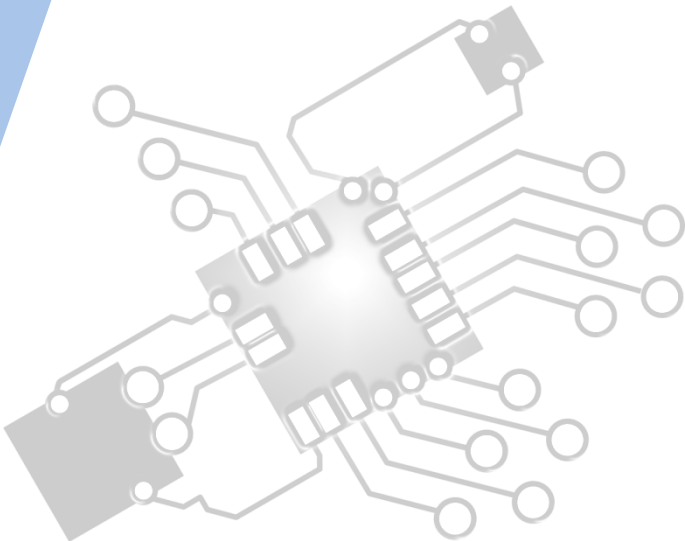




# ***Objects as a programming concept***

**IB Computer Science**



*Content developed by  
**Dartford Grammar School**  
Computer Science Department*



# HL Topics 1-7, D1-4



1: System design



2: Computer Organisation



3: Networks



4: Computational thinking



5: Abstract data structures



6: Resource management



7: Control



D: OOP

# HL & SL D.1 Overview

## D.1 Objects as a programming concept

D.1.1 Outline the general nature of an object

D.1.2 Distinguish between an object (definition, template or class) and instantiation

D.1.3 Construct unified modelling language (UML) diagrams to represent object designs

D.1.4 Interpret UML diagrams

D.1.5 Describe the process of decomposition into several related objects

D.1.6 Describe the relationships between objects for a given problem

D.1.7 Outline the need to reduce dependencies between objects in a given problem

D.1.8 Construct related objects for a given problem

D.1.9 Explain the need for different data types to represent data items

D.1.10 Describe how data items can be passed to and from actions as parameters



1: System design

2: Computer Organisation



3: Networks

4: Computational thinking



5: Abstract data structures

6: Resource management



7: Control

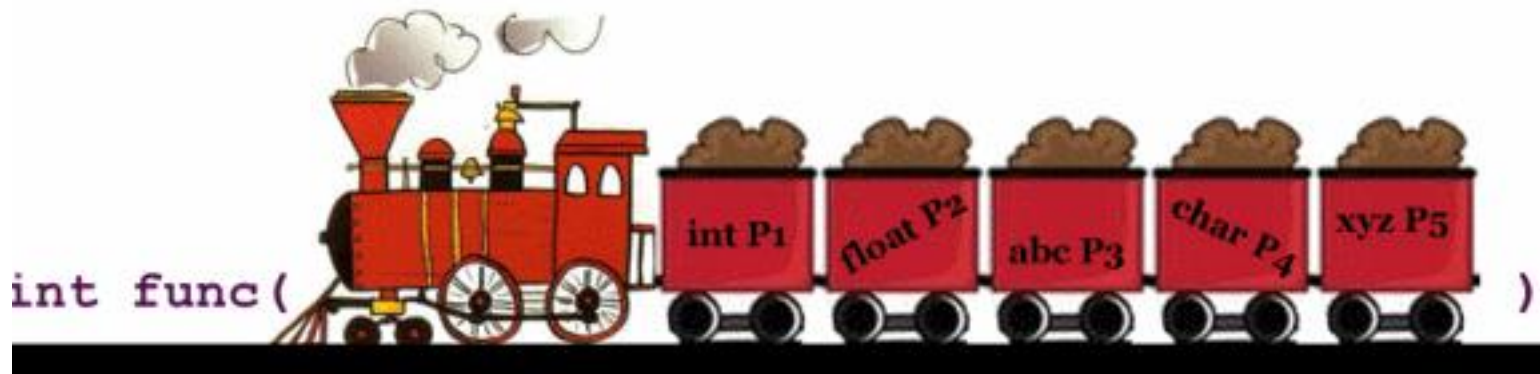
D: OOP





# Topic D.1.10

Describe how data items can be  
**passed to** and **from** actions as  
**parameters**

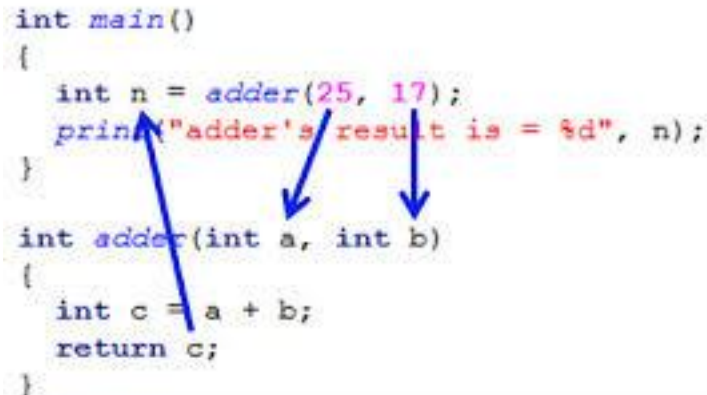


# Parameters

- **Parameters** allow us to **pass information** or instructions into functions and procedures.
- **Parameters** are the names of the information that we want to use in a function or procedure.
- The **values** passed in are called **arguments**.

```
int main()
{
    int n = adder(25, 17);
    printf("adder's result is = %d", n);
}

int adder(int a, int b)
{
    int c = a + b;
    return c;
}
```

A diagram illustrating the passing of arguments to a function. Three blue arrows point from the values '25' and '17' in the function call 'adder(25, 17)' in the 'main' function to the parameters 'a' and 'b' in the 'adder' function definition. A fourth blue arrow points from the variable 'n' in the 'main' function to the format specifier '%d' in the 'printf' statement, indicating that the value stored in 'n' is being printed.

