**Diesel Highlights**

Purely object-oriented language
- all data are instances of classes
- all operations & control structures
  via dynamically dispatched method calls

Multiply dispatched method calls

Closures, a.k.a. first-class lexically-nested functions

Static type system
- including fancy polymorphic types

Module system
- namespace management & encapsulation

Type-safe
Garbage-collected

**Class declarations**

To declare a class, use a class declaration, e.g.:

```plaintext
class shape;
class rectangle isa shape;
class rhombus isa shape;
class square isa rectangle, rhombus;
```

A class can have zero, one, or many superclasses
- multiple inheritance supported

A class *doesn’t* declare any of its fields or methods;
these are separate top-level declarations

Can add new superclasses to existing classes externally,
e.g. in separate source files!
```plaintext
class printable;
extend class shape isa printable;
```

Each class defines a new type
- a subclass is a subtype
**Field declarations**

To declare the instance variables of a class, use `field` declarations.

E.g.:

```kotlin
var field center(s:shape):point {new_origin()}
field width(r:rectangle):num;
field height(r:rectangle):num { r.width }
```

Fields are declared separately from classes:
- the field is related to its “containing” class via the type of the field’s argument
- each object of that type (or subtype) stores a value for the field
- can add new fields to classes externally!

Must say `var` for assignable field
- immutable by default

A field can have a default initial value
- can be an expression, e.g. computing the field’s initial value from the initial values of other fields

**Function declarations**

To declare a new top-level procedure, constructor, or method, use a `fun` declaration, e.g.:

```kotlin
fun new_point(x:num, y:num):point { ... }
fun new_origin():point { new_point(0,0) }
fun rect_area(r:rectangle):num {
    r.width * r.height }
fun move_rect(r:rectangle,
    new_center:point):void {
    r.center := new_center;
}
```

Functions are declared separately from classes:
- receiver argument (if any) is explicit
- constructors have explicit names
- can add new functions to classes externally!

Can have different functions with same name but different numbers of arguments (a kind of static overloading)

A function body is a sequence of zero or more statements, followed by an optional result expression (`void` if absent)
Method declarations

Override an existing function for particular kinds of arguments using a method declaration

- method has same name and number of arguments as overridden function
- one or more formals’ types declared using @ instead of :
  - method applies only to run-time arguments whose dynamic class is an instance of the @ type, called the specializer
- can override a method, too
  - more specific @ types override less specific ones

E.g.:

```language
fun resize(s:shape, dw:num, dh:num):shape {...}
method resize(r@rectangle,    
    dw:num, dh:num):rectangle {...}
method resize(s@square,      
    dw:num, dh:num):rectangle {...}
method resize(c@circle,      
    dw:num, dh:num):shape {...}
```

Method body same syntax as function body

Abstract classes and functions

A class can be abstract

- can’t have direct instances

E.g.:

```language
abstract class shape;
```

A function declared for an abstract class need not have a body

- must be overridden by some method for every concrete subclass

E.g.:

```language
fun resize(s:shape, dw:num, dh:num):shape;
 -- must have resize methods for all concrete
 -- subclasses of shape
```
Multiple dispatching

Can have multiple @ specialized formals in a method
⇒ multiple dispatching

E.g.:

```latex
fun = (s1:shape, s2:shape):bool \{ false \}
method = (r1@rectangle, r2@rectangle):bool \{ .. \}
method = (c1@circle, c2@circle):bool \{ ... \}
```

All arguments treated uniformly
- any can be specialized, or not
- any number can be specialized
- specialization is always based on\textit{ dynamic} argument class,
  not \textit{static} argument type

Method lookup rules

When invoke a function with some arguments
(a.k.a. send a message),
need to identify the right method to run
- consider a function with a body as a method with no @

Algorithm:

1. find set of\textit{ applicable} methods in invoked function
   - a method is applicable if, for each @\textit{C} formal, the dynamic class
     of the corresponding argument is equal to or a subclass of \textit{C}
   - if no applicable methods: report msg-not-understood error
2. select unique\textit{ most-specific} applicable method
   - a method is at least as specific as another if
     its specializers are uniformly at least as specific as the other’s
   - if no uniquely most specific method: report msg-ambiguous error
3. run it

Static typechecking checks for these method lookup errors
Constraints on method types

