 2 + 2 is 4, and 3 + 2 is 5.

<<
```

Braces Form

Brace form allows you to write multiple lines of code, but with no return value.

By following the condition of the loop with curly braces, ({}), we can use brace form. Simply insert your code inside the braces to set it to the body.

```
for (i <- ["Hello", "world!"]) {
    println(i);
    <> Prints i for each iteration.
}

<<

This loop iterates i over the list ["Hello", "world!"].
This ends up printing:
Hello
world!

<<
```

2.10.4 While Loops

What is a While Loop?

While loops are no doubt the simplest type of loop. They simply take in a true/false condition and repeat the code *while* the condition is true.

How to Make a While Loop

While loops start with the keyword '**while**', followed by the condition in parentheses.

After the condition, we have the body. JPizza offers two forms of bodies, the arrow form, and the braces form.

Arrow Form

Arrow form allows you to quickly write the loop on one line, and return a list afterward.

Using the assignment arrow, (=>), followed by an expression, we can set the body to the provided expression.

Once the loop finishes, it returns a list in order of each evaluation of the expression.

```
var a => 1;
var lp => while (a < 10) => a => -(a + abs(a) / a);

<> The equation -(a + abs(a) / a) simply causes a to tesselate and iterate, as in:
<>  1 -> -2
<> -5 ->  6

<<

This assigns lp to the value of the while loop.
This loop executes the body while a is less than 10.
In the body it reassigns a until it is greater than 10.
This returns [-2, 3, -4, 5, -6, 7, -8, 9, -10, 11].

<<
```

Braces Form

Brace form allows you to write multiple lines of code, but with no return value.

By following the condition of the loop with curly braces, ({}), we can use brace form. Simply insert your code inside the braces to set it to the body.

```
while (true) {
    var r => random();
    if (r > 0.5) { break; }
    println(r);
}

<<

This loop executes infinitely until it encounters the break keyword.
First it assigns r to a random number between 0 and 1.
Next, it breaks the loop if r is greater than 0.5.
If the loop continues to run, it prints the random number.

<<
```

2.10.5 Do-While Loops

Please read the section on while loops before reading

What is a Do-While Loop?

Do-while loops are very similar to while loops, except they run the body before checking the condition, compared to checking the condition before running the body. This causes do-while loops to run at least once before they stop.

How to Make a Do-While Loop

Do-while loops start with the keyword '**do**'. After **do**, we have the body, which can be defined the same as a while loop. Finally, we have the '**while**' keyword, followed by the condition in parentheses. This should end with a semicolon, since it is not a bracket. Your final loop should look something like this:

```
do {
  println("This condition is false!");
} while (1 == 2);
```

2.10.6 Keywords

Loops come with 2 special keywords, **break** and **continue**.

Break stops the loop instantly.

Continue skips the rest of the loops body and *continues* to the next iteration.

2.11 Functions

2.11.1 What is a Function?

Functions are special types that take in arguments, execute code using those arguments, and return a value. They can be used to repeat certain pieces of code with different values each time.

2.11.2 How to Make a Function

First, declare a new function with the '**fn**' keyword.

Name

This should then be followed by the function name. You can omitt the function name to make it anonymous. In the case of anonymous functions, it's better to use an exclamation mark to turn it into a lambda.

NOTE: *Anonymous functions must be followed by a semicolon, even if you're using curly braces, ({}).*

Arguments

After this, you put the name of each argument seperated by a comma inside of angle brackets, (<>). If your function takes in no arguments, the angle brackets can be omitted. Otherwise, there must be a space in between to prevent it from becoming a comment.

Body

Arrow Form

Arrow form allows you to quickly write the function on one line, and return a value afterward.

Using the temporary assignment arrow, (->), followed by an expression, we can set the return value to the provided expression.

```
fn addOne<x> -> x + 1;

<> This creates the function addOne, which takes in x
<> and returns x + 1.
```

Braces Form

Brace form allows you to write multiple lines of code, but with no automatic return value.

By using curly braces, ({}), we can use brace form. Simply insert your code inside the braces to set it to the body.

If you want to return a value, you must say it explicitly using the '**return**' keyword followed by the return value.

You can also return the last line of code automatically by omitting the semicolon on the last line.

```
fn add<x, y> {
    return x + y;
}

<> Creates the function add, which takes in x and y
<> and returns x + y.
```

2.11.3 Using Functions

Functions can be called by typing the function name followed by parentheses. Inside the parentheses is where the arguments go, each seperated by a comma. If there are no arguments, the parentheses can remain empty.

```
println(addOne(5));
<> Prints 6, since 5 is passed in as x and addOne returns x + 1.

println(add(3, 4));
<> Prints 7, since 3 and 4 are passed in as x and y, and add returns x + y.
```

You can also spread a list of arguments out over a function using the spread operator (`..`) followed by the list.

```
println(add(..[3, 4])); <> This is the same as add(3, 4), it spreads out the values into␣
↪the arguments.
```

2.11.4 Complex Functions

Static Typing

You can expand upon functions to use static typing rules like with variables earlier.

You can statically type arguments the same we statically type variables, by following the identifier with a colon and then the type name.

To statically type the return value, insert either the keyword '**yields**' or a colon and then the type name before the body of the function.

```
fn add<x: num, y: num> yields num {
    return x + y;
}

<> Statically types x and y to be numbers.
<> yields num makes the return value a number.
```

Generic Typing

Along with static typing, you can use generic typing. Simply follow the arguments with parenthesis containing the generic type names. When calling the function, pass in the types inside angle brackets after the arguments.

