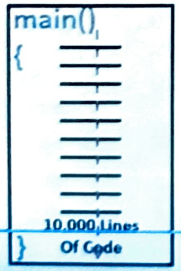
Need of Functions:

UNSTRUCTURED STYLE

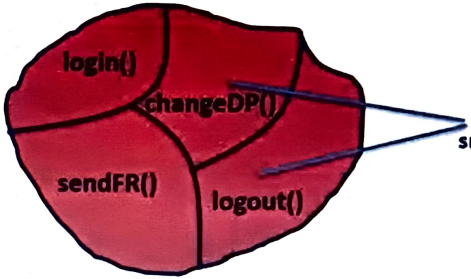
Small Task

Unstructured Programming/
Non-Structured Programming



STRUCTURED STYLE

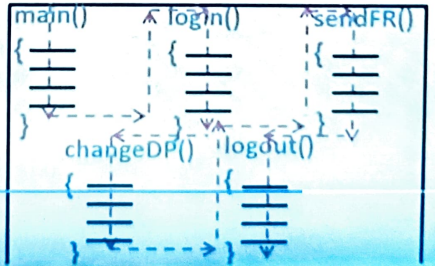
Large Task



Divide it into smaller sub-tasks

- Function
- Procedure
- Sub-program
- Module
- Sub-module
- Routine
- Sub-routine
- Method

Structured Programming/
Modular Programming/
Procedure-Oriented Programming/
Procedural Programming/
Functional Programming/...



Note:

- In unstructured style of programming, large or complex programs are coded as a single-continuous block of code. It makes the program code hardly-readable. Hence, it is difficult to modify and debug.
- Supports limited basic data types.
- Programs are executed sequentially and uses unstructured control statements such as goto.
- Suitable only for small and simple projects.

- Example: Earlier versions of BASIC, COBOL and FORTRAN.

- In structured style of programming, large or complex programs are converted into smaller blocks of code so that managing the code becomes easy. Producing readable code.
- Hence, it is easier to modify and debug. Supports variety of data types.
- Programs are executed sequentially and uses structured control statements such as if/else/for/while, etc.
- Suitable for large and complex projects.

- These blocks of code are implemented as functions.
- Therefore, a function is a block of code or collection of statements grouped together to perform a certain task or operation.
- However, the block of code only runs/executes when the function is called.

- Example: C, C++, Visual Basic, etc.

Syntax of Functions:

A Function must have the following 4 components:

1. **Name** of the function
2. **Input** to the function (*optional)
3. **Activity** performed by the function
4. **Output** from the function

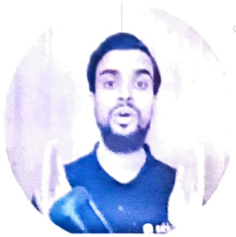
Function Definition:

```
output name(input)
{
    //activity or task
}
```

```
return_type name(parameters)
{
    //body of the function
}
```

Function Invocation:

```
name(arguments);
```



2 Major Advantages of Functions:

1. Modularity
2. Reusability

1. Modularity

Program to add & subtract 2 Numbers.

UNSTRUCTURED STYLE

```
#include <stdio.h>
void main()
{
   int a, b;
   int sum, diff;

   printf("Enter the two numbers to add:\n");
   scanf("%d %d", &a, &b);
   sum = a + b;
   printf("The sum is: %d", sum);

   printf("Enter the two numbers to subtract:\n");
   scanf("%d %d", &a, &b);
   diff = a - b;
   printf("The difference is: %d", diff);
}
```

Start

End

3 Tasks Found:

1. Addition – add()
2. Subtraction – sub()
3. Execution – main()

STRUCTURED STYLE

```
#include <stdio.h>
int add(int x, int y)
{
   int res = x + y;
   return res;
}
int sub(int x, int y)
{
   int res = x - y;
   return res;
}
void main()
{
   int a, b;
   int sum, diff;

   printf("Enter the two numbers to add:\n");
   scanf("%d %d", &a, &b);
   sum = add(a,b);
   printf("The sum is: %d", sum);

   printf("Enter the two numbers to subtract:\n");
   scanf("%d %d", &a, &b);
   diff = sub(a,b);
   printf("The difference is: %d", diff);
}
```

