Part 1

Classes

```
class Posn(x, y):
  extends Object
  method mdist(arg): this.x + this.y
  method addDist(arg): arg.mdist(0) + this.mdist(0)

class Posn3D(z):
  extends Posn
  method mdist(arg): this.z + super.mdist(arg)

new Posn3D(1, 2, 3).addDist(new Posn(3, 4))
```

Typechecking Programs with Classes

A well-formed program should never error with

- not a number

```
1 + new Posn(1, 2)
```

Typechecking Programs with Classes

A well-formed program should never error with

- not a number
- not an object

```
1.mdist(0)
```

Typechecking Programs with Classes

A well-formed program should never error with

- not a number
- not an object

```
1.x
```

Typechecking Programs with Classes

A well-formed program should never error with

- not a number
- not an object
- wrong field count

```
new Posn3D(1, 2)
```

Typechecking Programs with Classes

A well-formed program should never error with

- not a number
- not an object
- wrong field count
- not found
    - class, field, or method

```
new SquareCircle()
```

Typechecking Programs with Classes

A well-formed program should never error with

- not a number
- not an object
- wrong field count
- not found
  - class, field, or method

```
new Posn(1, 2).z
```

Typechecking Programs with Classes

A well-formed program should never error with

- not a number
- not an object
- wrong field count
- not found
  - class, field, or method

```
new Posn(1, 2).area()
```

Typechecking Programs with Classes

A well-formed program should never error with

- not a number
- not an object
- wrong field count
- not found
  - class, field, or method

```
class Circle():
  extends Object
  method area(arg): super.area(arg)
```

Typed Class Language

```
<Class>   ::=  class <Symbol>(<Field>, ...):
                 extends <Symbol>
                 <Method>
                 ...
<Field>   ::=  <Symbol> :: <Type>                                NEW
<Method>  ::=  method <Symbol>(arg :: <Type>) :: <Type>:         NEW
                 <Expr>
<Type>    ::=  Int                                               NEW
            |  <Symbol>                                          NEW
```

Part 2

Typechecking Programs with Classes

Is this program well-formed?

```
class Posn(x :: Int, y :: Int):
  extends Object
  method mdist(arg :: Int) :: Int:
    this.x.mdist(0) + this.y.mdist(0)

10
```

No — the `x` and `y` fields are not objects

Typechecking Programs with Classes

Is this program well-formed?

```
class Posn(x :: Int, y :: Int):
  extends Object
  method mdist(arg :: Int) :: Int:
    this.x + this.z

10
```

No — `Posn` has no `z` field

Typechecking Programs with Classes

Is this program well-formed?

```
class Posn(x :: Int, y :: Int):
  extends Object
  method mdist(arg :: Int) :: Int:
    this.x + this.get_y(0)

10
```

No — `Posn` has no `get_y` method

Typechecking Programs with Classes

Is this program well-formed?

```
class Posn(x :: Int, y :: Int):
  extends Object
  method mdist(arg :: Int) :: Posn:
    this.x + this.y

10
```

No — result type for `mdist` does not match body type

Typechecking Programs with Classes

Is this program well-formed?

```
class Posn(x :: Int, y :: Int):
  extends Object
  method mdist(arg :: Int) :: Int:
    this.x + this.y

10
```

Yes

Typechecking Programs with Classes

Is this program well-formed?

```
class Posn(x :: Int, y :: Int):
  extends Object
  method mdist(arg :: Int) :: Int:
    this.x + this.y

new Posn(12)
```

No — wrong number of fields in **`new`**

Typechecking Programs with Classes

Is this program well-formed?

```
class Posn(x :: Int, y :: Int):
  extends Object
  method mdist(arg :: Int) :: Int:
    this.x + this.y

new Posn(12, new Posn(1, 2))
```

No — wrong field type for first **new**

Typechecking Programs with Classes

Is this program well-formed?

```
class Posn(x :: Int, y :: Int):
  extends Object
  method mdist(arg :: Int) :: Int:
    this.x + this.y
  method clone(arg :: Int) :: Posn:
    new Posn(this.x, this.y)

new Posn(1, 2).clone(0)
```

Yes

Typechecking Programs with Classes

Is this program well-formed?

```
class Posn(x :: Int, y :: Int):
  extends Object
  method mdist(arg :: Int) :: Int:
    this.x + this.y
  method clone(arg :: Int) :: Posn:
    new Posn(this.x, this.y)

class Posn3D(z :: Int):
  extends Posn
  method mdist(arg :: Int) :: Int:
    this.z + super.mdist(arg)

new Posn3D(5, 7, 3)
```

