Object-Oriented Programming

Classes

- Classes are syntactic units used to define objects.

- They may contain instance variables, which will occur in each instance of the class, instance methods, which can be executed by objects of the class, and constructors, which are called automatically when an object is created using **new**.

- Classes may also have class variables and class methods, but these belong to the class itself and have no direct effect on the objects.

```
class MyClass
{
        private int value;

        MyClass(int n)
        {  value = n;  }

        void perform(int m)
        {       for (int k=1; k<=value; k++)
                    System.out.print (m*k + "  ");
        }

        int compute()
        {  return value*value;  }
}
```

Objects

- Objects are created from a class using the **new** operator, which invokes a constructor with matching parameter types.
- These objects may be assigned to variables declared of the type given by the class name.
- Each object has a copy of every instance variable in its class definition and in every superclass of that class.
- Instance methods in a class can be called only with an object of the class type (or a subclass).
- This object is called the *receiver* of the method and can be referred to by the keyword **this** inside of the method.

```
MyClass first = new MyClass(5);
MyClass second = new MyClass(3);

first.perform(6);
Prints:     6  12  18  24  30

second.perform(-4);
Prints:     -4  -8  -12
```

Constructors

- A constructor is a method that is called automatically when an object is created.
- If the programmer supplies no constructor, a default constructor with no parameters is provided.
- This default constructor disappears if the programmer writes one or more constructors in the class.
- In a constructor, **this**(…) calls another constructor of the same class with the given parameters and **super**(…) calls a constructor of its superclass with the given parameters.

Another constructor for MyClass

```
MyClass()
{ this(10); }
```

Inheritance

- A new class can be defined as a subclass of an existing class using the **extends** keyword.
- Then every object of the new subclass will have copies of the instance variables from its superclass (and its superclass and so on) as well as its own instance variables.
- It can also call instance methods defined in its superclass as long as they are visible (not private).
- Any instance method in the superclass can be overridden (re-defined) by writing a method in the subclass with the same signature.

- Any class definition without an extends clause is a subclass of Object by default.

- A variable of the superclass type may refer to an object of its class or an object of any of its subclasses (upcasting).

- If an overridden instance method is called on a variable of the superclass, the class of the object referred to determines which version of the overridden method will be executed. This property is known as *polymorphism* or *dynamic binding*.

- In an instance method, the identifier **this** refers to the object, the receiver, that is currently executing the instance method.

- The identifier **super** can be used to access instance methods (and variables) that have been overridden (and shadowed) in the subclass.

```
class MySub extends MyClass
{
    private boolean okay;

    MySub(int n, boolean b)
    {
        super(n);        // assigns n to value, a private
        okay = b;        //    instance variable
    }

    int compute()
    {
        if (okay) return super.compute();
        else return -1;
    }
}
```

What happens if we omit **super** in the definition of *compute*?

```
MyClass mc = new MyClass(33);
MySub ms = new MySub(12, true);
MyClass mcs = new MySub(-9, false);
```

```
    mc.perform(5)     calls parent method
    ms.perform(5)     calls parent method
    mc.compute()      calls parent method
    ms.compute()      calls child method
    mcs.compute()     calls child method (polymorphism)
```

Upcasting and Downcasting

- Upcasting refers to the mechanism in which an object from a subclass is assigned to a variable declared of the superclass type. No special operator is required since the subclass object "is-an" object of the superclass type automatically.

- Downcasting refers to the assignment of a superclass variable or the result from a function to a subclass variable, and it requires an explicit cast to the type of the subclass.

```
Upcasting:    mc = ms;
              mc = mcs;

Downcasting:  ms = (MySub)mcs;  // legal only because mcs
                                // refers to a MySub object
```

Illegal downcast: *ms = (MySub)mc;*

throws a ClassCastException

Downcasting

- Assignment of a superclass variable or function result to a subclass variable; requires an explicit cast to the type of the subclass.
- A variable of a superclass type can be cast to a variable of a subclass type only if it refers to an object of that same subclass type.
- If the object referred to by the superclass variable is not an object of the subclass, a ClassCastException is thrown signaling an error.
- Upcasting and downcasting of object types also applies to parameter passing and to any situation where an object of one type is impersonating another class type.

