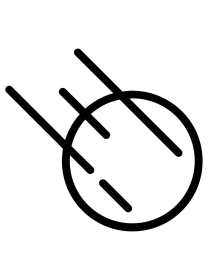
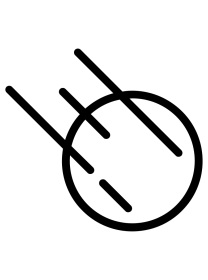


First-Class Patterns as Types

- We already have seen that patterns behave like data types, consider,
  - let x:%integer = v.
- Here the pattern %integer that matches all integer values limits what kind of values can be assigned to the variable x.
- That is precisely what type declarations do!

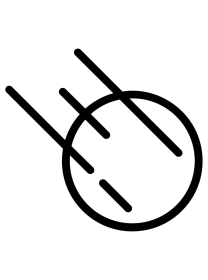


Subtypes

- First-class patterns can be used to define **subtypes of existing types**
- Consider for example,

```
let Pos_Int = pattern %[k if (k is %integer) and (k>0)]%.

let x:*Pos_Int = v.
```

- Here we can treat the pattern Pos_Int as a subtype of the integers, in effect we have
  - Pos_Int < integer

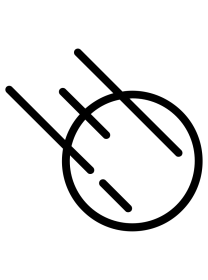


Supertypes

- We can use first class patterns to also define supertypes, consider

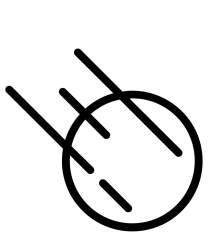
```
let Scalar = pattern %[x if (x is %integer) or (x is %real)]%.

let i:*Scalar = v.
```

- Here the second let statement is only successful if it fulfills the requirements of the pattern Scalar.
- In effect, Scalar acts like a supertype of real and integer
- or more precisely it acts like an **abstract base class** since you since you cannot instantiate a value of type Scalar.



Sub- and Supertypes

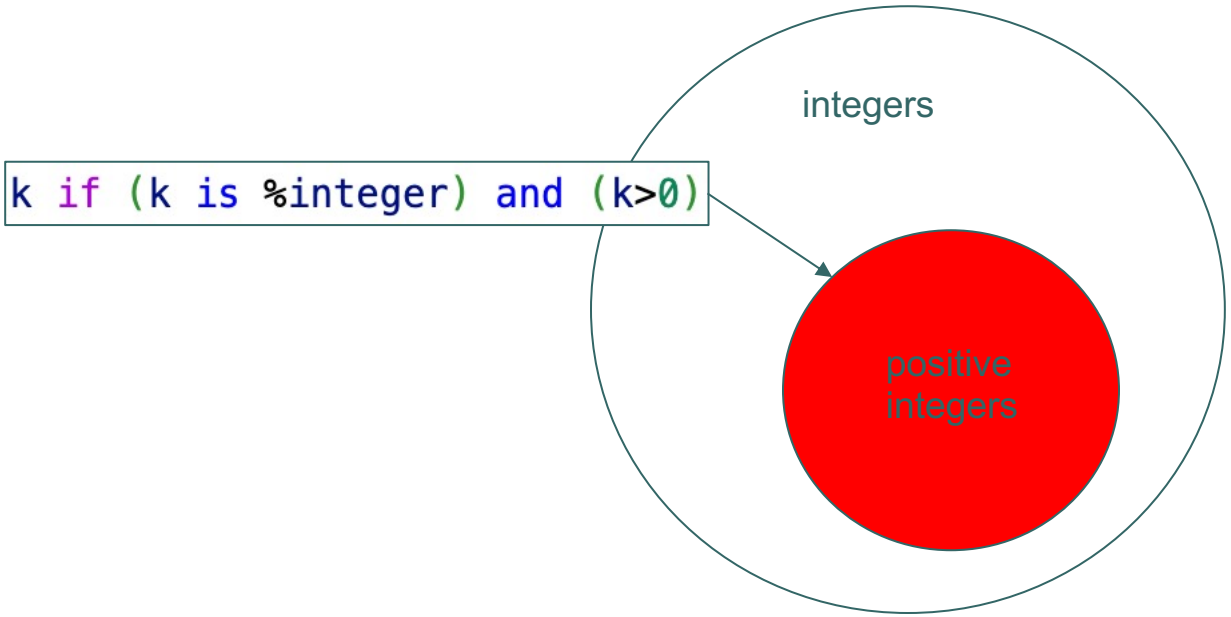
- We use first-class patterns to instantiate both subtypes and supertypes – how do they differ?

