10-0: **Classes**

- simpleJava classes are equivalent to C structs

```
class myclass {          struct mystruct {
   int x;                   int x;
   int y;                   int y;
   boolean z;               int z;
}                        }
```

- What do we need to add to simpleJava classes to make them true objects?

10-1: **Adding Methods to Classes**

- To extend simpleJava to be a true object oriented language, classes will need methods.
- New definition of classes:

```
class <classname> {
  <List of instance variable declarations,
   method prototypes, &
   method definitions>
}
```

- As in regular Java, instance variable declarations & method definitions can be interleaved.

10-2: **Adding Methods to Classes**

```
class Point {
    int xpos;
    int ypos;
    Point(int x, int y) {
       xpos = x;
       ypos = y;
    }
    int getX() {
       return xpos;
    }
    int getY() {
       return ypos;
    }
    void setX(int x) {
       xpos = x;
    }
   void setY(int y) {
       ypos = y;
   }
}
```

10-3: **Adding Methods to Classes**

```
class Rectangle {

 Point lowerleftpt;
 Point upperrightpt;

 Rectangle(Point lowerleft, Point upperright) {
   lowerleftpt = lowerleft;
   upperrightpt = upperright;
 }
 int length() {
   return upperrightpt.getX() - lowerleftpt.getX();
 }
 int height() {
   return upperrightpt.getY() - lowerleftpt.getY();
 }
 int area() {
   return length() * height();
 }
}
```

10-4: **"This" local variable**

- All methods have an implicit "this" local variable, a pointer to the data block of the class
- Original version:

```
Point(int x, int y) {
 xpos = x;
 ypos = y;
}
```

- Alternate version:

```
Point(int x, int y) {
 this.xpos = x;
 this.ypos = y;
}
```

10-5: **Compiler Changes for Methods**

- Modify Abstract Syntax Tree
- Modify Semantic Analyzer
  - Classes will need function environments
  - "This" pointer defined for methods
- Modify Abstract Assembly Generator
  - Maintain the "this" pointer

10-6: **Modifications to AST**

- What changes are necessary to the AST to add methods to classes?
  - Which tree nodes need to be changed?
  - How should they be modified?

10-7: **Modifications to AST**

- ASTClass: contain prototypes and method definitions as well as instance variable declarations
  - Add abstract class ASTClassElem
    - ASTMethod, ASTMethodPrototype, ASTInstanceVarDef would be subclasses
- What would .jj file look like?

10-8: **Changes to .jj file**

```
void classdef() :
{}
{
<CLASSS> <IDENTIFIER>
   <LBRACE> classElems() <RBRACE>
}

void classElems() :
{}
{
  (classElem())*
}

void  classElem() :
{}
{
   <IDENTIFIER> <IDENTIFIER>
     (((<LBRACK> <RBRACK>)* <SEMICOLON>) |
       (<LPAREN> formals() <RPAREN>) (<SEMICOLON> | <LBRACE> statements() <RBRACE>))
}
```

10-9: **Modifications to AST**

- Change AST to allow method calls

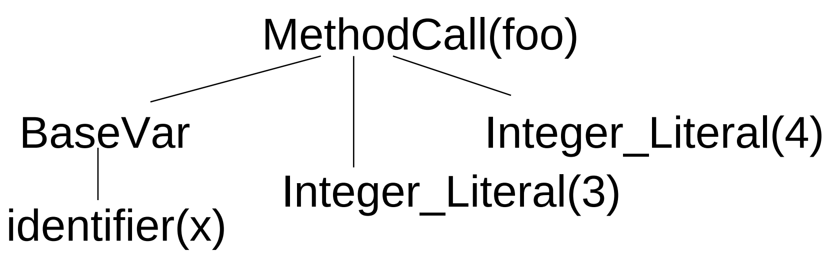
```
x.y.foo()
z[3].y[4].bar(x.foo())
```

10-10: **Modifications to AST**

- x.foo(3,4)

10-11: **Modifications to AST**

- x.foo(3,4)

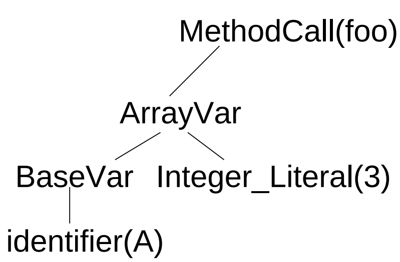


10-12: **Modifications to AST**

- A[3].foo()

10-13: **Modifications to AST**

- A[3].foo()

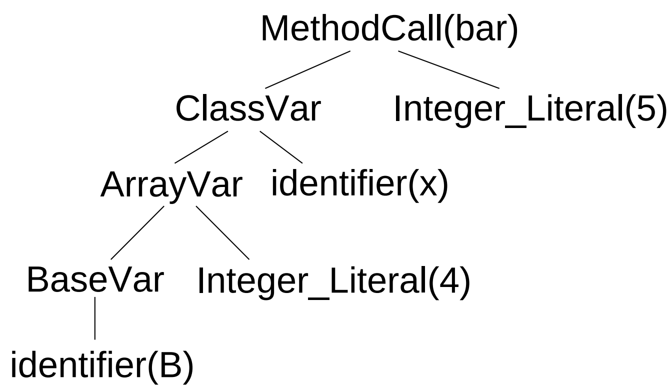


10-14: **Modifications to AST**

- B[4].x.foo(5)

10-15: **Modifications to AST**

- B[4].x.foo(5)



10-16: **Modifications to AST**

- Constructors have slightly different syntax than other functions

```
class Integer {

    int data;

    Integer(int value) {
        data = value;
    }

    int intValue() {
        return data;
    }
}
```

10-17: **Modifications to AST**

- For the constructor

```
    Integer(int value) {
        data = value;
    }
```

- Create the abstract syntax

```
    Integer Integer(int value) {
        data = value;
        return this;
    }
```

10-18: **Modifications to AST**

```
Integer(int value) {
    data = value;
}
```

- Create the abstract syntax

```
Integer Integer(int value) {
    data = value;
    return this;
}
```

- When will this not work?