Yes

Typechecking Programs with Classes

Is this program well-formed?

```
class Posn(x :: Int, y :: Int):
  extends Object
  method mdist(arg :: Int) :: Int:
    this.x + this.y
  method clone(arg :: Int) :: Posn:
    new Posn(this.x, this.y)

class Posn3D(z :: Int):
  extends Posn
  method mdist(arg :: Int) :: Posn:
    new Posn(10, 10)

new Posn3D(5, 7, 3)
```

No — override of `mdist` changes result type

Typechecking Programs with Classes

Is this program well-formed?

```
class Posn(x :: Int, y :: Int):
  extends Object
  method mdist(arg :: Int) :: Int:
    this.x + this.y
  method clone(arg :: Int) :: Posn:
    new Posn(this.x, this.y)

class Posn3D(z :: Int):
  extends Posn
  method mdist(arg :: Int) :: Int:
    this.z + super.mdist(arg)
  method clone(arg :: Int) :: Int:
    10

new Posn3D(5, 7, 3)
```

No — override of `clone` changes result type

Typechecking Programs with Classes

Is this program well-formed?

```
class Posn(x :: Int, y :: Int):
  extends Object
  method mdist(arg :: Int) :: Int:
    this.x + this.y
  method clone(arg :: Int) :: Posn:
    new Posn(this.x, this.y)

class Posn3D(z :: Int):
  extends Posn
  method mdist(arg :: Int) :: Int:
    this.z + super.mdist(arg)
  method clone(arg :: Int) :: Posn:
    new Posn3D(this.x, this.y,
               this.z)

new Posn3D(5, 7, 3)
```

Yes — which means that we need subtypes

Typechecking Summary

- Use class names as types
- Check for field and method existence
- Check field, method, and argument types
- Check fields against **`new`**
- Check consistency of overrides
- Treat subclasses as subtypes

Part 3

Datatypes

```
type ClassT
| classT(super_name : Symbol,
         fields : Listof(Symbol * Type),
         methods : Listof(Symbol * MethodT))

type MethodT
| methodT(arg_type : Type,
          result_type : Type,
          body_exp : ExpI)
```

Datatypes

```
type Type
| intT()
| objT(class_name :: Symbol)
```

Type Checking

```
fun typecheck(a :: ExpI,
              t_classes :: Listof(Symbol * ClassT)) :: Type:
  begin:
    map(fun (tc):
          typecheck_class(fst(tc), snd(tc), t_classes),
        t_classes)
    typecheck_expr(a, t_classes, objT(#'Object), intT())
```

Type Checking: Classes

```
fun typecheck_class(class_name :: Symbol,
                    t_class :: ClassT,
                    t_classes :: Listof(Symbol * ClassT)):
  match t_class
  | classT(super_name, fields, methods):
      map(fun (m):
            typecheck_method(snd(m), objT(class_name), t_classes)
            check_override(fst(m), snd(m), t_class, t_classes),
          methods)
```

Type Checking: Methods

```
fun typecheck_method(method :: MethodT,
                     this_type :: Type,
                     t_classes :: Listof(Symbol * ClassT)) :: Void:
  match method
  | methodT(arg_type, result_type, body_exp):
      if is_subtype(typecheck_expr(body_exp, t_classes,
                                   this_type, arg_type),
                    result_type,
                    t_classes)
      | #void
      | type_error(body_exp, to_string(result_type))
```

Type Checking: Method Overrides

```
fun check_override(method_name :: Symbol,
                   method :: MethodT,
                   this_class :: ClassT,
                   t_classes :: Listof(Symbol * ClassT)):
  def super_name = classT.super_name(this_class)
  def super_method:
    try:
      find_method_in_tree(method_name,
                          super_name,
                          t_classes)
      ~catch: method
  if methodT.arg_type(method) == methodT.arg_type(super_method)
       && methodT.result_type(method) == methodT.result_type(super_method)
  | #void
  | error(#'typecheck, "bad override of " +& method_name)
```

Part 4

Type Checking Expressions

```
def typecheck_expr :: (ExpI, Listof(Symbol * ClassT), Type, Type) -> Type:
  fun (expr, t_classes, this_type, arg_type):
    fun recur(expr):
      typecheck_expr(expr, t_classes, this_type, arg_type)
    ....
    match expr
    | ....
    | intI(n): intT()
    | ....
    | argI(): arg_type
    | thisI(): this_type
    | ....
```

