Part I
Classes

```java
{class posn extends object
    {x y}
    {mdist {+ {get this x} {get this y}}} 
    {addDist {+ {send arg mdist 0} 
        {send this mdist 0}}}}}

{class posn3D extends posn
    {z}
    {mdist {+ {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

```language
{+ 1 {new posn 1 2}}
```
Typechecking Programs with Classes

A well-formed program should never error with

• not a number
• not an object

{send 1 mdist 0}
Typechecking Programs with Classes

A well-formed program should never error with

• not a number
• not an object

\{\texttt{get} \ 1 \ \texttt{x}\}
Typechecking Programs with Classes

A well-formed program should never error with

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

{\texttt{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 square-circle}
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

{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

{send {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

```java
{class circle extends object
    {}}
    {area {super area arg}}}
```
Typed Class Language

```
<Class> ::= {class <Sym> extends <Sym>
               {<Field>*}
               <Method>*}
<Field> ::= [<Sym> : <Type>]
(Method) ::= {<Sym> : <Type> -> <Type> <Expr>}
<Type> ::= num
         | <Sym>
```
Part 2
Typechecking Programs with Classes

Is this program well-formed?

```plaintext
{class posn extends object
  {[x : num] [y : num]}
  {mdist : 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?

```
{class posn extends object
  {[x : num] [y : num]}
  {mdist : 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 : 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?

```
{class posn extends object
  {[x : num] [y : num]}
  {mdist : 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?

```
{class posn extends object
  {[x : num] [y : num]}
  {mdist : 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 : 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 : 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?

```java
{class posn extends object
  {[x : num] [y : num]}
  {mdist : num -> num
   {+ {get this x} {get this y}}}
  {clone : 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 : num -> num
        {+ {get this x} {get this y}}}
    {clone : num -> posn
        {new posn {get this x} {get this y}}}}

{class posn3D extends posn
    {[z : num]}
    {mdist : 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 : num -> num
        {+ {get this x} {get this y}]
    {clone : num -> posn
        {new posn {get this x} {get this y}]]

{class posn3D extends posn
    {[[z : num]]
    {mdist : 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 : num -> num
        {+ {get this x} {get this y}}}
    {clone : num -> posn
        {new posn {get this x} {get this y}}}}

{class posn3D extends posn
    {[z : num]}
    {mdist : num -> num
        {+ {get this z} {super mdist arg}}}
    {clone : num -> num
        10}}

{new posn3D 5 7 3}

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

Is this program well-formed?

```plaintext
{class posn extends object
  {[x : num] [y : num]}
  {mdist : num -> num
    {+ {get this x} {get this y} }}
  {clone : num -> posn
    {new posn {get this x} {get this y} }} }}

{class posn3D extends posn
  {[z : num]}
  {mdist : num -> num
    {+ {get this z} {super mdist arg} }}
  {clone : 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 (name : symbol)
   (super-name : symbol)
   (fields : (listof FieldT))
   (methods : (listof MethodT))])

(define-type FieldT
  [fieldT (name : symbol)
   (type : Type)])

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

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

```
(define (typecheck [a : ExprI] [t-classes : (listof ClassT)]) : Type
  (begin
    (map (lambda (t-class)
           (typecheck-class t-class t-classes))
         t-classes)
    (typecheck-expr a t-classes (numT) (objT 'bad))))
```
Type Checking: Classes

(define (typecheck-class [t-class : ClassT]
                            [t-classes : (listof ClassT)]

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