10-19: **Modifications to AST**

- Constructors
  - AST needs to be modified to allow constructors to take input parameters

10-20: **Changes in Semantic Analysis**

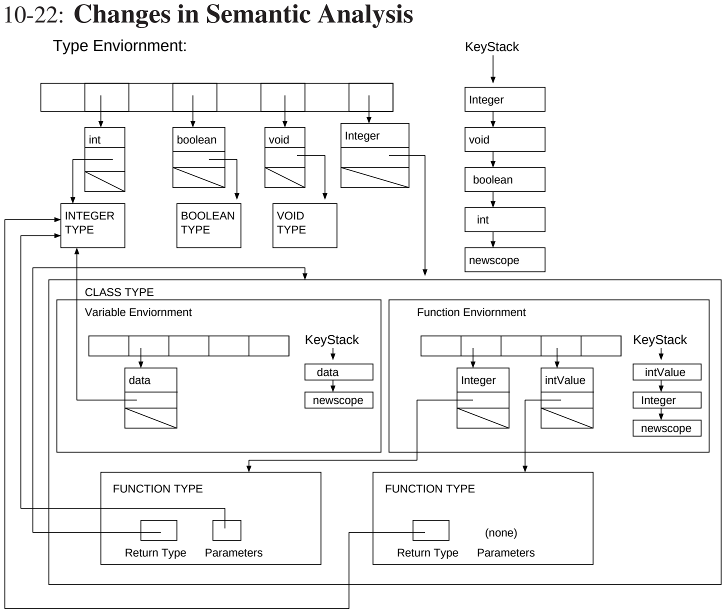
- Without methods, the internal representation of a class contains a only a variable enviornment
- How should this change if we add methods to classes?

10-21: **Changes in Semantic Analysis**

- Add Function Environment to to the internal representation of class types

```
class Integer {

    int data;

    Integer(int initvalue) {
        data = initvalue;
    }

    int intValue() {
        return data;
    }
}
```

10-22: **Changes in Semantic Analysis**



10-23: **Changes in Semantic Analysis**

- What do you do now to analyze a class?

10-24: **Changes in Semantic Analysis**

- What do you do now to analyze a class?
  - Create a new variable environment
  - Analyze (visit) the instance variable declarations
    - Adding each variable to the new variable enviornment
  - Create a new internal representation of a Class, using the new variable environment
  - Add Class to the class environment

10-25: **Changes in Semantic Analysis**

- To analyze a Class Defintion
  - Create a new, local function & variable environment
  - Build a new internal representation of a Class, using newly created (local) function & variable environments
  - Add the class to the type environment
  - Begin a new scope in the global variable environment & function environments
  - Add "this" variable to the variable environment, using this class type

10-26: **Changes in Semantic Analysis**

- To analyze a Class Defintion
  - Analyze variables (adding to global variable environment *and* the local variable environment)
  - Analyze function defintions (adding to global function environment *and* the local function environment)
  - End scopes in global environments

10-27: **Changes in Semantic Analysis**

- To analyze a method call `x.foo()`
  - Analyze variable `x`, which should return a class type
  - Look up the function `foo` in the function environment of the variable x
  - Return the type of foo()

10-28: **SA: Methods & Instance Vars**

```
class MethodExample {

   int instanceVar;

   MethodExample() {}

   int foo(int x) {
        return x + instanceVar;
```

```
    }

    int bar(int x) {
        return foo(x);
    }
}
```

10-29: **SA: Methods & Instance Vars**

- What extra work do we need to do to allow instaceVar to be seen in foo?

10-30: **SA: Methods & Instance Vars**

- What extra work do we need to do to allow instaceVar to be seen in foo?
  - None! InstanceVar is already in the global variable environment.
  - (We will need to do a little extra work to generate abstract assembly – why?)

10-31: **SA: Methods from Methods**

- What extra work do we need to do to allow bar to call the function foo?

10-32: **SA: Methods from Methods**

- What extra work do we need to do to allow bar to call the function foo?
  - None!
  - When we analyzed foo, we added the proper prototype to both the global function environment and the local function environment

10-33: **SA: Constructors**

- `new MyClass(3,4)`
  - Look up "MyClass" in the type environment, to get the definition of the class
  - Look up "MyClass" in the function environment for the class
  - Check to see that the number & types of parameters match
  - Return the type of MyClass

10-34: **SA: Example**

```
class SimpleClass {

  int x;
  int y;

  SimpleClass(int initialx, initialy) {
    x = initialx;
    y = initialy;
  }

  int average() {
    int ave;

    ave = (x + y) / 2;
    return ave;
  }
}
```

```
void main {
  SimpleClass z;
  int w;

  z = new SimpleClass(3,4);
  w = z.average();
}
```

10-35: **SA – Example**

- To analyze class `SimpleClass`
  - Create a new empty variable & function environment
  - Create a new class type that contains these environments
  - Begin a new scope in the global function & variable environments
  - Add "this" to the global variable environment, with type SimpleClass
  - Add `x` and `y` to *both* the local and global variable environment
  - Add the prototype for the constructor to the local and global environments

10-36: **SA – Example**

- To analyze class `SimpleClass` (continued)
  - Analyze the body of SimpleClass
  - Add prototype for average to both the local and global function environment
  - Analyze the body of average
  - End scope in global function & variable environments

10-37: **SA – Example**

- To analyze the body of `SimpleClass`
  - Begin a new scope in the global variable environment
  - Add `initialx` and `intialy` to the global variable environment (both with type INTEGER)
  - Analyze statement `x = initialx` using global environments
  - Analyze statement `y = initialy` using global environments
  - Analyze statement `return this;` using global environments
    - Added implicitly by the parser!
  - End scope in the global variable environment

