

The Motoko Programming Language

Last updated: 2025-03-13

Download v1.1 

Links

[GitHub Repo](#)

[Official Motoko Docs](#)

[Official Base Library Reference](#)

[Official ICRC1 Docs](#)

Author

Written by [@SamerWeb3](#)

Special thank you to OpenAI's ChatGPT for helping me learn quickly and assisting me with writing this text!

"Democratize access to the tech economy and make everyone industrious"

~Dominic Williams

Introduction

Motoko is a programming language developed by DFINITY for writing advanced blockchain programs ([smart contracts](#)) on the [Internet Computer](#).

Blockchain programs are a new kind of software that runs in the form of [canisters](#) on the Internet Computer blockchain instead of a personal computer or a data center server. This has several advantages:

- **Security:** Blockchain programs are guaranteed to run the way they are programmed and are therefore tamper-proof.
- **Autonomy:** Blockchain programs can run 'on their own' without an owner or controller.
- **Programmable scarcity:** Digital valuable items like currencies and tokens are just numbers in computer code.
- **Reduced Complexity:** Eliminates the need for traditional IT stack
- **Scalability:** Asynchronous message passing between [actors](#) allows for scalable systems

Motoko allows writing a broad range of blockchain applications from small microservices to large networks of smart contracts all interacting to serve millions of users. In this book we review the Motoko language features and describe how to use them with examples.

We start with [common programming concepts](#) found in any language. We then move on to special features for [programming on the Internet Computer](#). We also review the [Motoko Base Library](#) and tackle [advanced programming concepts](#). And finally, we go over [common canisters](#) (smart contracts) in the Internet Computer ecosystem and also describe common [community standards](#) within the context of Motoko Programming.

Along the way, we will develop ideas and techniques that allow us to build a [Tokenized Comments](#) section that demonstrates many cool features of Motoko and the Internet Computer.

Getting Started

[Motoko Playground](#) is the easiest way to deploy a *Hello World* example in Motoko.

Motoko Playground is an online code editor for Motoko. It allows us to use Motoko without having a [local development environment](#). It simplifies the process of deploying Motoko code to the Internet Computer.

Hello World!

1. Open [Motoko Playground](#) and click on `Hello, world` in the list of examples.
2. Then click on the `Main.mo` file in the files list on the left.
3. Checkout the code and hit the `Deploy` button on the top right.
4. Finally, click the `Install` button in the popup window.

Motoko Playground will now deploy your code to the Internet Computer!

You should see a `Candid UI` on the right hand side of your screen. Write something in the text box and click the `QUERY` button. The output should be displayed in the `OUTPUT LOG`.

Next steps

Exploring the Motoko Playground will help you gain familiarity with the basics of Motoko and related concepts.

It is recommended to explore the example projects. Taking the time to read every line, instead of just *skimming* through the code, will greatly increase your understanding.

While exploring you will encounter things you don't understand. Be comfortable with that and continue exploring carefully.

Whenever you feel ready, consider [installing](#) the Canister Development Kit.

How it works

When you deploy code in Motoko Playground, a couple of things happen:

- The Motoko code is compiled into a [Webassembly module](#).
- A new [canister](#) is created on the Internet Computer.
- The Webassembly module is *installed* in the canister.
- A [candid](#) interface file will be generated
- The Candid UI allows you to interact with the [actor](#) that is defined in the Motoko code

Common Programming Concepts

This chapter covers the fundamental building blocks of Motoko. If you are coming from another language, you are likely familiar with many of these concepts and you may want to skip this chapter.

Important topics in this chapter are (amongst other basic topics):

- Mutable vs. immutable variables
- Type annotation
- Objects, classes and modules

Variables

A variable is a *value* that has a *name*. It is defined by a *declaration*.

A variable is declared as follows:

```
let myVariable = 0;
```

The `let` keyword indicates we are declaring a variable. The name of the variable is `myVariable`. The value that is named is `0`.

Variable names must start with a letter, may contain lowercase and uppercase letters, may also contain numbers 0-9 and underscores `_`.

The convention is to use `lowerCamelCase` for variable names.

`let` declarations declare *immutable* variables. This means the value cannot change after it is declared.

Some examples of immutable variables:

```
let x = 12;  
let temperature = -5;  
let city = "Amsterdam";  
let isNotFunny = false;
```

A declaration ends with a *semicolon* `;`

The convention is to use spaces around the equals sign.

The declarations above all span one line but a declaration may span several lines as long as it ends with a semicolon.

Mutability

Variables that can change their value *after* declaring them are called *mutable variables*. They are declared using a `var` declaration:

```
var x = 1;
```

`x` is now the name of a mutable variable with the value `1`. If we want to change the value, in other words *mutate* it, we *assign* a new value to it:

```
x := 2;
```

`x` now has the value `2`.

The `:=` is called the assignment *operator*. It is used to assign a new value to an already declared variable.

We can change the value of a mutable variable as many times as we like. For example, we declare a mutable variable named `city` with text value `"Amsterdam"`

```
var city = "Amsterdam";
```

And we also declare two immutable variables.

```
let newYork = "New York";
let berlin = "Berlin";
```

Now we mutate `city` three times:

```
city := newYork;
city := berlin;
city := "Paris";
```

The last mutation was achieved by assigning a *string literal* to the variable name. The first two mutations were achieved by assigning the value of other immutable variables. It is also possible to assign the value of another mutable variable.

Comments

A one-line comment is written by starting the line with `//`.

```
// This is a comment
```

It's always recommended to clarify your code by commenting:

```
// A constant representing my phone number
let phoneNumber = 1234567890;
```

A comment could start at the end of a line:

```
var weight = 80; // In kilo grams!
```

And comments could span multiple lines by enclosing it in `/** */`:

```
/*
Multiline comment
A description of the code goes here
*/
```

Types

A type describes the *data type* of a value. Motoko has static types. This means that the type of every value is known when the Motoko code is being *compiled*.

Motoko can in many cases know the type of a variable without you doing anything:

```
let x = true;
```

In the example above the `true` value of variable name `x` has the `Bool` type. We did not state this explicitly but Motoko *infers* this information automatically for us.

In some cases the type is not obvious and we need to add the type ourselves. This is called *type annotation*. We can annotate the name of the variable like this:

```
let x : Bool = true;
```

With the colon `:` and the name of the type after the variable name, we tell Motoko that `x` is of type `Bool`.

We can also annotate the value:

```
let x = true : Bool;
```

Or both:

```
let x : Bool = true : Bool;
```

In this case it is unnecessary and makes the code ugly. The convention is to leave spaces around the colon.

The type keyword

We can always *rename* any type by using the `type` keyword. We could rename the `Bool` type to `B`:

```
type B = Bool;
```

```
let boolean : B = true;
```

We defined a new *alias* for the `Bool` type and named it `B`. We then declare a variable `boolean` of type `B`.

Primitive types

Primitive types are fundamental core data types that are not composed of more fundamental types. Some common ones in Motoko are:

- `Bool`
- `Nat`
- `Int`
- `Float`
- `Text`
- `Principal`
- `Blob`

See the [full list of all Motoko data types](#)

We can define arbitrary names for any type:

```
type Age = Nat;
```

This creates an alias (a second name) `Age` for the `Nat` type. This is useful for writing clear readable code. The convention is to use type names that start with a capital letter.

The variable name `age` is of type `Age`.

The unit type

The last type we will mention in this chapter is the *unit type* `()`. This type is also called the empty `tuple` type. It's useful in several places, for example in `functions` to indicate that a function does not return any specific type.

For now let's just look at one *ugly, strange and useless*, yet legal Motoko code example for the sake of learning:

```
let unitType : () = () : () ;
```

We declared a variable named `unitType` and type annotated this variable name with the unit type. Then we assigned the empty `tuple` value `()` to it and also annotated this value with the unit type.

Observe that we type annotate twice, once on the left hand side of the assignment and the other on the right hand side, like we did for variable `x` above.

Tuples

A tuple type is an ordered sequence of possibly different types enclosed in parenthesis:

```
type MyTuple = (Nat, Text);
```

Here is a variable with the `MyTuple` type.

```
let myTuple: MyTuple = (2, "motoko");
```

We can access the values of the tuple like this:

```
let motoko = myTuple.1;
```

By adding `.1` to `myTuple` we access the second element of the tuple. This is called *tuple projection*. The indexing starts at 0.

Another example:

```
let profile : (Text, Nat, Bool) = ("Anon", 100, true);
```

A tuple type is created and used without using an alias. The variable name `profile` is annotated with the tuple type. The value assigned to the variable is a tuple of values `("Anon", 100, true)`;

We access the first element like this:

```
let username: Text = profile.0;
```

Records

Records are like a *collection* of named values (variables). These values could be mutable or immutable. We assign a record to a variable name `peter`:

```
let peter = {
    name = "Peter";
    age = 18;
};
```

The record is enclosed with curly brackets `{}`. The example above is a record with two *field expressions*. A field expression consists of a variable name and its assigned value. In this case `name` and `age` are the names. `Peter` and `18` are the values. Field expressions end with a semicolon `;`

We could annotate the types of the variables like this:

```
let peter = {
    name : Text = "Peter";
    age : Nat = 18;
};
```

The record now has two type annotated fields. The whole record also has a type. We could write:

```
type Person = {
    name : Text;
    age : Nat;
};
```

This type declaration defines a new name for our type and specifies the type of the record. We could now start using this type to declare several variables of this same type:

```
let bob : Person = {
    name = "Bob";
    age = 20;
};

let alice : Person = {
    name = "Alice";
    age = 25;
};
```

Another example is a record with mutable contents:

```

type Car = {
    brand : Text;
    var mileage : Nat;
};

let car : Car = {
    brand = "Tesla";
    var mileage = 20_000;
};

car.mileage := 30_000;

```

We defined a new type `Car`. It has a mutable field `var mileage`. This field can be accessed by writing `car.mileage`. We then mutated the value of the mutable `mileage` variable to the value `30_000`;

Note, we used an underscore `_` in the natural number. This is allowed for readability and does not affect the value.

Object literal

Records are sometimes referred to as *object literals*. They are like the **string literals** we saw in earlier chapters. Records are a 'literal' value of an object. We will discuss **objects** and their types in an upcoming chapter.

In our examples above, the literal value for the `bob` variable was:

```
{
    name = "Bob";
    age = 20;
}
```

Variants

A variant is a type that can take on *one* of a fixed set of values. We define it as a set of values, for example:

```
type MyVariant = {
    #Black;
    #White;
};
```

The variant type above has two variants: `#Black` and `#White`. When we use this type, we have to choose one of the variants. Lets declare a variable of this type and assign the `#Black` variant value to it.

```
let myVariant : MyVariant = #Black;
```

Variants can also have an associated type attached to them. Here's an example of variants associated with the `Nat` type.

```
type Person = {
    #Male : Nat;
    #Female : Nat;
};

let me : Person = #Male 34;

let her : Person = #Female(29);
```

Note the two equivalent ways of using a variant value. The `#Male 34` defines the `Nat` value separated by the variant with a space. The `#Female(29)` uses `()` without a space. Both are the same.

Variants can have different types associated to them or no type at all. For example we could have an `OsConfig` variant type that in certain cases specifies an OS version.

```
type OsConfig = {
    #Mac;
    #Windows : Nat;
    #Linux : Text;
};

let linux = #Linux "Ubuntu";
```

In the case of the `#Linux` variant, the associated type is a `Text` value to indicate the Linux Distribution. In case of `#Windows` we use a `Nat` to indicate the Windows Version. And in case of `#Mac` we don't specify any version at all.

Note that the last variable declaration is not type annotated. That's fine, because Motoko will infer the type that we declared earlier.

Immutable Arrays

Arrays in Motoko are like an ordered sequences of values of a certain type. An immutable array is an ordered sequence of values that can't change after they are declared.

Here's a simple array:

```
let letters = ["a", "b", "c"];
```

We assigned an array value to our variable `letters`. The array consists of values of a certain type enclosed in angle brackets `[]`. The values inside the array have to be of the same type. And the whole array also has an *array type*.

```
type Letters = [Text];
let letters : Letters = ["a", "b", "c"];
```

We declare an array type `[Text]` and named it `Letters`. This type indicates an array with zero or more values of type `Text`. We did not have to declare the type up front. We could use the array type to annotate any variable:

```
let letters : [Text] = ["a", "b", "c"];
```

We used the type `[Text]` to annotate our variable. The array values now have to be of type `Text`.

We can access the values inside the array by *indexing* the variable. This is sometimes called *array projection*:

```
let a : Text = letters[0];
```

Indexing the variable `letters` by adding `[0]` allows us to access the first element in the array. In the example above, the variable `a` now has text value `"a"` which is of type `Text`.

But we have to take care when we try to access values inside an array. If we choose an index that does not exist in our array, the program wil stop executing with an error!

```
// WARNING: this will throw an out of bounds error
let d = letters[3];
```

To avoid indexing into an array outside its bounds, we could use a *method* that is available on all array types called `size()`. A method is just a **function** that is called on a named value.

```
let size : Nat = letters.size();
```

We now declared a variable named `size` which is of type `Nat` and assign the value returned by our method `.size()`. This method returns the total length of the array. In our case, this value would be `3`.

WARNING: Be careful, the last element of the array is `size - 1`! See example in [mutable arrays](#).

Arrays and mutable variables

An immutable array could be assigned to a mutable variable. The array values are still immutable, but the value of the variable (which is an immutable array) could change.

```
var numbers : [Nat] = [5, 3, 7];  
numbers := [2, 6, 7, 4, 7];
```

We declare a mutable variable `numbers` of type `[Nat]` and assign an array of `Nat` values to it. We could not change any of the values if we wanted, but we could assign a whole new array to the variable.

In the second line, we assigned another value of type `[Nat]` to our variable `numbers`. The variable `numbers` could be mutated many times as long as its assigned value is of type `[Nat]`.

Mutable Arrays

Mutable arrays are a sequence of ordered *mutable* values of a certain type. To define a mutable array, we use the `var` keyword *inside* the array.

```
let letters = [var "a", "b", "c"];  
let a : Text = letters[0];
```

We declared an *immutable variable* named `letters` and assigned an array value to it. Our array has the keyword `var` inside of it after the first bracket to indicate that the values are mutable. The `var` keyword is used only once at the beginning.

Notice, that array indexing works the same as for a immutable array.

We could be more explicit about the type of our variable by annotating it:

```
let letters : [var Text] = [var "a", "b", "c"];
```

Our mutable array is of type `[var Text]`. We could now mutate the values inside the array, as long as we assign new values of type `Text`.

```
letters[0] := "hello";
```

We can mutate values as many times as we like. Lets change the last value of our array:

```
let size = letters.size();  
letters[size - 1] := "last element";
```

We used the `.size()` method to obtain a `Nat` and used that to index into the array, thereby accessing the last element of the array and giving it a new `Text` value. The last element is `size - 1` because array indexing starts at 0 and the `.size()` method counts the size of the array starting at 1.

Our array has now the following value:

```
[var "hello", "b", "last element"]
```

Mutable arrays and mutable variables

We could also assign a mutable array to a mutable variable.

```
var numbers : [var Nat] = [var 8, 8, 3, 0];  
  
numbers[2] := 10; // mutate the value inside the array  
  
numbers := [var 1]; // mutate the value of the variable  
  
numbers := [var]; // mutate the value of the variable
```

We declared a mutable variable named `numbers`. We annotated the type of the variable with `[var Nat]` indicating that the value of this variable is a mutable array of `Nat` values. We then assigned a mutable array to the variable name. The array has the keyword `var` inside of it.

In the second line we access the third element by indexing and mutate the `Nat` value at that index.

In the third line, we mutate the value of the variable, which is a whole new mutable array with one single value.

In the last line, we mutate the value of the variable again, which is a whole new mutable array with zero values.

We could mutate the variable again, but the new value has to be of type `[var Nat]`

Operators

Operators are symbols that indicate several kinds of *operations* on values. They consist of one, two or three characters and are used to perform manipulations on typically one or two values at a time.

```
let x = 1 + 1;
```

The `+` character in the example above serves as an addition operator, which performs an *arithmetic operation*.

In this chapter we will cover several kinds of operators, namely *numeric*, *relational*, *bitwise*, and *assignment* operators. We will also cover *text concatenation*, *logical expressions* and *operator precedence*.

Numeric operators

Numeric operators are used to perform *arithmetic* operations on number types like `Nat`, `Int` or `Float`. Here's a list of all numeric operators:

- `+` addition
- `-` subtraction
- `*` multiplication
- `/` division
- `%` modulo
- `**` exponentiation

An example:

```
let a : Nat = (2 ** 4) / 4;
```

We used parentheses `(2 ** 4)` to indicate the order in which the operations need to be performed. The exponentiation happens first and the result is then divided by 4. The end result will be a value of type `Nat`.

The order in which the operations are performed is called *operator precedence*.

Relational operators

Relational operators are used to *relate* or compare two values. The result of the comparison is a value of type `Bool`.

- `==` is equal to
- `!=` is not equal to
- `<=` is less than or equal to
- `>=` is greater than or equal to
- `<` is less than (**must be enclosed in whitespace**)
- `>` is greater than (**must be enclosed in whitespace**)

Some examples:

```
let b : Bool = 2 > 3;
```

```
let c : Bool = (2 : Int) >= 2;
```

```
let d : Bool = 1.61 == 1.61;
```

In the first line we compared two `Nat` values. The result is the value `false` of type `Bool`.

Notice how we type annotated the *value* itself in the second line, therefore telling Motoko that we are now comparing two `Int` values. The result is the value `true` of type `Bool`.

In the last line we compared two `Float` values. The result is the value `true` of type `Bool`.

Assignment operators

We already encountered the most common assignment operator in [mutability](#), which is the `:=` operator. There are [many](#) assignment operators in Motoko. Lets just focus on some essential ones here:

- `:=` assignment (in place update)
- `+=` in place add
- `-=` in place subtract
- `*=` in place multiply
- `/=` in place divide
- `%=` in place modulo
- `**=` in place exponentiation

Lets use all of them in an example:

```
var number : Int = 5;
```

```
number += 2;
```

```
number
```

```
var number : Int = 5;
```

```
number -= 10;
```

```
number
```

```
var number : Int = 5;
```

```
number *= 2;
```

```
number
```

```
var number : Int = 6;
```

```
number /= 2;
```

```
number
```

```
var number : Int = 5;
```

```
number %= 5;
```

```
number
```

```
var number : Int = 5;
```

```
number **= 2;
```

```
number
```

We started by declaring a mutable variable named `number`, we annotated its name with the type `Int` and set its value equal to `5`. Then we *mutate* the variable multiple times using *assignment operators*.

Text concatenation

We can use the `#` sign to *concatenate* values of type `Text`.

```
let t1 : Text = "Motoko";  
let t2 : Text = "Programming";  
let result : Text = t1 # " " # t2;
```

In the last line we concatenate the two `Text` variables `t1` and `t2` with a *text literal* `" "` representing a space.

Logical expressions

Logical expressions are used to express common *logical operations* on values of type `Bool`. There are three types of logical expressions in Motoko.

not expression

The `not` expression takes only one *operand* of type `Bool` and *negates* the value:

```
let negate : Bool = not false;
let yes : Bool = not (1 > 2);
```

In the first line `negate` has boolean value `true`. It is set by negating the boolean *literal* `false`. In the second line, we negate the *boolean expression* `(1 > 2)`.

Both `negate` and `yes` are of type `Bool`. This type is *inferred*.

The *truth table* for `not` is

x	not x
<code>true</code>	<code>false</code>
<code>false</code>	<code>true</code>

and expression

The `and` expression takes two *operands* of type `Bool` and performs a *logical AND* operation.

```
let result : Bool = true and false;
```

`result` is now `false` according to the *truth table*.

x	y	x and y
<code>true</code>	<code>true</code>	<code>true</code>
<code>true</code>	<code>false</code>	<code>false</code>
<code>false</code>	<code>true</code>	<code>false</code>

x	y	x and y
false	false	false

or expression

The `or` expression takes two *operands* of type `Bool` and performs a *logical OR* operation.

```
let result : Bool = true or false;
```

`result` is now `true` according to the *truth table*.

x	y	x or y
true	true	true
true	false	true
false	true	true
false	false	false

Bitwise operators

Bitwise operators are used to manipulate the *bits* of number values.

Bitwise AND &

```
let a : Nat8 = 10; // binary: 0000_1010
let b : Nat8 = 6; // binary: 0000_0110
let c : Nat8 = a & b; // binary: 0010 (decimal: 2)
```

Bitwise OR |

```
let a : Nat8 = 10; // binary: 0000_1010
let b : Nat8 = 6; // binary: 0000_0110
let c : Nat8 = a | b; // binary: 0000_1110 (decimal: 14)
```

Bitwise XOR ^

```
let a : Nat8 = 10; // binary: 0000_1010
let b : Nat8 = 6; // binary: 0000_0110
let c : Nat8 = a ^ b; // binary: 0000_1100 (decimal: 12)
```

Bitwise Shift Left <<

```
let a : Nat8 = 10; // binary: 0000_1010
let b : Nat8 = a << 2; // binary: 0010_1000 (decimal: 40)
```

Bitwise Shift Right >>

```
let a : Nat8 = 10; // binary: 0000_1010
let b : Nat8 = a >> 2; // binary: 0000_0010 (decimal: 2)
```

Bitwise Rotate Left <>>

```
let a : Nat8 = 10; // binary: 0000_1010
let b : Nat8 = a <>> 2; // binary: 0010_1000 (decimal: 40)
```

Bitwise Rotate Right <>>

```
let a : Nat8 = 10; // binary: 0000_1010
let b : Nat8 = a <>> 2; // binary: 1000_0010 (decimal: 130)
```

Wrapping Addition +%

```
let a : Int8 = 127; // maximum value for a 8-bit signed integer
let b : Int8 = 1;
let c : Int8 = a +% b; // wraps around to -128
```

Wrapping Subtraction -%

```
let a : Int8 = -128; // minimum value for a 8-bit signed integer
let b : Int8 = 1;
let c : Int8 = a -% b; // wraps around to 127
```

Wrapping Multiplication *%

```
let a : Nat8 = 2;
let b : Nat8 = 128;
let c : Nat8 = a *% b; // wraps around to 0
```

Wrapping Exponentiation **%

```
let a : Nat8 = 2;
let b : Nat8 = 8;
let c : Nat8 = a **% b; // wraps around to 0
```

Operator precedence

Consider the following example:

```
let q : Float = 1 + 2 - 3 * 4 / 5;
```

We used 4 common arithmetic operations in one line. The result is a value of type `Float` and will be named `q`.

If we would perform the operations from left to right, we would do the following:

- We add 1 to 2 giving us 3
- We then subtract 3 from 3 giving us 0
- We multiply 0 with 4 giving us 0
- We divide 0 by 5 giving us 0

But if we run this code in Motoko, the value of `q` will be `0.6`! The reason for this is that the operations are **not** performed from left to right. Multiplication and division are performed first, followed by addition and subtraction.

Multiplication and division have a *higher precedence* than addition and subtraction. Every operator has a certain 'priority' compared to other operators. To see all the precedence rules of which operators are applied before others, checkout the [official docs](#).

To ensure an operation happens first before any other, we can enclose the values and the operator inside parenthesis.

```
let q : Float = ((1 + 2) - 3) * 4 / 5;
```

The value of `q` is now `0`.

Pattern Matching

When we discussed [tuples](#), [records](#) and [variants](#), we were building up *structured data types*. With pattern matching we are able to *decompose* structured data into its constituent parts.

Lets use pattern matching to decompose a tuple into its constituent parts. We first construct a tuple and assign it to a variable. Then we deconstruct it by naming its values one by one and bringing them into scope:

```
let individual = ("Male", 30);

let (gender, age) = individual;
```

The tuple `("Male", 30)` has a `"Male"` value and a `30` value. The second line *binds* these values to *newly created* variable names `gender` and `age`. We can now use these variables in this scope.

We could also type annotate the variable names to *check* the type of the values we are decomposing.

```
let (gender : Text, _ : Nat) = individual;
```

In this last example, we only brought `gender` into scope. Using the *wildcard* `_` indicates that we don't care about the second value which should have type `Nat`.

When we decompose `individual` into `gender` and `age` we say that the variable is *consumed* by the pattern `(gender, age)`.

Lets look at an example of pattern matching on a record:

```
let person = {
    name = "Peter";
    member = false;
};

let { name; member } = person;
```

In the example above, we define a record. We then decompose the record fields by using the names of the fields, thereby bringing these variables into scope.

Note we use the `{name; member}` pattern to consume the variable.

We don't have to use all the fields. We could use only one for example:

```
let { name } = person;
```

We could also rename the fields:

```
let { name = realName; member = groupMember } = person;
```

Type annotation within the patterns is allowed and recommended when its not obvious what the type should be.

Pattern matching is a powerful feature of Motoko with many more options and we will revisit it later in this book. For more info check the [official docs](#).

Functions

We begin our treatment of functions in this chapter by discussing a subset of possible functions in Motoko, namely *private* functions. We will explain private function arguments, argument type annotation, return type and the type of the function itself.

Private functions in Motoko may appear in [Records](#), [Objects](#), [Classes](#), [Modules](#), [Actors](#) and other places as well.

Other possible functions will be discussed in upcoming chapters:

- [Objects and Classes](#) public and private functions inside an object or class
- [Modules](#) public and private functions inside a module
- [Actors](#) `async` update, query and oneway shared functions
- [Actors](#) `async*` functions
- [Principals and Authentication](#) caller-identifying functions
- [Upgrades](#) system upgrade functions

Private functions

Lets start with most simple function in Motoko:

```
func myFunc() {};
```

The `func` keyword is indicating a *function declaration*. `myFunc` is an arbitrary name of the function followed by two parenthesis `()` and two curly brackets `{}`. The `()` are used for function *arguments* (inputs to the function) and the `{}` are used for the function *body*.

Note: The `()` in this context is not the same as the [unit type](#)!

We could *explicitly* tell Motoko that this is a private function by using the `private` keyword in front of the `func` keyword. This is not necessary though, because a function declaration defaults to a private function in Motoko unless declared otherwise.

Lets be more explicit about our private function, add one argument as an input and expand the body:

```
private func myFunc(x : Nat) {
    // function body
};
```

The function is now marked private. All arguments **must** be annotated. Type inference doesn't work here. In this case we take in one argument and name it `x`. We also type annotate it with `Nat`.

Lets proceed by adding a return type to this function and actually returning a value of that type:

```
private func myFunc(x : Nat) : Nat {
    return x;
};
```

After the function arguments we annotate the *return type* of this function with `: Nat`. If we don't annotate the return type of a private function, it defaults to the unit `()` return type.

Inside the body we return `x`, the same variable that was passed to the function. This is allowed because `x` also has type `Nat`, which is the expected return type for this function.

Lets simplify this function:

```
func myFunc(x : Nat) : Nat {
    x;
};
```

We removed the private keyword and simplified the return expression to just `x`. Even the semicolon `;` is gone. But note that we can't leave out the type annotations, like we did with variables. Type inference doesn't work on function declarations except for the defaulting unit type behavior mentioned above.

Lets write a useful private function and *call* it:

```
func concat(t1 : Text, t2 : Text) : Text {
    let result = t1 # t2;
    result;
};

let output = concat("Hello", "world");
```

Our function `concat` takes two arguments of type `Text`. It also returns a `Text` type.

We use the text concatenation *operator* `#` to *concatenate* the two arguments and assign the result to a new variable. Concatenation with `#` only works for `Text` types.

The result of the concatenation `t1 # t2` is another `Text`. We did not type annotate the variable `result`. Motoko automatically infers this for us.

We return `result` by placing it at the last line of the function without a `return` keyword and semicolon `;`. You could be explicit by adding the `return` keyword and even type annotate the `result` variable with a `: Text` type, but in this case it is not necessary.

Lastly, we *call* this function with two text *literals* as the arguments and assign its result to the variable `output`. Again, we don't have to annotate the type of this `output` variable, because this is obvious from the context and Motoko will infer this information for us.

Function type

The last concept for this chapter is the type of the *whole* function. A function's *typed arguments* and *return type* together are used to define a type for the function as a whole. The type of the function `concat` above is the following:

```
type Concat = (Text, Text) -> Text;  
let ourFunc : Concat = concat;
```

We used the type name `Concat` to define a new type `(Text, Text) -> Text`. This is the type of our function `concat`. The function type is constructed by joining three things:

- a *tuple* of types for the function argument types
- the `->` keyword
- the *return type* of the function

We use the `Concat` type to annotate the type of another variable `ourFunc` and assign the function name `concat` to it without the parenthesis and arguments like we did when we called the function. We basically renamed our function to `ourFunc`.

Options and Results

In this chapter we discuss the *Option* type and the *Result* variant type.

They are different concepts but they are generally used for the same purpose, namely specifying the return type of functions. Options are used more widely beyond function return types, whereas Results mostly appear as function return type.

Options

An option type is a type that can either have some value or have no value at all.

An *option type* is a type preceded by a question mark `?T`.

An *option value* is a value preceded by a question mark `?T`.

Some examples:

```
let a : ?Nat = ?202;
let b : ?Text = ?"DeFi";
let c : ?Bool = ?true;
```

Variable `a` is annotated with option type `?Nat` and assigned the option value `?202`. The option value must be of the same type as the option type, meaning the value `202` is of type `Nat`.

Every *option value* can have the value `null`, which is a special 'value' indicating the absence of a value. The `null` value has type `Null`. For example:

```
let x : ?Nat = null;
let y : ?Text = null : Null;
```

We assign the value `null` to variable `x` which has option type `?Nat`. In the second line we type annotated the `null` value with its `Null` type.

Lets use an option type as the return type of two functions:

```
func returnOption() : ?Nat {
    ?0;
};

func returnNull() : ?Nat {
    null;
};
```

The two functions both have `?Nat` as their return type. The first returns an option value and the second returns `null`.

In the next chapter about [control flow](#) we will demonstrate how we can use option types in a useful way.

Results

To fully understand Results we have to understand [generics](#) first, which are an advanced topic that we will cover later in this book. For now we will only cover a limited form of the Result variant type to give a basic idea of the concept.

A Result type is a [variant type](#). A simple definition could be:

```
type Result = {
    #ok;
    #err;
};
```

A value of this type may be one of two possible variants, namely `#ok` or `#err`. These variants are used to indicate either the *successful result* of a function or a *possible error* during the evaluation of a function.

Lets use the Result type in the same way we used Options above:

```
func returnOk() : Result {
    #ok;
};

func returnErr() : Result {
    #err;
};
```

Both functions have `Result` as their return type. The one returns the `#ok` variant and the other return the `#err` variant.

In the next chapter about [control flow](#) we will demonstrate how we can use the Result variant type in a useful way.

Control Flow

Control flow refers to the order in which a program is *executed*. We discuss three common *control flow* constructs in Motoko: `if` expressions, `if else` expressions and `switch` expressions.

These constructs are called *expressions* because they *evaluate* to a value of a certain type.

If Expression

An *if expression* is constructed with the `if` keyword followed by two expressions. The first expression is enclosed in parenthesis `()` and the second is enclosed with curly braces `{}`. They both evaluate to a value of a certain type.

The first expression after the `if` keyword has to evaluate to a value of type `Bool`. Based on the boolean value of the first expression, the `if` expression will either evaluate the second expression or it doesn't.

```
let condition = true;  
  
var number = 0;  
  
if (condition) { number += 1 };  
  
// number is now 1
```

The first expression evaluates to `true` so the second expression is evaluated and the code inside it is executed. Note we used the `+=` assignment **operator** to increment the mutable variable `number`.

If the first expression evaluates to `false`, then the second expression is **not** evaluated and the *whole if expression* will evaluate to the unit type `()` and the program continues.

If Else Expression

The `if else` expression starts with the `if` keyword followed by two sub-expressions (a condition and its associated branch) and ends with the `else` keyword and a third sub-expression:

```
if (condition) 1 else 2;
```

The condition has to be of type `Bool`. When the condition evaluates to the value `true`, the second sub-expression `1` is returned. When the condition evaluates to the value `false`, the third sub-expression `2` is returned.

When the branches are more complex expressions, they require curly braces:

```
if (condition) {} else {};
```

Unlike `if` expressions that lack an `else`, when the first sub-expression of an `if else` evaluates to `false`, the entire `if else` expression evaluates as the third sub-expression, not the unit value `()`.

For example, this `if else` expression evaluates to a value of a certain type `Text`, and we assign that value to a variable named `result`:

```
let result : Text = if (condition) {
    "condition was true";
} else {
    "condition was false";
};
```

Generally, the second and third sub-expressions of the `if else` expression must evaluate to a value of the same type.

Switch Expression

The `switch` expression is a powerful control flow construct that allows [pattern matching](#) on its input.

It is constructed with the `switch` keyword followed by an input expression enclosed in parenthesis `()` and a code block enclosed in curly braces `{}`. Inside this code block we encounter the `case` keyword once or several times depending on the input.

```
switch (condition) {
    case (a) {};
};
```

The `case` keyword is followed by a [pattern](#) and an expression in curly braces `{}`. Pattern matching is performed on the input and the possible values of the input are *bound* to the names in the pattern. If the pattern matches, then the expression in the curly braces evaluates.

Lets switch on a variant as an input:

```
type Color = { #Black; #White; #Blue };

let color : Color = #Black;

var count = 0;

switch (color) {
    case (#Black) { count += 1 };
    case (#White) { count -= 1 };
    case (#Blue) { count := 0 };
};
```

We defined a variant type `Color`, declared a variable `color` with that type and declared another mutable variable `count` and set it to zero. We then used our variable `color` as the input to our `switch` expression.

After every `case` keyword, we check for a possible value of our variant. When we have a match, we execute the code in the expression defined for that case.

In the example above, the color is `#Black` so the `count` variable will be incremented by one. The other cases will be skipped.

If all expressions after every `case (pattern)` evaluate to a value of the same type, then like in the example of the `if else` expression, we could assign the return value of the *whole* `switch` expression to a variable or use it anywhere else an expression is expected.

In the example above, our `switch` expression evaluates to `()`.

A little program

Lets combine some concepts we have learned so far. We will use a `Result`, an `Option`, a mutable variable, a `function` and a `switch` expression together:

```
type Result = {
    #ok : Nat;
    #err : Text;
};

type Balance = ?Nat;

var balance : Balance = null;

func getBalance(bal : ?Nat) : Result {
    switch (bal) {
        case (null) {
            #err "No balance!";
        };
        case (?amount) {
            #ok amount;
        };
    };
};
```

We started by defining a `Result` type with two variants `#ok` and `#err`. Each variant has an associated type namely `Nat` and `Text`.

Then we define an `Option` type called `Balance`. It is an optional value of type `?Nat`.

We then declare a mutable variable called `balance` and annotate it with the `Balance` type.

And lastly, we define a function that takes one argument of type `?Nat` and returns a value of type `Result`. The function uses a `switch` expression to check the value of our variable `balance`.

The `switch` expression checks two cases:

- In the case the value of `balance` is `null`, it returns the `#err` variant with an associated text. This is returned to the function body, which is then treated as the return value of the function.
- In the case the value of `balance` is some optional value `?amount`, it returns the `#ok` variant with an associated value `amount`.

In both cases we used *pattern matching* to check the values. In the last case we defined a new name `amount` to *bind* our value, in case we found some optional value of type `?Nat`. If so, then that value is now available through this new name.

Lets now call this function.

```
let amount : Result = getBalance(balance);  
  
balance := ?10;  
  
let amount2 : Result = getBalance(balance);
```

The first call will yield `#err "No balance!"` telling us that the balance is `null`. We then mutate the value of our balance to a new optional value `?10`. When we call the function again, we get `#ok 10`.

Note, we didn't have to annotate the `amount` variable with the `Result` type, but it does make it more clear to the reader what the expected type is of the function.

Objects and Classes

When we looked at [records](#) we saw that we could package named variables and their values inside curly braces, like `{ x = 0 }`. The type of such a record was of the form `{ x : Nat }`.

In this chapter, we will look at *objects*, which are like an advanced version of records. We will also look at *classes*, which are like 'factories' or 'constructors' for manufacturing objects.

Objects

Objects, like [records](#), are like a collection of named (possibly mutable) variables packaged in curly braces `{}`. But there are four key differences:

- We define an object by using the `object` keyword.
- We specify the *visibility* of the named variables with either the `public` or `private` keyword.
- We specify the *mutability* of the named variables with either the `let` or `var` keyword.
- Only `public` variables are part of the type of an object.

Here's a simple example:

```
let obj = object {
    private let x = 0;
};
```

We declared an object by using the `object` keyword before the curly braces. We also declared the variable `x` with the `let` keyword.

The type of this object is the empty object type `{}`. This is because the `x` variable was declared `private`.

A typed object with a public field could look like this:

```
type Obj = { y : Nat };

let obj : Obj = object {
    private let x = 0;
    public let y = 0;
};
```

We defined the type beforehand, which consists only of one named variable `y` of type `Nat`. Then we declared an object with a `public` variable `y`. We used the object type to annotate the name of the variable `obj`.

Notice that `x` is not part of the type, therefore it is not accessible from outside the object.

The values of the variables inside objects (and inside records as well) could also be a *function*. As we saw in [functions](#), functions also have a type and they could be assigned to a named variable.

```
let obj = object {
    private func f() {};
    private let x = f;
};
```

Inside the object, we first define a **private function** and then assign that function to a **private** immutable variable **x**. The type of this object is the empty object type **{ }** because there are no public variables.

Lets change the visibility of our **x** variable:

```
let obj = object {
    private func f() {};
    public let x = f;
};
```

This object now has the following type:

```
{ x : () -> () }
```

This is the type of an object with one *field* named **x**, which is of *function type* **() -> ()**. We could access this public field like this:

```
let result = obj.x();
let f = obj.x;
```

The first line actually calls the function and assigns the result to **result**. The second line only *references* the function and renames it.

Public functions in objects

Lets look at a useful object:

```
let balance = object {
    private let initialBalance = 100;

    public var balance = initialBalance;

    public func addAmount( amount : Nat ) : Nat {
        balance += amount;
        balance
    };
};
```

The first field is a private immutable variable named `initialBalance` with constant value `100`. The second field is a public mutable variable named `balance` initiated with the value of `initialBalance`.

Then we encounter a declaration of a public function named `addAmount`, which takes one argument `amount` of type `Nat` and returns a value of type `Nat`. The function adds the value of `amount` to `balance` using the 'in place add' `assignment operator` and finally returns the new value of `balance`.

This object has the following type:

```
{
    addAmount : Nat -> Nat;
    var balance : Nat;
}
```

This object type has two fields. The `addAmount` field has function type `Nat -> Nat`. And the second field is a `var` mutable variable of type `Nat`.

Classes

Classes in essence are nothing more than a *function* with a fancy name and special notation. A class, like any other function, takes in values of a certain type and returns a value of a certain type. The return type of a class is always an **object**. Classes are like 'factories' or 'templates' for objects.

Consider the following two type declarations:

```
type SomeObject = {};
type SomeClass = () -> SomeObject;
```

The first is the type of an empty object. The second is the type of a class. It's a function type that could take in a number of arguments and return an *instance* of an object of type **SomeObject**.

But classes have a special notation using the **class** keyword. We declare a class like this:

```
class MyClass() {
    private let a = 0;
    public let b = 1;
};
```

To use this class we have to create an instance of this class:

```
let myClassInstance = MyClass();
```

When we *instantiate* our class by calling **MyClass()** it returns an object. That object is now named **myClassInstance**.

In fact, we could set the expected type of the returned object by defining the type and annotating our variable with it:

```
type ObjectType = {
    b : Nat;
};

let anotherClassInstance : ObjectType = MyClass();
```

Now this class is not very useful and we could just have declared an object instead of declaring a class and instantiating it.

Let's declare a class that takes some arguments, instantiate two objects with it and use those objects:

```
class CryptoAccount(amount : Nat, multiplier : Nat) {
    private func calc(a : Nat, b : Nat) : Nat {
        a * b;
    };

    public var balance = calc(amount, multiplier);
};

let account1 = CryptoAccount(10, 5);
let account2 = CryptoAccount(10, 10);

account1.balance += 50;
account2.balance += 20;
```

Lets analyze the code line by line. Our class `CryptoAccount` takes in two values of type `Nat`. These are used once during initialization.

The first member of our class is a private function named `calc`. It just takes two values of type `Nat` and returns their product. The second member of the class is a mutable variable named `balance` which is declared by calling `calc` with the two values coming from our class.

Because this class only has one `public` field, the expected type of the object that is returned should be `{ var balance : Nat }`.

We then define two objects `account1` and `account2` by calling our class with different values. Both objects have a mutable public field named `balance` and we can use that to mutate the value of each.

Public functions in classes

The real power of classes is that they yield objects with *state* and a *public API* that operates on that state. The state could be any mutable variable, array or any other value that sits inside the object. The public API consists of one or more functions that operate on that state.

Here's an example:

```
class CryptoAccount(amount : Nat, multiplier : Nat) {
    public var balance = amount * multiplier;

    public func pay(amount : Nat) {
        balance += amount;
    };
};

let account = CryptoAccount(10, 5);

account.pay(50);
```

Our `CryptoAccount` class takes the same two arguments as before, but now has only two members. One is the public mutable `balance` variable. The second is a *public function*. Because there are two public fields, the expected type of the object returned from this class is

```
{
    pay : (Nat) -> ();
    balance : Nat;
}
```

After instantiating the `account` variable with our class, we can access the public function by calling it as a *method* of our object. When we write `account.pay(50)` we call that function, which in turn mutates the internal state of the object.

In this example the public function happens to operate on a public variable. It could also have mutated a private mutable variable of any type.

Modules and Imports

Modules, like [objects](#), are a collection of named variables, types, functions (and possibly more items) that together serve a special purpose. Modules help us *organize* our code. A module is usually defined in its own separate source file. It is meant to be *imported* in a Motoko program from its source file.

Modules are like a *limited* version of objects. We will discuss their [limitations](#) below.

Imports

Lets define a simple module in its own source file `module.mo`:

```
module {
    private let x = 0;
    public let name = "Peter";
};
```

This module has two fields, one private, one public. Like in the case of objects, only the public fields are 'visible' from the outside.

Lets now *import* this module from its source file and use its public field:

```
// main.mo
import MyModule "module";

let person : Text = MyModule.name;
```

We use the `import` keyword to import the module into our program. Lines starting with `import` are only allowed at the top of a Motoko file, before anything else.

We *declared* a new name `MyModule` for our module. And we referenced the module source file `module.mo` by including it in double quotes without the `.mo` extension.

We then used its public field `name` by referencing it through our chosen module name `MyModule`. In this case we assigned this `name` text value to a newly declared variable `person`.

Nested modules

Modules can contain other modules. Lets write a module with two *child* modules.

```
module {
    public module Person {
        public let name = "Peter";
        public let age = 20;
    };

    private module Info {
        public let city = "Amsterdam";
    };

    public let place = Info.city;
};
```

The top-level module has two *named* child modules `Person` and `Info`. The one is public, the other is private.

The public contents of the `Info` module are only visible to the top-level module. In this case the public `place` variable is assigned the value of the public field `city` inside the private `Info` child module.

Only the sub-module `Person` and variable `place` are accessible when imported.

```
// main.mo
import Mod "module-nested";

let personName = Mod.Person.name;

let city = Mod.place;
```

Public functions in modules

A module exposes a *public API* by defining public functions inside the module. In fact, this is what modules are mostly used for. A module with a very simple API could be:

```
module {
    private let MAX_SIZE = 10;

    public func checkSize(size : Nat) : Bool {
        size <= MAX_SIZE;
    };
}
```

This module has a private constant `MAX_SIZE` only available to the module itself. It's a convention to use capital letters for constants.

It also has a public function `checkSize` that provides some functionality. It takes an argument and computes an inequality using the private constant and the argument.

We would use this module like this:

```
// main.mo
import SizeChecker "module-public-functions";

let x = 5;

if (SizeChecker.checkSize(x)) {
    // do something
};
```

We used our newly chosen module name `SizeChecker` to reference the public function inside the module and call it with an argument `x` from the main file.

The expression `SizeChecker.checkSize(x)` evaluates to a `Bool` value and thus can be used as an argument in the if expression.

Public types in modules

Modules can also define private and public *types*. Private types are meant for internal use only, like private variables. Public types in a module are meant to be imported and used elsewhere.

Types are declared using the `type` keyword and their *visibility* is specified:

```
module {
    private type UserData = {
        name : Text;
        age : Nat;
    };

    public type User = (Nat, UserData);
};
```

Only the `User` type is visible outside the module. Type `UserData` can only be referenced inside a module and even used in a public type (or variable or function) declaration as shown above.

Type imports and renaming

Types are often given a local *type alias* (renamed) after importing them:

```
// main.mo
import User "types";

type User = User.User;
```

In the example above, we first import the module from file `types.mo` and give it the *module name* `User`.

Then we define a new *type alias* also named `User`, again using the `type` keyword. We reference the imported type by module name and type name: `User.User`.

We often use the same name for the module, the type alias and the imported type!

Public classes in modules

Modules can also define private and public `classes`. Private classes are rarely used internally. Public classes on the other hand are used widely. In fact, most modules define one *main class* named after the module itself. Public classes in a module are meant to be imported and used elsewhere as 'factories' for objects.

Classes in modules are declared using the `class` keyword and their *visibility* is specified:

```
module {
    public class MyClass(x : Nat) {
        private let start = x ** 2;
        private var counter = 0;

        public func addOne() {
            counter += 1;
        };

        public func getCurrent() : Nat {
            counter;
        };
    };
}
```

The class `MyClass` is visible outside the module. It can only be *instantiated* if it is imported somewhere else, due to [static limitations](#) of modules. Our class takes in one initial argument `x : Nat` and defines two internal private variables. It also exposes two public functions.

Class imports and class referencing

Classes in modules are imported from the *module source file* they are defined in. This file often has the same name as the class! Further more, the module alias locally is also given the same name:

```
// main.mo
import MyClass "MyClass";

let myClass = MyClass.MyClass(0);
```

In the example above, we first import the module from source file `MyClass.mo` and give it the *module name* `MyClass`.

Then we instantiate the class by referring to it starting with the module name and then the class `MyClass.MyClass(0)`. The `0` is the initial argument that our class takes in.

We often use the same name for the module file name, the module local name and the class!

Static expressions only

Modules are limited to *static* expressions only. This means that no computations can take place inside the module. Static means that no program is running. A module only defines code to be used elsewhere for computations.

A function call is non-static. Computations, like adding and multiplying are also non-static. Therefore the following code is not allowed inside modules:

```
module {
    // public let x = 1 + 1;

    // public func compute() {
    //     8 - 5
    // };

    public func compute(x : Nat, y : Nat) : Nat {
        x * y;
    };
}
```

The first line in the module tries to compute `1 + 1` which is a 'dynamic' operation. The second line tries to define a function which makes an internal computation, another non-static operation.

The last function `compute` is allowed because it only defines *how* to perform a computation, but does not actually perform the computation. Instead it is meant to be imported somewhere, like in an `actor`, to perform the computation on any values passed to it.

Module type

Module types are not used often in practice, but they do exist. Modules have types that look almost the same as object types. A type of a module looks like this:

```
type MyModule = module {
    x : Nat;
    f : () -> ();
};
```

This type describes a module with two public fields, one being a `Nat` and the other being a function of type `() -> ()`.

Assertions

Sometimes it is convenient to make sure some condition is *true* in your program before continuing execution. For that we can use an *assertion*.

An assertion is made using the `assert` keyword. It always acts on an expression of type `Bool`.

If the expression is `true` then the whole assertion evaluates to the unit type `()` and the program continues execution as normal. If the expression evaluates to `false` then the program *traps*.

```
let condition : Bool = 1 > 2;  
  
assert condition; // Program traps!
```

Because our `condition` is `false`, this program will trap at the assertion. The exact consequences of a trap inside a running canister will be explained [later in this book](#).

Internet Computer Programming Concepts

This chapter covers Internet Computer specific features of Motoko.

It is *recommended* to be familiar with the following items (and their types) from the last chapter:

- [functions](#)
- [records](#)
- [objects](#)
- [classes](#)
- [modules](#)

A comparison of these concepts with actors and actor classes is available:

Motoko Items Comparison Table

Before diving in, let's briefly describe four terms that look alike, but are not the same!

- **Actor** An abstract notion of a self-contained software execution unit that communicates with other actors through asynchronous message passing.
- **Canister** An instance of an actor that runs on the Internet Computer Blockchain.
- **WebAssembly Module** A file containing a binary representation of a program (that may have been written in Motoko) that is installed in a canister that runs on the Internet Computer.
- **Smart Contract** The traditional notion of a blockchain program with roughly the same security guarantees as canisters but with limited memory and computation power.

Actors

Actors, like [objects](#), are like a package of *state* and a *public API* that accesses and modifies that state. They are declared using the [actor](#) keyword and have an [actor type](#).

Top level actors

Today (Jan 2023), an actor in Motoko is defined as a top-level item in its own Motoko source file. Optionally, it may be preceded by one or more imports:

```
// actor.mo
import Mod "mod";

actor {
};

}
```

We declared an empty actor in its own source file `actor.mo`. It is preceded by an [import of a module](#) defined in `mod.mo` and named `Mod` for use inside the actor.

You may feel disappointed by the simplicity of this example, but this setup (one source file containing **only imports** and **one actor**) is the core of every Motoko program!

Actors vs. objects

Unlike [objects](#), actors may **only** have [private](#) (immutable or mutable) variables. We can only communicate with an actor by calling its [public shared functions](#) and never by directly modifying its [private](#) variables. In this way, the [memory state](#) of an actor is isolated from other actors. Only public shared functions (and some system functions) can change the memory state of an actor.

To understand actors it is useful to compare them with objects:

Public API

- [Public functions in actors](#) are accessible from outside the Internet Computer.
- [Public functions in objects](#) are only accessible from within your Motoko code.

Private and public variables

- Actors don't allow public ([immutable](#) or [mutable](#)) variables
- Objects do allow public ([immutable](#) or [mutable](#)) variables

Private functions

- [Private functions](#) in actors are not part of the [actor type](#)
- [Private functions](#) in [objects](#) are not part of the [object type](#)

Public shared functions

- Actors only allow [shared](#) public functions
- Objects only allow non-shared public functions

Class and Actor Class

- Actors have 'factory' functions called [Actor Classes](#)
- Objects have 'factory' functions called [Classes](#)

For a full comparison checkout: [Motoko Items Comparison Table](#)

Public Shared Functions in Actors

Actors allow *three kinds* of public functions:

1. Public [shared query](#) functions:
Can only *read* state
2. Public [shared update](#) functions:
Can *read* and *write* state
3. Public [shared oneway](#) functions:
Can *read* and *write*, but don't have any *return value*.

NOTE

Public shared query functions are fast, but don't have the full security guarantees of the

Internet Computer because they do not 'go through' consensus

Shared async types

The argument and return types of public shared functions are restricted to *shared types* only. We will cover shared types [later](#) in this book.

Query and *update* functions always have the special `async` return type.

Oneway functions always immediately return `()` regardless of whether they execute successfully.

These functions are only allowed inside *actors*. Here are their *function signatures* and *function types*.

Public shared query

```
public shared query func read() : async () { () };  
// Has type `shared query () -> async ()`
```

The function named `read` is declared with `public shared query` and returns `async ()`. This function is not allowed to modify the state of the actor because of the `query` keyword. It can only read state and return most types. The `shared query` and `async` keywords are part of the function type. Query functions are fast.

Public shared update

```
public shared func write() : async () { () };  
// Has type `shared Text -> async ()`
```

The function named `write` is declared with `public shared` and also returns `async ()`. This function is allowed to modify the state of the actor. There is no other special keyword (like `query`) to indicate that this is an update function. The `shared` and `async` keywords are part of the function type. Update functions take 2 seconds per call.

Public shared oneway

```
public shared func oneway() { () };

// Has type `shared () -> ()`
```

The function named `oneway` is also declared with `public shared` but does not have a return type. This function is allowed to modify the state of the actor. Because it has no return type, it is assumed to be a oneway function which always returns `()` regardless of whether they execute successfully. Only the `shared` keyword is part of their type. Oneway functions also take 2 seconds per call.

In our example none of the functions take any arguments for simplicity.

NOTE

The `shared` keyword may be left out and Motoko will assume the public function in an actor to be `shared` by default. To avoid confusion with public functions elsewhere (like `modules`, `objects` or `classes`) we will keep using the `shared` keyword for public functions in actors

A simple actor

Here's an actor with one *state* variable and some functions that *read* or *write* that variable:

```
actor {

    private var latestComment = "";

    public shared query func readComment() : async Text {
        latestComment;
    };

    public shared func writeComment(comment : Text) : async () {
        latestComment := comment;
    };

    public shared func deleteComment() {
        latestComment := "";
    };
}
```

This actor has one private mutable variable named `latestComment` of type `Text` and is initially set to the empty string `""`. This variable is not visible 'from the outside', but only available internally inside the actor. The actor also demonstrates the three possible public functions in any actor.

Then first function `readComment` is a *query* function that takes no arguments and only reads the state variable `latestComment`. It returns `async Text`.

The second function `writeComment` is an *update* function that takes one argument and modifies the state variable. It could return some value, but in this case it returns `async ()`.

The third function `deleteComment` is a *oneway* function that doesn't take any arguments and also modifies the state. But it can not return any value and always returns `()` regardless of whether it successfully updated the state or not.

Actor type

Only public shared functions are part of the type of an actor.

The type of the actor above is the following:

```
type CommentActor = actor {
    deleteComment : shared () -> ();
    readComment : shared query () -> async Text;
    writeComment : shared Text -> async ();
};
```

We named our actor type `CommentActor`. The type itself starts with the `actor` keyword followed by curly braces `{}` (like objects). Inside we find the three function names as fields of the type definition.

Every field name is a public shared function with its own *function type*. The order doesn't matter, but the Motoko orders them alphabetically.

`readComment` has type `shared query () -> async Text` indicating it is a *shared query* function with no arguments and `async Text` return type.

`writeComment` has type `shared Text -> async ()` indicating it is an *shared update* function that takes one `Text` argument and returns no value `async ()`.

`deleteComment` has type `shared () -> ()` indicating it is a *shared oneway* function that takes no arguments and always returns `()`.

From Actor to Canister

An **actor** is written in Motoko code. It defines *public shared functions* that can be accessed from outside the Internet Computer (IC). A **client**, like a laptop or a mobile phone, can send a request over the internet to call one of the public functions defined in an actor.

Here is the code for *one* actor defined in its own Motoko source file. It contains one public function.

```
// main.mo
actor {
    public shared query func hello() : async Text {
        "Hello world";
    };
}
```

We will *deploy* this actor to the Internet Computer!

Canister

A canister is like a home for an **actor**. It lives on the Internet Computer Blockchain. A canister is meant to 'host' actors and make their *public functions* available to other canisters and the wider Internet beyond the Internet Computer itself.

Each canister can host one actor. The *public interface* of the canister is defined by the *actor type* of the actor it hosts. The public interface is described using an *Interface Definition Language*. Every canister has a *unique id*.

The IC provides *system resources* to the canister, like:

- Connectivity: A canister can receive inbound and make outbound *canister calls*.
- Memory: A canister has *main working memory* and also has access to *stable memory*.
- Computation: The code running in a canister is executed by one processor thread and consumes *cycles*.

Code compiling and Wasm modules

Before an actor can be deployed to the Internet Computer, the Motoko code has to be *compiled* into a special kind of code. Compilation transforms the Motoko code into code that can *run* (be

executed) on the Internet Computer. This code is called Webassembly bytecode. The bytecode is just a file on your computer called a *Wasm module*.

It is this Wasm module that gets *installed* in a canister.

Deployment steps

To go from Motoko code to a running canister on the IC, the following happens:

1. The Motoko code is *compiled* into a *Wasm module*
2. An empty canister is registered on the Internet Computer
3. The Wasm module is uploaded to the canister

The public functions of the actor are now accessible on the Internet from any client!

Motoko Playground

Currently (Jan 2023), there are two main tools to achieve the deployment steps. The more advanced one is the [Canister Development Kit](#) called DFX. For now, we will use a simpler tool called [Motoko Playground](#).

The actor at the beginning of this chapter [is deployed](#) using Motoko Playground. To repeat the deployment for yourself, open the [this link](#) and do the following:

1. Check that there is one `Main.mo` file in the left side of the window
2. Check the Motoko actor code from this chapter
3. Click the blue `Deploy` button
4. In the popup window choose `Install`

The deployment steps will now take place and you can follow the process in the output `Log`.

Calling the actor from Motoko Playground

After the successful deployment of the actor in a new canister, we can now call our public function from the browser. On the right hand side of the window, there is a `Candid UI` with the name of our function `hello` and a button `QUERY`. Clicking the button returns the *return value* of our function to the browser window.

The [next chapter](#) will explain what is actually happening when you interact with a canister from a browser.

Canister Calls from Clients

To call a public function of an actor, a message has to be sent over the internet to the Internet Computer Blockchain. This message contains the [canister id](#), the name of the function to be called and the optional arguments for the function. The whole message is also *signed* by the sender.

There are several *libraries* for composing messages, sign them and send them in a [HTTP request](#) to the Internet Computer. The details of this mechanism and the libraries are outside the scope of this book, but we will mention them briefly to establish a general idea of how canister calls are achieved.

Canister Calls from a browser

The most common way to interact with an actor (hosted in a canister on the Internet Computer) is from a browser. Several [Typescript libraries](#) are available that facilitate the process of sending messages to canisters.

In fact, this is exactly what the [Candid UI interface](#) is doing when you call a function. Since Motoko Playground runs in the browser, it interacts with the Internet Computer by running Typescript code in the browser that uses the [Typescript libraries](#).

Canister Calls from DFX

Another way to send messages is from a client computer that runs some program instead of a browser. In fact any program that can access the Internet and issue [HTTP request](#) can send a message to a canister running on the IC.

One such program is called DFX, which is a [Canister Development Kit](#). We will describe how to send messages to canisters from DFX in later chapters.

Principals and Authentication

Principals are *unique identifiers* for either *canisters* or *clients* (external users). Both a canister or an external user can use a principal to *authenticate* themselves on the Internet Computer.

Principals in Motoko are a *primitive type* called `Principal`. We can work directly with this type or use an alternative textual representation of a principal.

Here's an example of the textual representation of a principal:

```
let principal : Text = "k5b7g-kwhqt-xj6wm-rcqej-uwwwp3-t2cf7-6banv-o3i66-q7dy7-
pvbof-dae";
```

The variable `principal` is of type `Text` and has a textual value of a principal.

Anonymous principal

There is one special principal called the *anonymous* principal. It used to authenticate to the Internet Computer anonymously.

```
let anonymous_principal : Text = "2vxss-fae";
```

We will use this principal [later](#) in this chapter.

The Principal Type

To convert a textual principal into a value of type `Principal` we can use the [Principal module](#).

```
import P "mo:base/Principal";

let principal : Principal = P.fromText("k5b7g-kwhqt-xj6wm-rcqej-uwwwp3-t2cf7-6banv-
o3i66-q7dy7-pvbof-dae");
```

We [import](#) the Principal module from the [Base Library](#) and name it `P`. We then defined a variable named `principal` of type `Principal` and assigned a value using the `.fromText()` method available in the Principal module.

We could now use our `principal` variable wherever a value is expected of type `Principal`.

Caller Authenticating Public Shared Functions

There is a special *message object* that is available to [public shared functions](#). Today (Jan 2023) it is only used to *authenticate* the caller of a function. In the future it may have other uses as well.