Polymorphism Example

Define a collection of classes representing geometric solids and including a method for computing their volumes.

The superclass provides a String instance variable for identification and a volume method to be overridden.

```
class Solid
{
        private String kind;

        Solid(String k)
        {
                kind = k;
        }

        String getKind()
        {
                return kind;
        }

        double volume()
        {
                return 0.0;             // This code is never executed
        }
}
```

```
class Sphere extends Solid
{
        private double radius;

        Sphere(double r)
        {
                super("Sphere");
                radius = r;
        }

        double volume()
        {
                return 4.0/3.0*Math.PI*radius*radius*radius;
        }
}

class Cube extends Solid
{
        private double length;

        Cube(double g)
        {
                super("Cube");
                length = g;
        }

        double volume()
        {
                return length*length*length;
        }
}

class Cone extends Solid
{
        private double radius, altitude;

        Cone(double r, double a)
        {
                super("Cone");
                radius = r;        altitude = a;
        }
```

```
	double volume()
	{
		return Math.PI*radius*radius*altitude/3.0;
	}
}

public class UseSolid
{
	public static void main(String [] args)
	{
		Solid [] list = new Solid [6];

		list[0] = new Cube(10);
		list[1] = new Cube(5);
		list[2] = new Sphere(10);
		list[3] = new Sphere(8);
		list[4] = new Cone(3, 5);
		list[5] = new Cone(8, 2);

		for (int k=0; k<list.length; k++)
			System.out.println(list[k].getKind() + " volume = "
							+ list[k].volume());
	}
}


/******************************************

% java UseSolid
Cube volume = 1000.0
Cube volume = 125.0
Sphere volume = 4188.790204786391
Sphere volume = 2144.660584850632
Cone volume = 47.1238898038469
Cone volume = 134.0412865531645

******************************************/
```

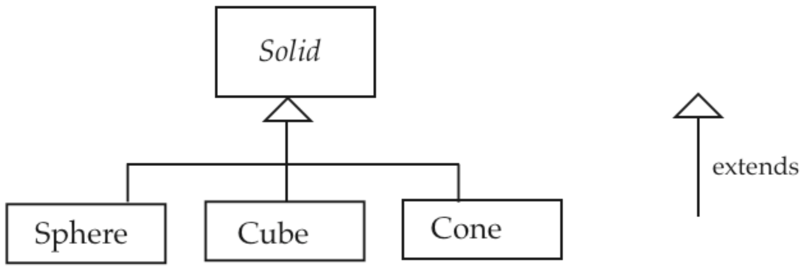
Abstract Classes

- The reserved word **abstract** can be used as a modifier for an instance method or a class.
- An abstract method has no body. It must be overridden in a subclass of its class to be made into a “concrete”, callable method.
- Any class with an abstract method must be declared as an abstract class, although any class can be made abstract using the modifier.
- An abstract class cannot be instantiated.
  It must be extended by a “concrete” subclass to create objects.

  Those objects may be assigned to a variable of the type of the abstract superclass (upcasting).

```
abstract class Solid
{
      private String kind;

      Solid(String k)
      {
            kind = k;
      }

      String getKind()
      {
            return kind;
      }

      abstract double volume();   // Note semicolon
}
```

Solid Hierachy



The program UseSolid works exactly the same with this abstract class Solid.

Interfaces

- An interface is a class-like unit that provides a specification of behavior (the syntax only) that classes must implement to make the behavior available.

  It contains only **public** instance method headers (no bodies) and **public static final** class variables.

- No objects can be instantiated directly from an interface. A class **implements** an interface by giving complete definitions of all of the methods in the interface.

