Chapter 12 Run-Time Data Structures

Memory Allocation and Deallocation

When executing a program written in the B- language, we need a block of memory for global variables, and a block of memory for each method that has been invoked, but has not completed execution.

In the interpreter:

- A static block of memory is allocated for global variables. This memory exists for the duration of the program's execution.
- A dynamic block of memory (an "activation record") is allocated when a method is invoked, and deallocated when the method is returned from. This memory allocation and deallocation is stack-like, and hence very simple to manage.
- Memory for array and class instance variables is allocated "in-line" as a part of the block of memory for the construct in which the variable is declared (either global memory or the activation record for a method).

  I do it this way, because I am trying to mimic an old-style language like C, that has only very simple memory management (static allocation of globals, and stack-based allocation of activation records). I want the same model when generating compiled code. My Alpha simulator is rather slow when performing sophisticated dynamic memory management and garbage collection, and I want the compiled code to execute quickly.
- String constants and a method table for each class type are also stored as separate blocks of memory. These never change, and exist for the lifetime of the program.

The Run-time Environment

The run-time environment in the B-- interpreter is represented by three pointers into run-time blocks of memory:

- A pointer to the current activation record (memory for the currently executing method).
- A pointer to the current instance ("this"), which is actually a portion of the space allocated for the globals or another method.
- A pointer to the global variables.

The layout for an activation record for a method is:

- (In the compiler's model, but not the interpreter's model of an activation record, there is space for the saving the values of registers).
- The space for local variables, in order.
- The space for the parameters, in order.

The layout for a class instance is:

- A pointer to the method table for the class instance.
- The fields of the class instance, ordered with superclass fields before subclass fields.

The method table is used at run-time to map a method declaration to its possibly overridden code. All instance methods are accessed through the method table for the instance.

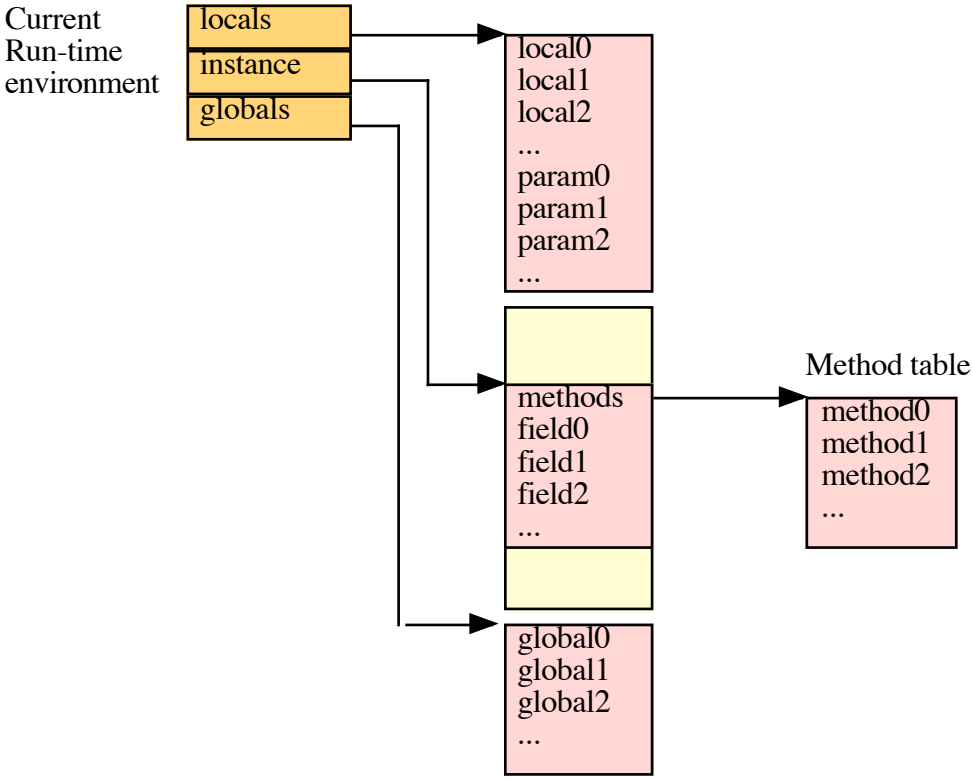
The layout for the method table for a class instance is

- The methods of the class instance, ordered with superclass methods before subclass methods. However, when a method overrides a method in a superclass, it replaces the superclass entry in the method table.