```
fn myGeneric<x: T>(T) -> println(`${x} has a generic type of ${T}`);

myGeneric(2)<num>;

<<
Console Output:
"2 has a generic type of num"
<<
```

Generic types can also be inferred. If the type has an argument that uses that type and the type is omitted, it will automatically infer that the type will be the same as the variable. This allows you to simplify the above call to:

```
myGeneric(2);
<> Since x is of type T, and the
<> provided x has a type of num,
<> it infers that T is num
```

Default Arguments

We can do even more with arguments after this too. We can add default values to arguments, so if they are left unspecified the default arguments are substituted in. After you start declaring default arguments however, each following argument must have a default value.

To add a default value, simply follow the argument with an equals sign and then the default value.

```
fn add<x: num, y: num = 1> yields num {
    return x + y;
}

<> Now y has a default value of 1.

println(add(5));
<> Prints 6, since 1 is substituted in for the missing y argument.

println(add(7, 2));
<> Prints 9, since y is accounted for.
```

Iterative Arguments

You can get any extra arguments passed in as a list by putting `..` followed by the name you want it to be assigned to at the end of your arguments. Following this with any more argument names will cause an error.

```
fn iterativePrint<..messages> {
  <> Iterates through each argument and prints it
  for (i <- messages) {
    print(i);
  }
```

(continues on next page)

(continued from previous page)

```
}

iterativePrint("a", "b", "c");
<> Passes in ["a", "b", "c"] to the function as the variable 'messages'
<> Ends up printing abc
```

Keyword Arguments

You can get a dictionary of specifically named arguments by putting a backslash after your last argument and then the name of what you want the dictionary to be passed in as. To give a function specifically named arguments, follow your last argument with a backslash when calling it. After the backslash give it all of your named arguments in the format `name: value`, each seperated by a comma.

```
fn printNamed<\ dict> {
  <> Iterates through every pair and prints it
  for (key <- list(dict)) {
    println(`${key}: ${dict[key]}`);
  }
}

printNamed(\ a: "b", c: 4, hello: "world");
<<
Passes {"a": "b", "c": 4, "hello": "world"} in to printNamed as dict
Prints out:
a: b
c: 4
hello: world
<<
```

Asynchronous Functions

By adding the keyword '**async**' after the **fn** keyword, we can make the function asynchronous. This means that the program doesn't wait for the function to finish before continuing, the function simply runs in the background.

```
import time;

fn async printTen<msg> {
  for (i -> 0:10) {
    println(msg);
    time::halt(500); <> Pauses for 500ms.
  }
}

printTen("This is asynchronous!");
printTen("This is also asynchronous!");

<> Both messages are printed ten times simultaneously.
```

Decorators

Similar to other languages, JPizza suppports decorators. Decorators are functions that are called before the function they are decorating. They can be used to modify the behavior of the function they are decorating.

```
fn myDecorator<func> -> fn <x> {
  println(x);
  return func(x) + 1;
}

/myDecorator/
fn coolFunc<x> -> 2x;

println(coolFunc(3));

<<
Console Output:
3
7
<<
```

2.12 Enumerators

2.12.1 What is an Enumerator?

Enumerators are values that consist of predefined constants. This essentialy means that we can define an enumerator with a bunch of keywords we want to use in our pgoram later on without having to give them values. JPizaz is special, where enums are their own type compared to most languages giving them integer values. Enums comparisons are only true when compared to enums of the same parent and name.

2.12.2 How to Make an Enumerator

You can define an enumerator using the '**enum**' keyword followed by curly braces, (`{}`). Inside the curly braces should be each keyword followed by a comma, including the last one.

To access enums from their parent, use the parent followed by the double colon operator, (`::`), and then the enum name.

```
enum PizzaToppings {
  Sausage,
  Pineapple,
  Pepperoni,
}

var favorite => PizzaToppings::Sausage;
if (favorite == PizzaToppings::Pineapple) {
  println("You are mentally unstable.");
}
```

2.12.3 Public Enumerators

You can also follow the **enum** keyword with **pub** to make each child publicly available.

```
enum pub PizzaToppings {
  Sausage,
  Pineapple,
  Pepperoni,
}

var favorite => Sausage; <> Works!
if (favorite == Pineapple) {
  println("You are mentally unstable.");
}
```

2.12.4 Complex Enumerators

Along with enums, you can add properties to each child. Adding properties essentially allows you to construct a unique instance of each child except it has additional properties that can be accessed. You can add properties to children by adding curly braces after the child name, and put each property seperated by a comma inside them. You can statically type each property by following it with a colon and then the type. You can then instance the child by calling it like a function.

```
enum Message {
  Quit,
  Move { x: num, y: num },
  Write { text: any },
  ChangeColor { r: num, g: num, b: num },
}

var write => Message::Write("mytext");
println(write::text); <> Prints "mytext"
```

To statically type enums further, you can use generic types with enum properties.

```
enum Option {
  Some(T){ val: T },
  None
}

var x => Some(13);
prinln(type(x)); <> Prints Option(num)
```

2.13 Structures

2.13.1 What are Structures?

Structures, or structs for short, are an easy way to create custom data structures. You can use structs to easily store several pieces of data all with unique names and access them later.

2.13.2 How to Make a Structure

You can define a structure similar to an enumerator. First, start with the keyword '**struct**', followed by the name of your data structure type.

After this comes the body. Open with curly braces, (`{}`), and list each attribute of the struct with commas in between. After closing with a curly bracket and a semicolon, your struct is complete.

```
struct Vector3 {
  x,
  y,
  z
}

<> Creates a new struct Vector3 with the attributes
<> x, y, and z.
```

2.13.3 Using Structures

You can create new instances of structures by calling them like a function and passing each attribute in. The order in which you pass the attributes in should follow the order of definition.

You can access data from a struct instance with the double colon operator, (`::`), followed by the attribute name.

