Both the type and sole value of the empty tuple.

Types start with an uppercase character, everything else with a lowercase letter.

Type variables stay maximally general (polymorphic), being restricted only by the ways the type is used. Generics force you to anticipate the maximum generality yourself when declaring the generic type's bounds.

... an interface. It creates a contract that defines the behaviors of types of that class.

... its name consists of only special characters and is infix by default.

It indicates a class constraint. The left hand side indicates class membership/s (comma separated) of the type variables used on the right hand side.

The type class for types that can be tested for equality.

`IO` and functions are not members.

The typeclass `Ord` defines a function `compare` which returns an `Ordering`, meaning either `GT`, `EQ`, or `LT`.

Its members can be presented as strings using the `show` function.

The `Read` typeclass defines `read` which does just that.

However, in order to know what type is desired, the result of `read` must be used in an expression or given a type annotation.

Haskell's term for a manifest type. It's not an annotation on a type in the sense of Scala's `@specialized`.

Haskell type annotations use the `::` followed by the type.

They can be enumerated.

They have an upper and a lower bound: `maxBound` and `minBound`.

```
ghci> :t 0
0 :: (Num t) => t
ghci> :t maxBound
maxBound :: (Bounded a) => a
```

```
ghci> :t fromIntegral
fromIntegral :: (Integral a, Num b) => a -> b
```

With a `where` clause following the guards, containing variable assignments, one per line, aligned.

Named functions can be created, using the normal syntax.

In a `where` clause. Sometimes this gets nested into one helper helping another, and so on.

- `let` is an expression, while `where` is just a syntactic construct.
- `where` doesn't have such a tight scope, so its bindings are active across multiple guards.

By separating the assignments with semicolons.

`let` expressions, being far more powerful than Scala's midstream definitions.

... case expressions.

```
case expr of pat1 -> res1
                ...
             patN -> resN
```