This message [object](#) has the following type:

```
type MessageObject = {
    caller : Principal;
};
```

We chose the name `MessageObject` arbitrarily, but the type `{ caller : Principal }` is a special *object type* available to public shared functions inside actors.

To use this object, we must [pattern match](#) on it in the function signature:

```
public shared(messageObject) func whoami() : async Principal {
    let { caller } = messageObject;
    caller;
};
```

Our public shared update function now specifies a *variable name* `messageObject` (enclosed in parenthesis `()`) in its signature after the `shared` keyword. We now named the special message object `messageObject` by pattern matching.

In the function body we pattern match again on the `caller` field of the object, thereby extracting the field name `caller` and making it available in the function body. The variable `caller` is of type `Principal` and is treated as the return value for the function.

The function still has the same function type regardless of the message object in the function signature:

```
type WhoAmI = shared () -> async Principal;
```

We did not have to pattern match inside the function body. A simple way to access the `caller` field of the message object is like this:

```
public shared query (msg) func call() : async Principal {
    msg.caller;
};
```

This time we used a public shared query function that returns the principal obtained from the message object.

Checking the Identity of the Caller

If an actor specifies public shared functions and is deployed, then anyone can call its publicly available functions. It is useful to know whether a caller is *anonymous* or *authenticated*.

Here's an actor with a public shared query function which checks whether its caller is anonymous:

```
import P "mo:base/Principal";

actor {
    public shared query ({ caller = id }) func isAnonymous() : async Bool {
        let anon = P.fromText("2vxsx-fae");

        if (id == anon) true else false;
    };
};
```

We now used pattern matching in the function signature to *rename* the `caller` field of the message object to `id`. We also declared a variable `anon` of type `Principal` and set it to the anonymous principal.

The function checks whether the *calling principal* `id` is equal to the *anonymous principal* `anon`.

Later in this book we will learn about *message inspection* where we can inspect all incoming messages before calling a function.

Async Data

Actors expose **public shared functions** to the outside world and other canisters. They define the *interface* for interacting with all Motoko programs running in **canisters** on the Internet Computer. Canisters can interact with other canisters or other **clients** on the internet. This interaction always happens by *asynchronous* function calls.

This chapter explains what kind of data is allowed *in* and *out* of Motoko programs during calls to and from canisters.

Shared Types

Incoming and outgoing data are defined by the *argument types* and *return types* of **public shared functions** inside actors. Incoming and outgoing data types are restricted to a *subset* of available Motoko types, called *shared types*.

Shared types are always *immutable* types or data structures composed of immutable types (like **records**, **objects** and **variants**).

Shared types get their name from being *sharable* with the outside world, that is the wider internet beyond the **actors** running in **canisters** on the Internet Computer.

Shared Types in Public Shared Functions

Only shared types are allowed for *arguments* and *return values* of public shared functions. We give examples of a custom *public shared function type* called **SharedFunction** to illustrate shared types. Recall that a public shared function type includes the **shared** and **async** keywords.

Here are the most important shared types in Motoko.

Shared Primitive Types

All **primitive types** (except the **Error** type) are shared.

```
type SharedFunction = shared Nat -> async Text;
```

The argument of type **Nat** and the return value of type **Text** are shared types.

Shared Option Types

Any shared type in Motoko can be turned into an **Option** type which remains shared.

```
type SharedFunction = shared ?Principal -> async ?Bool;
```

The argument of type **?Principal** and the return value of type **?Bool** are shared types.

Shared Tuple Types

[Tuples](#) consisting of shared types are also shared, that is they are *shared tuple types*.

```
type SharedFunction = shared (Nat, Int, Float) -> async (Principal, Text);
```

The argument `(Nat, Int, Float)` and the return value `(Principal, Text)` are shared tuple types.

Shared Immutable Array Types

[Immutable arrays](#) consisting of shared types are also shared, that is they are *shared immutable array types*.

```
type SharedFunction = shared [Int] -> async [Nat];
```

The types `[Int]` and `[Nat]` are shared immutable array types.

Shared Variant Types

[Variant types](#) that have *shared associated types* are also shared.

```
type GenderAge = {
    #Male : Nat;
    #Female : Nat;
};

type SharedFunction = shared GenderAge -> async GenderAge;
```

The variant type `GenderAge` is a shared type because `Nat` is also shared.

Shared Object Types

[Object types](#) that have *shared field types* are also shared.

```
type User = {
    id : Principal;
    genderAge : GenderAge;
};

type SharedFunction = shared User -> async User;
```

Object type `User` is a shared type because `Principal` and `GenderAge` are also shared types.

Shared Function Types

`Shared function types` are also shared types. This example shows a shared public function that has another shared public function as its argument and return type.

```
type CheckBalance = shared () -> async Nat;  
type SharedFunction = shared CheckBalance -> async CheckBalance;
```

`CheckBalance` is a shared type because it is the type of a public shared function.

Shared Actor Types

All `actor types` are shared types.

```
type Account = actor {  
    checkBalance : shared () -> async Nat;  
};  
  
type SharedFunction = shared Account -> async Account;
```

`Account` is a shared type because it is the type of an actor.

Candid

All *shared types and values* in Motoko have a corresponding *description* in the 'outside world'. This description defines the types and values independently of Motoko or any other language. These alternative descriptions are written in a special *Interface Description Language* called [Candid](#).

Shared Types

Candid has a slightly different notation (syntax) and keywords to represent [shared types](#).

Primitive types

[Primitive types](#) in Candid are written without capital letters:

Motoko	Candid
Bool	bool
Nat	nat
Int	int
Float	float64
Principal	principal
Text	text
Blob	blob

Option types

[Option types](#) in Candid are written with the `opt` keyword. An option type in Motoko like `? Principal` would be represented in Candid as:

```
opt principal
```

Tuple types

[Tuple types](#) in Candid have the same parenthesis notation `()`. A Motoko tuple `(Nat, Text, Principal)` would be represented in Candid as:

```
(nat, text, principal)
```

Immutable array types

The [immutable array type](#) `[]` is represented in Candid with the `vec` keyword.

A Motoko array type `[Nat]` in candid looks like this:

```
vec nat
```

Variant types

[Variant types](#) in Candid are written with the `variant` keyword and curly braces `{ }`. A Motoko variant like `{#A : Nat; #B : Text}` would be represented in Candid like this:

```
variant {
  A : nat;
  B : text
};
```

The `#` character is not used

Object types

[Object types](#) in Candid are written with the `record` keyword and curly braces `{ }`. A Motoko object type like `{name : Text; age : Nat}` in Candid looks like this:

```
record {
  name : text;
  age : nat
};
```

Public shared function types

[Public shared function types](#) in Candid have a slightly different notation. A shared public function type in Motoko like `shared () -> async Nat` would be represented in Candid like this:

```
() -> (nat)
```

Parentheses `()` are used around the arguments and return types. The `shared` keyword is not used because all representable functions in Candid are by default only [public shared functions](#).

Another example would be the Motoko public shared function type `shared query Bool -> async Text` which in Candid would be represented as:

```
(bool) -> (text) query
```

Note that the `query` keyword appears after the return types.

A Motoko *oneway* public shared function type `shared Nat -> ()` in Candid would be represented as:

```
(nat) -> () oneway
```

The type keyword

Type aliases (custom names) in Candid are written with the `type` keyword. A Motoko type alias like `type MyType = Nat` would be represented in Candid like this:

```
type MyType = nat
```

Actor Interfaces

An [actor](#) running in a [canister](#) has a Candid description of its interface. An actor interface consists of the functions in the [actor type](#) and any possible types used within the actor type. Consider the following actor in a Motoko source file `main.mo`:

```
// main.mo
import Principal "mo:base/Principal";

actor {
    public type User = (Principal, Text);

    public shared query ({ caller = id }) func getUser() : async User {
        (id, Principal.toText(id));
    };

    public shared func doSomething() { () };
}
```

Only public types and **public shared functions** are included in the *candid interface*. This actor has a **public type** and two **public shared functions**. Both of these are part of its public interface.

The actor could have other fields, but they won't be included in the Candid Interface.

We describe this actor's interface in a Candid **.did** file. A Candid Interface contains all the information an external user needs to interact with the actor from the "outside world". The Candid file for the actor above would be:

```
// candid.did
type User = record {
    principal;
    text;
};

service : {
    getUser: () -> (User) query;
    doSomething: () -> () oneway;
}
```

Our Candid Interface consists of two parts: a **type** and a **service**.

The **service : {}** lists the *names* and *types* of the public shared functions of the actor. This is the information needed to *interact* with the actor from "the outside" by calling its functions. In fact, actors are sometimes referred to as *services*.

The type reflects the public Motoko type **User** from our actor. Since this is a public Motoko type that is used as a return type in a public shared function, it is included in the Candid Interface.

NOTE

*The type alias **User** is a Motoko tuple **(Principal, Text)**. In Candid a custom type alias for*

a tuple is translated into `record { principal; text }`. Don't confuse it with the Candid tuple type `(principal, text)`!

Candid Serialization

Another important use of Candid is *data serialization* of [shared types](#). Data structures in Motoko, like in any other language, are not always stored as serial (contiguous) bytes in [main memory](#). When we want to *send* shared data in and out of a canisters or store data in [stable memory](#), we have to *serialize* the data before sending.

Motoko has built in support for serializing shared types into Candid format. A *higher order* data type like an object can be converted into a *binary blob* that would still have a shared type.

Consider the following relatively complex data structure:

```
type A = { #a : Nat; #b : Int };

type B = { #a : Int; #b : Nat };

type MyData = {
    title : Text;
    a : A;
    b : B;
};
```

Our object type `MyData` contains a `Text` field and fields of variant types `A` and `B`. We could turn a *value* of type `MyData` into a value of type `Blob` by using the `to_candid()` and `from_candid()` functions in Motoko.

```
let data : MyData = {
    title = "Motoko";
    a = #a(1);
    b = #a(-1);
};

let blob : Blob = to_candid (data);
```

We declared a variable of type `MyData` and assigned it a value. Then we *serialized* that data into a `Blob` by using `to_candid()`.

This blob can now be sent or received in arguments or return types of public shared functions or stored in [stable memory](#).

We could recover the original type by doing the opposite, namely deserializing the data back into a Motoko shared type by using `from_candid()`.

```
let deserialized_data : ?MyData = from_candid (blob);

switch (deserialized_data) {
    case null {};
    case (?data) {};
}

};
```

We declare a variable with option type `?MyData`, because the `from_candid()` function always returns an option of the original type. Type annotation is required for this function to work. We use a `switch` statement after deserializing to handle both cases of the option type.

Basic Memory Persistence

Every **actor** exposes a **public interface**. A deployed actor is sometimes referred to as a **service**. We interact with a service by calling the **public shared functions** of the underlying actor.

The state of the actor can not change during a **query** function call. When we call a query function, we could pass in some data as arguments. This data could be used for a *computation*, but no data could be stored in the actor during the query call.

The state of the actor may change during an **update** or **oneway** function call. Data provided as arguments (or data generated without arguments in the function itself) could be *stored* inside the actor.

Canister main memory

Every actor running in a canister has access to a 4GB main 'working' memory (in Feb 2023). This is like the RAM memory.

Code in actors directly acts on main memory:

- The values of (top-level) **mutable** and **immutable** variables are stored in main memory.
- **Public** and private functions in actors, that read and write data, do so from and to main memory.
- Imported **classes** from **modules** are instantiated as **objects** in main memory.
- **Imported functions** from modules operate on main memory.

The same applies for any other imported **item** that is used inside an actor.

Memory Persistence across function calls

Consider the actor from our **previous example** with only the mutable variable **latestComment** and the functions **readComment** and **writeComment**:

```

actor {
    private var latestComment = "";

    public shared query func readComment() : async Text {
        latestComment;
    };

    public shared func writeComment(comment : Text) : async () {
        latestComment := comment;
    };
}

```

The mutable variable `latestComment` is stored in main memory. Calling the update function `writeComment` *mutates* the state of the mutable variable `latestComment` in main memory.

For instance, we could call `writeComment` with an argument `"Motoko is great!"`. The variable `latestComment` will be set to that value in main memory. The mutable variable now has a *new state*. Another call to `readComment` would return the new value.

Service upgrades and main memory

Now, suppose we would like to *extend the functionality* of our service by adding another *public shared function* (like the `deleteComment` function). We would need to write the function in Motoko, edit our original Motoko source file and go through the deployment process again.

The redeployment of our actor will wipe out the main memory of the actor!

There are two main ways to upgrade the functionality of a service without resetting the *memory state* of the underlying actor.

This chapter describes *stable variables* which are a way to *persist* the state of mutable variables across *upgrades*. Another way is to use *stable memory*, which will be discussed later in this book.

Upgrades

An [actor](#) in Motoko is written in a source file (often called `main.mo`). The Motoko code is compiled into a [Wasm module](#) that is *installed* in a [canister](#) on the Internet Computer. The canister provides [main memory](#) for the actor to operate on and store its memory state.

If we want to incrementally develop and evolve a Web3 Service, we need to be able to *upgrade* our actor code.

New actor code results in a new Wasm module that is different from the older module that is running inside the canister. We need to replace the old module with the new one containing our upgraded actor.

Immutable Microservice

Some actors, may stay unchanged after their first installment in a canister. Small actors with one simple task are sometimes called *microservices* (that may be optionally *immutable* and not owned by anyone). A microservice consumes little resources (memory and cycles) and could operate for a long time without any upgrades.

Reinstall and upgrade

There are two ways to *update* the Wasm module of an actor running in a canister.

Reinstall

When we have an actor Wasm module running in a canister, we could always reinstall that same actor module or a new actor module inside the same canister. Reinstalling always causes the actor to be 'reset'. Whether reinstalling the same actor Wasm module or a new one, the operation is the same as if installing for the first time.

Reinstalling always wipes out the main memory of the canister and erases the state of the actor

Upgrade

We could also choose to *upgrade* the Wasm module of an actor. Then:

- The [actor interface](#) is checked for [backwards compatibility](#)
 - The [pre-upgrade system function](#) is run before the upgrade
 - The canister [Wasm module](#) is upgraded
 - [Stable variables](#) are restored
 - The [post upgrade system function](#) is run after the upgrade
-

NOTE

Pre and post upgrade hooks could trap and lead to loss of canister data and thus are not considered best practice.

Service upgrades and sub-typing

When upgrading a service, we may also upgrade the [public interface](#). This means that our [actor type](#) and [public interface description](#) may change.

An older [client](#) that is not aware of the change may still use the old interface. This can cause problems if for instance a client calls a function that no longer exists in the interface. This is called a *breaking change*, because it 'breaks' the older clients.

To avoid breaking changes, we could extend the functionality of a service by using [sub-typing](#). This preserves the *old interface* rather than the memory state and is called *backwards compatibility*. This will be discussed [later](#) in this book.

Upgrading and reinstalling in Motoko Playground and SDK

When we have an already running canister in [Motoko Playground](#) and we click [deploy](#) again, we are presented with the option to [upgrade](#) or [reinstall](#) (among other options).

The same functionality is provided when using the [SDK](#).

Stable Variables

To persist the state of an actor when [upgrading](#), we can declare [immutable](#) and [mutable](#) variables to be [stable](#). Stable variables must be of [stable type](#).

Modifying and Upgrading Stable Variables

The [state of an actor](#) is stored in the form of *immutable* and *mutable* variables that are declared with the `let` or `var` keywords. Variable declarations in actors always have [private visibility](#) (although the `private` keyword is not necessary and is assumed by default).

If we want to retain the state of our actor when [upgrading](#), we need to declare all its state variables as [stable](#). A variable can only be declared stable if it is of [stable type](#). Many, but not all types, are stable. Mutable and immutable stable variables are declared like this:

```
stable let x = 0;
stable var y = 0;
```

These variables now *retain* their state even after *upgrading* the actor code. Lets demonstrate this with an actor that implements a simple counter.

```
actor {
    stable var count : Nat = 0;

    public shared query func read() : async Nat { count };

    public shared func increment() : async () { count += 1 };
};
```

Our actor has a mutable variable `count` that is declared [stable](#). It's initial value is `0`. We also have a [query](#) function `read` and an [update](#) function `increment`. The value of `count` is shown below in a *timeline of state* in two instances of our counter actor: one with `var count` and the other with `stable var count`:

Time	Event	Value of	Value of
		<code>var count</code>	<code>stable var count</code>
1	First install of actor code	0	0
2	Call <code>increment()</code>	1	1

Time	Event	Value of	Value of
		var count	stable var count
3	Call <code>read()</code>	1	1
4	Call <code>increment()</code>	2	2
5	Upgrade actor code	0	2
6	Call <code>read()</code>	0	2
7	Call <code>increment()</code>	1	3
8	Call <code>increment()</code>	2	4
9	Reinstall actor code	0	0
10	Call <code>read()</code>	0	0
11	Call <code>increment()</code>	1	1
12	Call <code>increment()</code>	2	2

Time 1: Our initial value for `count` is 0 in both cases.

Time 2: An update function mutates the state in both cases.

Time 3: A query function does not mutate state in both cases.

Time 5: `stable var count` value is persisted after `upgrade`.

Time 6: `var count` is reset after upgrade.

Time 7: `var count` starts at 0, while `stable var count` starts at 2.

Time 10: `var count` and `stable var count` are both reset due to `reinstall`.

Stable types

A type is *stable* if it is `shared` and remains shared after ignoring any `var` keywords within it. An `object` with private functions is also stable. Stable types thus include all `shared types` plus some extra types that contain `mutable variables` and object types with private functions.

Stable variables can only be declared inside `actors`. Stable variables always have `private visibility`.

The following types for `immutable` or `mutable` variables in actors (in addition to all `shared types`) could be declared stable.

Stable Mutable Array

Immutable or mutable variables of `mutable array type` could be declared stable:

```
stable let a1 : [var Nat] = [var 0, 1, 2];
stable var a2 : [var Text] = [var "t1", "t2", "t3"];
```

Immutable variable a1 can be stable because [**var Nat**] is a mutable array type. *Mutable variable a2* can be stable because [**var Text**] is of mutable array type.

Stable Records with Mutable Fields

Immutable or mutable variables of *records with mutable fields* could be declared stable:

```
stable let r1 = { var x = 0 };
stable var r2 = { var x = 0; y = 1 };
```

Immutable variable r1 and *mutable variable r2* can be stable because they contain a stable type, namely a *record with a mutable field*.

Stable Objects with Mutable Variables

Immutable or mutable variables of *objects with (private or public) mutable variables* could be declared stable:

```
stable let o1 = object {
    public var x = 0;
    private var y = 0;
};

stable var o2 = object {
    public var x = 0;
    private var y = 0;
};
```

Immutable variable o1 and *mutable variable o2* can be stable because they contain a stable type, namely an *object with private and public mutable variables*.

Stable Objects with Private Functions

Immutable or mutable variables of *objects with private functions* could be declared stable:

```
stable let p1 = object { private func f() {} };  
stable var p2 = object { private func f() {} };
```

Immutable variable `p1` and *mutable variable* `p2` can be stable because they contain a stable type, namely an *object with a private function*.

Other stable types

On top all **shared types** and the types mentioned above, the following types could also be stable:

- **Tuples** of stable types
- **Option** types of any stable types
- **Variant** types with associated types that are stable

How it works

Declaring variable(s) `stable` causes the following to happen *automatically* when **upgrading** our canister with new actor code:

- The value of the variable(s) is *serialized* into an internal data format.
- The serialized data format is copied to **stable memory**.
- The upgraded actor code (in the form of a **Wasm module**) is installed in the **canister** and the state of the variables is lost.
- The values in data format inside stable memory are retrieved and **deserialized**.
- The variables of the new actor code are assigned the original serialized values.

Example of non-stable type

A non-stable type could be an **object with public functions**. A variable that contains such an object, could not be declared stable!

```
stable let q1 = object {  
    public func f() {};  
};
```

ERROR: Variable `q1` is declared stable but has non-stable type!

Advanced Types

Motoko has a modern *type system* that allows us to write powerful, yet concise code. In this chapter we will look at several features of Motoko's type system.

Understanding Motoko's type system is essential when we write code that is more sophisticated than what we have done up to this point.

- **Generic types** are widely used in **types declarations**, **functions** and **classes** inside **modules** of the **Base Library**.
- **Subtyping** is useful when we want to develop **actor interfaces** that remain *backwards compatible* with older versions.
- **Recursive types** are powerful when defining **core data structures**.

Generics

Let's ask [ChatGPT](#) how it would explain *generic types* to a beginner:

Q: How would you explain generic typing to a beginner?

ChatGPT: *Generics are a way of creating flexible and reusable code in programming. They allow you to write functions, classes, and data structures that can work with different types of data without specifying the type ahead of time.*

In our own words, generic types allow us to write code that *generalizes* over many possible types. In fact, that is where they get their name from.

Type parameters and type arguments

In Motoko, [custom types](#), [functions](#) and [classes](#) can specify *generic type parameters*. Type parameters have *names* and are declared by adding them in between angle brackets `< >`. The angle brackets are declared directly after the *name* of the type, function or class (before any other parameters).

Type parameters

Here's a custom type alias for a [tuple](#), that has the *conventional* generic type parameter `T`:

```
type CustomType<T> = (Nat, Int, T);
```

The type parameter `T` is supplied after the custom type name in angle brackets:

`CustomType<T>`. Then the generic type parameter is used inside the tuple. This indicates that a value of `CustomType` will have *some* type `T` in the tuple (alongside known types `Nat` and `Int`) without knowing ahead of time what type that would be.

The names of generic type parameters are by convention single capital letters like `T` and `U` or words that start with a capital letter like `Ok` and `Err`.

Generic parameter type names must start with a letter, may contain lowercase and uppercase letters, and may also contain numbers `0-9` and underscores `_`.

Type arguments

Lets use our `CustomType` :

```
let x : CustomType<Bool> = (0, -1, true);
```

The `CustomType` is used to *annotate* our variable `x`. When we actually *construct* a value of a generic type, we have to specify a specific type for `T` as a *type argument*. In the example, we used `Bool` as a type argument. Then we wrote a tuple of values, and used a `true` value where a value of type `T` was expected.

The same `CustomType` could be used again and again with different type arguments for `T`:

```
let y : CustomType<Float> = (100, -100, 0.5);
let z : CustomType<[Nat]> = (100, -100, [7, 6, 5]);
```

In the last example we used `[Nat]` as a type argument. This means we have to supply an *immutable array* of type `[Nat]` for the `T` in the tuple.

Generics in type declarations

Generics may be used in type declarations of *compound* types like objects and variants.

Generic variant

A commonly used generic type is the `Result` *variant type* that is available as a *public type* in the `Base Library` in the `Result module`.

```
public type Result<Ok, Err> = {
    #ok : Ok;
    #err : Err;
};
```

A type alias `Result` is declared, which has two type parameters `Ok` and `Err`. The type alias is used to refer to a variant type with two variants `#ok` and `#err`. The variants have *associated types* attached to them. The associated types are of generic type `Ok` and `Err`.

This means that the variants can take on many different associated types, depending on how we want to use our `Result`.

Results are usually used as a return value for functions to provide information about a possible *success* or *failure* of the function:

```
func checkUsername(name : Text) : Result<(), Text> {
    let size = name.size();
    if (size < 4) #err("Too short!") else if (size > 20) #err("To long!") else
#ok();
};
```

Our function `checkUsername` takes a `Text` argument and returns a value of type `Result<(), Text>`. The unit type `()` and `Text` are *type arguments*.

The function checks the size of the its argument `name`. In case `name` is shorter than 4 characters, it returns an 'error' by constructing a *value* for the `#err` variant and adding an associated value of type `Text`. The return value would in this case be `#err("Too short!")` which is a valid value for our `Result<(), Text>` variant.

Another failure scenario is when the argument is longer than 20 characters. The function returns `#err("To long!")`.

Finally, if the size of the argument is allowed, the function indicates success by constructing a value for the `#ok` variant. The associated value is of type `()` giving us `#ok()`.

If we use this function, we could `switch` on its return value like this:

```
let result = checkUsername("SamerWeb3");

switch (result) {
    case (#ok()): {};
    case (#err(error)): {};
}
```

We `pattern match` on the possible variants of `Result<(), Text>`. In case of the `#err` variant, we also *bind* the associated `Text` value to a new name `error` so we could use it inside the scope of the `#err` case.

Generic object

Here's a type declaration of an object type `Obj` that takes three *type parameters* `T`, `U` and `V`. These type parameters are used as types of some variables of the object.

```
type Obj<T, U, V> = object {
    a : T;
    b : T -> U;
    var c : V;
    d : Bool;
};
```

The first variable `a` is of generic type `T`. The second variable `b` is a [public function in the object](#) with generic arguments and return type `T -> U`. The third variable is mutable and is of type `V`. And the last variable `d` does not use a generic type.

We would use it like this:

```
let obj : Obj<Nat, Int, Text> = {
    a = 0;
    b = func(n : Nat) { -1 * n };
    var c = "Motoko";
    d = false;
};
```

We declare a variable `obj` of type `Obj` and use `Nat`, `Int` and `Text` as *type arguments*. Note that `b` is a function that takes a `Nat` and returns an `Int`.

Generics in functions

Generic types are also found in functions. Functions that allow type parameters are:

- Private and public functions in [modules](#) and [nested modules](#)
- Private and public functions in [objects](#)
- Private and public functions in [classes](#)
- **Only private functions** in [actors](#)

NOTE

Public shared functions in actors are **not allowed** to have generic type arguments.

Some [public functions in modules](#) of the [Base Library](#) are written with generic type parameters. Lets look the useful `init` public function found in the `Array` module of the Base Library:

```
public func init<X>(size : Nat, initialValue : X) : [var X]
// Function body is omitted
```

This function is used to construct a **mutable array** of a certain **size** filled with copies of *some initialValue*. The function takes one generic type parameter **x**. This parameter is used to specify the type of the initial value **initialValue**.

It may be used like this:

```
import Array "mo:base/Array";

let t = Array.init<Bool>(3, true);
```

We **import** the **Array.mo** module and name it **Array**. We access the function with **Array.init**. We supply *type arguments* into the angle brackets, in our case **<Bool>**. The second argument in the function (**initialValue**) is now of type **Bool**.

A mutable array will be constructed with **3** mutable elements of value **true**. The value of **t** would therefore be

[var true, true, true]

Generics in classes

Generics may be used in **classes** to construct **objects** with generic types.

Lets use the **Array.init** function again, this time in a class:

```
import Array "mo:base/Array";

class MyClass<T>(n : Nat, initVal : T) {
    public let array : [var T] = Array.init<T>(n, initVal);
};
```

The class **MyClass** takes one generic *type parameter* **T**, which is used three times in the class declaration.

1. The class takes two arguments: **n** of type **Nat** and **initVal** of generic type **T**.
2. The public variable **array** is annotated with **[var T]**, which is the type returned from the **Array.init** function.
3. The function **Array.init** is used inside the class and takes **T** as a type parameter.

Recall that a class is just a *constructor for objects*, so lets make an object with our class.

```
let myObject = MyClass<Bool>(2, true);  
  
// myClass.array now has value [true, true]  
myObject.array[0] := false;
```

We construct an object `myObject` by calling our class with a *type argument* `Bool` and two *argument values*. The second argument must be of type `Bool`, because that's what we specified as the type argument for `T` and `initVal` is of type `T`.

We now have an object with a public variable `array` of type `[var Bool]`. So we can reference an element of that array with the angle bracket notation `myObject.array[]` and mutate it.

Sub-typing

Motoko's modern type system allows us to think of some types as being either a *subtype* or a *supertype*. If a type `T` is a subtype of type `U`, then a value of type `T` can be used everywhere a value of type `U` is expected.

To express this relationship we write:

`T <: U`

`T` is a *subtype* of `U`. And `U` is a *supertype* of `T`.

Nat and Int

The most common subtype in Motoko is `Nat`. It is a subtype of `Int`, making `Int` its supertype.

A `Nat` can be used everywhere an `Int` is expected, but not the other way around.

```
func returnInt() : Int {
    0 : Nat;
};
```

The function `returnInt` has to return an `Int` or some subtype of `Int`. Since `Nat` is a subtype of `Int`, we could return `0 : Nat` where an `Int` was expected.

Here's another example of using a `Nat` where an `Int` is expected:

```
type T = (Nat, Int, Text);

let t : T = (1 : Nat, 2 : Nat, "three");
```

Our tuple type `T` takes an `Int` as the second element of the tuple. But we construct a value with a `Nat` as the second element.

We can not supply a `Nat` for the third element, because `Nat` is not a subtype of `Text`. Neither could we supply a negative number like `-10 : Int` where a `Nat` or `Text` is expected, because `Int` is not a subtype of `Nat` or `Text`.

Subtyping variants

A variant `v1` can be a subtype of some other variant `v2`, giving us the relationship `v1 <: v2`. Then we could use a `v1` everywhere a `v2` is expected. For a variant to be a *subtype* of another variant, it must contain a *subset* of the fields of its *supertype* variant.

Here's an example of two variants both being a subtype of another third variant:

```
type Red = {
    #red : Nat8;
};

type Blue = {
    #blue : Nat8;
};

type RGB = {
    #red : Nat8;
    #blue : Nat8;
    #green : Nat8;
```

Variant type `RGB` is a supertype of both `Red` and `Blue`. This means that we can use `Red` or `Blue` everywhere a `RGB` is expected.

```
func rgb(color : RGB) { () };

let red : Red = #red 255;
rgb(red);

let blue : Blue = #blue 100;
rgb(blue);

let green : RGB = #green 150;
rgb(green);
```

We have a function `rgb` that takes an argument of type `RGB`. We could construct values of type `Red` and `Blue` and pass those into the function.

Subtyping objects

The subtype-supertype relationship for objects and their fields, is reversed compared to [variants](#). In contrast to variants, an object subtype has more fields than its object supertype.

Here's an example of two object types `User` and `NamedUser` where `NamedUser <: User`.

```
type User = {
    id : Nat;
};

type NamedUser = {
    id : Nat;
    name : Text;
};
```

The supertype `User` only contains one field named `id`, while the subtype `NamedUser` contains two fields, one of which is `id` again. This makes `NamedUser` a subtype of `User`.

This means we could use a `NamedUser` wherever a `User` is expected, but not the other way around.

Here's an example of a function `getId` that takes an argument of type `User`. It will reference the `id` field of its argument and return that value which is expected to be a `Nat` by definition of the `User` type.

```
func getId(user : User) : Nat {
    user.id;
};
```

So we could construct a value of expected type `User` and pass it as an argument. But an argument of type `NamedUser` would also work.

```
let user : User = { id = 0 };
let id = getId(user);

let namedUser : NamedUser = {
    id = 1;
    name = "SamerWeb3";
};
let id1 = getId(namedUser);
```

Since `NamedUser` also contains the `id` field, we also constructed a value of type `NamedUser` and used it where a `User` was expected.

Backwards compatibility

An important use of subtyping is *backwards compatibility*.

Recall that `actors` have `actor interfaces` that are basically comprised of `public shared functions` and `public types in actors`. Because actors are `upgradable`, we may *change* the interface during

an upgrade.

It's important to understand what kind of changes to actors maintain *backwards compatibility* with existing clients and which changes *break* existing clients.

NOTE

Backwards compatibility depends on the [Candid interface](#) of an actor. Candid has more permissive subtyping rules on variants and objects than Motoko. This section describes Motoko subtyping.

Actor interfaces

The most common upgrade to an actor is the addition of a public shared function. Here are two versions of [actor types](#):

```
type V1 = actor {
    post : shared Nat -> async ();
};

type V2 = actor {
    post : shared Nat -> async ();
    get : shared () -> async Nat;
};
```

The first actor only has one public shared function `post`. We can upgrade to the second actor by adding the function `get`. This upgrade is backwards compatible, because the *function signature* of `post` did not change.

Actor `V2` is a subtype of actor `V1` (like in the case of [object subtyping](#)). This example looks similar to object subtyping, because `V2` includes all the functions of `V1`. We can use `V2` everywhere an actor type `V1` is expected.

Breaking backwards compatibility

Lets demonstrate an upgrade that would NOT be backwards compatible. Lets upgrade to `V3`:

```
type V3 = actor {
    post : shared Nat -> async ();
    get : shared () -> async Int;
};
```

We only changed the return type of the `get` function from `Nat` to `Int`. This change is NOT backwards compatible! The reason for this is that the new version of the public shared function `get` is not a *subtype* of the old version.

Subtyping public shared functions

Public shared functions in actors can have a subtype-supertype relationship. A public shared function `f1` could be a subtype of another public shared function `f2`, giving us the the relationship `f1 <: f2`.

This means that we can use `f1` everywhere `f2` is expected, but not the other way around. But more importantly, we could safely upgrade from the supertype `f2` to the subtype `f1` without breaking backwards compatibility.

Lets look at an example where `f1 <: f2`:

```
public shared func f1(a1 : Int) : async Nat { 0 : Nat };
public shared func f2(a2 : Nat) : async Int { 0 : Int };
```

Function `f1` is a subtype of `f2`. For this relationship to hold, two conditions must be satisfied:

- The *return type* of `f1` must be a subtype of the return type of `f2`
- The *argument(s)* of `f2` must be a subtype of the argument(s) of `f1`

The return type `Nat` of function `f1` is a subtype of return type `Int` of function `f2`.

The argument `Nat` of function `f2` is a subtype of argument `Int` of function `f1`.

The types of `f1` and `f2` in Motoko are:

```
type F1 = shared Int -> async Nat;
type F2 = shared Nat -> async Int;
```

Upgrading a public shared function of type `F2` to a public shared function of type `F1` is considered a backwards compatible upgrade.

Since `F1` is a subtype of `F2` and public shared functions are a [shared type](#), we could use these types like this:

```
public shared func test() : async F2 { f1 };
```

We are returning `f1` of type `F1` where a `F2` is expected.

Upgrading public shared functions

Suppose we have the following types:

```
type Args = {
    #a;
    #b;
};

type ArgsV2 = {
    #a;
    #b;
    #c;
};

type Result = {
    data : [Nat8];
};

type ResultV2 = {
    data : [Nat8];
    note : Text;
};
```

Variant `Args` is a subtype of variant `ArgsV2`. Also, object type `ResultV2` is a subtype of object type `Result`.

We may have a function that uses `Args` as the argument type and `Result` as the return type:

```
public func post(n : Args) : async Result { { data = [0, 0] } };
```

We could upgrade to a new version with types `ArgsV2` and `ResultV2` without breaking backwards compatibility, as long as we keep the same *function name*:

```
public func post(n : ArgsV2) : async ResultV2 {
    { data = [0, 0]; note = "" };
};
```

This is because `Args` is a *subtype* of `ArgsV2` and `Result` is a *supertype* of `ResultV2`.

Recursive Types

Lets ask everyone's favorite AI about recursive types:

Q: How would you define recursive types?

ChatGPT: Recursive types are types that can contain values of the same type. This self-referential structure allows for the creation of complex data structures. Recursive types are commonly used in computer programming languages, particularly in functional programming languages, to represent recursive data structures such as trees or linked lists.

Wow! And since Motoko provides an [implementation](#) of a *linked list*, a common [data structure](#) in programming languages, we will use that as an example.

List

Consider the following type declaration in the [List](#) module in the [Base Library](#):

```
type List<T> = ?(T, List<T>);
```

A [generic type](#) `List<T>` is declared which takes one [type parameter](#) `T`. It is a [type alias](#) for an [option](#) of a [tuple](#) with two elements.

The first element of the tuple is the type parameter `T`, which could take on any type during the construction of a value of this `List<T>` type by providing a [type argument](#).

The second element of the tuple is `List<T>`. This is where the same type is used *recursively*. The type `List<T>` contains a value of itself in its own definition.

This means that a `List` is a *repeating pattern* of elements. Each element is a tuple that contains a value of type `T` and a reference to the *tail* `List<T>`. The tail is just another list element of the same type with the next value and tail for the next value and so on...

Null list

Since our `List<T>` type is an option of a tuple, a value of type `<List<T>` could be `null` as long as we initialize a value with a [type argument](#) for `T`:

```
let list : List<Nat> = null;
```

Our variable `list` has type `List<Nat>` (a list of `Nat` values) and happens to have the value `null`.

The 'tail' of a list

The shortest list possible is a list with exactly one element. This means that it does not refer to a tail for the next value, because there is no next value.

```
let list1 : List<Nat> = ?(0, null);
```

The second element in the tuple, which should be of type `List<T>` is set to `null`. We use the `null list` as the second element of the tuple. This list could serve as the last element of a larger list.

The 'head' of a list

If we wanted to add another value to our list `list1` we could just define another value of type `List<T>` and have it point to `list1` as its tail:

```
let list2 : List<Nat> = ?(1, list1);
```

`list2` is now called the head of the list (the first element) and `list1` is the tail of our 2-element list.

We could repeat this process by adding another element and using `list2` as the tail. We could do this as many times as we like (within `memory` limits) and construct a list of many values.

Recursive functions

Suppose we have a list `bigList` with many elements. This means we are presented with a head of a list, which is a value of type `<List<T>`.

There several possibilities for this head value:

- The head value `bigList` could be the `null list` and in that case it would not contain any values of type `T`.

- Another possibility is that the head value `bigList` has a value of type `T` but does not refer to a tail, making it a single element list, which could be the last element of a larger list.
- Another possibility is that the head value `bigList` contains one value of type `T` and a tail, which is another value of type `List<T>`.

These possibilities are used in the `last<T>` function of the [List module](#). This function is used to retrieve the last element of a given list:

```
func last<T>(l : List<T>) : ?T {
    switch l {
        case null { null };
        case ?(x, null) { ?x };
        case ?(_, t) { last<T>(t) };
    };
}
```

We have a [generic function](#) `List<T>` with one [type parameter](#) `T`. It takes one argument `l` of type `List<T>`, which is going to be a head of some list.

If the function finds a last element, it returns an option of the value `?T` within it. If there is no last element it returns `null`.

In the function body, we [switch](#) on the argument `l : List<T>` and are presented with the three possibilities described above.

- In the case that `l` is the [null list](#), we return `null` because there would not be a last element.
- In the case that `l` is a list element `?(x, null)`, then we are dealing with the last element because there is not tail. We bind the name `x` to that last value by [pattern matching](#) and return an option `?x` of that value as is required by our function signature.
- In the case that `l` is a list element `?(_, t)`, then there is a tail with a next value. We use the wildcard `_` because we don't care about the value in this element, because it is not the last value. We do care about the tail for which we bind the name `t`. We now use the function `last<T>` *recursively* by calling it again inside itself and provide the tail `t` as the argument: `last<T>(t)`. The function is called again receiving a list which it now sees as a head that it can switch on again and so on until we find the last element. Pretty cool, if you ask me!

Type Bounds

When we express a [subtype-supertype relationship](#) by writing `T <: U`, then we say that `T` is a *subtype* by `U`. We can use this relationship between two types in the instantiation of [generic types in functions](#).

Type bounds in functions

Consider the following *function signature*:

```
func makeNat<T <: Number>(x : T) : Natural
```

It's a [generic function](#) that specifies a *type bound* `Number` for its generic type parameter `T`. This is expressed as `<T <: Number>`.

The function takes an argument of generic bounded type `T` and returns a value of type `Natural`. The function is meant to take a general kind of number and process it into a `Nat`.

The types `Number` and `Natural` are declared like this:

```
type Natural = {
    #N : Nat;
};

type Integer = {
    #I : Int;
};

type Floating = {
    #F : Float;
};

type Number = Natural or Integer or Floating;
```

The types `Natural`, `Integer` and `Floating` are just [variants](#) with one field and associated types `Nat`, `Int` and `Float` respectively.

The `Number` type is a *type union* of `Natural`, `Integer` and `Floating`. A type union is constructed using the `or` keyword. This means that a `Number` could be either a `Natural`, an `Integer` or a `Floating`.

We would use these types to implement our function like this:

```

import Int "mo:base/Int";
import Float "mo:base/Float";

func makeNat<T <: Number>(x : T) : Natural {
    switch (x) {
        case (#N n) {
            #N n;
        };
        case (#I i) {
            #N(Int.abs(i));
        };
        case (#F f) {
            let rounded = Float.nearest(f);
            let integer = Float.toInt(rounded);
            let natural = Int.abs(integer);
            #N natural;
        };
    };
}

```

After [importing](#) the `Int` and `Float` modules from the [Base Library](#), we declare our function and implement a [switch expression](#) for the argument `x`.

In case we find a `#N` we know we are dealing with a `Natural` and thus immediately return the the same variant and associated value that we refer to as `n`.

In case we find an `#I` we know we are dealing with an `Integer` and thus take the associated value `i` and apply the `abs()` function from the [Int module](#) to turn the `Int` into a `Nat`. We return a value of type `Natural` once again.

In case we find a `#F` we know we are dealing with a `Floating`. So we take the associated value `f` of type `Float`, round it off and convert it to an `Int` using functions from the [Float module](#) and convert to a `Nat` again to return a value of type `Natural` once again.

Lets test our function using some [assertions](#):

```

assert makeNat(#N 0) == #N 0;

assert makeNat(#I(-10)) == #N 10;

assert makeNat(#F(-5.9)) == #N 6;

```

We use arguments of type `Natural`, `Integer` and `Floating` with associated types `Nat`, `Int` and `Float` respectively. They all are accepted by our function.

In all three cases, we get back a value of type `Natural` with an associated value of type `Nat`.

The Any and None types

All types in Motoko are bounded by a special type, namely the `Any` type. This type is the *supertype* of all types and thus all types are a *subtype* of the `Any` type. We may refer to it as the *top type*. Any value or expression in Motoko can be of type `Any`.

Another special type in Motoko is the `None` type. This type is the *subtype* of all types and thus all types are a *supertype* of `None`. We may refer to it as the *bottom type*. No value in Motoko can have the `None` type, but some expressions can.

NOTE

Even though no value has type `None`, it is still useful for typing expressions that don't produce a value, such as infinite loops, early exits via `return` and `throw` and other constructs that divert control-flow (like `Debug.trap : Text -> None`)

For any type `T` in Motoko, the following *subtype-supertype* relationship holds:

`None <: T`

`T <: Any`

The Base Library

The Motoko Base Library is a collection of [modules](#) with basic functionality for working with *primitive types*, *utilities*, *data structures* and other functionality.

Most modules define a [*public type*](#) that is associated with the core use of the module.

The Base Library also includes [*IC system APIs*](#), some of which are still experimental (June 2023). We will visit them [later in this book](#).

Importing from the Base Library

An [*import*](#) from the base library looks like this:

```
import P "mo:base/Principal";
```

We imported the [*Principle*](#) module. The [*P*](#) in the example is an arbitrary *alias* (name) that we choose to *reference* our module. The import path starts with [*mo:base/*](#) followed by the *file name* where the module is defined (without the [*.mo*](#) extension).

The *file name* and our chosen *module name* do not have to be the same (like in the example above). It is a convention though to use the [*same name*](#):

```
import Principal "mo:base/Principal";
```

Our imported module, now named [*Principal*](#), has a [*module type*](#) `module { ... }` with public fields. It exposes several [*public types*](#) and [*public functions*](#) that we can now reference by using our alias:

```
let p : Principal = Principal.fromText("2vxsx-fae");
```

Note the two meanings of [*Principal*](#):

- The [*type annotation*](#) for variable [*p*](#) uses the always available [*primitive type*](#) [*Principal*](#). This *type* does not have to be imported.
- We used the [*.fromText\(\)*](#) *method* from the [*Principal*](#) module (public functions in [*modules*](#) or in [*objects or classes*](#) are often called methods) by referencing it through our chosen *module name* [*Principal*](#).

Primitive Types

As described [earlier](#) in this book, *primitive types* are core data types that are not *composed* of more fundamental types.

Primitive types do **not** need to be imported before they can be used in *type annotation*. But the Base Library does have *modules* for each primitive type to perform common tasks, like *converting* between the data types, *transforming* the data types, *comparing* the data types and other functionality.

In this chapter we visit some of the functionality provided by the Base Library modules for several primitive data types.

Bool

The *convention* is to name the *module alias* after the *file name* it is defined in:

```
import Bool "mo:base/Bool";
```

Comparison

```
Function equal  
Function notEqual  
Function compare
```

Conversion

```
Function toText
```

Logical operations

```
Function lognot  
Function logand  
Function logor  
Function logxor
```

Bool.equal

```
func equal(x : Bool, y : Bool) : Bool
```

The function `equal` takes two `Bool` arguments and returns a `Bool` value. It is equivalent to the `==` relational operator.

```
import Bool "mo:base/Bool";  
  
let a = true;  
let b = false;  
  
Bool.equal(a, b);
```

Bool.notEqual

```
func notEqual(x : Bool, y : Bool) : Bool
```

The function `notEqual` takes two `Bool` arguments and returns a `Bool` value. It is equivalent to the `!=` relational operator.

```
import Bool "mo:base/Bool";  
  
let a = true;  
let b = false;  
  
Bool.notEqual(a, b);
```

Bool.compare

```
func compare(x : Bool, y : Bool) : Order.Order
```

The function `compare` takes two `Bool` arguments and returns an `Order` variant value.

```
import Bool "mo:base/Bool";  
  
let a = true;  
let b = false;  
  
Bool.compare(a, b);
```

Bool.toText

```
func toText(x : Bool) : Text
```

The function `toText` takes one `Bool` argument and returns a `Text` value.

```
import Bool "mo:base/Bool";

let isPrincipal = true;

Bool.toText(isPrincipal);
```

Bool.lognot

```
func lognot(x : Bool) : Bool
```

The function `lognot` takes one `Bool` argument and returns a `Bool` value. It stands for *logical not*. It is equivalent to the [not expression](#).

```
import Bool "mo:base/Bool";

let positive = true;

Bool.lognot(positive);
```

Bool.logand

```
func logand(x : Bool, y : Bool) : Bool
```

The function `logand` takes two `Bool` arguments and returns a `Bool` value. It stands for *logical and*. It is equivalent to the [and expression](#).

```
import Bool "mo:base/Bool";

let a = true;
let b = false;

Bool.logand(a, b);
```

Bool.logor

```
func logor(x : Bool, y : Bool) : Bool
```

The function `logor` takes two `Bool` arguments and returns a `Bool` value. It stands for *logical or*. It is equivalent to the `or` expression.

```
import Bool "mo:base/Bool";  
  
let a = true;  
let b = false;  
  
Bool.logor(a, b);
```

Bool.logxor

```
func logxor(x : Bool, y : Bool) : Bool
```

The function `logxor` takes two `Bool` arguments and returns a `Bool` value. It stands for *exclusive or*.

```
import Bool "mo:base/Bool";  
  
let a = true;  
let b = true;  
  
Bool.logxor(a, b);
```

Nat

The *convention* is to name the *module alias* after the *file name* it is defined in:

```
import Nat "mo:base/Nat";
```

Conversion

Function `toText`

Comparison

```
Function min
Function max
Function compare
Function equal
Function notEqual
Function less
Function lessOrEqual
Function greater
Function greaterOrEqual
```

Numerical operations

```
Function add
Function sub Careful! Traps if result is a negative number
Function mul
Function div
Function rem
Function pow
```

Nat.toText

```
func toText(n : Nat) : Text
```

The function `toText` takes one `Nat` argument and returns a `Text` value.

```
import Nat "mo:base/Nat";  
  
let natural : Nat = 10;  
  
Nat.toText(natural);
```

Nat.min

```
func min(x : Nat, y : Nat) : Nat
```

The function `min` takes two `Nat` arguments and returns the smallest `Nat` value.

```
import Nat "mo:base/Nat";  
  
let a : Nat = 50;  
let b : Nat = 20;  
  
Nat.min(a, b);
```

Nat.max

```
func max(x : Nat, y : Nat) : Nat
```

The function `max` takes two `Nat` arguments and returns the largest `Nat` value.

```
import Nat "mo:base/Nat";  
  
let a : Nat = 50;  
let b : Nat = 20;  
  
Nat.max(a, b);
```

Nat.compare

```
func compare(x : Nat, y : Nat) : {#less; #equal; #greater}
```

The function `compare` takes two `Nat` arguments and returns an `Order` variant value.

```
import Nat "mo:base/Nat";

let a : Nat = 50;
let b : Nat = 20;

Nat.compare(a, b);
```

Nat.equal

```
func equal(x : Nat, y : Nat) : Bool
```

The function `equal` takes two `Nat` arguments and returns a `Bool` value. It is equivalent to the `== relational operator`.

```
import Nat "mo:base/Nat";

let a : Nat = 50;
let b : Nat = 20;

let isEqual = Nat.equal(a, b);

let isEqualAgain = a == b;
```

Nat.notEqual

```
func notEqual(x : Nat, y : Nat) : Bool
```

The function `notEqual` takes two `Nat` arguments and returns a `Bool` value. It is equivalent to the `!= relational operator`

```
import Nat "mo:base/Nat";

let a : Nat = 50;
let b : Nat = 20;

let isNotEqual = Nat.notEqual(a, b);

let notEqual = a != b;
```

Nat.less

```
func less(x : Nat, y : Nat) : Bool
```

The function `less` takes two `Nat` arguments and returns a `Bool` value. It is equivalent to the [< relational operator](#).

```
import Nat "mo:base/Nat";  
  
let a : Nat = 50;  
let b : Nat = 20;  
  
let isLess = Nat.less(a, b);  
  
let less = a < b;
```

Nat.lessOrEqual

```
func lessOrEqual(x : Nat, y : Nat) : Bool
```

The function `lessOrEqual` takes two `Nat` arguments and returns a `Bool` value. It is equivalent to the [<= relational operator](#).

```
import Nat "mo:base/Nat";  
  
let a : Nat = 50;  
let b : Nat = 50;  
  
let isLessOrEqual = Nat.lessOrEqual(a, b);  
  
let lessOrEqual = a <= b;
```

Nat.greater

```
func greater(x : Nat, y : Nat) : Bool
```

The function `greater` takes two `Nat` arguments and returns a `Bool` value. It is equivalent to the [> relational operator](#).

```
import Nat "mo:base/Nat";

let a : Nat = 50;
let b : Nat = 20;

let isGreater = Nat.greater(a, b);

let greater = a > b;
```

Nat.greaterOrEqual

```
func greaterOrEqual(x : Nat, y : Nat) : Bool
```

The function `greaterOrEqual` takes two `Nat` arguments and returns a `Bool` value. It is equivalent to the `>=` relational operator.

```
import Nat "mo:base/Nat";

let a : Nat = 50;
let b : Nat = 20;

let isGreaterOrEqual = Nat.greaterOrEqual(a, b);

let greaterOrEqual = a >= b;
```

Nat.add

```
func add(x : Nat, y : Nat) : Nat
```

The function `add` takes two `Nat` arguments and returns a `Nat` value. It is equivalent to the `+` numeric operator.

```
import Nat "mo:base/Nat";

let a : Nat = 50;
let b : Nat = 20;

let add = Nat.add(a, b);

let addition = a + b;
```

Nat.sub

```
func sub(x : Nat, y : Nat) : Nat
```

The function `sub` takes two `Nat` arguments and returns a `Nat` value. It is equivalent to the `-` numeric operator.

```
import Nat "mo:base/Nat";

let a : Nat = 50;
let b : Nat = 20;

let subtract = Nat.sub(a, b);

// Result has type `Int` because subtracting two `Nat` may trap due to underflow
```

NOTE

Both the `Nat.sub` function and the `-` operator on `Nat` values may cause a `trap` if the result is a negative number (underflow).

