The Move Book

For simple values, Move has a number of built-in primitive types. They're the foundation for all other types. The primitive types are:

Before we get to the primitive types, let's first take a look at how to declare and assign variables in Move.

Variables are declared using the let keyword. They are immutable by default, but can be made mutable by adding the mut keyword:

Where:

A mutable variable can be reassigned using the = operator.

Variables can also be shadowed by re-declaring them.

The bool type represents a boolean value - yes or no, true or false. It has two possible values: true and false , which are keywords in Move. For booleans, the compiler can always infer the type from the value, so there is no need to explicitly specify it.

Booleans are often used to store flags and to control the flow of the program. Please refer to the Control Flow section for more information.

Move supports unsigned integers of various sizes, from 8-bit to 256-bit. The integer types are:

While boolean literals like true and false are clearly booleans, an integer literal like 42 could be any of the integer types. In most of the cases, the compiler will infer the type from the value, usually defaulting to u64 . However, sometimes the compiler is unable to infer the type and will require an explicit type annotation. It can either be provided during assignment or by using a type suffix.

Move supports the standard arithmetic operations for integers: addition, subtraction, multiplication, division, and modulus (remainder). The syntax for these operations is:

For more operations, including bitwise operations, please refer to the Move Reference .

The types of the operands must match , or the compiler will raise an error. The result of the operation will be of the same type as the operands. To perform operations on different types, the operands need to be cast to the same type.

Move supports explicit casting between integer types. The syntax is as follows:

Note that parentheses around the expression may be required to prevent ambiguity:

A more complex example, preventing overflow:

Move does not support overflow / underflow; an operation that results in a value outside the range of the type will raise a runtime error. This is a safety feature to prevent unexpected behavior.

Variables and assignment

Variables are declared using the let keyword. They are immutable by default, but can be made mutable by adding the mut keyword:

```
bash let <variable_name>[: <type>] = <expression>; let mut <variable_name>[: <type>] =
<expression>;
```

Where:

```
bash let x: bool = true; let mut y: u8 = 42;
```

A mutable variable can be reassigned using the = operator.

```
bash y = 43;
```

Variables can also be shadowed by re-declaring them.

```
bash let x: u8 = 42; let x: u16 = 42;
```

The bool type represents a boolean value - yes or no, true or false. It has two possible values: true and false , which are keywords in Move. For booleans, the compiler can always infer the type from the value, so there is no need to explicitly specify it.

```
bash let x = true; let y = false;
```

Booleans are often used to store flags and to control the flow of the program. Please refer to the Control Flow section for more information.

Move supports unsigned integers of various sizes, from 8-bit to 256-bit. The integer types are:

```
bash let x: u8 = 42; let y: u16 = 42; // ... let z: u256 = 42;
```

While boolean literals like true and false are clearly booleans, an integer literal like 42 could be any of the integer types. In most of the cases, the compiler will infer the type from the value, usually defaulting to u64 . However, sometimes the compiler is unable to infer the type and will require an explicit type annotation. It can either be provided during assignment or by using a type suffix.

```
bash // Both are equivalent let x: u8 = 42; let x = 42u8;
```

Move supports the standard arithmetic operations for integers: addition, subtraction, multiplication, division, and modulus (remainder). The syntax for these operations is:

For more operations, including bitwise operations, please refer to the Move Reference .

The types of the operands must match , or the compiler will raise an error. The result of the operation will be of the same type as the operands. To perform operations on different types, the operands need to be cast to the same type.