10-38: **SA – Example**

- To analyze the body of `average`
  - Begin a new scope in the global variable environment
  - Add `ave` to the global variable environment with type INTEGER
  - Analyze the statement `ave = (x + y) / 2` using global environments
  - Analyze the statement `return ave` using global environments
  - End scope in local variable environment

10-39: **SA – Example**

- To analyze the body of main
  - Begin a new scope in the variable environment
  - Add `z` to the variable environment, with the type `SimpleClass`
  - Analyze the statement
    `z = new SimpleClass(3,4);`

10-40: **SA – Example**

- To analyze the body of main (continued)
    - `z = new SimpleClass(3,4);`
        - Look up SimpleClass in the type environment. Extract the function environment for SimpleClass
        - Look up SimpleClass in this function environment
        - Make sure the prototype for SimpleClass takes 2 integers
        - Look up the type of `z` in the global variable environment
        - Make sure the types match for the assignment statement

10-41: **SA – Example**

- To analyze the body of main (continued)
    - Analyze the statement `w = z.average();`
        - Look up `z` in the variable environment, and make sure that it is of type CLASS.
        - Using the function environment obtained from the CLASS type for `z`, look up the key `average`. Make sure that the function `average` takes zero input parameters.
        - Make sure the return type of `average` matches the type of `w`.
    - End scope in the variable environment

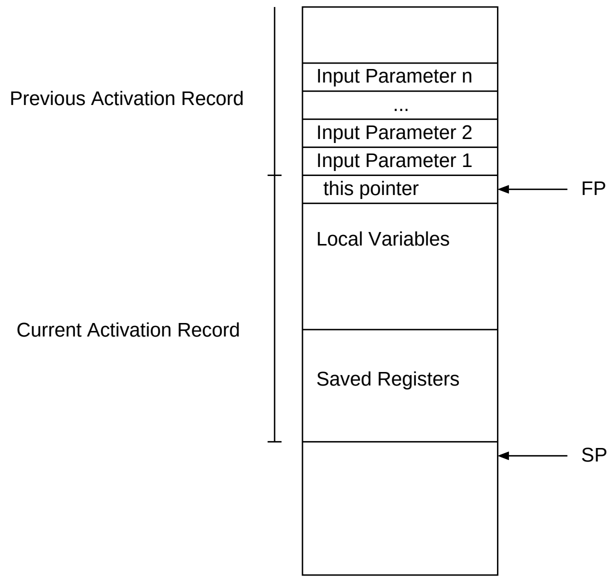
10-42: **Changes Required in AAT**

- We will aslo need to make some changes in the AAT generator
    - Maintain "this" pointer
        - Set the value of the "this" pointer at the beginning of a method call
    - Access instance variables using the "this" pointer.
        - `x = 3;` produces different code if `x` is an instance variable or a local variable.

10-43: **Activation Record for Methods**

- Activation records for methods will contain a "this" pointer
- "this" pointer will be the first item in the activation record
- Remainder of the activation record does not change
- "This" pointer is passed in as implicit 0th parameter

10-44: **Activation Record for Methods**



10-45: **Activation Record for Methods**

- To set up an activation record (at the beginning of a method call
  - Save registers, as normal
  - Set the FP to (SP + WORDSIZE)
    - So that the "this" pointer ends up in the correct activation record
    - Passed as the 0th parameter
  - "this" is at location FP
  - First local variable is at location FP-WORDSIZE
  - First input parameter is at location FP+WORDSIZE

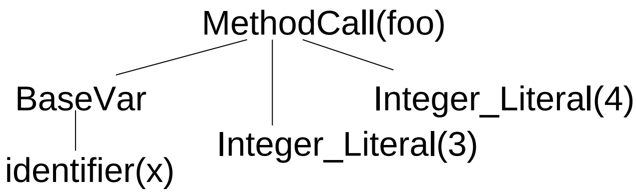
10-46: **AATs for Method Calls**

- Passing implicit "this" parameter
  - Each method call needs to be modified to pass in the implicit "this" parameter.
- Need to handle two types of method calls
  - Explicit method calls
    - `x.foo(3,4)`
  - Implicit Method Calls
    - Class contains methods `foo` and `bar`
    - `foo` calls `bar` (without using "this")

10-47: **Explicit Method Calls**

- `x.foo(3,4)`

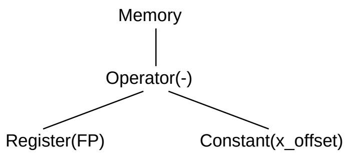
- AST:



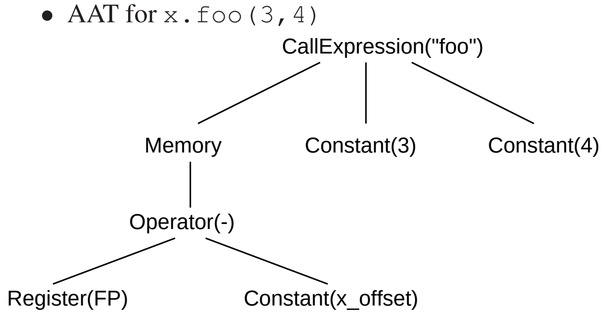
- What should the Abstract Assembly be for this method call?
- (What should we pass as the "this" pointer?)

10-48: **Explicit Method Calls**

- AAT for x:



- AAT for `x.foo(3,4)`



10-49: **Implicit Method Calls**

```
class MyClass {

   int foo(int y) {
      return y + 1;
   }

   void bar() {
      int x;
      x = foo(7);
   }
}

int myfunction(int a) {
   return a + 1;
}
```

```
void main() {
    int x;
    x = myfunction(3);
 }
```

10-50: **Implicit Method Calls**

- `x = myfunction()` in `main` is a function call – don't need to pass in a "this" pointer
- `x = foo(7)` in `bar` is a method call – need to pass in a "this" pointer
- Add another field to FunctionEntry: Method bit
  - false if entry is a function (no need for "this" pointer)
  - true if entry is a method (need 0th parameter for "this" pointer)

10-51: **Implicit Method Calls**

```
class MethodCalling {

   int foo(int y) {
      return y + 1;
   }

   void bar() {
      int x;
      x = foo(7);
   }
}
```

10-52: **Implicit Method Calls**

- We know `foo` is a method call
  - method bit set to true in function entry for `foo`
- Need to pass in the “this” pointer as 0th parameter
- How can we calculate the “this” pointer to pass in?

