SOLIDITY

Boolean

bool: The possible values are constants true and false.

Operators:
! (logical negation)
&& (logical conjunction, “and”)
|| (logical disjunction, “or”)
== (equality)
!= (inequality)

The operators || and && apply the common short-circuiting rules. This means that in the expression f(x) || g(y), if f(x) evaluates to true, g(y) will not be evaluated even if it may have side-effects.

Exponentiation

Exponentiation is only available for unsigned types. Please take care that the types you are using are large enough to hold the result and prepare for potential wrapping behaviour.

Division

Since the type of the result of an operation is always the type of one of the operands, division on integers always results in an integer. In Solidity, division rounds towards zero. This mean that int256(-5) / int256(2) == int256(-2). Note that in contrast, division on literals results in fractional values of arbitrary precision.

Addition, Subtraction Multiplication

Addition, subtraction and multiplication have the usual semantics. They wrap in two’s complement representation meaning that for example uint256(0) - uint256(1) == 2**256 - 1. You have to take these overflows into account when designing safe smart contracts.

The expression -x is equivalent to (T(0) - x) where T is the type of x. This means that -x will not be negative if the type of x is an unsigned integer type. Also, -x can be positive if x is negative.

There is another caveat also resulting from two’s complement

representation:
int x = -2**255;
assert(-x == x);

Integers

int /Uint : Signed and unsigned integers of various sizes. Keywords uint8 to uint256 in steps of 8 (unsigned of 8 up to 256 bits) and int8 to int256. uint and int are aliases for uint256 and int256, respectively.

Operators:
Comparisons: <=, <, ==, !=, >=, > (evaluate to bool)
Bit operators: &, |, ^ (bitwise exclusive or), ~ (bitwise negation)
Shift operators: << (left shift), >> (right shift)
Arithmetic operators: +, -, unary -, *, /, % (modulo), ** (exponentiation)

Modulo

The modulo operation a % n yields the remainder r after the division of the operand a by the operand n, where q = int(a / n) and r = a - (n * q). This means that modulo results in the same sign as its left operand (or zero) and a % n == -(-a % n) holds for negative a:

- int256(5) % int256(2) == int256(1)
- int256(5) % int256(-2) == int256(1)
- int256(-5) % int256(2) == int256(-1)
- int256(-5) % int256(-2) == int256(-1)

Fixed Size Byte Arrays

The value types bytes1, bytes2, bytes3, …, bytes32 hold a sequence of bytes from one to up to 32. byte is an alias for bytes1.
Operators:
Comparisons: <=, <, ==, !=, >=, > (evaluate to bool)
Bit operators: &, |, ^ (bitwise exclusive or), ~ (bitwise negation)
Shift operators: << (left shift), >> (right shift)
Index access: If x is of type bytes I, then x[k] for 0 <= k < I returns the k th byte (read-only).

The shifting operator works with any integer type as right operand (but returns the type of the left operand), which denotes the number of bits to shift by. Shifting by a negative amount causes a runtime exception

Members:
.length yields the fixed length of the byte array (read-only).

Dynamically Sized Byte Array

bytes:
Dynamically-sized byte array, see Arrays. Not a value-type!

string:
Dynamically-sized UTF-8-encoded string, see Arrays.
Not a value-type!

Shifts

The result of a shift operation has the type of the left operand, truncating the result to match the type.

For positive and negative x values, x << y is equivalent to x * 2**y.
For positive x values, x >> y is equivalent to x / 2**y.
For negative x values, x >> y is equivalent to (x + 1) / 2**y - 1 (which is the same as dividing x by 2**y while rounding down towards negative infinity).
In all cases, shifting by a negative y throws a runtime exception.

Address

The address type comes in two flavours, which are largely identical:
address: Holds a 20 byte value (size of an Ethereum address).
address payable: Same as address, but with the additional members transfer and send.
The idea behind this distinction is that address payable is an address you can send Ether to, while a plain address cannot be sent Ether.
Type conversions:
Implicit conversions from address payable to address are allowed, whereas conversions from address to address payable are not possible (the only way to perform such a conversion is by using an intermediate conversion to uint160).
Address literals can be implicitly converted to address payable.

Explicit conversions to and from address are allowed for integers, integer literals, bytes20 and contract types with the following caveat:
Conversions of the form address payable(x) are not allowed.
Instead the result of a conversion of the form address(x) has the type address payable, if x is of integer or fixed bytes type, a literal or a contract with a payable fallback function. If x is a contract without payable fallback function, then address(x) will be of type address. In external function signatures address is used for both the address and the address payable type.

Operators:
<=, <, ==, !=, >= and >

Members Of Address

For a quick reference of all members of address, see Members of Address Types.
balance and transfer
it is possible to query the balance of an address using the property balance and to send Ether (in units of wei) to a payable address using the transfer function:

```
address payable x = address(0x123);
address myAddress = address(this);
if (x.balance < 10 && myAddress.balance >= 10) x.transfer(10);
```

The transfer function fails if the balance of the current contract is not large enough or if the Ether transfer is rejected by the receiving account. The transfer function reverts on failure.