Overloading allows different methods to have same name, but different signatures where signature can differ by number of input parameters or type of input parameters or both. Overloading is related to compile time (or static) polymorphism

|  |
| --- |
| // Java program to demonstrate working of method  // overloading in Java.    public class Sum {        // Overloaded sum(). This sum takes two int parameters      public int sum(int x, int y) {          return (x + y);      }        // Overloaded sum(). This sum takes three int parameters      public int sum(int x, int y, int z) {           return (x + y + z);      }        // Overloaded sum(). This sum takes two double parameters      public double sum(double x, double y) {           return (x + y);      }        // Driver code      public static void main(String args[]) {          Sum s = new Sum();          System.out.println(s.sum(10, 20));          System.out.println(s.sum(10, 20, 30));          System.out.println(s.sum(10.5, 20.5));      }  } |

Output :

30

60

31.0

Example 2

// Demonstrate method overloading.   
class OverloadDemo {   
void test() {   
System.out.println("No parameters");   
}   
// Overload test for one integer parameter.   
void test(int a) {   
System.out.println("a: " + a);   
}   
// Overload test for two integer parameters.   
void test(int a, int b) {   
System.out.println("a and b: " + a + " " + b);   
}   
// overload test for a double parameter   
double test(double a) {   
System.out.println("double a: " + a);   
return a\*a;   
}   
}   
class Overload {   
public static void main(String args[]) {   
OverloadDemo ob = new OverloadDemo();   
double result;   
// call all versions of test()   
ob.test();   
ob.test(10);   
ob.test(10, 20);   
result = ob.test(123.2);   
System.out.println("Result of ob.test(123.2): " + result);   
}   
}

This program generates the following output:

No parameters   
a: 10   
a and b: 10 20   
double a: 123.2   
Result of ob.test(123.2): 15178.24

In order to overload a method, the argument lists of the methods must differ in either of these:  
1. Number of parameters.  
For example: This is a valid case of overloading

add(int, int)

add(int, int, int)

2. Data type of parameters.  
For example:

add(int, int)

add(int, float)

3. Sequence of Data type of parameters.  
For example:

add(int, float)

add(float, int)

**ADVERNTAGE OF METHOD OVERLOADED**

**Flexibility**

* Overloaded methods give programmers the flexibility to call a similar method for different types of data.
* Function overloading is done for code reusability, to save efforts, and also to save memory.

Overloaded methods give programmers the flexibility to call a similar method for different types of data. If you are working on a mathematics program, for example, you could use overloading to create several "multiply" classes, each of which multiplies a different number of type of argument: the simplest "multiply(int a, int b)" multiplies two integers; the more complicated method "multiply(double a, int b, int c)" multiplies one double by two integers -- you could then call "multiply" on any combination of variables that you created an overloaded method for and receive the proper result.

Constructors

* Overloading is also used on constructors to create new objects given different amounts of data.

For example, you could use overloading to create three different constructors for a "House" object with a house number, street name and color variables.

The simplest constructor "House()" takes no arguments and creates a house with default or empty variables.

A more complex constructor, "House(int houseNumber, String streetName)," creates a house with the specified house number and street name, but a default or empty color.

The most complex constructor, "House(int houseNumber, String streetName, String color),"

creates a house with all of the specified information, leaving nothing as default. You could then create a house object based on the information currently available, with the unavailable information left blank or at default.

Ambiguous References

Overloaded methods must use different numbers or types of arguments to avoid ambiguity. If you create two methods in the same class that have the same name and accept two integers as arguments, the Java compiler will be unable to distinguish between the two, even if the input variables have different names. For example, the method "add(int a, int b)" cannot exist in the same class as the method "add(int c, int d)."

Return Types

You must define a return type for each overloaded method. Methods can have different return types -- for example, "add(int a, int b)" may return an integer, while "add(double a, double b)" returns a double. However, Java cannot distinguish between two different methods based on their return type. Therefore, "int multiply(double a, double b)" cannot exist in the same class as "double multiply(double a, double b)."

**INLINE FUNCTION**

inline function is an optimization performed by java just in time compile. Inline enhancement feature to increase the execution time of a program.

**Pros** :-   
1. It speeds up your program by avoiding function calling overhead.  
2. It save overhead of variables push/pop on the stack, when function calling happens.  
3. It save overhead of return call from a function.  
4. It increases locality of reference by utilizing instruction cache.  
5. By marking it as inline, you can put a function definition in a header file (i.e. it can be included in multiple compilation unit, without the linker complaining)  
  
**Cons** :-  
1. It increases the executable size due to code expansion.   
2. C++ inlining is resolved at compile time. Which means if you change the code of the inlined function, you would need to recompile all the code using it to make sure it will be updated  
3. When used in a header, it makes your header file larger with information which users don’t care.  
4. As mentioned above it increases the executable size, which may cause thrashing in memory. More number of page fault bringing down your program performance.  
5. Sometimes not useful for example in embedded system where large executable size is not preferred at all due to memory constraints.

The following program demonstrates this concept:

|  |
| --- |
| #include <iostream>  using namespace std;  class operation  {      int a,b,add,sub,mul;      float div;  public:      void get();      void sum();      void difference();      void product();      void division();  };  inline void operation :: get()  {      cout << "Enter first value:";      cin >> a;      cout << "Enter second value:";      cin >> b;  }    inline void operation :: sum()  {      add = a+b;      cout << "Addition of two numbers: " << a+b << "\n";  }    inline void operation :: difference()  {      sub = a-b;      cout << "Difference of two numbers: " << a-b << "\n";  }    inline void operation :: product()  {      mul = a\*b;      cout << "Product of two numbers: " << a\*b << "\n";  } |
| inline void operation ::division()  {      div=a/b;      cout<<"Division of two numbers: "<<a/b<<"\n" ;  }    int main()  {      cout << "Program using inline function\n";      operation s;      s.get();      s.sum();      s.difference();      s.product();      s.division();      return 0;  } |

Output:

Enter first value: 45

Enter second value: 15

Addition of two numbers: 60

Difference of two numbers: 30

Product of two numbers: 675

Division of two numbers: 3

class Factorial {

static int factorial( int n ) {

if (n != 0)

return n \* factorial(n-1); // recursive call

else

return 1;

}

public static void main(String[] args) {

int number = 4, result;

result = factorial(number);

System.out.println(number + " factorial = " + result);

}

}

Output

4 factorial=24

down vote

Any algorithm implemented using recursion can also be implemented using iteration.

Why *not* to use recursion

1. It is usually slower due to the overhead of maintaining the stack.
2. It usually uses more memory for the stack.

Why *to* use recursion

1. Recursion adds clarity and (sometimes) reduces the time needed to write and debug code (but doesn't necessarily reduce space requirements or speed of execution).
2. Reduces time complexity.
3. Performs better in solving problems based on tree structures.

For example, the Tower of Hanoi problem is more easily solved using recursion as opposed to iteration.