Method argument and result types must conform to overridee function/method’s

- method’s @ formal types should be more specific than overridee’s [covariant]
- otherwise, wouldn’t override!
- safe, since tested dynamically via method lookup
- method’s : formal types should be as general as (typically, the same as) the overridee’s [contravariant]
- method’s result type can be more specific than overridee’s [covariant]

E.g.:

```plaintext
fun resize(s:shape, dw:num, dh:num):shape;
method resize(r@rectangle, dw:num, dh:num):rectangle {...}
method resize(s@square, dw:num, dh:num):rectangle {...}
method resize(c@circle, dw:num, dh:num):shape {...}
```

Constraints ensure that if a call to a function typechecks, then no matter what method is invoked, its formal and result types will be compatible with the caller’s expectations

Object creation

Create new instances of a class by evaluating new expressions

- can provide initial values for fields, or rely on fields’ defaults, which are evaluated separately for each object

E.g.:

```plaintext
fun new_rectangle(w:num, h:num):rectangle {
    new rectangle {
        -- center gets default value
        width := w, height := h }
}

fun new_square(w:num):square {
    new square {
        -- center gets default value
        -- height derived from width by default initializer
        width := w }
}
```

Good programming style:
- encapsulate all new expressions inside functions

Unlike traditional constructors, these functions:

+ can cache and return previously created objects
+ can create instances of different classes based on e.g. args
- cannot inherit field initialization code
Object declarations

Can declare one-of-a-kind objects using object declarations
• similar syntax to class declarations
• also can specify initial values for fields

E.g.:
object unit_square isa square { width := 1 };

Can inherit from and specialize on named objects just like
classes
• cannot do that for anonymous objects created with new

Can reference named objects directly just like global variables
• cannot do that for classes

Expressions and statements

Constants, e.g.: 3, -4, 5.6, "hi there
bob", 'a'
• all values are regular, first-class objects,
e.g. 3 is an instance of prim_int class

Vector constructors, e.g.: [], [3+x, y*z, f(x)]
• vectors are regular, first-class objects too

new expressions, e.g.: new rectangle { ... }

Identifiers, e.g.: x, joe_Bob_17, true, void
• reference local var, formal, global var, or named object

Variable declaration statements, e.g.:
let w := y * z;
let var x:int := w + f(w);
• variables must be initialized at declaration
• assignable variables and globals should be given types

Assignment stmts, e.g.: x := y * f(z);
• cannot assign to formals or non-var locals/globals
Messages

Use standard procedure-call syntax to invoke a function with zero or more arguments:

```plaintext
start_prog()
center(r)
set_center(r, c)
draw(r, window, loc)
```

Infix & prefix syntax:

```plaintext
x + - y << z ** q!i
```

- any sequence of punctuation chars is a legal infix or prefix message name
- implement with normal functions & methods
- can specify precedence & associativity

Syntactic sugar supports standard “dot notation”:

```plaintext
r.center ⇔ center(r)
r.set_center(c) ⇔ set_center(r, c)
r.draw(window, loc) ⇔ draw(r, window, loc)
```

Syntactic sugar for set_ messages to look like assignments:

```plaintext
r.center := c; ⇔ set_center(r, c);
```

Accessing fields

Access fields solely by sending messages

- to read a field named `f` of object `o`, send `f` message to `o`, to invoke “get accessor” implicit method
- syntactic sugar: `o.f`
- to update a (var) field named `f` of object `o` to new value `v`, send `set_f` message to `o` and `v`, to invoke “set accessor” implicit method
- syntactic sugar: `o.f := v`;

Syntactic sugar makes accessing fields by messages syntactically “natural”

- can access methods as if they were fields, too

Allows fields to be reimplemented as methods & vice versa, and allows fields to be overridden with methods & vice versa, without rewriting callers

No explicit accessor methods or “properties” needed
**Resends**

In overriding method, can invoke overridden method, e.g.:

```plaintext
class visible_rectangle isa rectangle;
method resize(r@visible_rectangle, dw:num, dh:num):rectangle {...}
    let new_r := resend(r, dw, dh);
    r.undisplay;
    new_r.display;
    new_r }
```

Can use to resolve ambiguities, e.g.:

```plaintext
class square isa rectangle, rhombus;
fun area(s:shape):num;
method area(r@rectangle):num { ... }
method area(r@rhombus):num { ... }
method area(s@square):num {
    resend(s@rectangle) }
```

(Like Java’s super)

**Closures**

First-class function objects

Used for:
- standard control structures (if, while, &, |, etc.)
- iterators (do, find, etc.)
- exception handling (fetch, store, etc.)

