B. Introduction to Data Types

There are two kinds of data types in Java, reference data types and primitive data types. Reference data types are *passed by* (or transferred by) name, where as primitive data types are passed by value.

For example, suppose you ask for the contents of a post office box by name. You say, “I want the contents of P.O. Box 320”. The box number serves to refer to the contents of the box. It would be much more difficult to ask for each item in the box by name. The box name is smaller than the names of all the items in the box. Addtionally, the number of items in the box is flexible.

A primitive data type, on the other hand, is of a size that is a function of the type itself. Thus, when a primitive data type is passed, the number of bits required to be copied is a function of the type .

While Appendix A describes the *syntax* of the primitive and reference data types, this section relates the syntax to the *semantics*.

B.1 The 8 basic data types in Java.

There are 8 basic data types in Java. Five of them are integer data types, 2 are floating point data types. The 8th data type is the *boolean* type. The boolean is valued at either *true* or *false*.

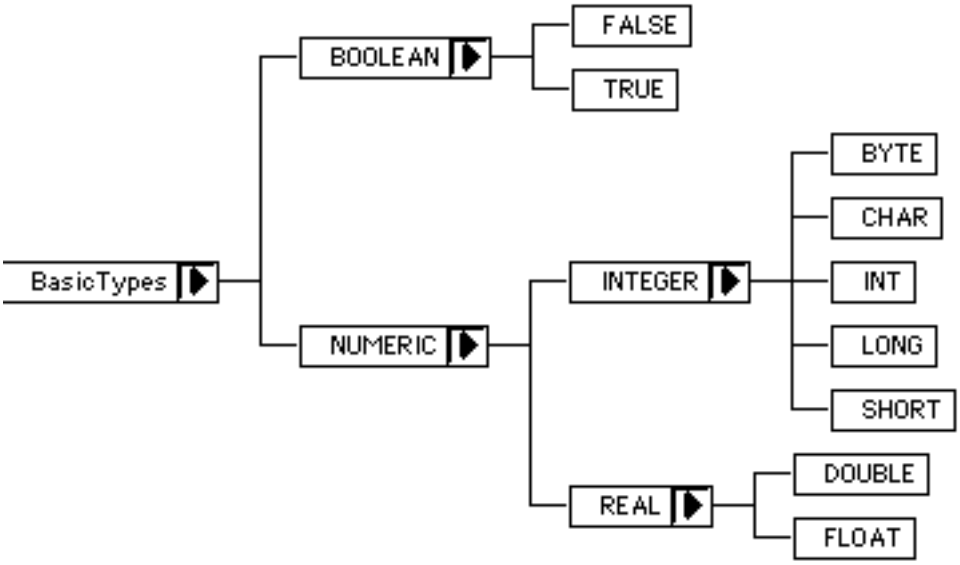

Figure B.2-1. The Taxonomy of basic data types.

Figure B.2-1 shows that the *integer* and *real* data types are kinds of *numeric* data types. Thus the Taxonomy represents an A.K.O (A Kind Of) relationship for all the basic data types. This relationship can be modeled with simple class inheritance. In fact, the class browser in CodeWarrior was used to generate Figure B.2-1 by the use of the following code:

```
public class BasicTypes {}
public class BOOLEAN extends BasicTypes {}
public class TRUE extends BOOLEAN {}
public class FALSE extends BOOLEAN {}
class NUMERIC extends BasicTypes {}
class INTEGER extends NUMERIC {}
class REAL extends NUMERIC {}
class CHAR extends INTEGER {}
class BYTE extends INTEGER {}
class SHORT extends INTEGER {}
class INT extends INTEGER {}
class LONG extends INTEGER {}
class FLOAT extends REAL {}
class DOUBLE extends REAL {}
```

More information is given about classes and class inheritance later in this appendix.

A summary of the different primitive data types is shown in Figure B.2-2.

type name	value	value	
boolean	1 bit	true, false	
char	16 bit	unicode	character
byte	8 bit	signed	$-2^{8-1} \ldots 2^{8-1}-1$
short	16 bit	signed	$-2^{16-1} \ldots 2^{16-1}-1$
int	32 bit	signed	$-2^{32-1} \ldots 2^{32-1}-1$
long	64 bit	signed	$-2^{64-1} \ldots 2^{64-1}-1$
float	32 bit	IEEE-754-1985	6-7 sig. figs.
double	64 bit		14-15 sig. figs.