- A variable declared of the interface type can be made to refer to an object of a class that implements the interface (upcasting).

- Polymorphism can be performed with a group of classes that implement a common interface.

```
interface Printable
{
    void printNum(int n);     // Automatically public and abstract
}
```

```
class FirstImpl implements Printable
{
        String name;

        FirstImpl(String s)
        {
                name = s;
        }

        public void printNum(int n)
        {
                System.out.println(name + " prints " + n);
        }
}

class SecondImpl implements Printable
{
        int ID;

        SecondImpl(int n)
        {
                ID = n;
        }

        public void printNum(int n)
        {
                System.out.println("Number" + ID + " prints " + n);
        }
}


FirstImpl fi = new FirstImpl("claude");
SecondImpl si = new SecondImpl(99);

Printable [] pt = new Printable[2];

pt[0] = fi;
pt[1] = si;
```

```
for (int k=0; k<pt.length; k++)
        pt[k].printNum(20*k+5);
```

Output

```
        claude prints 5
        Number99 prints 25
```

The array could also be created using:

```
        Printable [] pt = { fi, si };
```

Some Interfaces Defined in the Java API

java.lang.Cloneable	java.lang.Comparable
java.lang.Runnable	java.io.DataInput
java.util.Iterator	java.io.Serializable
java.util.Comparator	java.awt.event.ActionListener

Another Approach to Polymorphism

Put the abstract method in an interface.

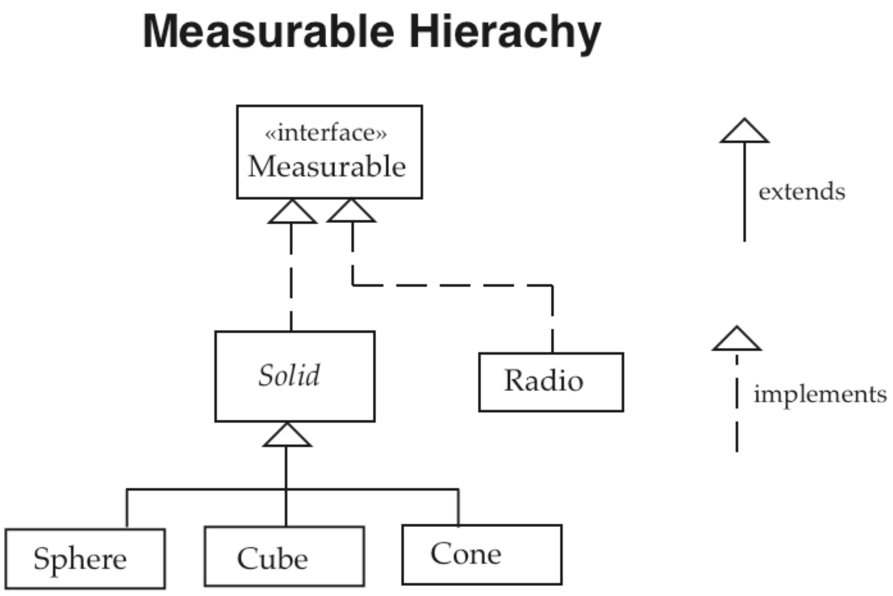
Need to put the method *getKind* there also for polymorphism to work.

```
interface Measurable
{
        String getKind();             // Automatically public
        double volume();              // Automatically public
}
```

```
abstract class Solid implements Measurable
{
        private String kind;

        Solid(String k)
        {   kind = k;  }

        public String getKind()           // public because of interface
        {   return kind;   }
}

class Sphere extends Solid
{
        private double radius;

        Sphere(double r)
        {
                super("Sphere");
                radius = r;
        }

        public double volume()        // public because of interface
        {
                return 4.0/3.0*Math.PI*radius*radius*radius;
        }
}

class Cube extends Solid
{ … }

class Cone extends Solid
{ … }
```