Nat.mul

```
func mul(x : Nat, y : Nat) : Nat
```

The function `mul` takes two `Nat` arguments and returns a `Nat` value. It is equivalent to the `*` numeric operator.

```
import Nat "mo:base/Nat";

let a : Nat = 50;
let b : Nat = 20;

let multiply = Nat.mul(a, b);

let multiplication = a * b;
```

Nat.div

```
func div(x : Nat, y : Nat) : Nat
```

The function `div` takes two `Nat` arguments and returns a `Nat` value. It is equivalent to the `/` numeric operator.

```
import Nat "mo:base/Nat";  
  
let a : Nat = 50;  
let b : Nat = 50;  
  
let divide = Nat.div(a, b);
```

Nat.rem

```
func rem(x : Nat, y : Nat) : Nat
```

The function `rem` takes two `Nat` arguments and returns a `Nat` value. It is equivalent to the `%` numeric operator.

```
import Nat "mo:base/Nat";  
  
let a : Nat = 50;  
let b : Nat = 50;  
  
let remainder = Nat.rem(a, b);  
  
let remains = a % b;
```

Nat.pow

```
func pow(x : Nat, y : Nat) : Nat
```

The function `pow` takes two `Nat` arguments and returns a `Nat` value. It is equivalent to the `**` numeric operator.

```
import Nat "mo:base/Nat";  
  
let a : Nat = 5;  
let b : Nat = 5;  
  
let power = Nat.pow(a, b);  
  
let pow = a ** b;
```

Int

The *convention* is to name the *module alias* after the *file name* it is defined in:

```
import Int "mo:base/Int";
```

Numerical operations

```
Function abs
Function neg
Function add
Function sub
Function mul
Function div
Function rem
Function pow
```

Conversion

```
Function toText
```

Comparison

```
Function min
Function max
Function equal
Function notEqual
Function less
Function lessOrEqual
Function greater
Function greaterOrEqual
Function compare
```

Int.abs

```
func abs(x : Int) : Nat
```

The function `abs` takes one `Int` value as a argument and returns a `Nat` value.

```
import Int "mo:base/Int";  
  
let integer : Int = -10;  
  
Int.abs(integer);
```

Int.neg

```
func neg(x : Int) : Int
```

The function `neg` takes one `Int` value as a argument and returns an `Int` value.

```
import Int "mo:base/Int";  
  
let integer : Int = -10;  
  
Int.neg(integer);
```

Int.add

```
func add(x : Int, y : Int) : Int
```

The function `add` takes two `Int` value as a argument and returns an `Int` value.

```
import Int "mo:base/Int";  
  
let a : Int = 30;  
let b : Int = 20;  
  
Int.add(a, b);
```

Int.sub

```
func sub(x : Int, y : Int) : Int
```

The function `sub` takes two `Int` value as a argument and returns an `Int` value.

```
import Int "mo:base/Int";  
  
let a : Int = 30;  
let b : Int = 20;  
  
Int.sub(a, b);
```

Int.mul

```
func mul(x : Int, y : Int) : Int
```

The function `mul` takes two `Int` value as a argument and returns an `Int` value.

```
import Int "mo:base/Int";  
  
let a : Int = 3;  
let b : Int = 5;  
  
Int.mul(a, b);
```

Int.div

```
func div(x : Int, y : Int) : Int
```

The function `div` takes two `Int` value as a argument and returns an `Int` value.

```
import Int "mo:base/Int";  
  
let a : Int = 30;  
let b : Int = 10;  
  
Int.div(a, b);
```

Int.rem

```
func rem(x : Int, y : Int) : Int
```

The function `rem` takes two `Int` value as a argument and returns an `Int` value.

```
import Int "mo:base/Int";  
  
let a : Int = 30;  
let b : Int = 20;  
  
Int.rem(a, b);
```

Int.pow

```
func pow(x : Int, y : Int) : Int
```

The function `pow` takes two `Int` as a argument and returns an `Int` value.

```
import Int "mo:base/Int";  
  
let a : Int = 3;  
let b : Int = -3;  
  
Int.pow(a, b);
```

Int.toText

```
func toText(x : Int) : Text
```

The function `toText` takes one `Int` value as a argument and returns a `Text` value.

```
import Int "mo:base/Int";  
  
let integer : Int = -10;  
  
Int.toText(integer);
```

Int.min

```
func min(x : Int, y : Int) : Int
```

The function `min` takes two `Int` value as arguments and returns an `Int` value.

```
import Int "mo:base/Int";  
  
let a : Int = 20;  
let b : Int = -20;  
  
Int.min(a, b);
```

Int.max

```
func max(x : Int, y : Int) : Int
```

The function `max` takes two `Int` value as arguments and returns an `Int` value.

```
import Int "mo:base/Int";  
  
let a : Int = 20;  
let b : Int = -20;  
  
Int.max(a, b);
```

Int.equal

```
func equal(x : Int, y : Int) : Bool
```

The function `equal` takes two `Int` value as arguments and returns an `Bool` value.

```
import Int "mo:base/Int";  
  
let a : Int = 20;  
let b : Int = -20;  
  
Int.equal(a, b);
```

Int.notEqual

```
func notEqual(x : Int, y : Int) : Bool
```

The function `notEqual` takes two `Int` value as arguments and returns an `Bool` value.

```
import Int "mo:base/Int";  
  
let a : Int = 20;  
let b : Int = -20;  
  
Int.notEqual(a, b);
```

Int.less

```
func less(x : Int, y : Int) : Bool
```

The function `less` takes two `Int` value as arguments and returns an `Bool` value.

```
import Int "mo:base/Int";  
  
let a : Int = 20;  
let b : Int = -20;  
  
Int.less(a, b);
```

Int.lessOrEqual

```
func lessOrEqual(x : Int, y : Int) : Bool
```

The function `lessOrEqual` takes two `Int` value as arguments and returns an `Bool` value.

```
import Int "mo:base/Int";  
  
let a : Int = 20;  
let b : Int = -20;  
  
Int.lessOrEqual(a, b);
```

Int.greater

```
func greater(x : Int, y : Int) : Bool
```

The function `greater` takes two `Int` value as arguments and returns an `Bool` value.

```
import Int "mo:base/Int";

let a : Int = 20;
let b : Int = -20;

Int.greater(a, b);
```

Int.greaterOrEqual

```
func greaterOrEqual(x : Int, y : Int) : Bool
```

The function `greaterOrEqual` takes two `Int` value as arguments and returns an `Bool` value.

```
import Int "mo:base/Int";

let a : Int = 20;
let b : Int = -20;

Int.greaterOrEqual(a, b);
```

Int.compare

```
func compare(x : Int, y : Int) : {#less; #equal; #greater}
```

The function `compare` takes two `Int` value as arguments and returns an `Order` variant value.

```
import Int "mo:base/Int";

let a : Int = 20;
let b : Int = -20;

Int.compare(a, b);
```

Float

The *convention* is to name the *module alias* after the *file name* it is defined in:

```
import Float "mo:base/Float";
```

Utility Functions

```
Function abs
Function sqrt
Function ceil
Function floor
Function trunc
Function nearest
Function copySign
Function min
Function max
```

Conversion

```
Function toInt
Function fromInt
Function toText
Function toInt64
Function fromInt64
```

Comparison

```
Function equalWithin
Function notEqualWithin
Function less
Function lessOrEqual
Function greater
Function greaterOrEqual
Function compare
```

Numerical Operations

```
Function neg
Function add
Function sub
Function mul
Function div
Function rem
Function pow
```

Mathematical Operations

```
Function sin
Function cos
Function tan
Function arcsin
Function arccos
Function arctan
Function arctan2
Function exp
Function log
```

Float.abs

```
func abs : (x : Float) -> Float
```

The function `abs` takes one `Float` value and returns a `Float` value.

```
import Float "mo:base/Float";

let pi : Float = -3.14;

Float.abs(pi);
```

Float.sqrt

```
func sqrt : (x : Float) -> Float
```

The function `sqrt` takes one `Float` value and returns a `Float` value.

```
import Float "mo:base/Float";  
  
Float.sqrt(9);
```

Float.ceil

```
func ceil : (x : Float) -> Float
```

The function `ceil` takes one `Float` value and returns a `Float` value.

```
import Float "mo:base/Float";  
  
let pi : Float = 3.14;  
  
Float.ceil(pi);
```

Float.floor

```
func floor : (x : Float) -> Float
```

The function `floor` takes one `Float` value and returns a `Float` value.

```
import Float "mo:base/Float";  
  
let pi : Float = 3.14;  
  
Float.floor(pi);
```

Float.trunc

```
func trunc : (x : Float) -> Float
```

The function `trunc` takes one `Float` value and returns a `Float` value.

```
import Float "mo:base/Float";  
  
Float.trunc(3.98);
```

Float.nearest

```
func nearest : (x : Float) -> Float
```

The function `nearest` takes one `Float` value and returns a `Float` value.

```
import Float "mo:base/Float";  
  
Float.nearest(3.5);
```

Float.copySign

```
func copySign : (x : Float) -> Float
```

The function `copySign` takes one `Float` value and returns a `Float` value.

```
import Float "mo:base/Float";  
  
let a : Float = -3.64;  
let b : Float = 2.64;  
  
Float.copySign(a, b);
```

Float.min

```
let min : (x : Float, y : Float) -> Float
```

The function `min` takes two `Float` value and returns a `Float` value.

```
import Float "mo:base/Float";  
  
let a : Float = -3.64;  
let b : Float = 2.64;  
  
Float.min(a, b);
```

Float.max

```
let max : (x : Float, y : Float) -> Float
```

The function `max` takes two `Float` value and returns a `Float` value.

```
import Float "mo:base/Float";  
  
let a : Float = -3.64;  
let b : Float = 2.64;  
  
Float.max(a, b);
```

Float.toInt

```
func toInt : Float -> Int
```

The function `toInt` takes one `Float` value and returns an `Int` value.

```
import Float "mo:base/Float";  
  
let pi : Float = 3.14;  
  
Float.toInt(pi);
```

Float.fromInt

```
func fromInt : Int -> Float
```

The function `fromInt` takes one `Int` value and returns a `Float` value.

```
import Float "mo:base/Float";  
  
let x : Int = -15;  
  
Float.fromInt(x);
```

Float.toText

```
func toText : Float -> Text
```

The function `toText` takes one `Float` value and returns a `Text` value.

```
import Float "mo:base/Float";  
  
let pi : Float = 3.14;  
  
Float.toText(pi);
```

Float.toInt64

```
func toInt64 : Float -> Int64
```

The function `toInt64` takes one `Float` value and returns an `Int64` value.

```
import Float "mo:base/Float";  
  
let pi : Float = 3.14;  
  
Float.toInt64(pi);
```

Float.fromInt64

```
func fromInt64 : Int64 -> Float
```

The function `fromInt64` takes one `Int64` value and returns a `Float` value.

```
import Float "mo:base/Float";  
  
let y : Int64 = -64646464;  
  
Float.fromInt64(y);
```

Float.equalWithin

```
func equalWithin(x : Float, y : Float, epsilon : Float) : Bool
```

The function `equalWithin` takes two `Float` and one `epsilon` value and returns a `Bool` value.

```
import Float "mo:base/Float";  
  
let a : Float = 51.2;  
let b : Float = 5.12e1;  
let epsilon : Float = 1e-6;  
  
Float.equalWithin(a, b, epsilon)
```

Float.notEqualWithin

```
func notEqualWithin(x : Float, y : Float, epsilon : Float) : Bool
```

The function `notEqualWithin` takes two `Float` and one `epsilon` value and returns a `Bool` value.

```
import Float "mo:base/Float";

let a : Float = 51.2;
let b : Float = 5.12e1;
let epsilon : Float = 1e-6;

Float.notEqualWithin(a, b, epsilon)
```

Float.less

```
func less(x : Float, y : Float) : Bool
```

The function `less` takes two `Float` value and returns a `Bool` value.

```
import Float "mo:base/Float";

let a : Float = 3.14;
let b : Float = 3.24;

Float.less(a, b);
```

Float.lessOrEqual

```
func lessOrEqual(x : Float, y : Float) : Bool
```

The function `lessOrEqual` takes two `Float` value and returns a `Bool` value.

```
import Float "mo:base/Float";

let a : Float = 3.14;
let b : Float = 3.24;

Float.lessOrEqual(a, b);
```

Float.greater

```
func greater(x : Float, y : Float) : Bool
```

The function `greater` takes two `Float` value and returns a `Bool` value.

```
import Float "mo:base/Float";

let a : Float = 5.12;
let b : Float = 3.14;

Float.greater(a, b);
```

Float.greaterOrEqual

```
func greaterOrEqual(x : Float, y : Float) : Bool
```

The function `greaterOrEqual` takes two `Float` value and returns a `Bool` value.

```
import Float "mo:base/Float";

let a : Float = 5.12;
let b : Float = 3.14;

Float.greaterOrEqual(a, b);
```

Float.compare

```
func compare(x : Float, y : Float) : {#less; #equal; #greater}
```

The function `compare` takes two `Float` value and returns an `Order` value.

```
import Float "mo:base/Float";

let a : Float = 5.12;
let b : Float = 3.14;

Float.compare(a, b);
```

Float.neg

```
func neg(x : Float) : Float
```

The function `neg` takes one `Float` value and returns a `Float` value.

```
import Float "mo:base/Float";  
  
Float.neg(5.12);
```

Float.add

```
func add(x : Float, y : Float) : Float
```

The function `add` takes two `Float` value and returns a `Float` value.

```
import Float "mo:base/Float";  
  
let a : Float = 5.50;  
let b : Float = 3.50;  
  
Float.add(a, b);
```

Float.sub

```
func sub(x : Float, y : Float) : Float
```

The function `sub` takes two `Float` value and returns a `Float` value.

```
import Float "mo:base/Float";  
  
let a : Float = 5.12;  
let b : Float = 3.12;  
  
Float.sub(a, b);
```

Float.mul

```
func mul(x : Float, y : Float) : Float
```

The function `mul` takes two `Float` value and returns a `Float` value.

```
import Float "mo:base/Float";  
  
let a : Float = 5.25;  
let b : Float = 4.00;  
  
Float.mul(a, b);
```

Float.div

```
func div(x : Float, y : Float) : Float
```

The function `div` takes two `Float` value and returns a `Float` value.

```
import Float "mo:base/Float";  
  
let a : Float = 6.12;  
let b : Float = 3.06;  
  
Float.div(a, b);
```

Float.rem

```
func rem(x : Float, y : Float) : Float
```

The function `rem` takes two `Float` value and returns a `Float` value.

```
import Float "mo:base/Float";  
  
let a : Float = 5.12;  
let b : Float = 3.12;  
  
Float.rem(a, b);
```

Float.pow

```
func pow(x : Float, y : Float) : Float
```

The function `pow` takes two `Float` value and returns a `Float` value.

```
import Float "mo:base/Float";  
  
let a : Float = 1.5;  
let b : Float = 2.0;  
  
Float.pow(a, b);
```

Float.sin

```
func sin : (x : Float) -> Float
```

The function `sin` takes one `Float` value and returns a `Float` value.

```
import Float "mo:base/Float";  
  
Float.sin(Float.pi / 2);
```

Float.cos

```
func cos : (x : Float) -> Float
```

The function `cos` takes one `Float` value and returns a `Float` value.

```
import Float "mo:base/Float";  
  
Float.cos(0);
```

Float.tan

```
func tan : (x : Float) -> Float
```

The function `tan` takes one `Float` value and returns a `Float` value.

```
import Float "mo:base/Float";  
  
Float.tan(Float.pi / 4);
```

Float.arcsin

```
func arcsin : (x : Float) -> Float
```

The function `arcsin` takes one `Float` value and returns a `Float` value. for more explanation look for [official documentation](#)

```
import Float "mo:base/Float";  
  
Float.arcsin(1.0)
```

Float.acos

```
func arccos : (x : Float) -> Float
```

The function `arccos` takes one `Float` value and returns a `Float` value. for more explanation look for [official documentation](#)

```
import Float "mo:base/Float";  
  
Float.acos(1.0)
```

Float.arctan

```
func arctan : (x : Float) -> Float
```

The function `arctan` takes one `Float` value and returns a `Float` value. for more explanation look for [official documentation](#)

```
import Float "mo:base/Float";  
  
Float.arctan(1.0)
```

Float.arctan2

```
func arctan2 : (y : Float, x : Float) -> Float
```

The function `arctan2` takes two `Float` value and returns a `Float` value. for more explanation look for [official documentation](#)

```
import Float "mo:base/Float";  
  
Float.arctan2(4, -3)
```

Float.exp

```
func exp : (x : Float) -> Float
```

The function `exp` takes one `Float` value and returns a `Float` value.

```
import Float "mo:base/Float";  
  
Float.exp(1.0)
```

Float.log

```
func log : (x : Float) -> Float
```

The function `log` takes one `Float` value and returns a `Float` value.

```
import Float "mo:base/Float";  
  
let exponential : Float = Float.e;  
  
Float.log(exponential)
```

Principal

To understand this *principal module*, it might be helpful to learn about [Principles](#) first.

The *convention* is to name the *module alias* after the *file name* it is defined in:

```
import Principal "mo:base/Principal";
```

Conversion

```
Function fromActor  
Function fromText  
Function toText  
Function toBlob  
Function fromBlob
```

Utility

```
Function isAnonymous  
Function hash
```

Comparison

```
Function equal  
Function notEqual  
Function less  
Function lessOrEqual  
Function greater  
Function greaterOrEqual  
Function compare
```

Principal.fromActor

```
func fromActor(a : actor { }) : Principal
```

The function `fromActor` takes one `actor` value and returns a `Principal` value.

```
import Principal "mo:base/Principal";

type Account = { owner : Principal; subaccount : ?Blob };

type Ledger = actor {
    icrc1_balance_of : shared query Account -> async Nat;
};

let ledger = actor ("ryjl3-tyaaa-aaaaa-aaaba-cai") : Ledger;

Principal.fromActor(ledger);
```

Principal.fromText

```
func fromText(t : Text) : Principal
```

The function `fromText` takes one `Text` value and returns a `Principal` value.

```
import Principal "mo:base/Principal";

let textualPrincipal : Text = "un4fu-tqaaa-aaaab-qadjq-cai";

Principal.fromText(textualPrincipal);
```

Principal.toText

```
func toText(p : Principal) : Text
```

The function `toText` takes one `Principal` value and returns a `Text` value.

```
import Principal "mo:base/Principal";

let principal : Principal = Principal.fromText("un4fu-tqaaa-aaaab-qadjq-cai");

Principal.toText(principal);
```

Principal.toBlob

```
func toBlob(p : Principal) : Blob
```

The function `toBlob` takes one `Principal` value and returns a `Blob` value.

```
import Principal "mo:base/Principal";  
  
let principal : Principal = Principal.fromText("un4fu-tqaaa-aaaab-qadjq-cai");  
  
Principal.toBlob(principal);
```

Principal.fromBlob

```
func fromBlob(b : Blob) : Principal
```

The function `fromBlob` takes one `Blob` value and returns a `Principal` value.

```
import Principal "mo:base/Principal";  
  
let principal : Principal = Principal.fromText("un4fu-tqaaa-aaaab-qadjq-cai");  
let blob : Blob = Principal.toBlob(principal);  
  
Principal.fromBlob(blob);
```

Principal.isAnonymous

```
func isAnonymous(p : Principal) : Bool
```

The function `isAnonymous` takes one `Principal` value and returns a `Bool` value.

```
import Principal "mo:base/Principal";  
  
let principal : Principal = Principal.fromText("un4fu-tqaaa-aaaab-qadjq-cai");  
  
Principal.isAnonymous(principal);
```

Principal.hash

```
func hash(principal : Principal) : Hash.Hash
```

The function `hash` takes one `Principal` value and returns a `Hash` value.

```
import Principal "mo:base/Principal";  
  
let principal = Principal.fromText("m7sm4-2iaaa-aaaab-qabra-cai");  
  
Principal.hash(principal);
```

Principal.equal

```
func equal(principal1 : Principal, principal2 : Principal) : Bool
```

The function `equal` takes two `Principal` value and returns a `Bool` value.

```
import Principal "mo:base/Principal";  
  
let principal1 = Principal.fromText("m7sm4-2iaaa-aaaab-qabra-cai");  
let principal2 = Principal.fromText("m7sm4-2iaaa-aaaab-qabra-cai");  
  
Principal.equal(principal1, principal2);
```

Principal.notEqual

```
func notEqual(principal1 : Principal, principal2 : Principal) : Bool
```

The function `notEqual` takes two `Principal` value and returns a `Bool` value.

```
import Principal "mo:base/Principal";  
  
let principal1 = Principal.fromText("m7sm4-2iaaa-aaaab-qabra-cai");  
let principal2 = Principal.fromText("m7sm4-2iaaa-aaaab-qabra-cai");  
  
Principal.notEqual(principal1, principal2);
```

Principal.less

```
func less(principal1 : Principal, principal2 : Principal) : Bool
```

The function `less` takes two `Principal` value and returns a `Bool` value.

```
import Principal "mo:base/Principal";

let principal1 = Principal.fromText("m7sm4-2iaaa-aaaab-qabra-cai");
let principal2 = Principal.fromText("jr7pa-rnspg-drbf2-ooaap-gmfep-773oe-z5oua-
hiycn-q3sjg-ffwie-tae");

Principal.less(principal1, principal2);
```

Principal.lessOrEqual

```
func lessOrEqual(principal1 : Principal, principal2 : Principal) : Bool
```

The function `lessOrEqual` takes two `Principal` value and returns a `Bool` value.

```
import Principal "mo:base/Principal";

let principal1 = Principal.fromText("m7sm4-2iaaa-aaaab-qabra-cai");
let principal2 = Principal.fromText("m7sm4-2iaaa-aaaab-qabra-cai");

Principal.lessOrEqual(principal1, principal2);
```

Principal.greater

```
func greater(principal1 : Principal, principal2 : Principal) : Bool
```

The function `greater` takes two `Principal` value and returns a `Bool` value.

```
import Principal "mo:base/Principal";

let principal1 = Principal.fromText("m7sm4-2iaaa-aaaab-qabra-cai");
let principal2 = Principal.fromText("m7sm4-2iaaa-aaaab-qabra-cai");

Principal.greater(principal1, principal2);
```

Principal.greaterOrEqual

```
func greaterOrEqual(principal1 : Principal, principal2 : Principal) : Bool
```

The function `greaterOrEqual` takes two `Principal` value and returns a `Bool` value.

```
import Principal "mo:base/Principal";

let principal1 = Principal.fromText("2vxssx-fae");
let principal2 = Principal.fromText("m7sm4-2iaaa-aaaab-qabra-cai");

Principal.greaterOrEqual(principal1, principal2);
```

Principal.compare

```
func compare(principal1 : Principal, principal2 : Principal) : {#less; #equal;
#greater}
```

The function `compare` takes two `Principal` value and returns an `Order` value.

```
import Principal "mo:base/Principal";

let principal1 = Principal.fromText("m7sm4-2iaaa-aaaab-qabra-cai");
let principal2 = Principal.fromText("m7sm4-2iaaa-aaaab-qabra-cai");

Principal.compare(principal1, principal2);
```

Text

The *convention* is to name the *module alias* after the *file name* it is defined in:

```
import Text "mo:base/Text";
```

Types

Type Pattern

Analysis

```
Function size
Function contains
Function startsWith
Function endsWith
Function stripStart
Function stripEnd
Function trimStart
Function trimEnd
Function trim
```

Conversion

```
Function fromChar
Function toIter
Function fromIter
Function hash
Function encodeUtf8
Function decodeUtf8
```

Comparison

```
Function equal
Function notEqual
```

```
Function less
Function lessOrEqual
Function greater
Function greaterOrEqual
Function compare
```

Transformation

```
Function replace
Function concat
Function join
Function map
Function translate
Function split
Function tokens
Function compareWith
```

Type Pattern

The `Pattern` type is a useful `variant` searching through `Text` values. If we traverse (visit each character of) a `Text`, we might look for a *text pattern* that matches a `Pattern`.

This could be a *character* in case of `#char : Char` or a *string* in case of a `#text : Text`. We could also provide a function of type `Char -> Bool` that looks at each character and performs some logic to test whether some *predicate* is true or false about that character.

```
type Pattern = {
    #char : Char;
    #text : Text;
    #predicate : (Char -> Bool);
};
```

Text.size

```
func size(t : Text) : Nat
```

The function `size` takes one `Text` value as a argument and returns a `Nat` value.

```
import Text "mo:base/Text";  
  
let text : Text = "blockchain";  
  
Text.size(text)
```

Text.contains

```
func contains(t : Text, p : Pattern) : Bool  
  
import Text "mo:base/Text";  
  
let text : Text = "blockchain";  
let letter : Text.Pattern = #char 'k';  
  
Text.contains(text, letter);
```

Text.startsWith

```
func startsWith(t : Text, p : Pattern) : Bool  
  
import Text "mo:base/Text";  
  
let text : Text = "blockchain";  
let letter : Text.Pattern = #text "block";  
  
Text.startsWith(text, letter);
```

Text.endsWith

```
func endsWith(t : Text, p : Pattern) : Bool
```

```
import Text "mo:base/Text";
import Char "mo:base/Char";

let text : Text = "blockchain";
let letter : Text.Pattern = #predicate(func c = Char.isAlphabetic(c));

Text.endsWith(text, letter);
```

Text.stripStart

```
func stripStart(t : Text, p : Pattern) : ?Text

import Text "mo:base/Text";

let text : Text = "@blockchain";
let letter : Text.Pattern = #char '@';

Text.stripStart(text, letter);
```

Text.stripEnd

```
func stripEnd(t : Text, p : Pattern) : ?Text

import Text "mo:base/Text";

let text : Text = "blockchain&";
let letter : Text.Pattern = #char '&';

Text.stripEnd(text, letter);
```

Text.trimStart

```
func trimStart(t : Text, p : Pattern) : Text
```

```
import Text "mo:base/Text";

let text : Text = "vvvblockchain";
let letter : Text.Pattern = #char 'v';

Text.trimStart(text, letter);
```

Text.trimEnd

```
func trimEnd(t : Text, p : Pattern) : Text

import Text "mo:base/Text";

let text : Text = "blockchain****";
let letter : Text.Pattern = #char '*';

Text.trimEnd(text, letter);
```

Text.trim

```
func trim(t : Text, p : Pattern) : Text

import Text "mo:base/Text";

let text : Text = "@@blockchain@@";
let letter : Text.Pattern = #char '@';

Text.trim(text, letter);
```

Text.fromChar

```
func fromChar : (c : Char) -> Text
```

The function `fromChar` takes one `Text` value as a argument and returns a `Char` value.

```
import Text "mo:base/Text";

let character : Char = 'c';

Text.fromChar(character)
```

Text.tolter

```
func toIter(t : Text) : Iter.Iter<Char>

import Text "mo:base/Text";
import Iter "mo:base/Iter";

let text : Text = "xyz";

let iter : Iter.Iter<Char> = Text.toIter(text);

Iter.toArray(iter)
```

Text.fromIter

```
func fromIter(cs : Iter.Iter<Char>) : Text

import Text "mo:base/Text";
import Array "mo:base/Array";
import Iter "mo:base/Iter";

let array : [Char] = ['I', 'C', 'P'];

let iter : Iter.Iter<Char> = Array.vals(array);

Text.fromIter(iter)
```

Text.hash

```
func hash(t : Text) : Hash.Hash
```

The function `hash` takes one `Text` value as a argument and returns a `Hash` value.

```
import Text "mo:base/Text";

let text : Text = "xyz";

Text.hash(text)
```

Text.encodeUtf8

```
func encodeUtf8 : Text -> Blob
```

The function `encodeUtf8` takes one `Text` value as a argument and returns a `Bool` value.

```
import Blob "mo:base/Blob";
import Text "mo:base/Text";

let t : Text = "ICP";

Text.encodeUtf8(t);
```

Text.decodeUtf8

```
func decodeUtf8 : Blob -> ?Text
```

The function `decodeUtf8` takes one `Blob` value as a argument and returns a `?Text` value. If the blob is not valid UTF8, then this function returns `null`.

```
import Blob "mo:base/Blob";
import Text "mo:base/Text";

let array : [Nat8] = [73, 67, 80];
let blob : Blob = Blob.fromArray(array);

Text.decodeUtf8(blob);
```

Text.equal

```
func equal(t1 : Text, t2 : Text) : Bool
```

The function `equal` takes two `Text` value as a argument and returns a `Bool` value.

```
import Text "mo:base/Text";

let a : Text = "text_A";
let b : Text = "text_A";

Text.equal(a, b);
```

Text.notEqual

```
func notEqual(t1 : Text, t2 : Text) : Bool
```

The function `notEqual` takes two `Text` value as a argument and returns a `Bool` value.

```
import Text "mo:base/Text";

let a : Text = "text_B";
let b : Text = "text_A";

Text.notEqual(a, b);
```

Text.less

```
func less(t1 : Text, t2 : Text) : Bool
```

The function `less` takes two `Text` value as a argument and returns a `Bool` value.

```
import Text "mo:base/Text";

let a : Text = "text_A";
let b : Text = "text_B";

Text.less(a, b);
```

Text.lessOrEqual

```
func lessOrEqual(t1 : Text, t2 : Text) : Bool
```

The function `lessOrEqual` takes two `Text` value as a argument and returns a `Bool` value.

```
import Text "mo:base/Text";  
  
let a : Text = "text_A";  
let b : Text = "text_B";  
  
Text.lessOrEqual(a, b);
```

Text.greater

```
func greater(t1 : Text, t2 : Text) : Bool
```

The function `greater` takes two `Text` value as a argument and returns a `Bool` value.

```
import Text "mo:base/Text";  
  
let a : Text = "text_B";  
let b : Text = "text_A";  
  
Text.greater(a, b);
```

Text.greaterOrEqual

```
func greaterOrEqual(t1 : Text, t2 : Text) : Bool
```

The function `greaterOrEqual` takes two `Text` value as a argument and returns a `Bool` value.

```
import Text "mo:base/Text";  
  
let a : Text = "text_B";  
let b : Text = "text_B";  
  
Text.greaterOrEqual(a, b);
```

Text.compare

```
func compare(t1 : Text, t2 : Text) : {#less; #equal; #greater}
```

The function `compare` takes two `Text` value as a argument and returns an `Order` value.

```
import Text "mo:base/Text";

let a : Text = "text_A";
let b : Text = "text_B";

Text.compare(a, b);
```

Text.replace

```
func replace(t : Text, p : Pattern, r : Text) : Text
```

Parameters	
Variable argument1	t : Text
Variable argument2	r : Text
Object argument	p : pattern
Return type	Iter.Iter<Text>

```
import Text "mo:base/Text";

let text : Text = "blockchain";
let letter : Text.Pattern = #char 'b';

Text.replace(text, letter, "c");
```

Text.concat

```
func concat(t1 : Text, t2 : Text) : Text
```

The function `concat` takes two `Text` value as a arguments and returns a `Text` value. It is equivalent to the `#` operator.

```
import Text "mo:base/Text";

let t1 : Text = "Internet";
let t2 : Text = "Computer";

Text.concat(t1, t2);
```

Text.join

```
func join(sep : Text, ts : Iter.Iter<Text>) : Text
```

Parameters	
Variable argument	sep : Text
Object argument	ts : Iter.Iter<Text>
Return type	Text

```
import Text "mo:base/Text";
import Array "mo:base/Array";
import Iter "mo:base/Iter";

let array : [Text] = ["Internet", "Computer", "Protocol"];

let iter : Iter.Iter<Text> = Array.vals(array);

let text : Text = "-";

Text.join(text, iter)
```

Text.map

```
func map(t : Text, f : Char -> Char) : Text
```

Parameters	
Variable argument	t : Text
Function argument	f : Char -> Char
Return type	Text

```
import Text "mo:base/Text";

let text : Text = "PCI";

func change(c : Char) : Char {
    if (c == 'P') { return 'I' } else if (c == 'I') { return 'P' } else {
        return 'C';
    };
};

Text.map(text, change)
```

Text.translate

```
func translate(t : Text, f : Char -> Text) : Text
```

Parameters	
Variable argument	t : Text
Function argument	f : Char -> Char
Return type	Text

```
import Text "mo:base/Text";

let text : Text = "*ICP*";

func change(c : Char) : Text {
    if (c == '*') { return "!" } else { return Text.fromChar(c) };
};

Text.translate(text, change)
```

Text.split

```
func split(t : Text, p : Pattern) : Iter.Iter<Text>
```

Parameters	
Variable argument	t : Text
Object argument	p : pattern

Parameters	
Return type	Iter.Iter<Text>

```
import Text "mo:base/Text";
import Iter "mo:base/Iter";

let text : Text = "this is internet computer";

let letter : Text.Pattern = #char ' ';

let split : Iter.Iter<Text> = Text.split(text, letter);

Iter.toArray(split)
```

Text.tokens

```
func tokens(t : Text, p : Pattern) : Iter.Iter<Text>
```

Parameters	
Variable argument	t : Text
Object argument	p : pattern
Return type	Iter.Iter<Text>

```
import Text "mo:base/Text";
import Iter "mo:base/Iter";

let text : Text = "abcabcabc";

let letter : Text.Pattern = #char 'a';

let token : Iter.Iter<Text> = Text.tokens(text, letter);

Iter.toArray(token)
```

Text.compareWith

```
func compareWith(t1 : Text, t2 : Text, cmp : (Char, Char) -> {#less; #equal;
#greater}) : {#less; #equal; #greater}
```

Parameters	
Variable argument1	t1 : Text
Variable argument2	t2 : Text
Function argument	cmp : (Char, Char) -> {#less; #equal; #greater}
Return type	{#less; #equal; #greater}

```

import Text "mo:base/Text";
import Order "mo:base/Order";
import Char "mo:base/Char";

let text1 : Text = "icp";
let text2 : Text = "ICP";

func compare(c1 : Char, c2 : Char) : Order.Order {
    Char.compare(c1, c2);
};

Text.compareWith(text1, text2, compare);

```

Char

In Motoko, a *character literal* is a single character enclosed in **single quotes** and has type **Char**. (As opposed to *text literals* of type **Text**, which may be multiple characters enclosed in **double quotes**.)

```
let char : Char = 'a';  
let text : Text = "a";
```

The *convention* is to name the *module alias* after the *file name* it is defined in:

```
import Char "mo:base/Char";
```

Conversion

```
Function toNat32  
Function fromNat32  
Function toText
```

Utility Function

```
Function isDigit  
Function isWhitespace  
Function isLowercase  
Function isUppercase  
Function isAlphabetic
```

Comparison

```
Function equal  
Function notEqual  
Function less  
Function lessOrEqual  
Function greater  
Function greaterOrEqual  
Function compare
```

Char.toNat32

```
func toNat32 : (c : Char) -> Nat32
```

The function `toNat32` takes one `Char` value and returns a `Nat32` value.

```
import Char "mo:base/Char";  
  
Char.toNat32('y');
```

Char.fromNat32

```
func fromNat32 : (w : Nat32) -> Char
```

The function `fromNat32` takes one `Nat32` value and returns a `Char` value.

```
import Char "mo:base/Char";  
  
Char.fromNat32(100 : Nat32);
```

Char.toText

```
func toText : (c : Char) -> Text
```

The function `toText` takes one `Char` value and returns a `Text` value.

```
import Char "mo:base/Char";  
  
Char.toText('C')
```

Char.isDigit

```
func isDigit(c : Char) : Bool
```

The function `isDigit` takes one `Char` value and returns a `Bool` value.

```
import Char "mo:base/Char";  
  
Char.isDigit('5');
```

Char.isWhitespace

```
let isWhitespace : (c : Char) -> Bool
```

The function `isWhitespace` takes one `Char` value and returns a `Bool` value.

```
import Char "mo:base/Char";  
  
Char.isWhitespace(' ');
```

Char.isLowercase

```
func isLowercase(c : Char) : Bool
```

The function `isLowercase` takes one `Char` value and returns a `Bool` value.

```
import Char "mo:base/Char";  
  
Char.isLowercase('a');
```

Char.isUppercase

```
func isUppercase(c : Char) : Bool
```

The function `isUppercase` takes one `Char` value and returns a `Bool` value.

```
import Char "mo:base/Char";  
  
Char.isUppercase('A');
```

Char.isAlphabetic

```
func isAlphabetic : (c : Char) -> Bool
```

The function `isAlphabetic` takes one `Char` value and returns a `Bool` value.

```
import Char "mo:base/Char";  
  
Char.isAlphabetic('y');
```

Char.equal

```
func equal(x : Char, y : Char) : Bool
```

The function `equal` takes two `Char` value and returns a `Bool` value.

```
import Char "mo:base/Char";  
  
let char1 : Char = 'a';  
let char2 : Char = 'b';  
  
Char.equal(char1, char2);
```

Char.notEqual

```
func notEqual(x : Char, y : Char) : Bool
```

The function `notEqual` takes two `Char` value and returns a `Bool` value.

```
import Char "mo:base/Char";  
  
let char1 : Char = 'a';  
let char2 : Char = 'b';  
  
Char.notEqual(char1, char2);
```

Char.less

```
func less(x : Char, y : Char) : Bool
```

The function `less` takes two `Char` value and returns a `Bool` value.

```
import Char "mo:base/Char";  
  
let char1 : Char = 'a';  
let char2 : Char = 'b';  
  
Char.less(char1, char2);
```

Char.lessOrEqual

```
func lessOrEqual(x : Char, y : Char) : Bool
```

The function `lessOrEqual` takes two `Char` value and returns a `Bool` value.

```
import Char "mo:base/Char";  
  
let char1 : Char = 'a';  
let char2 : Char = 'b';  
  
Char.lessOrEqual(char1, char2);
```

Char.greater

```
func greater(x : Char, y : Char) : Bool
```

The function `greater` takes two `Char` value and returns a `Bool` value.

```
import Char "mo:base/Char";  
  
let char1 : Char = 'b';  
let char2 : Char = 'a';  
  
Char.greater(char1, char2);
```

Char.greaterOrEqual

```
func greaterOrEqual(x : Char, y : Char) : Bool
```

The function `greaterOrEqual` takes two `Char` value and returns a `Bool` value.

```
import Char "mo:base/Char";  
  
let char1 : Char = 'b';  
let char2 : Char = 'a';  
  
Char.greaterOrEqual(char1, char2);
```

Char.compare

```
func compare(x : Char, y : Char) : {#less; #equal; #greater}
```

The function `compare` takes two `Char` value and returns an `Order` value.

```
import Char "mo:base/Char";  
  
let char1 : Char = 'a';  
let char2 : Char = 'b';  
  
Char.compare(char1, char2);
```

Bounded Number Types

Unlike `Nat` and `Int`, which are *unbounded* numbers, the *bounded number types* have a specified *bit length*. Operations that *overflow* (reach numbers beyond the *minimum* or *maximum* value defined by the bit length) on these bounded number types cause a `trap`.

The bounded *natural* number types `Nat8`, `Nat16`, `Nat32`, `Nat64` and the bounded *integer* number types `Int8`, `Int16`, `Int32`, `Int64` are all *primitive types* in Motoko and don't need to be imported.

This section covers *modules* with useful functionality for these types.

Nat8

The *convention* is to name the *module alias* after the *file name* it is defined in:

```
import Nat8 "mo:base/Nat8";
```

On this page

Constants

```
Value minimumValue  
Value maximumValue
```

Conversion

```
Function toNat  
Function toText  
Function fromNat  
Function fromIntWrap
```

Comparison

```
Function min  
Function max  
Function equal  
Function notEqual  
Function less  
Function lessOrEqual  
Function greater  
Function greaterOrEqual  
Function compare
```

Numerical Operations

```
Function add
Function sub
Function mul
Function div
Function rem
Function pow
```

Bitwise Operators

```
Function bitnot
Function bitand
Function bitor
Function bitxor
Function bitshiftLeft
Function bitshiftRight
Function bitrotLeft
Function bitrotRight
Function bittest
Function bitset
Function bitclear
Function bitflip
Function bitcountNonZero
Function bitcountLeadingZero
Function bitcountTrailingZero
```

Wrapping Operations

```
Function addWrap
Function subWrap
Function mulWrap
Function powWrap
```

minimumValue

```
let minValue : Nat8 = 0
```

maximumValue

```
let maximumValue : Nat8 = 255
```

Nat8.toNat

```
func toNat(i : Nat8) : Nat
```

The function `toNat` takes one `Nat8` value and returns a `Nat` value.

```
import Nat8 "mo:base/Nat8";  
  
let a : Nat8 = 255;  
  
Nat8.toNat(a);
```

Nat8.toText

```
func toText(i : Nat8) : Text
```

The function `toText` takes one `Nat8` value and returns a `Text` value.

```
import Nat8 "mo:base/Nat8";  
  
let b : Nat8 = 255;  
  
Nat8.toText(b);
```

Nat8.fromNat

```
func fromNat(i : Nat) : Nat8
```

The function `fromNat` takes one `Nat` value and returns a `Nat8` value.

```
import Nat8 "mo:base/Nat8";  
  
let c : Nat = 255;  
  
Nat8.fromNat(c);
```

Nat8.fromIntWrap

```
func fromIntWrap(i : Int) : Nat8
```

The function `fromIntWrap` takes one `Int` value and returns a `Nat8` value.

```
import Nat8 "mo:base/Nat8";  
  
let integer : Int = 356;  
  
Nat8.fromIntWrap(integer);
```

Nat8.min

```
func min(x : Nat8, y : Nat8) : Nat8
```

The function `min` takes two `Nat8` value and returns a `Nat8` value.

```
import Nat8 "mo:base/Nat8";  
  
let x : Nat8 = 15;  
let y : Nat8 = 10;  
  
Nat8.min(x, y);
```

Nat8.max

```
func max(x : Nat8, y : Nat8) : Nat8
```

The function `max` takes two `Nat8` value and returns a `Nat8` value.

```
import Nat8 "mo:base/Nat8";

let x : Nat8 = 15;
let y : Nat8 = 10;

Nat8.max(x, y);
```

Nat8.equal

```
func equal(x : Nat8, y : Nat8) : Bool
```

The function `equal` takes two `Nat8` value and returns a `Bool` value.

```
import Nat8 "mo:base/Nat8";

let x : Nat8 = 10;
let y : Nat8 = 10;

Nat8.equal(x, y);
```

Nat8.notEqual

```
func notEqual(x : Nat8, y : Nat8) : Bool
```

The function `notEqual` takes two `Nat8` value and returns a `Bool` value.

```
import Nat8 "mo:base/Nat8";

let x : Nat8 = 15;
let y : Nat8 = 10;

Nat8.notEqual(x, y);
```

Nat8.less

```
func less(x : Nat8, y : Nat8) : Bool
```

The function `less` takes two `Nat8` value and returns a `Bool` value.

```
import Nat8 "mo:base/Nat8";

let x : Nat8 = 15;
let y : Nat8 = 25;

Nat8.less(x, y);
```

Nat8.lessOrEqual

```
func lessOrEqual(x : Nat8, y : Nat8) : Bool
```

The function `lessOrEqual` takes two `Nat8` value and returns a `Bool` value.

```
import Nat8 "mo:base/Nat8";

let x : Nat8 = 15;
let y : Nat8 = 25;

Nat8.lessOrEqual(x, y);
```

Nat8.greater

```
func greater(x : Nat8, y : Nat8) : Bool
```

The function `greater` takes two `Nat8` value and returns a `Bool` value.

```
import Nat8 "mo:base/Nat8";

let x : Nat8 = 15;
let y : Nat8 = 10;

Nat8.greater(x, y);
```

Nat8.greaterOrEqual

```
func greaterOrEqual(x : Nat8, y : Nat8) : Bool
```

The function `greaterOrEqual` takes two `Nat8` value and returns a `Bool` value.

```
import Nat8 "mo:base/Nat8";

let x : Nat8 = 15;
let y : Nat8 = 10;

Nat8.greaterOrEqual(x, y);
```

Nat8.compare

```
func compare(x : Nat8, y : Nat8) : Bool
```

The function `compare` takes two `Nat8` value and returns an `Order` variant value.

```
import Nat8 "mo:base/Nat8";

let x : Nat8 = 10;
let y : Nat8 = 10;

Nat8.compare(x, y);
```

Nat8.add

```
func add(x : Nat8, y : Nat8) : Nat8
```

The function `add` takes two `Nat8` value and returns a `Nat8` value.

```
import Nat8 "mo:base/Nat8";

let x : Nat8 = 40;
let y : Nat8 = 10;

Nat8.add(x, y);
```

Nat8.sub

```
func sub(x : Nat8, y : Nat8) : Nat8
```

The function `sub` takes two `Nat8` value and returns a `Nat8` value.

```
import Nat8 "mo:base/Nat8";  
  
let x : Nat8 = 40;  
let y : Nat8 = 20;  
  
Nat8.sub(x, y);
```

Nat8.mul

```
func mul(x : Nat8, y : Nat8) : Nat8
```

The function `mul` takes two `Nat8` value and returns a `Nat8` value.

```
import Nat8 "mo:base/Nat8";  
  
let x : Nat8 = 20;  
let y : Nat8 = 10;  
  
Nat8.mul(x, y);
```

Nat8.div

```
func div(x : Nat8, y : Nat8) : Nat8
```

The function `div` takes two `Nat8` value and returns a `Nat8` value.

```
import Nat8 "mo:base/Nat8";  
  
let x : Nat8 = 50;  
let y : Nat8 = 10;  
  
Nat8.div(x, y);
```

Nat8.rem

```
func rem(x : Nat8, y : Nat8) : Nat8
```

The function `rem` takes two `Nat8` value and returns a `Nat8` value.

```
import Nat8 "mo:base/Nat8";

let x : Nat8 = 40;
let y : Nat8 = 15;

Nat8.rem(x, y);
```

Nat8.pow

```
func pow(x : Nat8, y : Nat8) : Nat8
```

The function `pow` takes two `Nat8` value and returns a `Nat8` value.

```
import Nat8 "mo:base/Nat8";

let x : Nat8 = 6;
let y : Nat8 = 3;

Nat8.pow(x, y);
```

Nat8.bitnot

```
func bitnot(x : Nat8) : Nat8
```

The function `bitnot` takes one `Nat8` value and returns a `Nat8` value.

```
import Nat8 "mo:base/Nat8";

let x : Nat8 = 254; // Binary : 11111110

Nat8.bitnot(x) // Binary : 00000001
```

Nat8.bitand

```
func bitand(x : Nat8, y : Nat8) : Nat8
```

The function `bitand` takes two `Nat8` value and returns a `Nat8` value.

```
import Nat8 "mo:base/Nat8";

let x : Nat8 = 255; // Binary : 11111111
let y : Nat8 = 15; // Binary : 00001111

Nat8.bitand(x, y) // Binary : 00001111
```

Nat8.bitor

```
func bitor(x : Nat8, y : Nat8) : Nat8
```

The function `bitor` takes two `Nat8` value and returns a `Nat8` value.

```
import Nat8 "mo:base/Nat8";

let x : Nat8 = 240; // Binary : 11110000
let y : Nat8 = 15; // Binary : 00001111

Nat8.bitor(x, y) // Binary : 11111111
```

Nat8.bitxor

```
func bitxor(x : Nat8, y : Nat8) : Nat8
```

The function `bitxor` takes two `Nat8` value and returns a `Nat8` value.

```
import Nat8 "mo:base/Nat8";

let x : Nat8 = 15; // Binary : 00001111
let y : Nat8 = 3; // Binary : 00000011

Nat8.bitxor(x, y) // Binary : 00001100
```

Nat8.bitshiftLeft

```
func bitshiftLeft(x : Nat8, y : Nat8) : Nat8
```

The function `bitshiftLeft` takes two `Nat8` value and returns a `Nat8` value.

```
import Nat8 "mo:base/Nat8";

let x : Nat8 = 48; // Binary : 00110000
let y : Nat8 = 2;

Nat8.bitshiftLeft(x, y) // Binary : 11000000
```

Nat8.bitshiftRight

```
func bitshiftRight(x : Nat8, y : Nat8) : Nat8
```

The function `bitshiftRight` takes two `Nat8` value and returns a `Nat8` value.

```
import Nat8 "mo:base/Nat8";

let x : Nat8 = 12; // Binary : 00001100
let y : Nat8 = 2;

Nat8.bitshiftRight(x, y) // Binary : 00000011
```

Nat8.bitrotLeft

```
func bitrotLeft(x : Nat8, y : Nat8) : Nat8
```

The function `bitrotLeft` takes two `Nat8` value and returns a `Nat8` value.

```
import Nat8 "mo:base/Nat8";

let x : Nat8 = 240; // Binary : 11110000
let y : Nat8 = 2;

Nat8.bitrotLeft(x, y) // Binary : 11000011
```

Nat8.bitrotRight

```
func bitrotRight(x : Nat8, y : Nat8) : Nat8
```

The function `bitrotRight` takes two `Nat8` value and returns a `Nat8` value.

```
import Nat8 "mo:base/Nat8";

let x : Nat8 = 15; // Binary : 00001111
let y : Nat8 = 2;

Nat8.bitrotRight(x, y) // Binary : 11000011
```

Nat8.bittest

```
func bittest(x : Nat8, p : Nat) : Bool
```

The function `bittest` takes one `Nat8` and one `Nat` value and returns a `Nat8` value.

```
import Nat8 "mo:base/Nat8";

let x : Nat8 = 12; // Binary : 00001100
let p : Nat = 2;

Nat8.bittest(x, p)
```

Nat8.bitset

```
func bitset(x : Nat8, p : Nat) : Bool
```

The function `bitset` takes one `Nat8` and one `Nat` value and returns a `Nat8` value.

```
import Nat8 "mo:base/Nat8";

let x : Nat8 = 12; // Binary : 00001100
let p : Nat = 1;

Nat8.bitset(x, p) // Binary : 00001110
```

Nat8.bitclear

```
func bitclear(x : Nat8, p : Nat) : Bool
```

The function `bitclear` takes one `Nat8` and one `Nat` value and returns a `Nat8` value.

```
import Nat8 "mo:base/Nat8";

let x : Nat8 = 12; // Binary : 00001100
let p : Nat = 3;

Nat8.bitclear(x, p) // Binary : 00000100
```

Nat8.bitflip

```
func bitflip(x : Nat8, p : Nat) : Bool
```

The function `bitflip` takes one `Nat8` and one `Nat` value and returns a `Nat8` value.

```
import Nat8 "mo:base/Nat8";

let x : Nat8 = 12; // Binary : 00001100
let p : Nat = 4;

Nat8.bitflip(x, p) // Binary : 00011100
```

Nat8.bitcountNonZero

```
let bitcountNonZero : (x : Nat8) -> Nat8
```

The function `bitcountNonZero` takes one `Nat8` value and returns a `Nat8` value.

```
import Nat8 "mo:base/Nat8";

let x : Nat8 = 12; // Binary : 00001100

Nat8.bitcountNonZero(x)
```

Nat8.bitcountLeadingZero

```
let bitcountLeadingZero : (x : Nat8) -> Nat8
```

The function `bitcountLeadingZero` takes one `Nat8` value and returns a `Nat8` value.

```
import Nat8 "mo:base/Nat8";

let x : Nat8 = 12; // Binary : 00001100

Nat8.bitcountLeadingZero(x)
```

Nat8.bitcountTrailingZero

```
let bitcountTrailingZero : (x : Nat8) -> Nat8
```

The function `bitcountTrailingZero` takes one `Nat8` value and returns a `Nat8` value.

```
import Nat8 "mo:base/Nat8";

let x : Nat8 = 12; // Binary : 00001100

Nat8.bitcountTrailingZero(x)
```

Nat8.addWrap

```
func addWrap(x : Nat8, y : Nat8) : Nat8
```

The function `addWrap` takes two `Nat8` value and returns a `Nat8` value. It is equivalent to the `+%` Bitwise operators.

```
import Nat8 "mo:base/Nat8";

let x : Nat8 = 255;
let y : Nat8 = 6;

Nat8.addWrap(x, y)
```

Nat8.subWrap

```
func subWrap(x : Nat8, y : Nat8) : Nat8
```

The function `subWrap` takes two `Nat8` value and returns a `Nat8` value. It is equivalent to the `-` `% Bitwise operators`.

```
import Nat8 "mo:base/Nat8";

let x : Nat8 = 0;
let y : Nat8 = 1;

Nat8.subWrap(x, y)
```

Nat8.mulWrap

```
func mulWrap(x : Nat8, y : Nat8) : Nat8
```

The function `mulWrap` takes two `Nat8` value and returns a `Nat8` value. It is equivalent to the `*%` `Bitwise operators`.

```
import Nat8 "mo:base/Nat8";

let x : Nat8 = 128;
let y : Nat8 = 2;

Nat8.mulWrap(x, y)
```

Nat8.powWrap

```
func powWrap(x : Nat8, y : Nat8) : Nat8
```

The function `powWrap` takes two `Nat8` value and returns a `Nat8` value. It is equivalent to the `**%` `Bitwise operators`.

```
import Nat8 "mo:base/Nat8";  
  
let x : Nat8 = 4;  
let y : Nat8 = 4;  
  
Nat8.powWrap(x, y)
```

Nat16

The *convention* is to name the *module alias* after the *file name* it is defined in:

```
import Nat16 "mo:base/Nat16";
```

On this page

Constants

Value `minimumValue`

Value `maximumValue`

Conversion

Function `toNat`

Function `toText`

Function `fromNat`

Function `fromIntWrap`

Comparison

Function `min`

Function `max`

Function `equal`

Function `notEqual`

Function `less`

Function `lessOrEqual`

Function `greater`

Function `greaterOrEqual`

Function `compare`

Numerical Operations

```
Function add
Function sub
Function mul
Function div
Function rem
Function pow
```

Bitwise Operators

```
Function bitnot
Function bitand
Function bitor
Function bitxor
Function bitshiftLeft
Function bitshiftRight
Function bitrotLeft
Function bitrotRight
Function bittest
Function bitset
Function bitclear
Function bitflip
Function bitcountNonZero
Function bitcountLeadingZero
Function bitcountTrailingZero
```

Wrapping Operations

```
Function addWrap
Function subWrap
Function mulWrap
Function powWrap
```

minimumValue

```
let minValue : Nat16 = 0;
```

maximumValue

```
let maximumValue : Nat16 = 65_535;
```

Nat16.toNat

```
func toNat(i : Nat16) : Nat
```

The function `toNat` takes one `Nat16` value and returns a `Nat` value.

```
import Nat16 "mo:base/Nat16";  
  
let a : Nat16 = 65535;  
  
Nat16.toNat(a);
```

Nat16.toText

```
func toText(i : Nat16) : Text
```

The function `toText` takes one `Nat16` value and returns a `Text` value.

```
import Nat16 "mo:base/Nat16";  
  
let b : Nat16 = 65535;  
  
Nat16.toText(b);
```

Nat16.fromNat

```
func fromNat(i : Nat) : Nat16
```

The function `fromNat` takes one `Nat` value and returns a `Nat16` value.

```
import Nat16 "mo:base/Nat16";

let number : Nat = 65535;

Nat16.fromNat(number);
```

Nat16.fromIntWrap

```
func fromIntWrap(i : Int) : Nat
```

The function `fromIntWrap` takes one `Int` value and returns a `Nat16` value.

```
import Nat16 "mo:base/Nat16";

let integer : Int = 65537;

Nat16.fromIntWrap(integer);
```

Nat16.min

```
func min(x : Nat16, y : Nat16) : Nat16
```

The function `min` takes two `Nat16` value and returns a `Nat16` value.

```
import Nat16 "mo:base/Nat16";

let x : Nat16 = 200;
let y : Nat16 = 100;

Nat16.min(x, y);
```

Nat16.max

```
func max(x : Nat16, y : Nat16) : Nat16
```

The function `max` takes two `Nat16` value and returns a `Nat16` value.

```
import Nat16 "mo:base/Nat16";

let x : Nat16 = 200;
let y : Nat16 = 100;

Nat16.max(x, y);
```

Nat16.equal

```
func equal(x : Nat16, y : Nat16) : Bool
```

The function `equal` takes two `Nat16` value and returns a `Bool` value.

```
import Nat16 "mo:base/Nat16";

let x : Nat16 = 10;
let y : Nat16 = 10;

Nat16.equal(x, y);
```

Nat16.notEqual

```
func notEqual(x : Nat16, y : Nat16) : Bool
```

The function `notEqual` takes two `Nat16` value and returns a `Bool` value.

```
import Nat16 "mo:base/Nat16";

let x : Nat16 = 100;
let y : Nat16 = 100;

Nat16.notEqual(x, y);
```

Nat16.less

```
func less(x : Nat16, y : Nat16) : Bool
```

The function `less` takes two `Nat16` value and returns a `Bool` value.

```
import Nat16 "mo:base/Nat16";

let x : Nat16 = 255;
let y : Nat16 = 256;

Nat16.less(x, y);
```

Nat16.lessOrEqual

```
func lessOrEqual(x : Nat16, y : Nat16) : Bool
```

The function `lessOrEqual` takes two `Nat16` value and returns a `Bool` value.

```
import Nat16 "mo:base/Nat16";

let x : Nat16 = 255;
let y : Nat16 = 256;

Nat16.lessOrEqual(x, y);
```

Nat16.greater

```
func greater(x : Nat16, y : Nat16) : Bool
```

The function `greater` takes two `Nat16` value and returns a `Bool` value.

```
import Nat16 "mo:base/Nat16";

let x : Nat16 = 150;
let y : Nat16 = 100;

Nat16.greater(x, y);
```

Nat16.greaterOrEqual

```
func greaterOrEqual(x : Nat16, y : Nat16) : Bool
```

The function `greaterOrEqual` takes two `Nat16` value and returns a `Bool` value.

```
import Nat16 "mo:base/Nat16";

let x : Nat16 = 150;
let y : Nat16 = 100;

Nat16.greaterOrEqual(x, y);
```

Nat16.compare

```
func compare(x : Nat16, y : Nat16) : Bool
```

The function `compare` takes two `Nat16` value and returns an `Order` variant value.

```
import Nat16 "mo:base/Nat16";

let x : Nat16 = 1001;
let y : Nat16 = 1000;

Nat16.compare(x, y);
```

Nat16.add

```
func add(x : Nat16, y : Nat16) : Nat16
```

The function `add` takes two `Nat16` value and returns a `Nat16` value.

```
import Nat16 "mo:base/Nat16";

let x : Nat16 = 400;
let y : Nat16 = 100;

Nat16.add(x, y);
```

Nat16.sub

```
func sub(x : Nat16, y : Nat16) : Nat16
```

The function `sub` takes two `Nat16` value and returns a `Nat16` value.

```
import Nat16 "mo:base/Nat16";  
  
let x : Nat16 = 400;  
let y : Nat16 = 200;  
  
Nat16.sub(x, y);
```

Nat16.mul

```
func mul(x : Nat16, y : Nat16) : Nat16
```

The function `mul` takes two `Nat16` value and returns a `Nat16` value.

```
import Nat16 "mo:base/Nat16";  
  
let x : Nat16 = 20;  
let y : Nat16 = 50;  
  
Nat16.mul(x, y);
```

Nat16.div

```
func div(x : Nat16, y : Nat16) : Nat16
```

The function `div` takes two `Nat16` value and returns a `Nat16` value.

```
import Nat16 "mo:base/Nat16";  
  
let x : Nat16 = 5000;  
let y : Nat16 = 50;  
  
Nat16.div(x, y);
```

Nat16.rem

```
func rem(x : Nat16, y : Nat16) : Nat16
```

The function `rem` takes two `Nat16` value and returns a `Nat16` value.

```
import Nat16 "mo:base/Nat16";

let x : Nat16 = 400;
let y : Nat16 = 150;

Nat16.rem(x, y);
```

Nat16.pow

```
func pow(x : Nat16, y : Nat16) : Nat16
```

The function `pow` takes two `Nat16` value and returns a `Nat16` value.

```
import Nat16 "mo:base/Nat16";

let x : Nat16 = 10;
let y : Nat16 = 4;

Nat16.pow(x, y);
```

Nat16.bitnot

```
func bitnot(x : Nat16) : Nat16
```

The function `bitnot` takes one `Nat16` value and returns a `Nat16` value.

```
import Nat16 "mo:base/Nat16";

let x : Nat16 = 65520; // Binary : 11111111_11110000

Nat16.bitnot(x) // Binary : 00000000_00001111
```

Nat16.bitand

```
func bitand(x : Nat16, y : Nat16) : Nat16
```

The function `bitand` takes two `Nat16` value and returns a `Nat16` value.

```
import Nat16 "mo:base/Nat16";

let x : Nat16 = 65530; // Binary : 11111111_11111111
let y : Nat16 = 5; // Binary : 00000000_00000101

Nat16.bitand(x, y) // Binary : 00000000_00000101
```

Nat16.bitor

```
func bitor(x : Nat16, y : Nat16) : Nat16
```

The function `bitor` takes two `Nat16` value and returns a `Nat16` value.

```
import Nat16 "mo:base/Nat16";

let x : Nat16 = 240; // Binary : 00000000_11110000
let y : Nat16 = 15; // Binary : 00000000_00001111

Nat16.bitor(x, y) // Binary : 00000000_11111111
```

Nat16.bitxor

```
func bitxor(x : Nat16, y : Nat16) : Nat16
```

The function `bitxor` takes two `Nat16` value and returns a `Nat16` value.

```
import Nat16 "mo:base/Nat16";

let x : Nat16 = 255; // Binary : 00000000_11111111
let y : Nat16 = 240; // Binary : 00000000_11110000

Nat16.bitxor(x, y) // Binary : 00000000_00001111
```

Nat16.bitshiftLeft

```
func bitshiftLeft(x : Nat16, y : Nat16) : Nat16
```

The function `bitshiftLeft` takes two `Nat16` value and returns a `Nat16` value.

```
import Nat16 "mo:base/Nat16";

let x : Nat16 = 15; // Binary : 00000000_00001111
let y : Nat16 = 4;

Nat16.bitshiftLeft(x, y) // Binary : 00000000_11110000
```

Nat16.bitshiftRight

```
func bitshiftRight(x : Nat16, y : Nat16) : Nat16
```

The function `bitshiftRight` takes two `Nat16` value and returns a `Nat16` value.

```
import Nat16 "mo:base/Nat16";

let x : Nat16 = 240; // Binary : 00000000_11110000
let y : Nat16 = 4;

Nat16.bitshiftRight(x, y) // Binary : 00000000_00001111
```

Nat16.bitrotLeft

```
func bitrotLeft(x : Nat16, y : Nat16) : Nat16
```

The function `bitrotLeft` takes two `Nat16` value and returns a `Nat16` value.

```
import Nat16 "mo:base/Nat16";

let x : Nat16 = 32768; // Binary : 10000000_00000000
let y : Nat16 = 1;

Nat16.bitrotLeft(x, y) // Binary : 00000000_00000001
```

Nat16.bitrotRight

```
func bitrotRight(x : Nat16, y : Nat16) : Nat16
```

The function `bitrotRight` takes two `Nat16` value and returns a `Nat16` value.

```
import Nat16 "mo:base/Nat16";

let x : Nat16 = 1; // Binary : 00000000_00000001
let y : Nat16 = 1;

Nat16.bitrotRight(x, y) // Binary : 10000000_00000000
```

Nat16.bittest

```
func bittest(x : Nat16, p : Nat) : Bool
```

The function `bittest` takes one `Nat16` and one `Nat` value and returns a `Nat16` value.

```
import Nat16 "mo:base/Nat16";

let x : Nat16 = 255; // Binary : 00000000_11111111
let p : Nat = 7;

Nat16.bittest(x, p)
```

Nat16.bitset

```
func bitset(x : Nat16, p : Nat) : Bool
```

The function `bitset` takes one `Nat16` and one `Nat` value and returns a `Nat16` value.

```
import Nat16 "mo:base/Nat16";

let x : Nat16 = 127; // Binary : 00000000_01111111
let p : Nat = 7;

Nat16.bitset(x, p) // Binary : 00000000_11111111
```

Nat16.bitclear

```
func bitclear(x : Nat16, p : Nat) : Bool
```

The function `bitclear` takes one `Nat16` and one `Nat` value and returns a `Nat16` value.

```
import Nat16 "mo:base/Nat16";

let x : Nat16 = 255; // Binary : 00000000_11111111
let p : Nat = 7;

Nat16.bitclear(x, p) // Binary : 00000000_01111111
```

Nat16.bitflip

```
func bitflip(x : Nat16, p : Nat) : Bool
```

The function `bitflip` takes one `Nat16` and one `Nat` value and returns a `Nat16` value.

```
import Nat16 "mo:base/Nat16";

let x : Nat16 = 255; // Binary : 00000000_11111111
let p : Nat = 7;

Nat16.bitflip(x, p) // Binary : 00000000_01111111
```

Nat16.bitcountNonZero

```
let bitcountNonZero : (x : Nat16) -> Nat16
```

The function `bitcountNonZero` takes one `Nat16` value and returns a `Nat16` value.

```
import Nat16 "mo:base/Nat16";

let x : Nat16 = 255; // Binary : 00000000_11111111

Nat16.bitcountNonZero(x)
```

Nat16.bitcountLeadingZero

```
let bitcountLeadingZero : (x : Nat16) -> Nat16
```

The function `bitcountLeadingZero` takes one `Nat16` value and returns a `Nat16` value.

```
import Nat16 "mo:base/Nat16";

let x : Nat16 = 255; // Binary : 00000000_11111111

Nat16.bitcountLeadingZero(x)
```

Nat16.bitcountTrailingZero

```
let bitcountTrailingZero : (x : Nat16) -> Nat16
```

The function `bitcountTrailingZero` takes one `Nat16` value and returns a `Nat16` value.

```
import Nat16 "mo:base/Nat16";

let x : Nat16 = 128; // Binary : 00000000_10000000

Nat16.bitcountTrailingZero(x)
```

Nat16.addWrap

```
func addWrap(x : Nat16, y : Nat16) : Nat16
```

The function `addWrap` takes two `Nat16` value and returns a `Nat16` value. It is equivalent to the `+%` Bitwise operators.

```
import Nat16 "mo:base/Nat16";

let x : Nat16 = 65535;
let y : Nat16 = 2;

Nat16.addWrap(x, y)
```

Nat16.subWrap

```
func subWrap(x : Nat16, y : Nat16) : Nat16
```

The function `subWrap` takes two `Nat16` value and returns a `Nat16` value. It is equivalent to the `-%` [Bitwise operators](#).

```
import Nat16 "mo:base/Nat16";  
  
let x : Nat16 = 0;  
let y : Nat16 = 2;  
  
Nat16.subWrap(x, y)
```

Nat16.mulWrap

```
func mulWrap(x : Nat16, y : Nat16) : Nat16
```

The function `mulWrap` takes two `Nat16` value and returns a `Nat16` value. It is equivalent to the `*%` [Bitwise operators](#).

```
import Nat16 "mo:base/Nat16";  
  
let x : Nat16 = 256;  
let y : Nat16 = 256;  
  
Nat16.mulWrap(x, y)
```

Nat16.powWrap

```
func powWrap(x : Nat16, y : Nat16) : Nat16
```

The function `powWrap` takes two `Nat16` value and returns a `Nat16` value. It is equivalent to the `**%` [Bitwise operators](#).

```
import Nat16 "mo:base/Nat16";\n\nlet x : Nat16 = 256;\nlet y : Nat16 = 2;\n\nNat16.powWrap(x, y)
```

Nat32

The *convention* is to name the *module alias* after the *file name* it is defined in:

```
import Nat32 "mo:base/Nat32";
```

On this page

Constants

Value `minimumValue`

Value `maximumValue`

Conversion

Function `toNat`

Function `toText`

Function `fromNat`

Function `fromIntWrap`

Comparison

Function `min`

Function `max`

Function `equal`

Function `notEqual`

Function `less`

Function `lessOrEqual`

Function `greater`

Function `greaterOrEqual`

Function `compare`

Numerical Operations

```
Function add
Function sub
Function mul
Function div
Function rem
Function pow
```