Start

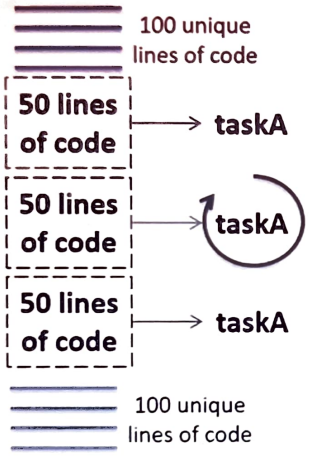
End

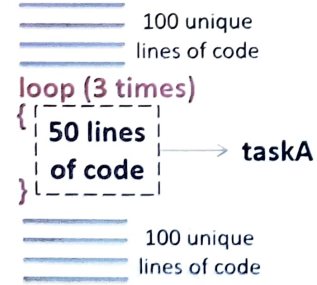
10 20

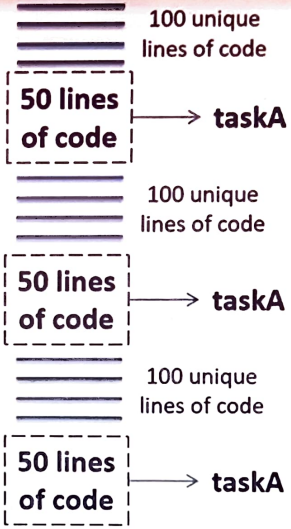
30

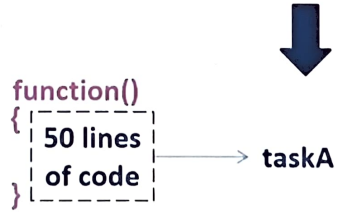
2. Reusability

Loops Vs Functions

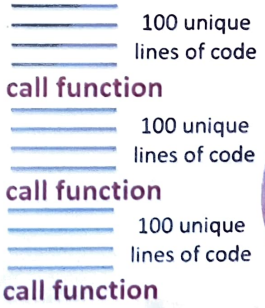








A function can be defined once and can be called any number of times by passing different arguments.



Check if a given character is Vowel or Consonant.

```
#include<stdio.h>
void main()
{
  ==========
  ==========
  char ch = 's';
  switch(ch)
  {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u': printf("Vowel");
          break;
    default: printf("Consonant");
  }
  ==========
  ==========
  ch = 'i';
  switch(ch)
  {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u': printf("Vowel");
          break;
    default: printf("Consonant");
  }
  ==========
  ==========
  ch = 'm';
  switch(ch)
  {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u': printf("Vowel");
          break;
    default: printf("Consonant");
  }
```

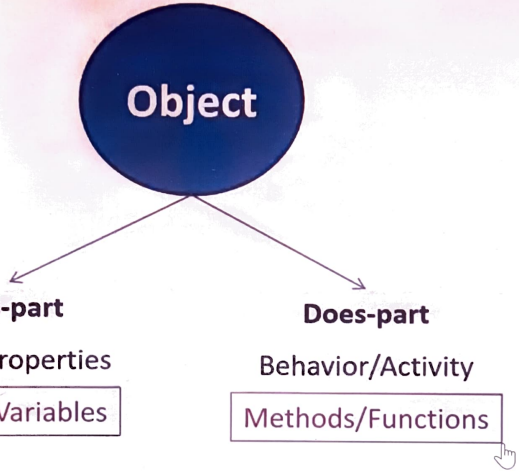
```
#include<stdio.h>

void checkAlphabet(char ch)
{
  switch(ch)
  {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u': printf("Vowel");
          break;
    default: printf("Consonant");
  }
}

void main ()
{
    ==========
    ==========
    checkAlphabet('s');
    ==========
    ==========
    checkAlphabet('i');
    ==========
    ==========
    checkAlphabet('m');
}
```

SKILLS

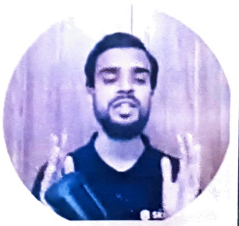
Advantages of Functions:

1. Makes the code modular
2. Enables code reusability
3. Reduces Redundancy
4. Easy to understand, manage & debug.

Methods in Java



1. A method, like a function, is a block of code or collection of statements grouped together to perform a certain task or operation. The difference is that a method is associated with an object, while a function is not.
2. In Java, methods are used to implement the behavior of the object.
3. A method must always be declared within a class.
4. A method can directly access all the instance variables of the enclosing class.

