

Java & Object-Oriented Programming

Java is a fairly new and very popular programming language designed to support secure, platform-independent programming.

It is a good alternative to C or C++, trading a bit of efficiency for easier programming, debugging and maintenance.

Java is routinely interpreted (at the byte-code level), making it significantly slower than compiled C or C++. However true Java compilers exist, and are becoming more widespread. (IBM's Jalapeno project is a good example). When compiled, Java's execution speed is close to that of C or C++.

Basic Notions

In Java data is either primitive or an object (an instance of some class).

All code is written inside classes, so Java programming consists of writing classes.

Primitive data types are quite close to those of C or C++:

boolean	(not a numeric type)
char	(Unicode, 16 bits)
byte	
short	
int	
long	(64 bits)
float	
double	

Objects

- All Java objects are instances of classes.
- All objects are heap-allocated, with automatic garbage collection.
- A reference to an object, in a variable, parameter or another object, is actually a pointer to some object allocated within the heap.
- No explicit pointer manipulation operations (like * or -> or ++) are needed or allowed.
- Example:

```
class Point {int x,y;}
Point data = new Point();
```

- Declaring an object reference (like class **Point**) *does not* automatically allocate space for an object. The reference is initialized to **null** unless an explicit initializer is included.
- Fields are accessed just as they are in C: **data.x** references field **x** in object **data**.
- Object references are automatically checked for validity (null or non-null). Hence

```
data.x = 0;
```

forces a run-time exception if **data** contains **null** rather than a valid object reference.

- Java makes it *impossible* for an object reference to access an illegal address. A reference is either `null` or a pointer to a valid, type-correct object in the heap. (This makes Java programs far more secure and reliable than C or C++ programs).

Class Members

Classes contain members. Class members are either fields (data) or methods (functions).

Example:

```
class Point {
    int x,y;
    void clear() {x=0; y=0;}
}
Point d = new Point();
d.clear();
```

A special method is a *constructor*.

A constructor has no result type. It is used only to define the initialization of an object after the object has been created.

Constructors may be *overloaded*.

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

Static Members

Class members may be *static*.

A static member is allocated only once—for all instances of the class.

Ordinary members (called *instance* members) apply only to a particular class instance (i.e., only one object created from the class definition).

```
class Point {
    int x,y;
    static int howMany = 0;
    Point() {x=0; y=0;
        howMany++;}
    static void reset() {
        howMany = 0;
    }
}
```

Static member functions (methods) may not access non-static data.
(Why?)

Static members are accessed using a class name rather than the name of an object reference.

For example,

```
Point.reset();
```

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 within the class itself.

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

Members not 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?).