Polymorphism

A closer look at types....

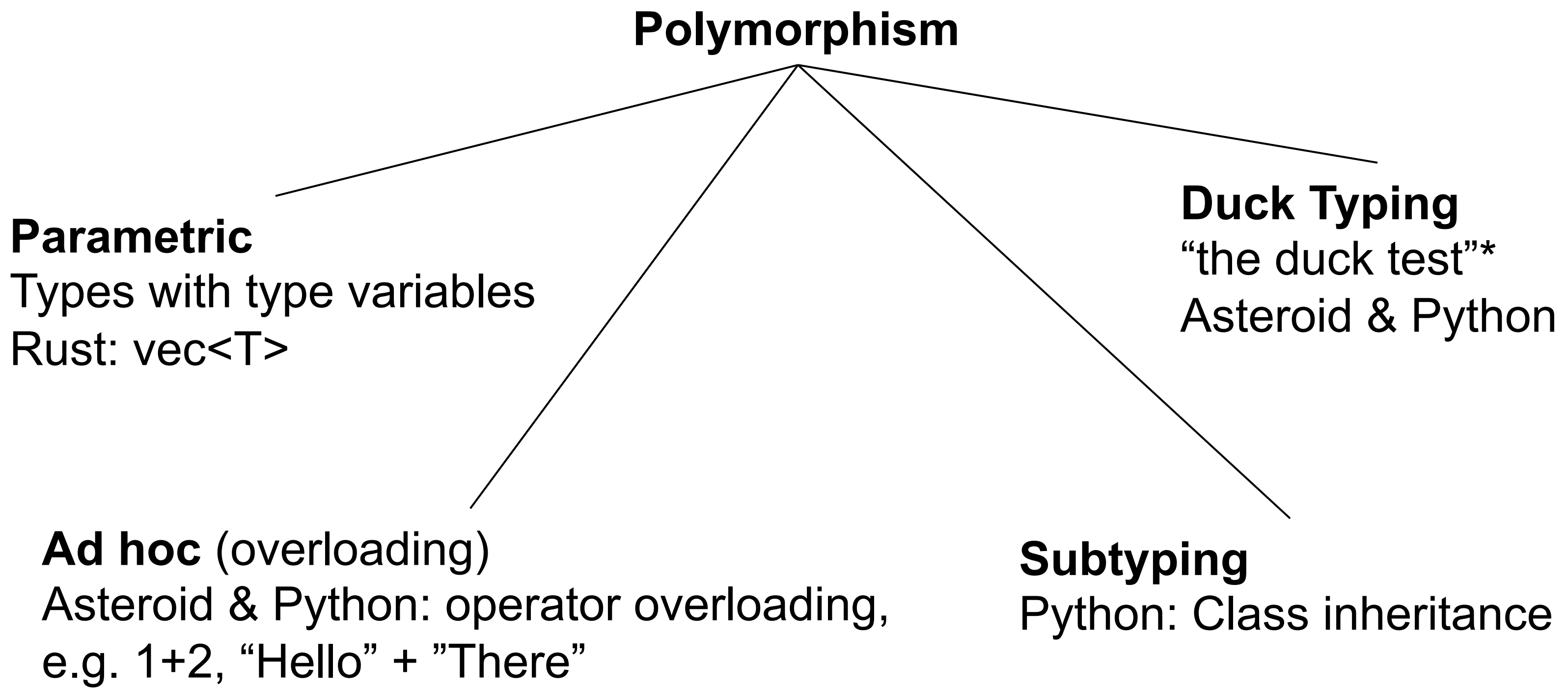
polymorphism ≡ comes from Greek meaning ‘many forms’

In programming:

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

Polymorphism

Different types of polymorphisms



Polymorphism

Parametric
Types with type variables
Rust: vec<T>

Duck Typing
“the duck test”*
Asteroid & Python

Ad hoc (overloading)
Asteroid & Python: operator overloading,
e.g. 1+2, “Hello” + ”There”

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

Polymorphism

- **Ad hoc polymorphism** or **overloading**: defines a function/operator name for an arbitrary set of individually specified types.
- **Parametric polymorphism**: when one or more types are not specified by name but by type variables that can represent any type.
- **Subtyping** or **subtype polymorphism**: when a name denotes instances of many different classes related by some common superclass.

https://en.wikipedia.org/wiki/Polymorphism_(computer_science)

Ad Hoc Polymorphism

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.

```
let s:%string = “abc” + “def”.


let i:%integer = 3 + 5.
```

Parametric Polymorphism

Def: A function/structure exhibits parametric polymorphism if it has a type that contains one or more type variables.

Example: Rust

Type Variable

```
struct Data<T> {
   value:T,
}
fn main() {
   //generic type of i32
   let t:Data<i32> = Data{value:350};
   println!("value is :{} ",t.value);
   //generic type of String
   let t2:Data<String> = Data{value:"Tom".to_string()};
   println!("value is :{} ",t2.value);
}
```

Source: https://www.tutorialspoint.com/rust/rust_generic_types.htm

Subtype Polymorphism

Def: A function or operator exhibits subtype polymorphism if one or more of its types have subtypes.

Subtype Polymorphism

Example: Java

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

```
TeaCup t = new TeaCup();
Cup c = t;
```

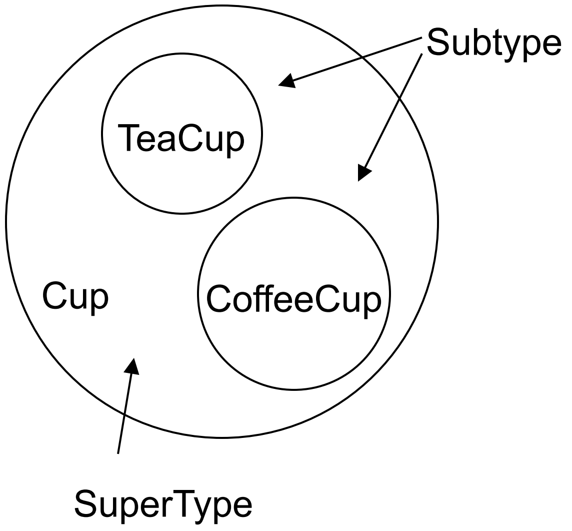
← $\underbrace{\text{type coercion: TeaCup} \rightarrow \text{Cup}}_{\text{safe!}}$

```
void fill (Cup c) {...}

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

fill(t);
fill(k);
```

} subtype polymorphism



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.**

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

Duck Typing

- Example: a polymorphic list with Duck Typing.
- Compare this to the subtype polymorphism example written in Rust…

```python
class Duck:
    def fly(self):
        print("Duck flying")

class Sparrow:
    def fly(self):
        print("Sparrow flying")

class Whale:
    def swim(self):
        print("Whale swimming")

for animal in Duck(), Sparrow(), Whale():
    animal.fly()
```

Polymorphic list

Duck Typing

- Duck typing can also be more flexible in that only the methods actually called at runtime need to be implemented.
- Most dynamically typed languages implement Duck Typing.