The layout for the global variables is

- The global variables, in order of declaration.

Global methods are not placed in a table, and can be invoked directly.




```
public class RunEnv {

/*
Represents the run time environment, composed of
the globals, current instance, and current invocation.
*/
     private PtrValue globals;
     private PtrValue instance;
     private PtrValue locals;

     public PtrValue globals() { return globals; }
     public PtrValue instance() { return instance; }
     public PtrValue locals() { return locals; }

     public RunEnv(
          PtrValue globals,
          PtrValue instance,
          PtrValue locals ) {
          this.globals = globals;
          this.instance = instance;
          this.locals = locals;
          }

     public RunEnv(
```

```
          PtrValue globals ) {
          this.globals = globals;
          this.instance = null;
          this.locals = null;
          }

     public PtrValue elementAt( Decl decl ) {
          switch ( decl.env().envKind() ) {
               case Env.ENV_GLOBAL:
                    return globals().elementAt( decl.offset().interp() );
               case Env.ENV_CLASS:
                    return instance().elementAt( decl.offset().interp() );
               case Env.ENV_METHOD:
                    return locals().elementAt( decl.offset().interp() );
               default:
                    throw new Error(
                         "Invalid section of "
                         + Decl.sectionText( decl.section() )
                         + " for identifier " + decl.ident() );
               }
          }

     public RunValue getValue( Decl decl ) {
          return elementAt( decl ).getValue();
          }
     }
```

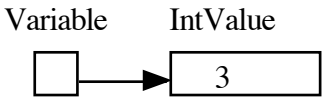
Run-time values

The RunValue class is an abstract class representing a run-time value. It has methods to convert the value to various primitive and compound types, which are overridden in subclasses.

```
public abstract class RunValue {
     public int intValue() {
          throw new Error( "Can't cast runtime " + this + " to int" );
          }

     public char charValue() {
          throw new Error( "Can't cast runtime " + this + " to char" );
          }

     public boolean boolValue() {
          throw new Error( "Can't cast runtime " + this + " to bool" );
          }

     public String stringValue() {
          throw new Error( "Can't cast runtime " + this + " to string" );
          }

     public PtrValue addressValue() {
          throw new Error( "Can't cast runtime " + this + " to reference" );
          }

     public DeclList methodTable() {
          throw new Error( "Can't cast runtime " + this + " to method table" );
          }

     public abstract boolean equals( RunValue otherValue );
     }
```

Primitive values

For example, the IntValue class is used to represent an integer value.



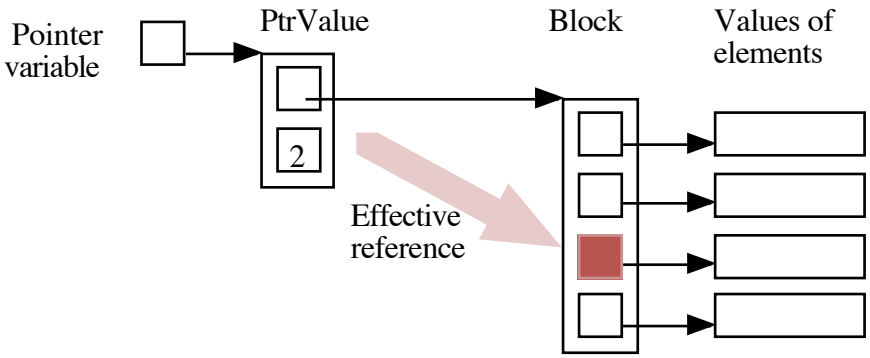

```
public class IntValue extends BasicValue {

     public final static IntValue defaultValue = new IntValue( 0 );

     private int value;

     public IntValue( int value ) {
          this.value = value;
          }

     public int intValue() {
          return value;
          }

     public double floatValue() {
          return ( double ) value;
          }

     public char charValue() {
          return ( char ) value;
          }

     public String toString() {
          return "" + value;
          }

     public String stringValue() {
          return "" + value;
          }

     public boolean equals( RunValue otherValue ) {
          return otherValue.intValue() == value;
          }
     }
```

Pointer Values

The Multiple class is used to represent run-time blocks. It contains a Vector of RunValue objects.

