6 - type systems

① monomorphic type system:
- each value has a single specific type

ex: C and most languages

② polymorphic type system:
- you define something and it could be used in other data types
  - ↳ ad-hoc polymorphism = overloading, functions that can be applied to different data types and behave differently
  - ↳ inclusion polymorphism = based on subtyping relation function applies to a type and all subtypes of the type (ex C types = char ⊆ short ⊆ int ⊆ long)
  - ↳ parametric polymorphism = functions that are general and can operate identically on different types (actual polymorphism)

haskell classes: you define a class that has functions in it, then you specify data types as instance of it to belong to that class / to use that function

```
class X a where
f:: a -> a
f a = a
```

```
data T = AA | BB deriving Sho
```

```
instance X T
a:: T
a = AA
f a -> AA
```

you are saying that these data types can use that class function (basically overloading)

you can also rewrite the function but the input and output patters must be same (ex f:: int -> int)

polymorphism = one simple function definition, and one implementation, every data type uses the same one (operates on multiple types uniformly)

overloading = we just have function name, implementation of the function could be totally different from each other (different functions for distinct types)
- ↳ binding is not only according to name but according to name and type
- context dependent overloading = based on name, parameter type and return type
- context independent overloading = based on function name and parameter type (no return type)
  - ↳ most languages use this

coercion (force) = making implicit type conversion for ease of programming
- ↳ ex: double x = k + 4.2 → double x = (double) k + 4.2
- ↳ most newer languages quit coercion completely (too complex): strict type checking

type inference = if type system does not force user to declare all types (like in C), language processor infers types, finds the most general type that fits.