Part I
Classes

{class Posn extends Object
  {x y}
  [mdist {arg} {+ {get this x} {get this y}}]
  [addDist {arg} {+ {send arg mdist 0}
      {send this mdist 0}}]
}

{class Posn3D extends Posn
  {z}
  [mdist {arg} {+ {get this z}
      {super mdist arg}}]
}

{send {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 {\text{new Posn 1 2}}\}
Typechecking Programs with Classes

A well-formed program should never error with

- not a number
- not an object

\{\texttt{send 1 mdist 0}\}
Typechecking Programs with Classes

A well-formed program should never error with

• not a number
• not an object

{get 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

```c
{get {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

\{\text{send} \ \{\text{new} \ \text{Posn 1 2}\} \ \text{area}\}
A well-formed program should never error with

• not a number

• not an object

• wrong field count

• not found
  ◦ class, field, or method

```java
{class Circle extends Object
{}
[area {arg} {super area arg}]}
```
Typed Class Language

\[
\text{<Class>} ::= \{ \text{class } \text{<Symbol>} \text{ extends } \text{<Symbol>}
\]
\[
\quad \{ \text{<Field>}\}^* \\
\quad \text{<Method>}\}^*
\]
\[
\text{<Field>} ::= [ \text{<Symbol>} : \text{<Type>} ]
\]
\[
\text{<Method>} ::= [ \text{<Symbol>} \{ [\text{arg} : \text{<Type>}] \} : \text{<Type>} \text{<Expr>} ]
\]
\[
\text{<Type>} ::= \text{num}
\]
\[
\quad | \quad \text{<Symbol>}
\]
Part 2
Typechecking Programs with Classes

Is this program well-formed?

```plaintext
{class Posn extends Object
  {[x : num] [y : num]}
  [mdist {[arg : num]} : num
    {+ {send {get this x} mdist 0}
     {send {get this y} mdist 0}]]}
```

10

**No** — the `x` and `y` fields are not objects
Typechecking Programs with Classes

Is this program well-formed?

```java
{class Posn extends Object
  {[x : num] [y : num]}
  [mdist {[arg : num]} : num
    {+ {get this x} {get this z}]]}

10

No — Posn has no z field
Typechecking Programs with Classes

Is this program well-formed?

```java
{class Posn extends Object
    {[x : num] [y : num]
    [mdist {[arg : num]} : num
        {+ {get this x} {send this get-y 0}]]}}
```

10

**No — Posn** has no **get-y** method
Typechecking Programs with Classes

Is this program well-formed?

```java
{class Posn extends Object
  {[x : num] [y : num]}
  [mdist {[arg : num]} : Posn
    {+ {get this x} {get this y}}]}

10

No — result type for mdist does not match body type
Typechecking Programs with Classes

Is this program well-formed?

```java
{class Posn extends Object
  {[x : num] [y : num]}
  [mdist {[arg : num]} : num
    {[get this x} {get this y}]}

10

Yes
```
Typechecking Programs with Classes

Is this program well-formed?

```java
{class Posn extends Object
  {[x : num] [y : num]}
  [mdist {[arg : num]} : num
    {+ {get this x} {get this y}]]}

{new Posn 12}

No — wrong number of fields in new
Typechecking Programs with Classes

Is this program well-formed?

```java
{class Posn extends Object
   {[x : num] [y : num]}
   [mdist {[arg : num]} : num
    {+ {get this x} {get 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 extends Object
  {{x : num} [y : num]}
  [mdist {{arg : num}} : num
    {+ {get this x} {get this y}}]
  [clone {{arg : num}} : Posn
    {new Posn {get this x} {get this y}}]}

{send {new Posn 1 2} clone 0}

Yes
Typechecking Programs with Classes

Is this program well-formed?

