OCR A Level Computer Science

Your notes

7.1 Programming Techniques

Contents

- Data Types
- Arithmetic, Logical & Boolean Operators
- Programming Constructs
- Selection
- Iteration
- Modularity, Functions & Procedures
- Parameter Passing
- Recursion
- Global & Local Variables
- Integrated Development Environment (IDE)
- Programming Classes, Objects, Methods & Attributes
- Programming Inheritance
- Programming Encapsulation
- Programming Polymorphism


Your notes

Data Types

What is a Data Type?

- A data type is a classification of data into groups according to the **kind of data they represent**
- Computers use different data types to represent different types of data in a program
- The basic data types include:
  - **Integer**: used to represent **whole numbers**, either positive or negative
    - Examples: 10, -5, 0
  - **Real**: used to represent **numbers with a fractional part**, either positive or negative
    - Examples: 3.14, -2.5, 0.0
  - **Char**: used to represent a **single character** such as a letter, digit or symbol
    - Examples: 'a', 'B', '5', '$'
  - **String**: used to represent a **sequence of characters**
    - Examples: "Hello World", "1234", "@#$%"
  - **Boolean**: used to represent **true or false** values
    - Examples: True, False
- It is important to choose the correct data type for a given situation to ensure accuracy and efficiency in the program

What is Casting?

- Casting is when you convert one data type to another data type
- When a user enters data into a program, this will more than likely be in a **string format**
- It's essential to convert some of this **string** data to a **numerical format** where possible
- For example, you may want to perform **calculations** on age-related data to determine if someone is eligible to vote
- Some programming languages can't execute numerical comparisons on text data, making this transformation crucial

- For example if you had "12" stored as a string and you wanted to know if this value was below 20
- Therefore, the string value of "12" will need to be cast as an integer to allow the comparison to take place


Your notes

Python example

int_value = int("123") // converts the string "123" to 123

float_value = float("3.14") // converts the string "3.14" to 3.14

Java example

int intValue = Integer.parseInt("123"); // converts the string "123" to 123

double doubleValue = Double.parseDouble("3.14"); // converts the string "3.14" to 3.14

Casting between data types

Conversion	Example	Output
From **Integer** to **Real**	int_value = 5 real_value = float(int_value)	5.0
From **Real** to **Integer**	real_value = 5.7 int_value = int(real_value)	5
From **String** to **Integer**	int_str = "10" int_value = int(int_str)	10
From **Integer** to **String**	value = 5 str_value = str(value)	"5"
From **Boolean** to **String**	bool_val = True str_val = str(bool_val)	"True"
From **String** to **Boolean**	bool_str = "True" bool_val = bool(bool_str)	True

Arithmetic, Logical & Boolean Operators


Your notes

Arithmetic Operators

What are Arithmetic Operators?

- Arithmetic operators are symbols or keywords used to perform mathematical calculations and operations on numerical values. They allow the computer to perform addition, subtraction, multiplication, division, and more.

Addition operator (+)

- The addition operator (+) is used to add numerical values together or **concatenate** strings:

```
01 sum = 5 + 3
02 fullName = 'John' + ' ' + 'Doe'
```

- In this example, the addition operator adds the values 5 and 3, resulting in 8
- It also concatenates the strings 'John', ' ', and 'Doe' to form the full name 'John Doe'

Subtraction operator (-)

- The subtraction operator (-) is used to subtract one numerical value from another:

```
01 difference = 10 - 4
```

- In this example, the subtraction operator subtracts the value 4 from 10, resulting in 6

Multiplication operator (*)

- The multiplication operator (*) is used to multiply numerical values together:

```
01 product = 3 * 5;
```

- In this example, the multiplication operator multiplies the values 3 and 5, resulting in 15

Exponentiation operator (^)

- The exponentiation operator (^) is used to multiply numerical values by the power of another number:

```
01 product = 3 ^ 3
```

- In this example, the exponentiation operator multiplies the value 3 to the power 3 (3*3*3), resulting in 27

The division operator (/)

- The division operator (/) is used to divide one numerical value by another:

```
01 quotient = 10 / 2
```