This language supports the notion of pointers to variables.

Achieving this in Java is a little tricky, because we cannot point directly to a variable. How do we get around this? All variables are contained in a Multiple, such as the local block for a method, or the global variables. We can represent the address of a variable by the address of the enclosing Multiple, together with the offset of the variable within the Multiple.



If we were implementing the interpreter in C, we could point directly to the variable.

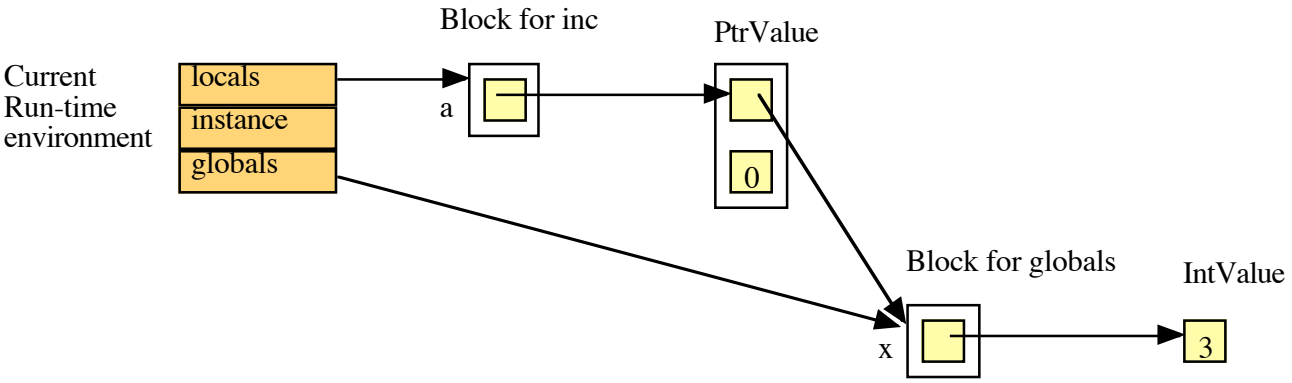
The PtrValue class represents a pointer to an address (for example, the address of a variable).

```
public class PtrValue extends RunValue {

     private Multiple base;
     private int offset;

     public PtrValue( Multiple base, int offset ) {
          this.base = base;
          this.offset = offset;
          }

     public PtrValue( Multiple base ) {
          this( base, 0 );
          }

     public PtrValue( String value ) {
          this( new Multiple( value ) );
          }

     public RunValue getValue() {
          return base.getValue( offset );
          }

     public void setValue( RunValue value ) {
          base.setValue( offset, value );
          }

     public PtrValue elementAt( int index ) {
          return new PtrValue( base, offset + index );
          }
     ...
     }
```

For example, suppose we had

```
void inc( ^int a; )
     begin
          a^++;
     end
int x = 3;
inc( &x );
```

