Employee<? extends Staff> e1 = new Employee<Staff>();   
Employee<? super Staff> e2 = new Employee<Staff>();   
e1.set(new Staff()); // error   
Staff p1 = e1.get(); // correct   
e2.set(new Staff()); // correct   
Staff p2 = e2.get(); // error

Functional languages

Control flow by function calls, sometimes concurrent calls  
No looping constructs but plenty of recursion  
Functions applied to arguments – there are no variables  
Referential transparency: a pure function always returns the same result

LISP (LISt Processing language)  
Lists and symbolic computation, not arrays and numeric computation  
Purely functional – supports only atoms and lists

Scheme  
List = ‘ ( a b c ), ‘( a ( a b ) b c ), etc –**quote,** **list** or **‘**  
List treated as function call by default (**func arg1 arg2 …**)  
No guarantee of order of evaluation.  
**Quote** function cancels argument evaluation, e.g. ‘( + 2 3) = ( + 2 3), not (5)  
**Quasiquote** [**`] unquote** [,] **unquote-splicing** [**,@’**] `(1 ,@’ ( + 2 3 )) = ( 1 + 2 3 )

Pair = written with (1 . 2), created with ( cons 1 2 )  
First element is **car**, second is **cdr**, e.g. ( car ( cons 1 2 ) ) => 1, ( cdr ( cons 1 2 ) ) => 2  
A list is a sequence of (nested) pairs – ( 1 2 3 ) is equivalent to ( 1 . ( 2 . ( 3 . () ) ) )  
If **cdr** is not empty, or (), improper list, e.g. ( 1 . ( 2 . ( 3 . 4 ) ) ) => ( 1 2 3 . 4 )  
**car** is a pointer to the value, **cdr** is a pointer to the next element  
**define(func/var arg1 arg2 … )** for new **set!(func/var arg1 arg2 … )** for existing   
**let( (var1 expr1) (var2 expr2) … (varn exprn) body)** - bind exprs to vars in unpredicatable order. Scope is body  
**let\*( (var1 expr1) (var2 expr2) … (varn exprn) body)** - bind exprs to vars in sequential order. Scope is body and each following var/exprs  
**letrec( (var1 expr1) (var2 expr2) … (varn exprn) body)** - bind exprs to vars in unpredictable order. Scope is body and all vars/exprs – exprs know that every var exists, but not its value.  
**cond ( ( op1 expr1a expr1b) ( op2 expr2a expr2b) … ( opn expr1n expr2n) )** – evaluates in order until one is true

Logical programming