Bitwise Operators

```
Function bitnot
Function bitand
Function bitor
Function bitxor
Function bitshiftLeft
Function bitshiftRight
Function bitrotLeft
Function bitrotRight
Function bittest
Function bitset
Function bitclear
Function bitflip
Function bitcountNonZero
Function bitcountLeadingZero
Function bitcountTrailingZero
```

Wrapping Operations

```
Function addWrap
Function subWrap
Function mulWrap
Function powWrap
```

minimumValue

```
let minValue : Nat32 = 0;
```

maximumValue

```
let maximumValue : Nat32 = 4_294_967_295;
```

Nat32.toNat

```
func toNat(i : Nat32) : Nat
```

The function `toNat` takes one `Nat32` value and returns a `Nat` value.

```
import Nat32 "mo:base/Nat32";  
  
let a : Nat32 = 4294967295;  
  
Nat32.toNat(a);
```

Nat32.toText

```
func toText(i : Nat32) : Text
```

The function `toText` takes one `Nat32` value and returns a `Text` value.

```
import Nat32 "mo:base/Nat32";  
  
let b : Nat32 = 4294967295;  
  
Nat32.toText(b);
```

Nat32.fromNat

```
func fromNat(i : Nat) : Nat32
```

The function `fromNat` takes one `Nat` value and returns a `Nat32` value.

```
import Nat32 "mo:base/Nat32";  
  
let number : Nat = 4294967295;  
  
Nat32.fromNat(number);
```

Nat32.fromIntWrap

```
func fromIntWrap(i : Int) : Nat32
```

The function `fromIntWrap` takes one `Int` value and returns a `Nat32` value.

```
import Nat32 "mo:base/Nat32";  
  
let integer : Int = 4294967295;  
  
Nat32.fromIntWrap(integer);
```

Nat32.min

```
func min(x : Nat32, y : Nat32) : Nat32
```

The function `min` takes two `Nat32` value and returns a `Nat32` value.

```
import Nat32 "mo:base/Nat32";  
  
let x : Nat32 = 2000;  
let y : Nat32 = 1001;  
  
Nat32.min(x, y);
```

Nat32.max

```
func max(x : Nat32, y : Nat32) : Nat32
```

The function `max` takes two `Nat32` value and returns a `Nat32` value.

```
import Nat32 "mo:base/Nat32";

let x : Nat32 = 2000;
let y : Nat32 = 2001;

Nat32.max(x, y);
```

Nat32.equal

```
func equal(x : Nat32, y : Nat32) : Bool
```

The function `equal` takes two `Nat32` value and returns a `Bool` value.

```
import Nat32 "mo:base/Nat32";

let x : Nat32 = 1000;
let y : Nat32 = 100;

Nat32.equal(x, y);
```

Nat32.notEqual

```
func notEqual(x : Nat32, y : Nat32) : Bool
```

The function `notEqual` takes two `Nat32` value and returns a `Bool` value.

```
import Nat32 "mo:base/Nat32";

let x : Nat32 = 1000;
let y : Nat32 = 1001;

Nat32.notEqual(x, y);
```

Nat32.less

```
func less(x : Nat32, y : Nat32) : Bool
```

The function `less` takes two `Nat32` value and returns a `Bool` value.

```
import Nat32 "mo:base/Nat32";

let x : Nat32 = 2000;
let y : Nat32 = 2500;

Nat32.less(x, y);
```

Nat32.lessOrEqual

```
func lessOrEqual(x : Nat32, y : Nat32) : Bool
```

The function `lessOrEqual` takes two `Nat32` value and returns a `Bool` value.

```
import Nat32 "mo:base/Nat32";

let x : Nat32 = 2000;
let y : Nat32 = 2500;

Nat32.lessOrEqual(x, y);
```

Nat32.greater

```
func greater(x : Nat32, y : Nat32) : Bool
```

The function `greater` takes two `Nat32` value and returns a `Bool` value.

```
import Nat32 "mo:base/Nat32";

let x : Nat32 = 2000;
let y : Nat32 = 1000;

Nat32.greater(x, y);
```

Nat32.greaterOrEqual

```
func greaterOrEqual(x : Nat32, y : Nat32) : Bool
```

The function `greaterOrEqual` takes two `Nat32` value and returns a `Bool` value.

```
import Nat32 "mo:base/Nat32";  
  
let x : Nat32 = 2000;  
let y : Nat32 = 1000;  
  
Nat32.greaterOrEqual(x, y);
```

Nat32.compare

```
func compare(x : Nat32, y : Nat32) : Bool
```

The function `compare` takes two `Nat32` value and returns an `Order` variant value.

```
import Nat32 "mo:base/Nat32";  
  
let x : Nat32 = 10000;  
let y : Nat32 = 9999;  
  
Nat32.compare(x, y);
```

Nat32.add

```
func add(x : Nat32, y : Nat32) : Nat32
```

The function `add` takes two `Nat32` value and returns a `Nat32` value.

```
import Nat32 "mo:base/Nat32";  
  
let x : Nat32 = 2000;  
let y : Nat32 = 1200;  
  
Nat32.add(x, y);
```

Nat32.sub

```
func sub(x : Nat32, y : Nat32) : Nat32
```

The function `sub` takes two `Nat32` value and returns a `Nat32` value.

```
import Nat32 "mo:base/Nat32";  
  
let x : Nat32 = 4000;  
let y : Nat32 = 3999;  
  
Nat32.sub(x, y);
```

Nat32.mul

```
func mul(x : Nat32, y : Nat32) : Nat32
```

The function `mul` takes two `Nat32` value and returns a `Nat32` value.

```
import Nat32 "mo:base/Nat32";  
  
let x : Nat32 = 200;  
let y : Nat32 = 50;  
  
Nat32.mul(x, y);
```

Nat32.div

```
func div(x : Nat32, y : Nat32) : Nat32
```

The function `div` takes two `Nat32` value and returns a `Nat32` value.

```
import Nat32 "mo:base/Nat32";  
  
let x : Nat32 = 5000;  
let y : Nat32 = 500;  
  
Nat32.div(x, y);
```

Nat32.rem

```
func rem(x : Nat32, y : Nat32) : Nat32
```

The function `rem` takes two `Nat32` value and returns a `Nat32` value.

```
import Nat32 "mo:base/Nat32";

let x : Nat32 = 4000;
let y : Nat32 = 1200;

Nat32.rem(x, y);
```

Nat32.pow

```
func pow(x : Nat32, y : Nat32) : Nat32
```

The function `pow` takes two `Nat32` value and returns a `Nat32` value.

```
import Nat32 "mo:base/Nat32";

let x : Nat32 = 10;
let y : Nat32 = 5;

Nat32.pow(x, y);
```

Nat32.bitnot

```
func bitnot(x : Nat32) : Nat32
```

The function `bitnot` takes one `Nat32` value and returns a `Nat32` value.

```
import Nat32 "mo:base/Nat32";

let x : Nat32 = 4294967040; // Binary : 11111111_11111111_11111111_00000000

Nat32.bitnot(x) // Binary : 00000000_00000000_00000000_11111111
```

Nat32.bitand

```
func bitand(x : Nat32, y : Nat32) : Nat32
```

The function `bitand` takes two `Nat32` value and returns a `Nat32` value.

```
import Nat32 "mo:base/Nat32";

let x : Nat32 = 4294967295; // Binary : 11111111_11111111_11111111_11111111
let y : Nat32 = 255; // Binary : 00000000_00000000_00000000_11111111

Nat32.bitand(x, y) // Binary : 00000000_00000000_00000000_11111111
```

Nat32.bitor

```
func bitor(x : Nat32, y : Nat32) : Nat32
```

The function `bitor` takes two `Nat32` value and returns a `Nat32` value.

```
import Nat32 "mo:base/Nat32";

let x : Nat32 = 240; // Binary : 00000000_00000000_00000000_11110000
let y : Nat32 = 15; // Binary : 00000000_00000000_00000000_00001111

Nat32.bitor(x, y) // Binary : 00000000_00000000_00000000_11111111
```

Nat32.bitxor

```
func bitxor(x : Nat32, y : Nat32) : Nat32
```

The function `bitxor` takes two `Nat32` value and returns a `Nat32` value.

```
import Nat32 "mo:base/Nat32";

let x : Nat32 = 255; // Binary : 00000000_00000000_00000000_11111111
let y : Nat32 = 240; // Binary : 00000000_00000000_00000000_11110000

Nat32.bitxor(x, y) // Binary : 00000000_00000000_00000000_00001111
```

Nat32.bitshiftLeft

```
func bitshiftLeft(x : Nat32, y : Nat32) : Nat32
```

The function `bitshiftLeft` takes two `Nat32` value and returns a `Nat32` value.

```
import Nat32 "mo:base/Nat32";

let x : Nat32 = 15; // Binary : 00000000_00000000_00000000_00001111
let y : Nat32 = 4;

Nat32.bitshiftLeft(x, y) // Binary : 00000000_00000000_00000000_11110000
```

Nat32.bitshiftRight

```
func bitshiftRight(x : Nat32, y : Nat32) : Nat32
```

The function `bitshiftRight` takes two `Nat32` value and returns a `Nat32` value.

```
import Nat32 "mo:base/Nat32";

let x : Nat32 = 240; // Binary : 00000000_00000000_00000000_11110000
let y : Nat32 = 4;

Nat32.bitshiftRight(x, y) // Binary : 00000000_00000000_00000000_00001111
```

Nat32.bitrotLeft

```
func bitrotLeft(x : Nat32, y : Nat32) : Nat32
```

The function `bitrotLeft` takes two `Nat32` value and returns a `Nat32` value.

```
import Nat32 "mo:base/Nat32";

let x : Nat32 = 4_278_190_080; // Binary : 1111111_00000000_00000000_00000000
let y : Nat32 = 8;

Nat32.bitrotLeft(x, y) // Binary : 00000000_00000000_00000000_11111111
```

Nat32.bitrotRight

```
func bitrotRight(x : Nat32, y : Nat32) : Nat32
```

The function `bitrotRight` takes two `Nat32` value and returns a `Nat32` value.

```
import Nat32 "mo:base/Nat32";

let x : Nat32 = 255; // Binary : 00000000_00000000_00000000_11111111
let y : Nat32 = 8;

Nat32.bitrotRight(x, y) // Binary : 11111111_00000000_00000000_00000000
```

Nat32.bittest

```
func bittest(x : Nat32, p : Nat) : Bool
```

The function `bittest` takes one `Nat32` and one `Nat` value and returns a `Nat32` value.

```
import Nat32 "mo:base/Nat32";

let x : Nat32 = 255; // Binary : 00000000_00000000_00000000_11111111
let p : Nat = 7;

Nat32.bittest(x, p)
```

Nat32.bitset

```
func bitset(x : Nat32, p : Nat) : Bool
```

The function `bitset` takes one `Nat32` and one `Nat` value and returns a `Nat32` value.

```
import Nat32 "mo:base/Nat32";

let x : Nat32 = 127; // Binary : 00000000_00000000_00000000_01111111
let p : Nat = 7;

Nat32.bitset(x, p) // Binary : 00000000_00000000_00000000_11111111
```

Nat32.bitclear

```
func bitclear(x : Nat32, p : Nat) : Bool
```

The function `bitclear` takes one `Nat32` and one `Nat` value and returns a `Nat32` value.

```
import Nat32 "mo:base/Nat32";

let x : Nat32 = 255; // Binary : 00000000_00000000_00000000_11111111
let p : Nat = 7;

Nat32.bitclear(x, p) // Binary : 00000000_00000000_00000000_01111111
```

Nat32.bitflip

```
func bitflip(x : Nat32, p : Nat) : Bool
```

The function `bitflip` takes one `Nat32` and one `Nat` value and returns a `Nat32` value.

```
import Nat32 "mo:base/Nat32";

let x : Nat32 = 255; // Binary : 00000000_00000000_00000000_11111111
let p : Nat = 7;

Nat32.bitflip(x, p) // Binary : 00000000_00000000_00000000_01111111
```

Nat32.bitcountNonZero

```
let bitcountNonZero : (x : Nat32) -> Nat32
```

The function `bitcountNonZero` takes one `Nat32` value and returns a `Nat32` value.

```
import Nat32 "mo:base/Nat32";

let x : Nat32 = 255; // Binary : 00000000_00000000_00000000_11111111

Nat32.bitcountNonZero(x)
```

Nat32.bitcountLeadingZero

```
let bitcountLeadingZero : (x : Nat32) -> Nat32
```

The function `bitcountLeadingZero` takes one `Nat32` value and returns a `Nat32` value.

```
import Nat32 "mo:base/Nat32";

let x : Nat32 = 255; // Binary : 00000000_00000000_00000000_11111111
Nat32.bitcountLeadingZero(x)
```

Nat32.bitcountTrailingZero

```
let bitcountTrailingZero : (x : Nat32) -> Nat32
```

The function `bitcountTrailingZero` takes one `Nat32` value and returns a `Nat32` value.

```
import Nat32 "mo:base/Nat32";

let x : Nat32 = 128; // Binary : 00000000_00000000_00000000_10000000
Nat32.bitcountTrailingZero(x)
```

Nat32.addWrap

```
func addWrap(x : Nat32, y : Nat32) : Nat32
```

The function `addWrap` takes two `Nat32` value and returns a `Nat32` value. It is equivalent to the `+%` [Bitwise operators](#).

```
import Nat32 "mo:base/Nat32";

let x : Nat32 = 4294967295;
let y : Nat32 = 1;

Nat32.addWrap(x, y)
```

Nat32.subWrap

```
func subWrap(x : Nat32, y : Nat32) : Nat32
```

The function `subWrap` takes two `Nat32` value and returns a `Nat32` value. It is equivalent to the `-%` [Bitwise operators](#).

```
import Nat32 "mo:base/Nat32";  
  
let x : Nat32 = 0;  
let y : Nat32 = 2;  
  
Nat32.subWrap(x, y)
```

Nat32.mulWrap

```
func mulWrap(x : Nat32, y : Nat32) : Nat32
```

The function `mulWrap` takes two `Nat32` value and returns a `Nat32` value. It is equivalent to the `*%` [Bitwise operators](#).

```
import Nat32 "mo:base/Nat32";  
  
let x : Nat32 = 65536;  
let y : Nat32 = 65536;  
  
Nat32.mulWrap(x, y)
```

Nat32.powWrap

```
func powWrap(x : Nat32, y : Nat32) : Nat32
```

The function `powWrap` takes two `Nat32` value and returns a `Nat32` value. It is equivalent to the `**%` [Bitwise operators](#).

```
import Nat32 "mo:base/Nat32";\n\nlet x : Nat32 = 65536;\nlet y : Nat32 = 2;\n\nNat32.powWrap(x, y)
```

Nat64

The *convention* is to name the *module alias* after the *file name* it is defined in:

```
import Nat64 "mo:base/Nat64";
```

On this page

Constants

```
Value minimumValue  
Value maximumValue
```

Conversion

```
Function toNat  
Function toText  
Function fromNat  
Function fromIntWrap
```

Comparison

```
Function min  
Function max  
Function equal  
Function notEqual  
Function less  
Function lessOrEqual  
Function greater  
Function greaterOrEqual  
Function compare
```

Numerical Operations

```
Function add
Function sub
Function mul
Function div
Function rem
Function pow
```

Bitwise Operators

```
Function bitnot
Function bitand
Function bitor
Function bitxor
Function bitshiftLeft
Function bitshiftRight
Function bitrotLeft
Function bitrotRight
Function bittest
Function bitset
Function bitclear
Function bitflip
Function bitcountNonZero
Function bitcountLeadingZero
Function bitcountTrailingZero
```

Wrapping Operations

```
Function addWrap
Function subWrap
Function mulWrap
Function powWrap
```

minimumValue

```
let minValue : Nat64 = 0;
```

maximumValue

```
let maximumValue : Nat64 = 18_446_744_073_709_551_615;
```

Nat64.toNat

```
func toNat(i : Nat64) : Nat
```

The function `toNat` takes one `Nat64` value and returns an `Nat` value.

```
import Nat64 "mo:base/Nat64";  
  
let a : Nat64 = 184467;  
  
Nat64.toNat(a);
```

Nat64.toText

```
func toText(i : Nat64) : Text
```

The function `toText` takes one `Nat64` value and returns a `Text` value.

```
import Nat64 "mo:base/Nat64";  
  
let b : Nat64 = 184467;  
  
Nat64.toText(b);
```

Nat64.fromNat

```
func fromNat(i : Nat) : Nat64
```

The function `fromNat` takes one `Nat` value and returns an `Nat64` value.

```
import Nat64 "mo:base/Nat64";  
  
let number : Nat = 184467;  
  
Nat64.fromNat(number);
```

Nat64.fromIntWrap

```
func fromIntWrap(i : Int) : Nat64
```

The function `fromIntWrap` takes one `Int` value and returns an `Nat64` value.

```
import Nat64 "mo:base/Nat64";  
  
let integer : Int = 18_446_744_073_709_551_616;  
  
Nat64.fromIntWrap(integer);
```

Nat64.min

```
func min(x : Nat64, y : Nat64) : Nat64
```

The function `min` takes two `Nat64` value and returns a `Nat64` value.

```
import Nat64 "mo:base/Nat64";  
  
let x : Nat64 = 2000;  
let y : Nat64 = 1001;  
  
Nat64.min(x, y);
```

Nat64.max

```
func max(x : Nat64, y : Nat64) : Nat64
```

The function `max` takes two `Nat64` value and returns a `Nat64` value.

```
import Nat64 "mo:base/Nat64";  
  
let x : Nat64 = 2000;  
let y : Nat64 = 2001;  
  
Nat64.max(x, y);
```

Nat64.equal

```
func equal(x : Nat64, y : Nat64) : Bool
```

The function `equal` takes two `Nat64` value and returns a `Bool` value.

```
import Nat64 "mo:base/Nat64";  
  
let x : Nat64 = 1000;  
let y : Nat64 = 100;  
  
Nat64.equal(x, y);
```

Nat64.notEqual

```
func notEqual(x : Nat64, y : Nat64) : Bool
```

The function `notEqual` takes two `Nat64` value and returns a `Bool` value.

```
import Nat64 "mo:base/Nat64";  
  
let x : Nat64 = 1000;  
let y : Nat64 = 1001;  
  
Nat64.notEqual(x, y);
```

Nat64.less

```
func less(x : Nat64, y : Nat64) : Bool
```

The function `less` takes two `Nat64` value and returns a `Bool` value.

```
import Nat64 "mo:base/Nat64";

let x : Nat64 = 2000;
let y : Nat64 = 2500;

Nat64.less(x, y);
```

Nat64.lessOrEqual

```
func lessOrEqual(x : Nat64, y : Nat64) : Bool
```

The function `lessOrEqual` takes two `Nat64` value and returns a `Bool` value.

```
import Nat64 "mo:base/Nat64";

let x : Nat64 = 2000;
let y : Nat64 = 2500;

Nat64.lessOrEqual(x, y);
```

Nat64.greater

```
func greater(x : Nat64, y : Nat64) : Bool
```

The function `greater` takes two `Nat64` value and returns a `Bool` value.

```
import Nat64 "mo:base/Nat64";

let x : Nat64 = 2000;
let y : Nat64 = 1000;

Nat64.greater(x, y);
```

Nat64.greaterOrEqual

```
func greaterOrEqual(x : Nat64, y : Nat64) : Bool
```

The function `greaterOrEqual` takes two `Nat64` value and returns a `Bool` value.

```
import Nat64 "mo:base/Nat64";  
  
let x : Nat64 = 2000;  
let y : Nat64 = 1000;  
  
Nat64.greaterOrEqual(x, y);
```

Nat64.compare

```
func compare(x : Nat64, y : Nat64) : Bool
```

The function `compare` takes two `Nat64` value and returns an `Order` variant value.

```
import Nat64 "mo:base/Nat64";  
  
let x : Nat64 = 10000;  
let y : Nat64 = 9999;  
  
Nat64.compare(x, y);
```

Nat64.add

```
func add(x : Nat64, y : Nat64) : Nat64
```

The function `add` takes two `Nat64` value and returns a `Nat64` value.

```
import Nat64 "mo:base/Nat64";  
  
let x : Nat64 = 2000;  
let y : Nat64 = 1200;  
  
Nat64.add(x, y);
```

Nat64.sub

```
func sub(x : Nat64, y : Nat64) : Nat64
```

The function `sub` takes two `Nat64` value and returns a `Nat64` value.

```
import Nat64 "mo:base/Nat64";  
  
let x : Nat64 = 4000;  
let y : Nat64 = 3999;  
  
Nat64.sub(x, y);
```

Nat64.mul

```
func mul(x : Nat64, y : Nat64) : Nat64
```

The function `mul` takes two `Nat64` value and returns a `Nat64` value.

```
import Nat64 "mo:base/Nat64";  
  
let x : Nat64 = 200;  
let y : Nat64 = 50;  
  
Nat64.mul(x, y);
```

Nat64.div

```
func div(x : Nat64, y : Nat64) : Nat64
```

The function `div` takes two `Nat64` value and returns a `Nat64` value.

```
import Nat64 "mo:base/Nat64";  
  
let x : Nat64 = 5000;  
let y : Nat64 = 500;  
  
Nat64.div(x, y);
```

Nat64.rem

```
func rem(x : Nat64, y : Nat64) : Nat64
```

The function `rem` takes two `Nat64` value and returns a `Nat64` value.

```
import Nat64 "mo:base/Nat64";  
  
let x : Nat64 = 4000;  
let y : Nat64 = 1200;  
  
Nat64.rem(x, y);
```

Nat64.pow

```
func pow(x : Nat64, y : Nat64) : Nat64
```

The function `pow` takes two `Nat64` value and returns a `Nat64` value.

```
import Nat64 "mo:base/Nat64";  
  
let x : Nat64 = 10;  
let y : Nat64 = 5;  
  
Nat64.pow(x, y);
```

Nat64.bitnot

```
func bitnot(x : Nat64) : Nat64
```

The function `bitnot` takes one `Nat64` value and returns a `Nat64` value.

```
import Nat64 "mo:base/Nat64";  
  
let x : Nat64 = 18_446_744_073_709_551_360;  
  
Nat64.bitnot(x)
```

Nat64.bitand

```
func bitand(x : Nat64, y : Nat64) : Nat64
```

The function `bitand` takes two `Nat64` value and returns a `Nat64` value.

```
import Nat64 "mo:base/Nat64";  
  
let x : Nat64 = 4294967295;  
let y : Nat64 = 255;  
  
Nat64.bitand(x, y)
```

Nat64.bitor

```
func bitor(x : Nat64, y : Nat64) : Nat64
```

The function `bitor` takes two `Nat64` value and returns a `Nat64` value.

```
import Nat64 "mo:base/Nat64";  
  
let x : Nat64 = 240;  
let y : Nat64 = 15;  
  
Nat64.bitor(x, y)
```

Nat64.bitxor

```
func bitxor(x : Nat64, y : Nat64) : Nat64
```

The function `bitxor` takes two `Nat64` value and returns a `Nat64` value.

```
import Nat64 "mo:base/Nat64";  
  
let x : Nat64 = 255;  
let y : Nat64 = 240;  
  
Nat64.bitxor(x, y)
```

Nat64.bitshiftLeft

```
func bitshiftLeft(x : Nat64, y : Nat64) : Nat64
```

The function `bitshiftLeft` takes two `Nat64` value and returns a `Nat64` value.

```
import Nat64 "mo:base/Nat64";  
  
let x : Nat64 = 15;  
let y : Nat64 = 4;  
  
Nat64.bitshiftLeft(x, y)
```

Nat64.bitshiftRight

```
func bitshiftRight(x : Nat64, y : Nat64) : Nat64
```

The function `bitshiftRight` takes two `Nat64` value and returns a `Nat64` value.

```
import Nat64 "mo:base/Nat64";  
  
let x : Nat64 = 240;  
let y : Nat64 = 4;  
  
Nat64.bitshiftRight(x, y)
```

Nat64.bitrotLeft

```
func bitrotLeft(x : Nat64, y : Nat64) : Nat64
```

The function `bitrotLeft` takes two `Nat64` value and returns a `Nat64` value.

```
import Nat64 "mo:base/Nat64";  
  
let x : Nat64 = 4_278_190_080;  
let y : Nat64 = 8;  
  
Nat64.bitrotLeft(x, y)
```

Nat64.bitrotRight

```
func bitrotRight(x : Nat64, y : Nat64) : Nat64
```

The function `bitrotRight` takes two `Nat64` value and returns a `Nat64` value.

```
import Nat64 "mo:base/Nat64";  
  
let x : Nat64 = 255;  
let y : Nat64 = 8;  
  
Nat64.bitrotRight(x, y)
```

Nat64.bittest

```
func bittest(x : Nat64, p : Nat) : Bool
```

The function `bittest` takes one `Nat64` and one `Nat` value and returns a `Nat64` value.

```
import Nat64 "mo:base/Nat64";  
  
let x : Nat64 = 255;  
let p : Nat = 7;  
  
Nat64.bittest(x, p)
```

Nat64.bitset

```
func bitset(x : Nat64, p : Nat) : Bool
```

The function `bitset` takes one `Nat64` and one `Nat` value and returns a `Nat64` value.

```
import Nat64 "mo:base/Nat64";  
  
let x : Nat64 = 127;  
let p : Nat = 7;  
  
Nat64.bitset(x, p)
```

Nat64.bitclear

```
func bitclear(x : Nat64, p : Nat) : Bool
```

The function `bitclear` takes one `Nat64` and one `Nat` value and returns a `Nat64` value.

```
import Nat64 "mo:base/Nat64";

let x : Nat64 = 255;
let p : Nat = 7;

Nat64.bitclear(x, p)
```

Nat64.bitflip

```
func bitflip(x : Nat64, p : Nat) : Bool
```

The function `bitflip` takes one `Nat64` and one `Nat` value and returns a `Nat64` value.

```
import Nat64 "mo:base/Nat64";

let x : Nat64 = 255;
let p : Nat = 7;

Nat64.bitflip(x, p)
```

Nat64.bitcountNonZero

```
let bitcountNonZero : (x : Nat64) -> Nat64
```

The function `bitcountNonZero` takes one `Nat64` value and returns a `Nat64` value.

```
import Nat64 "mo:base/Nat64";

let x : Nat64 = 255;

Nat64.bitcountNonZero(x)
```

Nat64.bitcountLeadingZero

```
let bitcountLeadingZero : (x : Nat64) -> Nat64
```

The function `bitcountLeadingZero` takes one `Nat64` value and returns a `Nat64` value.

```
import Nat64 "mo:base/Nat64";

let x : Nat64 = 255;

Nat64.bitcountLeadingZero(x)
```

Nat64.bitcountTrailingZero

```
let bitcountTrailingZero : (x : Nat64) -> Nat64
```

The function `bitcountTrailingZero` takes one `Nat64` value and returns a `Nat64` value.

```
import Nat64 "mo:base/Nat64";

let x : Nat64 = 128;

Nat64.bitcountTrailingZero(x)
```

Nat64.addWrap

```
func addWrap(x : Nat64, y : Nat64) : Nat64
```

The function `addWrap` takes two `Nat64` value and returns a `Nat64` value. It is equivalent to the `+%` [Bitwise operators](#).

```
import Nat64 "mo:base/Nat64";

let x : Nat64 = 18_446_744_073_709_551_615;
let y : Nat64 = 1;

Nat64.addWrap(x, y)
```

Nat64.subWrap

```
func subWrap(x : Nat64, y : Nat64) : Nat64
```

The function `subWrap` takes two `Nat64` value and returns a `Nat64` value. It is equivalent to the `-%` [Bitwise operators](#).

```
import Nat64 "mo:base/Nat64";

let x : Nat64 = 0;
let y : Nat64 = 2;

Nat64.subWrap(x, y)
```

Nat64.mulWrap

```
func mulWrap(x : Nat64, y : Nat64) : Nat64
```

The function `mulWrap` takes two `Nat64` value and returns a `Nat64` value. It is equivalent to the `*%` [Bitwise operators](#).

```
import Nat64 "mo:base/Nat64";

let x : Nat64 = 4294967296;
let y : Nat64 = 4294967296;

Nat64.mulWrap(x, y)
```

Nat64.powWrap

```
func powWrap(x : Nat64, y : Nat64) : Nat64
```

The function `powWrap` takes two `Nat64` value and returns a `Nat64` value. It is equivalent to the `**%` [Bitwise operators](#).

```
import Nat64 "mo:base/Nat64";  
  
let x : Nat64 = 4294967296;  
let y : Nat64 = 2;  
  
Nat64.powWrap(x, y)
```

Int8

The *convention* is to name the *module alias* after the *file name* it is defined in:

```
import Int8 "mo:base/Int8";
```

On this page

Constants

[Value minimumValue](#)

[Value maximumValue](#)

Conversion

[Function toInt](#)

[Function toText](#)

[Function fromInt](#)

[Function fromIntWrap](#)

[Function fromNat8](#)

[Function toNat8](#)

Comparison

[Function min](#)

[Function max](#)

[Function equal](#)

[Function notEqual](#)

[Function less](#)

[Function lessOrEqual](#)

[Function greater](#)

[Function greaterOrEqual](#)

[Function compare](#)

Numerical Operations

```
Function abs
Function neg
Function add
Function sub
Function mul
Function div
Function rem
Function pow
```

Bitwise Operators

```
Function bitnot
Function bitand
Function bitor
Function bitxor
Function bitshiftLeft
Function bitshiftRight
Function bitrotLeft
Function bitrotRight
Function bittest
Function bitset
Function bitclear
Function bitflip
Function bitcountNonZero
Function bitcountLeadingZero
Function bitcountTrailingZero
```

Wrapping Operations

```
Function addWrap
Function subWrap
Function mulWrap
Function powWrap
```

minimumValue

```
let minValue : Int8 = -128
```

maximumValue

```
let maxValue : Int8 = 127
```

Int8.toInt

```
funcToInt(i : Int8) : Int
```

The function `toInt` takes one `Int8` value and returns an `Int` value.

```
import Int8 "mo:base/Int8";  
  
let a : Int8 = -127;  
  
Int8.toInt(a);
```

Int8.toText

```
func toText(i : Int8) : Text
```

The function `toText` takes one `Int8` value and returns a `Text` value.

```
import Int8 "mo:base/Int8";  
  
let b : Int8 = -127;  
  
Int8.toText(b);
```

Int8.fromInt

```
func fromInt(i : Int) : Int8
```

The function `fromInt` takes one `Int` value and returns an `Int8` value.

```
import Int8 "mo:base/Int8";  
  
let integer : Int = -127;  
  
Int8.fromInt(integer);
```

Int8.fromIntWrap

```
func fromIntWrap(i : Int) : Int8
```

The function `fromIntWrap` takes one `Int` value and returns an `Int8` value.

```
import Int8 "mo:base/Int8";  
  
let integer : Int = 255;  
  
Int8.fromIntWrap(integer);
```

Int8.fromNat8

```
func fromNat8(i : Nat8) : Int8
```

The function `fromNat8` takes one `Nat8` value and returns an `Int8` value.

```
import Int8 "mo:base/Int8";  
  
let Nat8 : Nat8 = 127;  
  
Int8.fromNat8(Nat8);
```

Int8.toNat8

```
func toNat8(i : Int8) : Nat8
```

The function `toNat8` takes one `Int8` value and returns an `Nat8` value.

```
import Int8 "mo:base/Int8";  
  
let int8 : Int8 = -127;  
  
Int8.toNat8(int8);
```

Int8.min

```
func min(x : Int8, y : Int8) : Int8
```

The function `min` takes two `Int8` value and returns a `Int8` value.

```
import Int8 "mo:base/Int8";  
  
let x : Int8 = 15;  
let y : Int8 = -10;  
  
Int8.min(x, y);
```

Int8.max

```
func max(x : Int8, y : Int8) : Int8
```

The function `max` takes two `Int8` value and returns a `Int8` value.

```
import Int8 "mo:base/Int8";  
  
let x : Int8 = 15;  
let y : Int8 = -15;  
  
Int8.max(x, y);
```

Int8.equal

```
func equal(x : Int8, y : Int8) : Bool
```

The function `equal` takes two `Int8` value and returns a `Bool` value.

```
import Int8 "mo:base/Int8";  
  
let x : Int8 = 10;  
let y : Int8 = 10;  
  
Int8.equal(x, y);
```

Int8.notEqual

```
func notEqual(x : Int8, y : Int8) : Bool
```

The function `notEqual` takes two `Int8` value and returns a `Bool` value.

```
import Int8 "mo:base/Int8";  
  
let x : Int8 = -10;  
let y : Int8 = 10;  
  
Int8.notEqual(x, y);
```

Int8.less

```
func less(x : Int8, y : Int8) : Bool
```

The function `less` takes two `Int8` value and returns a `Bool` value.

```
import Int8 "mo:base/Int8";  
  
let x : Int8 = -15;  
let y : Int8 = -25;  
  
Int8.less(x, y);
```

Int8.lessOrEqual

```
func lessOrEqual(x : Int8, y : Int8) : Bool
```

The function `lessOrEqual` takes two `Int8` value and returns a `Bool` value.

```
import Int8 "mo:base/Int8";  
  
let x : Int8 = -15;  
let y : Int8 = -25;  
  
Int8.lessOrEqual(x, y);
```

Int8.greater

```
func greater(x : Int8, y : Int8) : Bool
```

The function `greater` takes two `Int8` value and returns a `Bool` value.

```
import Int8 "mo:base/Int8";  
  
let x : Int8 = -15;  
let y : Int8 = -10;  
  
Int8.greater(x, y);
```

Int8.greaterOrEqual

```
func greaterOrEqual(x : Int8, y : Int8) : Bool
```

The function `greaterOrEqual` takes two `Int8` value and returns a `Bool` value.

```
import Int8 "mo:base/Int8";  
  
let x : Int8 = 15;  
let y : Int8 = 10;  
  
Int8.greaterOrEqual(x, y);
```

Int8.compare

```
func compare(x : Int8, y : Int8) : Bool
```

The function `compare` takes two `Int8` value and returns an `Order` variant value.

```
import Int8 "mo:base/Int8";  
  
let x : Int8 = -15;  
let y : Int8 = -10;  
  
Int8.compare(x, y);
```

Int8.abs

```
func abs(x : Int8) : Int8
```

The function `abs` takes one `Int8` value and returns a `Int8` value.

```
import Int8 "mo:base/Int8";  
  
let x : Int8 = -40;  
  
Int8.abs(x);
```

Int8.neg

```
func neg(x : Int8) : Int8
```

The function `neg` takes one `Int8` value and returns a `Int8` value.

```
import Int8 "mo:base/Int8";  
  
let x : Int8 = -50;  
  
Int8.neg(x);
```

Int8.add

```
func add(x : Int8, y : Int8) : Int8
```

The function `add` takes two `Int8` value and returns a `Int8` value.

```
import Int8 "mo:base/Int8";  
  
let x : Int8 = -40;  
let y : Int8 = 10;  
  
Int8.add(x, y);
```

Int8.sub

```
func sub(x : Int8, y : Int8) : Int8
```

The function `sub` takes two `Int8` value and returns a `Int8` value.

```
import Int8 "mo:base/Int8";  
  
let x : Int8 = -40;  
let y : Int8 = -50;  
  
Int8.sub(x, y);
```

Int8.mul

```
func mul(x : Int8, y : Int8) : Int8
```

The function `mul` takes two `Int8` value and returns a `Int8` value.

```
import Int8 "mo:base/Int8";  
  
let x : Int8 = 20;  
let y : Int8 = -5;  
  
Int8.mul(x, y);
```

Int8.div

```
func div(x : Int8, y : Int8) : Int8
```

The function `div` takes two `Int8` value and returns a `Int8` value.

```
import Int8 "mo:base/Int8";  
  
let x : Int8 = 50;  
let y : Int8 = -10;  
  
Int8.div(x, y);
```

Int8.rem

```
func rem(x : Int8, y : Int8) : Int8
```

The function `rem` takes two `Int8` value and returns a `Int8` value.

```
import Int8 "mo:base/Int8";  
  
let x : Int8 = -40;  
let y : Int8 = 15;  
  
Int8.rem(x, y);
```

Int8.pow

```
func pow(x : Int8, y : Int8) : Int8
```

The function `pow` takes two `Int8` value and returns a `Int8` value.

```
import Int8 "mo:base/Int8";  
  
let x : Int8 = -5;  
let y : Int8 = 3;  
  
Int8.pow(x, y);
```

Int8.bitnot

```
func bitnot(x : Int8) : Int8
```

The function `bitnot` takes one `Int8` value and returns a `Int8` value.

```
import Int8 "mo:base/Int8";  
  
let x : Int8 = 127; // Binary : 01111111  
  
Int8.bitnot(x) // Binary : 10000000
```

Int8.bitand

```
func bitand(x : Int8, y : Int8) : Int8
```

The function `bitand` takes two `Int8` value and returns a `Int8` value.

```
import Int8 "mo:base/Int8";  
  
let x : Int8 = 127; // Binary : 01111111  
let y : Int8 = 15; // Binary : 00001111  
  
Int8.bitand(x, y)
```

Int8.bitor

```
func bitor(x : Int8, y : Int8) : Int8
```

The function `bitor` takes two `Int8` value and returns a `Int8` value.

```
import Int8 "mo:base/Int8";  
  
let x : Int8 = 112; // Binary : 01110000  
let y : Int8 = 15; // Binary : 00001111  
  
Int8.bitor(x, y) // Binary : 01111111
```

Int8.bitxor

```
func bitxor(x : Int8, y : Int8) : Int8
```

The function `bitxor` takes two `Int8` value and returns a `Int8` value.

```
import Int8 "mo:base/Int8";

let x : Int8 = 15; // Binary : 00001111
let y : Int8 = 14; // Binary : 00001110

Int8.bitxor(x, y) // Binary : 00000001
```

Int8.bitshiftLeft

```
func bitshiftLeft(x : Int8, y : Int8) : Int8
```

The function `bitshiftLeft` takes two `Int8` value and returns a `Int8` value.

```
import Int8 "mo:base/Int8";

let x : Int8 = 15; // Binary : 00001111
let y : Int8 = 4;

Int8.bitshiftLeft(x, y) // Binary : 11110000
```

Int8.bitshiftRight

```
func bitshiftRight(x : Int8, y : Int8) : Int8
```

The function `bitshiftRight` takes two `Int8` value and returns a `Int8` value.

```
import Int8 "mo:base/Int8";

let x : Int8 = 48; // Binary : 00110000
let y : Int8 = 2;

Int8.bitshiftRight(x, y) // Binary : 00001100
```

Int8.bitrotLeft

```
func bitrotLeft(x : Int8, y : Int8) : Int8
```

The function `bitrotLeft` takes two `Int8` value and returns a `Int8` value.

```
import Int8 "mo:base/Int8";

let x : Int8 = 15; // Binary : 00001111
let y : Int8 = 4;

Int8.bitrotLeft(x, y) // Binary : 11110000
```

Int8.bitrotRight

```
func bitrotRight(x : Int8, y : Int8) : Int8
```

The function `bitrotRight` takes two `Int8` value and returns a `Int8` value.

```
import Int8 "mo:base/Int8";

let x : Int8 = 48; // Binary : 00110000
let y : Int8 = 2;

Int8.bitrotRight(x, y) // Binary : 00001100
```

Int8.bittest

```
func bittest(x : Int8, p : Nat) : Bool
```

The function `bittest` takes one `Int8` and one `Nat` value and returns a `Int8` value.

```
import Int8 "mo:base/Int8";

let x : Int8 = 12; // Binary : 00001100
let p : Nat = 2;

Int8.bittest(x, p)
```

Int8.bitset

```
func bitset(x : Int8, p : Nat) : Bool
```

The function `bitset` takes one `Int8` and one `Nat` value and returns a `Int8` value.

```
import Int8 "mo:base/Int8";

let x : Int8 = 12; // Binary : 00001100
let p : Nat = 4;

Int8.bitset(x, p) // Binary : 00011100
```

Int8.bitclear

```
func bitclear(x : Int8, p : Nat) : Bool
```

The function `bitclear` takes one `Int8` and one `Nat` value and returns a `Int8` value.

```
import Int8 "mo:base/Int8";

let x : Int8 = 12; // Binary : 00001100
let p : Nat = 3;

Int8.bitclear(x, p) // Binary : 00000100
```

Int8.bitflip

```
func bitflip(x : Int8, p : Nat) : Bool
```

The function `bitflip` takes one `Int8` and one `Nat` value and returns a `Int8` value.

```
import Int8 "mo:base/Int8";

let x : Int8 = 12; // Binary : 00001100
let p : Nat = 4;

Int8.bitflip(x, p) // Binary : 00011100
```

Int8.bitcountNonZero

```
let bitcountNonZero : (x : Int8) -> Int8
```

The function `bitcountNonZero` takes one `Int8` value and returns a `Int8` value.

```
import Int8 "mo:base/Int8";  
  
let x : Int8 = 15; // Binary : 00001111  
  
Int8.bitcountNonZero(x)
```

Int8.bitcountLeadingZero

```
let bitcountLeadingZero : (x : Int8) -> Int8
```

The function `bitcountLeadingZero` takes one `Int8` value and returns a `Int8` value.

```
import Int8 "mo:base/Int8";  
  
let x : Int8 = 12; // Binary : 00001100  
  
Int8.bitcountLeadingZero(x)
```

Int8.bitcountTrailingZero

```
let bitcountTrailingZero : (x : Int8) -> Int8
```

The function `bitcountTrailingZero` takes one `Int8` value and returns a `Int8` value.

```
import Int8 "mo:base/Int8";  
  
let x : Int8 = 48; // Binary : 00110000  
  
Int8.bitcountTrailingZero(x)
```

Int8.addWrap

```
func addWrap(x : Int8, y : Int8) : Int8
```

The function `addWrap` takes two `Int8` value and returns a `Int8` value. It is equivalent to the `+%` Bitwise operators.

```
import Int8 "mo:base/Int8";

let x : Int8 = 127;
let y : Int8 = 2;

Int8.addWrap(x, y)
```

Int8.subWrap

```
func subWrap(x : Int8, y : Int8) : Int8
```

The function `subWrap` takes two `Int8` value and returns a `Int8` value. It is equivalent to the `-%` Bitwise operators.

```
import Int8 "mo:base/Int8";

let x : Int8 = -128;
let y : Int8 = 2;

Int8.subWrap(x, y)
```

Int8.mulWrap

```
func mulWrap(x : Int8, y : Int8) : Int8
```

The function `mulWrap` takes two `Int8` value and returns a `Int8` value. It is equivalent to the `*%` Bitwise operators.

```
import Int8 "mo:base/Int8";  
  
let x : Int8 = 127;  
let y : Int8 = 2;  
  
Int8.mulWrap(x, y)
```

Int8.powWrap

```
func powWrap(x : Int8, y : Int8) : Int8
```

The function `powWrap` takes two `Int8` value and returns a `Int8` value. It is equivalent to the `**%` [Bitwise operators](#).

```
import Int8 "mo:base/Int8";  
  
let x : Int8 = 6;  
let y : Int8 = 3;  
  
Int8.powWrap(x, y)
```

Int16

The *convention* is to name the *module alias* after the *file name* it is defined in:

```
import Int16 "mo:base/Int16";
```

On this page

Constants

[Value minimumValue](#)
[Value maximumValue](#)

Conversion

[Function toInt](#)
[Function toText](#)
[Function fromInt](#)
[Function fromIntWrap](#)
[Function fromNat16](#)
[Function toNat16](#)

Comparison

[Function min](#)
[Function max](#)
[Function equal](#)
[Function notEqual](#)
[Function less](#)
[Function lessOrEqual](#)
[Function greater](#)
[Function greaterOrEqual](#)
[Function compare](#)

Numerical Operations

```
Function abs
Function neg
Function add
Function sub
Function mul
Function div
Function rem
Function pow
```

Bitwise Operators

```
Function bitnot
Function bitand
Function bitor
Function bitxor
Function bitshiftLeft
Function bitshiftRight
Function bitrotLeft
Function bitrotRight
Function bittest
Function bitset
Function bitclear
Function bitflip
Function bitcountNonZero
Function bitcountLeadingZero
Function bitcountTrailingZero
```

Wrapping Operations

```
Function addWrap
Function subWrap
Function mulWrap
Function powWrap
```

minimumValue

```
let minValue : Int16 = -32_768;
```

maximumValue

```
let maxValue : Int16 = 32_767;
```

Int16.toInt

```
func toInt(i : Int16) : Int
```

The function `toInt` takes one `Int16` value and returns an `Int` value.

```
import Int16 "mo:base/Int16";  
  
let a : Int16 = -32768;  
  
Int16.toInt(a);
```

Int16.toText

```
func toText(i : Int16) : Text
```

The function `toText` takes one `Int16` value and returns a `Text` value.

```
import Int16 "mo:base/Int16";  
  
let b : Int16 = -32768;  
  
Int16.toText(b);
```

Int16.fromInt

```
func fromInt(i : Int) : Int16
```

The function `fromInt` takes one `Int` value and returns an `Int16` value.

```
import Int16 "mo:base/Int16";  
  
let integer : Int = -32768;  
  
Int16.fromInt(integer);
```

Int16.fromIntWrap

```
func fromIntWrap(i : Int) : Int16
```

The function `fromIntWrap` takes one `Int` value and returns an `Int16` value.

```
import Int16 "mo:base/Int16";  
  
let integer : Int = 65535;  
  
Int16.fromIntWrap(integer);
```

Int16.fromNat16

```
func fromNat16(i : Nat16) : Int16
```

The function `fromNat16` takes one `Nat16` value and returns an `Int16` value.

```
import Int16 "mo:base/Int16";  
  
let nat16 : Nat16 = 65535;  
  
Int16.fromNat16(nat16);
```

Int16.toNat16

```
func toNat16(i : Int16) : Nat16
```

The function `toNat16` takes one `Int16` value and returns an `Nat16` value.

```
import Int16 "mo:base/Int16";  
  
let int16 : Int16 = -32768;  
  
Int16.toNat16(int16);
```

Int16.min

```
func min(x : Int16, y : Int16) : Int16
```

The function `min` takes two `Int16` value and returns a `Int16` value.

```
import Int16 "mo:base/Int16";  
  
let x : Int16 = 200;  
let y : Int16 = 100;  
  
Int16.min(x, y);
```

Int16.max

```
func max(x : Int16, y : Int16) : Int16
```

The function `max` takes two `Int16` value and returns a `Int16` value.

```
import Int16 "mo:base/Int16";  
  
let x : Int16 = 200;  
let y : Int16 = 100;  
  
Int16.max(x, y);
```

Int16.equal

```
func equal(x : Int16, y : Int16) : Bool
```

The function `equal` takes two `Int16` value and returns a `Bool` value.

```
import Int16 "mo:base/Int16";  
  
let x : Int16 = 10;  
let y : Int16 = 10;  
  
Int16.equal(x, y);
```

Int16.notEqual

```
func notEqual(x : Int16, y : Int16) : Bool
```

The function `notEqual` takes two `Int16` value and returns a `Bool` value.

```
import Int16 "mo:base/Int16";  
  
let x : Int16 = 100;  
let y : Int16 = 100;  
  
Int16.notEqual(x, y);
```

Int16.less

```
func less(x : Int16, y : Int16) : Bool
```

The function `less` takes two `Int16` value and returns a `Bool` value.

```
import Int16 "mo:base/Int16";  
  
let x : Int16 = 255;  
let y : Int16 = 256;  
  
Int16.less(x, y);
```

Int16.lessOrEqual

```
func lessOrEqual(x : Int16, y : Int16) : Bool
```

The function `lessOrEqual` takes two `Int16` value and returns a `Bool` value.

```
import Int16 "mo:base/Int16";  
  
let x : Int16 = 255;  
let y : Int16 = 256;  
  
Int16.lessOrEqual(x, y);
```

Int16.greater

```
func greater(x : Int16, y : Int16) : Bool
```

The function `greater` takes two `Int16` value and returns a `Bool` value.

```
import Int16 "mo:base/Int16";  
  
let x : Int16 = 150;  
let y : Int16 = 100;  
  
Int16.greater(x, y);
```

Int16.greaterOrEqual

```
func greaterOrEqual(x : Int16, y : Int16) : Bool
```

The function `greaterOrEqual` takes two `Int16` value and returns a `Bool` value.

```
import Int16 "mo:base/Int16";  
  
let x : Int16 = 150;  
let y : Int16 = 100;  
  
Int16.greaterOrEqual(x, y);
```

Int16.compare

```
func compare(x : Int16, y : Int16) : Bool
```

The function `compare` takes two `Int16` value and returns an `Order` variant value.

```
import Int16 "mo:base/Int16";  
  
let x : Int16 = 1001;  
let y : Int16 = 1000;  
  
Int16.compare(x, y);
```

Int16.abs

```
func abs(x : Int16) : Int16
```

The function `abs` takes one `Int16` value and returns a `Int16` value.

```
import Int16 "mo:base/Int16";  
  
let x : Int16 = -40;  
  
Int16.abs(x);
```

Int16.neg

```
func neg(x : Int16) : Int16
```

The function `neg` takes one `Int16` value and returns a `Int16` value.

```
import Int16 "mo:base/Int16";  
  
let x : Int16 = -50;  
  
Int16.neg(x);
```

Int16.add

```
func add(x : Int16, y : Int16) : Int16
```

The function `add` takes two `Int16` value and returns a `Int16` value.

```
import Int16 "mo:base/Int16";  
  
let x : Int16 = 400;  
let y : Int16 = 100;  
  
Int16.add(x, y);
```

Int16.sub

```
func sub(x : Int16, y : Int16) : Int16
```

The function `sub` takes two `Int16` value and returns a `Int16` value.

```
import Int16 "mo:base/Int16";  
  
let x : Int16 = 400;  
let y : Int16 = 200;  
  
Int16.sub(x, y);
```

Int16.mul

```
func mul(x : Int16, y : Int16) : Int16
```

The function `mul` takes two `Int16` value and returns a `Int16` value.

```
import Int16 "mo:base/Int16";  
  
let x : Int16 = 20;  
let y : Int16 = 50;  
  
Int16.mul(x, y);
```

Int16.div

```
func div(x : Int16, y : Int16) : Int16
```

The function `div` takes two `Int16` value and returns a `Int16` value.

```
import Int16 "mo:base/Int16";  
  
let x : Int16 = 5000;  
let y : Int16 = 50;  
  
Int16.div(x, y);
```

Int16.rem

```
func rem(x : Int16, y : Int16) : Int16
```

The function `rem` takes two `Int16` value and returns a `Int16` value.

```
import Int16 "mo:base/Int16";  
  
let x : Int16 = 400;  
let y : Int16 = 150;  
  
Int16.rem(x, y);
```

Int16.pow

```
func pow(x : Int16, y : Int16) : Int16
```

The function `pow` takes two `Int16` value and returns a `Int16` value.

```
import Int16 "mo:base/Int16";  
  
let x : Int16 = 10;  
let y : Int16 = 4;  
  
Int16.pow(x, y);
```

Int16.bitnot

```
func bitnot(x : Int16) : Int16
```

The function `bitnot` takes one `Int16` value and returns a `Int16` value.

```
import Int16 "mo:base/Int16";

let x : Int16 = 255; // Binary : 00000000_11111111
Int16.bitnot(x) // Binary : 11111111_00000000
```

Int16.bitand

```
func bitand(x : Int16, y : Int16) : Int16
```

The function `bitand` takes two `Int16` value and returns a `Int16` value.

```
import Int16 "mo:base/Int16";

let x : Int16 = 255; // Binary : 00000000_11111111
let y : Int16 = 5; // Binary : 00000000_00000101

Int16.bitand(x, y)
```

Int16.bitor

```
func bitor(x : Int16, y : Int16) : Int16
```

The function `bitor` takes two `Int16` value and returns a `Int16` value.

```
import Int16 "mo:base/Int16";

let x : Int16 = 240; // Binary : 00000000_11110000
let y : Int16 = 15; // Binary : 00000000_00001111

Int16.bitor(x, y) // Binary : 00000000_11111111
```

Int16.bitxor

```
func bitxor(x : Int16, y : Int16) : Int16
```

The function `bitxor` takes two `Int16` value and returns a `Int16` value.

```
import Int16 "mo:base/Int16";

let x : Int16 = 255; // Binary : 00000000_11111111
let y : Int16 = 240; // Binary : 00000000_11110000

Int16.bitxor(x, y) // Binary : 00000000_00001111
```

Int16.bitshiftLeft

```
func bitshiftLeft(x : Int16, y : Int16) : Int16
```

The function `bitshiftLeft` takes two `Int16` value and returns a `Int16` value.

```
import Int16 "mo:base/Int16";

let x : Int16 = 15; // Binary : 00000000_00001111
let y : Int16 = 4;

Int16.bitshiftLeft(x, y) // Binary : 00000000_11110000
```

Int16.bitshiftRight

```
func bitshiftRight(x : Int16, y : Int16) : Int16
```

The function `bitshiftRight` takes two `Int16` value and returns a `Int16` value.

```
import Int16 "mo:base/Int16";

let x : Int16 = 240; // Binary : 00000000_11110000
let y : Int16 = 4;

Int16.bitshiftRight(x, y) // Binary : 00000000_00001111
```

Int16.bitrotLeft

```
func bitrotLeft(x : Int16, y : Int16) : Int16
```

The function `bitrotLeft` takes two `Int16` value and returns a `Int16` value.

```
import Int16 "mo:base/Int16";

let x : Int16 = 32767; // Binary : 01111111_11111111
let y : Int16 = 1;

Int16.bitrotLeft(x, y) // Binary : 11111111_11111110
```

Int16.bitrotRight

```
func bitrotRight(x : Int16, y : Int16) : Int16
```

The function `bitrotRight` takes two `Int16` value and returns a `Int16` value.

```
import Int16 "mo:base/Int16";

let x : Int16 = 32767; // Binary : 01111111_11111111
let y : Int16 = 1;

Int16.bitrotRight(x, y) // Binary : 10111111_11111111
```

Int16.bittest

```
func bittest(x : Int16, p : Nat) : Bool
```

The function `bittest` takes one `Int16` and one `Nat` value and returns a `Int16` value.

```
import Int16 "mo:base/Int16";

let x : Int16 = 255; // Binary : 00000000_11111111
let p : Nat = 7;

Int16.bittest(x, p)
```

Int16.bitset

```
func bitset(x : Int16, p : Nat) : Bool
```

The function `bitset` takes one `Int16` and one `Nat` value and returns a `Int16` value.

```
import Int16 "mo:base/Int16";

let x : Int16 = 127; // Binary : 00000000_01111111
let p : Nat = 7;

Int16.bitset(x, p) // Binary : 00000000_11111111
```

Int16.bitclear

```
func bitclear(x : Int16, p : Nat) : Bool
```

The function `bitclear` takes one `Int16` and one `Nat` value and returns a `Int16` value.

```
import Int16 "mo:base/Int16";

let x : Int16 = 255; // Binary : 00000000_11111111
let p : Nat = 7;

Int16.bitclear(x, p) // Binary : 00000000_01111111
```

Int16.bitflip

```
func bitflip(x : Int16, p : Nat) : Bool
```

The function `bitflip` takes one `Int16` and one `Nat` value and returns a `Int16` value.

```
import Int16 "mo:base/Int16";

let x : Int16 = 255; // Binary : 00000000_11111111
let p : Nat = 7;

Int16.bitflip(x, p) // Binary : 00000000_11011111
```

Int16.bitcountNonZero

```
let bitcountNonZero : (x : Int16) -> Int16
```

The function `bitcountNonZero` takes one `Int16` value and returns a `Int16` value.

```
import Int16 "mo:base/Int16";  
  
let x : Int16 = 255; // Binary : 00000000_11111111  
  
Int16.bitcountNonZero(x)
```

Int16.bitcountLeadingZero

```
let bitcountLeadingZero : (x : Int16) -> Int16
```

The function `bitcountLeadingZero` takes one `Int16` value and returns a `Int16` value.

```
import Int16 "mo:base/Int16";  
  
let x : Int16 = 255; // Binary : 00000000_11111111  
  
Int16.bitcountLeadingZero(x)
```

Int16.bitcountTrailingZero

```
let bitcountTrailingZero : (x : Int16) -> Int16
```

The function `bitcountTrailingZero` takes one `Int16` value and returns a `Int16` value.

```
import Int16 "mo:base/Int16";  
  
let x : Int16 = 128; // Binary : 00000000_10000000  
  
Int16.bitcountTrailingZero(x)
```

Int16.addWrap

```
func addWrap(x : Int16, y : Int16) : Int16
```

The function `addWrap` takes two `Int16` value and returns a `Int16` value. It is equivalent to the `+%` Bitwise operators.

```
import Int16 "mo:base/Int16";  
  
let x : Int16 = 32767;  
let y : Int16 = 2;  
  
Int16.addWrap(x, y)
```

Int16.subWrap

```
func subWrap(x : Int16, y : Int16) : Int16
```

The function `subWrap` takes two `Int16` value and returns a `Int16` value. It is equivalent to the `-%` Bitwise operators.

```
import Int16 "mo:base/Int16";  
  
let x : Int16 = -32767;  
let y : Int16 = 2;  
  
Int16.subWrap(x, y)
```

Int16.mulWrap

```
func mulWrap(x : Int16, y : Int16) : Int16
```

The function `mulWrap` takes two `Int16` value and returns a `Int16` value. It is equivalent to the `*%` Bitwise operators.

```
import Int16 "mo:base/Int16";

let x : Int16 = 32767;
let y : Int16 = 2;

Int16.mulWrap(x, y)
```

Int16.powWrap

```
func powWrap(x : Int16, y : Int16) : Int16
```

The function `powWrap` takes two `Int16` value and returns a `Int16` value. It is equivalent to the `**%` [Bitwise operators](#).

```
import Int16 "mo:base/Int16";

let x : Int16 = 255;
let y : Int16 = 2;

Int16.powWrap(x, y)
```

Int32

The *convention* is to name the *module alias* after the *file name* it is defined in:

```
import Int32 "mo:base/Int32";
```

On this page

Constants

[Value minimumValue](#)

[Value maximumValue](#)

Conversion

[Function toInt](#)

[Function toText](#)

[Function fromInt](#)

[Function fromIntWrap](#)

[Function fromNat32](#)

[Function toNat32](#)

Comparison

[Function min](#)

[Function max](#)

[Function equal](#)

[Function notEqual](#)

[Function less](#)

[Function lessOrEqual](#)

[Function greater](#)

[Function greaterOrEqual](#)

[Function compare](#)

Numerical Operations

```
Function abs
Function neg
Function add
Function sub
Function mul
Function div
Function rem
Function pow
```

Bitwise Operators

```
Function bitnot
Function bitand
Function bitor
Function bitxor
Function bitshiftLeft
Function bitshiftRight
Function bitrotLeft
Function bitrotRight
Function bittest
Function bitset
Function bitclear
Function bitflip
Function bitcountNonZero
Function bitcountLeadingZero
Function bitcountTrailingZero
```

Wrapping Operations

```
Function addWrap
Function subWrap
Function mulWrap
Function powWrap
```

minimumValue

```
let minValue : Int32 = -2_147_483_648
```

maximumValue

```
let maxValue : Int32 = 2_147_483_647
```

Int32.toInt

```
func toInt(i : Int32) : Int
```

The function `toInt` takes one `Int32` value and returns an `Int` value.

```
import Int32 "mo:base/Int32";  
  
let a : Int32 = -2147483648;  
  
Int32.toInt(a);
```