```java
{class Posn extends Object
  {{x : num] [y : num]}
  [mdist {{arg : num}] : num
    {+ {get this x} {get this y}}]
  [clone {{arg : num}] : Posn
    {new Posn {get this x} {get this y}}}
}

{class Posn3D extends Posn
  {{z : num]}
  [mdist {{arg : num}] : num
    {+ {get this z} {super mdist arg}}}
}

{new Posn3D 5 7 3}

Yes
```
Typechecking Programs with Classes

Is this program well-formed?

```java
{class Posn extends Object
  [[x : num] [y : num]]
  [mdist {[arg : num]} : num
   {+ {get this x} {get this y}}]
  [clone {[arg : num]} : Posn
   {new Posn {get this x} {get this y}}]}

{class Posn3D extends Posn
  [[z : num]]
  [mdist {[arg : num]} : 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?

```java
{class Posn extends Object
  {{[x : num] [y : num]}
    [mdist {[arg : num]} : num
    {+ {get this x} {get this y}]]
    [clone {[arg : num]} : Posn
    {new Posn {get this x} {get this y}]}]}

{class Posn3D extends Posn
  {{[z : num]}
    [mdist {[arg : num]} : num
    {+ {get this z} {super mdist arg}]]
    [clone {[arg : num]} : num
    {10}}]

{new Posn3D 5 7 3}

No — override of clone changes result type
```
Typechecking Programs with Classes

Is this program well-formed?

```java
{class Posn extends Object
  {[x : num] [y : num]}
  [mdist {[arg : num]} : num
    {+ {get this x} {get this y}}]
  [clone {[arg : num]} : Posn
    {new Posn {get this x} {get this y}}]}

{class Posn3D extends Posn
  {[z : num]}
  [mdist {[arg : num]} : num
    {+ {get this z} {super mdist arg}}]
  [clone {[arg : num]} : Posn
    {new Posn3D {get this x} {get this y}
      {get this z}}]}

{new Posn3D 5 7 3}
```

**Yes** — which means that we need subtypes
Typechecking Summary

• Use class names as type
• 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

(define-type ClassT
  (classT [super-name : Symbol]
  [fields : (Listof (Symbol * Type))]
  [methods : (Listof (Symbol * MethodT))]))

(define-type MethodT
  (methodT [arg-type : Type]
  [result-type : Type]
  [body-expr : ExpI]))
Datatypes

(define-type Type
  (numT)
  (objT [class-name : Symbol]))
Type Checking

(define (typecheck [a : ExpI]
    [t-classes : (Listof (Symbol * ClassT))]) : Type
  (begin
    (map (lambda (tc)
           (typecheck-class (fst tc) (snd tc) t-classes))
         t-classes)
    (typecheck-expr a t-classes (objT 'Object) (numT))))
Type Checking: Classes

(define (typecheck-class [class-name : Symbol]
    [t-class : ClassT]
    [t-classes : (Listof (Symbol * ClassT))])

(type-case ClassT t-class
    [(classT super-name fields methods)
        (map (lambda (m)
            (begin
              (typecheck-method (snd m) (objT class-name) t-classes)
              (check-override (fst m) (snd m) t-class t-classes))
            methods))])
Type Checking: Methods

(define (typecheck-method [method : MethodT]
    [this-type : Type]
    [t-classes : (Listof (Symbol * ClassT))]): ()
(type-case MethodT method
  [(methodT arg-type result-type body-expr)
    (if (is-subtype? (typecheck-expr body-expr t-classes
                          this-type arg-type)
                    result-type
                    t-classes)
        (values)
        (type-error body-expr (to-string result-type))))])}
Type Checking: Method Overrides

(define (check-override [method-name : Symbol]
    [method : MethodT]
    [this-class : ClassT]
    [t-classes : (Listof (Symbol * ClassT))])

(local [(define super-name
          (classT-super-name this-class))
        (define super-method
          (try
            (find-method-in-tree method-name
                                    super-name
                                    t-classes)
            (lambda () method)))]
      (if (and (equal? (methodT-arg-type method)
                        (methodT-arg-type super-method))
              (equal? (methodT-result-type method)
                      (methodT-result-type super-method)))
       (values)
       (error 'typecheck (string-append
                               "bad override of 
                               (to-string method-name))))))
Part 4
Type Checking Expressions

