A New Approach to OOP

- No classes – structures with behavior instead
- No (class) inheritance –traits/interfaces instead
- Limited, if any, member protection to facilitate structural pattern matching on objects

Structures with Behavior

Asteroid

```
structure Rectangle with
  data xdim.
  data ydim.
  function area with () do -- member function
    return this@xdim * this@ydim.
  end
end
```

Go

```
type rect struct {
    width int
    height int
}

func (r *rect) area() int {
    return r.width * r.height
}
```

Rust

```
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}
```

Python

```
class Shape:
    def __init__(self):
        print("instantiating a shape ol

class Rectangle(Shape):
    def __init__(self,a,b):
        super().__init__()
        self.xdim = a
        self.ydim = b
    def area(self):
        return self.xdim*self.ydim
```

ln008/rect.py

- Python takes a hybrid approach
  - Class inheritance structure but no member protection
  - Notice that because of duck typing we don’t need dynamic dispatching

Rust

- **Problem**: with the loss of inheritance how is subtype polymorphism supported in statically typed languages like Rust?
- **Answer**: Traits (sometimes called interfaces) allow the developer to attach additional behavior to a class where that behavior can be shared among many classes effectively allowing polymorphic behavior with dynamic dispatching.

Rust

Abstract function

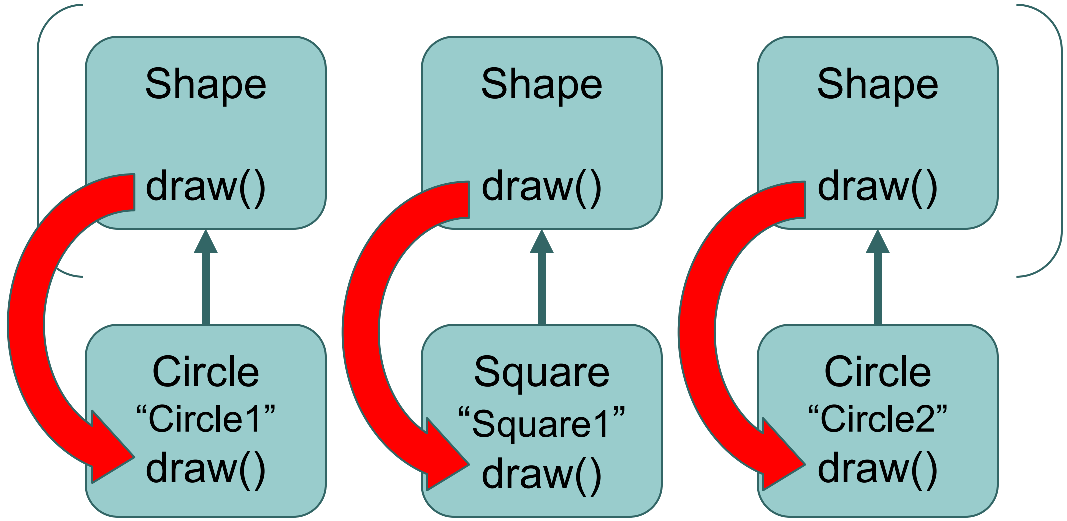
```rust
use std::vec::Vec;

trait Shape { fn draw(&self); }

struct Circle { name: String }
impl Circle { fn new(name: &str) -> Circle { Circle { name: name.to_string() } } }
impl Shape for Circle { fn draw(&self) {println!("Drawing a circle {}", self.name);} }

struct Square { name: String }
impl Square { fn new(name: &str) -> Square { Square { name: name.to_string() } } }
impl Shape for Square { fn draw(&self) { println!("Drawing a square {}", self.name); } }

fn main() {
    let mut v: Vec<Box<dyn Shape>> = Vec::new();
    v.push(Box::new(Circle::new("Circle1")));
    v.push(Box::new(Square::new("Square1")));
    v.push(Box::new(Circle::new("Circle2")));
    for shape in &v {
        shape.draw();
    }
}
```

ln008/subpoly.rs

Subtype Polymor

```
let mut v: Vec<Box<dyn Shape>> = Vec::new();
v.push(Box::new(Circle::new("Circle1")));
v.push(Box::new(Square::new("Square1")));
v.push(Box::new(Circle::new("Circle2")));
for shape in &v {
    shape.draw();
}
```

```
let mut v: Vec<Box<dyn Shape>> =
```



Here Shape is a trait not a superclass!

- Dynamic dispatch realizes when calling the draw function of the trait that an implementation of that trait function exists in the structure and calls it.

Object Composition vs Inheritance

- Many modern programming languages advocate for object composition rather than inheritance, e.g. Go, Rust, Asteroid
- In OOP inheritance as a subtype construction is often abused contributing to the issues mentioned earlier, consider
  - 'is-a' relation
  - Often abused for the implementation of a 'has-a' relation

```python
class Address:
    def __init__(self, street, city, state, zip):
        self.street = street
        self.city = city
        self.state = state
        self.zip = zip

class Person(Address):
    def __init__(self, name, age, street, city, state, zip):
        super().__init__(street, city, state, zip)
        self.name = name
        self.age = age

person = Person("John Doe", 30, "123 Main St", "Anytown", "CA", "12345")
```

Person is a subtype of Address!?!

In008/inheritance.py

Object Composition vs Inheritance

- Object composition solves this much cleaner and still enables pattern matching on objects

```
structure Address with
    data street.
    data city.
    data state.
    data zip.
end

structure Person with
    data name.
    data age.
    data address.
end

let address = Address("123 Main St", "Anytown", "CA", "12345").
let person = Person("John Doe", 30, address).

-- complete destructuring of the person object
--    => pattern matching on nested objects
let Person(name,age,Address(stree,city,state,zip)) = person.
```

True 'has-a' relation.

ln008/objcomp.ast