Int32.toText

```
func toText(i : Int32) : Text
```

The function `toText` takes one `Int32` value and returns a `Text` value.

```
import Int32 "mo:base/Int32";  
  
let b : Int32 = -2147483648;  
  
Int32.toText(b);
```

Int32.fromInt

```
func fromInt(i : Int) : Int32
```

The function `fromInt` takes one `Int` value and returns an `Int32` value.

```
import Int32 "mo:base/Int32";  
  
let integer : Int = -21474;  
  
Int32.fromInt(integer);
```

Int32.fromIntWrap

```
func fromIntWrap(i : Int) : Int32
```

The function `fromIntWrap` takes one `Int` value and returns an `Int32` value.

```
import Int32 "mo:base/Int32";  
  
let integer : Int = 4294967295;  
  
Int32.fromIntWrap(integer);
```

Int32.fromNat32

```
func fromNat32(i : Nat32) : Int32
```

The function `fromNat32` takes one `Nat32` value and returns an `Int32` value.

```
import Int32 "mo:base/Int32";  
  
let nat32 : Nat32 = 4294967295;  
  
Int32.fromNat32(nat32);
```

Int32.toNat32

```
func toNat32(i : Int32) : Nat32
```

The function `toNat32` takes one `Int32` value and returns an `Nat32` value.

```
import Int32 "mo:base/Int32";  
  
let int32 : Int32 = -967296;  
  
Int32.toNat32(int32);
```

Int32.min

```
func min(x : Int32, y : Int32) : Int32
```

The function `min` takes two `Int32` value and returns a `Int32` value.

```
import Int32 "mo:base/Int32";  
  
let x : Int32 = 2000;  
let y : Int32 = 1001;  
  
Int32.min(x, y);
```

Int32.max

```
func max(x : Int32, y : Int32) : Int32
```

The function `max` takes two `Int32` value and returns a `Int32` value.

```
import Int32 "mo:base/Int32";  
  
let x : Int32 = 2000;  
let y : Int32 = 2001;  
  
Int32.max(x, y);
```

Int32.equal

```
func equal(x : Int32, y : Int32) : Bool
```

The function `equal` takes two `Int32` value and returns a `Bool` value.

```
import Int32 "mo:base/Int32";  
  
let x : Int32 = 1000;  
let y : Int32 = 100;  
  
Int32.equal(x, y);
```

Int32.notEqual

```
func notEqual(x : Int32, y : Int32) : Bool
```

The function `notEqual` takes two `Int32` value and returns a `Bool` value.

```
import Int32 "mo:base/Int32";  
  
let x : Int32 = 1000;  
let y : Int32 = 1001;  
  
Int32.notEqual(x, y);
```

Int32.less

```
func less(x : Int32, y : Int32) : Bool
```

The function `less` takes two `Int32` value and returns a `Bool` value.

```
import Int32 "mo:base/Int32";  
  
let x : Int32 = 2000;  
let y : Int32 = 2500;  
  
Int32.less(x, y);
```

Int32.lessOrEqual

```
func lessOrEqual(x : Int32, y : Int32) : Bool
```

The function `lessOrEqual` takes two `Int32` value and returns a `Bool` value.

```
import Int32 "mo:base/Int32";  
  
let x : Int32 = 2000;  
let y : Int32 = 2500;  
  
Int32.lessOrEqual(x, y);
```

Int32.greater

```
func greater(x : Int32, y : Int32) : Bool
```

The function `greater` takes two `Int32` value and returns a `Bool` value.

```
import Int32 "mo:base/Int32";  
  
let x : Int32 = 2000;  
let y : Int32 = 1000;  
  
Int32.greater(x, y);
```

Int32.greaterOrEqual

```
func greaterOrEqual(x : Int32, y : Int32) : Bool
```

The function `greaterOrEqual` takes two `Int32` value and returns a `Bool` value.

```
import Int32 "mo:base/Int32";  
  
let x : Int32 = 2000;  
let y : Int32 = 1000;  
  
Int32.greaterOrEqual(x, y);
```

Int32.compare

```
func compare(x : Int32, y : Int32) : Bool
```

The function `compare` takes two `Int32` value and returns an `Order` variant value.

```
import Int32 "mo:base/Int32";  
  
let x : Int32 = 10000;  
let y : Int32 = 9999;  
  
Int32.compare(x, y);
```

Int32.abs

```
func abs(x : Int32) : Int32
```

The function `abs` takes one `Int32` value and returns a `Int32` value.

```
import Int32 "mo:base/Int32";  
  
let x : Int32 = -40;  
  
Int32.abs(x);
```

Int32.neg

```
func neg(x : Int32) : Int32
```

The function `neg` takes one `Int32` value and returns a `Int32` value.

```
import Int8 "mo:base/Int8";  
  
let x : Int8 = -50;  
  
Int8.neg(x);
```

Int32.add

```
func add(x : Int32, y : Int32) : Int32
```

The function `add` takes two `Int32` value and returns a `Int32` value.

```
import Int32 "mo:base/Int32";  
  
let x : Int32 = 2000;  
let y : Int32 = 1200;  
  
Int32.add(x, y);
```

Int32.sub

```
func sub(x : Int32, y : Int32) : Int32
```

The function `sub` takes two `Int32` value and returns a `Int32` value.

```
import Int32 "mo:base/Int32";  
  
let x : Int32 = 4000;  
let y : Int32 = 3999;  
  
Int32.sub(x, y);
```

Int32.mul

```
func mul(x : Int32, y : Int32) : Int32
```

The function `mul` takes two `Int32` value and returns a `Int32` value.

```
import Int32 "mo:base/Int32";  
  
let x : Int32 = 200;  
let y : Int32 = 50;  
  
Int32.mul(x, y);
```

Int32.div

```
func div(x : Int32, y : Int32) : Int32
```

The function `div` takes two `Int32` value and returns a `Int32` value.

```
import Int32 "mo:base/Int32";  
  
let x : Int32 = 5000;  
let y : Int32 = 500;  
  
Int32.div(x, y);
```

Int32.rem

```
func rem(x : Int32, y : Int32) : Int32
```

The function `rem` takes two `Int32` value and returns a `Int32` value.

```
import Int32 "mo:base/Int32";  
  
let x : Int32 = 4000;  
let y : Int32 = 1200;  
  
Int32.rem(x, y);
```

Int32.pow

```
func pow(x : Int32, y : Int32) : Int32
```

The function `pow` takes two `Int32` value and returns a `Int32` value.

```
import Int32 "mo:base/Int32";  
  
let x : Int32 = 10;  
let y : Int32 = 5;  
  
Int32.pow(x, y);
```

Int32.bitnot

```
func bitnot(x : Int32) : Int32
```

The function `bitnot` takes one `Int32` value and returns a `Int32` value.

```
import Int32 "mo:base/Int32";

let x : Int32 = 255; // Binary : 00000000_00000000_00000000_11111111
Int32.bitnot(x) // Binary : 11111111_11111111_11111111_00000000
```

Int32.bitand

```
func bitand(x : Int32, y : Int32) : Int32
```

The function `bitand` takes two `Int32` value and returns a `Int32` value.

```
import Int32 "mo:base/Int32";

let x : Int32 = 255; // Binary : 00000000_00000000_00000000_11111111
let y : Int32 = 255; // Binary : 00000000_00000000_00000000_11111111
Int32.bitand(x, y) // Binary : 00000000_00000000_00000000_11111111
```

Int32.bitor

```
func bitor(x : Int32, y : Int32) : Int32
```

The function `bitor` takes two `Int32` value and returns a `Int32` value.

```
import Int32 "mo:base/Int32";

let x : Int32 = 240; // Binary : 00000000_00000000_00000000_11110000
let y : Int32 = 15; // Binary : 00000000_00000000_00000000_00001111
Int32.bitor(x, y) // Binary : 00000000_00000000_00000000_11111111
```

Int32.bitxor

```
func bitxor(x : Int32, y : Int32) : Int32
```

The function `bitxor` takes two `Int32` value and returns a `Int32` value.

```
import Int32 "mo:base/Int32";

let x : Int32 = 255; // Binary : 00000000_00000000_00000000_11111111
let y : Int32 = 240; // Binary : 00000000_00000000_00000000_11110000

Int32.bitxor(x, y) // Binary : 00000000_00000000_00000000_00001111
```

Int32.bitshiftLeft

```
func bitshiftLeft(x : Int32, y : Int32) : Int32
```

The function `bitshiftLeft` takes two `Int32` value and returns a `Int32` value.

```
import Int32 "mo:base/Int32";

let x : Int32 = 15; // Binary : 00000000_00000000_00000000_00001111
let y : Int32 = 4;

Int32.bitshiftLeft(x, y) // Binary : 00000000_00000000_00000000_11110000
```

Int32.bitshiftRight

```
func bitshiftRight(x : Int32, y : Int32) : Int32
```

The function `bitshiftRight` takes two `Int32` value and returns a `Int32` value.

```
import Int32 "mo:base/Int32";

let x : Int32 = 240; // Binary : 00000000_00000000_00000000_11110000
let y : Int32 = 4;

Int32.bitshiftRight(x, y) // Binary : 00000000_00000000_00000000_00001111
```

Int32.bitrotLeft

```
func bitrotLeft(x : Int32, y : Int32) : Int32
```

The function `bitrotLeft` takes two `Int32` value and returns a `Int32` value.

```
import Int32 "mo:base/Int32";

let x : Int32 = 2_147_483_647; // Binary : 01111111_11111111_11111111_11111111
let y : Int32 = 1;

Int32.bitrotLeft(x, y) // Binary : 11111111_11111111_11111111_11111110
```

Int32.bitrotRight

```
func bitrotRight(x : Int32, y : Int32) : Int32
```

The function `bitrotRight` takes two `Int32` value and returns a `Int32` value.

```
import Int32 "mo:base/Int32";

let x : Int32 = 2_147_483_647; // Binary : 01111111_11111111_11111111_11111111
let y : Int32 = 1;

Int32.bitrotRight(x, y) // Binary : 10111111_11111111_11111111_11111111
```

Int32.bittest

```
func bittest(x : Int32, p : Nat) : Bool
```

The function `bittest` takes one `Int32` and one `Nat` value and returns a `Int32` value.

```
import Int32 "mo:base/Int32";

let x : Int32 = 255; // Binary : 00000000_00000000_00000000_11111111
let p : Nat = 7;

Int32.bittest(x, p)
```

Int32.bitset

```
func bitset(x : Int32, p : Nat) : Bool
```

The function `bitset` takes one `Int32` and one `Nat` value and returns a `Int32` value.

```
import Int32 "mo:base/Int32";

let x : Int32 = 127; // Binary : 00000000_00000000_00000000_01111111
let p : Nat = 7;

Int32bitset(x, p) // Binary : 00000000_00000000_00000000_11111111
```

Int32.bitclear

```
func bitclear(x : Int32, p : Nat) : Bool
```

The function `bitclear` takes one `Int32` and one `Nat` value and returns a `Int32` value.

```
import Int32 "mo:base/Int32";

let x : Int32 = 255; // Binary : 00000000_00000000_00000000_11111111
let p : Nat = 7;

Int32bitclear(x, p) // Binary : 00000000_00000000_00000000_01111111
```

Int32.bitflip

```
func bitflip(x : Int32, p : Nat) : Bool
```

The function `bitflip` takes one `Int32` and one `Nat` value and returns a `Int32` value.

```
import Int32 "mo:base/Int32";

let x : Int32 = 255; // Binary : 00000000_00000000_00000000_11111111
let p : Nat = 7;

Int32bitflip(x, p) // Binary : 00000000_00000000_00000000_01111111
```

Int32.bitcountNonZero

```
let bitcountNonZero : (x : Int32) -> Int32
```

The function `bitcountNonZero` takes one `Int32` value and returns a `Int32` value.

```
import Int32 "mo:base/Int32";  
  
let x : Int32 = 255; // Binary : 00000000_00000000_00000000_11111111  
  
Int32.bitcountNonZero(x)
```

Int32.bitcountLeadingZero

```
let bitcountLeadingZero : (x : Int32) -> Int32
```

The function `bitcountLeadingZero` takes one `Int32` value and returns a `Int32` value.

```
import Int32 "mo:base/Int32";  
  
let x : Int32 = 255; // Binary : 00000000_00000000_00000000_11111111  
  
Int32.bitcountLeadingZero(x)
```

Int32.bitcountTrailingZero

```
let bitcountTrailingZero : (x : Int32) -> Int32
```

The function `bitcountTrailingZero` takes one `Int32` value and returns a `Int32` value.

```
import Int32 "mo:base/Int32";  
  
let x : Int32 = 128; // Binary : 00000000_00000000_00000000_10000000  
  
Int32.bitcountTrailingZero(x)
```

Int32.addWrap

```
func addWrap(x : Int32, y : Int32) : Int32
```

The function `addWrap` takes two `Int32` value and returns a `Int32` value. It is equivalent to the `+%` Bitwise operators.

```
import Int32 "mo:base/Int32";  
  
let x : Int32 = 2_147_483_647;  
let y : Int32 = 1;  
  
Int32.addWrap(x, y)
```

Int32.subWrap

```
func subWrap(x : Int32, y : Int32) : Int32
```

The function `subWrap` takes two `Int32` value and returns a `Int32` value. It is equivalent to the `-%` Bitwise operators.

```
import Int32 "mo:base/Int32";  
  
let x : Int32 = -2_147_483_648;  
let y : Int32 = 1;  
  
Int32.subWrap(x, y)
```

Int32.mulWrap

```
func mulWrap(x : Int32, y : Int32) : Int32
```

The function `mulWrap` takes two `Int32` value and returns a `Int32` value. It is equivalent to the `*%` Bitwise operators.

```
import Int32 "mo:base/Int32";

let x : Int32 = 2_147_483_647;
let y : Int32 = 2;

Int32.mulWrap(x, y)
```

Int32.powWrap

```
func powWrap(x : Int32, y : Int32) : Int32
```

The function `powWrap` takes two `Int32` value and returns a `Int32` value. It is equivalent to the `**%` [Bitwise operators](#).

```
import Int32 "mo:base/Int32";

let x : Int32 = 65536;
let y : Int32 = 2;

Int32.powWrap(x, y)
```

Int64

The *convention* is to name the *module alias* after the *file name* it is defined in:

```
import Int64 "mo:base/Int64";
```

On this page

Constants

```
Value minimumValue  
Value maximumValue
```

Conversion

```
Function toInt  
Function toText  
Function fromInt  
Function fromIntWrap  
Function fromNat64  
Function toNat64
```

Comparison

```
Function min  
Function max  
Function equal  
Function notEqual  
Function less  
Function lessOrEqual  
Function greater  
Function greaterOrEqual  
Function compare
```

Numerical Operations

```
Function abs
Function neg
Function add
Function sub
Function mul
Function div
Function rem
Function pow
```

Bitwise Operators

```
Function bitnot
Function bitand
Function bitor
Function bitxor
Function bitshiftLeft
Function bitshiftRight
Function bitrotLeft
Function bitrotRight
Function bittest
Function bitset
Function bitclear
Function bitflip
Function bitcountNonZero
Function bitcountLeadingZero
Function bitcountTrailingZero
```

Wrapping Operations

```
Function addWrap
Function subWrap
Function mulWrap
Function powWrap
```

minimumValue

```
let minValue : Int64 = -9_223_372_036_854_775_808;
```

maximumValue

```
let maxValue : Int64 = 9_223_372_036_854_775_807;
```

Int64.toInt

```
func toInt(i : Int64) : Int
```

The function `toInt` takes one `Int64` value and returns an `Int` value.

```
import Int64 "mo:base/Int64";  
  
let a : Int64 = -92233;  
  
Int64.toInt(a);
```

Int64.toText

```
func toText(i : Int64) : Text
```

The function `toText` takes one `Int64` value and returns a `Text` value.

```
import Int64 "mo:base/Int64";  
  
let b : Int64 = -92233;  
  
Int64.toText(b);
```

Int64.fromInt

```
func fromInt(i : Int) : Int64
```

The function `fromInt` takes one `Int` value and returns an `Int64` value.

```
import Int64 "mo:base/Int64";  
  
let integer : Int = -92233;  
  
Int64.fromInt(integer);
```

Int64.fromIntWrap

```
func fromIntWrap(i : Int) : Int64
```

The function `fromIntWrap` takes one `Int` value and returns an `Int64` value.

```
import Int64 "mo:base/Int64";  
  
let integer : Int = 18_446_744_073_709_551_615;  
  
Int64.fromIntWrap(integer);
```

Int64.fromNat64

```
func fromNat64(i : Nat64) : Int64
```

The function `fromNat64` takes one `Nat64` value and returns an `Int64` value.

```
import Int64 "mo:base/Int64";  
  
let nat64 : Nat64 = 18_446_744_073_709_551_615;  
  
Int64.fromNat64(nat64);
```

Int64.toNat64

```
func toNat64(i : Int64) : Nat64
```

The function `toNat64` takes one `Int64` value and returns an `Nat64` value.

```
import Int64 "mo:base/Int64";  
  
let int64 : Int64 = -9551616;  
  
Int64.toNat64(int64);
```

Int64.min

```
func min(x : Int64, y : Int64) : Int64
```

The function `min` takes two `Int64` value and returns a `Int64` value.

```
import Int64 "mo:base/Int64";  
  
let x : Int64 = 2000;  
let y : Int64 = 1001;  
  
Int64.min(x, y);
```

Int64.max

```
func max(x : Int64, y : Int64) : Int64
```

The function `max` takes two `Int64` value and returns a `Int64` value.

```
import Int64 "mo:base/Int64";  
  
let x : Int64 = 2000;  
let y : Int64 = 2001;  
  
Int64.max(x, y);
```

Int64.equal

```
func equal(x : Int64, y : Int64) : Bool
```

The function `equal` takes two `Int64` value and returns a `Bool` value.

```
import Int64 "mo:base/Int64";  
  
let x : Int64 = 1000;  
let y : Int64 = 100;  
  
Int64.equal(x, y);
```

Int64.notEqual

```
func notEqual(x : Int64, y : Int64) : Bool
```

The function `notEqual` takes two `Int64` value and returns a `Bool` value.

```
import Int64 "mo:base/Int64";  
  
let x : Int64 = 1000;  
let y : Int64 = 1001;  
  
Int64.notEqual(x, y);
```

Int64.less

```
func less(x : Int64, y : Int64) : Bool
```

The function `less` takes two `Int64` value and returns a `Bool` value.

```
import Int64 "mo:base/Int64";  
  
let x : Int64 = 2000;  
let y : Int64 = 2500;  
  
Int64.less(x, y);
```

Int64.lessOrEqual

```
func lessOrEqual(x : Int64, y : Int64) : Bool
```

The function `lessOrEqual` takes two `Int64` value and returns a `Bool` value.

```
import Int64 "mo:base/Int64";  
  
let x : Int64 = 2000;  
let y : Int64 = 2500;  
  
Int64.lessOrEqual(x, y);
```

Int64.greater

```
func greater(x : Int64, y : Int64) : Bool
```

The function `greater` takes two `Int64` value and returns a `Bool` value.

```
import Int64 "mo:base/Int64";  
  
let x : Int64 = 2000;  
let y : Int64 = 1000;  
  
Int64.greater(x, y);
```

Int64.greaterOrEqual

```
func greaterOrEqual(x : Int64, y : Int64) : Bool
```

The function `greaterOrEqual` takes two `Int64` value and returns a `Bool` value.

```
import Int64 "mo:base/Int64";  
  
let x : Int64 = 2000;  
let y : Int64 = 1000;  
  
Int64.greaterOrEqual(x, y);
```

Int64.compare

```
func compare(x : Int64, y : Int64) : Bool
```

The function `compare` takes two `Int64` value and returns an `Order` variant value.

```
import Int64 "mo:base/Int64";  
  
let x : Int64 = 10000;  
let y : Int64 = 9999;  
  
Int64.compare(x, y);
```

Int64.abs

```
func abs(x : Int64) : Int64
```

The function `abs` takes one `Int64` value and returns a `Int64` value.

```
import Int8 "mo:base/Int8";  
  
let x : Int8 = -40;  
  
Int8.abs(x);
```

Int64.neg

```
func neg(x : Int64) : Int64
```

The function `neg` takes one `Int64` value and returns a `Int64` value.

```
import Int8 "mo:base/Int8";  
  
let x : Int8 = -50;  
  
Int8.neg(x);
```

Int64.add

```
func add(x : Int64, y : Int64) : Int64
```

The function `add` takes two `Int64` value and returns a `Int64` value.

```
import Int64 "mo:base/Int64";  
  
let x : Int64 = 2000;  
let y : Int64 = 1200;  
  
Int64.add(x, y);
```

Int64.sub

```
func sub(x : Int64, y : Int64) : Int64
```

The function `sub` takes two `Int64` value and returns a `Int64` value.

```
import Int64 "mo:base/Int64";  
  
let x : Int64 = 4000;  
let y : Int64 = 3999;  
  
Int64.sub(x, y);
```

Int64.mul

```
func mul(x : Int64, y : Int64) : Int64
```

The function `mul` takes two `Int64` value and returns a `Int64` value.

```
import Int64 "mo:base/Int64";  
  
let x : Int64 = 200;  
let y : Int64 = 50;  
  
Int64.mul(x, y);
```

Int64.div

```
func div(x : Int64, y : Int64) : Int64
```

The function `div` takes two `Int64` value and returns a `Int64` value.

```
import Int64 "mo:base/Int64";  
  
let x : Int64 = 5000;  
let y : Int64 = 500;  
  
Int64.div(x, y);
```

Int64.rem

```
func rem(x : Int64, y : Int64) : Int64
```

The function `rem` takes two `Int64` value and returns a `Int64` value.

```
import Int64 "mo:base/Int64";  
  
let x : Int64 = 4000;  
let y : Int64 = 1200;  
  
Int64.rem(x, y);
```

Int64.pow

```
func pow(x : Int64, y : Int64) : Int64
```

The function `pow` takes two `Int64` value and returns a `Int64` value.

```
import Int64 "mo:base/Int64";  
  
let x : Int64 = 10;  
let y : Int64 = 5;  
  
Int64.pow(x, y);
```

Int64.bitnot

```
func bitnot(x : Int64) : Int64
```

The function `bitnot` takes one `Int64` value and returns a `Int64` value.

```
import Int64 "mo:base/Int64";  
  
let x : Int64 = 9_223_372_036_854_775_807;  
  
Int64.bitnot(x)
```

Int64.bitand

```
func bitand(x : Int64, y : Int64) : Int64
```

The function `bitand` takes two `Int64` value and returns a `Int64` value.

```
import Int64 "mo:base/Int64";  
  
let x : Int64 = 4294967295;  
let y : Int64 = 255;  
  
Int64.bitand(x, y)
```

Int64.bitor

```
func bitor(x : Int64, y : Int64) : Int64
```

The function `bitor` takes two `Int64` value and returns a `Int64` value.

```
import Int64 "mo:base/Int64";  
  
let x : Int64 = 240;  
let y : Int64 = 15;  
  
Int64.bitor(x, y)
```

Int64.bitxor

```
func bitxor(x : Int64, y : Int64) : Int64
```

The function `bitxor` takes two `Int64` value and returns a `Int64` value.

```
import Int64 "mo:base/Int64";  
  
let x : Int64 = 255;  
let y : Int64 = 240;  
  
Int64.bitxor(x, y)
```

Int64.bitshiftLeft

```
func bitshiftLeft(x : Int64, y : Int64) : Int64
```

The function `bitshiftLeft` takes two `Int64` value and returns a `Int64` value.

```
import Int64 "mo:base/Int64";  
  
let x : Int64 = 15;  
let y : Int64 = 4;  
  
Int64.bitshiftLeft(x, y)
```

Int64.bitshiftRight

```
func bitshiftRight(x : Int64, y : Int64) : Int64
```

The function `bitshiftRight` takes two `Int64` value and returns a `Int64` value.

```
import Int64 "mo:base/Int64";  
  
let x : Int64 = 240;  
let y : Int64 = 4;  
  
Int64.bitshiftRight(x, y)
```

Int64.bitrotLeft

```
func bitrotLeft(x : Int64, y : Int64) : Int64
```

The function `bitrotLeft` takes two `Int64` value and returns a `Int64` value.

```
import Int64 "mo:base/Int64";  
  
let x : Int64 = 9_223_372_036_854_775_807;  
let y : Int64 = 1;  
  
Int64.bitrotLeft(x, y)
```

Int64.bitrotRight

```
func bitrotRight(x : Int64, y : Int64) : Int64
```

The function `bitrotRight` takes two `Int64` value and returns a `Int64` value.

```
import Int64 "mo:base/Int64";  
  
let x : Int64 = 9_223_372_036_854_775_807;  
let y : Int64 = 1;  
  
Int64.bitrotRight(x, y)
```

Int64.bittest

```
func bittest(x : Int64, p : Nat) : Bool
```

The function `bittest` takes one `Int64` and one `Nat` value and returns a `Int64` value.

```
import Int64 "mo:base/Int64";  
  
let x : Int64 = 255;  
let p : Nat = 7;  
  
Int64.bittest(x, p)
```

Int64.bitset

```
func bitset(x : Int64, p : Nat) : Bool
```

The function `bitset` takes one `Int64` and one `Nat` value and returns a `Int64` value.

```
import Int64 "mo:base/Int64";  
  
let x : Int64 = 127;  
let p : Nat = 7;  
  
Int64.bitset(x, p)
```

Int64.bitclear

```
func bitclear(x : Int64, p : Nat) : Bool
```

The function `bitclear` takes one `Int64` and one `Nat` value and returns a `Int64` value.

```
import Int64 "mo:base/Int64";  
  
let x : Int64 = 255;  
let p : Nat = 7;  
  
Int64.bitclear(x, p)
```

Int64.bitflip

```
func bitflip(x : Int64, p : Nat) : Bool
```

The function `bitflip` takes one `Int64` and one `Nat` value and returns a `Int64` value.

```
import Int64 "mo:base/Int64";  
  
let x : Int64 = 255;  
let p : Nat = 7;  
  
Int64.bitflip(x, p)
```

Int64.bitcountNonZero

```
let bitcountNonZero : (x : Int64) -> Int64
```

The function `bitcountNonZero` takes one `Int64` value and returns a `Int64` value.

```
import Int64 "mo:base/Int64";  
  
let x : Int64 = 255;  
  
Int64.bitcountNonZero(x)
```

Int64.bitcountLeadingZero

```
let bitcountLeadingZero : (x : Int64) -> Int64
```

The function `bitcountLeadingZero` takes one `Int64` value and returns a `Int64` value.

```
import Int64 "mo:base/Int64";  
  
let x : Int64 = 255;  
  
Int64.bitcountLeadingZero(x)
```

Int64.bitcountTrailingZero

```
let bitcountTrailingZero : (x : Int64) -> Int64
```

The function `bitcountTrailingZero` takes one `Int64` value and returns a `Int64` value.

```
import Int64 "mo:base/Int64";  
  
let x : Int64 = 128;  
  
Int64.bitcountTrailingZero(x)
```

Int64.addWrap

```
func addWrap(x : Int64, y : Int64) : Int64
```

The function `addWrap` takes two `Int64` value and returns a `Int64` value. It is equivalent to the `+%` Bitwise operators.

```
import Int64 "mo:base/Int64";  
  
let x : Int64 = 9_223_372_036_854_775_807;  
let y : Int64 = 1;  
  
Int64.addWrap(x, y)
```

Int64.subWrap

```
func subWrap(x : Int64, y : Int64) : Int64
```

The function `subWrap` takes two `Int64` value and returns a `Int64` value. It is equivalent to the `-%` Bitwise operators.

```
import Int64 "mo:base/Int64";  
  
let x : Int64 = -9_223_372_036_854_775_808;  
let y : Int64 = 1;  
  
Int64.subWrap(x, y)
```

Int64.mulWrap

```
func mulWrap(x : Int64, y : Int64) : Int64
```

The function `mulWrap` takes two `Int64` value and returns a `Int64` value. It is equivalent to the `*%` Bitwise operators.

```
import Int64 "mo:base/Int64";

let x : Int64 = 9_223_372_036_854_775_807;
let y : Int64 = 2;

Int64.mulWrap(x, y)
```

Int64.powWrap

```
func powWrap(x : Int64, y : Int64) : Int64
```

The function `powWrap` takes two `Int64` value and returns a `Int64` value. It is equivalent to the `**%` [Bitwise operators](#).

```
import Int64 "mo:base/Int64";

let x : Int64 = 4294967296;
let y : Int64 = 2;

Int64.powWrap(x, y)
```

Blob

A blob is an immutable sequences of bytes. Blobs are similar to arrays of bytes [Nat8] .

The *convention* is to name the *module alias* after the *file name* it is defined in:

```
import Blob "mo:base/Blob";
```

Conversion

```
Function fromArray  
Function toArray  
Function toArrayMut  
Function fromArrayMut
```

Utility Function

```
Function hash
```

Comparison

```
Function compare  
Function equal  
Function notEqual  
Function less  
Function lessOrEqual  
Function greater  
Function greaterOrEqual
```

Blob.fromArray

```
func fromArray(bytes : [Nat8]) : Blob
```

```
import Blob "mo:base/Blob";  
  
let a : [Nat8] = [253, 254, 255];  
  
Blob.fromArray(a);
```

Blob.toArray

```
func toArray(blob : Blob) : [Nat8]  
  
import Blob "mo:base/Blob";  
  
let blob : Blob = "\0a\0b\0c" : Blob;  
  
Blob.toArray(blob);
```

Blob.toArrayMut

```
func toArrayMut(blob : Blob) : [var Nat8]  
  
import Blob "mo:base/Blob";  
  
let blob : Blob = "\0a\0b\0c" : Blob;  
  
Blob.toArrayMut(blob);
```

Blob.fromArrayMut

```
func fromArrayMut(bytes : [var Nat8]) : Blob  
  
import Blob "mo:base/Blob";  
  
let a : [var Nat8] = [var 253, 254, 255];  
  
Blob.fromArrayMut(a);
```

Blob.hash

```
func hash(blob : Blob) : Nat32
```

The function `hash` takes one `Blob` value and returns a `Nat32` value.

```
import Blob "mo:base/Blob";

let blob : Blob = "\00\ff\00" : Blob;

Blob.hash(blob);
```

Blob.compare

```
func compare(blob1 : Blob, blob2 : Blob) : {#less; #equal; #greater}
```

The function `compare` takes two `Blob` value and returns an `Order` value.

```
import Blob "mo:base/Blob";

let blob1 : Blob = "\00\ef\00" : Blob;
let blob2 : Blob = "\00\ff\00" : Blob;

Blob.compare(blob1, blob2);
```

Blob.equal

```
func equal(blob1 : Blob, blob2 : Blob) : Bool
```

The function `equal` takes two `Blob` value and returns a `Bool` value.

```
import Blob "mo:base/Blob";

let blob1 : Blob = "\00\ff\00" : Blob;
let blob2 : Blob = "\00\ff\00" : Blob;

Blob.equal(blob1, blob2);
```

Blob.notEqual

```
func notEqual(blob1 : Blob, blob2 : Blob) : Bool
```

The function `notEqual` takes two `Blob` value and returns a `Bool` value.

```
import Blob "mo:base/Blob";  
  
let blob1 : Blob = "\00\ff\00" : Blob;  
let blob2 : Blob = "\00\ef\00" : Blob;  
  
Blob.notEqual(blob1, blob2);
```

Blob.less

```
func less(blob1 : Blob, blob2 : Blob) : Bool
```

The function `less` takes two `Blob` value and returns a `Bool` value.

```
import Blob "mo:base/Blob";  
  
let blob1 : Blob = "\00\ef\00" : Blob;  
let blob2 : Blob = "\00\ff\00" : Blob;  
  
Blob.less(blob1, blob2);
```

Blob.lessOrEqual

```
func lessOrEqual(blob1 : Blob, blob2 : Blob) : Bool
```

The function `lessOrEqual` takes two `Blob` value and returns a `Bool` value.

```
import Blob "mo:base/Blob";  
  
let blob1 : Blob = "\00\ef\00" : Blob;  
let blob2 : Blob = "\00\ff\00" : Blob;  
  
Blob.lessOrEqual(blob1, blob2);
```

Blob.greater

```
func greater(blob1 : Blob, blob2 : Blob) : Bool
```

The function `greater` takes two `Blob` value and returns a `Bool` value.

```
import Blob "mo:base/Blob";  
  
let blob1 : Blob = "\00\ff\00" : Blob;  
let blob2 : Blob = "\00\ef\00" : Blob;  
  
Blob.greater(blob1, blob2);
```

Blob.greaterOrEqual

```
func greaterOrEqual(blob1 : Blob, blob2 : Blob) : Bool
```

The function `greaterOrEqual` takes two `Blob` value and returns a `Bool` value.

```
import Blob "mo:base/Blob";  
  
let blob1 : Blob = "\00\ff\00" : Blob;  
let blob2 : Blob = "\00\ef\00" : Blob;  
  
Blob.greaterOrEqual(blob1, blob2);
```

Utility Modules

The Motoko Base Library provides several modules like [Result.mo](#) and [Iter.mo](#) for conveniently working with functions and data. Other modules expose IC system functionality like [Time.mo](#) and [Error.mo](#).

Iterators

The `Iter.mo` module provides useful *classes* and *public functions* for *iteration* on data sequences.

An iterator in Motoko is an `object` with a single method `next`. Calling `next` returns a `?T` where `T` is the type over which we are iterating. An `Iter` object is *consumed* by calling its `next` function until it returns `null`.

We can make an `Iter<T>` from any data sequence. Most data sequences in Motoko, like `Array`, `List` and `Buffer` provide functions to make an `Iter<T>`, which can be used to iterate over their values.

The type `Iter<T>` is an `object` type with one single function:

```
type Iter<T> = { next : () -> ?T }
```

The `next` function takes no arguments and returns `?T` where `T` is the type of the value being iterated over.

Example

```
import Array "mo:base/Array";

let a : [Nat] = [1, 2, 3];
let iter = Array.vals(a);

assert(?1 == iter.next());
assert(?2 == iter.next());
assert(?3 == iter.next());
assert(null == iter.next());
```

We use the `vals` function from the `Array.mo` module to make an `Iter<Nat>` from our array. We call its `next` function three times. After that the iterator is consumed and returns `null`.

Import

The *convention* is to name the *module alias* after the *file name* it is defined in:

```
import Iter "mo:base/Iter";
```

Ranges

This module includes two classes `range` and `revRange` to make ranges (and reversed ranges) of `Nat` or `Int` respectively. Ranges may be used in a for loop:

```
import Iter "mo:base/Iter";  
  
for (i in Iter.range(0,4)) {  
    // do something 5 times  
};  
  
for (i in Iter.revRange(4,0)) {  
    // do something 5 times  
};
```

We use the `in` keyword to iterate over the `Iter<Nat>` and bind the values to `i` in a for loop. The second for loop iterates over a `Iter<Int>`.

On this page

[Class range](#)

[Function next](#)

[Class revRange](#)

[Function next](#)

[Function iterate](#)

[Function size](#)

[Function map](#)

[Function filter](#)

[Function make](#)

[Function fromArray](#)

[Function fromArrayMut](#)

[Function fromList](#)

[Function toArray](#)

[Function toArrayMut](#)

[Function `toList`](#)[Function `sort`](#)

Class range

```
class range(x : Nat, y : Int)
```

range.next

```
func next() : ?Nat

import Iter "mo:base/Iter";

let myRange : Iter.Iter<Nat> = Iter.range(1, 3);

let num1 = myRange.next(); // returns 1
let num2 = myRange.next(); // returns 2
let num3 = myRange.next(); // returns 3
let num4 = myRange.next(); // returns null
```

Class revRange

```
class revRange(x : Int, y : Int)
```

revRange.next

```
func next() : ?Int
```

```
import Iter "mo:base/Iter";

let myRevRange = Iter.revRange(5, 8);

let number1 = myRevRange.next(); // returns 8
let number2 = myRevRange.next(); // returns 7
let number3 = myRevRange.next(); // returns 6
let number4 = myRevRange.next(); // returns 5
let number5 = myRevRange.next(); // returns null
```

Iter.iterate

```
func iterate<A>(
    xs : Iter<A>,
    f : (A, Nat) -> ()
) : ()
```

Parameters	
Generic parameters	A
Variable argument	xs : Iter<A>
Function argument	f : (A, Nat) -> ()
Return type	()

```
import Iter "mo:base/Iter";

let myRange : Iter.Iter<Nat> = Iter.range(1, 3);

var sum = 0;

func update(a : Nat, b : Nat) {
    sum += a;
};

Iter.iterate(myRange, update);
```

Iter.size

```
func size<A>(xs : Iter<A>) : Nat
```

Parameters	
Generic parameters	A
Variable argument	xs : Iter<A>
Return type	Nat

```
import Iter "mo:base/Iter";

let myRange : Iter.Iter<Nat> = Iter.range(1, 3);

Iter.size(myRange);
```

Iter.map

```
func map<A, B>(
    xs : Iter<A>,
    f : A -> B
) : Iter<B>
```

Parameters	
Generic parameter	A, B
Variable argument	xs : Iter<A>
Function argument	f : A -> B
Return type	Iter

```
import Iter "mo:base/Iter";

let myRange : Iter.Iter<Nat> = Iter.range(1, 3);

func change(n : Nat) : Int {
    n * -1;
}

let mapedIter = Iter.map<Nat, Int>(myRange, change);

Iter.toArray(mapedIter);
```

Iter.filter

```
func filter<A>(
    xs : Iter<A>,
    f : A -> Bool
) : Iter<A>
```

Parameters	
Generic parameters	A
Variable argument	xs : Iter<A>
Function argument	f : A -> Bool
Return type	Iter<A>

```
import Iter "mo:base/Iter";

let myRange : Iter.Iter<Nat> = Iter.range(1, 3);

func change(n : Nat) : Bool {
    n > 1;
};

let filterIter = Iter.filter<Nat>(myRange, change);

Iter.toArray(filterIter);
```

Iter.make

```
func make<A>(x : A) : Iter<A>
```

Parameters	
Generic parameters	A
Variable argument	x : A
Return type	Iter<A>

```
import Iter "mo:base/Iter";

let myRange : Iter.Iter<Nat> = Iter.make(3);

assert (?3 == myRange.next());
assert (?3 == myRange.next());
assert (?3 == myRange.next());
//.....
```

Iter.fromArray

```
func fromArray<A>(xs : [A]) : Iter<A>
```

Parameters	
Generic parameters	A
Variable argument	xs : [A]
Return type	Iter<A>

```
import Iter "mo:base/Iter";

let array = ["bitcoin", "ETH", "ICP"];

let myRange : Iter.Iter<Text> = Iter.fromArray(array);

Iter.size(myRange);
```

Iter.fromArrayMut

```
func fromArrayMut<A>(xs : [var A]) : Iter<A>
```

Parameters	
Generic parameters	A
Variable argument	xs : [var A]
Return type	Iter<A>

```
import Iter "mo:base/Iter";

let array = [var "bitcoin", "ETH", "ICP"];

let iter = Iter.fromArrayMut(array);

Iter.size(iter);
```

Iter.fromList

```
func fromList(xs : List<T>) : Iter
```

Parameters	
Variable argument	xs : List<T>
Return type	Iter

```
import List "mo:base/List";
import Iter "mo:base/Iter";

let list : List.List<Nat> = ?(0, ?(1, null));

let iter = Iter.fromList(list);

Iter.toArray(iter);
```

Iter.toArray

```
func toArray<A>(xs : Iter<A>) : [A]
```

Parameters	
Generic parameters	A
Variable argument	xs : Iter<A>
Return type	[A]

```
import Iter "mo:base/Iter";

let myRange : Iter.Iter<Nat> = Iter.range(1, 3);

Iter.toArray<Nat>(myRange);
```

Iter.toArray

```
func toArray<A>(xs : Iter<A>) : [var A]
```

Parameters	
Generic parameters	A
Variable argument	xs : Iter<A>
Return type	[var A]

```
import Iter "mo:base/Iter";

let myRange : Iter.Iter<Nat> = Iter.range(1, 3);

Iter.toArrayMut<Nat>(myRange);
```

Iter.toList

```
func list<A>(xs : Iter<A>) : List.List<A>
```

Parameters	
Generic parameters	A
Variable argument	xs : Iter<A>
Return type	List.List<A>

```
import Iter "mo:base/Iter";
import List "mo:base/List";

let myRange : Iter.Iter<Nat> = Iter.range(1, 3);

let list : List.List<Nat> = Iter.toList(myRange);

List.last(list);
```

Iter.sort

```
func sort<A>(
    xs : Iter<A>,
    compare : (A, A) -> Order.Order
) : Iter<A>
```

Parameters	
Generic parameters	A
Variable argument	xs : Iter<A>
Function argument	compare : (A, A) -> Order.Order
Return type	Iter<A>

```
import Iter "mo:base/Iter";
import Int "mo:base/Int";

let a : [Int] = [5, 2, -3, 1];

let i : Iter.Iter<Int> = a.vals();

let sorted : Iter.Iter<Int> = Iter.sort(i, Int.compare);

Iter.toArray(sorted)
```

Hash

See the [official docs](#) for an up to date reference of hashing functions.

Option

The [Option module](#) in the official reference contains several functions for working with Option types.

For an explanation of Options in Motoko see [this chapter](#) of this book..

Result

This is a subset of functions available in the [official reference](#) for working with `Result` variants. See [this chapter of this book](#) for more information on `Result`.

The *convention* is to name the *module alias* after the *file name* it is defined in:

```
import Result "mo:base/Result";
```

On this page

[Public type `Result<Ok, Err>`](#)

[Function `fromOption`](#)

[Function `toOption`](#)

[Function `isOk`](#)

[Function `isErr`](#)

Public type

The `Result<Ok, Err>` type is a [variant](#) with two [generic parameters](#). It can be used as the return type for functions to indicate either *success* or *error*. Both cases can be handled *programmatically*.

```
type Result<Ok, Err> = { #ok : Ok; #err : Err }
```

A `Result<Ok, Err>` could either be

- `#ok(x)` where `x` is of [generic type `Ok`](#)
- `#err(x)` where `x` is of [generic type `Err`](#)

We usually [import, rename](#) and instantiate `Result` with types for our own purpose and use it as the return type of a function.

```

import Result "mo:base/Result";

type MyResult = Result.Result<Nat, Text>;

func doSomething(b : Bool) : MyResult {
    switch (b) {
        case true #ok(0);
        case false #err("false");
    };
};

switch (doSomething(true)) {
    case (#ok(nat)) {};
    case (#err(text)) {};
};

}

```

We import `Result` and declare our own custom type `MyResult` by instantiating the `Result<Ok, Err>` with types `Nat` and `Text`.

Our function `doSomething` could either return a `#ok` with a `Nat` value or a `#err` with a `Text` value.

Both cases are handled *programmatically* in the last [switch expression](#). The return values associated with both cases are locally named `nat` and `text` and could be used inside the switch case body.

Result.fromOption

```

func fromOption<R, E>(x : ?R, err : E) : Result<R, E>

import Result "mo:base/Result";

Result.fromOption<Nat, Text>(?100, "error")

```

Result.toOption

```
func toOption<R, E>(r : Result<R, E>) : ?R
```

```
import Result "mo:base/Result";

let ok : Result.Result<Nat, Text> = #ok 100;

let opt : ?Nat = Result.toOption(ok);
```

Result.isOk

```
funcisOk(r : Result<Any, Any>) : Bool

import Result "mo:base/Result";

let ok : Result.Result<Nat, Text> = #ok 100;

Result.isOk(ok);
```

Result.isErr

```
func isErr(r : Result<Any, Any>) : Bool

import Result "mo:base/Result";

let err : Result.Result<Nat, Text> = #err "this is wrong";

Result.isErr(err);
```

Order

The *convention* is to name the *module alias* after the *file name* it is defined in:

```
import Order "mo:base/Order";
```

On this page

[Public type Order](#)

[Function isLess](#)

[Function isEqual](#)

[Function isGreater](#)

[Function equal](#)

Public Type

The `Order` variant type is used to represent three possible outcomes when comparing the *order* two values.

```
type Order = {  
    #less;  
    #equal;  
    #greater;  
};
```

When comparing the order of two values, we could either return:

- `#less` when the first value is less than the second value.
- `#equal` when both values are equal.
- `#greater` when the first value is greater than the second value.

Some types are naturally ordered like number types `Nat` and `Int`. But we may define an order for any type, even types for which there is no obvious natural order.

```

type Color = {
    #Red;
    #Blue;
};

func sortColor(c1 : Color, c2 : Color) : Order {
    switch ((c1, c2)) {
        case ((#Red, #Blue)) { #greater };
        case ((#Red, #Red)) { #equal };
        case ((#Blue, #Blue)) { #equal };
        case ((#Blue, #Red)) { #less };
    };
};

```

Here we define an order for our `Color` variant by defining a function that compares two values of `Color` and returns an `Order`. It treats `#Red` to be `#greater` than `#Blue`.

Order.isLess

```

func isLess(order : Order) : Bool

import Order "mo:base/Order";

let order : Order.Order = #less;

Order.isLess(order);

```

Order.isEqual

```

func isEqual(order : Order) : Bool

import Order "mo:base/Order";

let order : Order.Order = #less;

Order.isEqual(order);

```

Order.isGreater

```
func isGreater(order : Order) : Bool

import Order "mo:base/Order";

let order : Order.Order = #less;

Order.isGreater(order);
```

Order.equal

```
func equal(o1 : Order, o2 : Order) : Bool

import Order "mo:base/Order";

let icpToday : Order.Order = #less;

let icpTomorrow : Order.Order = #greater;

Order.equal(icpToday, icpTomorrow);
```

Error

See the [Error module](#) in the official reference and [Async Programming](#) for an example of working with Errors.

Debug

See the [Debug module](#) in the official reference and [Async Programming](#) for an example of working with Debug.

Data Structures

This chapter covers the most important *data structures* in Motoko.

Array

What is an array?

An *immutable* or *mutable* array of type `[T]` or `[var T]` is a sequence of values of type `T`. Each element can be accessed by its index, which is a `Nat` value that represents the position of the element in the array. Indexing starts at `0`. Some properties of arrays are:

- **Memory layout**

Arrays are stored in contiguous memory, meaning that all elements are stored next to each other in memory. This makes arrays more memory-efficient than some other data structures, like *lists*, where elements can be scattered throughout memory.

- **Fast indexing**

Arrays provide constant-time access to elements by index. This is because the memory address of any element can be calculated directly from its index and the starting memory address of the array.

- **Fixed size**

Once an array is created, its size cannot be changed. If you need to add or remove elements, you will have to create a new array of the desired size and copy the elements. This is different from other sequence data structures like *lists* or *buffers* that can grow or shrink dynamically.

- **Computational cost of copying**

Since arrays are stored in a contiguous block of memory, copying an array requires copying all its elements to a new memory location. This can be computationally expensive, especially for large arrays.

Import

The *convention* is to name the *module alias* after the *file name* it is defined in:

```
import Array "mo:base/Array";
```

Array.mo module public functions

Size

Function `size`

Initialization

Function `init`

Function `make`

Function `tabulate`

Function `tabulateVar`

Transformation

Function `freeze`

Function `thaw`

Function `sort`

Function `sortInPlace`

Function `reverse`

Function `flatten`

Comparison

Function `equal`

Mapping

Function `map`

Function `filter`

Function `mapEntries`

Function `mapFilter`

Function `mapResult`

Iteration

```
Function vals
Function keys
Function find
Function chain
Function foldLeft
Function foldRight
```

Array.size

The size of an array can be accessed by the `.size()` method:

```
let array : [Nat] = [0, 1, 2];
array.size()
```

The module also provides a dedicated function for the size of an array.

```
func size<X>(array : [X]) : Nat
```

Parameters	
Generic parameters	X
Variable argument	array : [X]
Return type	Nat

```
import Array "mo:base/Array";

let array : [Text] = ["ICP", "ETH", "USD", "Bitcoin"];

Array.size(array);
```

Array.init

Initialize a **mutable array** of type `[var X]` with a `size` and an initial value `initValue` of generic type `X`.

```
func init<X>(
    size : Nat,
    initialValue : X
) : [var X]
```

Parameters	
Generic parameters	X
Variable argument 1	size : Nat
Variable argument 2	initialValue : X
Return type	[var X]

```
import Array "mo:base/Array";

let size : Nat = 3;
let initialValue : Char = 'A';

let a : [var Char] = Array.init<Char>(size, initialValue);
```

Array.make

Make an **immutable** array with exactly one element of **generic type X**.

```
func make<X>(element : X) : [x]
```

Parameters	
Generic parameters	X
Variable argument	element : X
Return type	[X]

```
import Array "mo:base/Array";

let a : [Text] = Array.make<Text>("ICP");
```

Array.tabulate

The `tabulate` function generates an *immutable array* of generic type `X` and predefined `size` by using a generator function that takes the index of every element as an argument and produces the elements of the array.

```
func tabulate<X>(
    size : Nat,
    generator : Nat -> X
) : [X]
```

Parameters	
Generic parameters	<code>X</code>
Variable argument	<code>size : Nat</code>
Function argument	<code>generator : Nat -> X</code>
Return type	<code>[X]</code>

```
import Array "mo:base/Array";

let size : Nat = 3;

func generator(i : Nat) : Nat { i ** 2 };

let a : [Nat] = Array.tabulate<Nat>(size, generator);
```

Array.tabulateVar

The `tabulateVar` function generates a *mutable array* of generic type `X` and predefined `size` by using a generator function that takes the index of every element as an argument and produces the elements of the array.

```
func tabulateVar<X>(
    size : Nat,
    generator : Nat -> X
) : [var X]
```

Parameters	
Generic parameters	X
Variable argument	size : Nat
Function argument	generator : Nat -> X
Return type	[var X]

```
import Array "mo:base/Array";

let size : Nat = 3;

func generator(i : Nat) : Nat { i * 5 };

let a : [var Nat] = Array.tabulateVar<Nat>(size, generator);
```

Array.freeze

Freeze converts a **mutable array** of generic type X to a **immutable array** of the same type.

```
func freeze<X>(varArray : [var X]) : [X]
```

Parameters	
Generic parameters	X
Variable argument	varArray : [var X]
Return type	[X]

```
import Array "mo:base/Array";

let varArray : [var Text] = [var "apple", "banana", "cherry", "date",
"elderberry"];

varArray[1] := "kiwi";

let a : [Text] = Array.freeze<Text>(varArray);
```

Array.thaw

Thaw converts an **immutable array** of generic type X to a **mutable array** of the same type.

```
func thaw<X>(array : [X]) : [var X]
```

Parameters	
Generic parameters	X
Variable argument	array : [X]
Return type	[var X]

```
import Array "mo:base/Array";

let array : [Char] = ['h', 'e', 'l', 'l', 'o'];

let varArray : [var Char] = Array.thaw<Char>(array);
```

Array.sort

Sort takes an **immutable array** of **generic type X** and produces a second array which is sorted according to a comparing function **compare**. This comparing function compares two elements of type **X** and returns an **Order** type that is used to sort the array.

We could use a comparing function from the Base Library, like in the example below, or write our own custom comparing function, as long as its type is **(X, X) -> Order.Order**

```
func sort<X>(
    array : [X],
    compare : (X, X) -> Order.Order
) : [X]
```

Parameters	
Generic parameters	X
Variable argument	array : [X]
Function argument	compare : (X, X) -> Order.Order
Return type	[X]

```
import Array "mo:base/Array";
import Nat "mo:base/Nat";

let ages : [Nat] = [50, 20, 10, 30, 40];

let sortedAges : [Nat] = Array.sort<Nat>(ages, Nat.compare);
```

Index	ages : [Nat]	sortedAges : [Nat]
0	50	10
1	20	20
2	10	30
3	30	40
4	40	50

Array.sortInPlace

We can also 'sort in place', which behaves the same as `sort` except we mutate a **mutable array** instead of producing a new array. Note the function returns unit type `()`.

```
func sortInPlace<X>(
    array : [var X],
    compare : (X, X) -> Order.Order
) : ()
```

Parameters	
Generic parameters	X
Variable argument	array : [var X]
Function argument	compare : (X, X) -> Order.Order
Return type	()

```
import Array "mo:base/Array";
import Nat "mo:base/Nat";

var ages : [var Nat] = [var 50, 20, 10, 30, 40];

Array.sortInPlace<Nat>(ages, Nat.compare);
```

Index	ages : [var Nat]	ages : [var Nat]
0	50	10
1	20	20
2	10	30
3	30	40
4	40	50

Array.reverse

Takes an [immutable array](#) and produces a second array with elements in reversed order.

```
func reverse<X>(array : [X]) : [X]
```

Parameters	
Generic parameters	X
Variable argument	array : [X]
Return type	[X]

```
import Array "mo:base/Array";

let rank : [Text] = ["first", "second", "third"];

let reverse : [Text] = Array.reverse<Text>(rank);
```

Index	rank : [Text]	reverse : [Text]
0	"first"	"third"
1	"second"	"second"
2	"third"	"first"

Array.flatten

Takes an array of arrays and produces a single array, while retaining the original ordering of the elements.

```
func flatten<X>(arrays : [[X]]) : [X]
```

Parameters	
Generic parameters	X
Variable argument	arrays : [[X]]
Return type	[X]

```
import Array "mo:base/Array";

let arrays : [[Char]] = [['a', 'b'], ['c', 'd'], ['e']];
let newArray : [Char] = Array.flatten(arrays);
```

Index	arrays : [[Char]]	newArray : [Char]
0	['a', 'b']	'a'
1	['c', 'd']	'b'
2	['e']	'c'
3		'd'
4		'e'

Array.equal

Compare each element of two arrays and check whether they are all equal according to an `equal` function of type `(X, X) -> Bool`.

```
func equal<X>(
    array1 : [X],
    array2 : [X],
    equal : (X, X) -> Bool
) : Bool
```

Parameters	
Generic parameters	X
Variable argument1	array1 : [X]
Variable argument2	array2 : [X]
Function argument	equal : (X, X) -> Bool
Return type	Bool

```

import Array "mo:base/Array";

let class1 = ["Alice", "Bob", "Charlie", "David"];

let class2 = ["Alice", "Bob", "Charlie", "Eve"];

func nameEqual(name1 : Text, name2 : Text) : Bool {
    name1 == name2;
};

let isNameEqual : Bool = Array.equal<Text>(class1, class2, nameEqual);

```

Array.map

The mapping function `map` iterates through each element of an [immutable array](#), applies a given transformation function `f` to it, and creates a new array with the transformed elements. The input array is of [generic type](#) `[X]`, the transformation function takes elements of type `X` and returns elements of type `Y`, and the resulting array is of type `[Y]`.

```

func map<X>(
    array : [X],
    f : X -> Y
) : [Y]

```

Parameters	
Generic parameters	<code>X</code>
Variable argument	<code>array : [X]</code>
Function argument	<code>f : X -> Y</code>
Return type	<code>[Y]</code>

```

import Array "mo:base/Array";

let array1 : [Bool] = [true, false, true, false];

func convert(x : Bool) : Nat {
    if x return 1 else 0;
};

let array2 : [Nat] = Array.map<Bool, Nat>(array1, convert);

```

Index	array1 : [Bool]	array2 : [Nat]
0	true	1
1	false	0
2	true	1
3	false	0

Array.filter

The `filter` function takes an **immutable array** of elements of **generic type X** and a predicate function `predicate` (that takes a `X` and returns a `Bool`) and returns a new array containing only the elements that satisfy the predicate condition.

```
func filter<X>(
    array : [X],
    predicate : X -> Bool
) : [X]
```

Parameters	
Generic parameters	X
Variable argument	array : [X]
Function argument	predicate : X -> Bool
Return type	[X]

```
import Array "mo:base/Array";

let ages : [Nat] = [1, 2, 5, 6, 9, 7];

func isEven(x : Nat) : Bool {
    x % 2 == 0;
};

let evenAges : [Nat] = Array.filter<Nat>(ages, isEven);
```

Index	ages : [Nat]	evenAges : [Nat]
0	1	2
1	2	6
2	5	

Index	ages : [Nat]	evenAges : [Nat]
3	6	
4	9	
5	7	

Array.mapEntries

The `mapEntries` function takes an [immutable array](#) of elements of [generic type](#) `[X]` and a function `f` that accepts an element and its index (a `Nat` value) as arguments, then returns a new array of type `[Y]` with elements transformed by applying the function `f` to each element and its index.

```
func mapEntries<X,Y>(
    array : [X],
    f : (X, Nat) -> Y
) : [Y]
```

Parameters	
Generic parameters	X, Y
Variable argument	array : [X]
Function argument	f : (X, Nat) -> Y
Return type	[Y]

```
import Array "mo:base/Array";
import Nat "mo:base/Nat";
import Int "mo:base/Int";

let array1 : [Int] = [-1, -2, -3, -4];

func map(x : Int, y : Nat) : Text {
    Int.toText(x) # ";" # Nat.toText(y);
};

let array2 : [Text] = Array.mapEntries<Int, Text>(array1, map);
```

Index	array1 : [Int]	array2 : [Text]
0	-1	"-1; 0"
1	-2	"-2; 1"

Index	array1 : [Int]	array2 : [Text]
2	-3	"-3; 2"
3	-4	"-4; 3"

Array.mapFilter

```
func mapFilter<X,Y>(
    array : [X],
    f : X -> ?Y
) : [Y]
```

Parameters	
Generic parameters	X, Y
Variable argument	array : [X]
Function argument	f : X -> ?Y
Return type	[Y]

```
import Array "mo:base/Array";
import Nat "mo:base/Nat";

let array1 = [1, 2, 3, 4, 5];

func filter(x : Nat) : ?Text {
    if (x > 3) null else ?Nat.toText(100 / x);
};

let array2 = Array.mapFilter<Nat, Text>(array1, filter);
```

Index	array1 : [Nat]	array2 : [Text]
0	1	"100"
1	2	"50"
2	3	"33"
3	4	
3	5	

Array.mapResult

```
func mapResult<X, Y, E>(
    array : [X],
    f : X -> Result.Result<Y, E>
) : Result.Result<[Y], E>
```

Parameters	
Generic parameters	X, Y, E
Variable argument	array : [X]
Function argument	f : X -> Result.Result<Y, E>
Return type	Result.Result<[Y], E>

```
import Array "mo:base/Array";
import Result "mo:base/Result";

let array1 = [4, 5, 2, 1];

func check(x : Nat) : Result.Result<Nat, Text> {
    if (x == 0) #err "Cannot divide by zero" else #ok(100 / x);
};

let mapResult : Result.Result<[Nat], Text> = Array.mapResult<Nat, Nat, Text>
(array1, check);
```

Index	array1 : [Nat]	mapResult : #ok : [Nat]
0	4	25
1	5	20
2	2	50
3	1	100

Array.vals

```
func vals<X>(array : [X]) : I.Iter<X>
```

Parameters	
Generic parameters	X

Parameters	
Variable argument	array : [X]
Return type	I.Iter<X>

```
import Array "mo:base/Array";

let array = ["ICP", "will", "grow", "?"];

var sentence = "";

for (value in array.vals()) {
    sentence #= value # " ";
};

sentence
```

Array.keys

```
func keys<X>(array : [X]) : I.Iter<Nat>
```

Parameters	
Generic parameters	X
Variable argument	array : [X]
Return type	I.Iter<Nat>

```
import Array "mo:base/Array";
import Iter "mo:base/Iter";

let array = [true, false, true, false];

let iter = array.keys();

Iter.toArray(iter)
```

Array.find

```
func find<X>(
    array : [X],
    predicate : X -> Bool
) : ?X
```

Parameters	
Generic parameters	X
Variable argument	array : [X]
function argument	predicate : X -> Bool
Return type	?X

```
import Array "mo:base/Array";

let ages = [18, 25, 31, 37, 42, 55, 62];

func isGreaterThanOrEqual40(ages : Nat) : Bool {
    ages > 40;
};

let firstAgeGreaterThanOrEqual40 : ?Nat = Array.find<Nat>(ages, isGreaterThanOrEqual40);
```

Array.chain

```
func chain<X, Y>(
    array : [X],
    k : X -> [Y]
) : [Y]
```

Parameters	
Generic parameters	X, Y
Variable argument	array : [X]
Function argument	k : X -> [Y]
Return type	[Y]

```
import Array "mo:base/Array";

let array1 = [10, 20, 30];

func chain(x : Nat) : [Int] { [x, -x] };

let array2 : [Int] = Array.chain<Nat, Int>(array1, chain);
```

Index	array1 : [Nat]	array2 : [Int]
0	10	10
1	20	-10
2	30	20
3		-20
4		30
5		-30

Array.foldLeft

```
func foldLeft<X, A>(
    array : [X],
    base : A,
    combine : (A, X) -> A
) : A
```

Parameters	
Generic parameters	X, A
Variable argument1	array : [X]
Variable argument2	base : A
Function argument	combine : (A, X) -> A
Return type	A

```
import Array "mo:base/Array";

let array = [40, 20, 0, 10];

func add(a : Nat, b : Nat) : Nat {
    a + b;
};

let base : Nat = 30;

let sum : Nat = Array.foldLeft<Nat, Nat>(array, base, add);
```

Array.foldRight

```
func foldRight<X, A>(
    array : [X],
    base : A,
    combine : (X, A) -> A
) : A
```

Parameters	
Generic parameters	X, A
Variable argument1	array : [X]
Variable argument2	base : A
Function argument	combine : (X, A) -> A
Return type	A

```
import Array "mo:base/Array";
import Nat "mo:base/Nat";

let array1 = [7, 8, 1];

func concat(a : Nat, b : Text) : Text {
    b # Nat.toText(a);
};

let base : Text = "Numbers: ";

Array.foldRight<Nat, Text>(array1, base, concat);
```

List

The difference between a list and an [array](#) is that an array is stored as one *contiguous block of bytes* in memory and a list is 'scattered' around without the elements having to be adjacent to each other. The advantage is that we can use memory more efficiently by filling the memory more flexibly. The downside is that for operations on the whole list, we have to visit each element one by one which may be computationally expensive.