- In this example, the division operator divides the value 10 by 2, resulting in 5

The modulus operator (MOD)

- The modulus operator (MOD) returns the remainder when one numerical value is divided by another:

```
01 remainder = 10 MOD 3
```

- In this example, the modulus operator divides the value 10 by 3 and returns the remainder, which is 1. 3 will go into 10 a total of three times and there will be a remainder of 1

Operator precedence (BIDMAS / BODMAS)

- Arithmetic operators follow the rules of operator precedence, which determine the order in which operators are evaluated. Parentheses () can be used to control the order of evaluation:

```
01 result = 2 + 3 * 4
02 adjustedResult = (2 + 3) * 4
```

- In the first example, without parentheses, the multiplication (*) is performed before the addition (+), resulting in 14
- In the second example, with parentheses, the addition is evaluated first, resulting in 5, which is then multiplied by 4, resulting in 20

Logical Operators

What are Logical Operators?

- Logical or comparison operators are symbols or keywords used to compare values and return a **boolean** result. They allow a comparison of variables, or expressions to determine relationships, equality, or inequality between them.

Equal to (==) operator

- The equal to operator (==) compares two values and returns true if they are equal, and false otherwise

```
01 x = 5
02 y = 6
03 print(x == y)
```

- In this example, the equal to operator compares the value 5 with the value 6. These are not the same number so false is printed

Not equal to (!=) operator


Your notes

- The not equal to the operator (!=) compares two values and returns true if they are not equal, and false if they are equal

```
01 x = 5
02 y = 7
03 print(x != y)
```

- In this example, the not equal to operator compares the value 5 with the value 7. As they're not equal, true is printed

Greater than (>) and less than (<) operators

- The greater than operator (>) compares two values and returns true if the left operand is greater than the right operand. The less than operator (<) returns true if the left operand is less than the right operand

```
01 x = 5
02 y = 10
03 print(x > y)
04 print(x < y)
```

- In this example, the greater than operator compares the value 5 with the value 10, resulting in false
- The less than operator compares 5 with 10, resulting in true

Greater than or equal to (>=) and less than or equal to (<=) operators

- The greater than or equal to operator (>=) compares two values and returns true if the left operand is greater than or equal to the right operand. The less than or equal to operator (<=) returns true if the left operand is less than or equal to the right operand:

```
01 x = 5
02 y = 10
03 print(x >= y)
04 print(x <= y)
```

- In this example, the greater than or equal operator compares 5 with 10, resulting in false
- The less than or equal to operator compares 5 with 10, resulting in true

Boolean Operators

What are Boolean Operators?

- Boolean operators are symbols or keywords used to combine and manipulate **boolean** values. They allow the computer to perform logical operations, such as combining conditions, negating values, and determining whether an expression is true or false.

Your notes

AND operator

- The AND operator **returns** true if both operands are true, otherwise returns false:

```
01 x = 5
02 y = 10
03 z = 15
04 result = (x < y) AND (y < z)
```

- In this example, the expression (x < y) AND (y < z) evaluates to true because both conditions are true. If any of the conditions were false, the result would be false

OR operator

- The OR operator **returns** true if at least one of the operands is true, and false if both operands are false:

```
01 a = 5
02 b = 10
03 c = 15
04 result = (a > b) OR (b < c)
```

- In this example, the expression (a > b) OR (b < c) evaluates to true because the second condition is true, even though the first condition is false. If both conditions were false, the result would be false

NOT operator

- The NOT operator is used to negate a Boolean value. It returns true if the operand is false, and false if the operand is true:

```
01 value = false
02 result = NOT value
```

- In this example, the not operator negates the value of false, resulting in true


Your notes

Your notes

Programming Constructs

What is a Programming Construct?