Syntax
- `& (formals) { zero or more stmts; result expr }, e.g.:
  & (i:int, j:int) { let x := i + j; x*x }
- `& (int, int):int is the type of this closure
- if no formals, can omit &(), e.g.: { print("hi"); }

Examples of use:
- `if (i > j, { i }, { j })
- `[3,4,5].do(& (x:int) { x.print; })
- `table.fetch(key, { error("key is absent") })

Invoke closure by sending `eval` with right number of arguments
- `let cl := & (i:int) { i.print_line; }
- ... eval(cl, 5);`
Non-local returns

Can exit a function/method early via a non-local return from a nested closure

```java
{ ...; ^ result }
{ ...; ^ }
```

Example:

```java
fun find_index(s:string,
    value:char,
    if_absent:&():int
  ):int {
  s.do_associations(&{i:int, v:char}{
    if(v = value, { ^ i });
  });
  eval(if_absent) }

fun find_index(s:string, value:char):int {
  find_index(s, value,
      { error("not found") })
}
```

Parameterization

Can parameterize classes & functions
- functions can be implicitly parameterized using `` notation

Can provide upper bounds for parameter types

```java
abstract class collection[T];
abstract class table[Key <= comparable[Key],
    Value]
    isa collection[Value];
class array[Value] isa table[int,Value];

fun fetch(t:table[Key, Value], k:Key):Value;
fun find_key(
    t:table[Key, Value<=comparable[Value]],
    val:Value,
    if_absent:&():Key) :Key {
  t.do_associations(&{k:Key,v:Value}{
    if(v = val, { ^ k });
  });
  eval(if_absent) }
```

Explicit type parameters must be provided by client
Implicit formal type parameters inferred from argument types
Special types

any
  • type of anything (akin to Object in Java)

void
  • special object & type used for functions that don’t return a useful result

none
  • result type of functions that do not return normally, e.g. error, loop, exit argument closures

dynamic
  • like any, but disables static checking
  • the default type for formals & result, if explicit types omitted

type1 & type2
  • anything that is both a type1 and a type2

type1 | type2
  • anything that is either a type1 or a type2

Modules

Can wrap declarations in a module declaration, for encapsulation and namespace management
  • mark named declarations as public, protected (the default), or private to control access outside the module
  • var fields have two names, with independent access control
  • different modules can declare same names to mean different things

Can reference visible module contents using module$\text{id}

module Shapes {
  public abstract class shape;
  public get protected set
    var field center(s:shape):Points$point;
  public fun area(s:shape):num;
  fun shape_helper(s:shape):num { ... }
}

let s:Shapes$shape := ...;
let a:num := Shapes$area(s);
**Module imports**

Can import a module to give importing scope direct access to imported module’s public names

E.g.:

```plaintext
module Shapes {  
    import Points;  
    public abstract class shape;  
    public get protected set  
        var field center(s:shape):point;  
    public fun area(s:shape):num;  
        fun shape_helper(s:shape):num { ... }  
    }  
import Shapes;  

let s:shape := ...;  
let a:num := area(s);  
```

**Module extensions**

Can declare that one module extends another module, to import other module and gain access to its protected things

```plaintext
module Rectangles;  
    public extends Shapes;  
    public class rectangle isa shape;  
    public field width(r:rectangle):num;  
    public field height(r:rectangle):num;  
    public fun new_rectangle(w:num, h:num ):rectangle {...}  
    fun rect_area(r:rectangle):num { ... }  
    method area(r@rectangle):num { r.rect_area }  
end module Rectangles;  
```
More on modules

Can write any of

```
module Name { ... }
module Name; ... end module Name;
module Name; ... end module;
module Name; ... <end of file>
```

interchangeably

Can declare a module within a module

- nested module declaration specifies its visibility

Can add new declarations to an existing module’s body

externally, e.g. in separate source files!

```
extend module Shapes {
    public fun = (s1:shape, s2:shape):bool {false}
}
extend module Rectangles {
    method =(r1@rectangle, r2@rectangle):bool...
}
```

Programs and files

A Diesel program is a file containing declarations and statements

- declarations visible throughout their scope
- (mutual) recursion without forward declarations or header files
- statements executed in textual order
- no main function necessary
- access command-line arguments using argv object in standard library

To spread programs across multiple files,

use `include` declarations

- an included file can include other files
- by default, Diesel programs implicitly include `prelude.diesel` to get standard library