```
struct Message {
  username,
  contents
}

var msg => Message("John Doe", "Hello world!");
<> Creates a new message instance.

println(msg::contents);
<> Gets the contents of the message and
<> prints them out.
```

2.14 Objects

*This chapter assumes you have read the **Functions** chapter.*

2.14.1 What are Objects?

Objects are essentially custom types that you can create. You can give them custom attributes and methods that they can execute, along with custom properties like handling for addition and other operations.

2.14.2 How to Make an Object

Due to this being JPizza, objects are defined using the '**class**' keyword, followed by the name of the object and curly braces, ({}). Inside the curly braces will be all of our code.

*Note: the original keyword for objects was **recipe** to fit the Pizza theme, but it was changed to **class** to be more standardized. You can also use **obj**!*

Attributes

How to Declare Attributes

There are many ways to declare attributes, but attributes must be declared otherwise they will not stay inside the object.

The simplest way is to simply list off the attribute names, each seperated by a comma. You can also list each off one at a time. To assign a default value to an attribute, you can have the attribute name followed by an arrow and then the value, like variable assignment. To statically type an attribute, you can follow the name with a colon and then the type.

Finally, you can add modifiers to attributes. You can use the `pub` keyword to make them public, meaning they can be accessed outside of the object, which is the default. Prefixing it with `prv` makes it private, so it can only be accessed internally. The `static` modifier makes it so that it can be accessed from the object directly, and not from an instance.

```
class Pizza {
  topping, breading;

  topping;
  breading;

  topping: String => "...";
  breading: String => "...";

  pub topping;
  prv breading => "Shh!";

  static topping => "topping being worked on...";
  static breading => "breading in progress...";
  <> Declares the attributes topping and breading in several different ways to show off␣
↪the different methods.
}
```

How to Access and Set Attributes

Attributes can be accessed similar to variables, by simply referencing the associated name. However, if there is a variable in scope of the same name, there are alternative methods. You can use the `this` keyword followed by the double colon operator, (`::`), and then the identifier. You can also use the **attr** keyword followed by the identifier.

To set attributes, you can set them similar to variables, by following the identifier with an arrow and a value, however if there is a varible in scope of the same name, you can use alternative methods. You can use the **attr** keyword followed by the identifier, an arrow, and a value.

Constructor

What is a Constructor?

The constructor is a function that is called when you make a new instance of the object. The constructor can accept arguments, or it can just run default code.

How to Make a Constructor

To define a constructor, use the '**ingredients**' keyword. Then follow it with arguments similar to a function. Angle brackets with the parameters inside, omitt if there aren't any.

Finally, use curly braces, ({}), with your constructor code inside.

```
class Pizza {
  topping, breading;
  <> Declares the attributes topping and breading.

  ingredients<t, b> {
    attr topping => t;
    attr breading => b;

    <> Creates a constructor that takes in t and b,
    <> then assigns toppings to t and breading to b.
  }

}
```

Complex Constructors

Constructors support the same complex features as functions, but constructors do not return a value.

You can also use generic typing on constructors the same as you would functions. To pass in generic types, you would simply append the generic types to the end of the class call in angle brackets.

```
var cls => GenericClass(args)<types>;
```

Methods

What is a Method?

Methods are functions that the object can execute from its current instance. Similar to functions, they can accept arguments, and they can use both brace and arrow form, but they cannot be anonymous.

How to Make a Method

To define a method, use the '**mthd**' keyword, followed by the arguments and the body. *Note: you can also use **md** or **method** instead of **mthd**.*

```
class Pizza {
  ... <> Previous code.
  mthd details {
    println("I am a pizza with " + this::breading
              + "breading and " + this::topping + ".");
  }
}
```

Complex Methods

Methods support static typing, asynchronous execution, generic typing, default arguments, and everything else functions have.

However, methods have another special trait, which is the '**bin**' keyword. **Bin** is short for built-in, and allows you to override built-in methods, like addition.

To make a method a **bin** method, simply write the **bin** keyword after the **mthd** keyword.

```
class Number {
  value;
  ingredients<val> {
    attr value => val;
  }

  mthd bin add<o> -> this::value + o::value
  <> Overrides the built-in addition function.

}
```

Methods also share the same modifiers as attributes, such as `pub`, `prv`, and `static`. By default, methods are public, but if you use a lot of different publicities, specifying `pub` may help.

To apply these modifiers, simply write them after the **mthd** keyword.

```
class MyCoolMethods {
  mthd pub publicMethod {
    println("I am public!");
  }

  mthd prv privateMethod {
```

(continues on next page)

(continued from previous page)

```
    println("I am private! Wait, how did you call me??");
  }

  mthd static staticMethod {
    println("You can call me anywhere!");
  }
}
```

2.14.3 Using Objects

To create a new instance of an object, you can call it like a function. Any arguments you pass in are passed into the constructor.

You can access methods and attributes of the instance using the double colon operator, (`::`). On the left of the operator should be the instance, and the attribute/method name should be on the right.

```
class Pizza {
  ...
}

var myPizza => Pizza("sausage", "pretzel");
<> Creates a new pizza, passing "sausage" and "pretzel" into the constructor.

myPizza::details();
<> Calls the details function of the pizza instance.
```

2.14.4 Complex Objects

Objects have some special properties that make things easier.

Inheritance

Inheritance simply means that we can give an object a parent object, and it "inherits" all its attributes, methods, and constructor properties. These can be overwritten simply by defining it in the object, but it's useful if you have several objects all with the same methods. You can just make a parent object!

To give an object a parent, you can use the temporary assignment arrow, (`->`), after the object name and follow it with the name of the parent object.