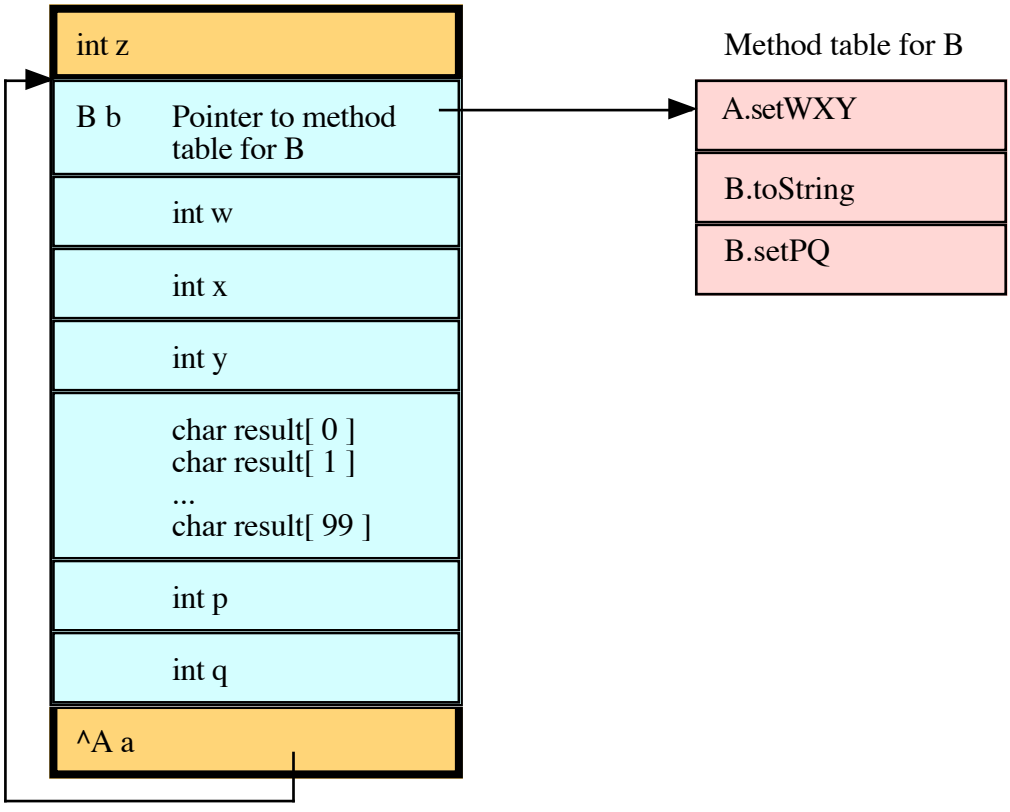
Then we would generate a data structure



In-line storage of arrays and class instances

Arrays and class instances are stored as part of the space allocated for the global variables or the activation record of a function. We do not create new blocks of memory.

For example, consider Program/scope. The global run-time environment looks like



Representation of class instances

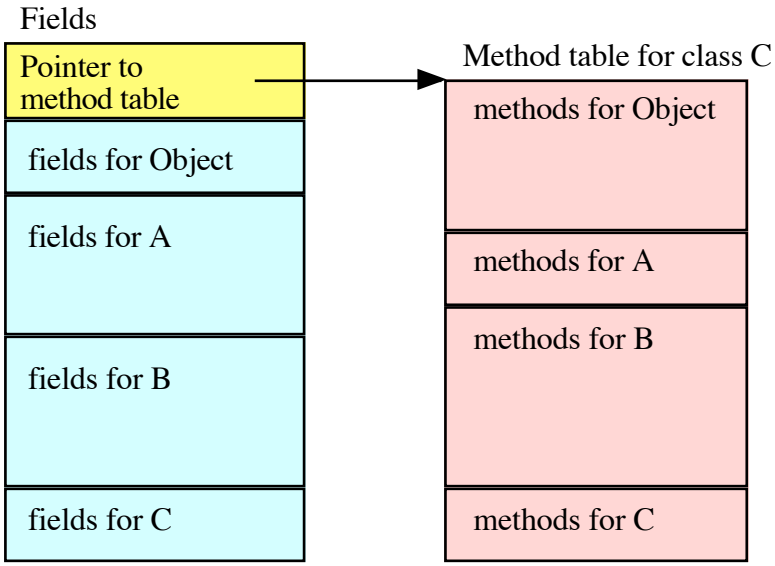
A class instance has an entry at the beginning, which points to the method table for the class. The fields of the class instance follow.

Fields and methods are laid out in order from superclass to subclass. This means that the first portion of the data structure for a subclass has the same layout as the data structure for superclass, and by ignoring the additional data, we can consider an instance of a subclass as being an instance of the superclass.