- A programming construct determines the **order in which lines of code are executed**
- There are three programming constructs. These are
  - **Sequence**
  - **Iteration**
  - **Branching** (also known as selection)

Sequence

- Sequence refers to lines of code which are run **one line at a time**
- The lines of code are run in the **order** that they written from the first line of code to the last line of code

Branching (Selection)

- Branching, also known as selection is when the **flow of the program is interrupted and a condition is tested**
- The outcome of this condition will then determine which lines or block of code is run next
- There are two ways to write selection statements:
  - **if**... **elseif**... **else**... statements - this is when you test conditions sequentially
  - **switch**... **case**... statements - this is when you test an expression against multiple possible constant values (known as cases)

Iteration

- Iteration is **repeating a line or a block of code** using a loop
- Iteration can be:
  - **count controlled** - this is when the code is repeated a fixed number of times (e.g. using a for loop)
  - **condition controlled** - this is when the code is repeated until a condition is met (e.g. using a while loop or a do while loop)

Identifying programming constructs

- You can identify which programming constructs are used by looking at certain keywords
- The keywords **if, elseif, else, endif, switch, case** indicate that the construct is **selection**
- The keywords **for, while, do** indicate that the construct is **iteration**
- If none of these keywords are used, this is a good indication that the construct is **sequence**

```
01 numbers = []
02 # Ask the user to enter 5 numbers
03 for i in range(5):
04     num = int(input("Enter a number: "))
05     numbers.append(num)
06 # Assume the first number is the largest
07 largest_number = numbers[0]
08 # Identifying the largest number
09 for num in numbers:
10     if num > largest_number:
11         largest_number = num
12 print("The largest number is:", largest_number)
```

- In this example, **iteration** is used twice:
    1. Repeating code to ask the user to enter 5 numbers (indicated by the keyword **for** on line 03)
    2. Repeating code to identify the largest number (indicated by the keyword **for** on line 09)
- In this example, **selection** is used once to determine if a number is bigger than the currently stored biggest number (indicated by the keyword **if** on line 10) and if so it will update the largest number on line 11
- **Sequence** is any lines of code which does not use iteration or selection. In this example, the first and last lines of code are **sequence**


Your notes

Selection

Your notes

Selection

What is Selection?

- Selection is used within the **branching** (selection) **programming construct**
- They are used to **test conditions** and the **outcome of this condition** will then determine which **lines or block of code is run next**
- There are two ways to write selection statements:
  - **if**... **elseif**... **else...**
  - **switch**... **case**...

Writing a condition

- Think of it like writing a **yes/no question**
- For example
  - Is the number bigger than 10?
  - Is the number between 50 and 100?
  - Is the answer Paris?
- If the question can't be answered with yes/no then it needs to be **rewritten in this way**

Using more than one operator

- Using one operator is **most common** but sometimes two are needed. More than two operations can be used but it gets more complicated as it involves using **Boolean** operators
- Imagine a program where the user has to enter a number based on the role of a dice. The number needs to be between 1 and 6
- The yes/no question could be is the number between 1 and 6 - but it would be structured slightly differently in the code
- `if number >=1 and number <=6 then`
- The first check is if the number is greater than or equal to 1
- The second check is if the number is less than or equal to 6

- The final check is if both sides are true

If Statements

Your notes

- An if elseif else statement will let you choose a line/lines of code to run if a condition is true or false
- Below are three ways to use if statements:

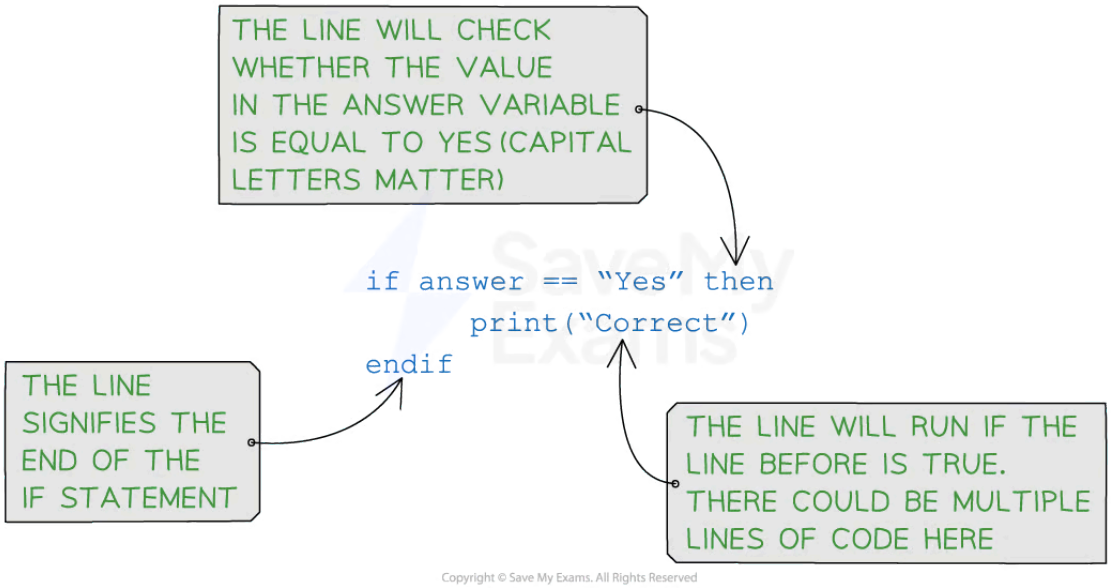
1. if...
2. if... else...
3. if... elseif... else...

