polymorphism comes from Greek meaning ‘many forms’

In programming:

Def: A function or operator is polymorphic if it has at least two possible types.

Read MPL Chap 8
Polymorphism

Different types of polymorphisms

- **Ad hoc** (overloading)
  Asteroid & Python: operator overloading, e.g. `1+2`, "Hello" + "There"

- **Parametric**
  Types with type variables
  Rust: `vec<T>`

- **Duck Typing**
  "the duck test"*
  Asteroid & Python

- **Subtyping**
  Python: Class inheritance

*If it looks like a duck, swims like a duck, and quacks like a duck, then it probably *is* a duck. --Wikipedia
Def: An **overloaded function name or operator** is one that has at least two definitions, all of different types.

**Example:** In Asteroid the ‘+’ operator is overloaded. It can function as a string concatenation operator or as an addition operator depending on the type context – polymorphism!

```
Asteroid Version 1.1.3
(c) University of Rhode Island
Type "asteroid -h" for help
Press CTRL-D to exit
|ast> "abc"+"def" == "abcdef"
  true
|ast> 3+5 == 8
  true
```
Def: A function/structure exhibits **parametric polymorphism** if it has a type that contains one or more **type variables**.

**Example:** Rust

```rust
struct Data<T> {
    value:T,  // T is a type variable
}

fn main() {
    // instantiating Data with i32 data
    let t:Data<i32> = Data{value:350};
    println!("value is :{} ", t.value);

    // instantiating Data with String data
    let t2:Data<String> = Data{value:"Tom".to_string()};
    println!("value is :{} ", t2.value);
}
```

Source: https://www.tutorialspoint.com/rust/rust_generic_types.htm
Def: A function or operator exhibits **subtype polymorphism** if one or more of its **types** have subtypes.
Example: Java

class Cup { ... };
class CoffeeCup extends Cup { ... };
class TeaCup extends Cup { ... };

TeaCup t = new TeaCup();
Cup c = t; ← type coercion: TeaCup → Cup

void fill (Cup c) {...}

TeaCup t = new TeaCup();
CoffeeCup k = new CoffeeCup();

fill(t);
fill(k); } subtype polymorphism

safe!
Duck Typing

- Duck typing in computer programming is an application of the duck test—"If it walks like a duck and it quacks like a duck, then it must be a duck"—to determine if an object can be used for a particular purpose.
  - With normal typing, suitability is determined by an object's type.
  - In duck typing, an object's suitability is determined by the presence of certain methods and properties, rather than the type of the object itself. No common base type!

https://en.wikipedia.org/wiki/Duck_typing
Duck Typing

- Example: a polymorphic list with Duck Typing.

```plaintext
lutz$ asteroid ducktyping.ast
a duck can fly
a plane can fly
lutz$

--- A demonstration of duck typing
load system io.

-- define some types with the property 'fly'
structure Duck with
    function fly with none do
        io @println "a duck can fly".
    end
end

structure Plane with
    function fly with none do
        io @println "a plane can fly".
    end
end

-- create a polymorphic list
let l = [Duck(),Plane()].

-- use the interface that is common to all the objects
for e in l do
    e @fly ().
end
```

Polymorphic list: list with many different types
Duck Typing

- Duck typing is not possible in statically typed languages like Rust, C++, and Java.
- Instead, in these languages, one has to rely on subtype polymorphism in order to construct a polymorphic list.
import java.util.*;

abstract class ThingsThatFly { // base class of the hierarchy
    abstract void fly();
}

class Duck extends ThingsThatFly {
    void fly() {
        System.out.println("a duck can fly");
    }
}

class Plane extends ThingsThatFly {
    void fly() {
        System.out.println("a plane can fly");
    }
}

Can only declare lists of a single type!

class Main {
    public static void main(String args[]) {
        // create a list of ThingsThatFly
        ArrayList<ThingsThatFly> list = new ArrayList<ThingsThatFly>();
        list.add(new Duck());
        list.add(new Plane());
        // print the arraylist objects
        for (int i = 0; i < list.size(); i++) {
            list.get(i).fly();
        }
    }
}
MPL chap 8