```
class Parent {
  inheritedAttribute;
  ingredients<x> {
    attr inheritedAttribute => x;
  }

  mthd inheritedFunction<y> -> this::inheritedAttribute + y

}

class Child -> Parent {
```

(continues on next page)

(continued from previous page)

```
  ingredients<x> {
    attr inheritedAttribute => x + 2;
    <> Overrides the default constructor and replaces it with this one, which adds 2 to␣
↪x before assigning it to v.
  }
}

println(Parent(5)::inheritedFunction(2));
<> Prints 7.

println(Child(5)::inheritedFunction(2));
<> Prints 9, since 2 is added to x in the overridden constructor.
```

2.15 Scopes

2.15.1 What is a Scope?

A scope can be thought of like a bubble that contains local variables, but once you exit the scope all the variables are lost. When you call a function, the code in the function is in a new scope, but then it exits the scope once the call finishes. This is why you can't access variables initialized inside the function.

2.15.2 How to Make a New Scope

You can create a scope using the **scope** keyword followed by the code you want to run inside curly braces. You can also name scopes by putting the name inside square brackets after **scope**. This allows for a cleaner traceback in runtime errors.

```
var x => 5;

scope [my_cool_scope] {
  var y => 15;
  println(x + y);
}
<> The variable y can no longer be used!

scope {
  println("Unnamed scope!");
}
```

2.15.3 Returning Values from Scopes

You can return values from scopes just like in functions, and you can then use that value in your code.

```
var x => scope {
  var y => 9 + 10;
  return y;
} <> Assigns x to the return value of the scope, 21 in this case.
```

2.16 Packages

2.16.1 Scripts

A useful feature of Pizza is the import and packages system.

You can import scripts with the '**import**' keyword followed by the script name, (*minus the file extension*). This allows you to access all the script's attributes like its functions, variables, classes, and other components. This can be done similarly to accessing class attributes, using the double colon operator, (`::`).

To import the script under a different name, you can use the **as** keyword followed by the preferred name.

You can import both local and global scripts. To make a script global, make a new folder under `~/.jpizza/modules/` with the same name as the script, (*minus the file extension*), then put the script in there.

```
<> otherScript.devp

fn sayHello -> println("Hello another world!");
```

```
<> mainScript.devp

import otherScript;

otherScript::sayHello();
<> Prints "Hello another world!".

import otherScript as o_script;
<> Imports otherScript with the name o_script.

o_script::sayHello();
<> Also prints "Hello another world!".
```

2.16.2 Extensions

You can extend JPizza with extensions written in Java that function as packs of new libraries. This can be done with the **extend** keyword followed by the extension name (*minus the file extension*). This gives you access to all of the libraries the extension has, which can then be imported.

You can extend both local and global extensions. To make an extension global, make a new folder under `~/.jpizza/extensions/` with the same name as the extension, (*minus the file extension*), and then put the jar in there.

```
extend MyCoolExtension;
import mycoollibrary;

mycoollibrary::myCoolFunction();
<> Runs myCoolFunction
```

2.16.3 Destructuring

You can use a feature known as destructuring to extract certain methods from imports and use them in the global scope. This can generally be applied to any data object such as classes and structs, but is most useful in the case of imports.

To destructure an object, simply do **var** followed by each attribute name separated by spaces inside of curly braces. Then follow that with an assignment arrow and the expression you want to destructure, in this case an import.

To destructure everything from an import, you can use a star (*) inside curly braces instead of names.

```
<> otherScript.devp

fn sayHello -> println("Hello another world!");

fn sayGoodbye -> println("Goodbye another world!");
```

```
<> mainScript.devp

var { sayHello } => import otherScript;
<> Moves sayHello from otherScript into the current scope.

sayHello();
<> Prints "Hello another world!".

var { * } => import otherScript;
<> Moves all functions from otherScript into the current scope.

sayGoodbye();
<> Prints "Goodbye another world!".
```

2.17 Headers

Your code functionality can be modified using header statements. Header statements start with a hash symbol, followed by the header name. If the header requires arguments, seperate each argument with a space.

An example of a header is the **func** header, which runs a function as a main function after the script is executed. This is useful if you want a main function that only runs when the script is executed, and not when it's imported or interacted with in other ways.

This header takes in one argument, the name of the main function. The main function must take in one argument, which is the command line arguments.

```
#func myMainFunction;

fn myMainFunction<args: list> {
```

(continues on next page)

(continued from previous page)

```
  println("Hello world!");
}

<> At the end of the script, myMainFunction is called.
```

Some headers are context based, meaning if you put them somewhere like inside a function, it will only affect how the function treats the code, but it won't affect the code globally.

This is useful for headers like **memoize**, which caches function inputs and instantly returns the value instead of running the function code if the same inputs are ever given to a function.

2.18 Error Handling

Error handling in JPizza is similar to a language called Rust. We can make functions return a **Resolve** object which contains an error and a value. If the function failed, the object will only contain the error, and trying to access the value will result in another error. You can check if a resolve is error free with the `ok` function.

2.18.1 How to Get a Resolve Object

To get a **Resolve** object, we need to make our function a catcher function. We can do this by append a pair of brackets `[]` to the end of a functions arguments. If no arguments, then name, no name then keyword. To create a demo function, we're going to use the '**throw**' keyword. This allows us to create a custom error by "throwing" the statement following it as an error message.

```
fn myIffyFunction[] {
  var r => random();

  if (r > 0.5)
    throw 'Fail!!!';

  return r;
}

<> This function will fail 50% of the time and return a random number the other 50%.
```

Note about 'throw': If you provide two strings separated by a comma, it will use the first string as the error type and the second as the details.

2.18.2 Handling Resolves

There are several special functions that we can use to handle **Resolves**.

The first function is `ok`. Doing `ok(res)` will return a bool determining whether the object is error free.

If our object is error free, we can safely call `resolve(res)`, which will return the value of the **Resolve**. Beware! If there is an error and no value, this will throw an error. Always do some form of `if (ok(res))` before.

The next function is `catch(res)`, which will return the error type and error message of the object as a list. We can then handle the error as we please.

If we want to throw the error anyway, we can call `fail(res)`, which throws the error stored.

