

Polymorphism

A closer look at types....

polymorphism \equiv comes from Greek meaning 'many forms'

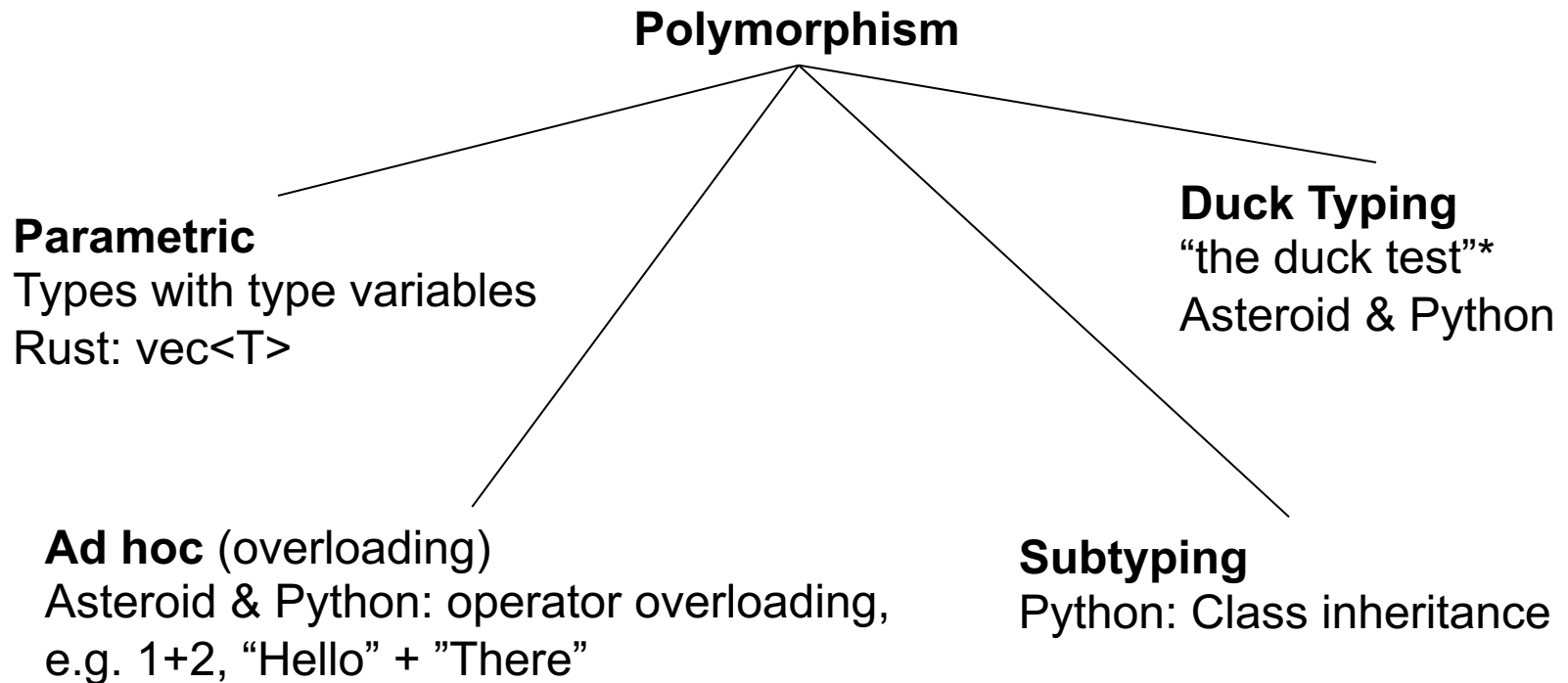
In programming:

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

Read MPL Chap 8

Polymorphism

Different types of polymorphisms



*If it looks like a duck, swims like a duck, and quacks like a duck, then it probably *is* a duck. --Wikipedia

Ad Hoc Polymorphism (overloading)

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
ast> █
```

Parametric Polymorphism

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

Example: Rust

```
1  struct Data<T> {  
2      value:T,    // T is a type variable  
3  }  
4  
5  fn main() {  
6      // instantiating Data with i32 data  
7      let t:Data<i32> = Data{value:350};  
8      println!("value is :{} ",t.value);  
9  
10     // instantiating Data with String data  
11     let t2:Data<String> = Data{value:"Tom".to_string()};  
12     println!("value is :{} ",t2.value);  
13 }
```

Subtype Polymorphism

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

Subtype Polymorphism

Example: Java

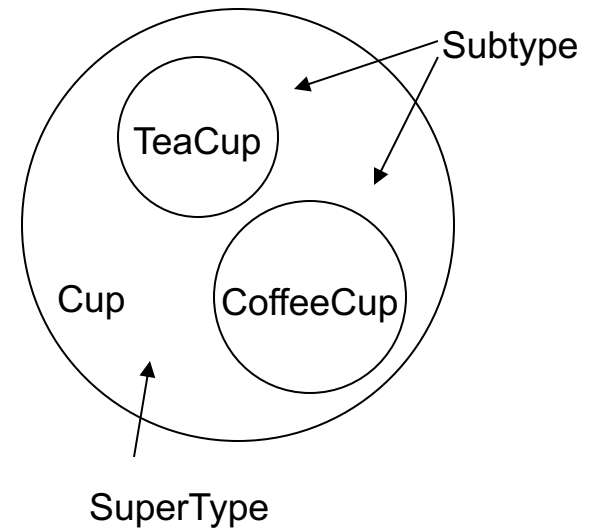
```
class Cup { ... };  
class CoffeeCup extends Cup { ... };  
class TeaCup extends Cup { ... };
```

```
TeaCup t = new TeaCup();  
Cup c = t; ← type coercion: TeaCup → Cup  
                        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. No common base type!**

Duck Typing

- Example: a polymorphic list with Duck Typing.

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

```
1  -- A demonstration of duck typing
2  load system io.
3
4  -- define some types with properties and the
5  -- shared property that the object of those
6  -- types can fly
7  structure Duck with
8  |   function fly with none do
9  |   |   io @println "a duck can fly".
10 |   end
11 |   function swim with none do
12 |   |   io @println "a duck can swim".
13 |   end
14 end
15
16 structure Plane with
17 |   function fly with none do
18 |   |   io @println "a plane can fly".
19 |   end
20 |   function taxi with none do
21 |   |   io @println "a plane can taxi".
22 |   end
23 end
24
25 -- create a polymorphic list
26 let l = [Duck(),Plane(),Duck(),Plane()].
27
28 -- use the interface that is common to all the objects
29 for e in l do
30 |   e @fly ().
31 end
```

Polymorphic list