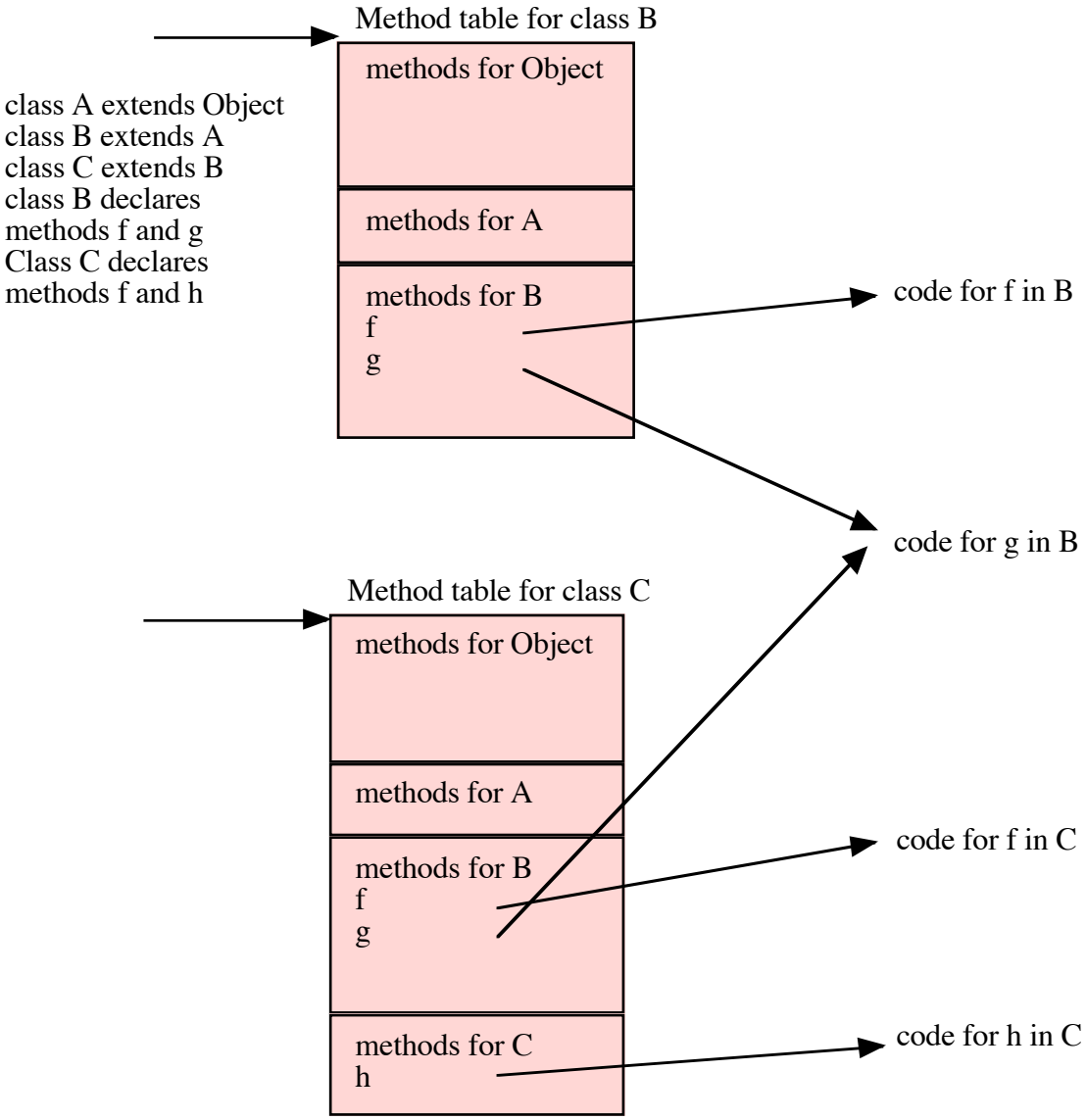
This is what makes it possible to have a variable declared as a pointer to an instance of the superclass actually point to an instance of the subclass.