Type Checking Expressions

```
def typecheck_expr :: (ExpI, Listof(Symbol * ClassT), Type, Type) -> Type:
  fun (expr, t_classes, this_type, arg_type):
    fun recur(expr):
      typecheck_expr(expr, t_classes, this_type, arg_type)
    fun typecheck_nums(l, r):
      match recur(l)
      | intT():
          match recur(r)
          | intT(): intT()
          | ~else: type_error(r, "Int")
      | ~else: type_error(l, "Int")
    match expr
    | ....
    | plusI(l, r): typecheck_nums(l, r)
    | multI(l, r): typecheck_nums(l, r)
    | ....
```

Type Checking Expressions

```
def typecheck_expr :: (ExpI, Listof(Symbol * ClassT), Type, Type) -> Type:
  fun (expr, t_classes, this_type, arg_type):
    fun recur(expr):
      typecheck_expr(expr, t_classes, this_type, arg_type)
    ....
    match expr
    | ....
    | newI(class_name, exprs):
        def arg_types = map(recur, exprs)
        def field_types = get_all_field_types(class_name, t_classes)
        if (length(arg_types) == length(field_types)
               && foldl(fun (b, r): r && b,
                        #true,
                        map2(fun (t1, t2):
                               is_subtype(t1, t2, t_classes),
                             arg_types,
                             field_types)))
        | objT(class_name)
        | type_error(expr, "field type mismatch")
    | ....
```

Type Checking Expressions

```
def typecheck_expr :: (ExpI, Listof(Symbol * ClassT), Type, Type) -> Type:
  fun (expr, t_classes, this_type, arg_type):
    fun recur(expr):
      typecheck_expr(expr, t_classes, this_type, arg_type)
    ....
    match expr
    | ....
    | getI(obj_exp, field_name):
        match recur(obj_exp)
        | objT(class_name):
            find_field_in_tree(field_name,
                               class_name,
                               t_classes)
        | ~else: type_error(obj_exp, "object")
    | ....
```

Type Checking Expressions

```
def typecheck_expr :: (ExpI, Listof(Symbol * ClassT), Type, Type) -> Type:
  fun (expr, t_classes, this_type, arg_type):
    fun recur(expr):
      typecheck_expr(expr, t_classes, this_type, arg_type)
    ....
    match expr
    | ....
    | sendI(obj_exp, method_name, arg_exp):
        def obj_type = recur(obj_exp)
        def arg_type = recur(arg_exp)
        match obj_type
        | objT(class_name):
            typecheck_send(class_name, method_name,
                           arg_exp, arg_type,
                           t_classes)
        | ~else:
            type_error(obj_exp, "object")
    | ....
```

Type Checking Expressions

```
def typecheck_expr :: (ExpI, Listof(Symbol * ClassT), Type, Type) -> Type:
  fun (expr, t_classes, this_type, arg_type):
    fun recur(expr):
      typecheck_expr(expr, t_classes, this_type, arg_type)
    ....
    match expr
    | ....
    | superI(method_name, arg_exp):
        def arg_type = recur(arg_exp)
        def this_class = find(t_classes, objT.class_name(this_type))
        typecheck_send(classT.super_name(this_class),
                       method_name,
                       arg_exp, arg_type,
                       t_classes)
    | ....
```

Type Checker: Sends

```
fun typecheck_send(class_name :: Symbol,
                   method_name :: Symbol,
                   arg_exp :: ExpI,
                   arg_type :: Type,
                   t_classes :: Listof(Symbol * ClassT)):
  match find_method_in_tree(method_name,
                            class_name,
                            t_classes)
  | methodT(arg_type_m, result_type, body_exp):
      if is_subtype(arg_type, arg_type_m, t_classes)
      | result_type
      | type_error(arg_exp, to_string(arg_type_m))
```