Syntax of an if... statement

The syntax of an if the statement consists of the if keyword, followed by a condition, and a code block that is executed if the condition evaluates to true.

```
if condition then
  // Code to be executed if the condition is true
endif
```

Pseudocode example



if statement example pseudocode

Python example

```
number = 5
if number > 0:
  print('The number is positive.')
```

- In this example, the if statement checks if the value of the variable number is greater than 0. If the condition is true, the message 'The number is positive.' is output

Java example

```
public class Main {
  public static void main(String[] args) {
    int number = 5;
    if (number > 0) {
      System.out.println("The number is positive.");
    }
  }

}
```

Syntax of an if... else... statement

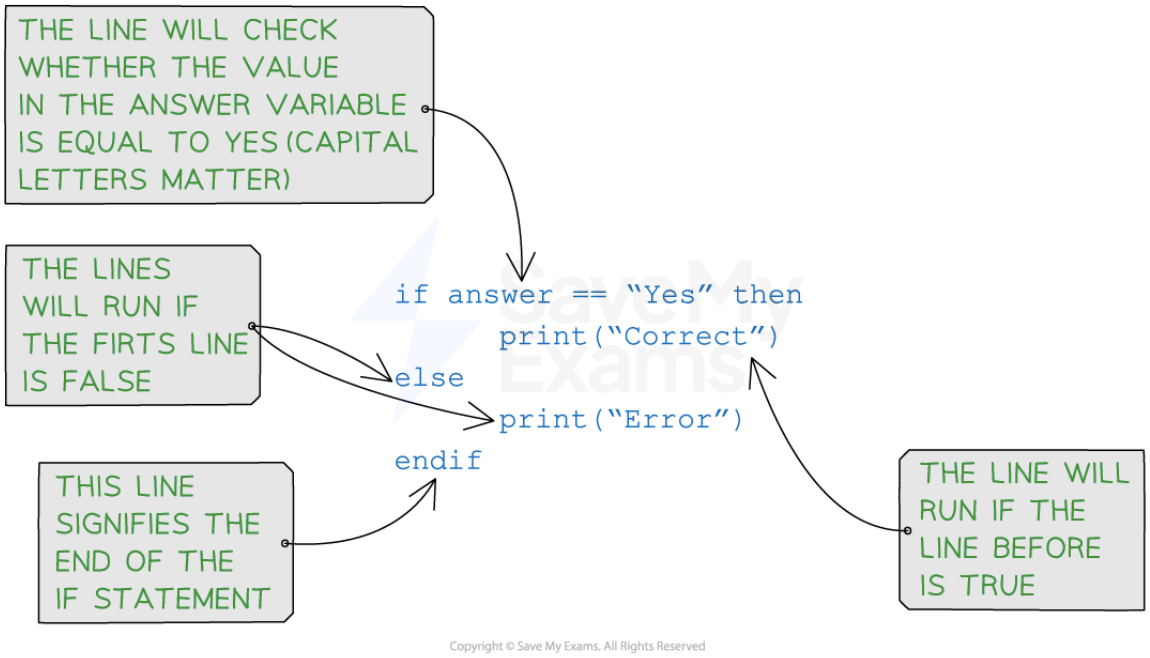
The if statement can be extended with an else clause to specify an alternative block of code that is executed when the condition evaluates to false.

```
if condition then
 // Code to be executed if the condition is true
else
// Code to be executed if the condition is false
endif
```