Consider another kind of object that has a “volume”.

```
class Radio implements Measurable
{
      public String getKind()
      {  return "Radio";  }

      public double volume()
      {  return 99.99;
      }
}
```



```
public class UseSolid
{
	public static void main(String [] args)
	{
		Measurable [] list = new Measurable [7];
		list[0] = new Cube(10);
		list[1] = new Cube(5);
		list[2] = new Sphere(10);
		list[3] = new Sphere(8);
		list[4] = new Cone(3, 5);
		list[5] = new Cone(8, 2);
		list[6] = new Radio();
		for (int k=0; k<list.length; k++)
			System.out.println(list[k].getKind() + " volume = "
							+ list[k].volume());
	}
}

/*****************************************
% java UseSolid
Cube volume = 1000.0
Cube volume = 125.0
Sphere volume = 4188.790204786391
Sphere volume = 2144.660584850632
Cone volume = 47.1238898038469
Cone volume = 134.0412865531645
Radio volume = 99.99
*****************************************/
```

Reference Types

	May refer to
• class	an object of that class or of one of its subclasses
• array	an array of the appropriate type
• abstract class	an object of a “concrete” subclass of the abstract type
• interface	an object of a class that implements the interface

Variables of the reference types may have the value **null** or may refer to objects.

The last two types cannot be instantiated.

final

The reserved word **final** can be used as a modifier for a class, for an instance method, or for a variable.

- A **final** class may not be subclassed (extended).
- A **final** instance method may not be overridden (turns off polymorphism).
- A **final** variable (any kind) must be initialized when declared and may not be subsequently altered.

  Exception: A **final** instance variable may be initialized in a constructor.

Wrapper Classes

Recall: The eight primitive types are *not* objects, for efficiency reasons

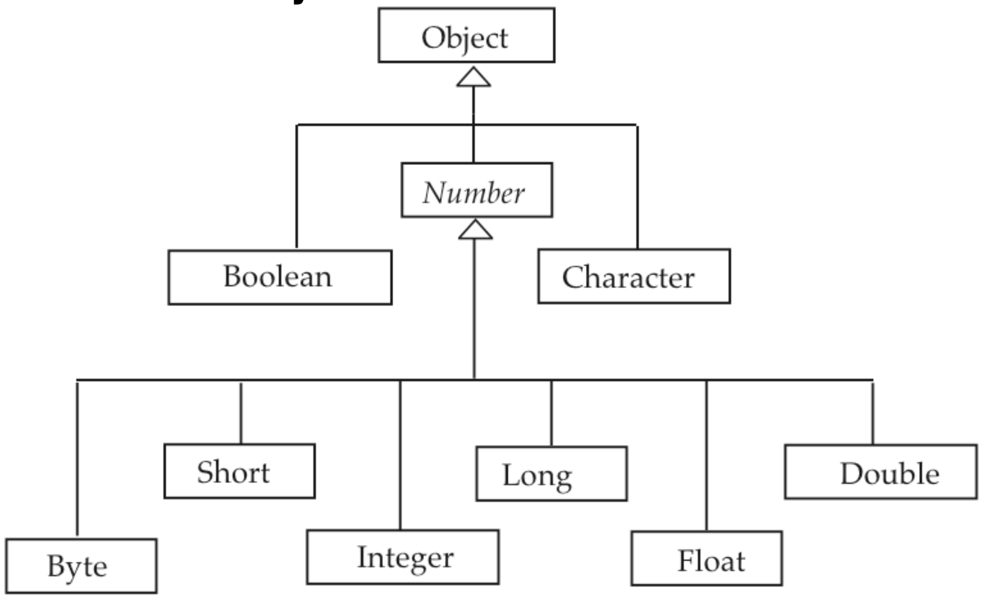
boolean, **char**,
byte, **short**, **int**, **long**,
float, **double**

Wrapper classes define objects, each of which encapsulates one unchangeable primitive value.

Boolean, Character,
Byte , Short, Integer, Long,
Float, Double

Wrapper classes found in package *java.lang*, which is visible to your program by default.