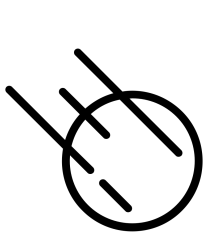


Sub- and Supertypes

- **Subtypes**: the pattern definition adds conditions that **contract** a given data type

```
let Pos_Int = pattern %[k if (k is %integer) and (k>0)]%.
```



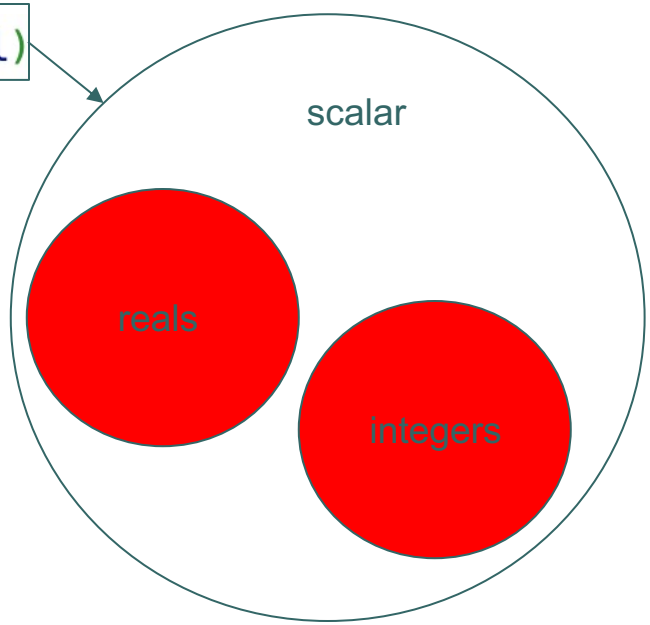


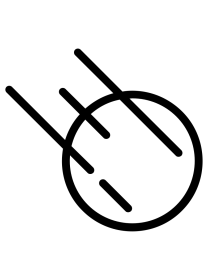
Sub- and Supertypes

- **Supertypes**: the pattern definition **expands** given data types so that the supertype pattern covers more objects than any given data type within the pattern definition.

```
let Scalar = pattern %[x if (x is %integer) or (x is %real)]%.
```

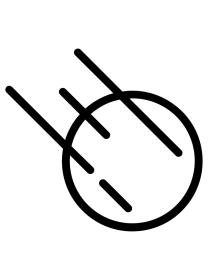
```
x if (x is %integer) or (x is %real)
```





Programming with Patterns as Data Types

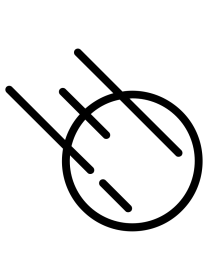
- We an impose a certain amount of type safety with patterns as data types
  - Specification of function domains
  - Type safety for objects using patterns as types in constructors
  - Subtype polymorphism



Function Domains

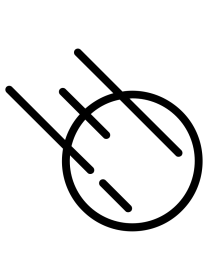
```
let Pos_Int = pattern %[(x:%integer) if x>0]%.

function fact
   with 0 do
      1
   with n:*Pos_Int do
      n*fact(n-1)
end

assert (fact 3 == 6).
```

ln016/fact.ast



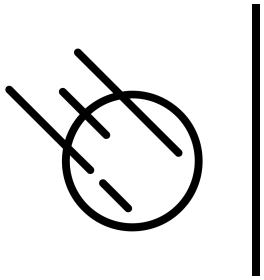
Objects

```
structure Address with
    data street.
    data city.
    data zip.
    function __init__ with (street:%string,city:%string,zip:%string) do
        let this@street = street.
        let this@city = city.
        let this@zip = zip.
    end
end

structure Person with
    data name.
    data profession.
    data address.
    function __init__ with (name:%string,profession:%string,address:%Address) do
        let this@name = name.
        let this@profession = profession.
        let this@address = address.
    end
end

let joe = Person("Joe","Carpenter",Address("532 Main Street","Newport","02840")).
```