Move supports explicit casting between integer types. The syntax is as follows:

```
bash <expression> as <type>
```

Note that parentheses around the expression may be required to prevent ambiguity:

```
bash let x: u8 = 42; let y: u16 = x as u16; let z = 2 * (x as u16); // ambiguous, requires
parentheses
```

A more complex example, preventing overflow:

```
bash let x: u8 = 255; let y: u8 = 255; let z: u16 = (x as u16) + ((y as u16) * 2);
```

Move does not support overflow / underflow; an operation that results in a value outside the range of the type will raise a runtime error. This is a safety feature to prevent unexpected behavior.

```bash let x = 255u8; let y = 1u8;

// This will raise an error let z = x + y; ```

## Booleans

The bool type represents a boolean value - yes or no, true or false. It has two possible values: true and false , which are keywords in Move. For booleans, the compiler can always infer the type from the value, so there is no need to explicitly specify it.

```
bash let x = true; let y = false;
```

Booleans are often used to store flags and to control the flow of the program. Please refer to the Control Flow section for more information.

Move supports unsigned integers of various sizes, from 8-bit to 256-bit. The integer types are:

```
bash let x: u8 = 42; let y: u16 = 42; // ... let z: u256 = 42;
```

While boolean literals like true and false are clearly booleans, an integer literal like 42 could be any of the integer types. In most of the cases, the compiler will infer the type from the value, usually defaulting to u64 . However, sometimes the compiler is unable to infer the type and will require an explicit type annotation. It can either be provided during assignment or by using a type suffix.

```
bash // Both are equivalent let x: u8 = 42; let x = 42u8;
```

Move supports the standard arithmetic operations for integers: addition, subtraction, multiplication, division, and modulus (remainder). The syntax for these operations is:

For more operations, including bitwise operations, please refer to the Move Reference .

The types of the operands must match , or the compiler will raise an error. The result of the operation will be of the same type as the
```

operands. To perform operations on different types, the operands need to be cast to the same type.

Move supports explicit casting between integer types. The syntax is as follows:

```
bash <expression> as <type>
```

Note that parentheses around the expression may be required to prevent ambiguity:

```
bash let x: u8 = 42; let y: u16 = x as u16; let z = 2 * (x as u16); // ambiguous, requires
parentheses
```

A more complex example, preventing overflow:

```
bash let x: u8 = 255; let y: u8 = 255; let z: u16 = (x as u16) + ((y as u16) * 2);
```

Move does not support overflow / underflow; an operation that results in a value outside the range of the type will raise a runtime error. This is a safety feature to prevent unexpected behavior.

```bash let x = 255u8; let y = 1u8;

// This will raise an error let z = x + y; ```

## Integer Types

Move supports unsigned integers of various sizes, from 8-bit to 256-bit. The integer types are:

```
bash let x: u8 = 42; let y: u16 = 42; // ... let z: u256 = 42;
```

While boolean literals like true and false are clearly booleans, an integer literal like 42 could be any of the integer types. In most of the cases, the compiler will infer the type from the value, usually defaulting to u64 . However, sometimes the compiler is unable to infer the type and will require an explicit type annotation. It can either be provided during assignment or by using a type suffix.

```
bash // Both are equivalent let x: u8 = 42; let x = 42u8;
```

Move supports the standard arithmetic operations for integers: addition, subtraction, multiplication, division, and modulus (remainder). The syntax for these operations is:

For more operations, including bitwise operations, please refer to the Move Reference .

The types of the operands must match , or the compiler will raise an error. The result of the operation will be of the same type as the operands. To perform operations on different types, the operands need to be cast to the same type.

Move supports explicit casting between integer types. The syntax is as follows:

```
bash <expression> as <type>
```

Note that parentheses around the expression may be required to prevent ambiguity:

```
bash let x: u8 = 42; let y: u16 = x as u16; let z = 2 * (x as u16); // ambiguous, requires
parentheses
```

A more complex example, preventing overflow:

```
bash let x: u8 = 255; let y: u8 = 255; let z: u16 = (x as u16) + ((y as u16) * 2);
~~~

Move does not support overflow / underflow; an operation that results in a value outside the range of the type will raise a runtime error. This is a safety feature to prevent unexpected behavior.

```bash let x = 255u8; let y = 1u8;

// This will raise an error let z = x + y; ```

## Further reading
~~~