Figure B.2-2. Basic Data Type Summary

In some books the basic data types are called *primitive* data types. Unlike other languages, Java has no facility for defining new basic data types.

B.1.1 Integer Data Types

There are five integer data types of different sizes and ranges. Four of the integer data types are signed integers (i.e., an integer that can be negative) and are expressed by using two's complement notation. Let *n* represent the number of bits needed to make up the integer data type. The most significant bit is used to store the sign of the number and the remaining *n*-1 bits are used to represent the value. The minimum value is $-2^{(n-1)}$ and the maximum value is $2^{(n-1)}-1$.

Integer Type	Bits Width	min range	max range
byte	8	-128	127
short	16	-32768	32767
int	32	-2147483647	-2147483647
long	64	"-9223372036854775808"	"9223372036854775807"

Fig. B.2.1-1. Signed Integer Data Types

The values used to set an *integer* data type are expressed as a sequence of decimal, octal, or hexadecimal digits. Fig. B.2.1-2 shows how to formulate constants in Java using the base of 10, 8 and 16.

decimal	octal	hexidecimal
42	"077"	0x2a

Fig. B.2.1-2. Constants in varying Radix

A *0* is used to denote an octal number. The 0x or 0X prefix denotes a hexadecimal number. Symbols corresponding to the decimal value of 10 - 15 are written as A - F. Integer literals are assumed to be of type *int* unless they end in the letter *L* or *l*.

The integer data type may be cast into the *char* data type, and then printed. Additional, the radix of the data's' string representation may be changed using API invocations. For example:

```
public class IntegerTest {
      public static void main(String args[]) {
        println("hello world");
        printHex(65535);
        for (int i=24; i < 52; i++){
          print((char)i+"");
        }
      }
      public static void printHex(int i) {
        println(Integer.toString(i,16));
      }
      public static void print(int i) {
        System.out.print(i);
      }
      public static void println(int i) {
        System.out.println(i);
      }
      public static void print(Object o) {
        System.out.print(o);
      }
      public static void println(Object o) {
        System.out.println(o);
      }

      public static void numberTest(
          int i,
          double d) {
        float f = 1.2f;
        short s = 3;
        byte b = -4;
```

```
          char c = 'J';
          boolean t = true;
          long l = 3294;
          System.out.println(
            "i="+i+
            "\nd="+d+
            "\nf="+f+
            "\ns="+s+
            "\nb="+b+
            "\nc="+c+
            "\nt="+t+
            "\nl="+l+
            "\ni="+
            Integer.toString(i,36)+
            "+"+
            Integer.toString(s,36)+
            "="+
            Integer.toString(s+i,36));

            f = (float)d;
            b = (byte)f;
            c = (char)f;

        }
    }
```

The above code will print:

```
    hello world
    ffff
          - !"#$%&'()*+,-./0123
```

B.1.2 Elementary Example of primitive data types

Primitive types must be initialized before being used in Java.

For example, in the following code:

```
    public class TrivialApplication {

          public static void main(String args[]) {
            int x;
            System.out.println( x );
          }

    }
```

a variable, *x* , was declared, but not initialized before being accessed. This is a *compile-time error* in Java and the compiler emits:

```
Error   : Variable x may not have been initialized.
TrivialApplication.java line 7     System.out.println( x );
```

The data types of *byte*, *short*, *int*, *long*, *float* and *double* are signed. The data types of *boolean* and *char* are unsigned. The *char* data type in Java is represented by an international standard for character representation, called Unicode [Unicode]. For example:

```
char c = ‘a’;
```

It is possible to assign a numeric literal to a character-typed variable and a character literal to an integer variable:

```
char theChar = 48;
integer theValue = ‘a’;
```

All real numbers in Java are stored either as *single* or *double* precision variables, called *float* and *double*. The Java *reals* use the IEEE 754-1985 format [IEEE]. The smallest and largest positive non zero values for floats range from 1.40239846e-45 to 3.40282347e+38 (Float.MIN_VALUE and Float.MAX_VALUE). The smallest and largest positive non zero values for doubles range from 4.94065645841246544e-324 to 1.79769313486231570e+308 (Double.MIN_VALUE and Double.MAX_VALUE).

