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.

Polymorphism

Different type of polymorphisms

Polymorphism

Ad hoc (overloading)
Rust/Haskell: 1+2, 1.0+2.0

Parametric
Rust: vec<T>
Haskell: [a]

Subtyping
Rust: via Traits
Haskell: via Type Classes

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 Java the ‘+’ operator is overloaded.

String s = “abc” + “def”;

+: String * String → String

int i = 3 + 5;

+: int * int → int

Polymorphism

<u>Example</u>: Java allows user defined polymorphism with overloaded function names.

```
bool f (char a, char b) {
        return a == b;
}
```

$f : char * char \rightarrow bool$

```
bool f (int a, int b) {
        return a == b;
}
```

$f : int * int \rightarrow bool$

Parametric Polymorphism

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

Example: Haskell

```
> f (x,y) = (x==y)
> :type f
f :: Eq a => (a, a) -> Bool
```

polytype
(poly ≡ many)

Example: C++ and Java
C++ and Java have templates and Rust has generics that support parametric polymorphism.

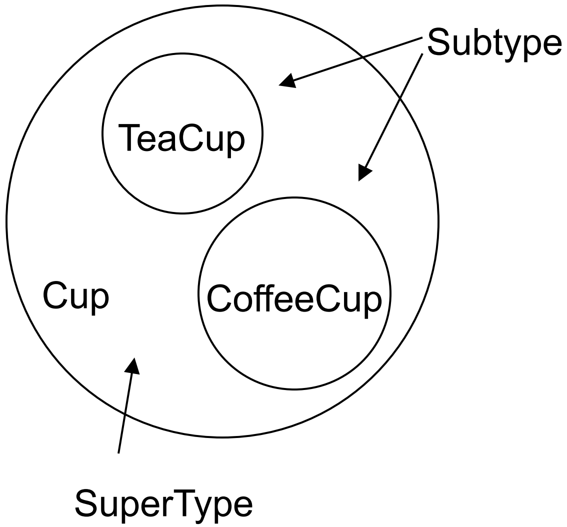
Subtype Polymorphism

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

Subtype Polymorphism

<u>Example</u>: Java

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




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

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

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

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

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

subtype polymorphism

Subtype Polymorphism

```
// Define a trait we want our objects to have
trait Waddles {
    fn waddles (&self);
}
// Define our objects and implement the traits
struct Duck { name : String }
impl Waddles for Duck {
    fn waddles(&self) { println!("{} the duck waddles on land", self.name);}
}
struct Penguin { name : String }
impl Waddles for Penguin {
    fn waddles(&self) { println!("{} the penguin waddles on ice", self.name);}
}
struct Woodchuck { name : String }
impl Waddles for Woodchuck {
    fn waddles(&self) { println!("{} the woodchuck waddles low to the ground", self.name);}
}
// polymorphic programming with traits
fn main() {
    let animals: [&Waddles;3] = [
        &Duck {name: "Polly".to_string()},
        &Penguin {name: "Schubert".to_string()},
        &Woodchuck {name: "Wally".to_string()}
    ];

    for i in 0..animals.len() {
        animals[i].waddles();
    }
}
```

Polymorphic List

Screenshot

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.