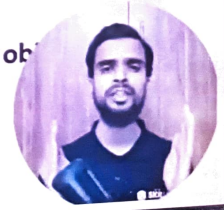


Method Definition in Java

```
modifiers  return_type method_name(parameters) exception_list
{

   //body of the method

}
```

Method declarations have six components, in order:

1. Modifiers— decides the visibility and accessibility of the method.
2. The return type—the data type of the value returned by the method, or void if the method does not return a value.
3. The method name— identifier, name of the behavior
4. The parameter list in parenthesis—a comma-separated list of input parameters, preceded by their data types.
   If there are no parameters, we must use empty parentheses.
5. An exception list— list of exceptions ducked.
6. The method body, enclosed between braces—the method's code, activity/behavior exhibited by the ob



Classification of Methods

PW SKILLS

Predefined Methods/
Inbuilt Methods/
Library Methods

Eg: println();
nextInt();
indexOf();
equals();

User-Defined Methods/
Custom Methods

Eg: eating();
writing();
adding();
guessNum();

4 Types of Methods

Type-1:

No Input

```
void add()
```

No Output

Type-2:

No Input

```
int add()
```

Output

Type-3:

Input

```
void add(int a, int b)
```

No Output

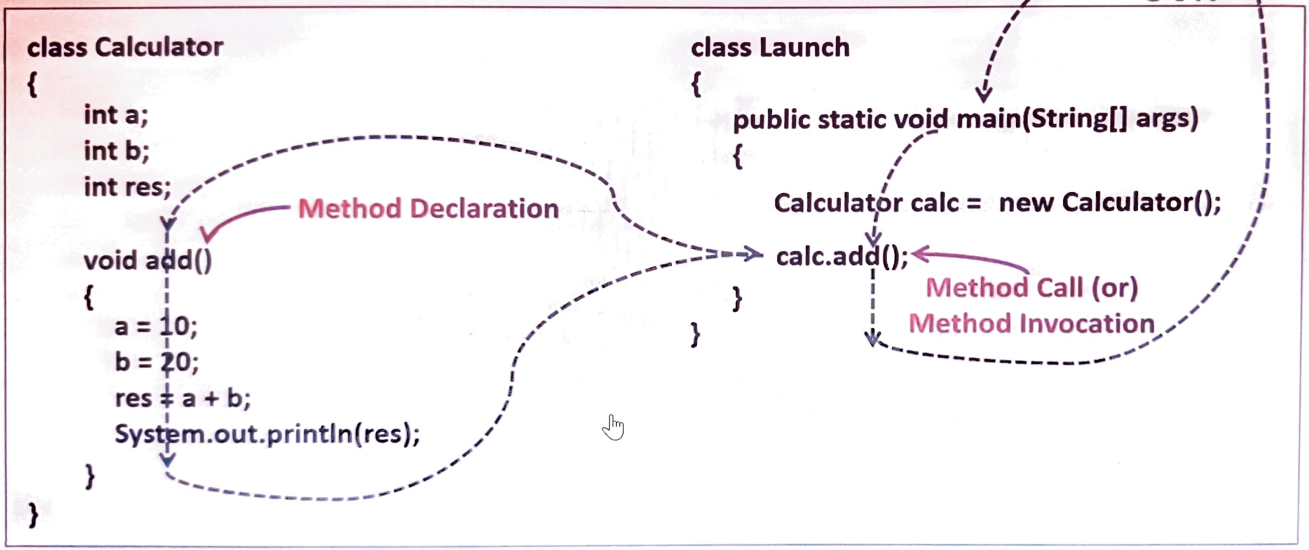
Type-4:

Input

```
int add(int a, int b)
```

Output

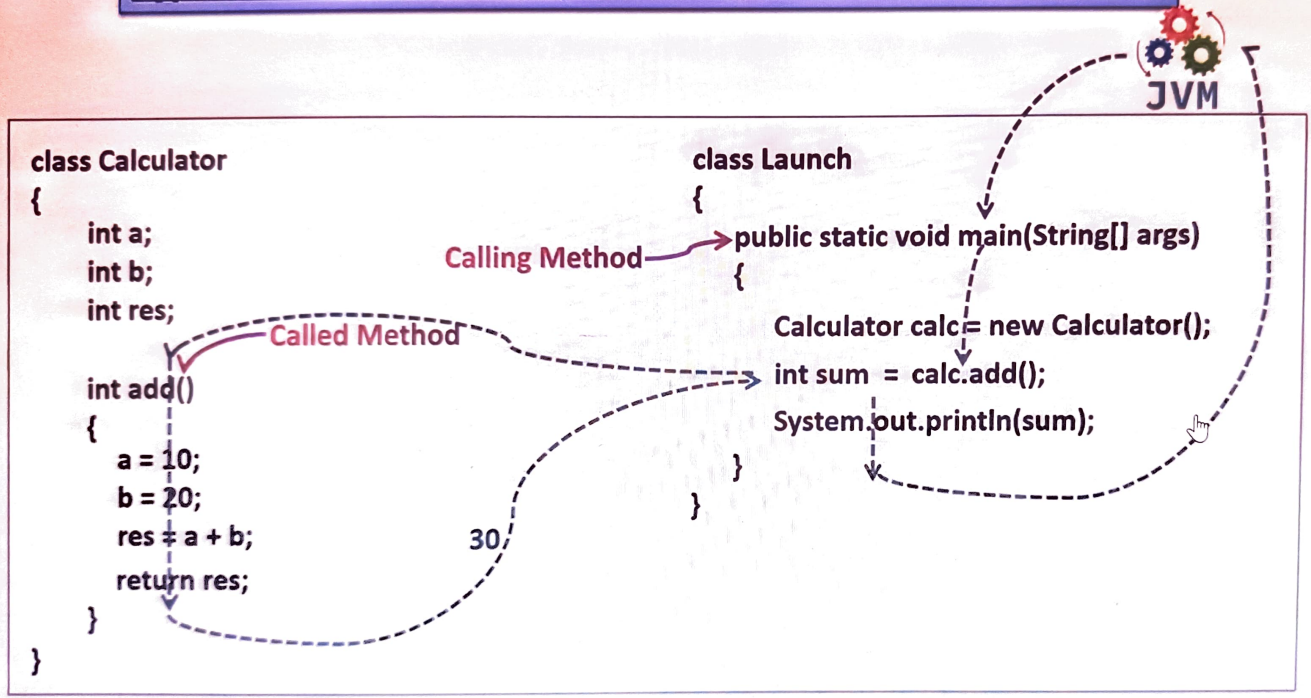
Type-1: Method with no parameters and no return value



OUTPUT:

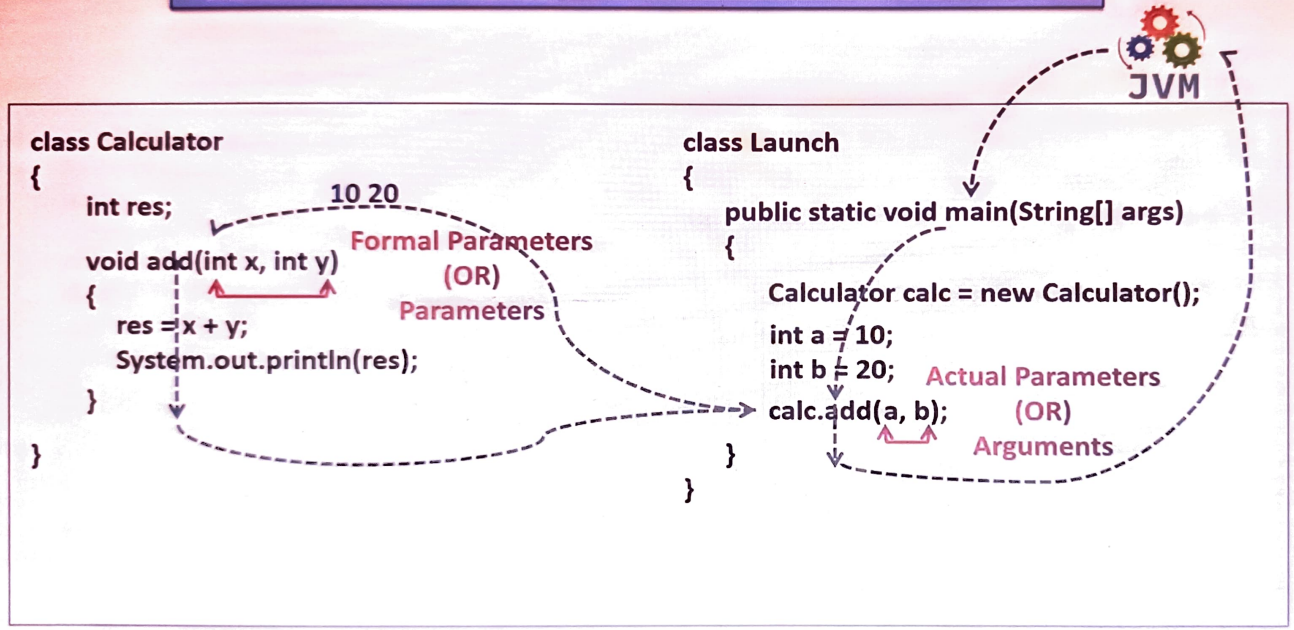
30

Type-2: Method with no parameters but with a return value



OUTPUT: 30

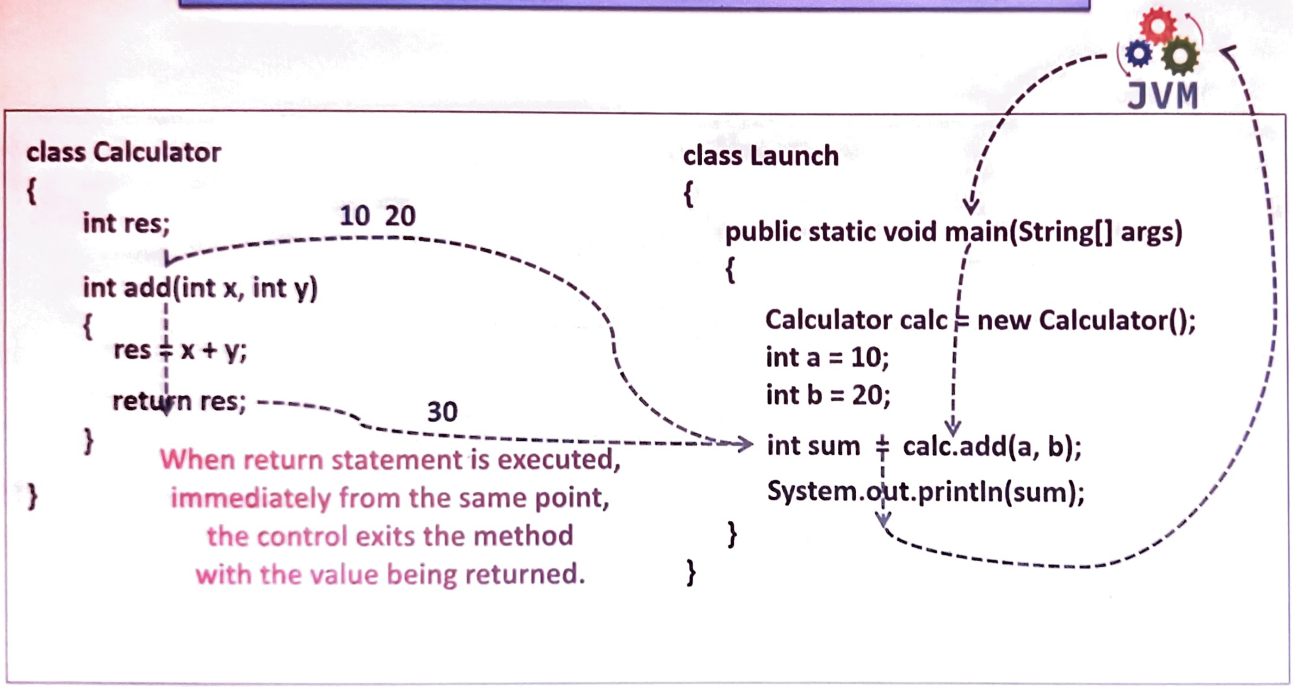
Type-3: Method with parameters and no return value



OUTPUT:

30

Type-4: Method with parameters and a return value



OUTPUT: 30