For example:

```
class FpError {
      public static void main(String argv[]) {
        float fmin = Float.MIN_VALUE;
        float fmax = Float.MAX_VALUE;
        double dmin = Double.MIN_VALUE;
        double dmax = Double.MAX_VALUE;
        String b = " ";
        System.out.println(fmin + b + fmax +
          b + dmin + b + dmax);
      }
}
```

The above outputs:

```
1.4E-45 3.4028235E38 4.9E-324 1.7976931348623157E308
```

The following program shows how to code the escape sequences into Java:

```
public interface Char {

      char backspace = '\b';
      char horizontalTab = '\t';
      char newLine = '\n';
      char formFeed = '\f';
      char carrageReturn = '\r';
      char doubleQuote ='\"';
      char singleQuote ='\'';
      char backSlash = '\\';
      char maxOctal = '\377';
      char minOctal = '\000';
      char maxUnicode = '\uFFFF';
      char minUnicode = '\u0000';
}
```

Note how the back slash "\" is used as an escape character. Thus, to print a back slash, you must insert two back-slashes, '\\'. Also note, the unsigned 16 bit characters use '\uxxxx' to indicate the hexadecimal representation of the character.

B.2 Classes

This section introduces basic object oriented ideas and usage. In the first section we describe the concept of a class.

B.2.1 Class Concepts

An instance of a class is called an *object*. An object can contain both the data structure of a program and the algorithms needed to manipulate the data structure. As a result of containing the data structures, an instance is said to contain *properties*. As a result of having both properties and the methods for manipulating them, an instance is said to be modular.

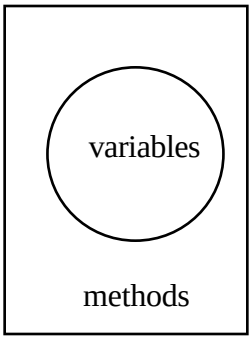

Fig. B.2.1-1 Variables and Methods in a Class

The class name consists of an identifier (which may be of any length), and the class may *extend* another class. In Java, classes are able to form an AKO (A-Kind-Of) taxonomy, as described in Appendix A. For example, if a *mammal* has hair, then we can write:

```
public class Mammal {
      boolean hasHair = true;
}
```

Methods in the *Mammal* class can alter the properties in an instance of a mammal. For example:

```
public class Mammal {
      boolean hasHair = true;
      public void makeBald() {
        hasHair = false;
      }
}
```

In Java, we define classes of objects and then make instances of them. The data structures in a class can be primitive data types or reference data types (i.e., instances of other classes). Since classes can communicate with one another, this can cause interdependence between classes.

An instance of a class has a type. The *new* operator is the *only* way to make an instance of a class. For example:

```
class point {
      public double x,y;
}
```

```
point p1 = new point();
p1.x = 10;
p1.y = 11;
```

Java allows inheritance between classes and subclasses. Subclasses can inherit properties and methods from the superclasses. This permits subclasses to build on the methods and data structures of the parent class. To build on the methods and data structures of a parent class, a subclass is constructed that *extends* the parent class. For example:

```
public class Human extends Mammal {
```

In this case, the *Human* class extends the *Mammal* class. This means that the Human is a kind of Mammal. As a result, the *Human* class is a subclass of the *Mammal* class and inherits properties (like *hasHair*).

All classes descend from the *Object* class. As a result the Object class is called the *primordial* class. As a result of being a subclass of the Object class, all instances are able to support the *toString* method. The *toString* method creates a string representation of an instance. The string representation of the object is used to assist in printing.

B.2.2 Overloaded Methods

A class's methods enable manipulation of class member variables. Special method, called the *constructor* enables the creation of new class instances. The constructor must have the same name as the class. The constructor returns nothing, not even void. If no constructor is specified then a default constructor is provided. The default constructor takes *null* as an argument. The default constructor is overridden when another constructor is specified. For example:

```
class Lamp {
      boolean on;
      int Wattage;
      Lamp (int w) {
        Wattage = w;
      }
      Lamp ( ) {
      Wattage = 100;
```

```
        }
}
```

The *Lamp* constructor has been overloaded with two versions. The first version will support:

```
Lamp dim = new Lamp(40);
```

While the second version supports the constructor invocation:

```
Lamp bright = new Lamp();
```

Java requires that the methods have different signatures. The signature of the method is determined by the number of arguments and their compile-time types.