10-53: **Implicit Method Calls**

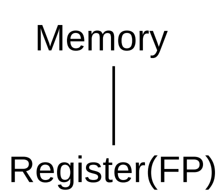
- We know `foo` is a method call
  - method bit set to true in function entry for `foo`
- Need to pass in the “this” pointer as 0th parameter
- How can we calculate the “this” pointer to pass in?
  - Same as the “this” pointer of the current function

10-54: **Implicit Method Calls**

- Any time a method is called implicitly, the “this” pointer to send in is:

10-55: **Implicit Method Calls**

- Any time a method is called implicitly, the “this” pointer to send in is:

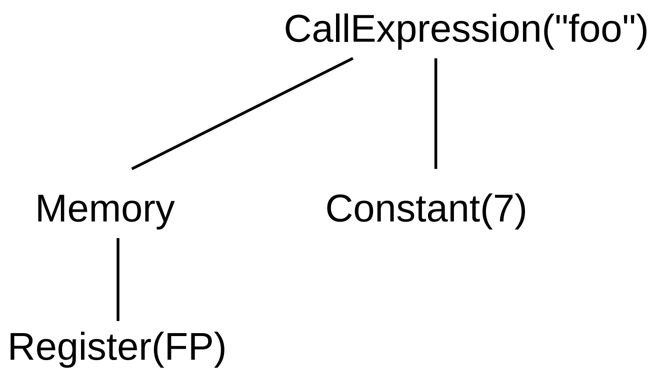


10-56: **Implicit Method Calls**

- Abstract Assembly for `foo(7)`

10-57: **Implicit Method Calls**

- Abstract Assembly for `foo(7)`



10-58: **Constructor Calls**

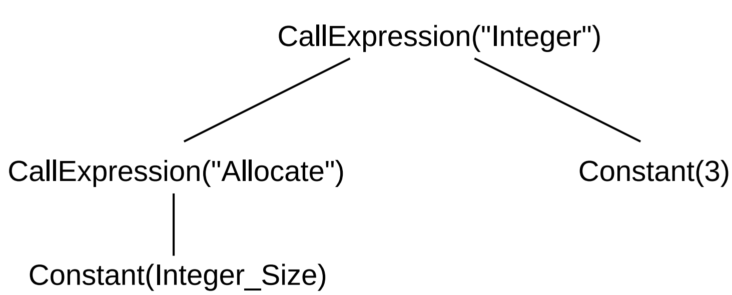
- Just like any other method call
- **But...** we need an initial "this" pointer
- No space has been allocated yet!

10-59: **Constructor Calls**

- The AAT for a constructor call needs to:
  - Allocate the necessary space for the object
  - Call the constructor method, passing in the appropriate "this" pointer
- What should the AAT for `new Integer(3)` be?

10-60: **Constructor Calls**

- The AAT for a constructor call needs to:
  - Allocate the necessary space for the object
  - Call the constructor method, passing in the appropriate "this" pointer
- What should the AAT for `new Integer(3)` be?



10-61: **AATs for Instance Variables**

```
class InstanceVsLocal {
  int instance;

  void method() {
     int local;

     local = 3;       /* line A */
     instance = 4;    /* line B */
  }
```

- Stack / heap contents during `method`?
- AAT for line A?
- AAT for line B?

10-62: **Instance vs. Local Variables**

- Instance variables and local variables are implemented differently.
- Need to know which variables are local, and which variables are instance variables (just like methods)
- Add instanceVar bit to VariableEntry
  - true for is instance variable
  - false for local variables / parameters

10-63: **Instance Variable Offsets**

- Keep track of two offsets (using two globals)
  - Local variable offset
  - Instance variable offset
- At the beginning of a class definition:
  - Set instance variable offset to 0
  - Insert "this" pointer into the variable environment, as a *local variable*
- At the beginning of each method
  - set the local variable offset to -WORDSIZE
  - Remember the "this" pointer!

10-64: **Instance Variable Offsets**

- When an instance variable declaration is visited:
  - Add variable to local & global variable environments, using the instance variable offset, with instance bit set to true
  - Decrement instance variable offset by WORDSIZE
- When a local variable declaration is visited:
  - Add variable to only the global variable environment, using the local variable offset, with instance bit set to false

- Decrement local variable offset by WORDSIZE

10-65: **AATs for Instance Variables**

- For a base variable:
  - If it is a local variable, proceed as before
  - If it is an instance variable
    - Add an extra "Memory" node to the top of the tree
    - Need to do nothing else!

10-66: **AATs for Instance Variables**

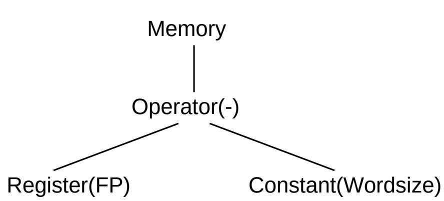
```
class InstanceVsLocal {
    int instance;

    void method() {
       int local;

       local = 3;
       instance = 4;
    }

}
```

10-67: **AATs for Instance Variables**

- Insert `instance` to the global variable environment, with the "instance variable" bit set to 1, with offset 0
- Insert `local` to the global variable environment, with the "instance variable" bit set to 0, with offset WORDSIZE (remember "this" pointer!)
- Abstract Assembly for `local`:

10-68: **AATs for Instance Variables**

- Insert `instance` to the global variable environment, with the "instance variable" bit set to 1, with offset 0
- Insert `local` to the global variable environment, with the "instance variable" bit set to 0, with offset WORDSIZE (remember "this" pointer!)
- Abstract Assembly for `local`:



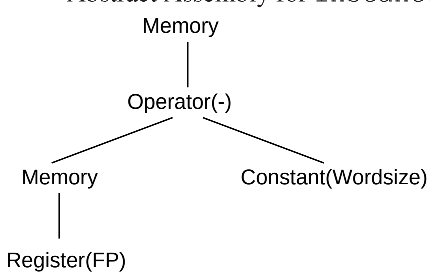
10-69: **AATs for Instance Variables**

- Insert `instance` to the global variable environment, with the "instance variable" bit set to 1, with offset 0

- Insert `local` to the global variable environment, with the "instance variable" bit set to 0, with offset WORDSIZE (remember "this" pointer!)
- Abstract Assembly for `instance`

10-70: **AATs for Instance Variables**

- Insert `instance` to the global variable environment, with the "instance variable" bit set to 1, with offset 0
- Insert `local` to the global variable environment, with the "instance variable" bit set to 0, with offset WORDSIZE (remember "this" pointer!)
- Abstract Assembly for `instance`



10-71: **Instance vs. Local Variables**

```
class MyClass {

 int instance1;
 int instance2;

 void method(int param) {
   int local;

   local = instance1 - instance2 + param;
 }
}
```

- Stack / heap contents during `method`?
- AAT for assignment statement?

10-72: **AATs for Instance Variables**

- What about class variables and array variables?

```
class Class1 {
   int x;
   int y[];
}
class Class2 {

  Class1 C1;
  int array[];
  Class1 C2[];
  void method() {
      array[2] = 3;
      C1.x = 3;
      C1.y[2] = 4;
      C2[3].y[4] = 5;
  }
}
```

```
void main() {
    Class2 C2 = new Class2();
    C2.C1 = new Class1();
    C2.C1.y = new int[10];
    C2.array = new int[10];
    C2.C2 = new Class1[5];
    C2.C2[3] = new Class1();
    C2.C2[3].y = new int[5];

    C2.method();
}
```

10-73: **Code Generation**

- When methods are added to classes, what changes are necessary in the code generation stage?

10-74: **Code Generation**

- When methods are added to classes, what changes are necessary in the code generation stage?
- None!
  - The AAT structure is not changed
  - Prior modifications create legal AAT
  - Code Generator should work unchanged.

10-75: **Inheritance**

```
class Point {
   int xpos;
   int ypos;

   Point(int x, int y) {
      xpos = x;
      ypos = y;
   }

   int getX() {                void setX(int x) {
      return xpos;               xpos = x;
   }                           }

   int getY() {                void setY(int y) {
      return ypos;               ypos = y
   }                           }

}
```

10-76: **Inheritance**

```
class Circle extends Point {
    int radiusval;

    Circle(int x, int y, int radius) {
       xpos = x;
       ypos = y;
       radiusval = radius;
    }

   int getRadius() {
       return radiusval;
   }

   void setRadius(int radius) {
      radiusval = radius;
   }
}
```

10-77: **Inheritance**

- What changes are necessary to the lexical analyzer to add inheritance?

10-78: **Inheritance**

- What changes are necessary to the lexical analyzer to add inheritance?
  - Add keyword “extends”
  - No other changes necessary

10-79: **Inheritance**

- What changes are necessary to the Abstract Syntax Tree for adding inheritance?

10-80: **Inheritance**

- What changes are necessary to the Abstract Syntax Tree for adding inheritance?
  - Add a “subclass-of” field to the class definition node
  - “subclass-of” is a String
    - Examples for point, circle

10-81: **Inheritance**

- What changes are necessary to the Semantic Analyzer for adding inheritance?

10-82: **Inheritance**

- What changes are necessary to the Semantic Analyzer for adding inheritance?
  - Allow subclass access to all methods & instance variables of superclass
  - Allow assignment of a subclass value to a superclass variable

10-83: **Inheritance**

- What changes are necessary to the Semantic Analyzer for adding inheritance?
  - Add everything in the environment of superclass to the environment of the subclass
  - Add a “subclass-of” pointer to internal representation of types
  - On assignment, if types are different, follow the “subclass-of” pointer of RHS until types are equal, or run out of superclasses.

10-84: **Environment Management**

- Case 1

```
class baseclass {
    int a;
    boolean b;
}
class subclass extends baseclass {
   boolean c;
   int d;
}
```

- `baseclass` contains 2 instance variables (`a` and `b`)
- `subclass` contains 4 instance variables (`a, b, c` and `d`

10-85: **Environment Management**

- Case 2

```
class baseclass2 {
    int a;
    boolean b;
}
class subclass2 extends baseclass2 {
   int b;
   boolean c;
}
```

- baseclass2 contains `a, b`
- subclass2 contains 4 instance variables, only 3 are accessible `a, b (int), c`

10-86: **Environment Management**

- Case 2

```
class baseclass2 {
    int a;
    boolean b;
}
class subclass2 extends baseclass2 {
   int b;
   boolean c;
}
```

- subclass2 contains 4 instance variables, only 3 are accessible `a, b (int), c`
- How could we get at the boolean value of b?

10-87: **Environment Management**

- Case 3

```
class baseclass3 {
   int foo() {
      return 2;
   }
   int bar() {
      return 3;
   }
}
class subclass3 extends baseclass3 {
   int foo() {
      return 4;
  }
}
```

10-88: **Environment Management**

- When subclass A extends a base class B
    - Make clones of the variable & function environments of B
    - Start A with the clones
    - Add variable and function definitions as normal

10-89: **Environment Management**

- To analyze a class A which extends class B
    - begin scope in the global variable and function environments
    - Look up the definition of B in the type environment
    - Set superclass pointer of A to be B
    - Add all instance variables in B to variable environment for B, and the global variable environment

  - Add all function definitions in B to the function environment for A and the global function environment

10-90: **Environment Management**

- To analyze a class A which extends class B (continued)
  - Add "this" pointer to the variable environment of A
    - Overriding the old "this" pointer, which was of type B
  - Analyze the definition of A, as before
  - End scope in global function & variable environments

10-91: **Assignment Statements**

- To analyze an assignment statement
  - Analyze LHS and RHS recursively
  - If types are not equal
    - If RHS is a class variable, follow the superclass pointer of RHS until either LHS = RHS, or reach a null superclass
- Use a similar method for input parameters to function calls

10-92: **Abstract Assembly**

- What changes are necessary in the abstract assembly generator for adding inheritance?