```
<> Assume the "myIffyFunction" from earlier is defined here.

var res => myIffyFunction();
<> ^^ Assigns the result to res.
<> VV Checks if the result is error free.
if (ok(res)) {
  println(resolve(res));
  <> Prints the random number returned.

} else {
  println("Encountered error: " + catch(res));
  <> Prints "Encountered error: [<Error Name>, <Error Message>]".

}
```

2.19 Intended Errors

2.19.1 Throw

Like mentioned in the chapter 'Error Handling', we can use **throw** to throw a custom error. Throw can be used in two forms, we can provide a single string that will be used as the error message, or two strings that will be the error name and error message. If no error name is provided, it will default to "Thrown".

```
throw "Custom Error Type", "You suck!!";
throw "You suck, generally!!";
```

2.19.2 Assertion

Assertion allows you to throw errors if a given condition is false. Assertions can be defined by simply using the **assert** keyword followed by a condition.

```
assert "abc" == "abc"; <> Ok!
assert  123  ==  456 ; <> Uh oh! Assertion Error is brought up!
```

2.20 References

2.20.1 What is a Reference?

References work like boxes that contain a value. When a reference is copied or passed somewhere, it still contains the same value. References allow you to mutate the value inside, and it modifies the value across all references that use it.

2.20.2 How to Make a Reference

You can make a reference using an ampersand (&) followed by the expression. You can dereference something, or get the value the reference is pointing to, by using a star (*).

Assignment operations like =>, +=, ++, and more are implemented on references, and directly mutate the value. This allows you to do things like edit items at list indices directly by making a list of references.

```
var lis => [&1, &2, &3, &4, &5];

lis[0] => 23;
lis[1]++;
lis[2] -= 15;

println(lis);
<> Prints [23, 3, -12, 4, 5].

println(3 + *lis[0]);
<> Prints 26.
```

CHAPTER

THREE

NATIVE FUNCTIONS

3.1 In / Out

- `print<value>` : Prints the given value.
- `println<value>` : Prints the given value along with a new line.
- `printback<value>` : Prints and returns the given value.
- `field<prompt>` : Prompts the user with the given text and waits for an input.
- `nfield<prompt>` : Prompts the user with the given text and waits for a numerical input.
- `clear` : Clears the console
- `sim<text>` : Simulates the text as code.
- `run<filepath>` : Runs the given file.

3.2 Type Checking

- `isNumber<value>` : Returns if the given value is a number.
- `isString<value>` : Returns if the given value is a string.
- `isList<value>` : Returns if the given value is a list.
- `isFunction<value>` : Returns if the given value is a function.
- `isBoolean<value>` : Returns if the given value is boolean.
- `type<value>` : Returns the type of the given value as a string.

3.3 List Modifiers

- `append<list, value>` : Appends the given value to the list.
- `remove<list, value>` : Removes the given value from the list.
- `pop<list, index>` : Removes the item at the given index from the list.
- `extend<list, list>` : Returns the concatenation of the two lists.
- `insert<list, item, index>` : Inserts the given item at the given index of the list.
- `setIndex<list, item, index>` : Sets the given index to the given item of the list.

- `size<list>` : Returns the size of the given list.
- `choose<list>` : Chooses a random item from the list.
- `contains<list, value>` : Returns true if the list contains the value.
- `sublist<list, start, end>` : Returns the sublist between the start and end indices.
- `join<str, list>` : Returns a string of each item in the list seperated by `str`, for example: `join(" ", ["Hello,", "World!"]) -> "Hello, World!"`
- `indexOf<list, item>` : Returns the index of the item in the list. Returns -1 if the item is not in the list.

3.4 Dictionary Modifiers

- `get<dict, key>` : Returns the value of a key in the dictionary.
- `set<dict, key, value>` : Sets the value of a key in the dictionary.
- `overset<dict, key, value>` : Replaces the value of a key in the dictionary if it exists.
- `delete<dict, key>` : Removes a key from the dictionary.
- `contains<dict, value>` : Returns true if the dictionary contains the value.

3.5 Numerical Functions

- `log<value, base>` : Gets the logarithm of the provided value using the base.
- `round<value>` : Rounds the value to the nearest whole number.
- `floor<value>` : Rounds the value down.
- `ceil<value>` : Rounds the value up.
- `abs<value>` : Gets the absolute value of a number.
- `random` : Returns a random number between 0 and 1.
- `randint<min, max>` : Returns a random number between the min and max.
- `floating<value>` : Returns if the number is a floating point number.
- `max<a, b>` : Returns the maximum value between `a` and `b`.
- `min<a, b>` : Returns the minimum value between `a` and `b`.
- `sin<x>` : Returns the sine of `x`.
- `cos<x>` : Returns the cosine of `x`.
- `tan<x>` : Returns the tangent of `x`.
- `arcsin<x>` : Returns the inverse sin of `x`.
- `arccos<x>` : Returns the inverse cosine of `x`.
- `arctan<x>` : Returns the inverse tangent of `x`.
- `arctan2<y, x>` : Returns the inverse tangent of `y` and `x`.
- `doubleStr<number, precision>` : Returns the number as a string with the given decimal precision.

3.6 Type Conversions

- `str<value>` : Converts the given value to a string.
- `num<value>` : Converts the given value to a number.
- `bool<value>` : Converts the given value to boolean.
- `list<value>` : Converts the given value to a list.
- `dict<value>` : Converts the given value to a dictionary.
- `byter<value>` : Converts a list of numbers into a byte array.
- `chr<value>` : Converts a number into a character.
- `chrs<value>` : Converts a list of numbers into a string.

3.7 String Modifiers

- `split<string, delimiter>` : Splits the string into a list every time the delimiter is found.
- `contains<string, value>` : Returns true if the string contains the value.
- `strUpper<string>` : Returns the string with all letters uppercase.
- `strLower<string>` : Returns the string with all letters lowercase.
- `strShift<string>` : Returns the string with all characters in the form they would be if you were to hold shift.
- `strUnshift<string>` : Returns the string with all characters in their form if you were to not hold shift. (Inverse of `strShift`)
- `substr<str, start, end>` : Returns the substring between the start and end indices.
- `replace<str, old, new>` : Replaces each instance of `old` in `str` with `new`.
- `escape<str>` : Escapes every escape sequence in the string. For example, `"\\n"` turns into `"\n"`.
- `unescape<str>` : Unescapes every escape sequence in the string. For example, `"\n"` turns into `"\\n"`.
- `parseNum<str>` : Parses the string as a number.

3.8 Object Functions

- `getattr<instance, attribute>` : Uses the vanilla get method to access the instances value.
- `hasattr<instance, attribute>` : Returns true if the instance has the attribute in question.

CHAPTER

FOUR

BIN METHODS

4.1 Type Methods

- `number` : Returns the object as a number.
- `boolean` : Returns the object as boolean.
- `dictionary` : Returns the object as a dictionary.
- `function` : Returns the object as a function.
- `list` : Returns the object as a list.
- `string` : Returns the object as a string.
- `type` : Returns the object's type as a string.

4.2 Numerical Operations

- `add<other>` : Returns the sum of the object and other.
- `sub<other>` : Returns the difference of the object and other.
- `mul<other>` : Returns the product of the object and other.
- `div<other>` : Returns the quotient of the object and other.
- `fastpow<other>` : Returns the object raised to the power of other.
- `mod<other>` : Returns the object modulo other.

4.3 Collection Operations

- `get<other>` : Returns `object.other`.
- `bracket<other>` : Returns `object[other]`.

4.4 Conditionals

- `eq<other>` : Returns true if the object equals other.
- `ne<other>` : Returns true if the object doesn't equal other.
- `lt<other>` : Returns true if the object is less than other.
- `lte<other>` : Returns true if the object is less than or equal to other.

CHAPTER

FIVE

HEADERS

5.1 Execution

- `func <functionName>` : If the program is being directly executed, at the end of the script it will run the function specified and pass in a list of command line arguments as a parameter.
- `object <recipeName>` : If the program is being directly executed, at the end of the script it will call the static method named main of the given class and pass in a list of command line arguments as a parameter.

5.2 Optimization

- `memoize` : Caches function calls so when functions are called with arguments that have been called before, it will automatically return the value instead of running the function.

5.3 Exports

- `export <args>` : Allows only the specified arguments to be imported into other scripts.
- `export_to <target>` : Changes what types of files can import the script.
  - `all` : All files can import the script.
  - `package` : Only scripts in the same package can import the script.
  - `none` : No files can import the script.
- `package <packageName>` : Changes the package the script is in.

CHAPTER

SIX

NATIVE LIBRARIES

6.1 Time

