Visibility of Class Members

Class members may be declared as *public*, *private* or *protected*.

Public members may be accessed from outside a class.

Private members of a class may be accessed only from with the class itself.

Protected members may be accessed only from with the class itself or from within one of its subclasses.

Members nor marked public, private or protected are shared at the package level—similar to C++'s friend mechanism.

Example:

```
class Customer {
   int id;
   private int pinCode;
}
Customer me = new Customer();
me.id = 1234; //OK
me.pinCode = 7777;
 //Compile-time error
```

In a class, a special method, **`main`**, declared as

```
static public void
    main(String[] args)
```

is automatically executed when a class is run.

`main` is very useful as a "test driver" for auxiliary and library classes.

Final Members

A field may be declared *final* making it effectively a constant.

```
class Point {
    int x,y;
    static final Point origin
        = new Point(0,0);
    Point(int xin, int yin) {
       x = xin; y = yin;
    }
}
```

Final fields may be used to create constants within a class:

```
class Card {
  final static int Clubs = 1;
  final static int Diamonds = 2;
  final static int Hearts = 3;
  final static int Spades = 4;
  int suit = Spades;
}
```

Inside a class suit names are available for use without qualification. E.g.,

```
int suit = Spades;
```

Outside a class, the field names must be qualified using the class name:

```
Card c = new Card();
c.suit = Card.Clubs;
```

Methods may also be marked as final. This forbids redeclaration in a subclass, allowing a more efficient implementation. Security may also be improved if a key method is known to be unchangeable.

Java Arrays

In Java, arrays are implemented as a special kind of class. Arrays of primitive types are implemented as an object that contains a block of values within it. Arrays of objects are implemented as an object that contains a block of object *references* within it. Allocating an array of objects *does not* allocate the objects themselves. Hence within an array of objects, some positions may reference actual objects while other may contain **`null`** (this can be advantageous) if objects are large.

Multi-dimensional arrays are arrays of arrays. Arrays within an array *need not* all have the same size.

Hence we may see

```
int[][] TwoDim = new int[3][];
TwoDim[0] = new int[1];
TwoDim[1] = new int[2];
TwoDim[2] = new int[3];
```

The size of an array is part of its value; not its type.

Thus

```
int [] A = new int[10];
int [] B = new int[5];
A = B;
```

is valid.

Pascal showed that making an array's size part of its type is undesirable. (Why?)

Still, forcing an array to have a fixed size can be necessary (e.g., an array indexed by months). (How do we simulate a fixed-size array?).

Subclassing in Java

When a new class is defined in terms of an existing class, the new class *extends* the existing class. The new class *inherits* all public and protected members of its *parent* (or *base*) class.

The new class may add new methods or fields. It may also redefine inherited methods or fields.

```
class Point {
    int x,y;
     Point(int xin, int yin) {
       x = xin; y = yin;
    }
    static float dist(
      Point P1, Point P2) {
     return (float) Math.sqrt(
      (P1.x-P2.x)*(P1.x-P2.x)+
      (P1.y-P2.y)*(P1.y-P2.y));
    }
}
```

```
class Point3 extends Point {
    int z;
     Point3(int xin, int yin,
            int zin) {
       super(xin,yin); z=zin;
    }
    static float dist(
      Point3 P1, Point3 P2) {
      float d=Point.dist(P1,P2);
     return (float) Math.sqrt(
      (P1.z-P2.z)*(P1.z-P2.z)+
        d*d);
    }
}
```

Note that although **`Point3`** redefines **`dist`**, the old definition of **`dist`** is still available by using the parent class as a qualifier (**`Point.dist`**).

The same is true for fields that are hidden when a field in a parent is redeclared.

Non-static methods are automatically *virtual*: a redefined method is automatically used in all inherited methods including those defined in parent classes that think they are using an earlier definition of the class.

Example:

```
class C {
  void DoIt()(PrintIt();}
  void PrintIt()
     {println("C rules!");}
}
class D extends C {
  void PrintIt()
      {println("D rules!");}
  void TestIt() {DoIt();}
}
D dvar = new D();
dvar.TestIt();
D rules! is printed.
```

Static methods in Java are *not* virtual (this can make them easier to implement efficiently).

Abstract Classes and Methods

Sometimes a Java class is not meant to be used by itself because it is intentionally incomplete.

Rather, the class is meant to be starting point for the creation (via subclassing) of more complete classes.

Such classes are *abstract*.

Example:

```
abstract class Shape {
  Point location;
}
class Circle extends Shape {
  float radius;
}
```

Methods can also be made abstract to indicate that their actual definition will appear in subclasses:

```
abstract class Shape {
  Point location;
  abstract float area();
}
class Circle extends Shape {
  float radius;
  float area(){
   return Math.pi*radius*radius;
  }
}
```

Subtyping and Inheritance

We can use a subtyping mechanism, as found in C++ or Java, for two different purposes:

- We may wish to inherit the actual implementations of classes and members to use as the basis of a more complete or extended class. To inherit an implementation, we say a given class "extends" an existing class:

```
class Derived extends Base
  { ... };
```

Class **`Derived`** contains all of the members of **`Base`** *plus* any others it cares to add.

