Overloading and Ad-hoc Polymorphism

Classes usually allow overloading of method names, if only to support multiple constructors.

That is, more than one method definition with the same name is allowed within a class, as long as the method definitions differ in the number and/or types of the parameters they take.

For example,

```cpp
class MyClass {
    int f(int i) { ... }
    int f(float g) { ... }
    int f(int i, int j) { ... }
}
```
Overloading is sometimes called “ad hoc” polymorphism, because, to the programmer, it appears that one method can take a variety of different parameter types. This isn’t true polymorphism because the methods have different bodies; there is no sharing of one definition among different parameter types. There is no guarantee that the different definitions do the same thing, even though they share a common name.
Issues in Overloading

Though many languages allow overloading, few allow overloaded methods to differ only on their result types. (Neither C++ nor Java allow this kind of overloading, though Ada does). For example,

```cpp
class MyClass {
    int f() { ... }
    float f() { ... }
}
```

is illegal. This is unfortunate, since methods with the same name and parameters, but different result types, could be used to automatically convert result values to the type demanded by the context of call.
Why is this form of overloading usually disallowed?

It’s because overload resolution (deciding which definition to use) becomes much harder. Consider

class MyClass {
    int   f(int i, int j) { ... }  
    float f(float i, float j) { ... }  
    float f(int i, int j) { ... }  
}

in

    int a = f( f(1,2), f(3,4) );
which definitions of $f$ do we use in each of the three calls? Getting the correctly answer can be tricky, though solution algorithms do exist.
Operator Overloading

Some languages, like C++ and C#, allow operators to be overloaded. You may add new definitions to existing operators, and use them on your own types. For example,

```c++
class MyClass {
    int i;
    public:
        int operator+(int j) {
            return i+j; }
}
MyClass c;
int i = c+10;
int j = c.operator+(10);
int k = 10+c;  // Illegal!
```
The expression $10+c$ is illegal because there is no definition of + for the types int and MyClass&. We can create one by using C++’s friend mechanism to insert a definition into MyClass that will have access to MyClass’s private data:

```cpp
class MyClass {
    int i;

group public:

    int operator+(int j) {
        return i+j;
    }

    friend int operator+ (int j, MyClass& v){
        return j+v.i;
    }
}

MyClass c;
int k = 10+c;  // Now OK!
```
C++ limits operator overloading to existing predefined operators. A few languages, like Algol 68 (a successor to Algol 60, developed in 1968), allow programmers to define brand new operators.

In addition to defining the operator itself, it is also necessary to specify the operator’s precedence (which operator is to be applied first) and its associativity (does the operator associate from left to right, or right to left, or not at all). Given this extra detail, it is possible to specify something like

```c++
op +++ prec = 8;
int op +++(int& i, int& j) {
    return (i++)+(j++);
}
```

(Why is `int&` used as the parameter type rather than `int`?)
Parameter Binding

Almost all programming languages have some notion of binding an actual parameter (provided at the point of call) to a formal parameter (used in the body of a subprogram).

There are many different, and inequivalent, methods of parameter binding. Exactly which is used depends upon the programming language in question.

Parameter Binding Modes include:

- Value: The formal parameter represents a local variable initialized to the value of the corresponding actual parameter.
• Result: The formal parameter represents a local variable. Its final value, at the point of return, is copied into the corresponding actual parameter.

• Value/Result: A combination of the value and results modes. The formal parameter is a local variable initialized to the value of the corresponding actual parameter. The formal’s final value, at the point of return, is copied into the corresponding actual parameter.

• Reference: The formal parameter is a pointer to the corresponding actual parameter. All references to the formal parameter indirectly access the corresponding actual parameter through the pointer.
• **Name:** The formal parameter represents a block of code (sometimes called a thunk) that is evaluated to obtain the value or address of the corresponding actual parameter. Each reference to the formal parameter causes the thunk to be reevaluated.

• **Readonly (sometimes called Const):** Only reads of the formal parameter are allowed. Either a copy of the actual parameter’s value, or its address, may be used.
What Parameter Modes do Programming Languages Use?

- C: Value mode except for arrays which pass a pointer to the start of the array.
- C++: Allows reference as well as value modes. E.g.,
  \[ \text{int } f(\text{int } a, \text{ int } & b) \]
- C#: Allows result (out) as well as reference and value modes. E.g.,
  \[ \text{int } g(\text{int } a, \text{ out int } b) \]
- Java: Scalar types (\text{int}, \text{float}, \text{char}, etc.) are passed by value; objects are passed by reference (references to objects are passed by value).
- Fortran: Reference (even for constants!)
- Ada: Value/result, reference, and readonly are used.
Example

```c
void p(value int a,
       reference int b,
       name int c) {
    a=1; b=2; print(c)
}
int i=3, j=3, k[10][10];
p(i,j,k[i][j]);
```

What element of $k$ is printed?

- The assignment to $a$ does not affect $i$, since $a$ is a value parameter.
- The assignment to $b$ does affect $j$, since $b$ is a reference parameter.
- $c$ is a name parameter, so it is evaluated whenever it is used. In the print statement $k[i][j]$ is printed. At that point $i=3$ and $j=2$, so $k[3][2]$ is printed.