10-93: **Abstract Assembly**

- What changes are necessary in the abstract assembly generator for adding inheritance?
  - At the beginning of a class definition, set the instance variable offset = size of instance variables in superclass, instead of 0
    - When instance variables are added to subclass, use the same offsets that they had in superclass.
  - No other changes are necessary!

10-94: **Code Generation**

- What changes are necessary in the code generator for adding inheritance?

10-95: **Code Generation**

- What changes are necessary in the code generator for adding inheritance?
- None – generate standard Abstract Assembly Tree

10-96: **Inheritance**

- Adding inheritance without virtual functions can lead to some odd behavior

10-97: **Inheritance**

```
class base {
  int foo() {
    return 3;
  }
}

class sub extends base {
  int foo() {
    return 4;
  }
}
```

```
void main() {
  base A = new base();
  base B = new sub();
  sub C = new sub();

  print(A.foo());
  print(B.foo());
  print(C.foo());
}
```

10-98: **Inheritance**

- Adding inheritance without virtual functions can lead to some odd behavior
  - Hard-to-find bugs in C++
  - Why java does uses virtual functions
  - Non-virtual (static, final) cannot be overridden

10-99: **Access Control**

```
class super {
  int x;
  public int y;
  private int z;

  void foo() {
      x = 1;
      z = 2;
  }
}

class sub extends super {
  private int a;

   void bar() {
      z = 3;
      a = 4;
   }
}
```

```
void main () {
   super superclass;
   sub subclass;
   superclass = new super();
   subclass = new sub();
   superclass.x = 5;
   superclass.z = 6;
   subclass.y = 7;
   subclass.a = 8;
}
```

10-100: **Access Control**

```
class super {
  int x;
  public int y;
  private int z;

  void foo() {
      x = 1;  /* Legal */
      z = 2;  /* Legal */
  }
}

class sub extends super {
  private int a;

   void bar() {
      z = 3; /* Illegal */
      a = 4; /* Legal */
   }
}
```

```
void main () {
   super superclass;
   sub subclass;
   superclass = new super();
   subclass = new sub();
   superclass.x = 5;  /* Legal */
   superclass.z = 6;  /* Illegal */
   subclass.y = 7;    /* Legal */
   subclass.a = 8;    /* Illegal */
}
```

10-101: **Access Control**

- Changes required in Lexical Analyzer

10-102: **Access Control**

- Changes required in Lexical Analyzer
  - Add keywords “public” and “private”

10-103: **Access Control**

- Changes required in Abstract Syntax Tree

10-104: **Access Control**

- Changes required in Abstract Syntax Tree
  - Add extra bit to methods and instance variables – public or private

10-105: **Access Control**

- Changes required in Semantic Analyzer

10-106: **Access Control**

- Changes required in Semantic Analyzer
  - Allow access to a variable within a class
  - Deny Access to variable outside of class
- How can we do this?

10-107: **Access Control**

- Changes required in Semantic Analyzer
  - Allow access to a variable within a class
  - Deny Access to variable outside of class
- Use the global variable environent to access variables inside class
- Use the local variable environment to access variables outside class

(examples) 10-108: **Access Control**

- When analyzing a public instance variable declaration
  - `public int y;`
  - Add `y` to both the local and global variable environment
- When analyzing a private instance variable declaration
  - `private int z;`
  - Add `z` to *only* the global variable environment

10-109: **Access Control**

- If we add `z` to only the global variable environment
  - When we access `z` from within the class, it will be found
  - When we access `z` from outside the class, it will *not* be found
  - Need to add a hack for getting this.x to work correctly ...

10-110: **Access Control**

- Changes required in the Assembly Tree Generator
  - Private variables are no longer added to the private variable environment
  - Can no longer use the size of the variable environment as the size of the class
  - Need to add a “size” field to our internal representation of class types

10-111: **Access Control**

- Changes required in the Code Generator

10-112: **Access Control**

- Changes required in the Code Generator
  - We are still producing valid abstract assembly
  - No further changes are necessary

10-113: **Overloading Functions**

- Multiple functions (or methods in the same class) with the same name
- Use the # and types of the parameters to distinguish between them

```
int foo(int x);
int foo(boolean z);
void foo(int x, int y);
```

- Calls:

```
x = foo(3);
x = foo(true);
foo(3+5, foo(true));
```

10-114: **Overloading Functions**

- Just as in regular Java, can’t overload based on the *return type* of a function or method.
- Why not?

10-115: **Overloading Functions**

```
int foo(int x);
int foo(boolean y);

int bar(int x);
boolean bar(int x);

z = foo(bar(3));
```

- What should the compiler do?

10-116: **Overloading Functions**

- Changes required in the Lexical Analyzer

10-117: **Overloading Functions**

- Changes required in the Lexical Analyzer
  - Not adding any new tokens
  - No changes required

10-118: **Overloading Functions**

- Changes required to the Abstract Syntax:

10-119: **Overloading Functions**

- Changes required to the Abstract Syntax:
  - None!

10-120: **Overloading Functions**

- Changes required to the Semantic Analyzer
  - Need to distinguish between:
    - `int foo(int a, int b)`
    - `int foo(boolean c, int d)`

10-121: **Overloading Functions**

- Need to distinguish between:
  - `int foo(int a, int b)`
  - `int foo(boolean b, int d)`
- We could use `foointint` and `foobooleanint` as keys
  - Problems?

10-122: **Overloading Functions**

- `foo(3+4, bar(3,4));`
  - Need to convert (3+4) to "int", bar(3+4) to "int" (assuming bar returns an integer)
  - Better solution?