Part 5

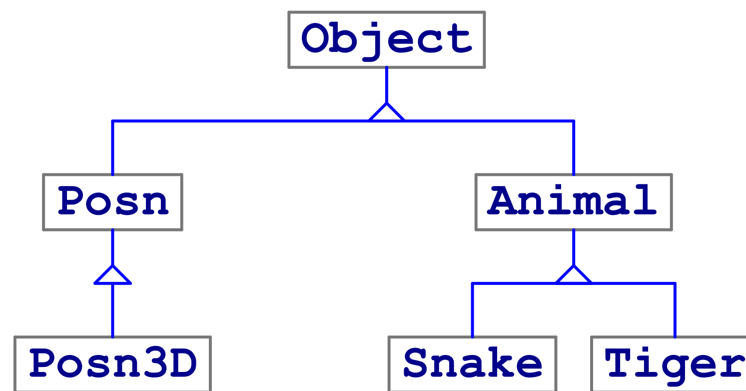
Type Checker: Subtypes

```
class Posn(x :: Int, y :: Int):
  extends Object
  method mdist(arg :: Int) :: Int:
    this.x + this.y
  method addDist(arg :: Posn) :: Int:
    this.mdist(0) + arg.mdist(0)
```

```
class Posn3D(z :: Int):
  extends Posn
  method mdist(arg :: Int) :: Int:
    this.z + super.mdist(arg)
```

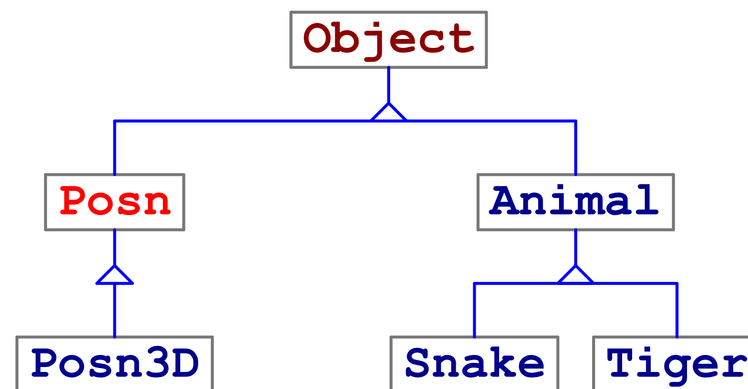
```
new Posn3D(7, 5, 3).mdist(0)
```

Type Checker: Subtypes



Type Checker: Subtypes

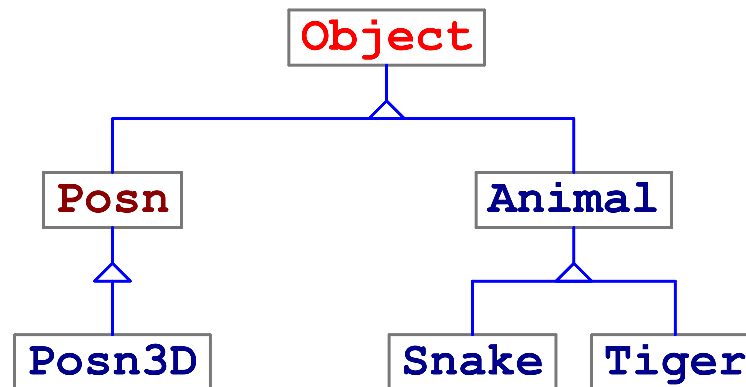
`Posn` is a subtype of `Object`?



Yes — starting from `Posn` reaches `Object`

Type Checker: Subtypes

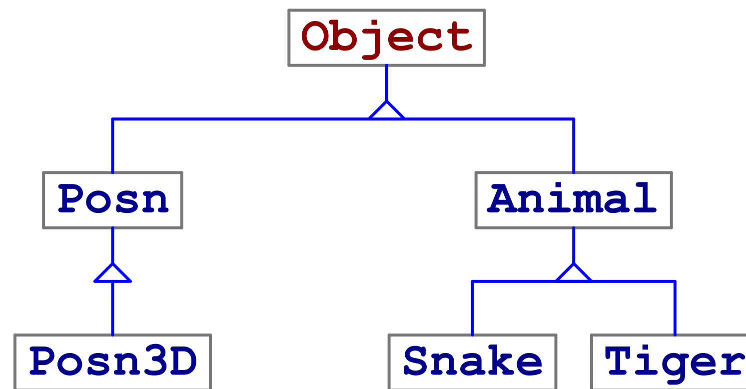
`Object` is a subtype of `Posn`?