```
(define (typecheck-method [method : MethodT] [this-type : Type] [t-classes : (listof ClassT)])
  (type-case MethodT method
    [methodT (name arg-type result-type body-expr)
      (if (is-subtype? (typecheck-expr body-expr t-classes arg-type this-type)
        result-type
t-classes)
        (values)
        (type-error body-expr
          (to-string result-type)))]))
```
(define (check-override [method : MethodT]
    [this-class : ClassT]
    [t-classes : (listof ClassT)]))

(local [(define super-name
          (classT-super-name this-class))
        (define super-method
          (try
           ; Look for method in superclass:
           (find-method-in-tree (methodT-name method)
                                 (find-classT super-name t-classes)
                                 t-classes)
           ; no such method in superclass:
           (lambda () method)))]
       (if (and (equal? (methodT-arg-type method)
                         (methodT-arg-type super-method))
                (equal? (methodT-result-type method)
                         (methodT-result-type type super-method))
                (values)
                (error 'typecheck (string-append
                                "bad override of 
                                (to-string (methodT-name method)))))))
Part 4
Type Checking Expressions

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

(define typecheck-expr : (ExprI (listof ClassT) Type Type -> Type)
  (lambda (expr t-classes arg-type this-type)
    (local [(define (recur expr)
              (typecheck-expr expr t-classes arg-type this-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 ExprI expr
      ....
      [plusI (l r) (typecheck-nums l r)]
      [multI (l r) (typecheck-nums l r)]
      ....)))
Type Checking Expressions

(define typecheck-expr : (ExprI (listof ClassT) Type Type -> Type)
  (lambda (expr t-classes arg-type this-type)
    (local [(define (recur expr)
            (typecheck-expr expr t-classes arg-type this-type))
      ....
    (type-case ExprI 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))
                true
              (map2 (lambda (t1 t2)
                    (is-subtype? t1 t2 t-classes))
                arg-types
                field-types))
            (objT class-name)
            (type-error expr "field type mismatch"))]
      ....)))))

49
Type Checking Expressions

(define typecheck-expr : (ExprI (listof ClassT) Type Type -> Type)
  (lambda (expr t-classes arg-type this-type)
    (local [(define (recur expr)
               (typecheck-expr expr t-classes arg-type this-type))
      ....]
    (type-case ExprI expr
      ....
      [getI (obj-expr field-name)
        (type-case Type (recur obj-expr)
          [objT (class-name)
            (local [(define t-class
                         (local [(define t-class
                                      (find-classT class-name t-classes))
                        (define field
                          (find-field-in-tree field-name
t                            t-class
t                            t-classes))])
              (type-case FieldT field
                [fieldT (name type) type]]))]
          [else (type-error obj-expr "object")]]
      ....)))}
Type Checking Expressions

(define typecheck-expr : (ExprI (listof ClassT) Type Type -> Type)
  (lambda (expr t-classes arg-type this-type)
    (local [(define (recur expr)
      (typecheck-expr expr t-classes arg-type this-type))
      ....]
    (type-case ExprI expr
      ....
      [sendI (obj-exp expr method-name arg-exp)
      (local [(define obj-type (recur obj-exp))
        (define arg-type (recur arg-exp))]
      (type-case Type 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

(define typecheck-expr : (ExprI (listof ClassT) Type Type -> Type)
 (lambda (expr t-classes arg-type this-type)
     (local [(define (recur expr)
                 (typecheck-expr expr t-classes arg-type this-type))
         ....]
     (type-case ExprI expr
         ....
         [superI (method-name arg-exprr)
                 (local [(define arg-type (recur arg-exprr))
                         (define this-class
                         (find-classT (objT-class-name this-type) t-classes))]
                 (typecheck-send (classT-super-name this-class) method-name
                                 arg-exprr arg-type t-classes))]
         ....)))))
(define (typecheck-send [class-name : symbol]
    [method-name : symbol]
    [arg-expr : ExprI]
    [arg-type : Type]
    [t-classes : (listof ClassT)])
(type-case MethodT (find-method-in-tree
    method-name
    (find-classT class-name t-classes)
    t-classes)
    [methodT (name 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)))]))
Type Checker: Subtypes

```
(define (is-subclass? name1 name2 t-classes)
  (cond
    [(equal? name1 name2) true]
    [(equal? name1 'object) false]
    [else
     (type-case ClassT (find-classT name1 t-classes)
        [classT (name 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 false])]
    [else (equal? t1 t2)])
)```
Part 5
Interpreter

(define interp-t : (ExprI (listof ClassT) -> Value)
  (lambda (a t-classes)
    (interp-i a
     (map strip-types t-classes)))))

(define strip-types : (ClassT -> ClassI)
  (lambda (t-class)
    (type-case ClassT t-class
      [classT (name super-name fields methods)
        (classI name
          super-name
          (map fieldT-name fields)
          (map (lambda (m)
            (type-case MethodT m
              [methodT (name arg-type res-type body-expr)
                (methodI name body-expr)])
          methods))]))))
Implementing Classes

```plaintext
{class posn extends object
  {[x : num] [y : num]}
  {mdist : num -> num
    {+ {get this x} {get this y}]]
  }
  {addDist : posn -> num
    {+ {send this mdist 0} {send arg mdist 0}]]]]
  }
{class posn3D extends posn
  {[z : num]}
  {mdist : num -> num
    {+ {get this z} {super mdist arg}]]
  }
  {send {new posn3D 7 5 3} mdist 0}

{class posn extends object
  {x y]
    {mdist {+ {get this x} {get this y}]]
  }
  {addDist {+ {send this mdist 0} {send arg mdist 0]}}

{class posn3D extends posn
  {z]
    {mdist {+ {get this z} {super mdist arg}]]
  }
  {send {new posn3D 7 5 3} mdist 0}

{class posn
  {x y]
    {mdist {+ {get this x} {get this y}]]
  }
  {addDist {+ {dsend this mdist 0} {dsend arg mdist 0]}}

{class posn3D
  {x y z]
    {mdist {+ {get this z} {ssend this posn mdist arg}]]
  }
  {addDist {+ {dsend this mdist 0} {dsend arg mdist 0]}}
  {dsend {new posn3D 7 5 3} mdist 0}
```