# Example: Parameters

- We can create a procedure that draws a square - but for it to be useful we need to also be able to specify how large it should be.
- The following example creates a procedure called 'square'.
- In the line where we define the name for this procedure, we have included a **variable** called 'distance' inside the brackets.
- Distance is a **parameter** - it allows us to pass a value into the procedure for it to use.

```
fred = turtle.Turtle()  
def square (distance):  
    for x in range (0,4):  
        fred.forward(distance)  
        fred.right(90)
```

Distance is a **parameter** - it allows us to pass a value into the procedure for it to use. Running the following code would create a square of size 50 for each side:

```
square (50)
```

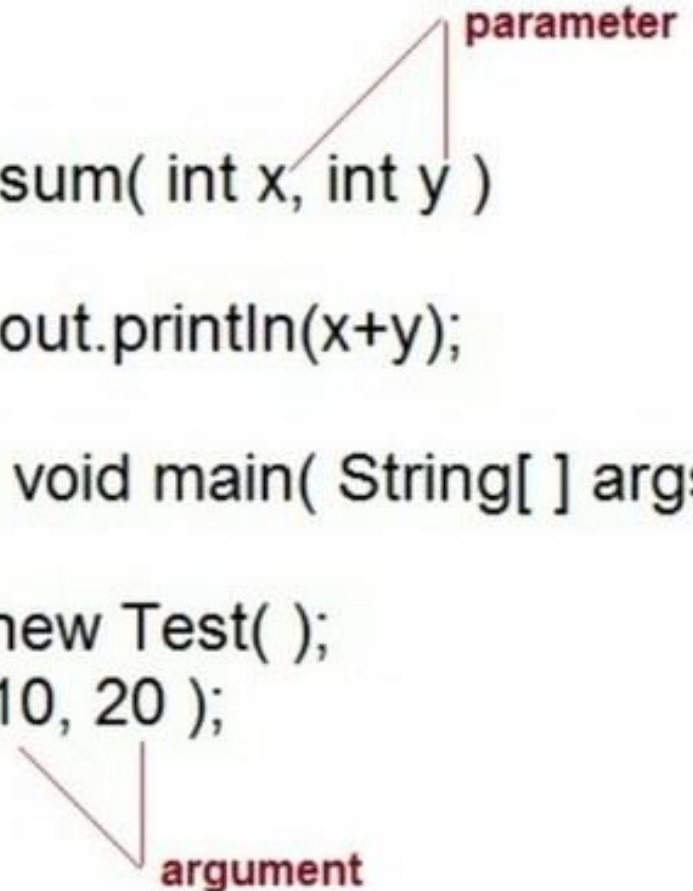
The **50** here is called the **argument** - because it is a value passed in.

# Parameter vs Argument

```
public void sum( int x, int y )  
{  
    System.out.println(x+y);  
}  
public static void main( String[ ] args )  
{  
    Test b=new Test( );  
    b.sum( 10, 20 );  
}
```

parameter

argument

A diagram with two red arrows. One arrow points from the word 'parameter' to the parameter 'int y' in the function signature of the 'sum' method. The other arrow points from the word 'argument' to the value '20' in the method call 'b.sum( 10, 20 );'.



# Declaring methods in Java

**return\_type** - `int` is the return type here, so the function will return an integer

**function\_name** - `product` is the function name

**parameters** - `int x` and `int y` are the parameters. So this function is expecting to be passed 2 integers


```
int product(int x, int y)
```

```
{  
    return (x * y);  
}
```

**function body** - the function body in this case just contains a basic statement `return (x * y);`

```
public class MyMethods {
```

```
    int total() {  
        int a_Value = 10 + 10;  
  
        return a_Value;  
    }
```

A large green arrow pointing from the right towards the 'total()' method definition.

No input parameter,  
but function returns a value

```
    void print_text() {  
  
        System.out.println( "Some Text Here" );  
    }
```

A large red arrow pointing from the right towards the 'print\_text()' method definition.

No input  
parameter,  
no return  
value – void

```
    int total(int aNumber) {  
        int a_Value = aNumber + 20;  
  
        return a_Value;  
    }
```

A large purple arrow pointing from the right towards the 'total(int aNumber)' method definition.

Takes input parameter,  
and function returns a value

```
}
```

# Java is strictly “Pass by Value”!

Consider the following Java program that passes a primitive type to function.

```
public class Main
{
    public static void main(String[] args)
    {
        int x = 5;
        change(x);
        System.out.println(x);
    }
    public static void change(int x)
    {
        x = 10;
    }
}
```

Output:

5

# Explanation

- We pass an **int** to the method **change ()** and as a result the change in the value of that integer is not reflected in the **main** method.
- Java **creates a copy** of the variable being passed in the method and then do the manipulations.
- Hence the change is not reflected in the **main** method.