For more on the List data structures visit [recursive types](#).

The *convention* is to name the [module alias](#) after the [file name](#) it is defined in:

```
import List "mo:base/List";
```

On this page

- [Function nil](#)
- [Function isNil](#)
- [Function push](#)
- [Function last](#)
- [Function pop](#)
- [Function size](#)
- [Function get](#)
- [Function reverse](#)
- [Function iterate](#)
- [Function map](#)
- [Function filter](#)
- [Function partition](#)
- [Function mapFilter](#)
- [Function mapResult](#)
- [Function append](#)
- [Function flatten](#)
- [Function take](#)
- [Function drop](#)
- [Function foldLeft](#)
- [Function foldRight](#)
- [Function find](#)
- [Function some](#)
- [Function all](#)

```
Function merge
Function compare
Function equal
Function tabulate
Function make
Function replicate
Function zip
Function zipWith
Function split
Function chunks
Function fromArray
Function fromVarArray
Function toArray
Function toVarArray
Function toIter
```

List.nil

```
func nil<T>() : List<T>
```

parameters	
Generic parameters	T
Return type	List<T>

Example

```
import List "mo:base/List";

List.nil<Int>();
```

List.isNil

```
func isNil<T>(l : List<T>) : Bool
```

parameters	
Generic parameters	T
Variable argument	l : List<T>
Return type	Bool

Example

```
import List "mo:base/List";

let nil : List.List<Int> = List.nil<Int>();

List.isNil(nil);
```

List.push

```
func push<T>(
    x : T
    l : List<T>
) : List<T>
```

parameters	
Generic parameters	T
Variable argument1	x : T
Variable argument2	l : List<T>
Return type	List<T>

Example

```
import List "mo:base/List";

let nil : List.List<Int> = List.nil<Int>();

List.push(-1, nil);
```

List.last

```
func last<T>(l : List<T>) : ?T
```

parameters	
Generic parameters	T
Variable argument	l : List<T>
Return type	?T

Example

```
import List "mo:base/List";

let list : List.List<Int> = ?(0, ?(-1, null));

List.last(list);
```

List.pop

```
func pop<T>(l : List<T>) : (?T, List<T>)
```

parameters	
Generic parameters	T
Variable argument	l : List<T>
Return type	(?T, List<T>)

Example

```
import List "mo:base/List";

let list : List.List<Int> = ?(0, ?(-1, null));

List.pop(list);
```

List.size

```
func size<T>(l : List<T>) : Nat
```

parameters	
Generic parameters	T
Variable argument	l : List<T>
Return type	Nat

Example

```
import List "mo:base/List";

let list : List.List<Int> = ?(1, ?(0, ?(-1, null)));

List.size(list);
```

List.get

```
func get<T>(
    l : List<T>
    n : Nat
) : ?T
```

parameters	
Generic parameters	T
Variable argument1	l : List<T>
Variable argument2	n : Nat
Return type	?T

Example

```
import List "mo:base/List";

let list : List.List<Int> = ?(0, ?(-1, null));

List.get(list, 0);
```

List.reverse

```
func reverse<T>(l : List<T>) : List<T>
```

parameters	
Generic parameters	T
Variable argument	l : List<T>
Return type	List<T>

Example

```
import List "mo:base/List";

let list : List.List<Int> = ?(1, ?(0, ?(-1, null)));

List.reverse(list);
```

List.iterate

```
func iterate<T>(
    l : List<T>
    f : T -> ()
) : ()
```

parameters	
Generic parameters	T

parameters	
Variable argument	<code>l : List<T></code>
Function argument	<code>f : T -> ()</code>
Return type	<code>()</code>

Example

```
import List "mo:base/List";
import Iter "mo:base/Iter";

let list : List.List<Int> = ?(-3, ?(-2, ?(-1, null)));

var number : Int = 5;

func edit(x : Int) : () {
    number += x;
};

let iterate : () = List.iterate(list, edit);

List.toArray(list);
```

List.map

```
func map<T, U>(
    l : List<T>
    f : T -> U
) : List<U>
```

parameters	
Generic parameters	<code>T, U</code>
Variable argument	<code>l : List<T></code>
Function argument	<code>f : T -> U</code>
Return type	<code>List<U></code>

Example

```
import List "mo:base/List";
import Int "mo:base/Int";

let list : List.List<Int> = ?(-3, ?(-2, ?(-1, null)));

func change(x : Int) : Text {
    Int.toText(x);
};

List.map<Int, Text>(list, change);
```

List.filter

```
func filter<T>(
    l : List<T>
    f : T -> Bool
) : List<T>
```

parameters	
Generic parameters	T
Variable argument	l : List<T>
Function argument	f : T -> Bool
Return type	List<T>

Example

```
import List "mo:base/List";
import Int "mo:base/Int";

let list : List.List<Int> = ?(1, ?(0, ?(-1, null)));

func change(x : Int) : Bool {
    x > 0;
};

List.filter<Int>(list, change);
```

List.partition

```
func partition<T>(
    l : List<T>
    f : T -> Bool
) : (List<T>, List<T>)
```

parameters	
Generic parameters	T
Variable argument	l : List<T>
Function argument	f : T -> Bool
Return type	(List<T>, List<T>)

Example

```
import List "mo:base/List";
import Int "mo:base/Int";

let list : List.List<Int> = ?(1, ?(0, ?(-1, ?(-2, null))));

func change(x : Int) : Bool {
    x >= 0;
};

List.partition<Int>(list, change);
```

List.mapFilter

```
func mapFilter<T, U>(
    l : List<T>
    f : T -> ?U
) : List<U>
```

parameters	
Generic parameters	T, U
Variable argument	l : List<T>

parameters	
Function argument	f : T -> ?U
Return type	List<U>

Example

```
import List "mo:base/List";
import Int "mo:base/Int";

let list : List.List<Int> = ?(1, ?(0, ?(-1, null)));

func change(x : Int) : ?Text {
    if (x >= 0) {
        ?(Int.toText(x));
    } else {
        null;
    };
};

List.mapFilter<Int, Text>(list, change);
```

List.mapResult

```
func mapResult<T, R, E>(
    xs : List<T>
    f : T -> Result.Result<R, E>
) : Result.Result<List<R>, E>
```

parameters	
Generic parameters	T, R, E
Variable argument	xs : List<T>
Function argument	f : T -> Result.Result<R, E>
Return type	Result.Result<List<R>, E>

Example

```

import List "mo:base/List";
import Result "mo:base/Result";

let list : List.List<Int> = ?(1, ?(0, ?(-1, null)));

func result(x : Int) : Result.Result<Int, Text> {
    if (x >= 0) {
        #ok(x);
    } else {
        #err("it is negative list element");
    };
};

List.mapResult<Int, Int, Text>(list, result);

```

List.append

```

func append<T>(
    l : List<T>
    m : List<T>
) : (List<T>)

```

parameters	
Generic parameters	T
Variable argument1	l : List<T>
Variable argument2	m : List<T>
Return type	List<T>

Example

```

import List "mo:base/List";

let listN : List.List<Int> = ?(-3, ?(-2, ?(-1, null)));
let listP : List.List<Int> = ?(2, ?(1, ?(0, null)));

List.append<Int>(listP, listN);

```

List.flatten

```
func flatten<T>(
    l : List<List<T>>
) : (List<T>)
```

parameters	
Generic parameters	T
Variable argument	l : List<List<T>>
Return type	List<T>

Example

```
import List "mo:base/List";

let listN : List.List<Int> = ?(-3, ?(-2, ?(-1, null)));
let listP : List.List<Int> = ?(2, ?(1, ?(0, null)));

let list0flists : List.List<List.List<Int>> = ?(listN, ?(listP, null));

List.flatten<Int>(list0flists);
```

List.take

```
func take<T>(
    l : List<T>
    n : Nat
) : (List<T>)
```

parameters	
Generic parameters	T
Variable argument1	l : List<T>
Variable argument2	n : Nat
Return type	List<T>

Example

```
import List "mo:base/List";

let list : List.List<Int> = ?(-3, ?(-2, ?(-1, null)));

List.take<Int>(list, 2);
```

List.drop

```
func drop<T>(
    l : List<T>
    n : Nat
) : (List<T>)
```

parameters	
Generic parameters	T
Variable argument1	l : List<T>
Variable argument2	n : Nat
Return type	List<T>

Example

```
import List "mo:base/List";

let list : List.List<Int> = ?(-3, ?(-2, ?(-1, null)));

List.drop<Int>(list, 2);
```

List.foldLeft

```
func foldLeft<T, S>(
    list : List<T>
    base : S
    combine : (S, T) -> S
) : S
```

parameters	
Generic parameters	T, S
Variable argument1	list : List<T>
Variable argument2	base : S
Function argument	combine : (S, T) -> S
Return type	S

Example

```
import List "mo:base/List";

let list : List.List<Int> = ?(-3, ?(-2, ?(-1, null)));
let base : Int = 10;

func change(x : Int, y : Int) : Int {
    x + y;
};

List.foldLeft<Int, Int>(list, 2, change);
```

List.foldRight

```
func foldRight<T, S>(
    list : List<T>
    base : S
    combine : (T, S) -> S
) : S
```

parameters	
Generic parameters	T, S
Variable argument1	list : List<T>
Variable argument2	base : S
Function argument	combine : (T, S) -> S
Return type	S

Example

```
import List "mo:base/List";

let list : List.List<Int> = ?(-3, ?(-2, ?(-1, null)));
let base : Int = 10;

func change(x : Int, y : Int) : Int {
    x + y;
};

List.foldRight<Int, Int>(list, 2, change);
```

List.find

```
func find<T>(
    l : List<T>
    f : T -> Bool
) : ?T
```

parameters	
Generic parameters	T
Variable argument	l : List<T>
Function argument	f : T -> Bool
Return type	?T

Example

```
import List "mo:base/List";

let list : List.List<Int> = ?(-3, ?(-2, ?(-1, null)));

let base : Int = 10;

func change(x : Int) : Bool {
    x > -2;
};

List.find<Int>(list, change);
```

List.some

```
func some<T>(
    l : List<T>
    f : T -> Bool
) : Bool
```

parameters	
Generic parameters	T
Variable argument	l : List<T>
Function argument	f : T -> Bool
Return type	Bool

Example

```
import List "mo:base/List";

let list : List.List<Int> = ?(-3, ?(-2, ?(-1, null)));

let base : Int = 10;

func change(x : Int) : Bool {
    x == -2;
};

List.some<Int>(list, change);
```

List.all

```
func all<T>(
    l : List<T>
    f : T -> Bool
) : Bool
```

parameters	
Generic parameters	T
Variable argument	l : List<T>
Function argument	f : T -> Bool
Return type	Bool

Example

```
import List "mo:base/List";

let list : List.List<Int> = ?(-3, ?(-2, ?(-1, null)));
let base : Int = 10;

func change(x : Int) : Bool {
    x > 0;
};

List.all<Int>(list, change);
```

List.merge

```
func merge<T>(
    l1 : List<T>
    l2 : List<T>
    lessThanOrEqual : (T, T) -> Bool
) : List<T>
```

parameters	
Generic parameters	T
Variable argument1	l1 : List<T>
Variable argument2	l2 : List<T>
Function argument	lessThanOrEqual : (T, T) -> Bool
Return type	List<T>

Example

```
import List "mo:base/List";

let list1 : List.List<Int> = ?(-3, ?(-2, ?(-1, null)));
let list2 : List.List<Int> = ?(-3, ?(-2, ?(-1, null)));

func change(x : Int, y : Int) : Bool {
    x <= y;
};

List.merge<Int>(list1, list2, change);
```

List.compare

```
func compare<T>(
    l1 : List<T>
    l2 : List<T>
    compare : (T, T) -> Order.Order
) : Order.Order
```

parameters	
Generic parameters	T
Variable argument1	l1 : List<T>
Variable argument2	l2 : List<T>
Function argument	compare : (T, T) -> Order.Order
Return type	Order.Order

Example

```
import List "mo:base/List";
import Int "mo:base/Int";
import Order "mo:base/Order";

let list1 : List.List<Int> = ?(-3, ?(-2, ?(-1, null)));
let list2 : List.List<Int> = ?(-3, ?(-2, ?(-1, null)));

List.compare<Int>(list1, list2, Int.compare);
```

List.equal

```
func equal<T>(
    l1 : List<T>
    l2 : List<T>
    equal : (T, T) -> Bool
) : Bool
```

parameters	
Generic parameters	T
Variable argument1	l1 : List<T>
Variable argument2	l2 : List<T>
Function argument	equal : (T, T) -> Bool
Return type	Bool

Example

```
import List "mo:base/List";
import Int "mo:base/Int";

let list1 : List.List<Int> = ?(-3, ?(-2, ?(-1, null)));
let list2 : List.List<Int> = ?(-3, ?(-2, ?(-1, null)));

List.equal<Int>(list1, list2, Int.equal);
```

List.tabulate

```
func tabulate<T>(
    n : Nat
    f : Nat -> T
) : List<T>
```

parameters	
Generic parameters	T
Variable argument	n : Nat
Function argument	f : Nat -> T
Return type	List<T>

Example

```
import List "mo:base/List";

func change(x : Int) : Int {
    x * 2;
};

List.tabulate<Int>(3, change);
```

List.make

```
func make<T>(n : T) : List<T>
```

parameters	
Generic parameters	T
Variable argument	n : T
Return type	List<T>

Example

```
import List "mo:base/List";
import Int "mo:base/Int";

List.make<Int>(3);
```

List.replicate

```
func replicate<T>(
    n : Nat
    x : T
) : List<T>
```

parameters	
Generic parameters	T
Variable argument1	n : Nat
Variable argument2	x : T
Return type	List<T>

Example

```
import List "mo:base/List";
import Int "mo:base/Int";

List.replicate<Int>(3, 3);
```

List.zip

```
func zip<T, U>(
    xs : List<T>
    ys : List<U>
) : List<(T, U)>
```

parameters	
Generic parameters	T, U
Variable argument1	xs : List<T>
Variable argument2	ys : List<U>
Return type	List<(T, U)>

Example

```
import List "mo:base/List";

let listN : List.List<Int> = ?(-3, ?(-2, ?(-1, null)));
let listP : List.List<Int> = ?(1, ?(0, null));

List.zip<Int, Int>(listN, listP);
```

List.zipWith

```
func zipWith<T, U, V>(
    xs : List<T>
    ys : List<U>
    f : (T, U) -> V
) : List<V>
```

parameters	
Generic parameters	T, U, V
Variable argument1	xs : List<T>
Variable argument2	ys : List<U>
Function argument2	f : (T, U) -> V
Return type	List<V>

Example

```
import List "mo:base/List";

let listN : List.List<Int> = ?(-3, ?(-2, ?(-1, null)));
let listP : List.List<Int> = ?(0, ?(0, null));

func edit(x : Int, y : Int) : Int {
    x * y;
};

List.zipWith<Int, Int, Int>(listN, listP, edit);
```

List.split

```
func split<T>(
    n : Nat
    xs : List<T>
) : (List<T>, List<T>)
```

parameters	
Generic parameters	T
Variable argument1	n : Nat
Variable argument2	xs : List<T>
Return type	(List<T>, List<T>)

Example

```
import List "mo:base/List";

let list : List.List<Int> = ?(-3, ?(-2, ?(-1, null)));
List.split<Int>(2, list);
```

List.chunks

```
func chunks<T>(
    n : Nat
    xs : List<T>
) : List<List<T>>
```

parameters	
Generic parameters	T
Variable argument1	n : Nat
Variable argument2	xs : List<T>
Return type	List<List<T>>

Example

```
import List "mo:base/List";

let list : List.List<Int> = ?(-3, ?(-2, ?(-1, null)));

List.chunks<Int>(1, list);
```

List.fromArray

```
func fromArray<T>(xs : [T]) : List<T>
```

parameters	
Generic parameters	T
Variable argument	xs : [T]
Return type	List<T>

Example

```
import List "mo:base/List";

let array : [Int] = [-2, -1, 0, 1, 2];

List.fromArray<Int>(array);
```

List.fromVarArray

```
func fromVarArray<T>(
    xs : [var T]
) : List<T>
```

parameters	
Generic parameters	T
Variable argument	xs : [var T]
Return type	List<T>

Example

```
import List "mo:base/List";

let varArray : [var Int] = [var -2, -1, 0, 1, 2];

List.fromVarArray<Int>(varArray);
```

List.toArray

```
func toArray<T>(xs : List<T>) : [T]
```

parameters	
Generic parameters	T
Variable argument	xs : List<T>

parameters	
Return type	[T]

Example

```
import List "mo:base/List";

let list : List.List<Int> = ?(-3, ?(-2, ?(-1, null)));

List.toArray<Int>(list);
```

List.toVarArray

```
func toVarArray<T>(xs : List<T>) : [var T]
```

parameters	
Generic parameters	T
Variable argument	xs : List<T>
Return type	[var T]

Example

```
import List "mo:base/List";

let list : List.List<Int> = ?(-3, ?(-2, ?(-1, null)));

List.toVarArray<Int>(list);
```

List.tolter

```
func toIter<T>(xs : List<T>) : Iter.Iter<T>
```

parameters	
Generic parameters	T

parameters	
Variable argument	xs : List<T>
Return type	Iter.Iter<T>

Example

```
import List "mo:base/List";
import Iter "mo:base/Iter";

let list : List.List<Int> = ?(-3, ?(-2, ?(-1, null)));
let iter : Iter.Iter<Int> = List.toIter<Int>(list);

var number : Int = 10;

for (i in iter) {
    number += i;
};

number
```

Buffer

A Buffer in Motoko is a *growable* data structure that houses elements of [generic type X](#). The [Buffer Base Module](#) contains a [class Buffer](#) (same name as module) with [class methods](#). The module also offers many [public functions](#).

The *convention* is to name the [module alias](#) after the [file name](#) it is defined in:

```
import Buffer "mo:base/Buffer";
```

On this page

Type [Buffer.Buffer<X>](#)
Class [Buffer.Buffer<X>\(initCapacity : Nat\)](#)

Class methods

- Method [size](#)
- Method [add](#)
- Method [get](#)
- Method [getOpt](#)
- Method [put](#)
- Method [removeLast](#)
- Method [remove](#)
- Method [clear](#)
- Method [filterEntries](#)
- Method [capacity](#)
- Method [reserve](#)
- Method [append](#)
- Method [insert](#)
- Method [insertBuffer](#)
- Method [sort](#)
- Method [vals](#)

Module public functions

- Function [isEmpty](#)
- Function [contains](#)
- Function [clone](#)
- Function [max](#)
- Function [min](#)

Function `equal`
Function `compare`
Function `toText`
Function `hash`
Function `indexof`
Function `lastIndexof`
Function `indexofBuffer`
Function `binarySearch`
Function `subBuffer`
Function `isSubBufferOf`
Function `isStrictSubBufferOf`
Function `prefix`
Function `isPrefixOf`
Function `isStrictPrefixOf`
Function `suffix`
Function `isSuffixOf`
Function `isStrictSuffixOf`
Function `forAll`
Function `forSome`
Function `forNone`
Function `toArray`
Function `toVarArray`
Function `fromArray`
Function `fromVarArray`
Function `fromIter`
Function `trimToSize`
Function `map`
Function `iterate`
Function `mapEntries`
Function `mapFilter`
Function `mapResult`
Function `chain`
Function `foldLeft`
Function `foldRight`
Function `first`
Function `last`
Function `make`
Function `reverse`
Function `merge`
Function `removeDuplicates`

```
Function partition
Function split
Function chunk
Function groupBy
Function flatten
Function zip
Function zipWith
Function takeWhile
Function dropWhile
```

Type Buffer.Buffer<X>

The `Buffer` module contains a [public type](#) `Buffer<X>` with the same name. It's convenient to [rename the type](#) locally:

```
import Buffer "mo:base/Buffer";

type Buffer<X> = Buffer.Buffer<X>;

type BufNat = Buffer.Buffer<Nat>;
```

In the first line we declare a local [type alias](#) `Buffer<X>` by referring to the type inside the module. This new local type name takes in a [generic type parameter](#) `<X>`.

In the second line we declare another local alias `BufNat` which takes no parameters. It is always a `Buffer` of `Nat`.

Class Buffer.Buffer<X>

```
Buffer.Buffer<X>(initCapacity : Nat)
```

The `Buffer<X>` class takes one argument `initCapacity` of type `Nat`, which represent the initial capacity of the buffer.

To construct a buffer object, we use the `Buffer` class:

```
import Buffer "mo:base/Buffer";

type Buffer<X> = Buffer.Buffer<X>;

let myBuffer : Buffer<Nat> = Buffer.Buffer<Nat>(100);
```

We construct an `object` `myBuffer` of type `Buffer<Nat>` by calling the `class` `Buffer.Buffer` with type parameter `Nat` and initial capacity `100`.

Class methods

myBuffer.size

```
func size() : Nat
```

The function `size` takes no argument and returns a `Nat` value.

```
import Buffer "mo:base/Buffer";

let intStorage = Buffer.Buffer<Int>(10);

intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);

intStorage.size()
```

myBuffer.add

```
func add(element : X) : ()
```

The function `add` takes one generic type `X` argument and returns a `()` value.

```
import Buffer "mo:base/Buffer";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);

Buffer.toArray(intStorage)
```

myBuffer.get

```
func get(index : Nat) : X
```

The function `get` takes one `Nat` argument and returns a generic type value `X`.

```
import Buffer "mo:base/Buffer";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);

intStorage.get(2);
```

myBuffer.getOpt

```
func getOpt(index : Nat) : ?X
```

The function `getOpt` takes one `Nat` argument and returns a generic type value `?X`.

```
import Buffer "mo:base/Buffer";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);
```

myBuffer.put

```
func put(index : Nat, element : X) : ()
```

The function `put` takes one `Nat` and one generic type `X` argument and returns a `()` value.

```
import Buffer "mo:base/Buffer";

let intStorage = Buffer.Buffer<Int>(3);

intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);

intStorage.put(3, 2);

Buffer.toArray(intStorage);
```

myBuffer.removeLast

```
func removeLast() : ?X
```

The function `removeLast` takes no argument and returns a generic type value `?X`.

```
import Buffer "mo:base/Buffer";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);

let removeLast : ?Int = intStorage.removeLast();

Buffer.toArray(intStorage);
```

myBuffer.remove

```
func remove(index : Nat) : X
```

The function `remove` takes one `Nat` argument and returns a generic type value `X`.

```
import Buffer "mo:base/Buffer";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);

let remove : Int = intStorage.remove(1);

Buffer.toArray(intStorage);
```

myBuffer.clear

```
func clear() : ()
```

The function `clear` takes no argument and returns a `()` value.

```
import Buffer "mo:base/Buffer";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);

intStorage.clear();

Buffer.toArray(intStorage);
```

myBuffer.filterEntries

```
func filterEntries(predicate : (Nat, X) -> Bool) : ()
```

Parameters	
Function argument	(Nat, X) -> Bool
Return type	()

```
import Buffer "mo:base/Buffer";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-2);
intStorage.add(2);
intStorage.add(3);
intStorage.add(4);
intStorage.add(7);

func check(index : Nat, value : Int) : Bool {
    value % 2 == 0;
};

intStorage.filterEntries(check);

Buffer.toArray(intStorage);
```

myBuffer.capacity

```
func capacity() : Nat
```

The function `capacity` takes no argument and returns a `Nat` value.

```
import Buffer "mo:base/Buffer";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);

intStorage.capacity();
```

myBuffer.reserve

```
func reserve(capacity : Nat) : ()
```

The function `reserve` takes one `Nat` argument and returns a `()` value.

```
import Buffer "mo:base/Buffer";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);

intStorage.reserve(4);
```

myBuffer.append

```
func append(buffer2 : Buffer<X>) : ()
```

The function `append` takes one generic type `Buffer<X>` argument and returns a `()` value.

```
import Buffer "mo:base/Buffer";

let intStorage1 = Buffer.Buffer<Int>(0);

intStorage1.add(-1);
intStorage1.add(0);
intStorage1.add(1);

let intStorage2 = Buffer.Buffer<Int>(0);
intStorage2.add(2);

intStorage1.append(intStorage2);

Buffer.toArray(intStorage1);
```

myBuffer.insert

```
func insert(index : Nat, element : X) : ()
```

The function `insert` takes one `Nat` and one generic type `X` argument and returns a `()` value.

```
import Buffer "mo:base/Buffer";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);

intStorage.insert(3, 2);

Buffer.toArray(intStorage);
```

myBuffer.insertBuffer

```
func insertBuffer(index : Nat, buffer2 : Buffer<X>) : ()
```

The function `insertBuffer` takes one `Nat` and one generic type `Buffer<X>` argument and returns a `()` value.

```
import Buffer "mo:base/Buffer";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);

let buffer1 = Buffer.Buffer<Int>(2);
buffer1.add(4);

intStorage.insertBuffer(3, buffer1);

Buffer.toArray(intStorage);
```

myBuffer.sort

```
func sort(compare : (X, X) -> Order.Order) : ()
```

Parameters	
Function argument	(X, X) -> Order.Order
Return type	()

```
import Buffer "mo:base/Buffer";
import Int "mo:base/Int";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);

intStorage.sort(Int.compare);

Buffer.toArray(intStorage);
```

myBuffer.vals

```
func vals() : { next : () -> ?X }
```

The function `vals` takes no argument and returns a `Iter` value.

```
import Buffer "mo:base/Buffer";
import Iter "mo:base/Iter";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);

let iter : Iter.Iter<Int> = intStorage.vals();

Iter.toArray(iter);
```

Module public functions

Buffer.isEmpty

```
func isEmpty<X>(buffer : Buffer<X>) : Bool
```

Parameters	
Generic parameters	X

Parameters	
Variable argument	buffer : Buffer<X>
Return type	Bool

```
import Buffer "mo:base/Buffer";

let intStorage = Buffer.Buffer<Int>(0);

Buffer.isEmpty(intStorage);
```

Buffer.contains

```
func contains<X>(
    buffer : Buffer<X>
    element : X
    equal : (X, X) -> Bool
) : Bool
```

Parameters	
Generic parameters	X
Variable argument1	buffer : Buffer<X>
Variable argument2	element : X
Function argument	equal : (X, X) -> Bool
Return type	Bool

```
import Buffer "mo:base/Buffer";
import Int "mo:base/Int";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);

let element : Int = 2;

Buffer.contains(intStorage, element, Int.equal);
```

Buffer.clone

```
func clone<X>(buffer : Buffer<X>) : Buffer<X>
```

Parameters	
Generic parameters	X
Variable argument	buffer : Buffer<X>
Return type	Buffer<X>

```
import Buffer "mo:base/Buffer";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);

let clone = Buffer.clone(intStorage);

Buffer.toArray(clone);
```

Buffer.max

```
func max<X>(
    buffer : Buffer<X>
    compare : (X, X) -> Order
) : ?X
```

Parameters	
Generic parameters	X
Variable argument	buffer : Buffer<X>
Function argument	compare : (X, X) -> Order
Return type	?X

```
import Buffer "mo:base/Buffer";
import Int "mo:base/Int";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);

Buffer.max<Int>(intStorage, Int.compare)
```

Buffer.min

```
func min<X>(
    buffer : Buffer<X>
    compare : (X, X) -> Order
) : ?X
```

Parameters	
Generic parameters	X
Variable argument	buffer : Buffer<X>
Function argument	compare : (X, X) -> Order
Return type	?X

```
import Buffer "mo:base/Buffer";
import Int "mo:base/Int";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);

Buffer.min<Int>(intStorage, Int.compare)
```

Buffer.equal

```
func equal<X>(
    buffer1 : Buffer<X>
    buffer2 : Buffer<X>
    equal : (X, X) -> Bool
) : Bool
```

Parameters	
Generic parameters	X
Variable argument1	buffer1 : Buffer<X>
Variable argument2	buffer2 : Buffer<X>
Function argument	equal : (X, X) -> Bool
Return type	Bool

```
import Buffer "mo:base/Buffer";
import Int "mo:base/Int";

let intStorage1 = Buffer.Buffer<Int>(0);

intStorage1.add(-1);
intStorage1.add(0);
intStorage1.add(1);

let intStorage2 = Buffer.Buffer<Int>(0);

intStorage2.add(-1);
intStorage2.add(0);
intStorage2.add(1);

Buffer.equal(intStorage1, intStorage2, Int.equal);
```

Buffer.compare

```
func compare<X>(
    buffer1 : Buffer<X>
    buffer2 : Buffer<X>
    compare : (X, X) -> Order.Order
) : Order.Order
```

Parameters	
Generic parameters	X
Variable argument1	buffer1 : Buffer<X>
Variable argument2	buffer2 : Buffer<X>
Function argument	compare : (X, X) -> Order.Order
Return type	Order.Order

```

import Buffer "mo:base/Buffer";
import Order "mo:base/Order";
import Int "mo:base/Int";

let intStorage1 = Buffer.Buffer<Int>(0);

intStorage1.add(-1);
intStorage1.add(0);
intStorage1.add(1);

let intStorage2 = Buffer.Buffer<Int>(0);

intStorage2.add(-1);
intStorage2.add(0);
intStorage2.add(1);

Buffer.compare<Int>(intStorage1, intStorage2, Int.compare);

```

Buffer.toText

```

func toText<X>(
    buffer : Buffer<X>
    toText : X -> Text
) : Text

```

Parameters	
Generic parameters	X
Variable argument	buffer : Buffer<X>
Function argument	toText : X -> Text
Return type	Text

```
import Buffer "mo:base/Buffer";
import Int "mo:base/Int";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);

Buffer.toText(intStorage, Int.toText)
```

Buffer.indexOf

```
func indexOf<X>(
    element : X
    buffer : Buffer<X>
    equal : (X, X) -> Bool
) : ?Nat
```

Parameters	
Generic parameters	X
Variable argument1	element : X
Variable argument2	buffer : Buffer<X>
Function argument	equal : (X, X) -> Bool
Return type	?Nat

```
import Buffer "mo:base/Buffer";
import Int "mo:base/Int";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);

Buffer.indexOf<Int>(1, intStorage, Int.equal);
```

Buffer.lastIndexOf

```
func indexOf<X>(
    element : X
    buffer : Buffer<X>
    equal : (X, X) -> Bool
) : ?Nat
```

Parameters	
Generic parameters	X
Variable argument1	element : X
Variable argument2	buffer : Buffer<X>
Function argument	equal : (X, X) -> Bool
Return type	?Nat

```
import Buffer "mo:base/Buffer";
import Int "mo:base/Int";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);
intStorage.add(0);
intStorage.add(0);

Buffer.lastIndexOf<Int>(0, intStorage, Int.equal)
```

Buffer.indexOfBuffer

```
func indexOfBuffer<X>(
    subBuffer : Buffer<X>
    buffer : Buffer<X>
    equal : (X, X) -> Bool
) : ?Nat
```

Parameters	
Generic parameters	X

Parameters	
Variable argument1	subBuffer : Buffer<X>
Variable argument2	buffer : Buffer<X>
Function argument	equal : (X, X) -> Bool
Return type	?Nat

```

import Buffer "mo:base/Buffer";
import Int "mo:base/Int";

let intStorage1 = Buffer.Buffer<Int>(0);

intStorage1.add(-3);
intStorage1.add(-2);
intStorage1.add(-1);
intStorage1.add(0);
intStorage1.add(1);

let intStorage2 = Buffer.Buffer<Int>(0);

intStorage2.add(-1);
intStorage2.add(0);
intStorage2.add(1);

Buffer.indexOfBuffer<Int>(intStorage2, intStorage1, Int.equal)

```

Buffer.binarySearch

```

func binarySearch<X>(
    element : X
    buffer : Buffer<X>
    compare : (X, X) -> Order.Order
) : ?Nat

```

Parameters	
Generic parameters	X
Variable argument1	element : X
Variable argument2	buffer : Buffer<X>
Function argument	compare : (X, X) -> Order.Order
Return type	?Nat

```
import Buffer "mo:base/Buffer";
import Int "mo:base/Int";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-3);
intStorage.add(-2);
intStorage.add(-1);

Buffer.binarySearch<Int>(-1, intStorage, Int.compare);
```

Buffer.subBuffer

```
func subBuffer<X>(
    buffer : Buffer<X>
    start : Nat
    length : Nat
) : Buffer<X>
```

Parameters	
Generic parameters	X
Variable argument1	buffer : Buffer<X>
Variable argument2	start : Nat
variable argument3	length : Nat
Return type	Buffer<X>

```
import Buffer "mo:base/Buffer";
import Int "mo:base/Int";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-3);
intStorage.add(-2);
intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);

let subBuffer : Buffer.Buffer<Int> = Buffer.subBuffer(intStorage, 2, 2);

Buffer.toText<Int>(subBuffer, Int.toText)
```

Buffer.isSubBufferOf

```
func isSubBufferOf<X>(
    subBuffer : Buffer<X>
    buffer : Buffer<X>
    equal : (X, X) -> Bool
) : Bool
```

Parameters	
Generic parameters	X
Variable argument1	subBuffer : Buffer<X>
Variable argument2	buffer : Buffer<X>
Function argument	equal : (X, X) -> Bool
Return type	Bool

```
import Buffer "mo:base/Buffer";
import Int "mo:base/Int";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-3);
intStorage.add(-2);
intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);

let subIntStorage = Buffer.Buffer<Int>(0);

intStorage.add(-3);
intStorage.add(-2);

Buffer.isSubBufferOf(subIntStorage, intStorage, Int.equal)
```

Buffer.isStrictSubBufferOf

```
func isStrictSubBufferOf<X>(
    subBuffer : Buffer<X>
    buffer : Buffer<X>
    equal : (X, X) -> Bool
) : Bool
```

Parameters	
Generic parameters	X
Variable argument1	subBuffer : Buffer<X>
Variable argument2	buffer : Buffer<X>
Function argument	equal : (X, X) -> Bool
Return type	Bool

```
import Buffer "mo:base/Buffer";
import Int "mo:base/Int";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-2);
intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);

let subIntStorage = Buffer.Buffer<Int>(0);

intStorage.add(-1);
intStorage.add(0);

Buffer.isStrictSubBufferOf(subIntStorage, intStorage, Int.equal)
```

Buffer.prefix

```
func prefix<X>(
    buffer : Buffer<X>
    length : Nat
) : Buffer<X>
```

Parameters	
Generic parameters	X
Variable argument1	buffer : Buffer<X>
variable argument2	length : Nat
Return type	Buffer<X>

```
import Buffer "mo:base/Buffer";
import Int "mo:base/Int";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-2);
intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);
intStorage.add(2);

let prefix : Buffer.Buffer<Int> = Buffer.prefix(intStorage, 3);

Buffer.toText(prefix, Int.toText)
```

Buffer.isPrefixOf

```
func isPrefixOf<X>(
    prefix : Buffer<X>
    buffer : Buffer<X>
    equal : (X, X) -> Bool
) : Bool
```

Parameters	
Generic parameters	X
Variable argument1	prefix : Buffer<X>
Variable argument2	buffer : Buffer<X>
Function argument	equal : (X, X) -> Bool
Return type	Bool

```

import Buffer "mo:base/Buffer";
import Int "mo:base/Int";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-2);
intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);
intStorage.add(2);

let prefix = Buffer.Buffer<Int>(0);

intStorage.add(-2);
intStorage.add(-1);
intStorage.add(0);

Buffer.isPrefixOf(prefix, intStorage, Int.equal)

```

Buffer.isStrictPrefixOf

```

func isStrictPrefixOf<X>(
    prefix : Buffer<X>
    buffer : Buffer<X>
    equal : (X, X) -> Bool
) : Bool

```

Parameters	
Generic parameters	X
Variable argument1	prefix : Buffer<X>
Variable argument2	buffer : Buffer<X>
Function argument	equal : (X, X) -> Bool
Return type	Bool

```

import Buffer "mo:base/Buffer";
import Int "mo:base/Int";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-2);
intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);
intStorage.add(2);

let prefix = Buffer.Buffer<Int>(0);

intStorage.add(-2);
intStorage.add(-1);
intStorage.add(0);

Buffer.isStrictPrefixOf(prefix, intStorage, Int.equal)

```

Buffer.suffix

```

func suffix<X>(
    buffer : Buffer<X>
    length : Nat
) : Buffer<X>

```

Parameters	
Generic parameters	X
Variable argument1	buffer : Buffer<X>
variable argument2	length : Nat
Return type	Buffer<X>

```

import Buffer "mo:base/Buffer";
import Int "mo:base/Int";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-2);
intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);
intStorage.add(2);

let suffix : Buffer.Buffer<Int> = Buffer.suffix(intStorage, 3);

Buffer.toText(suffix, Int.toText)

```

Buffer.isSuffixOf

```

func isSuffixOf<X>(
    suffix : Buffer<X>
    buffer : Buffer<X>
    equal : (X, X) -> Bool
) : Bool

```

Parameters	
Generic parameters	X
Variable argument1	suffix : Buffer<X>
Variable argument2	buffer : Buffer<X>
Function argument	equal : (X, X) -> Bool
Return type	Bool

```

import Buffer "mo:base/Buffer";
import Int "mo:base/Int";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-2);
intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);
intStorage.add(2);

let suffix = Buffer.Buffer<Int>(0);

intStorage.add(0);
intStorage.add(1);
intStorage.add(2);

Buffer.isSuffixOf(suffix, intStorage, Int.equal)

```

Buffer.isStrictSuffixOf

```

func isStrictSuffixOf<X>(
    suffix : Buffer<X>
    buffer : Buffer<X>
    equal : (X, X) -> Bool
) : Bool

```

Parameters	
Generic parameters	X
Variable argument1	suffix : Buffer<X>
Variable argument2	buffer : Buffer<X>
Function argument	equal : (X, X) -> Bool
Return type	Bool

```

import Buffer "mo:base/Buffer";
import Int "mo:base/Int";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-2);
intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);
intStorage.add(2);

let suffix = Buffer.Buffer<Int>(0);

intStorage.add(0);
intStorage.add(1);
intStorage.add(2);

Buffer.isStrictSuffixOf(suffix, intStorage, Int.equal)

```

Buffer.forAll

```

func forAll<X>(
    buffer : Buffer<X>
    predicate : X -> Bool
) : Bool

```

Parameters	
Generic parameters	X
Variable argument	buffer : Buffer<X>
Function argument	predicate : X -> Bool
Return type	Bool

```

import Buffer "mo:base/Buffer";
import Int "mo:base/Int";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);

func check(x : Int) : Bool {
    x > 0;
};

Buffer.forAll<Int>(intStorage, check);

```

Buffer.forSome

```

func forSome<X>(
    buffer : Buffer<X>
    predicate : X -> Bool
) : Bool

```

Parameters	
Generic parameters	X
Variable argument	buffer : Buffer<X>
Function argument	predicate : X -> Bool
Return type	Bool

```

import Buffer "mo:base/Buffer";
import Int "mo:base/Int";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);

func check(x : Int) : Bool {
    x > 0;
};

Buffer.forSome<Int>(intStorage, check);

```

Buffer.forNone

```
func forNone<X>(
    buffer : Buffer<X>
    predicate : X -> Bool
) : Bool
```

Parameters	
Generic parameters	X
Variable argument	buffer : Buffer<X>
Function argument	predicate : X -> Bool
Return type	Bool

```
import Buffer "mo:base/Buffer";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);

func check(x : Int) : Bool {
    x > 1;
};

Buffer.forNone<Int>(intStorage, check);
```

Buffer.toArray

```
func toArray<X>(buffer : Buffer<X>) : [X]
```

Parameters	
Generic parameters	X
Variable argument	buffer : Buffer<X>
Return type	[X]

```
import Buffer "mo:base/Buffer";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);

Buffer.toArray<Int>(intStorage)
```

Buffer.toVarArray

```
func toVarArray<X>(buffer : Buffer<X>) : [var X]
```

Parameters	
Generic parameters	X
Variable argument	buffer : Buffer<X>
Return type	[var X]

```
import Buffer "mo:base/Buffer";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);

Buffer.toVarArray<Int>(intStorage)
```

Buffer.fromArray

```
func fromArray<X>(array : [X]) : Buffer<X>
```

Parameters	
Generic parameters	X
Variable argument	array : [X]
Return type	Buffer<X>

```
import Buffer "mo:base/Buffer";
import Int "mo:base/Int";

let array : [Int] = [-1, 0, 1];

let buffer : Buffer.Buffer<Int> = Buffer.fromArray<Int>(array);

Buffer.toText(buffer, Int.toText);
```

Buffer.fromVarArray

```
func fromArray<X>(array : [var X]) : Buffer<X>
```

Parameters	
Generic parameters	X
Variable argument	array : [var X]
Return type	Buffer<X>

```
import Buffer "mo:base/Buffer";
import Int "mo:base/Int";

let array : [Int] = [-1, 0, 1];

let buffer : Buffer.Buffer<Int> = Buffer.fromArray<Int>(array);

Buffer.toText(buffer, Int.toText);
```

Buffer.fromIter

```
func fromIter<X>(iter : { next : () -> ?X }) : Buffer<X>
```

Parameters	
Generic parameters	X
Variable argument	iter : { next : () -> ?X }
Return type	Buffer<X>

```

import Buffer "mo:base/Buffer";
import Iter "mo:base/Iter";
import Int "mo:base/Int";

let array : [Int] = [-1, 0, 1];

let iter : Iter.Iter<Int> = array.vals();

let buffer = Buffer.fromIter<Int>(iter);

Buffer.toText(buffer, Int.toText);

```

Buffer.trimToSize

func trimToSize<X>(buffer : Buffer<X>) : ()

Parameters	
Generic parameters	X
Variable argument	buffer : Buffer<X>
Return type	()

```

import Buffer "mo:base/Buffer";

let intStorage = Buffer.Buffer<Int>(10);

intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);

Buffer.trimToSize<Int>(intStorage);

intStorage.capacity()

```

Buffer.map

```

func map<X, Y>(
    buffer : Buffer<X>
    f : X -> Y
) : Buffer<Y>

```

Parameters	
Generic parameters	X, Y
Variable argument	buffer : Buffer<X>
Function argument	f : X -> Y
Return type	Buffer<Y>

```

import Buffer "mo:base/Buffer";
import Int "mo:base/Int";

let intStorage = Buffer.Buffer<Int>(10);

intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);

func change(x : Int) : Int {
    x ** 2;
};

let newBuffer : Buffer.Buffer<Int> = Buffer.map<Int, Int>(intStorage, change);

Buffer.toText(newBuffer, Int.toText)

```

Buffer.iterate

```

func iterate<X>(
    buffer : Buffer<X>
    f : X -> ()
) : ()

```

Parameters	
Generic parameters	X
Variable argument	buffer : Buffer<X>
Function argument	f : X -> ()
Return type	()

```

import Buffer "mo:base/Buffer";

let intStorage = Buffer.Buffer<Int>(10);

intStorage.add(-2);
intStorage.add(-1);
intStorage.add(0);

func change(x : Int) : () {
    number += x;
};

var number : Int = 10;

Buffer.iterate<Int>(intStorage, change)

```

Buffer.mapEntries

```

func mapEntries<X, Y>(
    buffer : Buffer<X>
    f : (Nat, X) -> Y
) : Buffer<Y>

```

Parameters	
Generic parameters	X, Y
Variable argument	buffer : Buffer<X>
Function argument	f : (Nat, X) -> Y
Return type	Buffer<Y>

```

import Buffer "mo:base/Buffer";

let intStorage = Buffer.Buffer<Int>(10);

intStorage.add(-2);
intStorage.add(-1);
intStorage.add(0);

func change(x : Int, y : Int) : Int {
    x + y + 1;
};

let newBuffer : Buffer.Buffer<Int> = Buffer.mapEntries<Int, Int>(intStorage,
change);

Buffer.toArray<Int>(newBuffer)

```

Buffer.mapFilter

```

func mapFilter<X, Y>(
    buffer : Buffer<X>
    f : X -> ?Y
) : Buffer<Y>

```

Parameters	
Generic parameters	X, Y
Variable argument	buffer : Buffer<X>
Function argument	f : X -> ?Y
Return type	Buffer<Y>

```

import Buffer "mo:base/Buffer";

let intStorage = Buffer.Buffer<Int>(10);

intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);
intStorage.add(2);

func filter(x : Int) : ?Int {
    if (x > 0) {
        ?(x * 10);
    } else {
        null;
    };
};

let newBuffer : Buffer.Buffer<Int> = Buffer.mapFilter<Int, Int>(intStorage,
filter);

Buffer.toArray<Int>(newBuffer)

```

Buffer.mapResult

```

func mapResult<X, Y, E>(
    buffer : Buffer<X>
    f : X -> Result.Result<Y, E>
) : Result.Result<Buffer<Y>, E>

```

Parameters	
Generic parameters	X, Y, E
Variable argument	buffer : Buffer<X>
Function argument	f : X -> Result.Result<Y, E>
Return type	Result.Result<Buffer<Y>, E>

```

import Buffer "mo:base/Buffer";
import Result "mo:base/Result";

let intStorage = Buffer.Buffer<Int>(10);

intStorage.add(0);
intStorage.add(1);
intStorage.add(2);

func filter(x : Int) : Result.Result<Int, Text> {
    if (x > 0) {
        #ok(x * 10);
    } else {
        #err("got negative number");
    };
};

let result : Result.Result<Buffer.Buffer<Int>, Text> = Buffer.mapResult<Int, Int, Text>(intStorage, filter);

func toArray(arg : Buffer.Buffer<Int>) : [Int] {
    let array : [Int] = Buffer.toArray<Int>(arg);
};

Result.mapOk<Buffer.Buffer<Int>, [Int], Text>(result, toArray)

```

Buffer.chain

```

func chain<X, Y>(
    buffer : Buffer<X>
    k : X -> Buffer<Y>
) : Buffer<Y>

```

Parameters	
Generic parameters	X, Y
Variable argument	buffer : Buffer<X>
Function argument	k : X -> Buffer<Y>
Return type	Buffer<Y>

```

import Buffer "mo:base/Buffer";

let intStorageA = Buffer.Buffer<Int>(0);

intStorageA.add(1);
intStorageA.add(2);
intStorageA.add(3);

func change(x : Int) : Buffer.Buffer<Int> {

    let intStorageB = Buffer.Buffer<Int>(2);

    intStorageB.add(x);
    intStorageB.add(x ** 3);
    intStorageB;
};

let chain = Buffer.chain<Int, Int>(intStorageA, change);

Buffer.toArray(chain);

```

Buffer.foldLeft

```

func foldLeft<A, X>(
    buffer : Buffer<X>
    base : A
    combine : (A, X) -> A
) : A

```

Parameters	
Generic parameters	A, X
Variable argument1	buffer : Buffer<X>
Variable argument2	base : A
Function argument	combine : (A, X) -> A
Return type	A

```

import Buffer "mo:base/Buffer";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-1);
intStorage.add(1);
intStorage.add(2);

func change(x : Int, y : Int) : Int {
    x + y;
};

Buffer.foldLeft<Int, Int>(intStorage, 0, change)

```

Buffer.foldRight

```

func foldLeft<A, X>(
    buffer : Buffer<X>
    base : A
    combine : (X, A) -> A
) : A

```

Parameters	
Generic parameters	A, X
Variable argument1	buffer : Buffer<X>
Variable argument2	base : A
Function argument	combine : (X, A) -> A
Return type	A

```

import Buffer "mo:base/Buffer";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-1);
intStorage.add(1);
intStorage.add(2);

func change(x : Int, y : Int) : Int {
    x + y;
};

Buffer.foldRight<Int, Int>(intStorage, 0, change);

```

Buffer.first

```
func first<X>(buffer : Buffer<X>) : X
```

Parameters	
Generic parameters	X
Variable argument	buffer : Buffer<X>
Return type	X

```
import Buffer "mo:base/Buffer";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-1);
intStorage.add(1);
intStorage.add(1);
```

Buffer.last

```
func last<X>(buffer : Buffer<X>) : X
```

Parameters	
Generic parameters	X
Variable argument	buffer : Buffer<X>
Return type	X

```
import Buffer "mo:base/Buffer";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-1);
intStorage.add(1);
intStorage.add(1);

Buffer.last<Int>(intStorage)
```

Buffer.make

```
func make<X>(element : X) : Buffer<X>
```

Parameters	
Generic parameters	X
Variable argument	element : X
Return type	Buffer<X>

```
import Buffer "mo:base/Buffer";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-1);
intStorage.add(1);
intStorage.add(1);

let make : Buffer.Buffer<Int> = Buffer.make<Int>(2);

Buffer.toArray(make)
```

Buffer.reverse

```
func reverse<X>(buffer : Buffer<X>) : ()
```

Parameters	
Generic parameters	X
Variable argument	buffer : Buffer<X>
Return type	()

```
import Buffer "mo:base/Buffer";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);

let reverse : () = Buffer.reverse<Int>(intStorage);

Buffer.toArray(intStorage)
```

Buffer.merge

```
func merge<X>(
    buffer1 : Buffer<X>
    buffer2 : Buffer<X>
    compare : (X, X) -> Order
) : Buffer<X>
```

Parameters	
Generic parameters	X
Variable argument1	buffer1 : Buffer<X>
Variable argument2	buffer2 : Buffer<X>
Function argument	combine : (X, X) -> Order
Return type	BufferX

```
import Buffer "mo:base/Buffer";
import Int "mo:base/Int";

let intStorage1 = Buffer.Buffer<Int>(0);

intStorage1.add(-1);
intStorage1.add(0);
intStorage1.add(1);

let intStorage2 = Buffer.Buffer<Int>(0);

intStorage2.add(-2);
intStorage2.add(2);
intStorage2.add(3);

let merged : Buffer.Buffer<Int> = Buffer.merge<Int>(intStorage1, intStorage2,
Int.compare);

Buffer.toArray<Int>(merged)
```

Buffer.removeDuplicates

```
func removeDuplicates<X>(
    buffer : Buffer<X>
    compare : (X, X) -> Order
) : ()
```

Parameters	
Generic parameters	X
Variable argument	buffer : Buffer<X>
Function argument	compare : (X, X) -> Order
Return type	()

```
import Buffer "mo:base/Buffer";
import Int "mo:base/Int";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);
intStorage.add(1);
intStorage.add(1);

let removeDuplicates : () = Buffer.removeDuplicates<Int>(intStorage, Int.compare);

Buffer.toArray(intStorage)
```

Buffer.partition

```
func partition<X>(
    buffer : Buffer<X>
    predicate : X -> Bool
) : (Buffer<X>, Buffer<X>)
```

Parameters	
Generic parameters	X
Variable argument	buffer : Buffer<X>

Parameters	
Function argument	<code>predicate : X -> Bool</code>
Return type	<code>(Buffer<X>, Buffer<X>)</code>

```

import Buffer "mo:base/Buffer";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-1);
intStorage.add(2);
intStorage.add(-2);
intStorage.add(1);
intStorage.add(3);

func part(x : Int) : Bool {
    x > 0;
};

let partitions = Buffer.partition<Int>(intStorage, part);

let tuple : ([Int], [Int]) = (Buffer.toArray(partitions.0),
Buffer.toArray(partitions.1))

```

Buffer.split

```

func split<X>(
    buffer : Buffer<X>
    index : Nat
) : (Buffer<X>, Buffer<X>)

```

Parameters	
Generic parameters	<code>X</code>
Variable argument1	<code>buffer : Buffer<X></code>
Variable argument2	<code>index : Nat</code>
Return type	<code>(Buffer<X>, Buffer<X>)</code>

```

import Buffer "mo:base/Buffer";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-2);
intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);
intStorage.add(2);

let splits = Buffer.split<Int>(intStorage, 2);

let tuple : ([Int], [Int]) = (Buffer.toArray<Int>(splits.0), Buffer.toArray<Int>(splits.1))

```

Buffer.chunk

```

func chunk<X>(
    buffer : Buffer<X>
    size : Nat
) : Buffer<Buffer<X>>

```

Parameters	
Generic parameters	X
Variable argument1	buffer : Buffer<X>
Variable argument2	size : Nat
Return type	(Buffer<Buffer<X>>)

```

import Buffer "mo:base/Buffer";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-2);
intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);
intStorage.add(2);

let chunk : Buffer.Buffer<Buffer.Buffer<Int>> = Buffer.chunk<Int>(
    intStorage,
    3,
);
;

let array : [Buffer.Buffer<Int>] = Buffer.toArray<Buffer.Buffer<Int>>(chunk);

let array0 : [Int] = Buffer.toArray<Int>(array[0]);
let array1 : [Int] = Buffer.toArray<Int>(array[1]);
(array0, array1);

```

Buffer.groupBy

```

func groupBy<X>(
    buffer : Buffer<X>
    equal : (X, X) -> Bool
) : Buffer<Buffer<X>>

```

Parameters	
Generic parameters	X
Variable argument	buffer : Buffer<X>
Function argument	equal : (X, X) -> Bool
Return type	(Buffer<Buffer<X>>)

```

import Buffer "mo:base/Buffer";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-2);
intStorage.add(-2);
intStorage.add(0);
intStorage.add(0);
intStorage.add(2);
intStorage.add(2);

func edit(x : Int, y : Int) : Bool {
    x == y;
};

let grouped : Buffer.Buffer<Buffer.Buffer<Int>> = Buffer.groupBy<Int>(
    intStorage,
    edit,
);
let array : [Buffer.Buffer<Int>] = Buffer.toArray<Buffer.Buffer<Int>>(grouped);

let array0 : [Int] = Buffer.toArray<Int>(array[0]);
let array1 : [Int] = Buffer.toArray<Int>(array[1]);
let array2 : [Int] = Buffer.toArray<Int>(array[2]);

(array0, array1, array2);

```

Buffer.flatten

```
func flatten<X>(buffer : Buffer<Buffer<X>>) : Buffer<X>
```

Parameters	
Generic parameters	X
Variable argument	buffer : Buffer<Buffer<X>>
Return type	Buffer<X>

```

import Buffer "mo:base/Buffer";

let intStorageN = Buffer.Buffer<Int>(0);

intStorageN.add(-3);
intStorageN.add(-2);
intStorageN.add(-1);

let intStorageP = Buffer.Buffer<Int>(0);

intStorageP.add(0);
intStorageP.add(1);
intStorageP.add(3);

let bufferStorage = Buffer.Buffer<Buffer.Buffer<Int>>(1);

bufferStorage.add(intStorageN);
bufferStorage.add(intStorageP);

let flat : Buffer.Buffer<Int> = Buffer.flatten<Int>(bufferStorage);

Buffer.toArray<Int>(flat);

```

Buffer.zip

```

func zip<X, Y>(
    buffer1 : Buffer<X>
    buffer2 : Buffer<X>
) : Buffer<(X, Y)>

```

Parameters	
Generic parameters	X, Y
Variable argument1	buffer1 : Buffer<X>
Variable argument2	buffer2 : Buffer<X>
Return type	Buffer<(X, Y)>

```

import Buffer "mo:base/Buffer";

let intStorageN = Buffer.Buffer<Int>(0);

intStorageN.add(-3);
intStorageN.add(-2);
intStorageN.add(-1);

let intStorageP = Buffer.Buffer<Int>(0);

intStorageP.add(3);
intStorageP.add(2);

let zipped : Buffer.Buffer<(Int, Int)> = Buffer.zip<Int, Int>(
    intStorageN,
    intStorageP,
);

Buffer.toArray<(Int, Int)>(zipped)

```

Buffer.zipWith

```

func zipWith<X, Y, Z>(
    buffer1 : Buffer<X>
    buffer2 : Buffer<Y>

    zip : (X, Y) -> Z

) : Buffer<Z>

```

Parameters	
Generic parameters	X, Y, Z
Variable argument1	buffer1 : Buffer<X>
Variable argument2	buffer2 : Buffer<Y>
Function argument	zip : (X, Y) -> Z
Return type	Buffer<Z>

```

import Buffer "mo:base/Buffer";

let intStorageN = Buffer.Buffer<Int>(0);

intStorageN.add(-3);
intStorageN.add(-2);
intStorageN.add(-1);

let intStorageP = Buffer.Buffer<Int>(0);

intStorageP.add(3);
intStorageP.add(2);

func edit(x : Int, y : Int) : Int {
    x * y;
};

let zipped : Buffer.Buffer<Int> = Buffer.zipWith<Int, Int, Int>(
    intStorageN,
    intStorageP,
    edit,
);

Buffer.toArray<Int>(zipped)

```

Buffer.takeWhile

```

func takeWhile<X>(
    buffer : Buffer<X>
    predicate : X -> Bool
) : (Buffer<X>)

```

Parameters	
Generic parameters	X
Variable argument	buffer : Buffer<X>
Function argument	predicate : X -> Bool
Return type	(Buffer<X>)

```

import Buffer "mo:base/Buffer";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-3);
intStorage.add(-2);
intStorage.add(-1);
intStorage.add(0);

func check(x : Int) : Bool {
    x < 0;
};

let newBuffer : Buffer.Buffer<Int> = Buffer.takeWhile<Int>(
    intStorage,
    check,
);

Buffer.toArray<Int>(newBuffer)

```

Buffer.dropWhile

```

func dropWhile<X>(
    buffer : Buffer<X>
    predicate : X -> Bool
) : (Buffer<X>)

```

Parameters	
Generic parameters	X
Variable argument	buffer : Buffer<X>
Function argument	predicate : X -> Bool
Return type	(Buffer<X>)

```
import Buffer "mo:base/Buffer";

let intStorage = Buffer.Buffer<Int>(0);

intStorage.add(-3);
intStorage.add(-2);
intStorage.add(-1);
intStorage.add(0);
intStorage.add(1);

func check(x : Int) : Bool {
    x < 0;
};

let newBuffer : Buffer.Buffer<Int> = Buffer.dropWhile<Int>(
    intStorage,
    check,
);
Buffer.toArray<Int>(newBuffer);
```

HashMap

The *convention* is to name the *module alias* after the *file name* it is defined in:

```
import HashMap "mo:base/HashMap";
```

On this page

Type `HashMap.HashMap<K, V>`
Class `HashMap.HashMap<K, V>`

Class methods

Method `get`
Method `size`
Method `put`
Method `replace`
Method `delete`
Method `remove`
Method `keys`
Method `vals`
Method `entries`

Module public functions

Function `clone`
Function `fromIter`
Function `map`
Function `mapFilter`

Type `HashMap.HashMap<K, V>`

The `HashMap` module contains a *public type* `HashMap<K, V>` with the same name. It's convenient to *rename the type* locally:

```
import HashMap "mo:base/HashMap";  
  
type HashMap<K, V> = HashMap.HashMap<K, V>;  
  
type MapTextInt = HashMap<Text, Int>;
```

In the second line we declare a local **type alias** `HashMap<K, V>` by referring to the type inside the module. This new local type name takes in two **generic type parameters** `<K, V>`.