It is important to realise that the mapping of an invocation to an offset in the method table is done at compile time, and depends only on the declared type of the object and actual parameters. The compiler does not even need to know what subclasses might exist that extend the class the object is declared as.

class A extends Object
class B extends A
class C extends B



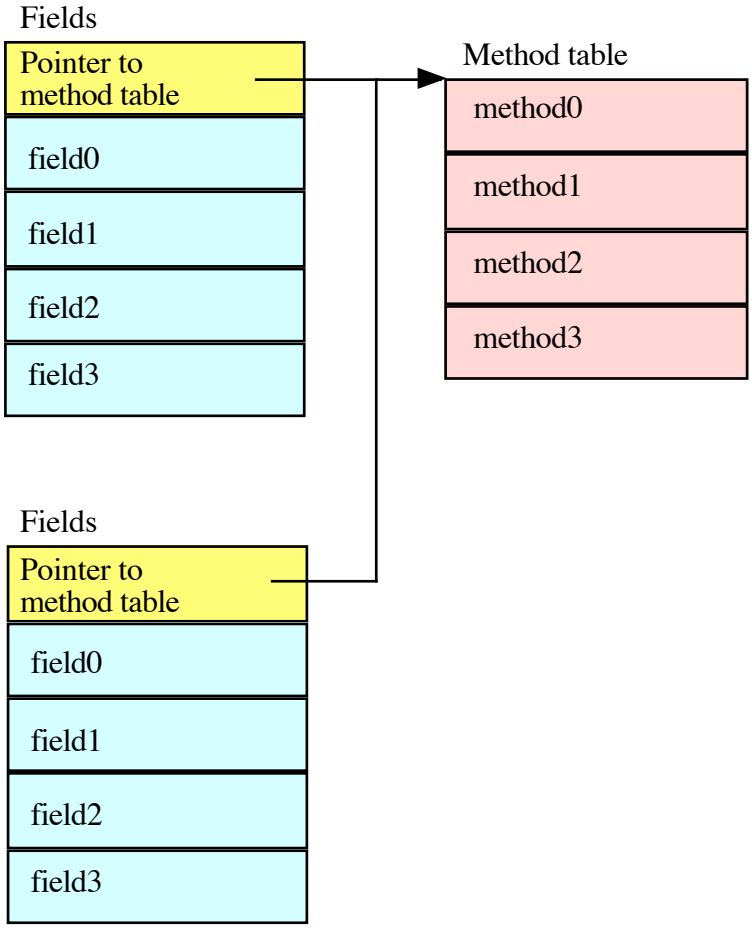
When a method is overridden in the subclass, the entry for the method in the superclass portion of the method table is replaced by a reference to the code for the overridden method.



It is possible to make a variable of type pointer to a superclass point to an instance of a subclass. The portion of the class instance accessed will be the portion for the superclass.

Remember the mapping of invocation to offset is based entirely on the declared types of the object and parameters to the method. but, the method table used is the method table for the actual type of the object, and hence if the method has been overridden, will have been replaced by the method declared in the subclass.

There is only one method table for a class, no matter how many instances of a class exist, and the method table is computed at compile time, and indexed at run time. This makes the implementation of the overriding of methods very efficient.



To implement Java, it is necessary to have some mechanism for referring to the instance of the class Class that represents the type of the object. One way of doing this would be to have an additional field that points to the instance of the class Class. Another way would be to have this pointer as the first entry in the method table. The first way gives faster access time, and the second way takes up less space (because the method table is shared between all instances). The instance of the class Class corresponding to a type can be used to contain not only reflective information about the type, but also the values of static fields.

Calculation of Offsets

The Offset class is used to represent the offset of a variable or method from an appropriate “base pointer”.

B-- is a combined compiler and interpreter, and the size of data is different for each. An Offset instance contains both offsets.

```
public class Offset {

     private static int roundup( int value, int align ) {
          return ( value + align - 1 ) / align * align;
          }

     private int interp = 0;
     private int compile = 0;
     public int interp() { return interp; }
```

```
    public int compile() { return compile; }

    public Offset( int interp, int compile ) {
        this.interp = interp;
        this.compile = compile;
        }

    public Offset() {
        this( 0, 0 );
        }

    public void set( int interp, int compile ) {
        this.interp = interp;
        this.compile = compile;
        }

    public void set( Offset offset ) {
        this.interp = offset.interp;
        this.compile = offset.compile;
        }

    public Offset copy() {
        return new Offset( interp, compile );
        }

    public Offset inc( Offset size ) {
        Offset value = copy();
        this.interp += size.interp();
        this.compile += roundup( size.compile(), Type.WORD );
        return value;
        }

    public void align() {
        compile = roundup( compile, Type.WORD );
        }
    ...
    }
```