B.2.3 Getter and Setter Methods

It is generally a good idea to use *getter and setter* methods when trying to read or write a class variable from outside of a class. In order to force the usage of the *getter* and *setter* methods, it is typical to declare the instance variables *private*. For example:

```
public class GetterSetter {
      private float salary = 40000;

      public float getSalary() {
        return salary;
      }

      public void setSalary(float _salary) {
        salary = _salary;
      }
}
```

The *GetterSetter* example shows that the salary cannot be accessed directly. A *public* method, *getSalary* enables the reading of salary. To set the salary, only the setSalary method can be used for external classes. For example:

```
public class Broken {
      GetterSetter gs = new GetterSetter();
      void test() {
        gs.salary = 10;
      }
}
```

Produces a compile time error:

```
Error   : Variable salary in class GetterSetter not accessible
       from class Broken.
GetterSetter.java line 15     gs.salary = 10;
```

Getter and setter methods have become important in the area of *Java beans*. Java beans are a component technology that is beyond the scope of this appendix to discuss.

In some books the getter method is called the *accessor* and the setter method is called the *mutator*.

B.2.4 Casting

Type conversion in Java is called *casting*. When casting is performed, it is a run-time operation. Casting is able to convert only between compatible types and always results in a value, not a variable.

Sometimes the only way to know for sure when types are compatible is to run the program. If a ClassCastException is thrown at run-time, then the type conversion failed.

It is always correct to cast an instance from a subclass to its superclass. For example:

```
1.    for (int i=0; i < v.size(); i++) {
2.        s = (Shape) v.elementAt(i);
3.        s.print();
4.        }
```

In line 1, an instance of a Vector, *v* is accessed for size. The elements in the vector are accessed using line 2. Note that each element in the vector is a class that extends the Shape class. It is always correct to cast the subclass of the Shape class back into the superclass. This enables print() method invocation on each shape in the vector instance.

B.2.5 Null

One of the literals of Java is *null*. Null is what you get when nothing has been created. For example:

```
if (some_object != null) {
      System.out.println(“Object Exists!”);
}
```

Null has a *null* type and is the default value for any type that has not been created. For example:

```
class Test {
      Lamp l;
}
Test t = new test();
```

At this point, *tl.* is equal to null. To make an instance of the *Lamp* instance variable, *l*, you must first create an instance of *Test* and then instance *l*. Before this time, the *l* instance variable will be *null*. For example:

```
Test t = new Test();
t.l = new Lamp();
```

B.2.6 Subclassing and Super

One feature of the Java class is that it can intrinsically represent taxonomic structures. The taxonomic structures are formed by Java classes when a sub-class *extends* a superclass. This type of extension is called direct inheritance. Thus, in terms of knowledge representation, Java classes can represent the AKO (a-kind-of) relationship. In addition, Java classes can represent the has-a relationship using the class member variables. For example, we can represent the statement: “A student is a-kind-of human” by creating a student class that extends the human class. We can also represent the statement: “The student has-a pencil” by placing a class member variable of pencil class type into the student class construct. In the following section we present the syntax of Java and its relationship to the semantics of Java.

A class may be used to provide a container for an instance variable of any primitive type. For example:

```
class Lamp {
      boolean on;
}
...
Lamp l = new Lamp ( );
l.on = true;
```

A Java class may be used to store a reference to named constants:

```
class Constants {
      static final double PIon2 = Math.PI / 2;
}
```

Notice that these class examples have no methods. When one class extends another, we are sub-classing a superclass. The sub-class will inherit the member-variables, and methods, of the superclass. In the case of a name conflict, the sub-class implementation always over-rides the superclass implementation. For example:

```
class Lamp extends Constants {
      double power = 100 / PIon2; // watts
      boolean on = true;
      }
```

The power in the Lamp class is set using a PIon2 constant that is inherited from the Constants class. In this case, it is not strictly correct to say that the Lamp is a-kind-of Constants and thus the *extends* is being used as a programming convenience, not a means for knowledge representation.