Java Class Hierarchy



Note: Number is an abstract class.

Every object in Java is an Object.

Purposes of Wrapper Classes

- Home for constants associated with the type.
- Home for class methods defined on values of the type.
- A way to treat primitive values as objects so they can be used by methods and classes that expect objects.

Using Wrappers

```
public class Wrap
{
    public static void main(String [] args)
    {
        Object x = new Character('A');
        checkOut(x);

        x = new Double(99.999);
        checkOut(x);

        x = new Boolean(false);
        checkOut(x);

        x = new Integer(1109);
        checkOut(x);
    }
```

```
   static void checkOut(Object ob)
   {
      if (ob instanceof Double)
      {
         Double d = (Double)ob;
         System.out.println("double value = " + d.doubleValue());
      }
      else if (ob instanceof Boolean)
      {
         Boolean b = (Boolean)ob;
         System.out.println("boolean value = " + b.booleanValue());
      }
      else if (ob instanceof Character)
      {
         Character c = (Character)ob;
         System.out.println("char value = " + c.charValue());
      }
      else if (ob instanceof Integer)
         System.out.println("int value = " + ((Integer)ob).intValue());
   }
}
```

Output:

```
char value = A
double value = 99.999
boolean value = false
int value = 1109
```

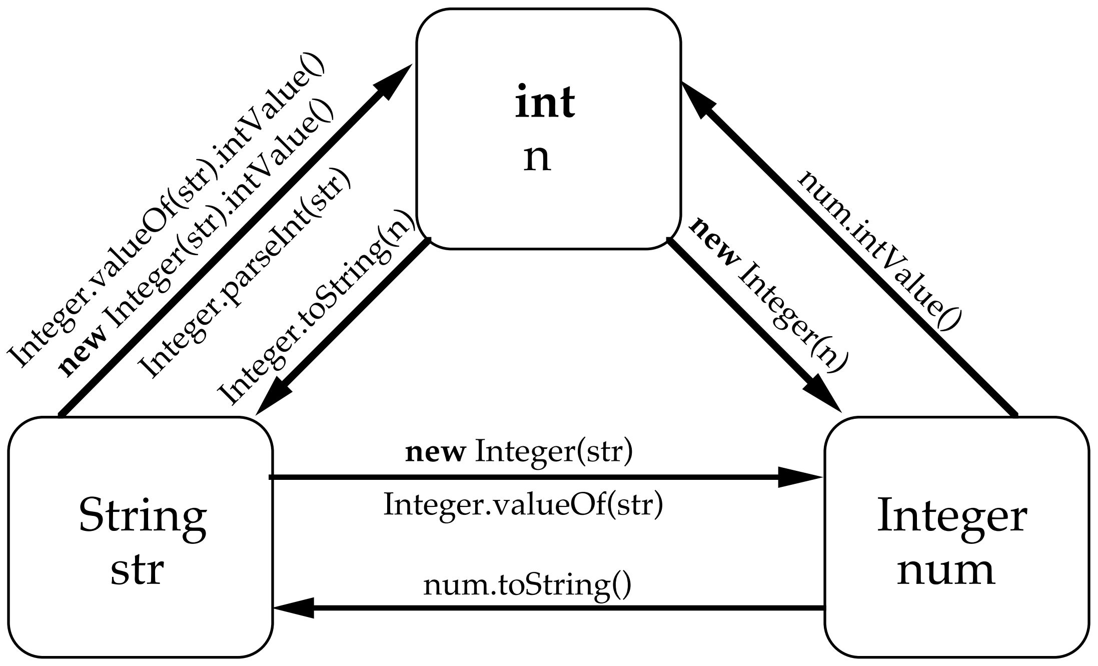
Integer Class

```
public final class java.lang.Integer extends java.lang.Number
{
// Fields
    public final static int MAX_VALUE;
    public final static int MIN_VALUE;

// Constructors
    public Integer (int value);
    public Integer (String s);

// Instance methods
    public byte byteValue ();
    public short shortValue ();
    public int intValue ();
    public long longValue ();
    public float floatValue ();
    public double doubleValue ();
    public boolean equals (Object obj);
    public String toString ();

// Class methods
    public static int parseInt (String s);
    public static String toString (int i);
    public static Integer valueOf (String s);
}
```