E.g.

```
include "shapes.diesel";
```
Some standard control structures

if(test, { then });
if(test, { then }, { else }) -- returns a value
if_false(...);

test & { other_test } -- short-circuiting
test | { other_test } -- short-circuiting
not(test)

loop({ ... ^ ... });

while({ test }, { body });
while_false(...);
until({ body }, { test });
until_false(...);

exit(&{break:&():none}
  ... eval(break); ... });
exit_value(&{break:&(resultType):none}
  ... eval(break, result); ... });
loop_exit(...);
loop_exit_value(...);
loop_exit_continue(&{break,continue}{});
loop_exit_value_continue(&{brk,cont}{});

Standard operations for all objects

Printing:
print_string -- return printable string
print, print_line -- print out print_string

Comparing:
==, !== -- compare objects' identities
=, != -- compare comparable objects' values
Some standard classes and objects

bool
  true, false

num
  integer
    int -- limited-precision integers
    big_int -- arbitrary-precision integers
  float
    single_float
    double_float

character
  char -- ascii
  unicode_char

pair, triple, quadruple, quintuple

mb[type] -- type | absent
  absent

file -- unix files

Standard collection classes and functions

abstract collection[T]
  length, is_empty, non_empty
do, includes, find, pick_any
copy

abstract unordered_collection[T]
sets and bags

abstract ordered_collection[T]
linked lists

abstract table[Key,Value]
  hash tables, association lists

abstract indexed[Value]
  isa table[int,Value],
    ordered_collection[Value]
arrays, vectors, strings

abstract sorted_collection[T <= ordered]
binary trees, skiplists
Unordered collections

abstract unordered_collection[T]
    isa collection[T]
    add, add_all
    remove, remove_some, remove_any, remove_all

abstract bag[T] isa unordered_collection[T]

class list_bag[T] isa bag[T]
    new_list_bag[T]

abstract set[T] isa unordered_collection[T]
    union, intersection, difference
    is_disjoint, is_subset

class list_set[T] isa set[T]
    new_list_set[T]
class hash_set[T <= hashable] isa set[T]
    new_hash_set[T]
class bit_set[T] isa set[T]
    new_bit_set[T]

Ordered collections

abstract ordered_collection[T]
    isa collection[T]
    do (over 2-4 ordered collections in parallel)
    add_first, add_last, remove_first/_last
    || (concatenate)
    flatten (for collections of strings)

abstract list[T] isa ordered_collection[T]
    first, rest
    set_first, set_rest

class simple_list[T] isa list[T]
    cons
object nil[T] isa simple_list[T]
    • cannot add in place to simple lists

class m_list[T] isa list[T]
    new_m_list[T]
Keyed tables

abstract table[Key,Value]
  isa collection[Value]
do_associations, includes_key, find_key
fetch, !
store, set_,! , fetch_or_init
remove_key, remove_some_keys, remove_all

class assoc_table[K,V] isa table[K,V]
  new_assoc_table[K,V]

class hash_table[K <= hashable,V]
  isa table[K,V]
  new_hash_table[K,V]

Indexed collections

abstract indexed[T] isa ordered_collection[T],
  table[int,T];
  first, second, ... , fourth, last
set_first, ... , set_last
includes_index, find_index
pos, contains
swap, sort

Fixed length (no add, remove):
class vector[T] isa indexed[T]
class i_vector[T] isa vector[T]
  new_i_vector[T](len, filler)
  new_i_vector_init[T](len, & (i) { value })
  new_i_vector_init_from[T](c, & (c_i)(value))
class m_vector[T] isa vector[T]
  new_m_vector[ _ init [ _ from ]] [ T ] (...)

Extensible:
class array[T] isa indexed[T]
  new_array[T]()
  new_array[_ init [ _ from ]] [ T ] (...)

new_X_init_from is like ML’s map
Strings

abstract string isa indexed[char]
  to_lower_case, to_upper_case
  copy_from
  has_prefix, has_suffix
  remove_prefix, remove_suffix
  pad, pad_left, pad_right
  parse_as_int, parse_as_float
  print

Fixed length:
abstract vstring isa string

class i_vstring isa vstring
  new_i_vstring(len, filler)
  new_i_vstring_init(len, &(i){value})
  new_i_vstring_init_from(c, &(c_i){value})
  • "..." is an i_vstring

class m_vstring[T] isa vstring
  new_m_vstring[_init[_from]](...)