No — starting from `Object` doesn't reach `Posn`

Type Checker: Subtypes

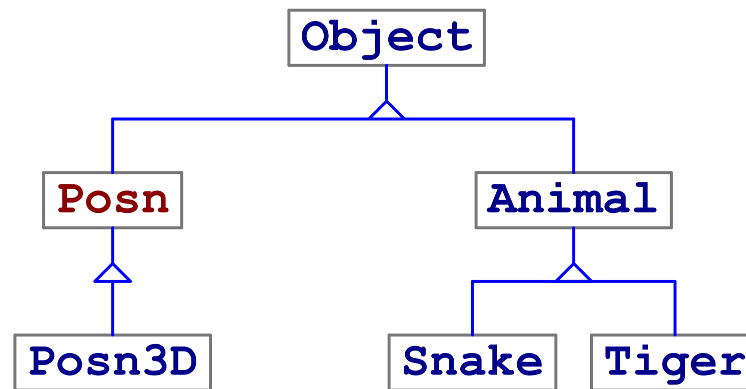
Object is a subtype of **Object**?



Yes — match at start

Type Checker: Subtypes

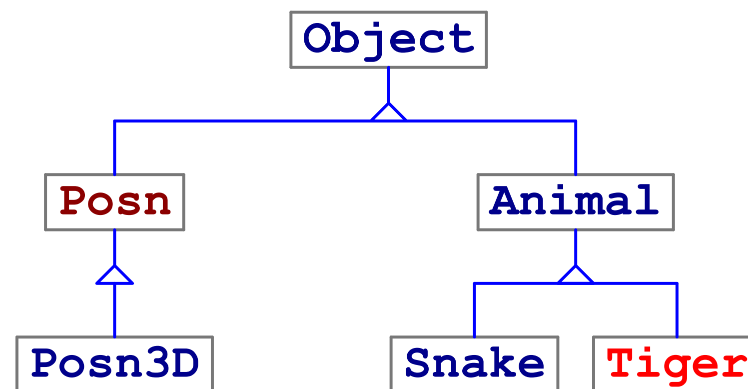
Posn is a subtype of **Posn**?



Yes — match at start

Type Checker: Subtypes

`Tiger` is a subtype of `Posn`?



No — starting from `Tiger` doesn't reach `Posn`

Type Checker: Subtypes

```
fun is_subclass(name1, name2, t_classes):
  cond
  | name1 == name2: #true
  | name1 == #'Object: #false
  | ~else:
      match find(t_classes, name1)
      | classT(super_name, fields, methods):
          is_subclass(super_name, name2, t_classes)

fun is_subtype(t1, t2, t_classes):
  match t1
  | objT(name1):
      match t2
      | objT(name2):
          is_subclass(name1, name2, t_classes)
      | ~else: #false
  | ~else: t1 == t2
```

Part 6

Implementing Classes

ClassT

types

```
class Posn(x :: Int, y :: Int):
  extends Object
  method mdist(arg :: Int) :: Int:
    this.x + this.y
  method addDist(arg :: Posn) :: Int:
    this.mdist(0) + arg.mdist(0)
class Posn3D(z :: Int):
  extends Posn
  method mdist(arg :: Int) :: Int:
    this.z + super.mdist(arg)
new Posn3D(7, 5, 3).mdist(0)
```

ClassI

inheritance

super

```
class Posn(x, y):
  extends Object
  method mdist(arg): this.x + this.y
  method addDist(arg): this.mdist(0) + arg.mdist(0)
class Posn3D(z):
  extends Posn
  method mdist(arg): this.z + super.mdist(arg)
new Posn3D(7, 5, 3).mdist(0)
```

Class

method dispatch

fields

```
class Posn(x, y):
  method mdist(arg): this.x + this.y
  method addDist(arg): this.mdist(0) + arg.mdist(0)
class Posn3D(x, y, z):
  method mdist(arg): this.z + (this :: Posn).mdist(arg)
  method addDist(arg): this.mdist(0) + arg.mdist(0)
new Posn3D(7, 5, 3).mdist(0)
```

Interpreter

```
def interp_t :: (ExpI, Listof(Symbol * ClassT)) -> Value:
  fun (a, t_classes):
    interp_i(a,
             map(fun (c):
                   values(fst(c), strip_types(snd(c))),
                 t_classes))

def strip_types :: ClassT -> ClassI:
  fun (t_class):
    match t_class
    | classT(super_name, fields, methods):
        classI(super_name,
               map(fst, fields),
               map(fun (m):
                     values(fst(m),
                            match snd(m)
                            | methodT(arg_type, result_type, body_exp):
                                body_exp),
                   methods))
```