Examples

```
  Integer m = new Integer("77");
  Integer n = new Integer(77);

  m.intValue() and n.intValue()     both return 77
```

m.toString() and n.toString()	both return "77"
m == n	returns false
m.equals(n)	returns true
Integer.MAX_VALUE	returns 2147483647
Integer.valueOf("13")	returns an Integer object
Integer.valueOf("13").intValue()	returns 13
Integer.parseInt("13")	returns 13
Double.valueOf("4.5").doubleValue()	returns 4.5, a double
Double.parseDouble("25.7")	returns 25.7, a double



Exceptions

What happens if we call

```
Integer.parseInt("12e4")?
```

The string cannot be parsed as an integer, so a runtime error occurs:

```
java.lang.NumberFormatException: 12e4
```

In Java, runtime errors are called exceptions.
When such an error occurs, we say an exception has been thrown.
Java has an extensive facility for handling exceptions, which are themselves objects.

Command Line Arguments

Java provides a mechanism for supplying command line arguments when a program is started.

These arguments are treated as instances of String and are placed in the array specified as the formal parameter for the main method.

Consider the following program:

```
public class Args
{
    public static void main(String [] args)
    {
        System.out.println("Number of args = " + args.length);

        for (int k=0; k<args.length; k++)
           System.out.println("Argument " + k + " = " + args[k]);
    }
}
```

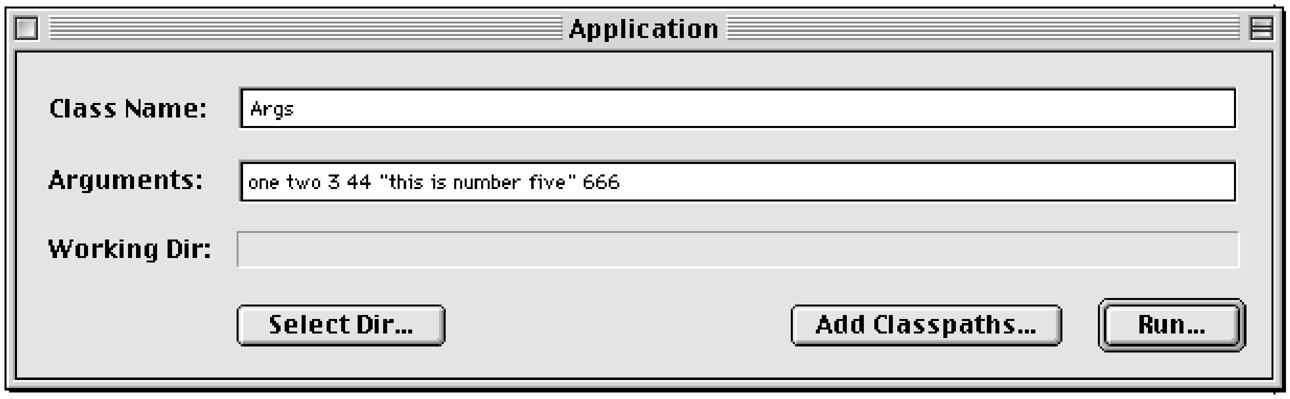
In a command line system, such as Unix, the program Args can be invoked using the format:

```
% java Args one two 3 44 "this is number five" 666
```

Output

```
 Number of args = 6
Argument 0 = one
Argument 1 = two
Argument 2 = 3
Argument 3 = 44
Argument 4 = this is number five
Argument 5 = 666
```

In early versions of CodeWarrior, the arguments were supplied in the Application window, as shown below:



Class Name: Args

Arguments: one two 3 44 "this is number five" 666

Working Dir:

Kinds of Identifiers

1. Module (collection of definitions) identifiers

- Class
- Package
- Interface

2. Method identifiers

- Class methods
- Instance methods
- Constructors

3. Variable identifiers

- Class variables
- Instance variables
- Local variables
- Method parameters (constructor parameters)
- Exception handler parameters

Scope (Visibility) of Identifiers

1. **Class scope**

- The entire class (at least)

  Class methods
  Instance methods (and constructors)
  Class variables (including constants)
  Instance variables

- Order of these declarations makes no difference with one exception.

```
int m = n;
int n = 5;          // is illegal (also for static)
```

- These variables are automatically initialized to default values unless they are **final** (constants).

2. **Block scope** (method scope)

- Method parameters: Entire method.
- Local variables: Point of declaration to the end of the block (method).

  Block command:

```
{
    // declarations and commands
}
```

- Local variable declarations may hide class and instance variables, but not other local variables: local variables may not be hidden in inner scopes.

Inner Classes

All of the classes defined by us so far have been “top-level” classes, meaning that they lie at the top level of packages and files.

As a consequence, names of classes and their members must be visible to all classes in a package or to none at all.

Inner classes allow us to restrict visibility by making a class “local” to another class or to a method.

An inner class may be defined inside:

1. Another class.
2. A method (a block).
3. An expression (using an anonymous class).

Each object created from an inner class is “associated” with an object created from the class in which it is defined.

Inner classes may not have static members.

The name of an inner class must be different from the names of all of its enclosing classes.

Example

Define a class, Chief, whose objects have an **int** value as their state with an instance method that can alter the value.

Define a second class, Helper, whose objects are responsible for printing the values stored in Chief objects. Each instance of Helper contains a reference to the Chief object that it is responsible for.

A Helper object can be created either by using

- A Helper constructor directly or
- An instance method *mkHelper* in Chief (a factory method).

The system is tested by a main method in a third class, called ChiefTest, that creates two Chief objects and two associated Helper objects.

```
class Chief
{
    int value;          // should be private

    Chief(int n) {  value = n;  }

    void alter()  {  value = 2*value+10;  }

    Helper mkHelper()           // must be an instance method
    {  return new Helper(this);  }
}
```

```
class Helper
{
    private Chief chf;

    Helper(Chief c)
    {  chf = c;  }

    void showValue()
    {  System.out.println(chf.value);  }
}


public class ChiefTest
{
    public static void main(String [] args)
    {
        Chief a = new Chief(5);
        Chief b = new Chief(8);

        Helper ahelp = a.mkHelper();

        Helper bhelp = new Helper(b);

        ahelp.showValue();    bhelp.showValue();

        a.alter();            b.alter();

        ahelp.showValue();    bhelp.showValue();
    }
}
```

Output

```
    5
    8
    20
    26
```

Drawbacks

- The state variable in Chief cannot be private.
- Each Helper object needs to maintain an explicit reference to the Chief object that it is associated with.

The following version solves the same problem using an inner class for Helper.

```
class Chief
{
    private int value;
//---- Inner Class --------------------------
    class Helper
    {
        void showValue()
        {  System.out.println(value);  }
    }
//----------------------------------------------
    Chief(int n)  {  value = n;  }
    void alter()  {  value = 2*value+10;  }

    Helper mkHelper()            // must be an instance method
    {  return new Helper();  }
               // Return an instance of Helper whose
               // enclosing instance will be this.
}
```