In the third line we declare another local alias `MapTextInt` which takes no parameters. It is always a `HashMap<Text, Int>`.

Class `HashMap`.`HashMap<K, V>`

```
HashMap.HashMap<K, V>(
    initCapacity : Nat,
    keyEq : (K, K) -> Bool,
    keyHash : K -> Hash.Hash
)
```

To construct a hashmap object, we use the `HashMap` class:

```
import HashMap "mo:base/HashMap";
import Text "mo:base/Text";

type HashMap<K, V> = HashMap.HashMap<K, V>;

let hashmap : HashMap<Text, Int> = HashMap.HashMap<Text, Int>(5, Text.equal,
Text.hash);
```

Class methods

`hashmap.size`

```
func size() : Nat
```

The function `size` takes no argument and returns a `Nat` value.

```
import HashMap "mo:base/HashMap";
import Text "mo:base/Text";

let map = HashMap.HashMap<Text, Int>(5, Text.equal, Text.hash);

map.put("Rohit", 30);
map.put("Kohli", 28);
map.put("Rahul", 27);

map.size()
```

hashmap.get

```
func get(key : K) : (value : ?V)
```

The function `get` takes one argument of type `K` and returns a value of type `?V`.

```
import HashMap "mo:base/HashMap";
import Text "mo:base/Text";

let map = HashMap.HashMap<Text, Int>(5, Text.equal, Text.hash);

map.put("Rohit", 30);
map.put("Kohli", 28);
map.put("Rahul", 27);

map.get("Kohli")
```

hashmap.put

```
func put(key : K, value : V)
```

The function `put` takes one argument of type `K` and returns a value of type `V`.

```
import HashMap "mo:base/HashMap";
import Text "mo:base/Text";

let map = HashMap.HashMap<Text, Int>(5, Text.equal, Text.hash);

map.put("Rohit", 30);
map.put("Kohli", 28);
map.put("Rahul", 27);

map.put("Surya", 26)
```

hashmap.replace

```
func replace(key : K, value : V) : (oldValue : ?V)
```

The function `replace` takes one argument of type `K` and one of type `V` returns a value of type `?V`.

```
import HashMap "mo:base/HashMap";
import Text "mo:base/Text";

let map = HashMap.HashMap<Text, Int>(5, Text.equal, Text.hash);

map.put("Rohit", 30);
map.put("Kohli", 28);
map.put("Rahul", 27);

map.replace("Rohit", 29)
```

hashmap.delete

```
func delete(key : K)
```

The function `delete` takes one argument of type `K` and returns nothing.

```
import HashMap "mo:base/HashMap";
import Text "mo:base/Text";

let map = HashMap.HashMap<Text, Int>(5, Text.equal, Text.hash);

map.put("Rohit", 30);
map.put("Kohli", 28);
map.put("Rahul", 27);

map.delete("Rahul")
```

hashmap.remove

```
func remove(key : K) : (oldValue : ?V)
```

The function `remove` takes one argument of type `K` and returns a value of type `?V`.

```
import HashMap "mo:base/HashMap";
import Text "mo:base/Text";

let map = HashMap.HashMap<Text, Int>(5, Text.equal, Text.hash);

map.put("Rohit", 30);
map.put("Kohli", 28);
map.put("Rahul", 27);

map.remove("Kohli")
```

hashmap.keys

```
func keys() : Iter.Iter<K>
```

The function `keys` takes nothing and returns an `Iterator` of type `K`.

```

import HashMap "mo:base/HashMap";
import Text "mo:base/Text";
import Iter "mo:base/Iter";

let map = HashMap.HashMap<Text, Int>(5, Text.equal, Text.hash);

map.put("Rohit", 30);
map.put("Kohli", 28);
map.put("Rahul", 27);

let iter : Iter.Iter<Text> = map.keys();

Iter.toArray<Text>(iter);

```

hashmap.vals

```
func vals() : Iter.Iter<V>
```

The function `vals` takes nothing and returns an `Iterator` of type `V`.

```

import HashMap "mo:base/HashMap";
import Text "mo:base/Text";
import Iter "mo:base/Iter";

let map = HashMap.HashMap<Text, Int>(5, Text.equal, Text.hash);

map.put("Rohit", 30);
map.put("Kohli", 28);
map.put("Rahul", 27);

let iter : Iter.Iter<Int> = map.vals();

Iter.toArray<Int>(iter);

```

hashmap.entries

```
func entries() : Iter.Iter<(K, V)>
```

The function `entries` takes nothing and returns an `Iterator` of type tuple `(K, V)`.

```

import HashMap "mo:base/HashMap";
import Text "mo:base/Text";
import Iter "mo:base/Iter";

let map = HashMap.HashMap<Text, Int>(5, Text.equal, Text.hash);

map.put("Rohit", 30);
map.put("Kohli", 28);
map.put("Rahul", 27);

let iter : Iter.Iter<(Text, Int)> = map.entries();

Iter.toArray<(Text, Int)>(iter);

```

Module public functions

HashMap.clone

```

func clone<K, V>(
    map : HashMap<K, V>,
    keyEq : (K, K) -> Bool,
    keyHash : K -> Hash.Hash
) : HashMap<K, V>

```

Parameters	
Generic parameters	K, V
Variable argument	map : HashMap<K, V>
Function argument 1	keyEq : (K, K) -> Bool
Function argument 2	keyHash : K -> Hash.Hash
Return type	HashMap<K, V>

```

import HashMap "mo:base/HashMap";
import Text "mo:base/Text";

let map = HashMap.HashMap<Text, Int>(5, Text.equal, Text.hash);

map.put("sachin", 100);
map.put("kohli", 74);
map.put("root", 50);

let copy = HashMap.clone(map, Text.equal, Text.hash);

copy.get("kohli")

```

HashMap.fromIter

```

func fromIter<K, V>(
    iter : Iter.Iter<(K, V)>,
    initCapacity : Nat,
    keyEq : (K, K) -> Bool,
    keyHash : K -> Hash.Hash
) : HashMap<K, V>

```

Parameters	
Generic parameters	K, V
Variable argument1	iter : Iter.Iter<(K, V)>
Variable argument2	initCapacity : Nat
Function argument 1	keyEq : (K, K) -> Bool
Function argument 2	keyHash : K -> Hash.Hash
Return type	HashMap<K, V>

```

import HashMap "mo:base/HashMap";
import Iter "mo:base/Iter";
import Text "mo:base/Text";

let array : [(Text, Int)] = [("bitcoin", 1), ("ETH", 2), ("ICP", 20)];

let iter : Iter.Iter<(Text, Int)> = array.vals();

let size : Nat = array.size();

let map : HashMap.HashMap<Text, Int> = HashMap.fromIter<Text, Int>(
    iter,
    size,
    Text.equal,
    Text.hash,
);

map.get("bitcoin")

```

HashMap.map

```

func map<K, V1, V2>(
    hashMap : HashMap<K, V1>,
    keyEq : (K, K) -> Bool,
    keyHash : K -> Hash.Hash,
    f : (K, V1) -> V2
) : HashMap<K, V2>

```

Parameters	
Generic parameters	K, V1, V2
Variable argument1	hashMap : HashMap<K, V1>
Function argument 1	keyEq : (K, K) -> Bool
Function argument 2	keyHash : K -> Hash.Hash
Function argument 3	f : (K, V1) -> V2
Return type	HashMap<K, V2>

```

import HashMap "mo:base/HashMap";
import Text "mo:base/Text";

let map = HashMap.HashMap<Text, Int>(5, Text.equal, Text.hash);

map.put("sachin", 100);
map.put("kohli", 74);
map.put("root", 50);

func edit(k : Text, v : Int) : Int {
    v * 2;
};

let mapping : HashMap.HashMap<Text, Int> = HashMap.map<Text, Int, Int>(
    map,
    Text.equal,
    Text.hash,
    edit,
);

```

HashMap.mapFilter

```

func mapFilter<K, V1, V2>(
    hashMap : HashMap<K, V1>,
    keyEq : (K, K) -> Bool,
    keyHash : K -> Hash.Hash,
    f : (K, V1) -> ?V2
) : HashMap<K, V2>

```

Parameters	
Generic parameters	K, V1, V2
Variable argument1	hashMap : HashMap<K, V1>
Function argument 1	keyEq : (K, K) -> Bool
Function argument 2	keyHash : K -> Hash.Hash
Function argument 3	f : (K, V1) -> ?V2
Return type	HashMap<K, V2>

```
import HashMap "mo:base/HashMap";
import Text "mo:base/Text";

let map = HashMap.HashMap<Text, Int>(5, Text.equal, Text.hash);

map.put("sachin", 100);
map.put("kohli", 74);
map.put("root", 50);

func edit(k : Text, v : Int) : ?Int {
    if (v > 0) {
        ?(v * 2);
    } else {
        null;
    };
};

let mapFilter : HashMap.HashMap<Text, Int> = HashMap.mapFilter<Text, Int, Int>(
    map,
    Text.equal,
    Text.hash,
    edit,
);
mapFilter.get("root")
```

RBTree

The *convention* is to name the *module alias* after the *file name* it is defined in:

```
import RBTree "mo:base/RBTree";
```

On this page

Type `RBTree.Color`

Type `RBTree.Tree<K, V>`

Class `RBTree.RBtree<K, V>`

Class methods

Method `share`

Method `unShare`

Method `get`

Method `replace`

Method `put`

Method `delete`

Method `remove`

Method `entries`

Method `entriesRev`

Module public functions

Function `iter`

Function `size`

Type `RBTree.Color`

```
import RBTree "mo:base/RBTree";
```

```
type Color = RBTree.Color;
```

```
type RedBlue = Color;
```

Type RBTree.Tree<K, V>

```
import RBTree "mo:base/RBTree";  
  
type Tree<Text, Int> = RBTree.Tree<Text, Int>;  
  
type MyTree = Tree<Text, Int>;
```

Class RBTree.RBtree<K,V>

```
class RBTree<K, V>(compare : (K, K) -> 0.Order)
```

To construct a rbtree object, we use the `RBTree` class:

```
import RBTree "mo:base/RBTree";  
import Text "mo:base/Text";  
  
let textIntTree = RBTree.RBTree<Text, Int>(Text.compare);
```

Class methods

rbtree.share

```
func share() : Tree<K, V>
```

The function `share` takes no argument and returns an value of type `Tree<K, V>`.

Example

```
import RBTree "mo:base/RBTree";
import Text "mo:base/Text";

let textIntTree = RBTree.RBTree<Text, Int>(Text.compare);
textIntTree.put("bitcoin", 1);
textIntTree.put("ETH", 2);

let share : RBTree.Tree<Text, Int> = textIntTree.share();
let treeSize : Nat = RBTree.size(share)
```

rbtree.unShare

```
func unShare(t : Tree<K, V>) : ()
```

Parameters	
Variable argument	t : Tree<K, V>
Return type	()

Example

```
import RBTree "mo:base/RBTree";
import Text "mo:base/Text";
import Iter "mo:base/Iter";

let textIntTree = RBTree.RBTree<Text, Int>(Text.compare);
textIntTree.put("bitcoin", 1);
textIntTree.put("ETH", 2);

let share : RBTree.Tree<Text, Int> = textIntTree.share();

let unshare : () = textIntTree.unshare(share);

let iter : Iter.Iter<(Text, Int)> = textIntTree.entries();

let array : [(Text, Int)] = Iter.toArray(iter)
```

rbtree.get

```
func get(key : K) : ?V
```

Parameters	
Variable argument	key : K
Return type	?V

Example

```
import RBTree "mo:base/RBTree";
import Text "mo:base/Text";

let textIntTree = RBTree.RBTree<Text, Int>(Text.compare);
textIntTree.put("bitcoin", 1);
textIntTree.put("ETH", 2);

let get : ?Int = textIntTree.get("bitcoin");
```

rbtree.replace

```
func replace(key : K, value : V) : ?V
```

Parameters	
Variable argument1	key : K
Variable argument2	value : V
Return type	?V

Example

```
import RBTree "mo:base/RBTree";
import Text "mo:base/Text";

let textIntTree = RBTree.RBTree<Text, Int>(Text.compare);
textIntTree.put("bitcoin", 1);
textIntTree.put("ETH", 2);

let replace : ?Int = textIntTree.replace("bitcoin", 2);
```

rbtree.put

```
func put(key : K, value : V) : ()
```

Parameters	
Variable argument1	key : K
Variable argument2	value : V
Return type	()

Example

```
import RBTree "mo:base/RBTree";
import Text "mo:base/Text";
import Iter "mo:base/Iter";

let textIntTree = RBTree.RBTree<Text, Int>(Text.compare);
textIntTree.put("bitcoin", 1);
textIntTree.put("ETH", 2);
textIntTree.put("ICP", 3);

let iter : Iter.Iter<(Text, Int)> = textIntTree.entries();

let array : [(Text, Int)] = Iter.toArray(iter)
```

rbtree.delete

```
func delete(key : K) : ()
```

Parameters	
Variable argument	key : K
Return type	()

Example

```
import RBTree "mo:base/RBTree";
import Text "mo:base/Text";
import Iter "mo:base/Iter";

let textIntTree = RBTree.RBTree<Text, Int>(Text.compare);
textIntTree.put("bitcoin", 1);
textIntTree.put("ETH", 2);
textIntTree.put("ICP", 3);

let delete : () = textIntTree.delete("bitcoin");

let iter : Iter.Iter<(Text, Int)> = textIntTree.entries();

let array : [(Text, Int)] = Iter.toArray(iter)
```

rbtree.remove

```
func remove(key : K) : ?V
```

Parameters	
Variable argument	key : K
Return type	?V

Example

```
import RBTree "mo:base/RBTree";
import Text "mo:base/Text";
import Iter "mo:base/Iter";

let textIntTree = RBTree.RBTree<Text, Int>(Text.compare);
textIntTree.put("bitcoin", 1);
textIntTree.put("ETH", 2);
textIntTree.put("ICP", 3);

let remove : ?Int = textIntTree.remove("bitcoin");

let iter : Iter.Iter<(Text, Int)> = textIntTree.entries();

let array : [(Text, Int)] = Iter.toArray(iter)
```

rbtree.entries

```
func entries() : I.Iter<(K, V)>
```

The function `entries` takes no argument and returns an value of type `I.Iter<(K, V)>`.

Example

```
import RBTree "mo:base/RBTree";
import Text "mo:base/Text";
import Iter "mo:base/Iter";

let textIntTree = RBTree.RBTree<Text, Int>(Text.compare);
textIntTree.put("bitcoin", 1);
textIntTree.put("ETH", 2);
textIntTree.put("ICP", 3);

let entries : Iter.Iter<(Text, Int)> = textIntTree.entries();

let array : [(Text, Int)] = Iter.toArray(entries)
```

rbtree.entriesRev

```
func entriesRev() : I.Iter<(K, V)>
```

Parameters	
Variable argument	()
Return type	I.Iter<(K, V)>

Example

```
import RBTree "mo:base/RBTree";
import Text "mo:base/Text";
import Iter "mo:base/Iter";

let textIntTree = RBTree.RBTree<Text, Int>(Text.compare);
textIntTree.put("bitcoin", 1);
textIntTree.put("ETH", 2);
textIntTree.put("ICP", 3);

let entriesRev : Iter.Iter<(Text, Int)> = textIntTree.entriesRev();

let array : [(Text, Int)] = Iter.toArray(entriesRev)
```

Module public functions

RBTree.iter

```
func iter<X, Y>(
    tree : Tree<X, Y>,
    direction : {#fwd; #bwd}
) : I.Iter<(X, Y)>
```

Parameters	
Generic parameters	X, Y
Variable argument1	tree : Tree<X< Y>

Parameters	
Variable argument2	direction : {#fwd; #bwd}
Return type	I.Iter<(X, Y)>

Example

```

import RBTree "mo:base/RBTree";
import Text "mo:base/Text";
import Iter "mo:base/Iter";

let textIntTree = RBTree.RBTree<Text, Int>(Text.compare);
textIntTree.put("bitcoin", 1);
textIntTree.put("ETH", 2);
textIntTree.put("ICP", 3);

let share : RBTree.Tree<Text, Int> = textIntTree.share();

let iter : Iter.Iter<(Text, Int)> = RBTree.iter<Text, Int>(share, #bwd);

let array : [(Text, Int)] = Iter.toArray(iter)

```

RBTree.size

```

func size<X, Y>(
    t : Tree<X, Y>
) : Nat

```

Parameters	
Generic parameters	X, Y
Variable argument	t : Tree<X<< Y>>
Return type	Nat

Example

```
import RBTree "mo:base/RBTree";
import Text "mo:base/Text";

let textIntTree = RBTree.RBTree<Text, Int>(Text.compare);
textIntTree.put("bitcoin", 1);
textIntTree.put("ETH", 2);

let share : RBTree.Tree<Text, Int> = textIntTree.share();
let treeSize : Nat = RBTree.size(share)
```

More Data Structures

Besides the more [common data structures](#), the Motoko Base Library also includes other useful data structures.

We will not cover them in this book, but we include them as a reference to the [official docs](#).

AssocList

The [AssocList](#) module provides a [linked-list](#) of key-value pairs of generic type `(K, V)`.

Deque

The [Deque](#) module provides a double-ended queue of a generic element type `T`.

Stack

The [Stack](#) module provides a class `Stack<X>` for a minimal *Last In, First Out* stack of elements of type `X`.

IC APIs

An overview of modules for the Internet Computer System API's.

Time

The *convention* is to name the *module alias* after the *file name* it is defined in.

The time module exposes one function `now` that returns the IC system time represented as nanoseconds since 1970-01-01 as an `Int`.

```
import Time "mo:base/Time";

let currentTime : Int = Time.now();
```

Time is constant within `async` call

The system time is **constant** within one `async` function call and any sub calls.

```
import Time "mo:base/Time";

actor {
    type Time = Time.Time;

    func time1() : Time { Time.now() };

    public shared query func time2() : async Time { Time.now() };

    public shared func time() : async (Time, Time, Time) {
        let t1 = time1();
        let t2 = await time2();
        let t3 = Time.now();

        (t1, t2, t3);
    };
};
```

We `import` the `module` and declare an `actor`. The Time module exposes one `type Time` that that is equal to `Int`. We bring it into scope by `renaming` it.

We then declare a *private function* `time1` and a *query function* `time2` that both return the system `Time`. And we declare a third *update function* `time` that calls the first function, *awaits* the second function and request the system time once more.

All three time values returned in the `tuple` will be *equal*.

Monotonically increasing time

The time, as observed by the canister smart contract, is *monotonically increasing* after each function call. This is also the case across [canister upgrades](#).

This means that we are guaranteed to get an increasingly larger `Time` value when calling our function `time` multiple times.

Timer

Timers on the IC can be set to schedule *one-off* or *periodic tasks* using the `setTimer` and `recurringTimer` functions respectively.

These functions take a `Duration` variant that specifies the time in either seconds or nanoseconds. Also, a callback function (the *job* to be performed) with type `() -> async ()` is passed to the timer functions that will be called when the timer expires.

`setTimer` and `recurringTimer` return a `TimerId` that can be used to cancel the timer before it expires. Timers are canceled with `cancelTimer`.

The *convention* is to name the *module alias* after the *file name* it is defined in.

```
import Timer "mo:base/Timer";
```

On this page

[Type Duration](#)

[Type TimerId](#)

[Function setTimer](#)

[Function recurringTimer](#)

[Function cancelTimer](#)

Type Duration

```
type Duration = {#seconds : Nat; #nanoseconds : Nat}
```

Type TimerId

```
type TimerId = Nat
```

Timer.setTimer

```
func setTimer(d : Duration, job : () -> async ()) : TimerId
```

Timer.recurringTimer

```
func recurringTimer(d : Duration, job : () -> async ()) : TimerId
```

Timer.cancelTimer

```
func cancelTimer : TimerId -> ()
```

Certified Data

Recall that *query* calls do not have the same security guarantees as *update* calls because query calls do not 'go through' consensus.

To verify the authenticity of data returned by a query call, we can use the `CertifiedData` API. This is an advanced feature that we are mentioning here for completeness, but we will not cover it in depth.

For more information visit:

- The [Official Docs](#)
- The [Official Base library Reference](#)

The *convention* is to name the *module alias* after the *file name* it is defined in.

```
import CertifiedData "mo:base/CertifiedData";
```

On this page

[Function set](#)

[Function getCertificate](#)

CertifiedData.set

```
func set : (data : Blob) -> ()
```

CertifiedData.getCertificate

```
func getCertificate : () -> ?Blob
```

Random

The *convention* is to name the *module alias* after the *file name* it is defined in.

```
import Random "mo:base/Random";
```

On this page

[Class Finite](#)

[Function blob](#)
[Function byte](#)
[Function coin](#)
[Function range](#)
[Function binomial](#)
[Function byteFrom](#)
[Function coinFrom](#)
[Function rangeFrom](#)
[Function binomialFrom](#)

Randomness on the IC

The IC provides a source of randomness for use in canister smart contracts. For non-cryptographic use cases, it is relatively simple to use, but for cryptographic use case, some care is required, see the [Official Base Library Reference](#) and the [Official Docs](#) for more information.

To obtain random numbers you need a *source of entropy*. Entropy is represented as a 32 byte [Blob](#) value. You may provide your own entropy [Blob](#) as a seed for random numbers or request 'fresh' entropy from the IC. Requesting entropy from the IC can only be done from within a [shared](#) function.

After obtaining a blob of entropy, you may use the [Finite](#) class to instantiate an object with methods to 'draw' different forms of randomness, namely:

- A random [Nat8](#) value with [byte](#)
- A random [Bool](#) value with [coin](#)
- A random [Nat](#) value within a (possibly) large range with [range](#)
- A random [Nat8](#) value of the number of heads in a series of [n](#) coin flips with [binomial](#)

When your entropy is 'used up', the `Finite` class methods will return `null`.

Alternatively, you may use `byteFrom`, `coinFrom`, `rangeFrom` and `binomialFrom` to obtain a single random value from a given `Blob` seed.

Class Finite

```
class Finite(entropy : Blob)
```

Random.blob

```
func blob : shared () -> async Blob
```

Random.byte

```
func byte() : ?Nat8
```

Random.coin

```
func coin() : ?Bool
```

Random.range

```
func range(p : Nat8) : ?Nat
```

Random.binomial

```
func binomial(n : Nat8) : ?Nat8
```

Random.byteFrom

```
func byteFrom(seed : Blob) : Nat8
```

Random.coinFrom

```
func coinFrom(seed : Blob) : Bool
```

Random.rangeFrom

```
func rangeFrom(p : Nat8, seed : Blob) : Nat
```

Random.binomialFrom

```
func binomialFrom(n : Nat8, seed : Blob) : Nat8
```

Experimental Motoko Modules

We reference these modules for completeness, but will not cover them in depth.

NOTE

These system APIs are experimental and may change in the future.

ExperimentalCycles

This module allows for working with cycles during canister calls.

See the [official docs](#) and the [Base Library Reference](#) for more information.

ExperimentalInternetComputer

This module exposes low-level system functions for calling other canisters and instruction counting.

See the [Base Library Reference](#) for more information.

ExperimentalStableMemory

This module exposes low-level system functions for working with stable memory.

See [Stable Storage](#) and the [Base Library Reference](#) for more information.

Advanced Concepts

This chapter covers *advanced concepts* in Motoko and the Internet Computer (IC). After this chapter you will have enough knowledge to build fully on-chain Web3 applications!

Motoko and the Internet Computer

Motoko is a programming language that is designed to write *advanced smart contracts* on the IC. As a programmer, you don't need to understand the deeper details of how the IC blockchain functions. But since Motoko code is written for the IC, it is important for an advanced programmer to be familiar with some concepts of the IC.

Actors written in Motoko have direct access to *system functionality* on the IC. We have already seen dedicated syntax for IC features, like **stable variables** and **caller authentication**.

In this chapter we will dive deeper into Motoko and the features that allow us to utilize many important features of the IC.

Async Programming

The asynchronous programming paradigm of the Internet Computer is based on the [actor model] (https://en.wikipedia.org/wiki/Actor_model).

Actors are isolated units of code and state that communicate with each other by calling each others' **shared functions** where each shared function call triggers one or more **messages** to be sent and executed.

To master Motoko programming on the IC, we need to understand how to write *asynchronous code*, how to **correctly handle async calls using try-catch** and how to safely modify state in the presence of concurrent activity.

NOTE

From now on, we will drop the optional `shared` keyword in the declaration of `shared functions` and refer to these functions as simply 'shared functions'.

On this page

Async and Await

Inter-Canister Calls

Importing other actors

Inter-Canister Query Calls

Messages and Atomicity

Atomic functions

Shared functions that `await`

Atomic functions that send messages

State commits and message sends

Messaging restrictions

Message ordering

Errors and Traps

Errors

Traps

Async* and Await*

`await` and `await*`

Atomic `await*`

Non-atomic `await*`

Keeping track of state commits and message sends

Table of asynchronous functions in Motoko

Try-Catch Expressions

Async and Await

A call to a [shared function](#) immediately returns a *future* of type `async T`, where T is a [shared type](#).

A future is a value representing the (future) result of a concurrent computation. When the computation completes (with either a value of type `T` or an [error](#)), the result is stored in the future.

A *future* of type `async T` can be *awaited* using the `await` keyword to retrieve the value of type `T`. Execution is paused at the await until the future is complete and produces the result of the computation, by returning the value of type `T` or [throwing the error](#).

- Shared function calls and awaits are only allowed in an [asynchronous context](#).
- Shared function calls immediately return a future but, unlike ordinary function calls, do not suspend execution of the caller at the call-site.
- Awaiting a future using `await` suspends execution until the result of the future, a value or [error](#), is available.

NOTE

While execution of a shared function call is suspended at an `await` or `await`, other messages can be processed by the actor, possibly changing its state.*

Actors can call their own [shared functions](#). Here is an actor calling its own shared function from another shared function:

```
actor A {
    public query func read() : async Nat { 0 };

    public func call_read() : async Nat {
        let future : async Nat = read();

        let result = await future;

        result + 1
    };
}
```

The first call to `read` immediately returns a future of type `async Nat`, which is the [return type](#) of our [shared query](#) function `read`. This means that the caller does not wait for a response from `read` and immediately continues execution.

To actually retrieve the `Nat` value, we have to `await` the future. The actor sends a [message](#) to itself with the request and halts execution until a response is received.

The result is a `Nat` value. We increment it in the code block after the `await` and use as the return value for `call_read`.

NOTE

Shared functions that `await` are not atomic. `call_read()` is executed as several separate messages, see [shared functions that `await`](#).

Inter-Canister Calls

Actors can also call the shared functions of other actors. This always happens from within an [asynchronous context](#).

Importing other actors

To call the shared functions of other actors, we need the [actor type](#) of the external actor and an [actor reference](#).

```

actor B {
    type ActorA = actor {
        read : query () -> async Nat;
    };

    let actorA : ActorA = actor ("7po3j-syaaa-aaaal-qbqea-cai");

    public func callActorA() : async Nat {
        await actorA.read();
    };
}

```

The actor type we use may be a *supertype* of the external actor type. We declare the actor type `actor { }` with only the shared functions we are interested in. In this case, we are importing the actor from the previous example and are only interested in the `query function read`.

We declare a variable `actorA` with actor type `ActorA` and assign an actor reference `actor()` to it. We supply the `principal id` of actor A to reference it. We can now refer to the shared function `read` of actor A as `actorA.read`.

Finally, we `await` the shared function `read` of actor A yielding a `Nat` value that we use as a return value for our `update function callActorA`.

Inter-Canister Query Calls

Calling a `shared function` from an actor is currently (May 2023) only allowed from inside an `update function` or `oneway function`, because query functions don't yet have `message send capability`.

'Inter-Canister Query Calls' will be available on the IC in the future. For now, only ingress messages (from `external clients` to the IC) can request a fast query call (`without going through consensus`).

Messages and atomicity

From the [official docs](#):

A message is a set of consecutive instructions that a subnet executes for a canister.

We will not cover the terms 'instruction' and 'subnet' in this book. Lets just remember that a single call to a `shared function` can be split up into several messages that execute separately.

A call to a shared function of any actor A, whether from an *external client*, from itself or from another actor B (as an [Inter-Canister Call](#)), results in an *incoming message* to actor A.

A single message is executed *atomically*. This means that the code executed within one message either executes successfully or not at all. This also means that any *state changes* within a single message are either all committed or none of them are committed.

These state changes also include any messages sent as calls to shared functions. These calls are queued locally and only sent on successful commit.

An `await` ends the current message and splits the execution of a function into separate messages.

Atomic functions

An atomic function is one that executes within one single message. The function either executes successfully or has no effect at all. If an atomic function fails, we know for sure its state changes have not been committed.

If a shared function does not `await` in its body, then it is executed atomically.

```
actor {
    var state = 0;

    public func atomic() : async () {
        state += 11; // update the state

        let result = state % 2; // perform a computation

        state := result; // update state again
    };

    public func atomic_fail() : async () {
        state += 1; // update the state

        let x = 0 / 0; // will trap

        state += x; // update state again
    };
}
```

Our function `atomic` mutates the state, performs a computation and mutates the `state` again. Both the computation and the state mutations belong to the same message execution. The whole function either succeeds and commits its final state or it fails and does not commit any changes at all. Moreover, the intermediate state changes are not observable from other messages. Execution is all or nothing (i.e. transactional).

The second function `atomic_fail` is another atomic function. It also performs a computation and state mutations within one single message. **But this time, the computation traps and both state mutations are not committed, even though the trap happened after the first state mutation.**

Unless an `Error` is thrown *intentionally* during execution, the order in which computations and state mutations occur is not relevant for the atomicity of a function. The whole function is executed as a single message that either fails or succeeds in its entirety.

Shared functions that await

The `async-await` example earlier looks simple, but there is a lot going on there. The function call is executed as several separate messages:

The first line `let future : async Nat = read()` is executed as part of the first message.

The second line `let result = await future;` keyword ends the first message and any state changes made during execution of the message are committed. The `await` also calls `read()` and suspends execution until a response is received.

The call to `read()` is executed as a separate message and could possibly be executed remotely in another actor, see [inter-canister calls](#). The message sent to `read()` could possibly result in several messages if the `read()` function also `await`s in its body.

In this case `read()` can't `await` in its body because it is a [query function](#), but if it were an [update](#) or [oneway](#) function, it would be able to send [messages](#).

If the sent message executes successfully, a *callback* is made to `call_read` that is executed as yet another message as the last line `result + 1`. The callback writes the result into the future, completing it, and resumes the code that was waiting on the future, i.e. the last line `result + 1`.

In total, there are **three messages**, two of which are executed inside the calling actor as part of `call_read` and one that is executed elsewhere, possibly a remote actor. In this case `actor A` sends a message to itself.

NOTE

Even if we don't `await` a future, a `message` could still be sent and `remote code execution` could be initiated and change the state remotely or locally, see [state commits](#).

Atomic functions that send messages

An atomic function may or may not mutate local state itself, but execution of that atomic function could still cause multiple messages to be sent (by calling shared functions) that each may or may not execute successfully. In turn, these callees may change local or remote state, see [state commits](#).

```
actor {
    var s1 = 0;
    var s2 = 0;

    public func incr_s1() : async () {
        s1 += 1;
    };

    public func incr_s2() : async () {
        s2 += 1;
        ignore 0 / 0;
    };

    // A call to this function executes successfully
    // and increments `s1`, but not `s2`
    public func atomic() : async () {
        ignore incr_s1();
        ignore incr_s2();
    };
};
```

We have two state variables `s1` and `s2` and two shared functions that mutate them. Both functions `incr_s1` and `incr_s2` are each executed atomically as single messages. (They do not `await` in their body).

`incr_s1` should execute successfully, while `incr_s2` will trap and revert any state mutation.

A call to `atomic` will execute successfully without mutating the state during its own execution. When `atomic` exits successfully [with a result](#), the calls to `incr_s1` and `incr_s2` are committed (without `await`) and two separate messages are sent, see [state commits](#).

Now, `incr_s1` will mutate the state, while `incr_s2` does not. The values of `s1` and `s2`, after the successful atomic execution of `atomic` will be `1` and `0` respectively.

These function calls could have been calls to shared function in remote actors, therefore initiating remote execution of code and possible remote state mutations.

NOTE

We are using the `ignore` keyword to ignore return types that are not the empty tuple `()`. For

example, `0 / 0` should in principle return a `Nat`, while `incr_s1()` returns a future of type `async ()`. Both are ignored to resume execution.

State commits and message sends

There are several points during the execution of a shared function at which *state changes* and *message sends* are irrevocably committed:

1. Implicit exit from a shared function by producing a result

The function executes until the end of its body and produces the expected return value with which it is declared. If the function did not `await` in its body, then it is executed atomically within a single message and state changes will be committed once it produces a result.

```
var x = 0;

public func mutate_atomically() : async Text {
    x += 1;
    "changed state and executed till end";
};
```

2. Explicit exit via return

Think of an early return like:

```
var y = 0;

public func early_return(b : Bool) : async Text {
    y += 1; // mutate state

    if (b) return "returned early";

    y += 1; // mutate state

    "executed till end";
};
```

If condition `b` is `true`, the `return` keyword ends the current message and state is committed up to that point only. If `b` is true, `y` will only be incremented once.

3. Explicit throw expressions

When an `error` is thrown, the state changes up until the error are committed and execution of the current message is stopped.

4. Explicit `await` expressions

As we have seen in the [shared functions that `await`](#) example, an `await` ends the current message, commits state up to that point and splits the execution of a function into separate messages.

[See official docs](#)

Messaging Restrictions

The Internet Computer places restrictions on when and how canisters are allowed to communicate. In Motoko this means that there are restrictions on when and where the use of *async expressions* is allowed.

An expression in Motoko is said to occur in an *asynchronous context* if it appears in the body of an `async` or `async*` expression.

Therefore, calling a shared function from outside an asynchronous context is not allowed, because calling a shared function requires a message to be sent. For the same reason, calling a shared function from an [actor class](#) constructor is also not allowed, because an actor class is not an asynchronous context and so on.

Examples of messaging restrictions:

- Canister installation can execute code (and possibly [trap](#)), but not send messages.
- A canister query method cannot send messages ([yet](#))
- The `await` and `await*` constructs are only allowed in an asynchronous context.
- The `throw` expression (to throw an [error](#)) is only allowed in an asynchronous context.
- The `try-catch` expression is only allowed in an asynchronous context. This is because error handling is supported for *messaging errors* only and, like messaging itself, confined to asynchronous contexts.

[See official docs](#)

Message ordering

Messages in an actor are always executed sequentially, meaning one after the other and never in parallel. As a programmer, you have no control over the order in which *incoming messages* are executed.

You can only control the order in which you send messages to other actors, with the guarantee that, for a particular destination actor, they will be executed in the order you sent them. Messages sent to different actors may be executed out of order.

You have no guarantee on the order in which you receive the results of any messages sent.

Consult the [official docs](#) for more information on this topic.

Errors and Traps

During the execution of a message, an *error* may be thrown or a *trap* may occur.

Errors

An `Error` is thrown *intentionally* using the `throw` keyword to inform a caller that something is not right.

- State changes up until an error within a message are committed.
- Code after an error within a message is **NOT** executed, therefore state changes after an error within a message are **NOT** committed.
- Errors can be handled using `try-catch` expressions.
- Errors can only be thrown in an [asynchronous context](#).
- To work with errors we use the `Error` module in the [base library](#).

```
import Error "mo:base/Error";

actor {
    var state = 0;

    public func incrementAndError() : async () {
        state += 1;
        throw Error.reject("Something is not quite right, but I'm aware of it");
        state += 1;
    };
}
```

We import the `Error` module.

We have a shared functions `incrementAndError` that mutates `state` and throws an `Error`. We increment the value of `state` once before and once after the error throw. The function does not return `()`. Instead it results in an error of type `Error` (see `Error` module).

The first state mutation is committed, but the second is not.

After `incrementAndError`, our mutable variable `state` only incremented once to value `1`.

Traps

A trap is an *unintended* non-recoverable runtime failure caused by, for example, division-by-zero, out-of-bounds array indexing, numeric overflow, cycle exhaustion or assertion failure.

- A trap during the execution of a single message causes the entire message to fail.
- State changes before and after a trap within a message are **NOT** committed.
- A message that traps will return a special error to the sender and those errors can be caught using `try-catch`, like other errors, see [catching a trap](#).
- A trap may occur intentionally for development purposes, see [Debug.trap\(\)](#)
- A trap can happen anywhere code runs in an actor, not only in an [asynchronous context](#).

```
import Debug "mo:base/Debug";

actor {
    var state = 0;

    public func incrementAndTrap() : async () {
        state += 1;
        Debug.trap("Something happened that should not have happened");
        state += 1;
    };
}
```

We import the `Debug` module.

The shared function `incrementAndTrap` fails and does not return `()`. Instead it causes the execution to trap (see [Debug module](#)).

Both the first and second state mutation are NOT committed.

After `incrementAndTrap`, our mutable variable `state` is not changed at all.

NOTE

Usually a trap occurs without `Debug.trap()` during execution of code, for example at underflow or overflow of [bounded types](#) or other runtime failures, see [traps](#).

[Assertions](#) also generate a trap if their argument evaluates to `false`.

Async* and Await*

Recall that 'ordinary' **shared functions** with `async` return type are part of the *actor type* and therefore publicly visible in the *public API* of the actor. Shared functions provide an **asynchronous context** from which other shared functions can be called and awaited, because awaiting a shared function requires a **message** to be sent.

Private non-shared functions with `async*` return type provide an **asynchronous context** without exposing the function as part of the public API of the actor.

A call to an `async*` function immediately returns a *delayed computation* of type `async* T`. Note the `*` that distinguishes this from the future `async T`. The computation **needs** to be awaited using the `await*` keyword (in any asynchronous context `async` or `async*`) to produce a result. This was not the case with un-awaited `async` calls, see **atomic functions that send messages**.

For demonstration purposes, lets look at an example of a private `async*` function, that does not use its asynchronous context to call other `async` or `async*` functions, but instead only performs a *delayed computation*:

```
actor {
    var x = 0;

    // non-shared private func with `async*` return type
    private func compute() : async* Nat {
        x += 1001;
        x %= 3;
        x;
    };

    // non-shared private func that `await*` in its body
    private func call_compute() : async* Nat {
        let future = compute(); // future has no effect until `await*`
        await* future; // computation is performed here
    };

    // public shared func that `await*` in its body
    public func call_compute2() : async Nat {
        await* compute(); // computation is performed here
    };
};
```

`compute` is a **private function** with `async* Nat` return type. Calling it directly yields a *computation* of type `async* Nat` and resumes execution without blocking. This computation needs to be awaited using `await*` for the computation to actually execute (unlike the case with 'ordinary' `async` **atomic functions that send messages**).

`await*` also suspends execution, until a result is obtained. We could also pass around the computation within our actor code and only `await*` it when we actually need the result.

We `await*` our function `compute` from within an [asynchronous context](#).

We `await*` our function `compute` from within an 'ordinary' shared `async` function `call_compute` or from within another private `async*` like `call_compute2`.

In the case of `call_compute` we obtain the result by first declaring a future and then `await*`ing it. In the case of `call_compute2` we `await*` the result directly.

`compute` and `call_compute` are not part of the [actor type](#) and [public API](#), because they are [private functions](#).

await and await*

Private non-shared `async*` functions can both `await` and `await*` in their body.

await always commits the current state and triggers a new message send, where await*:

- does not commit the current state
- does not necessarily trigger new message sends
- could be executed as part of the current message

An `async*` function that only uses `await*` in its body to await computations of other `async*` functions (that also don't 'ordinarily' `await` in their body), is executed as a single message and is guaranteed to be atomic. This means that either all `await*` expressions within a `async*` function are executed successfully or none of them have any effect at all.

This is different from 'ordinary' `await` where each `await` triggers a new message and splits the function call into [several messages](#). State changes from the current message are then committed each time a new `await` happens.

The call to a private non-shared `async*` function is split up into [several messages](#) only when it executes an ordinary `await` on a future, either directly in its body, or indirectly, by calling `await*` on a computation that itself executes an ordinary `await` on a future.

You can think of `await*` as indicating that this `await*` may perform zero or more ordinary `await`s during the execution of its computation.

NOTE

One way to keep track of possible `awaits` and state mutations within `async*` functions is to use the `Star.m0` library, which is not covered in this book, but recommended.

```
actor {
    var x = 0;

    public func incr() : async () { x += 1 };

    private func incr2() : async* () { x += 1 };

    private func call() : async* () {
        x += 1; // first state mutation
        await incr(); // current state is committed, new message send occurs

        x += 1; // third state mutation
        await* incr2(); // state is not committed, no message send occurs

        // state is committed when function exits successfully
    };
};
```

We `await` and `await*` from within a private non-shared `async*` function named `call`.

The `await` suspends execution of the current message, commits the first state mutation and triggers a new message send. In this case the actor sends a message to itself, but it could have been a call to a remote actor.

The sent message is executed, mutating the state again.

When a result is returned we resume execution of `call` within a second message. We mutate the state a third time.

The `await*` acts as if we substitute the body of `incr2` for the line `await* incr2()`. The third state mutation is **not** committed before execution of `incr2()`. No message is sent.

The third and fourth state mutation are committed when we successfully exit `call()`, see [state commits](#).

Atomic await*

Here's an example of a nested `await*` within an `async*` function calls:

```
actor {
    var x = 0;

    private func incr() : async* () {
        await* incr2();
    };

    private func incr2() : async* () { x += 1 };

    private func atomic() : async* () {
        await* incr();
    };
}
```

Because `incr()` and `incr2()` do not 'ordinarily' `await` in their body, a call to `atomic()` is executed as a single message. It behaves as if we substitute the body of `incr()` for the `await*` `incr()` expression and similarly substitute the body of `incr2()` for the `await* incr2()` expression.

Non-atomic `await*`

The [asynchronous context](#) that `incr()` provides could be used to `await` 'ordinary' `async` functions.

The key difference is that `await` commits the current message and triggers a new message send, where `await*` doesn't.

Here's an example of a 'nested' `await`:

```
actor {
    var x = 0;

    private func incr() : async* () {
        await incr2();
    };

    public func incr2() : async () { x += 1 };

    private func non_atomic() : async* () {
        await* incr();
    };
}
```

This time `incr2()` is a public shared function with `async ()` return type.

Our function `non_atomic()` performs an `await*` in its body. The `await*` to `incr()` contains an 'ordinary' `await` and thus suspends execution until a result is received. A commit of the state is triggered up to that point and a new message is sent, thereby splitting the call to `non_atomic()` into two messages like we [discussed earlier](#).

This happens because `incr2()` uses an 'ordinary' `await` in its body.

Keeping track of state commits and message sends

Consider the following scenario's:

- an `await*` for an `async*` function that performed an 'ordinary' `await` in its body.
- an `await*` for an `async*` function that performed an `await*` in its body (**for an `async*` function that does not 'ordinarily' `await`s in its body**) or performs no awaits at all.

In every case, our code should handle [state commits and message sends](#) correctly and only mutate the state when we intend to do so.

NOTE

The `Star.mo` library declares a [result type](#) that is useful for handling `async*` functions. We highly recommend it, but will not cover it here.

Table of asynchronous functions in Motoko

Visibility	Async context	Return type	Awaited with	CS and TM**
<code>public</code>	yes	<code>async</code> future	<code>await</code>	yes
<code>private</code>	yes	<code>async</code> future	<code>await</code>	yes
<code>private</code>	yes	<code>async*</code> computation	<code>await*</code>	maybe

** Commits state and triggers a new message send.

The `private` function with 'ordinary' `async` return type is not covered in this chapter. It behaves like a public function with 'ordinary' `async` return type, meaning that when `await`ed, state is committed up to that point and a new message send is triggered. Because of its private visibility, it is not part of the [actor type](#).

Try-Catch Expressions

To correctly handle awaits for `async` and `async*` functions, we need to write code that also deals with scenario's in which functions do not execute successfully. This may be due to an `Error` or a `trap`.

In both cases, the failure can be '*caught*' and handled safely using a *try-catch expression*.

Consider the following failure scenario's:

- an `await` or `await*` for an `async` or `async*` function that throws an `Error` .
- an `await` or `await*` for an `async` or `async*` function that `traps` during execution.

In every case, our code should handle function failures (`errors` or `traps`) correctly and only mutate the state when we intend to do so.

In the following examples, we will use `Result<ok, Err>` from the `base library` as the return type of our functions.

Lets `try` to `await*` a private `async*` function and `catch` any possible errors or traps:

```

import Result "mo:base/Result";
import Error "mo:base/Error";

actor {
    type Result = Result.Result<(), (Error.ErrorCode, Text)>

    var state = 0;

    // should return `async ()`, but in some cases throws an Error
    private func error() : async* () {
        state += 1;
        if (state < 10) throw Error.reject("intentional error");
    };

    public func test_error() : async Result {
        // try to `await*` or catch any error or trap
        let result : Result = try {
            await* error();

            #ok();
        } catch (e) {
            let code = Error.code(e);
            let message = Error.message(e);

            #err(code, message);
        };
        result
    };
}

```

We start by importing `Result` and `Error` from the [base library](#) and declare a `Result` type with associated types for our purpose.

Note that our private `async*` function `error()` should, in normal circumstances, return a `()` when `await*` ed. But it performs a check and *intentionally* `throw`s an `Error` in exceptional circumstances to alert the caller that something is not right.

To account for this case, we `try` the function first in a try-block `try {}`, where we `await*` for it. If that succeeds, we know the function returned a `()` and we return the `#ok` variant as our `Result`.

But in case an error is thrown, we `catch` it in the catch-block and give it a name `e`. This error has type `Error`, which is a non-shared type that can only happen in an [asynchronous context](#).

We analyze our error `e` using the methods from the [Error module](#) to obtain the `ErrorCode` and the error message as a `Text`. We return this information as the associated type of our `#err` variant for our `Result`.

Note, we bind the return value of the whole try-catch block expression to the variable `result` to demonstrate that it is indeed an expression that evaluates to a value. In this case, the try-catch expression evaluates to a `Result` type.

Catching a trap

The same try-catch expression can be used to catch a `trap` that occurs during execution of an `async` or `async*` function.

In the case of an `async*` function, you will only be able to catch traps that occur after a proper `await` (in another message). Any traps before that cannot be caught and will roll back the entire computation up to the previous commit point.

A trap would surface as an `Error` with variant `#canister_error` from the type `ErrorCode`. This error could be handled in the `catch` block.

Scalability

Actor Classes

In the same way [classes](#) are *constructor functions* for [objects](#), similarly *actor classes* are *constructors* for actors. An *actor class* is like a template for *programmatically* creating actors of a specific [actor type](#).

But unlike ordinary [public classes](#) (that are usually declared inside a [module](#)), a single *actor class* is written in its own separate `.mo` file and is [imported](#) like a module.

See '[Actor classes](#)' and '[Actor classes generalize actors](#)' in the official documentation for more information.

NOTE

For programmatically managing actor classes, also check out [Actor class management](#)

A simple actor class

```
// Actor class in separate source file `actor-class.mo`
actor class User(username : Text) {
    var name = username;

    public query func getName() : async Text { name };

    public func setName(newName : Text) : async () {
        name := newName
    };
}
```

We use the `actor class` keywords followed by a name with parentheses `User()` and an optional input argument list, in this case `username : Text`.

The body of the actor class, like any actor, may contain [variables](#), [private](#) or [shared functions](#), [type declarations](#), [private](#) [async*](#) [functions](#), etc.

Actor class import

We import the actor class like we would import a [module](#).

```
// `main.mo`
import User "actor-class";
```

We **import** the actor class from `actor-class.mo`, a file in the same directory. We chose the name `User` for the module to represent the actor class.

The **module type** of `User` now looks like this:

```
module {
    type User = {
        getName: query () -> async Text;
        setName: Text -> async ();
    };
    User : (Text) -> async User;
};
```

This module contains two fields:

1. The type of the actor that we can 'spin up' with this actor class.

In this case, the actor type `User` consists of two **shared functions**, one of which is a **query function**.

2. The constructor function that creates an actor of this type.

The function `User` takes a `Text` argument and returns a future `async User` for an actor of type `User`.

NOTE

In the module above, the name `User` is used as the name of a type and a function, see imports. The line `User : (Text) -> async User;` first uses the name `User` as function name and then as a type name in `async User`.

Installing an instance of an actor class

The function `User : (Text) -> async User` can be called and **awaited** from an **asynchronous context** from within a running actor. Lets refer to this actor as the **Parent** actor.

The `await` for `User` initiates an *install* of a new instance of the actor class as a new '**Child**' actor running on the IC.

```
let instance = await User.User("Alice");
```

A new canister is created with the **Parent** actor as the single **controller** of the new canister.

The `await` yields an actor `actor {}` with actor type `User`. This actor can be stored locally in the **Parent** actor and used as a reference to interact with the **Child** actor.

Multi-canister scaling

Whenever you need to scale up your application to multiple actors (running in multiple canisters), you could use actor classes to repeatedly install new instances of the same actor class.

```
// `main.mo`
import Principal "mo:base/Principal";
import Buffer "mo:base/Buffer";

import User "actor-class";

actor {
    let users = Buffer.Buffer<User.User>(1);

    public func newUser(name : Text) : async Principal {
        let instance = await User.User(name);
        users.add(instance);

        Principal.fromActor(instance);
    };
};

};

This actor declares a Buffer of type Buffer<User.User> with User.User (our actor type from our actor class module) as a generic type parameter. The buffer is named users and has initial capacity of 1. We can use this buffer to store instances of newly created actors from our actor class.

```

The shared function `newUser` takes a `Text` and uses that as the argument to `await` the function `User.User`. This yields a new actor named `instance`.

We add the new actor to the buffer (`users.add(instance)`) to be able to interact with it later.

Finally, we return the principal of the new actor by calling `Principal.fromActor(instance)`.

NOTE

On the IC we actually need to provide some `cycles` with the call to the actor constructor `User.User()`. On Motoko Playground, this code may work fine for testing purposes.

Calling child actors

The last example is not very useful in practice, because we can't interact with the actors after they are installed. Lets add some functionality that allows us to call the shared functions of our **Child** actors.

```
// `main.mo`
import Principal "mo:base/Principal";
import Buffer "mo:base/Buffer";
import Error "mo:base/Error";

import User "actor-class";

actor {
    let users = Buffer.Buffer<User.User>(1);

    public func newUser(name : Text) : async Principal {
        let instance = await User.User(name);
        users.add(instance);

        Principal.fromActor(instance);
    };

    public func readName(index : Nat) : async Text {
        switch (users.getOpt(index)) {
            case (?user) { await user.getName() };
            case (null) { throw (Error.reject "No user at index") };
        };
    };

    public func writeName(index : Nat, newName : Text) : async () {
        switch (users.getOpt(index)) {
            case (?user) { await user.setName(newName) };
            case (null) { throw (Error.reject "No user at index") };
        };
    };
};
```

We added two **shared functions** `readName` and `writeName`. They both take a `Nat` to index into the `users` buffer. They use the `getOpt` function ('method' from the [Buffer Class](#)) in a **switch expression** to test whether an actor exists at that index in the buffer.

If an actor exists at that index, we bind the name `user` to the actor instance and so we can call and `await` the shared functions of the **Child** actor by referring to them like `user.getName()`. Otherwise, we throw an `Error`.

Stable Storage

Every canister on the Internet Computer (IC) currently (May 2023) has 4GB of [working memory](#). This memory is wiped out each time the canister is [upgraded](#).

On top of that, each canister currently (May 2023) gets an additional 48GB of storage space, which is called *stable storage*. This storage is persistent and is not wiped out during reinstallation or upgrade of a canister.

We can *directly* use this memory by interacting with *experimental* the [low-level API](#) from [base library](#), but this is not ideal in many cases.

Work is underway for so called *stable data structures* that are initiated and permanently stored in stable memory. See for example [this library](#) for a Stable Hashmap.

Stable memory and upgrades

To preserve the [working memory](#) during [upgrades](#), we may use [system functions](#) like `pre_upgrade` and `post_upgrade` to temporarily store the data in *stable storage* during the upgrade and copied back over to the working memory after the upgrade.

System API's

An overview of the Internet Computer System API's.

Message Inspection

A running canister on the Internet Computer (IC) can receive calls to its *public shared functions* that may take a *message object* that contains information about the *function call*. The function could for example implement *caller authentication* based on the information provided in the message object.

Additionally, the IC provides *message inspection* functionality to inspect *update* or *oneway* calls (and even *update calls to query functions*) and either *accept* or *decline* the call before running a public shared function.

NOTE

*Message inspection is executed by a *single replica* and does not provide the full security guarantees of going through consensus.*

The `inspect` system function

In Motoko, this functionality can be implemented as a *system function* called `inspect`. This function takes a *record* as an argument and returns a `Bool` value. If we *name* the record argument `args`, then the *function signature* looks like this:

```
system func inspect(args) : Bool
```

Note that the *return type* is NOT `async`. Also, this function CANNOT update the *state of the actor*.

How it works

The `inspect` function is run before any external *update* call (via an IC ingress message) to an *update*, *oneway* or *query* function of an *actor*. The call to the *actor* is then *inspected* by the `inspect` system function.

NOTE

The most common way to call a `query` function is through a query call, but query functions could also be called by an update call (less common). The latter calls are slower but more

trustworthy: they require consensus and are charged in cycles. Update calls to query methods are protected by message inspection too, though query call to query methods are not!

The argument to the `inspect` function (an object we call `args` in the function signature above) is provided by the IC and contains information about:

- The name of the `public shared function` that is being called and a function to obtain its arguments
- The `caller` of the function
- The *binary content* of the *message argument*.

The `inspect` function may examine this information about the call and decide to either *accept* or *decline* the call by returning either `true` or `false` respectively.

NOTE

Ingress query calls to `query functions` and any calls from `other canisters` are NOT inspected by the `inspect` system function.

The `inspect` function argument

The argument to the `inspect` system function is provided by the IC. The type of this argument depends on the `type of the actor`. Lets consider an `actor` of the following `actor type`:

```
type myActor = actor {
    f1 : shared () -> async ();
    f2 : shared Nat -> async ();
    f3 : shared Text -> ();
};
```

The functions `f1` and `f2` are *update* functions and `f3` is a *oneway* function. Also, `f2` takes a `Nat` argument and `f3` takes a `Text` argument.

The argument to the `inspect` system function (which we call `args` in this example) for this specific `actor` will be a `record` of the following type:

```
type CallArgs = {
    caller : Principal;
    arg : Blob;
    msg : {
        #f1 : () -> ();
        #f2 : () -> Nat;
        #f3 : () -> Text;
    };
};
```

This record contains three fields with *predefined* names:

- The `caller` field is always of type `Principal`.
- The `arg` field is always of type `Blob`.
- The `msg` field is always a `variant` type and its fields will depend on the `type of the actor` that this `inspect` function is defined in.

The `msg` variant and 'function-argument-accessor' functions

The `msg` `variant` inside the argument for the `inspect` system function will have a field for every public shared function of the `actor`. The variant *field names* `#f1`, `#f2` and `#f3` correspond to the *function names* `f1`, `f2` and `f3`.

But the *associated types* for the variant fields **ARE NOT** the types of the `actor functions`. Instead, the types of the variant fields `#f1`, `#f2` and `#f3` are 'function-argument-accessor' functions that we can call (if needed) inside the `inspect` system function.

In our example these 'function-argument-accessor' function types are:

```
() -> ();
() -> Nat;
() -> Text;
```

These functions return the arguments that were supplied during the call to the public shared function. They always have unit argument type but return the argument type of the corresponding shared function. The return type of each accessor thus depends on the shared function that it is inspecting.

For example function `f2` takes a `Nat`. If we wanted to *inspect* this value, we could call the *associated function* of variant `#f2`, which is of type `() -> Nat`. This function will provide the actual argument passed in the call to the public shared function.

Actor with Message Inspection

Lets implement the `inspect` system function for the example `actor` given above:

```
import Principal "mo:base/Principal";

actor {
    public shared func f1() : async () {};
    public shared func f2(n : Nat) : async () {};
    public shared func f3(t : Text) {};

    type CallArgs = {
        caller : Principal;
        arg : Blob;
        msg : {
            #f1 : () -> ();
            #f2 : () -> Nat;
            #f3 : () -> Text;
        };
    };
}

system func inspect(args : CallArgs) : Bool {
    let caller = args.caller;
    if (Principal.isAnonymous(caller)) { return false };

    let msgArg = args.msg;
    if (msgArg.size() > 1024) { return false };

    switch (args.msg) {
        case (#f1 _) { true };
        case (#f2 f2Args) { f2Args() < 100 };
        case (#f3 f3Args) { f3Args() == "some text" };
    };
};
};
```

We implemented an actor that *inspects* any call to its `public shared functions` and performs checks on `update` and `oneway` functions before they are called.

First, the `Principal` module is imported and the `actor` is declared with three `public shared functions` `f1`, `f2` and `f3`.

Then, we defined `CallArgs` as the record type of the *expected* argument to the `inspect` system function. We then declare the system function and use `CallArgs` to annotate the `args` argument.

We can now access all the information for any function call inside the `inspect` function through the *chosen* name `args`. (We could have chosen any other name)

Inside `inspect` we first check whether the `caller` field of `args`, which is a `Principal`, is equal to the anonymous principal. If so, then `inspect` returns `false` and the call is *rejected*. Anonymous principals can't call any functions of this `actor`.

Another check is performed on the size of the `arg` field value of our `args` record. If the *binary message argument* is larger than `1024` bytes, then the call is *rejected* by returning `false` once more.

Our `inspect` implementation ends with a switch expression that checks every case of the `msg` variant inside `args`.

In the case function `f1` is called, we ignore the possible arguments supplied to it by using the wildcard `_` for the 'function-variable-accessor' function and always *accept* the call by always returning `true`.

In case function `f2` is called, we bind the 'function-argument-accessor' function to the local name `f2Args` and run it to get the value of the argument to `f2`, which is a `Nat`. If this value is smaller then `100`, we *accept* the call, otherwise we *reject*.

In case function `f3` is called, we bind the 'function-argument-accessor' function to the local name `f3Args` and run it to get the value of the argument to `f3`, which is a `Text`. If this value is equal to `"some text"`, we *accept* the call, otherwise we *reject*.