Pseudocode example


Your notes

Your notes



if else statement example pseudocode

Python example

```
number = -3
if number > 0:
    print('The number is positive.')
else:
    print('The number is not positive.')
```

In this example, if the value of number is greater than 0, the message 'The number is positive.' is output. Otherwise, the message 'The number is not positive.' is output.

Java example

```
public class Main {
    public static void main(String[] args) {
        int number = -3;
        if (number > 0) {
            System.out.println("The number is positive.");
        } else {
            System.out.println("The number is not positive.");
        }
```

```
  }
}
```

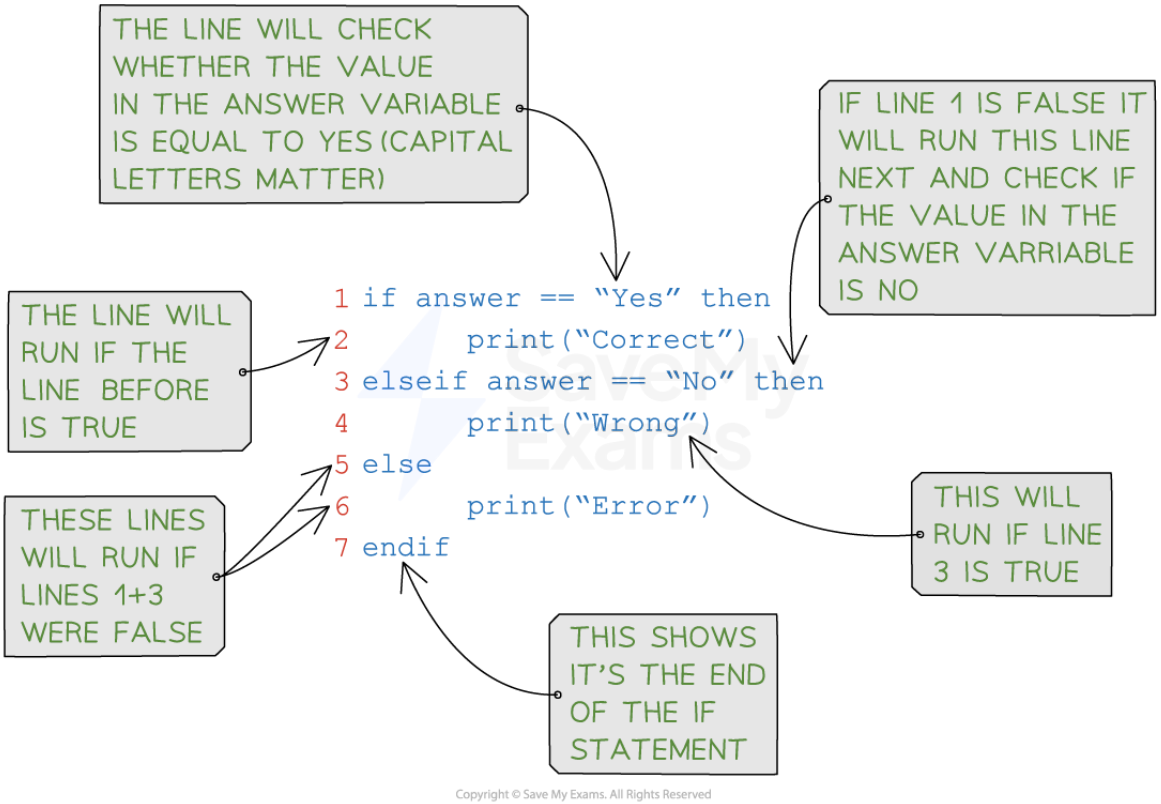
Your notes

Syntax of an if... elseif... else... statement

The elseif clause specifies additional conditions to check if the initial if condition is false. This allows the handling of multiple scenarios in a more complex decision-making process:

```
if condition then
  // Code to be executed if the condition is true
elseif condition then
  // Code to be executed if the condition is true
else
  // Code to be executed if all conditions are false
endif
```

Pseudocode example



if elseif else statement example pseudocode

Python example

```
score = 85
if score >= 90:
    print('Excellent!')
elif score >= 80:
    print('Good.')
elif score >= 70:
    print('Fair.')
else:
    print('Needs improvement.')
```

In this example, the if elseif else statement evaluates the value of score to determine the corresponding message based on the score range.