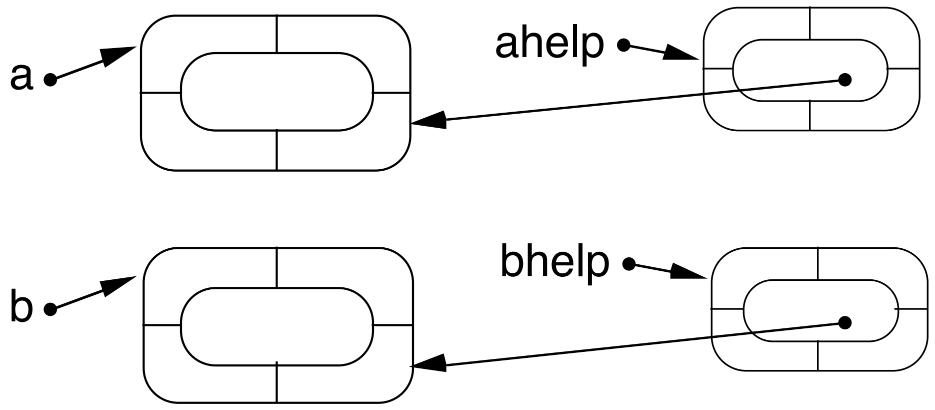
```
public class ChiefTestInner
{
    public static void main(String [] args)
    {
        Chief a = new Chief(5);
        Chief b = new Chief(8);

        Helper ahelp = a.mkHelper();
                // Get a Helper object enclosed by a's object.

        Helper bhelp = b.new Helper();
                // Get a Helper object enclosed by b's object.
                // Note the new syntax for new.

        ahelp.showValue();    bhelp.showValue();

        a.alter();            b.alter();

        ahelp.showValue();    bhelp.showValue();
    }
}
```

Each Helper object has an implicit reference to the object for which it was created.

Writing "b.**new** Helper()" builds a Helper object associated with b.

Scoping Properties

- All identifiers in Chief are visible inside of Helper.
- All identifiers in Helper are visible inside of Chief.
- Helper cannot be accessed from outside of Chief because it is declared private.



Anonymous Classes

Anonymous classes (classes with no name) may appear in expressions that expect a class item following a **new** operator.

Syntax of a **new** expression with an anonymous class:

> **new** SuperType(*constructor parameters*)
> { instance *methods and instance variables of inner class* }

The SuperType may be:

1. A class:
   Then the anonymous class extends SuperType, and the parameters are for a call to a superclass constructor.

2. An interface:
   Then the anonymous class implements SuperType (and extends Object), and there can be no constructor parameters.

Observation

Since an anonymous class has no name, it cannot have an explicit constructor—only the default constructor provided by Java is available.

Note Difference

```
Domino d = new Domino(2,2,true);

Domino e = new Domino(2,2,true)
{    public String toString()
     {  return getLow() + "--" + getHigh();  }
};         // semicolon required at end of a declaration
```

The addition of inner classes to Java requires no change in the Java virtual machine.

The compiler translates all inner classes into regular Java classes.

Java also has static inner classes, called nested classes.

These classes are not associated with objects and are of little use in OOP.

Using an Anonymous Classes

Suppose we want to create another implementation of the interface Printable. This implementation will only be used once.

Change the code as follows.

```
Printable [] pt = new Printable[3];

pt[0] = fi;

pt[1] = si;

pt[2] = new Printable()
        {
            public void printNum(int n)
            {
                System.out.println(">>>" + n + "<<<");
            }
        };


for (int k=0; k<pt.length; k++)
    pt[k].printNum(20*k+5);
```

Output

```
claude prints 5
Number99 prints 25
>>>45<<<
```

Member Visibility

Use the term "member" for a variable or a method in a class.

Visibility depends on the modifier on the class as well as the modifier on the member.

	Same class	**Other class same package**	**Subclass other package**	**Any class**
public class				
public member	Yes	Yes	Yes	Yes
protected member	Yes	Yes	Yes	
member	Yes	Yes		
private member	Yes			
class				
public member	Yes	Yes		
protected member	Yes	Yes		
member	Yes	Yes		
private member	Yes			
private class (an inner class)				**Enclosing class**
public member	Yes			Yes
protected member	Yes			Yes
member	Yes			Yes
private member	Yes			Yes