10-123: **Overloading Functions**

- `foo(3+4, bar(3,4));`
  - Convert the pointer to the internal representation of an integer to a string
  - Append this string to "foo"
  - Use new string as key to define function
    - `foo13518761351876`
  - From `3+4` and `bar(3,4)`, we can get at the pointer to the internal representation of the type

10-124: **Overloading Functions**

- Once we have expanded the key for functions to include the *types* of the input parameters, what further work is needed?

10-125: **Overloading Functions**

- Once we have expanded the key for functions to include the *types* of the input parameters, what further work is needed?
  - None!

10-126: **Recursive Classes**

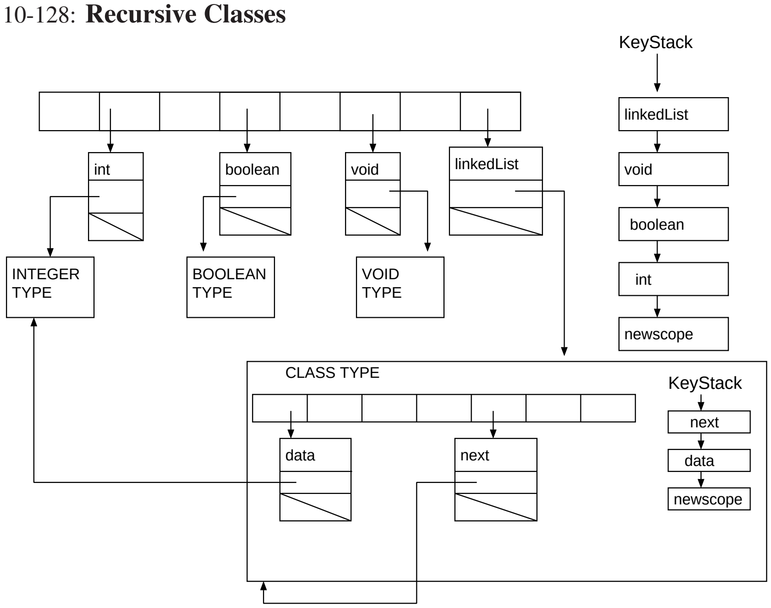
- Recursive classes allow for linked data structures

```
class linkedList {
  int data;
  linkedList next;
}
```

10-127: **Recursive Classes**

- Changes necessary to allow recursive classes
  - Add keyword "null"
  - Add "null expression" to AST
  - Add class to type environment before class has been completely examined
  - Allow "null" expression to be used for any class value

10-128: **Recursive Classes**



10-129: **Recursive Classes**

- Modifications to Semantic Analyzer

  - On assignment – if LHS is a class, RHS may be null
  - For any function call – if formal is a class, actual may be null
  - Comparison operations: ==, != – If either side is a class, the other can be null

10-130: **Virtual Methods**

```
class super {
   int foo() {
      return 1;
   }
}
class sub {
   int foo() {
      return 2;
   }
}
void main() {
  super x = new sub();
  print(x.foo());
}
```

10-131: **Virtual Methods**

- If the language uses static methods (as described so far), the static type of the variable defines which method to use
  - In previous example, static methods would print out 1
  - C++ uses static methods (unless specified as "virtual")
- If the language uses virtual methods, the type of the actual variable defines which method to use
  - In previous example, print out 2
  - Java uses *only* virtual methods (avoids some of the bizarre errors that can occur with C++)

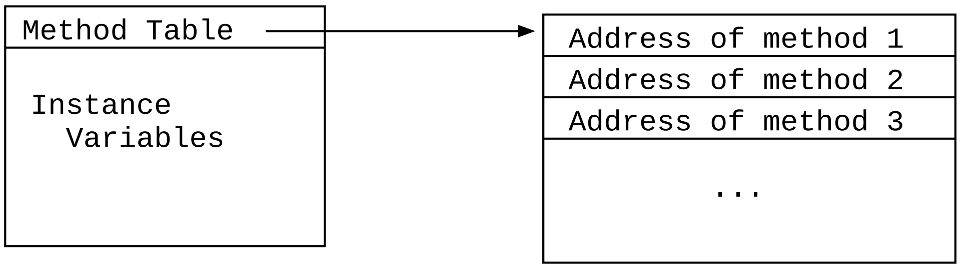
10-132: **Virtual Methods**

```
class superclass {
   int x;
   void foo() {
      ...
   }
   void bar() {
      ...
   }
}
class subclass extends superclass {  void main() {
   int y;
   void bar() {                         superclass a;
      ...                               a = new superclas();
   }                                    a.bar(); /* Point A */
   void g() {                           a = new subclass()
      ...                               a.bar(); /* Point B */
   }                                  }
}
```

10-133: **Virtual Methods**

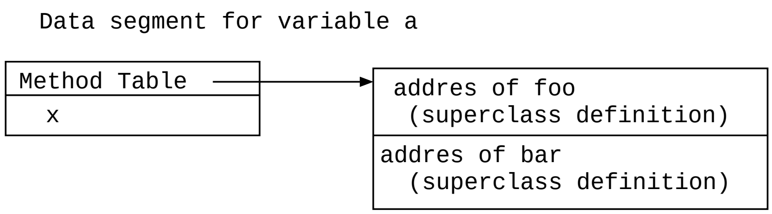
- We need to generate the exact same code for:
  - `a.bar()` at Point A
  - `a.bar()` at Point B

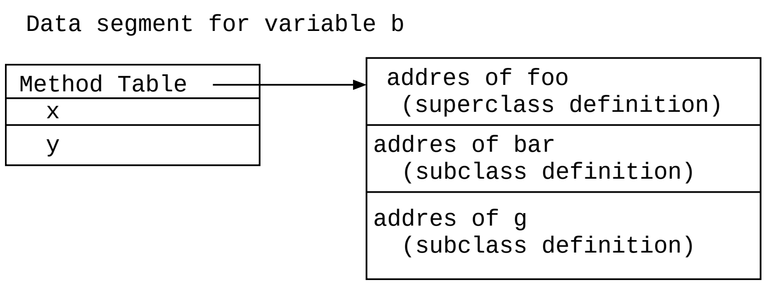
- Even though they will do different things at run time
- Function pointers to the rescue!