Java example

```
public class Main {
    public static void main(String[] args) {
        int score = 85;
        if (score >= 90) {
            System.out.println("Excellent!");
        } else if (score >= 80) {
            System.out.println("Good.");
        } else if (score >= 70) {
            System.out.println("Fair.");
        } else {
            System.out.println("Needs improvement.");
        }
    }
}
```

Examiner Tips and Tricks

- You can use as many elseif... statements as you want to but it might be clearer to use a switch case statement.
- Check whether to use an and or an or operator in the condition as it's easy to get these mixed up. For example:
    - if number <0 and number >100 will check if the number is both below 0 and greater than 100 (however a number cannot be both less than 0 and greater than 100)


Your notes

- if number <0 or number >100 will check if the number is between 1 and 99 (this will check that the number is either less than 0 or greater than 100)


Your notes

Worked Example

Two people play a counting game. The rules of the game are as follows:
• The first player starts at 1
• Each player may enter one, two or three numbers on their turn, and the numbers must be in ascending order
• Players take it in turns to choose
• The player who chooses "15" loses the game

For example, if the first player chooses three numbers (1, 2, 3) then the second player could choose one number (4), two numbers (4, 5) or three numbers (4, 5, 6). The first player then takes another go.

Write an algorithm using pseudocode that allows two players to play this game. The algorithm should:
• Alternate between player 1 and player 2
• Ask the player how many numbers they would like to choose, ensuring that this is between 1 and 3
• Display the numbers that the player has chosen
• Display a suitable message to say which player has won once the number 15 has been displayed

8 marks

How to answer this question:

- We'll start by asking the player how many numbers they want to choose and check they've entered a number between 1 and 3. We'll do this as a loop which will keep asking them how many numbers until they've entered either 1, 2 or 3

```
choice=0

while choice < 1 or choice > 3:

 choice = input("how many numbers?")

endwhile
```

- Now we'll display the numbers the player has chosen. We'll need to set num=1 at the start of the code

```
for y = 1 to choice

 print(num)
```

```
 num = num + 1
next y
```

- Now we'll make this code run until a player chooses 15

```
num = 1
while num <= 15
 choice = 0
 while choice < 1 or choice > 3
  choice = input("how many numbers?")
 endwhile
 for y = 1 to choice
  print(num)
  num = num + 1
 next y
endwhile
print(turn + " wins!")
```

- Now we'll get the players to take turns by setting whose turn it is before the loop and then swapping whose go it is during the loop

```
turn = "player 1"
if turn == "player 1" then
 turn = "player 2"
else
 turn = "player 1"
endif
```

Answer:

```
num = 1
turn = "player 1"
while num <= 15
  print(turn + "'s turn")
  choice = 0
  while choice < 1 or choice > 3
```

Your notes

```
    choice = input("how many numbers?")
  endwhile
  for y = 1 to choice
    print(num)
    num = num + 1
  next y
  //swap turn
  if turn == "player 1" then
    turn = "player 2"
  else
    turn = "player 1"
  end if
endwhile
print(turn + " wins!")
```


Your notes

Case Statements

- switch/case is a type of conditional statement that provides an alternative way to perform **multiple comparisons** based on the value of an expression
- These statements are particularly useful when you have a **single expression** that you want to compare against **multiple possible values**

Syntax of a switch case statement

- The syntax of a switch case statement consists of the switch keyword followed by an expression
- This expression is evaluated, and its value is then compared against various case labels. If a match is found, the corresponding block of code is executed
- The default keyword is optional and specifies a block of code to be executed if none of the case labels match the expression

```
switch expression
 case value1:
  // Code to be executed if the expression matches value1
 case value2:
  // Code to be executed if the expression matches value2
 // more case statements if needed
 default:
  // Code to be executed if no case matches the expression
endswitch
```