When the offsets for parameters and local variables within methods are computed, they have to take into account the run-time layout of an activation record. To support method invocations with additional parameters, and the fact that only the invoked method knows the amount of space required for the locals, not the invoker, an activation record is laid out with the local variables first, followed by the parameters. (The offset for the compiler excludes the space for saving the values of registers.)

```
public class MethodDeclNode extends DeclStmtNode implements DeclNode {
    ...
    public void genOffset() {
        decl.genOffset();
        localEnv.varOffset().set( 0, 0 );
        body.genOffset();
        localEnv.paramStart().set( localEnv.varOffset() );
        formalParams.genOffset();
        }
    ...
    }
```

The offsets for instance fields and methods have to take into account the layout of class instances. The field table for a class instance starts with a pointer to the method table, followed by the fields, with superclass fields before subclass fields. The method table for a class instance has superclass

methods before subclass methods, but a method that overrides a method in the subclass is placed in the offset corresponding to the overridden method.

```
public class Env {
     ...
     public void setClassOffset() {
          if ( extendedEnv == null ) {
               varOffset.set( 1, Type.WORD );
               methodOffset().set( 0, 0 );
               }
          else {
               varOffset.set( extendedEnv.varOffset() );
               methodOffset.set( extendedEnv.methodOffset() );
               }
          }

     public void createMethodTable( DeclList methodTable ) {
          if ( extendedEnv != null )
               extendedEnv.createMethodTable( methodTable );
          for ( int i = 0; i < localDecls.size(); i++ ) {
               Decl decl = localDecls.elementAt( i );
               switch ( decl.section() ) {
                    case Decl.DECL_METHOD:
                         methodTable.setElementAt( decl, decl.offset().interp() );
                         break;
                    }
               }
          }

     public void createMethodTable() {
          createMethodTable( methodTable );
          }
     ...
     }

public class ClassDeclNode extends DeclStmtNode implements DeclNode {
     ...
     public void genOffset() {
          decl.genOffset();
          localEnv.setClassOffset();
          memberDeclList.genOffset();
          localEnv.createMethodTable();
          }
          ...
     }
```

If a method is overridden in the subclass, it is given the same offset in the method table as the method in the superclass it overrides. This is actually one of the few places in the compiler that deals with overriding. The fact that methods are accessed indirectly via a method table, and this magic piece of code are what make overriding work.

Note that there is no overriding of fields.

```
public class Decl {
     ...
     public void genOffset() {
          switch ( section ) {
               case DECL_METHOD:
                    if ( env.envKind() == Env.ENV_CLASS ) {
                         Env extendedEnv = env.extendedEnv();
                         if ( extendedEnv != null )
                              overRiddenDecl =
                                   extendedEnv.searchExtended( ident );
```

```
                    if ( overRiddenDecl != null ) {
                         if ( overRiddenDecl.section() != DECL_METHOD )
                              throw new Error(
                                   "Overridden declaration " + overRiddenDecl
                                   + " must be method" );
                         MethodType methodType = ( MethodType ) type;
                         MethodType overRiddenMethodType =
                              ( MethodType ) overRiddenDecl.type();
                         if ( ! methodType.equals( overRiddenMethodType ) )
                              throw new Error(
                                   "Overridden declaration " + overRiddenDecl
                                   + " must have same type" );
                         offset = overRiddenDecl.offset();
                         }
                    else {
                         offset = env.methodOffset().inc( type.size() );
                         }
                    env.methodOffset().align();
                    }
               else
                    offset = new Offset( 0, 0 );
               break;
          case DECL_NONE:
          case DECL_CLASS:
               offset = new Offset( 0, 0 );
               break;
          case DECL_VAR:
          case DECL_PARAM:
               offset = env.varOffset().inc( type.size() );
               env.varOffset().align();
               break;
          default:
               throw new Error(
                    "Invalid section of " + sectionText( section() ) );
          }
     Print.error().debugln( "Offset " + ident + " = " + offset );
     }
...
}
```