Subtype Polymorphism

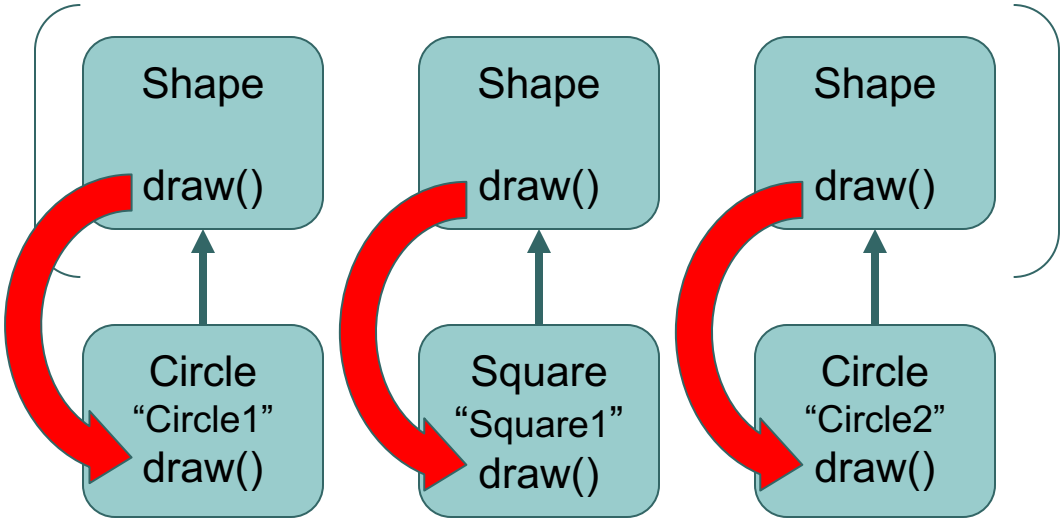
- In statically typed languages such as Java and Rust subtype polymorphism allows us to have type safe polymorphic containers
- Recall our Rust Shape container In008 slide pack slide 5



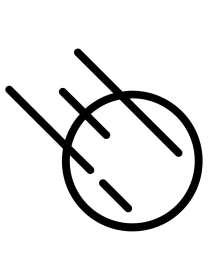
Subtype Polymor

```
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>> =
```

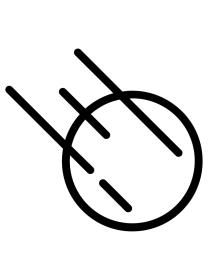


- Dynamic dispatch realizes when calling the draw function of the trait that an implementation of that trait function exists in the structure and calls it.



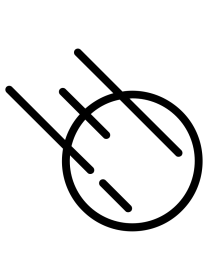
Subtype Polymorphism

- Dynamically typed languages like Python and Asteroid achieve polymorphic containers via Duck Typing.
- However, these containers are not as type safe as subtype polymorphic containers since any object that supports the required behavior will fit into the container.



Subtype Polymorphism

- In Asteroid we can recover a certain amount of type safety using first-class patterns
- We use first-class patterns as types that allow us to define subtype-supertype relation ships
  - ☞ subtype polymorphism

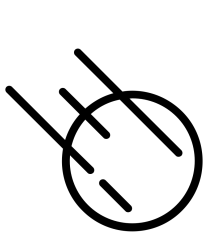


Subty

Note: if we were to try to add anything but circles and squares to the list the 'Shape_List' pattern would fail!

```
load system io.

structure Circle with
    data name.
    -- draw interface
    function draw with () do
        io @println ("Drawing a circle "+this@name).
    end
end

structure Square with
    data name.
    -- draw interface
    function draw with () do
        io @println ("Drawing a square "+this@name).
    end
end

let Shape = pattern %[x if (x is %Circle) or (x is %Square)]%
let Shape_List = pattern %[(x:%list)
      if x @reduce(lambda with (acc,e) do acc and (e is *Shape),true)]%

let v :*Shape_List = [].
let v :*Shape_List = v + [Circle("Circle1")].
let v :*Shape_List = v + [Square("Square1")].
let v :*Shape_List = v + [Circle("Circle2")].

for i in range (len v) do
    v@i @draw ().
end
```

ln014/subtypes.ast



Subtype Polymorphism

- Alternatively, we can construct the list in one go and then check for type safety.

```
load system io.

structure Circle with ⋯
end

structure Square with ⋯
end

let Shape = pattern %[x if (x is %Circle) or (x is %Square)]%
let Shape_List = pattern %[(x:%list)
    if x @reduce(lambda with (acc,e) do acc and (e is *Shape),true)]%

let v = [].
let v @append(Circle("Circle1")).
let v @append(Square("Square1")).
let v @append(Circle("Circle2")).

assert(v is *Shape_List).

for i in range (len v) do
   v@i @draw ().
end
```