On the other hand:

```
class Student extends Human {
      Pencil p;
}
class Human {
      boolean bald = false;
}
```

Now we represent the statement that “Doug is a bald student with a pencil”:

```
Student doug = new Student();
doug.p = new Pencil();
doug.bald = true;
```

Super is a keyword that permits a subclass to call-upon the instance variable or method of the superclass. For example:

```
public class SuperDemo {
      public void print() {
        System.out.println("Oh, super man!");
      }
}

public class SubclassDemo extends SuperDemo{
      public void print() {
        System.out.println("We are going down, uh huh!");
        super.print();
      }
      public static void main(String args[]) {
```

```
        SubclassDemo scd = new SubclassDemo();
        scd.print();
      }
}
```

Outputs:

```
We are going down, uh huh!
Oh, super man!
```

Note that the *print* method has a different side-effect when invoked from the *SubclassDemo* than from the *SuperDemo*. We say that the *SubclassDemo* overrode the *print()* method in *SuperDemo*. In order to invoke the superclasse's print statement, a *super.print()* was needed.

Consider the following example:

```
/**
      An example of storing the AKO hierarchy
      using classes.
*/
public class Mammal {
      private boolean hasHair = true;
}
public class Human extends Mammal {

      public static void main(String args[]) {
        System.out.println( "Hello World!" );

      }
}
public class Doggy extends Mammal {
}
public class Student extends Human {
}

public class Professor extends Human {
}
```

Fig. B.2.6-1 shows the hierarchy in the *Professor* class.

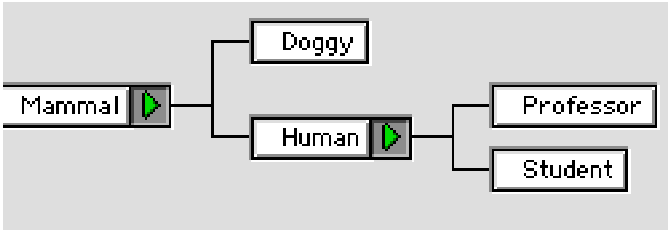

Fig. B.2.6-1. The Hierarchy in the *Mammal* class.

Fig. B.2.6-1 shows that a *Doggy* is a kind of *Mammal*. It also shows that a *Student* is a kind of *Human* and that a *Human* is a kind of *Mammal*, etc.

In summary,

1. Subclassing permits code reuse by inheritance
2. Subclassing permits the incorporation of super class properties
3. Subclass creates and AKO hierarchy.

B.2.8 Inner Classes

Inner classes are classes that are defined locally within another class. These are called *nested* classes. Nested classes can only be used within the containing classes. For example:

```
/*
      This example shows
      how to create a nested class.
      The Outer class is class called "Outer".
      The Inner class is class called "Inner".
      The Outer class contains the Inner class.
      Inner classes are useful for defining temporary
      reference data types.
*/
class Outer {
      Inner i = new Inner();
      public static void main(String args[]) {
        Outer o = new Outer();
        o.i.print();
      }
      class Inner {
        void print() {
          System.out.println("hello from Inner class!");
        }
      }
```

```
}
```

The output follows:

```
hello from Inner class!
```

The *inner* classes are temporary and locally scoped. This means that classes that do not contain the *Outer* class are unable to make instances of the *Inner* class directly. For example:

```
class DontWork {
      Inner i = new Inner();
}
```

creates the syntax error:

```
Error   : Class Inner not found.
Outer.java line 23    Inner i = new Inner();
```

Thus the class *DontWork* cannot compile.

In the following example, methods are overridden in an inner *implementation* of a non-inner class. This consititutes a new kind of design pattern (as of JDK 1.1) called an *adapter*. The adapter pattern has seen increased use in the GUI class libraries, but can lead to confusing code. It is probably better software engineering to limit the use of adapters to small classes. An example adapter follows:

```
public interface Accountable {
      public double getBalance();
      public void setBalance(double d);
}

public class AccountableAdapter implements Accountable {
      public double getBalance(){return 0;};
      public void setBalance(double d){};
}

public class Customer {
      AccountableAdapter aa
        = new AccountableAdapter() {
          public void setBalance(double d) {
               System.out.println("balance ="+d);
               super.setBalance(d);
          }
        };
}
```

In summary,

1. Inner classes are always dynamically allocated.
2. Instances of inner classes are available externally.
3. You cannot make a new instance of an inner class externally.
4. Inner class duration is temporary.
5. Inner classes are locally scoped.
6. Inner classes may not have static members.