10-134: **Virtual Methods**

- Previously, the data segment held only the instance variables of the class
- Now, the data segment will also hold a pointer to a function table
  - Only need one table / class (not one table / instance)





10-135: **Virtual Methods**

10-136: **Virtual Methods**

- Function Environment
  - Previously, we needed to store the assembly language label of the function in the function environment
  - Now, we need to store the offset in the function table for the function

10-137: **Virtual Methods**

Environments for `superclass`

Function Environment		Variable Environment	
key	value	key	value
foo	0	x	4
bar	4		

Environments for `subclass`

Function Environment		Variable Environment	
key	value	key	value
foo	0	x	4
bar	4	y	8
g	8		

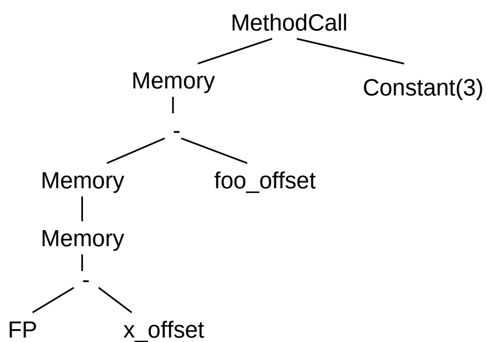
10-138: **Virtual Methods**

- When a method `x.foo()` is called
  - Look up `x` in the function environment

        - Returns a class type, which contains a local function environment
    - Look up `foo` in the local function environment
        - Returns the offset of foo in the function table
    - Output appropriate code

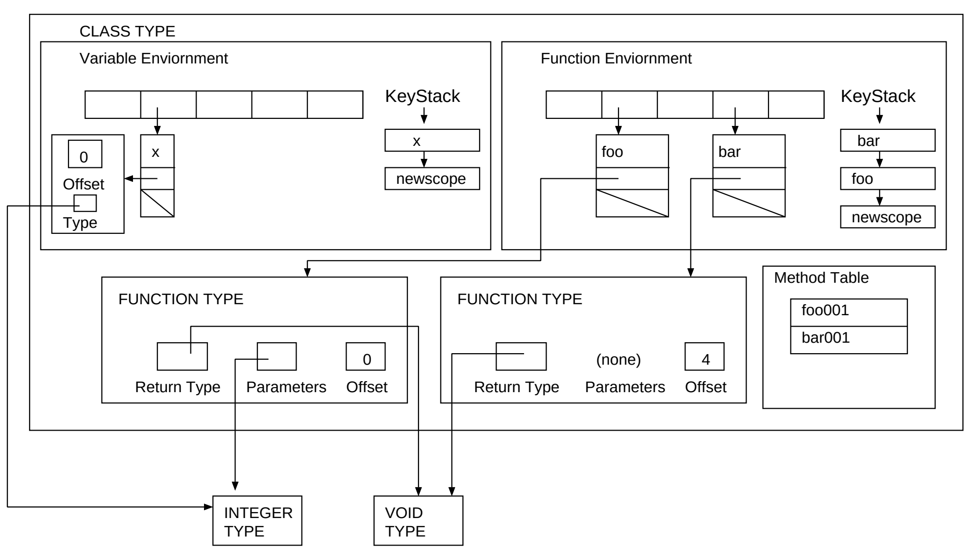
10-139: **Virtual Methods**

- When a method `x.foo(3)` is called
    - Output appropriate code
        - Extend our AAT to allow *expressions* as well as labels for function calls



10-140: **Virtual Methods Example**

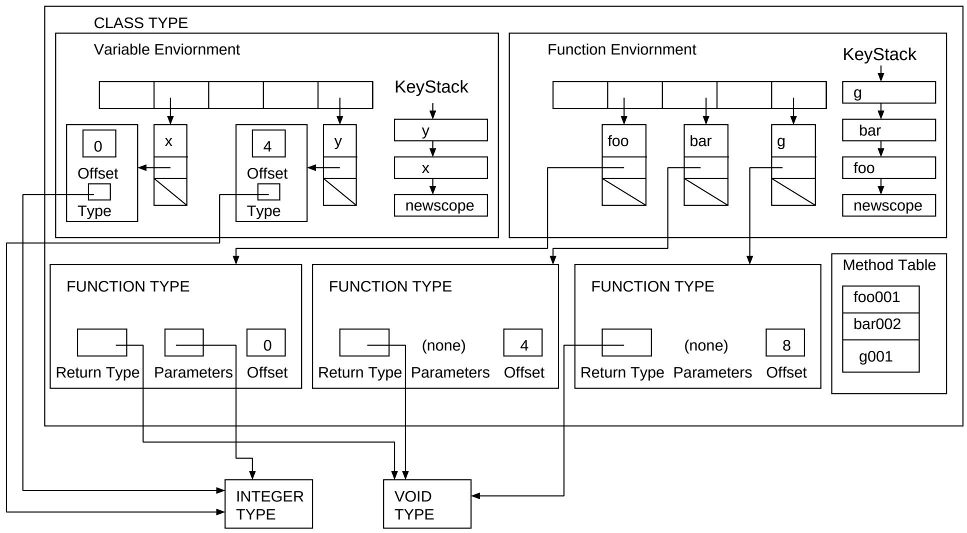
```
class baseClass {
   int x;

   void foo(int x) {
       /* definition of foo */
   }
   void bar() {
      /* definition of bar */
   }
}
```



10-141: **Virtual Methods Example**

10-142: **Virtual Methods Example**

```
class extendedClass {
  int y;

  void bar() {
    /*  definition of bar */
  }

  void g() {
    /* definition of g */
  }
}
```



10-143: **Virtual Methods Example**