- We may wish to inherit an *interface*—a set of method names and values that will be available for use.
  To inherit (or claim) an interface, we use a Java interface definition.
  An interface doesn't implement anything; rather, it gives a name to a set of operations or values that may be available within one or more classes.

Why are Interfaces Important?

Many classes, although very different, share a common subset of values or operations. We may be willing to use any such class as long as only interface values or operations are used.

For example, many objects can be ordered (or at least partially-ordered) using a “less than” operation.

If we always implement less than the same way, for example,

```
boolean lessThan(Object o1,
                 Object o2);
```

then we can create an interface that admits all classes that know about the **`lessThan`** function:

```
interface Compare {
 boolean lessThan(Object o1,
                  Object o2);
}
```

Now different classes can each implement the **`Compare`** interface, proclaiming to the world that they know how to compare objects of the class they define:

```
class IntCompare implements Compare {
 public boolean lessThan(Object i1,
                         Object i2){
 return ((Integer)i1).intValue() <
        ((Integer)i2).intValue();}
}
class StringCompare implements
  Compare {
 public boolean lessThan(Object i1,
                         Object i2){
 return
((String)i1).compareTo((String)i2)<0;
}}
```

The advantage of using interfaces is that we can now define a method or class that only depends on the given interface, and which will accept *any* type that implements that interface.

```
class PrintCompare {
  public  static void printAns(
   Object v1, Object v2, Compare c){
    System.out.println(
     v1.toString() + " < " +
     v2.toString() + " is " +
     new Boolean(c.lessThan(v1,v2))
       .toString());
} }
class Test {
  public static void
    main(String args[]){
     Integer i1 = new Integer(2);
     Integer i2 = new Integer(1);
     PrintCompare.printAns(
       i1,i2,new IntCompare());
     String s2 = "abcdef";
     String s1 = "xyzaa";
     PrintCompare.printAns(
      s1,s2,new StringCompare());}}
```

Since classes may have many methods and modes of use or operation, a given class may implement many different interfaces. For example, many classes support the **`Clonable`** interface, which states that objects of the class may be duplicated (cloned).

Exceptions in Java

Java provides a fairly elaborate exception handling mechanism based on the throw-catch model.

All exceptions are objects, required to be a subclass of **`Throwable`**.

Class **`Throwable`** has two subclasses, **`Exception`** and **`Error`**. Class **`Exception`** has a subclass **`RuntimeException`**.

Exceptions may be explicitly thrown (using a **`throw`** statement) or they may be implicitly thrown as the result of a run-time error.

For example, an **`ArithmeticException`** is thrown for certain run-time arithmetic errors, like division by zero.

Unlike other languages, Java divides exceptions into two general classes: *checked* and *unchecked*.

A checked exception *must* either be caught (using a try-catch block) or propagated (by marking a method as throwing the exception).

This means that checked exceptions cannot be ignored—you must be prepared to catch them or you must "advertise" to your callers that you may throw an exception back to them.

Unchecked exceptions need not be caught or marked as potentially thrown. This makes exception handling for such exceptions optional. Unchecked exceptions are typically those that might occur so

often (like **`NullPointerException`** or **`ArithmeticException`**) that forced checks could unnecessarily clutter a program without significant benefit.

How are checked and unchecked exceptions distinguished?

- Any exception that is a member (or subclass) of **`Error`** or **`RuntimeException`** is unchecked.
- All other exceptions must be checked.

Exceptions are propagated dynamically:

- When an exception is thrown (explicitly or implicitly) the innermost try-catch block that can "catch" the exception is selected, and

the catch block that matches the exception is executed.

- A catch block "catches" a given exception if the class of the exception is the same as the class used in the catch. An exception that is a subclass of the catch's exception class will also be caught.
  Thus an catch that handles class **`Throwable`** catches *all* exceptions.
- If no catch can handle the exception in the current method, a return to the method's caller is forced, and the exception is rethrown from the point of call.
- This process is repeated until a catch that can handle the exception is found or until we force a return from the main method.

- If a return from the main method is forced, no handler exists. A run-time error message is printed ("Uncaught exception") and execution is terminated.
- One of the limitations of Java's exception mechanism (and similar mechanisms found in other languages) is that there is no "retry" mechanism. Once an exception is thrown, we never go back to the point where the exception occurred. This is why Scheme's call/cc mechanism is considered so special and unique.

Example:

```
class badValue extends Exception{
  float value;
  badValue(float f) {value=f;} }

float sqrt(float val)
    throws badValue {
  if (val < 0.0)
     throw new badValue(val);
  else return
       (float) Math.sqrt(val); }

try {
  System.out.println(
    "Ans = " + sqrt(-123.0));
} catch (badValue b) {
  System.out.println(
    "Can't take sqrt of "+b)
}
```