B.2.9 This

The *this* reference can be used when a reference is required as an argument to a method. For example, the following class supports a print method that enables it to print itself:

```
public class ThisTest {
      public static void main(String args[]) {
        ThisTest tt = new ThisTest();
        tt.print();
      }
      public void print() {
        System.out.println(this);
      }
      public String toString() {
        return "Hello from ThisTest!!";
      }
}
```

Will output:

```
Hello from ThisTest!!
```

The *this* reference permits disambiguation between the arguments to a method and the class member variables. For example:

```
class Constructor {
      int x,y;
      Constructor(int x, int y) {
        this.x = x; // disambiguate between x and this.x
```

```
        this.y = y; // which y is y?
      }
      Constructor() {
        this(10,20); // default values being set with this!
      }
      void print() {
        System.out.println("x,y="+x+","+y);
      }
      public static void main(String args[]) {
        Constructor c = new Constructor();
        c.print();
      }
}
```

In the above example we see that the constructor that takes not arguments *Constructor()* invokes the constructor that takes arguments, *Constructor(int x, int y)*. This permits a default value to be set for the variables.

The *this* reference permits a class to make a reference to an instance of itself. This can be very useful, particularly when working with *inner* classes. For example:

```
class Outer {
      int x=10;
      Inner i = new Inner();
      public static void main(String args[]) {
        Outer o = new Outer();
        o.i.print();
      }
      class Inner {
        int x=20;
        void print() {
          System.out.println("inner x= "+x);
          System.out.println("outer x= "+Outer.this.x);
        }
      }
}
```

Will print:

```
inner x= 20
outer x= 10
```

Thus, the usage of the *this* reference in *Outer.this* permits a reference to the instance variable in the outer class, from within the inner class.

B.2.7 Interfaces

An interface is like an abstract class with only abstract methods and constant fields. The interface can hold no method implementations and is defined just like a class except that it uses the keyword *interface* rather than class. What follows is the MBNF for the interface declaration:

```
interfaceDeclaration →
        < modifier > "interface" identifier [ "extends" interfaceName
        < "," interfaceName > ] "{" < fieldDeclaration > "}" .

fieldDeclaration →
        ( [docComment] (
            methodDeclaration |
            constructorDeclaration |
            variableDeclaration ) ) |
          staticInitializer | ";" .

interfaceName →
        identifier | ( packageName "." identifier ) .
```

Note that an interface declaration can extend multiple interfaces. The interface can serve as another reference type, but can never be instanced. Thus classes that implement an interface can always be cast back to the interface type. Here are a few of the properties of classes and interfaces:

1. Class and interface names populate the name-space of a package and should be unique to prevent name-space conflicts.
2. Interfaces can be declared as abstract, but this is not needed.
3. Interface variables can never be set and are allocated with storage at compile time. It is possible to declare interface variables at *static* and *final*, but this is not needed.
4. All fields must be initialized.
5. Interfaces have no common ancestor, like *Object*.
6. Field names in an interface must be unique, or it is a syntax error.

For example:

```
public interface RealDumb {
       double PI = 4;

      }
public interface Dum {
      double PI = 3;
}

public interface MixedUp extends Dum, RealDumb
      {double foo=PI;}
Error   : Reference to PI is ambiguous. It is defined in
       interface real_dumb and interface dum_constants.
constants.java line 17    {double foo=PI;}
```

The following are some correct uses of interfaces. For the first example we show how interfaces may be used to group constants together:

```
public interface constants {
       double Pi_on_180 = Math.PI / 180;
       double PI = Math.PI;
       double Pi_on_2 = Math.PI/2;
       double Pi_on_4 = Math.PI/4;
}
```