```
import time;
```

- `halt<ms>` : Pauses the program for the specified amount of milliseconds.
- `stopwatch<func>` : Returns the amount of milliseconds taken to execute the specified function.
- `epoch` : Gets the current epoch time in milliseconds.

6.2 Gens

```
import gens;
```

- `range<start, stop, step>` : Returns a list of numbers starting at `start` and ending at `stop`, increasing by `step`.
- `linear<start, stop, step, slope, y-inter>` : Returns a list of numbers starting at `start` and ending at `stop`, increasing by `step` with the function `x -> slope * x + y-inter` applied to each.
- `quadratic<start, stop, step, a, b, c>` : Returns a list of numbers starting at `start` and ending at `stop`, increasing by `step`, with the function `x -> a * x * x + b * x + c` applied to each.

6.3 IOFile

```
import iofile;
```

6.3.1 String Data

- `readFile<dir>` : Reads the contents of the specified file and returns the contents as a string.
- `writeFile<dir, string>` : Writes the string to the specified file. If the file does not exist, it will be created. Returns true if the file had to be created, false otherwise.

6.3.2 File Creation

- `fileExists<dir>` : Returns true if the specified file exists, else false.
- `makeDirs<dir>` : Makes all missing directories along the path.
- `deleteFile<path>` : Deletes the file at the given path.

6.3.3 Directories

- `listDirContents<dir>` : Returns a list of the directories contents.
- `isDirectory<path>` : Returns true if the given path is a directory, else false.

6.3.4 Working Directory

- `setCWD<dir>` : Sets the current working directory to the given directory.
- `getCWD` : Returns the path of the current working directory.

6.3.5 Serialization

- `readSerial<dir>` : Reads and deserializes a stored JPizza object in the specified file and returns the object.
- `readBytes<dir>` : Reads the contents of the specified file and returns the contents as a byte array.
- `writeSerial<dir, value>` : Serializes the given value and stores it at the given path. If the value is a bytearray, the bytes will be written to the file. If the file does not exist, it will be created. Returns true if the file had to be created, false otherwise.

6.4 Sys

```
import sys;
```

- `os` : Returns the users OS as a string.
- `home` : Returns the users home path.
- `execute<cmd>` : Executes the given command in the command line and returns the output.
- `executeFloor<cmdArgs>` : Executes the given command arguments in the command line and returns the output.
- `disableOut` : Disables output to the console.
- `enableOut` : Enables output to the console.
- `jpv` : Returns the current JPizza version.
- `envVarExists<var>` : Returns if an environment variable exists or not.
- `getEnvVar<var>` : Returns the value of the given environment variable. Throws an error if it doesn't exist.
- `setEnvVar<var, value>` : Sets the given environment variable to the given value.
- `getProp<prop>` : Gets the given system property.
- `setProp<prop, val>` : Sets the given system property to the given value.

- `exit<code>` : Exits the program with the given code.

6.5 HTTPx