(define typecheck-expr : (ExpI (Listof (Symbol * ClassT)) Type Type -> Type)
(lambda (expr t-classes this-type arg-type)
  (local [(define (recur expr)
    (typecheck-expr expr t-classes this-type arg-type))
    ....]
  (type-case ExpI expr
    ....
    [(numI n) (numT)]
    ....
    [(argI) arg-type]
    [(thisI) this-type]
    ....))))
Type Checking Expressions

(define typecheck-expr : (ExpI (Listof (Symbol * ClassT)) Type Type -> Type)
  (lambda (expr t-classes this-type arg-type)
    (local [(define (recur expr)
               (typecheck-expr expr t-classes this-type arg-type))
            (define (typecheck-nums l r)
              (type-case Type (recur l)
                [(numT)
                 (type-case Type (recur r)
                   [(numT) (numT)]
                   [else (type-error r "num")]]]
                [else (type-error l "num")]))]
     (type-case ExpI expr
       .....
       [(plusI l r) (typecheck-nums l r)]
       [(multI l r) (typecheck-nums l r)]
       .....)))
Type Checking Expressions

(define typecheck-expr : (ExpI (Listof (Symbol * ClassT)) Type Type -> Type)
  (lambda (expr t-classes this-type arg-type)
    (local [(define (recur expr)
               (typecheck-expr expr t-classes this-type arg-type))
           ....]
    (type-case ExpI expr
      ....
      [(newI class-name exprs)
       (local [(define arg-types (map recur exprs))
                (define field-types
                  (get-all-field-types class-name t-classes))]
       (if (and (= (length arg-types) (length field-types))
                 (foldl (lambda (b r) (and r b))
                       #t
                       (map2 (lambda (t1 t2)
                               (is-subtype? t1 t2 t-classes)
                               arg-types
                               field-types)))
        (objT class-name)
        (type-error expr "field type mismatch"))]
      ....)))))
Type Checking Expressions

(define typecheck-expr : (ExpI (Listof (Symbol * ClassT)) Type Type -> Type)
(lambda (expr t-classes this-type arg-type)
  (local [(define (recur expr)
                (typecheck-expr expr t-classes this-type arg-type))
           ....]
    (type-case ExpI expr
      ....
      [(getI obj-expr field-name)
       (type-case Type (recur obj-expr)
         [(objT class-name)
          (find-field-in-tree field-name class-name t-classes)]
        [else (type-error obj-expr "object")]]
      ....)))))
(define typecheck-expr : (ExpI (Listof (Symbol * ClassT)) Type Type -> Type)
(lambda (expr t-classes this-type arg-type)
  (local [(define (recur expr)
            (typecheck-expr expr t-classes this-type arg-type))
    ....]
(type-case ExpI expr
  ....
  [(sendI obj-expr method-name arg-expr)
    (local [(define obj-type (recur obj-expr))
             (define arg-type (recur arg-expr))]
      (type-case Type obj-type
        [(objT class-name)
          (typecheck-send class-name method-name
                           arg-expr arg-type
                           t-classes)]
        [else
          (type-error obj-expr "object")]]))
  ....))))
Type Checking Expressions

(define typecheck-expr : (ExpI (Listof (Symbol * ClassT)) Type Type -> Type)
  (lambda (expr t-classes this-type arg-type)
    (local [(define (recur expr)
              (typecheck-expr expr t-classes this-type arg-type))
      ....]
    (type-case ExpI expr
      ....
      [(superI method-name arg-expr)
        (local [(define arg-type (recur arg-expr))
                  (define this-class
                    (find t-classes (objT-class-name this-type)))]
          (typecheck-send (classT-super-name this-class)
                           method-name
                           arg-expr arg-type
                           t-classes))]
      ....)))
Type Checker: Sends