Pattern matching and field renaming

Instead of defining the type `CallArgs` beforehand and referring to the object argument `args` by name, we could use *pattern matching* and rename the fields in the function signature:

```
system func inspect({
    caller : Principal = id;
    msg : {
        #f1 : Any;
        #f2 : () -> Nat;
        #f3 : () -> Text;
    };
}) : Bool {
    switch (msg) {
        case (#f2 f2Args) { f2Args() < 100 };
        case _ { not Principal.isAnonymous(id) };
    };
};
```

We now declared `inspect` with a *record pattern* as the argument.

We pattern match on the `caller` field and rename it to `id`. We could now just refer to `id` inside the function.

By **subtyping**, we are allowed to ignore the `arg` field all together.

And notice that we use the `Any` type as the *associated type* of the `#f1` variant, because we don't care about the variable values supplied in calls to the `f1` function.

Inside the function, we switch on `msg` and only handle the case where function `f2` is called. If the variable value to `f2` is less than `100`, we *accept*, otherwise we *reject* the call.

In all other cases, we check whether `id` is the **anonymous principal**. If it is, we *reject*, otherwise we *accept* the call.

Message inspection vs Caller identifying functions

The `inspect` system function should not solely be used for *secure access control* to `actors`. It may be used to *reject* possibly unwanted calls to an `actor` before they are executed. Without message inspection, all calls are accepted by default, executed and charged for in `cycles`.

Message inspection is executed by a single replica (without full consensus, like a `query call`) and its result could be spoofed by a malicious boundary node.

Secure access control checks can only be performed by:

1. Implementing **public shared functions** in an `actor` which implement the **caller identification** pattern.
2. Additionally guard incoming calls by the `inspect` system function.

Timers

Timers on the IC can be set to schedule *one-off* or *periodic* tasks.

For more information go to:

- The Base Library page on [Timers](#) in this book
- The [official documentation](#) on timers
- The [Official Base Library Reference](#) for more information.

Certified Variables

[Recall](#) that *query* calls do not have the same security guarantees as *update* calls because query calls do not 'go through' consensus.

To verify the authenticity of data returned by a query call, we can use the [CertifiedData](#) API. This is an advanced feature that we are mentioning here for completeness, but we will not cover it in depth.

For more information visit:

- The [Base Library Page](#) of this book
- The [Official Docs](#)
- The [Official Base library Reference](#)

Pre-upgrade and Post-upgrade

For a demonstration of the use of the `preupgrade` and `postupgrade` system functions, see the [tokenized comments example](#).

NOTE

`preupgrade` and `postupgrade` system functions may trap during execution and data may be lost in the process. Work is underway to improve canister upgrades by working with `stable storage`.

Cryptographic Randomness

The IC provides a source of cryptographic randomness for use in canister smart contracts. For more information checkout:

- The [Base Library Page](#) in this Book
- The [Official Base Library Reference](#)
- The [Official Docs](#)

Project Deployment

While [Motoko Playground](#) is very convenient for trying out small snippets of Motoko code, developing canisters requires a development environment with more features.

In this chapter we will look at:

- [Installing](#) the SDK
- [Compile Motoko](#) code into WASM modules
- [Configure](#) and deploy canisters locally
- Managing [identities](#)
- Setup a [cycles wallet](#)
- Deploy to [mainnet](#)

dfx commands

In this chapter we only cover the most essential commands in `dfx`. For a full overview of `dfx` commands, see the [official docs](#).

Remember that all `dfx` commands and subcommands allow us to add the `--help` flag to print out useful information about the use of the command.

Installing the SDK

The [SDK \(Software Development Kit\)](#) or sometimes called the CDK (Canister Development Kit) is a command-line program (and related tools) that you can run on your personal computer to develop canisters on the Internet Computer (IC).

After installing, the main program you will use (to manage and deploy canisters from the command-line) is called `dfx`.

NOTE

As of April 2023: Work is underway for a Graphical User Interface for `dfx`

NOTE

The [official docs](#) provide more information on installing the SDK on all platforms

Install steps

On Linux, MacOS or Windows WSL, we can install and configure the SDK in four steps.

Step 1: Install

Run this script in the terminal:

```
sh -ci "$(curl -fsSL https://internetcomputer.org/install.sh)"
```

This will download and install the `dfx` binary in `/home/USER/bin`.

Step 2: Add to PATH

Add the `/home/USER/bin` directory to your PATH variable by editing your `/home/USER/.bashrc` file. Add these lines to the end of `.bashrc`.

```
#DFX
export PATH="/home/xps/bin/:$PATH"
```

Then run this command to activate the previous step

```
source .bashrc
```

If everything went well, then you can check your installation with

```
dfx --version
```

This should print the version of `dfx` that is installed.

Step 3: Configure networks.json

To configure the local and mainnet networks used by `dfx` create a `networks.json` file in `/home/USER/.config/dfx/networks.json` with the following

```
{
  "local": {
    "bind": "127.0.0.1:8080",
    "type": "ephemeral",
    "replica": {
      "subnet_type": "application"
    }
  },
  "ic": {
    "providers": ["https://mainnet.dfinity.network"],
    "type": "persistent"
  }
}
```

Step 4: Run for the first time

Now run `dfx` for the first time

```
dfx start
```

This should create a version cache for `dfx` located at `/home/USER/.cache/dfinity/versions/`

Dependencies

For `dfx` to work correctly, you need to have [Node.js](#) v16.0.0 (or higher) installed on your system.

Uninstall

To uninstall `dfx` and related files you can run the `uninstall.sh` script. From your home directory run

```
./.cache/dfinity/uninstall.sh
```

NOTE

The [official docs](#) provide more information on installing the SDK on all platforms

Local Deployment

We will setup a new project for a Motoko canister from scratch and demonstrate core functionality of the SDK. To deploy a "Hello World", you may consult the [official docs](#).

dfx project from scratch

After [installing](#) the SDK you can make a new folder for your project. We will call our project `motime`. Make sure your project has the following folder structure:

```
motime
└── dfx.json
└── src/
    └── main.mo
```

dfx.json

The main configuration file is `dfx.json`. Lets make a custom configuration for our project. Copy and paste the following into the `dfx.json` in the root of your project.

```
{
  "canisters": {
    "motime": {
      "type": "motoko",
      "main": "src/main.mo"
    }
  }
}
```

This defines one single canister called `motime`. We specify that this **canister** is based an **actor** written in Motoko by setting `type` field to `"motoko"`. We also specify the Motoko file path in the `main` field.

This is enough for a basic canister build from one single Motoko source code file.

main.mo

Inside the `src` folder, make a file called `main.mo` and put the following actor code inside.

```
actor {
    public query func hello(name : Text) : async Text {
        return "Hello, " # name # "!";
    };
}
```

Starting the local replica

After setting up project files, we need to start a local replica that serves as a local 'testnet' for development purposes. Start the replica by running

```
dfx start
```

Two things should be outputted in your terminal:

- An indication that `networks.json` is used
- A link to a locally running dashboard to monitor your replica

Create empty canister

Now the local replica is running, lets create an empty canister.

```
dfx canister create motime
```

This creates a temporary `.dfx` folder in the root of your project. After this step, you should have a `canister_ids.json` under `.dfx/local/`. This file contains a *canister id* (which is a [principal](#)) of the empty canister now running on your local replica.

Build Motoko code

Now we can compile the Motoko code into a [wasm file](#) by running

```
dfx build motime
```

If the build succeeds, the outputs will be stored in `.dfx/local/canisters/motime/`. This folder contains, amongst other things, a `motime.wasm` file (the compiled Motoko actor) and a `motime.did` file (the [Interface Description](#)).

Installing the wasm in the canister

Now we can install the wasm module in the canister we created.

```
dfx canister install motime
```

If this succeeds, you now have a canister running on your local replica.

Calling the canister

To interact with the running canister from the command line, run this command

```
dfx canister call motime hello motoko
```

The output should be `("Hello, motoko!")` indicating that the function inside `main.mo` was called successfully.

dfx deploy

There is a command that combines the previous steps into one step. You need a running replica before running this command.

```
dfx deploy motime
```

This command creates a canister (if it doesn't exist already), compiles the code and installs the wasm module in one step.

NOTE

For a full overview of dfx commands, see the [official docs](#)

Canister Status

Once we have a (locally) [running canister](#), we can interact with it from the command-line using `dfx`. We can [query call](#) the [management canister](#) for example to retrieve information about the status of a canister.

Once you have a (locally) running canister, you can run the following command to get its status (assuming our canister is called `motime` in `dfx.json`)

```
dfx canister status motime
```

This should return information about our canister, like

- **Controllers** A list of [principals](#) that control the canister
- **Memory size** The [working memory](#) used by the canister
- **Balance** The [cycles balance](#) of the canister
- **Module hash** The [wasm module hash](#) of the canister

The canister status response contains more information, which we didn't describe here. For a full overview of the canister status response and other functions we could call on the [management canister](#), we need to consider the [Candid interface](#) and corresponding Motoko code for that interface.

In [chapter 8](#), we will look at how to call the [management canister](#) from within Motoko.

Identities and PEM Files

When we interact with the Internet Computer (IC), we use a [principal](#) for authentication. In fact, a principal is just a *transformed* version of a *public key* that is derived from a *private key* according to a Digital Signature Scheme.

NOTE

For more detailed information about [digital signatures](#), consult the [IC Interface specification](#)

Users of applications on the IC, will typically use [Internet Identity](#) (or some other authentication tool like a hardware or software wallet) to manage and store private keys and generate signatures to sign calls and authenticate to services.

As a developer and user of `dfx`, you will work with private keys directly. This maybe for testing purposes, where you might want to generate many private keys to emulate many identities, or for deployment purposes, where you may want to have several developer identities that you control, store and backup.

Default identity in `dfx`

`dfx` automatically generates a *default private key* and stores it in a file `.config/dfx/identity/default/identity.pem`. The private key inside looks like this:

```
-----BEGIN EC PRIVATE KEY-----
MHQCAQEEIA6ajHcVaiWbLxxDCZkgS/04sS/PuVvliY0DBHwAm4vfoAcGBSuBBAK
oUQDQgAE2iARSkW7j/wK1HmW4x+PoSBU8XJzHBuPnf1YalhsW8EgVtCKw015eTNI
PazRT3s7hSFMIccji9DJgJlpyJ804w==
-----END EC PRIVATE KEY-----
```

This private key is stored in either encrypted or unencrypted form on your hard disk. In the case of an encrypted private key, you need to enter a password each time you want to use the key. Otherwise, the private key is stored in 'raw' form. Be careful, this is very insecure!

Generating a new identity

We can generate a new identity with `dfx` by running

```
dfx identity new NAME
```

where `NAME` is the new name for your new identity. You will be prompted for a password.

For development, it might be useful to have 'throw away keys' without a password for easy testing of `dfx` features. For this we could run

```
dfx identity new NAME --storage-mode=plaintext
```

This will immediately generate a new private key and store it in folder in `.config/dfx/identity/`.

Managing identities

Once we have multiple identities, we can list them by running

```
dfx identity list
```

This gives us a list of identities that are stored in `.config/dfx/identity/`. The currently selected identity will have an asterisk `*`.

We could switch between them by running

```
dfx identity use NAME
```

where `name` is the identity we want to switch to.

Anonymous identity

Besides the *default identity* (which is unique), `dfx` also allows us to use the *anonymous identity* to interact with our canisters. To switch to the anonymous identity we run

```
dfx identity use anonymous
```

Cycles and ICP

Canisters are charged by the Internet Computer (IC) for using resources (like *memory* and *computation*) in a token called *cycles*. The more resources a canister uses, the more cycles it has to pay.

Cycles can only be held by canisters. User **principals** can't directly own cycles on the system.

NOTE

There are different ways a service can associate a cycles balance with a user principal. Each user could get a dedicated cycles canister or cycles balances are accounted for separately while the service keeps the client's cycles in a 'treasury' canister.

Converting ICP into cycles

ICP is the main token of the IC ecosystem. Its price varies and is subject to market fluctuations. ICP is used as a **governance token** inside neurons to vote on proposals in the system. Governance is outside of the scope of this book.

Another use of ICP is the ability to convert it into cycles. There is a ***cycles minting canister*** (CMC) that accepts ICP in exchange for cycles. The CMC is also aware of the current market price of ICP and ensures that *1 trillion cycles* (1T) always costs 1 **SDR**.

This mechanism ensures that the price of cycles remains stable over time. The tokenomics of ICP are outside the scope of this book. For more information on this, see [the official docs](#).

Cycles wallet

As a developer, you need to hold and manage cycles to deploy canisters and pay for resources. For that you need to setup a **cycles wallet**.

Cycles Wallet

Until now, we have been using the *default identity* in `dfx` to deploy a project *locally* on our machine. When we deployed our project, `dfx` automatically created a local *cycles wallet* with some test cycles to deploy and run canisters locally.

When we want to create and deploy canisters on the Internet Computer Mainnet, we need *real* cycles. For that we can use a *cycles wallet canister*.

We recommend to make use of several services that work together to create and manage your cycles wallet:

- [Internet Identity](#)
- [NNS Dapp](#)
- [dfx developer identity](#)

Step 1: Create an Internet Identity anchor

Internet Identity (II) is outside the scope of this book. For more info on how to securely create an II Anchor and use it, please refer [to this page](#).

Step 2: Transfer ICP to your NNS Dapp Wallet

Once you have an II Anchor, you can use it to log in into the [NNS Dapp](#). The NNS Dapp features an ICP Wallet to securely store ICP tokens. To obtain ICP tokens, one typically purchases them on an exchange like Binance or Coinbase and then transfers them from an exchange into the NNS Dapp wallet.

We will not cover the steps for obtaining ICP tokens and will assume that you have some ICP in your NNS Dapp.

Step 3: Create a new canister in the NNS Dapp

In the NNS Dapp, select the 'My Canisters' tab. There you have the option to create a new canister.

A new canister needs an initial amount of cycles. When creating a new canister, you may choose an amount of ICP to convert into cycles to fuel the canister. The conversion rate of ICP / Cycles is **determined** by the real world price of ICP. When you choose an amount of ICP, the corresponding amount of cycles you get is shown. Currently, April 2023, the minimum amount of cycles for creating a new canister is 2T cycles plus a small fee.

Choose an amount of your liking and create a new canister. You should now see your canister in the NNS Dapp. It should say that your canister is using 0 MB of storage, which is correct since your canister is empty and has no **wasm module installed**.

We will use this canister to install a cycles wallet.

Step 4: Add your developer identity as a controller of the canister

Now **create a developer identity** on your computer using **dfx**. If you plan on using this identity with significant amounts of cycles and/or ICP, you should back it up first.

Now get your developer identity principal by **selecting** your identity using **dfx identity use** and running

```
dfx identity get-principal
```

dfx should print out the **textual representation** of your principal. Copy this principal and head back to your NNS Dapp.

Click on your canister and you should see one 'controller' of your canister. This controller is the principal tied to your Internet Identity. We want to add the developer identity from **dfx** to the list of controllers of the canister. To do so, click on 'add controller' and paste your developer identity principal.

You should now see two principals that control your new canister.

Step 5: Deploy a cycles wallet to the canister

Now your developer identity has control over the new canister, we can actually install **wasm modules** on it. In this case, we will install a cycles wallet wasm module in the canister using **dfx**. To do this, you need to copy the principal of your canister from the NNS Dapp and run

```
dfx identity deploy-wallet CANISTER_PRINCIPAL --network ic
```

where 'CANISTER_PRINCIPAL' is the principal of your newly created canister in NNS Dapp.

If the command is successful, you should now have a cycles wallet on the IC Mainnet which is controlled by two principals, namely your:

- Developer Identity
- Internet Identity

`dfx` will add a `wallets.json` file in your identity folder at

`.config/dfx/identity/IDENTITY/wallets.json`

where 'IDENTITY' is your developer identity that you used during the deployment of the cycles wallet. This file associates your cycles wallet with your developer identity. To check this you can:

- Open the file and check whether the canister id in `wallets.json` matches the canister principal in NNS Dapp.
- Run `dfx identity get-wallet --network ic` which should print out the same canister principal

Additionally you may run

```
dfx wallet balance --network ic
```

which should retrieve the cycles balance of what is now your cycles wallet canister. This amount should correspond to the initial amount you converted in step 3.

Using your cycles wallet

Since your cycles wallet on the mainnet is now linked to your developer identity, `dfx` will use it whenever you run commands with the `--network ic` flag. In particular, when you deploy a new canister to the IC Mainnet, your cycles wallet will be used to fuel the canister with cycles. It is therefore important to have a sufficient balance of cycles in your cycles wallet.

You can fuel your cycles wallet with additional cycles by:

- Using the 'add cycles' function in the NNS Dapp. You will need ICP in your NNS Dapp Wallet for that.
- Manually 'top up' your cycles canister wallet using `dfx`. You need ICP on an account in the [ICP Ledger](#) controlled by your developer principle for that.

- Programmatically send ICP to the *cycles minting canister* and request a cycles top-up to your canister.

IC Deployment

Now we have our [wallet canister](#) setup on the Internet Computer (IC) with some [cycles](#), we are ready to deploy our first canister to the IC Mainnet.

Assuming the same project that we [deployed locally](#), we can now use the same commands that we used to create, install and call a canister, only this time we add the `--network ic` flag to indicate to `dfx` that we want to execute these commands on the IC Mainnet.

NOTE

Using the `--network ic` will cost [cycles](#) that will be payed with your [cycles wallet](#)

IC Mainnet Deployment

Make sure you have the same setup as in the [local deployment](#) example.

Create empty canister

First check your cycles balance on your cycles wallet by running `dfx wallet balance --network ic`. Now create a new canister on the IC Mainnet by running

```
dfx canister create motime --network ic
```

If successful, you should see a `canister_ids.json` file in the root of your project. This will contain the canister principal id of your new canister on the IC Mainnet.

NOTE

`dfx` will attempt to fuel the new canister with a default amount of cycles. In case your wallet does not have enough cycles to cover the default amount, you may choose to add the `--with-cycles` flag and specify a smaller amount.

Now, check your wallet balance again (by running `dfx wallet balance --network ic`) to confirm that the amount of cycles for the canister creation has been deducted.

Also, check the cycles balance of the new canister by running

```
dfx canister status motime --network ic
```

Build Motoko code

Now we can compile the Motoko code into a [wasm file](#) by running

```
dfx build motime --network ic
```

If the build succeeds, the outputs will be stored in `.dfx/ic/canisters/motime/`. This folder contains, amongst other things, a `motime.wasm` file (the compiled Motoko actor) and a `motime.did` file (the [Interface Description](#)).

Installing the wasm in the canister

Now we can install the wasm module in the canister we created.

```
dfx canister install motime --network ic
```

If this succeeds, you now have a canister with an actor running on the IC Mainnet.

Calling the canister

To interact with the running canister from the command line, run this command

```
dfx canister call motime hello motoko --network ic
```

The output should be `("Hello, motoko!")` indicating that the function inside `main.mo` was called successfully.

dfx deploy

There is a command that combines the previous steps into one step.

```
dfx deploy motime --network ic
```

This command creates a canister (if it doesn't exist already), compiles the code and installs the wasm module in one step.

Deleting a canister

To delete a canister and retrieve its cycles back to your cycles wallet, we need to first stop the canister by updating its [status](#).

```
dfx canister stop motime --network ic
```

This should stop the canister from running and will allow us now to delete it by running

```
dfx canister delete motime --network ic
```

This should delete the canister and send the remaining cycles back to your cycles wallet. Check this by running `dfx wallet balance --network ic`. Also check that the `canister_ids.json` file in the root of your project has been updated and the old canister principal has been removed.

NOTE

For a full overview of dfx commands, see the [official docs](#)

Common Internet Computer Canisters

In this chapter, we will cover common Internet Computer canisters, namely the:

- [IC Management Canister](#)
- [Ledger Canister](#)
- [Cycles Minting Canister](#)

Local deployment of ledger and cmc canisters

To follow along and run the examples in this chapter, you need to [deploy local](#) instances of the `ledger` and `cmc` canisters.

NOTE

The IC Management Canister is not installed locally, because it's actually a 'pseudo-canister' that does not really exist with code and state on the IC.

Make sure you are using an [identity for development](#) and testing (optionally with encryption disabled for running commands without a password).

Using `dfx 0.14.0`, follow these steps:

Step 1

Open `.config/dfx/networks.json` and change the `subnet_type` to system. The network configuration should look like this:

```
{
  "local": {
    "bind": "127.0.0.1:8080",
    "type": "ephemeral",
    "replica": {
      "subnet_type": "system"
    }
  },
  "ic": {
    "providers": ["https://mainnet.dfinity.network"],
    "type": "persistent"
  }
}
```

Save and close the file.

Step 2

Run `dfx` in any directory with the `--clean` flag:

```
dfx start --clean
```

The replica should start and a link to the replica dashboard should be shown in the terminal.

Step 3

Open a new terminal window (leaving `dfx` running in the first terminal) and run:

```
dfx nns install --ledger-accounts $(dfx ledger account-id)
```

We are adding the `--ledger-accounts` flag with the default account of your identity as the argument. This way, the Ledger Canister is locally initialized with a certain amount of ICP to use for testing.

This command should install many common canisters on your local replica. We will use two of those in this chapter and you should verify that they were installed in the last step. The output should contain (among other canisters)

<code>nns-ledger</code>	<code>ryjl3-tyaaa-aaaaa-aaaba-cai</code>
<code>nns-cycles-minting</code>	<code>rkp4c-7iaaa-aaaaa-aaaca-cai</code>

The canister ids don't change over time and are the same in local replicas and mainnet.

Step 4

Verify the balance of the default account for your identity by running:

```
dfx ledger balance
```

In `dfx 0.14.0` this should print `1000000000.0000000 ICP`.

IC Management Canister

The Internet Computer (IC) provides a *management canister* to manage canisters programmatically. This canister, like any other canister, has a [Candid Interface](#) and can be called by other canisters or [ingress messages](#).

In this chapter, we will only look at a subset of the interface for *canister management*.

On this page	Files
Official full Candid Interface File	<code>ic-management.did</code>
Subset of Interface in Motoko	<code>ic-management-interface.mo</code>
Importing	<code>ic-management-import.mo</code>
Public function calls	<code>ic-management-public-functions.mo</code>

Motoko Interface

This is a subset of the interface as a [Motoko module](#). It only includes *canister management* related types and functions. It is available as `ic-management-interface.mo`

Types

- `canister_id`
- `canister_settings`
- `definite_canister_settings`
- `wasm_module`
- `Self`

Public functions

- `create_canister`
- `install_code`
- `start_canister`
- `canister_status`
- `deposit_cycles`
- `update_settings`

- `stop_canister`
- `uninstall_code`
- `delete_canister`

```
module {
    public type canister_id = Principal;

    public type canister_settings = {
        freezing_threshold : ?Nat;
        controllers : ?[Principal];
        memory_allocation : ?Nat;
        compute_allocation : ?Nat;
    };

    public type definite_canister_settings = {
        freezing_threshold : Nat;
        controllers : [Principal];
        memory_allocation : Nat;
        compute_allocation : Nat;
    };

    public type wasm_module = [Nat8];

    public type Self = actor {
        canister_status : shared { canister_id : canister_id } -> async {
            status : { #stopped; #stopping; #running };
            memory_size : Nat;
            cycles : Nat;
            settings : definite_canister_settings;
            idle_cycles_burned_per_day : Nat;
            module_hash : ?[Nat8];
        };

        create_canister : shared { settings : ?canister_settings } -> async {
            canister_id : canister_id;
        };

        delete_canister : shared { canister_id : canister_id } -> async ();

        deposit_cycles : shared { canister_id : canister_id } -> async ();

        install_code : shared {
            arg : [Nat8];
            wasm_module : wasm_module;
            mode : { #reinstall; #upgrade; #install };
            canister_id : canister_id;
        } -> async ();

        start_canister : shared { canister_id : canister_id } -> async ();

        stop_canister : shared { canister_id : canister_id } -> async ();

        uninstall_code : shared { canister_id : canister_id } -> async ();

        update_settings : shared {
            canister_id : Principal;
            settings : canister_settings;
        } -> async ();
    };
}
```

```
};  
};
```

Import

We import the management canister by importing the interface file and declaring an actor by principle `aaaaaa-aa` and type it as the `Self` (which is declared in the interface).

```
import Interface "ic-management-interface";  
import Cycles "mo:base/ExperimentalCycles";  
  
actor {  
    // The IC Management Canister ID  
    let IC = "aaaaaa-aa";  
    let ic = actor(IC) : Interface.Self;  
}
```

We can now reference the canister as `ic`.

We also imported `ExperimentalCycles` because some of our function calls require cycles to be added.

Public functions

The source file with function calls is [available here](#) including a test to run all functions. You can deploy it [locally](#) for testing.

Create canister

To create a new canister, we call the `create_canister` function.

```
create_canister : shared { settings : ?canister_settings } -> async {  
    canister_id : canister_id;  
};
```

The function may take an optional `canisters_settings` record to set initial settings for the canister, but this argument may be `null`.

The function returns a record containing the `Principal` of the newly created canister.

NOTE

To create a new canister, you must add cycles to the call using the [ExperimentalCycles](#) module

Example

```
var canister_principal : Text = "";

func create_canister() : async* () {
    Cycles.add(10 ** 12);

    let newCanister = await ic.create_canister({ settings = null });

    canister_principal := Principal.toText(newCanister.canister_id);
};
```

Canister status

To get the current status of a canister we call `canister_status`. We only provide a simple record with a `canister_id` (principal) of the canister we are interested in. Only *controllers* of the canister can ask for its settings.

```
canister_status : shared { canister_id : canister_id } -> async {
    status : { #stopped; #stopping; #running };
    memory_size : Nat;
    cycles : Nat;
    settings : definite_canister_settings;
    idle_cycles_burned_per_day : Nat;
    module_hash : ?[Nat8];
};
```

The function returns a record containing the `status` of the canister, the `memory_size` in bytes, the `cycles` balance, a `definite_canister_settings` with its current settings, the `idle_cycles_burned_per_day` which indicates the average cycle consumption of the canister and a `module_hash` if the canister has a wasm module installed on it.

Example

```
var controllers : [Principal] = [];

func canister_status() : async* () {
    let canister_id = Principal.fromText(canister_principal);

    let canisterStatus = await ic.canister_status({ canister_id });

    controllers := canisterStatus.settings.controllers;
};
```

Deposit cycles

To deposit cycles into a canister we call `deposit_cycles`. Anyone can call this function.

We only need to provide a record with the `canister_id` of the canister we want to deposit into.

```
deposit_cycles : shared { canister_id : canister_id } -> async () ;
```

NOTE

To deposit cycles into a canister, you must add cycles to the call using the `ExperimentalCycles` module

Example

```
func deposit_cycles() : async* () {
    Cycles.add(10 ** 12);

    let canister_id = Principal.fromText(canister_principal);

    await ic.deposit_cycles({ canister_id });
};
```

Update settings

To update the settings of a canister, we call `update_settings` and provide the `canister_id` together with the new `canister_settings`.

```
update_settings : shared {
    canister_id : Principal;
    settings : canister_settings;
} -> async ();
```

Example

```
func update_settings() : async* () {
    let settings : Interface.canister_settings = {
        controllers = ?controllers;
        compute_allocation = null;
        memory_allocation = null;
        freezing_threshold = ?(60 * 60 * 24 * 7);
    };

    let canister_id = Principal.fromText(canister_principal);

    await ic.update_settings({ canister_id; settings });
};
```

Uninstall code

To *uninstall* (remove) the wasm module from a canister we call `uninstall_code` with a record containing the `canister_id`. Only controllers of the canister can call this function.

```
uninstall_code : shared { canister_id : canister_id } -> async ();
```

Example

```
func uninstall_code() : async* () {
    let canister_id = Principal.fromText(canister_principal);

    await ic.uninstall_code({ canister_id });
};
```

Stop canister

To *stop* a running canister we call `stop_canister` with a record containing the `canister_id`. Only controllers of the canister can call this function.

```
stop_canister : shared { canister_id : canister_id } -> async ();
```

Example

```
func stop_canister() : async* () {
    let canister_id = Principal.fromText(canister_principal);

    await ic.stop_canister({ canister_id });
};
```

Start canister

To *start* a stopped canister we call `start_canister` with a record containing the `canister_id`. Only controllers of the canister can call this function.

```
start_canister : shared { canister_id : canister_id } -> async () ;
```

Example

```
func start_canister() : async* () {
    let canister_id = Principal.fromText(canister_principal);

    await ic.start_canister({ canister_id });
};
```

Delete canister

To *delete* a stopped canister we call `delete_canister` with a record containing the `canister_id`. Only stopped canisters can be deleted and only controllers of the canister can call this function.

```
delete_canister : shared { canister_id : canister_id } -> async () ;
```

Example

```
func delete_canister() : async* () {
    let canister_id = Principal.fromText(canister_principal);

    await ic.delete_canister({ canister_id });
};
```

Install code

To install a wasm module in a canister, we call `install_code`. Only *controllers* of a canister can call this function.

We need to provide a wasm module install arguments as `[Nat8]` arrays. We also pick a `mode` to indicate whether we are freshly installing or `upgrading` the canister. And finally, we provide the *canister id* (principal) that we want to install code into.

```
install_code : shared {
    arg : [Nat8];
    wasm_module : wasm_module;
    mode : { #reinstall; #upgrade; #install };
    canister_id : canister_id;
} -> async ();
```

This function is *atomic* meaning that it either succeeds and returns `()` or it has no effect.

Test

To test all the functions, we `await*` all of them in a `try-catch` block inside a regular shared public function. This test is available in `ic-management-public-functions.mo`.

```
public func ic_management_canister_test() : async { #OK; #ERR : Text } {
    try {
        await* create_canister();
        await* canister_status();

        await* deposit_cycles();
        await* update_settings();
        await* uninstall_code();

        await* stop_canister();
        await* start_canister();
        await* stop_canister();

        await* delete_canister();

        #OK;
    } catch (e) {
        #ERR(Error.message(e));
    };
};
```

Our function either returns `#OK` or `#ERR` with a caught error message that is converted into text.

ICP Ledger

ICP tokens are held on a canister that implements a token ledger. We refer to it as the Ledger Canister. This ICP Ledger Canister exposes a [Candid Interface](#) with ledger functionality, like sending tokens and querying balances.

We will only focus on the `icrc1` part of the interface. The full interface is available as a file [here](#) and [online in a 'ledger explorer'](#).

For detailed information about the `icrc1` standard, you may review [chapter 9](#).

On this page	Files
Official full Candid Interface File	<code>ic-management.did</code>
Subset of Interface in Motoko	<code>icp-ledger-interface.mo</code>
Importing	<code>icp-ledger-import.mo</code>
Public function calls	<code>icp-ledger-public-functions.mo</code>

Motoko Interface

This is a subset of the interface as a [Motoko module](#). It only includes `icrc1` related types and functions. It is available as `icp-ledger-interface.mo`

Note, that the types are slightly different from the `icrc1` standard, but the functions are the same.

Types

- [Account](#)
- [MetadataValue](#)
- [Result](#)
- [StandardRecord](#)
- [TransferArg](#)
- [TransferError](#)
- [Self](#)

Public functions

- `icrc1_name`
- `icrc1_symbol`
- `icrc1_decimals`
- `icrc1_fee`
- `icrc1_metadata`
- `icrc1_minting_account`
- `icrc1_supported_standards`
- `icrc1_total_supply`
- `icrc1_balance_of`
- `icrc1_transfer`

```

module {
    public type Account = { owner : Principal; subaccount : ?[Nat8] };

    public type MetadataValue = {
        #Int : Int;
        #Nat : Nat;
        #Blob : [Nat8];
        #Text : Text;
    };

    public type Result = { #Ok : Nat; #Err : TransferError };

    public type StandardRecord = { url : Text; name : Text };

    public type TransferArg = {
        to : Account;
        fee : ?Nat;
        memo : ?[Nat8];
        from_subaccount : ?[Nat8];
        created_at_time : ?Nat64;
        amount : Nat;
    };

    public type TransferError = {
        #GenericError : { message : Text; error_code : Nat };
        #TemporarilyUnavailable;
        #BadBurn : { min_burn_amount : Nat };
        #Duplicate : { duplicate_of : Nat };
        #BadFee : { expected_fee : Nat };
        #CreatedInFuture : { ledger_time : Nat64 };
        #TooOld;
        #InsufficientFunds : { balance : Nat };
    };
}

public type Self = actor {
    icrc1_balance_of : shared query Account -> async Nat;

    icrc1_decimals : shared query () -> async Nat8;

    icrc1_fee : shared query () -> async Nat;

    icrc1_metadata : shared query () -> async [(Text, MetadataValue)];

    icrc1_minting_account : shared query () -> async ?Account;

    icrc1_name : shared query () -> async Text;

    icrc1_supported_standards : shared query () -> async [StandardRecord];

    icrc1_symbol : shared query () -> async Text;

    icrc1_total_supply : shared query () -> async Nat;

    icrc1_transfer : shared TransferArg -> async Result;
}

```

```
    }  
}
```

Import

We import the ICP ledger canister by importing the interface file and declaring an actor by principle `ryjl3-tyaaa-aaaaa-aaaba-cai` and type it as the `Self` type (which is declared in the interface).

NOTE

If you are testing locally, you should have the Ledger Canister [installed locally](#).

```
import Interface "icp-ledger-interface";  
  
actor {  
    // The Ledger Canister ID  
    let ICP = "ryjl3-tyaaa-aaaaa-aaaba-cai";  
    let icp = actor(ICP) : Interface.Self;  
}
```

We can now reference the icp ledger canister as `icp`.

Public functions

These functions are available in `icp-ledger-public-functions.mo` together with a test function. To test these functions, please read the [test](#) section.

icrc1_name

```
icrc1_name : shared query () -> async Text;
```

Example

```
func name() : async* Text {  
    await icp.icrc1_name();  
};
```

icrc1_symbol

```
icrc1_symbol : shared query () -> async Text;
```

Example

```
func symbol() : async* Text {
    await icp.icrc1_symbol();
};
```

icrc1_decimals

```
icrc1_decimals : shared query () -> async Nat8;
```

Example

```
func decimals() : async* Nat8 {
    await icp.icrc1_decimals();
};
```

icrc1_fee

```
icrc1_fee : shared query () -> async Nat;
```

Example

```
func fee() : async* Nat {
    await icp.icrc1_fee();
};
```

icrc1_metadata

```
icrc1_metadata : shared query () -> async [(Text, MetadataValue)];
```

Example

```
func metadata() : async* [(Text, Interface.MetadataValue)] {
    await icp.icrc1_metadata();
};
```

icrc1_minting_account

```
icrc1_minting_account : shared query () -> async ?Account;
```

Example

```
func minting_account() : async* ?Interface.Account {
    await icp.icrc1_minting_account();
};
```

icrc1_supported_standards

```
icrc1_supported_standards : shared query () -> async [StandardRecord];
```

Example

```
func supported_standards() : async* [Interface.StandardRecord] {
    await icp.icrc1_supported_standards();
};
```

icrc1_total_supply

```
icrc1_total_supply : shared query () -> async Nat;
```

Example

```
func total_supply() : async* Nat {
    await icp.icrc1_total_supply();
};
```

icrc1_balance_of

```
icrc1_balance_of : shared query Account -> async Nat;
```

Example

```
func balance(acc : Interface.Account) : async* Nat {
    await icp.icrc1_balance_of(acc);
};
```

icrc1_transfer

```
icrc1_transfer : shared TransferArg -> async Result;
```

Example

```
func transfer(arg : Interface.TransferArg) : async* Interface.Result {
    await icp.icrc1_transfer(arg);
};
```

Test

Before we can run the test, we have to

- have a [local instance](#) of the Ledger Canister
- [deploy](#) the `icp-ledger-public-functions.mo` actor locally and name the canister `icp-ledger` in your `dfx.json`.
- transfer some ICP to the canister

If you followed the steps for [local ledger canister deployment](#), you should have ICP on the default account of your identity. Then you can send ICP to your canister with these steps.

Step 1

First export your canister id to an environment variable. Assuming your canister name in `dfx.json` is `icp-ledger` run:

```
export CANISTER_ID=$(dfx canister id icp-ledger)
```

Step 2

Then export your the default account id belonging to your canister principal by using the `dfx ledger account-id` command with the `--of-principal` flag:

```
export ACCOUNT_ID=$(dfx ledger account-id --of-principal $CANISTER_ID)
```

Step 3

Check the ICP balance on your local Ledger of the default accounts of your identity and canister id with these commands.

Identity balance:

```
dfx ledger balance
```

Canister balance:

```
dfx ledger balance $ACCOUNT_ID
```

Step 4

Finally, send ICP to your canister with the `dfx ledger transfer` command:

```
dfx ledger transfer --amount 1 $ACCOUNT_ID --memo 0
```

Now check your balances again to check the transfer.

Step 5

To test all the Ledger public functions, we run this test. (Available in [icp-ledger-public-functions.mo](#))

For testing the `icrc1_balance_of` function, you need to replace the principal with your own principal.

For the `icrc1_transfer` function, we use a specific subaccount by making a 32 byte [Nat8] array with the first byte set to `1`.

```
public func test() : async { #OK : Interface.Result; #ERR : Text } {
    try {

        // Query tests
        ignore await* name();
        ignore await* symbol();
        ignore await* decimals();
        ignore await* fee();
        ignore await* metadata();
        ignore await* minting_account();
        ignore await* supported_standards();
        ignore await* total_supply();

        // Balance_of test
        // Replace with your principal
        let principal : Principal = Principal.fromText("gfpvm-mqv27-7sz2a-
n mav4-isngk-exxnl-g73x3-memx7-u5xbq-3alvq-dqe");

        let account : Interface.Account = {
            owner = principal;
            subaccount = null;
        };

        ignore await* balance(account);

        // Transfer test
        var sub = Array.init<Nat8>(32, 0);
        sub[0] := 1;
        let subaccount = Array.freeze(sub);

        let account2 : Interface.Account = {
            owner = principal;
            subaccount = ?subaccount;
        };

        let arg : Interface.TransferArg = {
            from_subaccount = null;
            to = account2;
            amount = 100_000_000; // 1 ICP;
            fee = null;
            memo = null;
            created_at_time = null;
        };

        let result = await* transfer(arg);

        #OK result

    } catch (e) {
        #ERR(Error.message(e));
    };
};
```

After running this test locally, you should check your canister balance again to verify that the `icrc1_transfer` function was successful and thus the whole test was executed.

Cycles Minting Canister

The Cycles Minting Canister (CMC) can mint cycles by converting [ICP into cycles](#). We can 'top up' a canister with [cycles](#) by sending ICP to the CMC and receiving cycles in return.

Top up a canister locally

The easiest way to top up a canister with cycles is to use the `dfx` command line tool.

For this, we need a locally running test canister to top up. We can use the canister `motime` that we deployed in [this chapter](#).

Assuming your canister name in `dfx.json` is `motime`, run

```
dfx canister status motime
```

This should print your [canister status](#). Please note your cycles balance.

Now top up the canister by running:

```
dfx ledger top-up $(dfx canister id motime) --amount 1
```

This command will automatically convert 1 ICP into cycles and deposit the cycles into your canister.

Now check your canister status again, to see that your cycles balance has increased.

Top up a canister on mainnet

The same `dfx top-up` command can be used to top up a canister running on mainnet. For this to work, you must use an identity that holds ICP on the mainnet Ledger Canister.

Step 1

Make sure you have a [identity set up](#) and print its default account with:

```
dfx ledger account-id
```

Send real ICP to this account. Now check your balance with:

```
dfx ledger balance --network ic
```

Step 2

Assuming you deployed `motime` on mainnet, check its cycle balance:

```
dfx canister status motime --network ic
```

And top it up with:

```
dfx ledger top-up $(dfx canister id motime) --amount 1 --network ic
```

Now check the cycle balance again to verify that it has increased.

Internet Computer Standards

This chapter covers common Internet Computer [Community Standards](#).

ICRC1

Checkout the [official documentation for ICRC1](#)

ICRC1 is a standard for fungible tokens on the Internet Computer (IC). The standard specifies the types of *data*, the *interface* and certain *functionality* for fungible tokens on the IC.

The standard is defined in a [Candid file](#) accompanied by an [additional description](#) of the intended behavior of any ICRC1 token.

NOTE

ICRC is an abbreviation of 'Internet Computer Request for Comments' and is chosen for historical reasons related to token developments in blockchains such as Ethereum and the popular ERC standards (Ethereum Request for Comments)

On this page

ICRC1 Types

- [General Types](#)
- [Account Types](#)
- [Transaction Types](#)

ICRC1 Interface

- [General token info](#)
- [Ledger functionality](#)
- [Metadata and Extensions](#)

Transaction deduplication

- [IC time and Client time](#)
- [Deduplication algorithm](#)

How it works

In essence, an ICRC1 token is an actor that maintains data about *accounts* and *balances*.

The token *balances* are represented in Motoko as simple [Nat](#) values belonging to certain [accounts](#). Transferring tokens just comes down to subtracting from one balance of some account and adding to another balance of another account.

Because an **actor** runs on the IC blockchain (and is therefore tamperproof), we can trust that a correctly programmed actor would never 'cheat' and that it would correctly keep track of all balances and transfers.

NOTE

If the token actor is controlled by one or more entities (its controllers), then its security depends on the trustworthiness of the controllers. A fully decentralized token actor is one that is controlled by either a DAO or no entity at all!

ICRC1 types

The required **data types** for interacting with an ICRC1 token are defined in the official [ICRC-1.did](#) file. We will cover their Motoko counterparts here. We have grouped all ICRC1 types in three groups: **general**, **account** and **transaction** types.

General types

The standard defines three simple types that are used to define **more complex types**.

```
type Timestamp = Nat64;

type Duration = Nat64;

type Value = {
    #Nat : Nat;
    #Int : Int;
    #Text : Text;
    #Blob : Blob;
};
```

The **Timestamp** type is another name for a **Nat64** and represents the number of nanoseconds since the UNIX epoch in UTC timezone. It is used in the definition of **transaction types**.

The **Duration** type is also a **Nat64**. It does not appear anywhere else in the specification but it may be used to represent the number of nanoseconds between two **Timestamp**s.

The **Value** type is a **variant** that could either represent a **Nat**, an **Int**, a **Text** or a **Blob** value. It is used in the **icrc1_metadata** function.

Account types

A *token balance* always belongs to one [Account](#).

```
type Account = {  
    owner : Principal;  
    subaccount : ?Subaccount;  
};  
  
type Subaccount = Blob;
```

An `Account` is a `record` with two fields: `owner` (of type `Principal`) and `subaccount`. This second field is an `optional` `Subaccount` type, which itself is defined as a `Blob`. This blob has to be exactly *32 bytes* in size.

This means that one account is *always* linked to one specific **Principal**. The notion of a *subaccount* allows for every principal on the IC to have **many** ICRC1 accounts.

Default account

Each principal has a *default account* associated with it. The default account is constructed by setting the `subaccount` field to either `null` or supplying a `?Blob` with only `0` bytes.

Account examples

Here's how we declare some accounts:

`account1` is the *default account* for the principal `un4fu-tqaaa-aaaab-qadjq-cai`. It is constructed by first making a `Principal` from the textual representation and making `subaccount` equal to `null`.

For `account2` we make a custom `Blob` from a `[Nat8]` array with 32 bytes. For no particular reason, we set the first four bytes to `1`. We also used *name punning* for the `subaccount` field.

NOTE

We use the same principal (`un4fu-tqaaa-aaaab-qadjq-cai`) for both accounts. A principal can have *many accounts*.

One principal, many accounts

In our [last example](#), we used a `?Blob` with 32 bytes for our `subaccount`. The first 4 bytes already allow for $256 * 256 * 256 * 256 = 4\,294\,967\,296$ different subaccounts.

The maximum amount of subaccounts for each principal is much much larger and equals $256^{32} = 2^{256} = 1.16 * 10^{77}$, a natural number with 78 digits! A number like that can be represented in Motoko by a single `Nat`.

Textual representation of an Account

<https://github.com/dfinity/ICRC-1/pull/98>

Transaction types

The standard specifies two data types for the execution of token transfers (transactions) from one `account` to another. These are the `TransferArgs` `record` and the `TransferError` `variant`, which are used as the argument and part of the return type of the `icrc1_transfer` function.

```

type TransferArgs = {
    from_subaccount : ?Subaccount;
    to : Account;
    amount : Nat;
    fee : ?Nat;
    memo : ?Blob;
    created_at_time : ?Timestamp;
};

type TransferError = {
    #BadFee : { expected_fee : Nat };
    #BadBurn : { min_burn_amount : Nat };
    #InsufficientFunds : { balance : Nat };
    #TooOld;
    #CreatedInFuture : { ledger_time : Timestamp };
    #Duplicate : { duplicate_of : Nat };
    #TemporarilyUnavailable;
    #GenericError : { error_code : Nat; message : Text };
};

```

The `TransferArgs` record has six fields, four of which are **optional** types. Only the `to` and `amount` fields have to be specified for the most basic transaction.

1. `from_subaccount` is an **optional** `Subaccount` (`?Blob`) and specifies whether to use a specific 32 byte subaccount to send from. The sender `Account` (containing a `Principal`) is NOT specified because this will be inferred from the `transfer function`, which is a **caller identifying function**. This ensures that no one can spend tokens except the owner of the account.
2. `to` is an `Account` and specifies the recipients accounts.
3. `amount` is a `Nat` and specifies the amount of tokens to be svariantent measured by the smallest subunits possible of the token (defined by the `token decimals`).
4. `fee` is a `?Nat` that specifies an **optional** fee to be payed by the sender for a transaction measured by the smallest subunits possible of the token (defined by the `token decimals`).
5. `memo` is a `?Blob` and specifies **optional** 32 byte binary data to include with a transaction.
6. `created_at_time` is a `?Timestamp` which specifies an **optional transaction time** for a transaction which maybe used for `transaction deduplication`.

The `TransferError` **variant** is used as the possible error type in the return value of the `icrc1_transfer` function. It specifies several failure scenarios for a transaction.

1. `#BadFee` is returned when something is wrong with the senders fee in `TransferArgs` and informs about the expected fee through its associated type `{ expected_fee : Nat }`.
2. `#BadBurn` is returned when a `burn transaction` tries to burn a too small amount of tokens and informs about the minimal burn amount through `{ min_burn_amount : Nat }`.
3. `#InsufficientFunds` is returned when the sender `Account` balance is smaller than the `amount` to be sent plus fee (if required). It returns the senders balance through `{`

- ```
balance : Nat } .
```
4. `#TooOld` and `#CreatedInFuture` are returned if a transaction is not made within a specific [time window](#). They are used for [transaction deduplication](#).
  5. `#TemporarilyUnavailable` is returned when token transfers are temporarily halted, for example during maintenance.
  6. `#GenericError` allows us to specify any other error that may happen by providing error info through `{ error_code : Nat; message : Text }`.

The records in the associated types of `TransferError` may contain even more information about the failure by [subtyping](#) the records. The same applies for the fields of `TransferArgs`. An implementer of a token may choose to add more fields for their application and still be compliant with the standard.

## ICRC1 Interface

For a token to comply with ICRC1 it must implement a set of specified [public shared functions](#) as part of the [actor type](#). The token implementation may implement more functions in the actor and still remain compliant with the standard, see [actor subtyping](#).

We have grouped the functions into three groups and used an actor type alias `ICRC1_Interface` which is not part of the standard:

```

import Result "mo:base/Result";

type Result<Ok, Err> = Result.Result<Ok, Err>;

type ICRC1_Interface = actor {

 // General token info
 icrc1_name : shared query () -> async (Text);
 icrc1_symbol : shared query () -> async (Text);
 icrc1_decimals : shared query () -> async (Nat8);
 icrc1_fee : shared query () -> async (Nat);
 icrc1_total_supply : shared query () -> async (Nat);

 // Ledger functionality
 icrc1_minting_account : shared query () -> async (?Account);
 icrc1_balance_of : shared query (Account) -> async (Nat);
 icrc1_transfer : shared (TransferArgs) -> async (Result<Nat, TransferError>);

 // Metadata and Extensions
 icrc1_metadata : shared query () -> async ([(Text, Value)]);
 icrc1_supported_standards : shared query () -> async ([{
 name : Text;
 url : Text;
 }]);
};


```

## General token info

The standard specifies five [query functions](#) that take no arguments and return general info about the token.

1. `icrc1_name` returns the name of the token as a `Text`.
2. `icrc1_symbol` return the ticker symbol as a `Text`.
3. `icrc1_decimals` returns the maximum number of decimal places of a unit token as a `Nat8`. Most tokens have 8 decimal places and the smallest subunit would be `0.00_000_001`.
4. `icrc1_fee` returns the fee to be paid for transfers of the token as a `Nat` measured in smallest subunits of the token (defined by the token decimals). The fee may be 0 tokens.
5. `icrc1_total_supply` returns the *current total supply* as a `Nat` measured in smallest subunits of the token (defined by the token decimals). The total supply may change over time.

## Ledger functionality

ICRC1 intentionally excludes some ledger implementation details and does not specify how an actor should keep track of token balances and transaction history. It does specify three important **shared functions** to interact with the ledger, regardless of how the implementer of the standard chooses to implement the ledger.

1. **icrc1\_minting\_account** is a **query function** that takes no arguments and returns **? Account**, an **optional** account. The token ledger *may* have a special account called the *minting account*. If this account exists, it would serve two purposes:
  - Token amounts sent **TO** the minting account would be *burned*, thus removing them from the total supply. This makes a token *deflationary*. Burn transactions have no fee.
  - Token amounts sent **FROM** the minting account would be *minted*, thus freshly created and added to the total supply. This makes a token *inflationary*. Mint transactions have no fee.
2. **icrc1\_balance\_of** is a **query function** that takes an **Account** as an argument and returns the balance of that account as a **Nat** measured in smallest subunits of the token (defined by the token decimals).
3. **icrc1\_transfer** is the main and most important function of the standard. It is the only **update function** that **identifies its caller** and is meant to transfer token amounts from one account to another. It takes the **TransferArgs** record as an argument and returns a result **Result<Nat, TransferError>**.

This function should perform important checks before approving the transfer:

- Sender and receiver account are not the same
- Sender has a balance bigger than amount to be sent plus fee
- Sender fee is not too small
- The subaccounts and memo are exactly 32 bytes
- The transaction is not a **duplicate** of another transaction
- The amount is larger than the minimum burn amount (in case of a burn transaction)
- And any other logic that we may want to implement...

If anything fails, the function returns **#Err(txError)** where **txError** is a **TransferError** indicating what has gone wrong.

If all checks are met, then the transfer is recorded and the function returns **#Ok(txIndex)** where **txIndex** is a **Nat** that represents the transaction index for the recorded transaction.

## Metadata and Extensions

1. `icrc1_metadata` is a `query` function that takes no arguments and returns an array of tuples `(Text, Value)`. The array may be empty. The tuples represent *metadata* about the token that may be used for client integrations with the token. The tuple represents a key-value store. The data does not have to be constant and may change in time. Notice that we use the `Value variant` in the tuple.
  - The `Text` is *namespaced* and consists of two parts separated by a colon: `namespace:key`. The first part `namespace` may not contain colons (`:`) and represents a 'domain' of metadata. The second part `key` is the actual key which is a name for the `Value`.
  - The `icrc1` namespace is reserved for keys from the ICRC1 standard itself, like `icrc1:name`, `icrc1:symbol`, `icrc1:decimals` and `icrc1:fee`. Other keys could be added as part of the `icrc1` metadata domain, for example `icrc1:logo`.
  - Another domain of metadata could be `stats` for providing statistics about the distribution of tokens. Some keys could be `stats:total_accounts`, `stats:average_balance` and `stats:total_tx`.
2. `icrc1_supported_standards` is a `query` function that returns an array of records `{ name : Text; url : Text }`. Each record contains info about *another* standard that may be implemented by this ICRC1 token. The ICRC1 token standard is intentionally designed to be very simple with the expectation that it will be extended by other standards. This *modularity of standards* allows for flexibility. (Not every token needs all capabilities of advanced token ledgers)

## Transaction deduplication

Usually, when a client makes a transfer, the token canister responds with a `Result<Nat, TransferError>`. This way, the client knows whether the transfer was successful or not.

If the client fails to receive the transfer response (for some reason, like a network outage) from the canister, it has no way of knowing whether the transaction was successful or not. The client could implement some logic to check the balance of the account, but that's not a perfect solution because it would still not know why a transfer may have been rejected.

To offer a solution for this scenario (a client misses the response after making a transfer and doesn't know whether the transaction was successful), ICRC1 specifies *transaction deduplication* functionality. An *identical transaction* submitted more than once within a certain *time window* will not be accepted a second time.

The time window could be a period of say 24 hours. This way, a frequent user (like an exchange or trader perhaps) could label their transactions with a `created_at_time` and a (possibly

unique) `memo`. The client could, in the case of a missed response, send the same transaction again within the allowed time window and learn about the status of the transaction.

If the initial transaction was successful, a correct implementation of ICRC1 would not record the transfer again and notify the client about the existing transaction by returning an error

`#Duplicate : { duplicate_of : Nat }` with an index of the existing transaction (a `Nat` value).

## IC time and Client time

An actor (the *host*) who receives a transaction can get the current *IC time* through the [Time Base Module](#).

A client could be a canister and in that case it would also get its time from the IC. But a client could also be a browser. In both cases, the *host time* and *client time* are not perfectly synced.

The *client* may specify a `created_at_time` which is different from the IC time as perceived by an actor who receives a transaction at some point in time.

The token canister may even receive a transaction that appears to have been made in the future! The reason for this is that *client* and *IC* are not in perfect time sync.

## Deduplication algorithm

If a transaction contains both a valid `memo` and a `created_at_time`, then a correct implementation of ICRC1 should make two checks before accepting the transaction:

- The transaction should fall within a specific *time window*
- The transaction is not a *duplicate* of another transaction within that time window

This time window is measured relative to IC time. Lets declare the IC time and relevant `Duration`s.

```
import Time "mo:base/Time";

let now = Time.now();

let window : Duration = 24 * 60 * 60 * 10**9;

let drift : Duration = 60 * 10**9;
```

The time window spans from `now - (window + drift)` to `now + drift`.

- If the `created_at_time` is smaller than `now - (window + drift)`, the response should an error `#TooOld`.
- If the `created_at_time` is larger than `now + drift`, the response should be an error `#CreatedInFuture: { ledger_time : Timestamp }` where `ledger_time` is equal to `now`.
- If the transaction falls within the time window, then there must not exist a duplicate transaction in the ledger within the same window. If its does exist, an error `#Duplicate : { duplicate_of : Nat }` should be returned. A duplicate transaction is one with all parameters equal to the current transaction that is awaiting approval, including the exact `memo` and `created_at_time`.
- Otherwise `created_at_time` falls within the time window and has no duplicates and thus should be accepted by returning `#Ok(TxIndex)` where `TxIndex` is the index at which the transaction is recorded.

# Tokenized Comments Example

This chapter demonstrates how one can reward user interaction with virtual tokens. It's a simple comment section where users (after logging in) can leave a comment and like other comments.

**The purpose is to demonstrate how to use common programming language features in Motoko.**

We use a virtual token with a capped supply, but we intentionally don't allow transfers. This way, the token can be used to reward users, but it can't be traded on exchanges.

---

## WARNING

*This is NOT a digital crypto currency. Tokens can not be sent. The tokenomics model is not economically sound. The model is for code writing demonstrations only.*

---

---

## NOTE

*The token is not an [ICRC1](#) compliant token*

---

Rules are simple:

- You can only comment and like interact after logging in with [Internet Identity](#).
- You may only comment once per 10 minutes and like once per minute.
- You earn 10 tokens for a posted comment
- You earn 1 token for every like you receive.

We start with a total supply of 10 000 tokens. When all tokens are given out, no more tokens can be earned.

## Project setup

The source code is [available here](#). The project consists of two canisters:

- a backend canister written in Motoko
- a frontend UI built with Vite-SvelteKit-Tailwind.

The frontend is uploaded to a separate *frontend assets canister* that can serve the website to a browser. The frontend canister calls the backend canister's shared functions to interact with

the backend.

---

## NOTE

*The frontend, the authentication with [Internet Identity](#) and the interaction with the backend from within Javascript code are outside the scope of this book.*

---

## Backend canister source code

The backend canister source code is setup with the following files:

- `main.mo` contains the main actor, [datastores](#), [shared functions](#) and [system upgrade functions](#)
- `Constants.mo` contains the constants used in the project
- `Types.mo` contains the types used in the project
- `Utils.mo` contains utility functions used in the project
- `Comments.mo` contains the the implementations of the main shared functions

## Datastores

We use a `state` object of type `State` to hold the datastores:

```
public type State = object {
 users : Users;
 commentStore : CommentStore;
 var commentHistory : CommentHistory;
 var treasury : Treasury;
};
```

The datastore types are:

```
public type Users = HashMap.HashMap<Principal, User>;
public type CommentStore = HashMap.HashMap<CommentHash, Comment>;
public type CommentHistory = List.List<CommentHash>;
public type Treasury = Nat;
```

The hashmaps `Users` and `CommentStore` are [immutable variables](#) in our `state` object. This means they hold a hashmap object with callable methods, but they cannot be replaced.

`CommentHistory` and `Treasury` on the other hand are [mutable variables](#). This means they hold a mutable variable that can be replaced with a new value. This happens when we deduct from the treasury or when we add a new comment to the comment history.

## Shared functions

The `main.mo` contains the [public API](#) of the actor. Most of the logic implementation for the shared functions is factored out to `Comments.mo` for clear code organization.

We only perform checks for the [identity of the caller](#) and pass the datastores as arguments to the functions in `Comments.mo`.

## State mutations

The `state` object and its stores are only updated by functions in `Comments.mo`. The updates always occur within an [atomically executed](#) functions. The functions that update the state are `register`, `postComment` and `likeComment`.

Note that `register` always returns `QueryUser`, but `postComment` and `likeComment` return `PostResult` and `LikeResult`, which may return an error, see [Result](#).

`postResult` performs all the checks before any state is updated. If any check fails, the function returns an error and the state is not updated. If all checks pass, the function updates the state and returns the result. The response tells us whether the state was successfully updated or not.

`likeComment` returns `async* LikeResult`. The reason for this is that the function may `throw` an `Error` if any of its checks fail. (Errors can only be thrown in an asynchronous context).

Errors are only thrown in `likeComment` if something unexpected happens, like when a like is submitted to a comment which doesn't exist. Since the frontend is programmed to be able to do that, a call like that should never happen.

Recall that state updates up until an error are persisted in `async` and `async*` functions. In the case of `likeComment` the `users` hashmap may be updated before an `Error` but balances are only updated if all checks are met.

## Upgrading

The `state` object initialized in `main.mo` from [stable var arrays](#) which are initially empty. The `state` object is used in working memory and filled with data when posts are made. The `state` object is not persisted during [upgrades](#), so when the canister is upgraded, the data is lost.

Therefore, we use the `preupgrade` system function to copy all the data to the `stable` array variables before an upgrade. The datastores then are initialized again from the `stable` variables.

In the `postupgrade` system function, we empty out the `stable` arrays to save [memory](#).

---

## NOTE

*preupgrade* and *postupgrade* system functions may trap during execution and data may be lost in the process. Work is underway to improve canister upgrades by working with [stable storage](#).

---

## Constants

All constants used in the project are defined in `Constants.mo`. This way, we can easily change the constants in one place and the changes are reflected throughout the project.

## DEMO

The comments canister is [live on mainnet](#).

# Tables

- [Table of asynchronous functions in Motoko](#)

- [Motoko Items Comparison Table](#)