Here is one where there is a large array of symbols being stored:

```
public interface CplusplusText {
      public static String cplusplusReservedWords[] = {
        "asm",
        "auto",
        "break",
        "case",
        "catch",
        "char",
        "class",
        "const",
        "continue",
        "default",
        "delete",
        "do",
        "double",
        "else",
        "enum",
        "extern",
        "float",
        "friend",
```

```
        "for",
        "goto",
        "if",
        "inline",
        "int",
        "long",
        "new",
        "operator",
        "private",
        "protected",
        "public",
        "register",
        "return",
        "short",
        "signed",
        "sizeof",
        "static",
        "struct",
        "switch",
        "this",
        "throw",
        "try",
        "typedef",
        "union",
        "unsigned",
        "virtual",
        "void",
        "volatile",
        "while"
      };
```

Using interfaces, Java has multiple inheritance of prototypes. It also has multiple inheritance of constants. Consider the following example:

```
interface X {
      double PI = Math.PI;
      double PiOn2 = PI/2;
}
interface Y {
      double E = Math.E;
}
interface Z extends X,Y{
}

public class
```

```
        GetsConstantsFromZ
          implements Z {

        public static void main(
          String args[]) {
          System.out.println(
            "PI="+PI);
          System.out.println(
            "E="+E);

        }
}
```

The *GetsConstancesFromZ* actually inherits constants from both the *X* and *Y* interfaces. It is a compile-time error to have a name-conflict in multiply inherited interfaces.

Here is an example of the interface extending multiple interfaces.

```
package interfaceExample;
public interface Drawable {
      public void draw();
}
public interface Movable {
      public void move(double x, double y);
}

public interface GraphicsObject extends Movable, Drawable {
}
public class Mammal implements GraphicsObject {
      private boolean hasHair = true;
      private double x = 0;
      private double y = 0;

      public void move(double _x, double _y) {
        x = _x;
        y = _y;
      }
      public void draw() {}; // does nothing right now.
}
public class Human extends Mammal {

      public static void main(String args[]) {
        System.out.println( "Hello World!" );
```

```
        }
}
public class Doggy extends Mammal {
}
public class Student extends Human {
}

public class Professor extends Human {
}
```

Figure B.2.8-1 shows the output of the class hierarchy browser in Metrowerks when used to view the relationships described above.

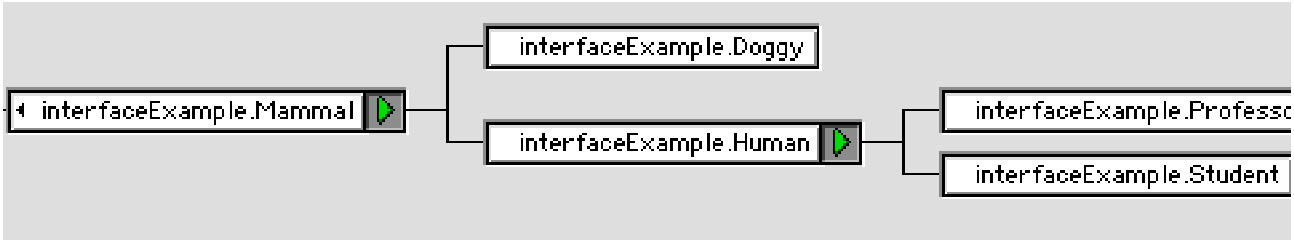

Fig. B.2.8-1. Interface Example showing A.K.O Hierarchy

B.3 Summary

This appendix covers some of the basic concepts of the data types. The two basic types covered were the reference data types and the basic data types. Brief coverage was made for the 8 basic data types. The primary reference data type of interest is the *class* data type. There are other data types, but these are covered in the following appendix. The coverage of the class data type included coverage of the *getter*and *setter* methods. Brief coverage was given to *casting* and the *null* operator.

Casting is a rather difficult topic and deserves a must larger coverage. We shall expound on casting more when we cover *Container Classes* in a later appendix. The *inner class* coverage was also brief. This shortened version tells you almost all you need to know. Coverage could be expanded to include anonymous classes, but this will be described once we start to cover *adapters* in a later appendix.

Finally, we covered the *this* reference. The *this* reference is not often used, but is, none-the-less important to understand, particularly when accessing outer class variables.