```
import httpx;
```

- `getRequest<url, headers>` : Sends a **GET** request to the given url with the given headers.
- `deleteRequest<url, headers>` : Sends a **DELETE** request to the given url with the given headers.
- `postRequest<url, headers, body>` : Sends a **POST** request to the given url with the given headers and body.
- `traceRequest<url, headers>` : Sends a **TRACE** request to the given url with the given headers and body.
- `patchRequest<url, headers, body>` : Sends a **PATCH** request to the given url with the given headers and body.
- `putRequest<url, headers, body>` : Sends a **PUT** request to the given url with the given headers and body.
- `optionsRequest<url, headers>` : Sends an **OPTIONS** request to the given url with the given headers and body.
- `connectRequest<url, headers>` : Sends a **CONNECT** request to the given url with the given headers and body.
- `headRequest<url, headers>` : Sends a **HEAD** request to the given url with the given headers and body.

6.6 Pretzel

```
import pretzel;
```

- `init<host, port>` : Initializes the webserver on `host:port`.
- `route<addr, func>` : Whenever `addr` is queried, it sends data about the request to the function passed in, and sends back the data the function returns.
- `start` : Starts the webserver.

6.6.1 Data

Route functions are given data that looks like this:

```
{
    "method": "Request Method, usually GET or POST",
    "uri": "The uri used, like /test?abc=xyz",
    "body": "The body passed in.",
    "headers": {
        "the": ["headers given"],
        "from": ["post requests"]
    }
}
```

The expected return type should be of:

```
{
    "code": 200, // The response code, like 200, 400, 404, etc.
    "header": "<h1>The value to return</h1>"
}
```

6.7 Puddle

```
import pdl;
```

6.7.1 Server Functions

- `host<port>` : Starts a server on the given port and returns the server ID.
- `accept<id>` : Accepts a connection from the server with the given ID and returns a connection ID.

6.7.2 Connection Functions

- `connect<host, port>` : Connects to the given host and port and returns a connection ID.
- `write<id, bytes, offset, len>` : Writes the given bytes to the connection with the given ID.
- `read<id, offset, len>` : Reads bytes from the connection with the given ID.

6.8 JSON

```
import json;
```

- `loads<string>` : Converts JSON string to a list/dictionary.
- `dumps<object>` : Converts a list/dictionary to a JSON string.

6.9 AWT

```
import awt;
```

Colors for `awt` should be a Tuple of 3 integers in the range 0 to 255, like `Tuple(255, 0, 0)` for red. `Tuple(r, g, b)`

To start using `awt`, you must call `awt::init()`. If not, all other functions will fail!

6.9.1 Window Functions

You can create multiple windows, but only one can be active at a time.

The library stores an array of all windows, and the active window. You can change the index of the active window by calling `awt::setWindow(index)`.

The library has a "focus window" that when closed, will exit the program. You can change the focus window by calling `awt::focuswWindow()` to focus the current window.

- `awt::createWindow` : Creates a new window and returns its index.
- `awt::setWindow<index>` : Sets the active window to the given index.
- `awt::focusWindow` : Sets the focus window to the current window.
- `awt::windowIndex` : Returns the index of the current window.
- `awt::windowCount` : Returns the number of windows.
- `awt::width` : Returns the width of the current window.
- `awt::height` : Returns the height of the current window.

6.9.2 Draw Functions

- `drawPoly<points, color>` : Draws a polygon with the vertices being the given points and shaded in the given color.
- `tracePoly<points, color>` : Draws the outline of a polygon with the vertices being the given points and shaded in the given color.
- `drawOval<x, y, width, height, color>` : Draws an oval of the given color with the center at (x, y) with dimensions width x height.
- `drawCircle<r, x, y, color>` : Draws a circle of the given color with the center at (x, y) and a radius of r.
- `drawRect<x, y, width, height, color>` : Draws a rectangle of the given color with the center at (x, y) with dimensions width x height.
- `drawSquare<length, x, y, color>` : Draws a square of the given color with the center at (x, y) with dimensions length x length.
- `drawText<text, x, y, color>` : Writes text at (x, y) in the given color.
- `drawImage<filepath, x, y>` : Draws the image at the given path at (x, y).
- `drawSizedImage<filepath, x, y, w, h>` : Draws the image at the given path at (x, y) with the given dimensions w x h.
- `setPixel<x, y, color>` : Sets the pixel at (x, y) to the given color.
- `drawLine<start, end, color>` : Draws a line from the start point to the end point in the given color.

6.9.3 Config

- `setTitle<title>` : Sets the window title to the given title.
- `setSize<width, height>` : Sets the window dimensions to width x height.
- `setIcon<filepath>` : Sets the window icon to the given image.
- `setFont<name, type, size>` : Sets the current font to the given font with formatting of the given type and of the given size. Types include:
  - `"B"` -> Bold
  - `"I"` -> Italic
  - `"P"` -> Plain
- `setBackgroundColor<color>` : Sets the background to the given color.
- `lockSize<bool>` : Disables/enables the ability to resize the window.
- `setStrokeSize<width>` : Changes the stroke size.
- `exit` : Exits the window.

6.9.4 Rendering

- `start` : Starts rendering and opens the window.
- `clear` : Clears the canvas.
- `refresh` : Refreshes the canvas.
- `refreshLoop<refreshRate>` : Refreshs the canvas at the given refresh rate in the background.
- `refreshUnloop` : Stops the refresh loop.
- `screenshot<filepath>` : Saves the current canvas to an image file which will be stored at the given path.
- `fps` : Returns the given FPS.
- `gpuCompute<bool>` : Sets gpu computing to true/false.

6.9.5 QRendering

QRendering is an alternative to rendering. With the default rendering, any functions that modify the canvas, like drawing a circle, directly modify the canvas as soon as they're called. With QRendering, the functions are put on to a sort of stack, and when you call the update command they are all drawn at once. This can help when you're doing lots of drawing, as you might rerender the screen while only half of your drawing is complete. QRendering makes it so that everything will render at once.

- `toggleQRender` : Toggles QRendering.
- `qrender` : Toggles QRendering (alias for toggleQRender).
- `qUpdate` : Pushs all the changes to the canvas.

6.9.6 Mouse Input

- `mouseDown<button>` : Returns if the given button is being held down. `0` is LMB, `1` is MMB, `2` is RMB.
- `mousePos` : Returns the given mouse position as `[x, y]`.
- `mouseIn` : Returns if the mouse is in the window.

6.9.7 Keyboard Input

Keys should be given as strings, like `"a"`, `" "`, or `"enter"`.

- `keyDown<key>` : Returns if the given key is being held down.
- `keyTyped<key>` : Returns if the key was pressed.
- `keyString` : Returns a string of all the pressed keys, like `"abc"` if keys A, B, and C were pressed.

6.9.8 Audio

- `playSound<filepath>` : Plays the sound at the path.

6.9.9 General Functions

- `chooseFile<path, filter, mode>` : Opens a file picker and returns the chosen file. Originates at the given path. Opens a save prompt if the mode is `awt::SAVE`, opens an open prompt if the mode is `awt::OPEN`. If the given filter is `null`, no filter will be applied. Otherwise, a filter should be given in the format `["Message to display", "extension"]`, such as `["PNG files (*.png)", "png"]`.

6.9.10 Boilerplate

Some simple boilerplate code for creating an app is provided below.