(define (typecheck-send [class-name : Symbol]
    [method-name : Symbol]
    [arg-expr : ExpI]
    [arg-type : Type]
    [t-classes : (Listof (Symbol * ClassT))])

(type-case MethodT (find-method-in-tree
    method-name
    class-name
    t-classes)

  [(methodT arg-type-m result-type body-expr)
   (if (is-subtype? arg-type arg-type-m t-classes)
       result-type
       (type-error arg-expr (to-string arg-type-m)))]))
Part 5
Type Checker: Subtypes

{class Posn extends Object
  {[x : num] [y : num]}
  [mdist {[arg : num]} : num
    {+ {get this x} {get this y}]]
  [addDist {[arg : Posn]} : num
    {+ {send this mdist 0} {send arg mdist 0}]]}

{class Posn3D extends Posn
  {[z : num]}
  [mdist {[arg : num]} : num
    {+ {get this z} {super mdist arg}]]
{send {new Posn3D 7 5 3} mdist 0}
Type Checker: Subtypes

```
Object
    Posn
    Posn3D
    Animal
    Snake
    Tiger
```
Type Checker: Subtypes

\textbf{Posn} is a subtype of \textbf{Object}?

Yes — starting from \textbf{Posn} reaches \textbf{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

(define (is-subclass? name1 name2 t-classes)
  (cond
    [(equal? name1 name2) #t]
    [(equal? name1 'Object) #f]
    [else
      (type-case ClassT (find t-classes name1)
        [(classT super-name fields methods)
          (is-subclass? super-name name2 t-classes)]))]

(define (is-subtype? t1 t2 t-classes)
  (type-case Type t1
    [(objT name1)
      (type-case Type t2
        [(objT name2)
          (is-subclass? name1 name2 t-classes)]
        [else #f])]
    [else (equal? t1 t2)])
Part 6
Implementing Classes

```java
{class Posn extends Object
  {{x : num} [y : num]}
  [mdist {{arg : num}} : num
    {+ {get this x} {get this y}}]
  [addDist {{arg : Posn}} : num
    {+ {send this mdist 0} {send arg mdist 0}}]}
{class Posn3D extends Posn
  {{z : num]}
  [mdist {{arg : num}} : num
    {+ {get this z} {super mdist arg}}]}
{send {new Posn3D 7 5 3} mdist 0}

{class Posn extends Object
  {x y}
  [mdist {arg} {+ {get this x} {get this y}}]
  [addDist {arg} {+ {send this mdist 0} {send arg mdist 0}}]}
{class Posn3D extends Posn
  {z}
  [mdist {arg} {+ {get this z} {super mdist arg}}]}
{send {new Posn3D 7 5 3} mdist 0}

{class Posn
  {x y}
  [mdist {arg} {+ {get this x} {get this y}}]
  [addDist {arg} {+ {dsend this mdist 0} {dsend arg mdist 0}}]}
{class Posn3D
  {x y z}
  [mdist {arg} {+ {get this z} {ssend this Posn mdist arg}}]
  [addDist {arg} {+ {dsend this mdist 0} {dsend arg mdist 0}}]}
{dsend {new Posn3D 7 5 3} mdist 0}
```
Interpreter

(define interp-t : (ExpI (Listof (Symbol * ClassT)) -> Value)
  (lambda (a t-classes)
    (interp-i a
      (map (lambda (c)
          (values (fst c) (strip-types (snd c)))
        t-classes))))

(define strip-types : (ClassT -> ClassI)
  (lambda (t-class)
    (type-case ClassT t-class
      [(classT super-name fields methods)
        (classI
          super-name
          (map fst fields)
          (map (lambda (m)
              (values (fst m)
                (type-case MethodT (snd m)
                  [(methodT arg-type result-type body-expr)
                    body-expr]]))
          methods)]))))