3. OVERLOADING METHODS

Method overloading is a salient feature in Object-Oriented Programming (OOP). It lets you declare the same method multiple times with different argument lists. In this guide, we are going to discuss how we can achieve method overloading in C#.

Method overloading is a form of polymorphism in OOP. Polymorphism allows objects or methods to act in different ways, according to the means in which they are used. One such manner in which the methods behave according to their argument types and number of arguments is method overloading.

Overloading happens when you have two methods with the same name but different signatures (or arguments). In a class we can implement two or more methods with the same name. Overloaded methods are differentiated based on the number and type of parameter passed as arguments to the methods. If we try to define more than one method with the same name and the same number of arguments then the compiler will throw an error.

The advantage of method overloading is that it increases code readability and maintainability. Although it is possible to have methods with the same name that perform a totally different function, it is advised that overloaded methods must have similarities in the way they perform.

Overloading Methods

It's very easy to create a class with overloaded methods, just define methods with the same name but with different argument lists.

Metod overloading can be achieved by the following:

- By changing the number of parameters in a method
- By changing the order of parameters in a method
- By using different data types for parameters

Let's look at a very common example, to find the area of any polygon.

```
public class Area {

public double area(double s) {

    double area = s * s;
```

```
    return area;    }
  public double area(double l, double b) {
    double area = l * b;
    return area;
   }
}
```

In the above code, the method area() is defined *twice*. First, it's defined with one argument to find the area of the square; and second, it's defined with two arguments length and breadth, to find the area of the rectangle.,

Invoking Overloaded Methods

To invoke the overloaded methods, call the method with the exact arguments. For example, if we want to invoke the area() method to find the area of a square, we will pass only one argument.

```
Area a = new Area();
double side = 3.3;
double square = a.area(side);
Console.WriteLine(square);
```

Output:

10.89

Similarly to find the area of the rectangle we would want to write the following,

```
Area a = new Area();
double length = 3.3;
double breadth = 4.9;
double rect = a.area(length, breadth);
```

```
Console.WriteLine(rect);
```

Output:

16.17

3.1 GENERIC SUBPROGRAMS

- A generic or polymorphic subprogram
  - takes parameters of different types on different activations, or
  - executes different code on different activations
- Overloaded subprograms offer
  - ad hoc polymorphism
- A subprogram where the type of a parameter is described by a type expression with a generic parameter
  - parametric polymorphism

Generic Subprograms in Ada

- Ada
  - types, subscript ranges, constant values, etc., can be generic in subprograms, packages

Example.

```
generic
    type element is private;
    type vector is array (INTEGER range <>)  of element;
    procedure Generic_Sort (list: in out vector);
    procedure Generic_Sort (list : in out vector)
```

```
        is temp: element;
begin
  for i in list'FIRST.. i'PRED(list'LAST) loop
        for j in i'SUCC(i)..list'LAST loop
          if list(i) > list(j) then
            temp := list(i);
            list(i) := list(j);
            list(j) := temp;
          end if;
        end loop; -- for j
  end loop; --for i
End Generic_Sort
procedure Integer_Sort is new Generic_Sort
   (element => INTEGER; vector => INTEGER_ARRAY);
```

Generic Subprograms Parameters in Ada

- Ada generics can be used for parameters that are subprograms
  - Note: the generic part is a subprogram
- Example:

```
generic with function fun(x: FLOAT) return FLOAT;
procedure integrate
  (from: in FLOAT; to: in FLOAT; result: out FLOAT) is
        x: FLOAT;
```

```
begin
        ...
        x := fun(from);
        ...
end;
integrate_fun is new integrate(fun => my_fun);
```

Parametric Polymorphism in C++

C++

-Template functions

e.g.

```
template <class Type>
        Type max(Type first, Type second) {
         return first > second ? first : second;
        }
```

C++ Template Functions

- Template functions are instantiated implicitly when
  - the function is named in a call, or
  - its address is taken with the & operator

Example:

```
template <class Type>
void generic_sort(Type list[], int len) {
   int i, j;
   Type temp;
 for (i = 0; i < len - 2; i++)
```

```
    for (j = i + 1; j < len - 1; j++) {
      if (list[i] > list[j]) {
        temp = list [i]; list[i] = list[j]; list[j] = temp;
      } //** end if
    } //** end for j
  } //** end for i
} //** end of generic_sort
…
float number_list[100];
generic_sort(number_list, 100);          // Implicit instantiation
```

Generics in Java

Java provides generic types

Syntax:

class_name<generic_class {, generic_class}*>

e.g.: class MyList<Element> extends ArrayList<Element> {…}

e.g.: class MyMap<Key,Value> extends HashMap<Key,Value> {…}

1.Types substituted when used:

e.g.: MyList<String> courses = new MyList<String>();

e.g.: MyMap<String,String> table = new MyMap<String,String>();

2.Type-checking works!

courses.add("ics313"); // ok

courses.add(313); // incorrect, 313 is not String

3.No casts needed!

String course = courses.get(0); //(String)courses.get(0)??

4.Generic types can be restricted to subclasses

class UrlList<Element extends URL> {…}

5.Generic types can be used in "for-each" clause

```
for (String course : courses) {
  System.out.println (course.toUpperCase());
}
```

6.Generic types are used in standard libraries

- Collections:
  - List, ArrayList, Map, Set

7.Primitive types can be substituted for generic types

class MyList<Element> extends ArrayList<Element> {…}

MyList<Integer> numbers = new MyList<Integer>()

numbers.add(313);

int sum = 0;

for (int number : numbers) {sum += number;}

Generics in Java: Parameters, Return Type

Generic types can be used

- as formal parameters

```
class MyList<Element> extends ArrayList<Element> {
  int occurences(Element element) {
    …
      if (element.equals(this)) sum++;
    …
  }
}
```

even as return type

```
class MyList<Element> extends ArrayList<Element> {
  Element[] toArray() {
    Element[] array = new Element[this.size()];
    …
    return array;
  }
}
```

Generic Methods in Java

- Method can be made depend on a generic type
  - Generic type precedes method's return type
  - Generic type can be used to
    - As the return type
    - To declare formal parameters
    - To declare local variables

E.g.

```
<T> T succ(T value, T [] array) {
   T element = null;
   ...
   return element;
}
```

Actual type is inferred from the call

```
Integer [] array = {1, 2, 3, 4, 5, 6};
int successor = succ(3, array);
```