```
var { Tuple } => import std;
import awt;
awt::init();

awt::setSize(800, 800);
awt::start();

awt::refreshLoop(1000 / 60);
awt::toggleQRender();

awt::drawCircle(100, 400, 400, Tuple(50, 50, 50));
awt::qUpdate();
```

CHAPTER

SEVEN

STANDARD LIBRARIES

7.1 STD

```
import std;
```

7.1.1 Enumerators

Option

Public

Serves as a way to dynamically and safely return a nullable value.

Best served with pattern matching.

- `Some { val }`
- `None`

StaticOption

Public

Serves as a way to statically and safely return a nullable value.

Best served with pattern matching.

- `Box(T) { val:  T }`
- `Empty(T)`

7.1.2 Classes

Array Class

`Array(..items)<T>;` The Array class serves as an object oriented statically typed list. You can, of course, use the **any** type to dynamically type it.

- `add<item #T>` - Adds the given item to the array.
- `remove<item #T>` - Removes the given item from the array.
- `insert<item #T, index #num>` - Inserts the given item at the given index.

- `iter<func #function>` - Returns an *Iter* class that uses the given function and iterates over each item in the array.
- `pop<index #num>` - Pops the given index from the array and returns the item.
- `size` - Returns the size of the array.
- `addAll<..items>` - Adds all the items to the array.
- `slice<start #num, end #num>` - Returns a slice of the array between indices `start` and `end`.
- `indexOf<item>` - Returns the index of the given item.
- `contains<item>` - Returns true if the array contains the given item, else false.
- `join<str #String>` - Returns a string where each item in the list is seperated by the given str.

Map Class

`Map(..pairs)<K, V>`; The Map class serves as an object oriented statically typed dictionary. You can, of course, use the **any** type to dynamically type it.

When constructing the map, you should provide each key value as pair, such as `["a", "b"]` or `[1, 2]`.

- `get<other #K>` - Returns the value associated with the given key. Fails if the key is not in the map.
- `getOrDefault<other #K, default>` - Returns the value associated with the given key. If the key is not in the map, returns the provided default.
- `set<key #K, value #V>` - Sets the given key in the map to the given value.
- `del<key #K>` - Deletes the given key from the map.
- `iter<func #function>` - Returns an *Iter* class that uses the given function and iterates over each pair in the map.
- `size` - Retruns the size of the map.
- `contains<key>` - Returns true if the map contains the given key, else false.
- `keyArray` - Returns an array of each key in the map.
- `pairArray` - Returns an array of each key-value pair.

Iter Class

Should not be constructed manually Allows you to iterate over values in unique fashions.

- `collect` - Returns each item iterated over as a new object.
- `stream` - Iterates over each item without returning any values.

Tuple Class

`Tuple(..items);` Offers as a way to statically return several different types in an immutable fashion.

- `size` - Returns the size of the tuple.
- `contains<item>` - Returns true if the tuple contains the given item, else false.

7.2 Socks

```
import socks;
```

7.2.1 SocketConnection Object

Note: *You will never construct this object yourself.*

- `send<msg>` - Sends the client the given message.
- `sendBytes<bytearray>` - Sends the client the given byte array.
- `recv` - Recieves data from the client.
- `recvBytes<length>` - Recieves a byte array from the client of the given length.
- `recvAllBytes` - Recieves a byte array from the client.
- `close` - Closes the connection.

7.2.2 Socket Object

`Socket(port#num);`

- `listen` - Returns a SocketConnection Object once a client connects to the server.
- `close` - Closes the server and all its SocketConnections.

7.2.3 SocketClient Object

`SocketClient(host#String, port#num);`

- `send<msg>` - Sends the server the given message.
- `sendBytes<msg>` - Sends the server the given message.
- `recv` - Recieves data from the server.
- `recvBytes<length>` - Recieves data from the server of the given length.
- `recvAllBytes` - Recieves data from the server.
- `close` - Closes the connection.

CHAPTER

EIGHT

RESOURCES

- VSCode - A Visual Studio Code plugin that adds syntax highlighting support.
- Video Tutorials - A video tutorial series over JPizza2 and some of its native libraries.
- GitHub Repo - The open source GitHub repository for the latest version of JPizza.
- Discord - The official JPizza discord.