Python example

```
grade='b'
```

```
match (grade):
    case 'a':
      print("Excellent")
    case 'b':
      print("Good")
    case 'c':
      print("Fair")
    case_:
      print("Needs improvement")
```

Java example

```
public class Main {
  public static void main(String[] args) {
    char grade = 'b';
    switch (grade) {
      case 'a':
        System.out.println("excellent");
        break;
      case 'b':
        System.out.println("good");
        break;
      case 'c':
        System.out.println("fair");
        break;
      default:
        System.out.println("needs improvement");
    }
  }
}
```


Your notes

Iteration


Your notes

Iteration

What is Iteration?

- Iteration is the process of doing something **more than once (repeat),** otherwise known as a **loop**
- A loop can be **count controlled** which means the code is repeated a fixed number of times
- A loop can also be **condition controlled** which means the code is repeated until a condition is met
- Three common loops are:
  - **for loops** (count controlled)
  - **while loops** (condition controlled)
  - **do while loops** (condition controlled)

For Loops

A for loop is a count controlled loop that will repeat a fixed number of times. It provides a concise and structured way to perform repetitive tasks.

Syntax of a for loop

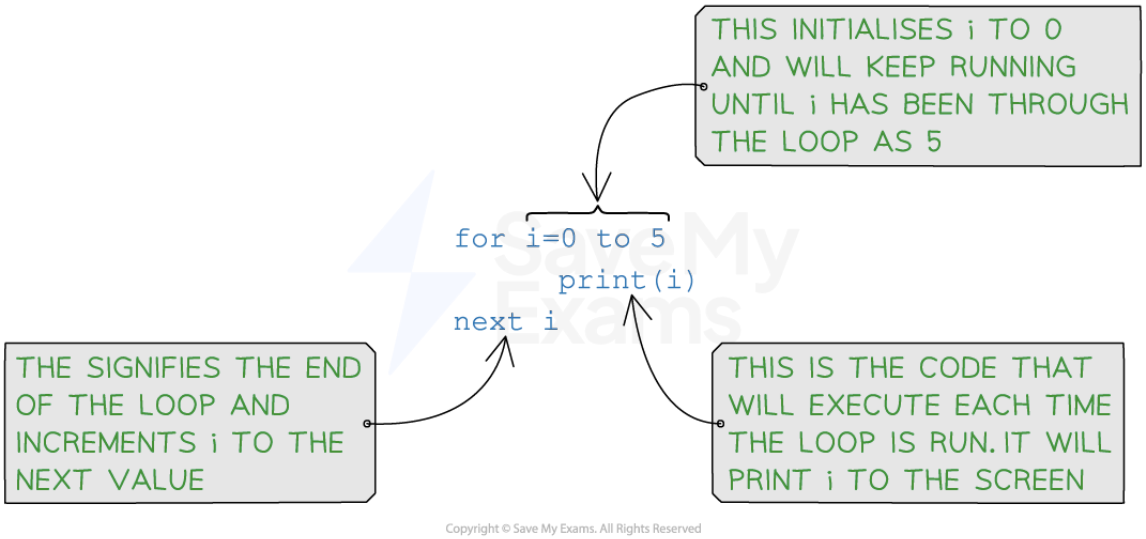
The syntax of a for loop consists of three main parts:

```
01 for i = x to y
02  // Code to be executed in each iteration
03 next i
```

1. **Initialisation**: The initialisation is executed only once at the beginning of the loop. It is used to initialise a counter variable that controls the loop's execution which is i in this example
2. **Range**: The range that the count variable will increment through
3. **Increment/Decrement**: The default is to increment by 1 each time unless specified

Pseudocode example

Your notes



for loop example pseudocode

Python example

```
01 for i in range(0,6):
02     print i
```

In Python, the range specifies the numbers used in the loop. The final number (6 in this case) is 1 higher than the number we want to run the loop with.

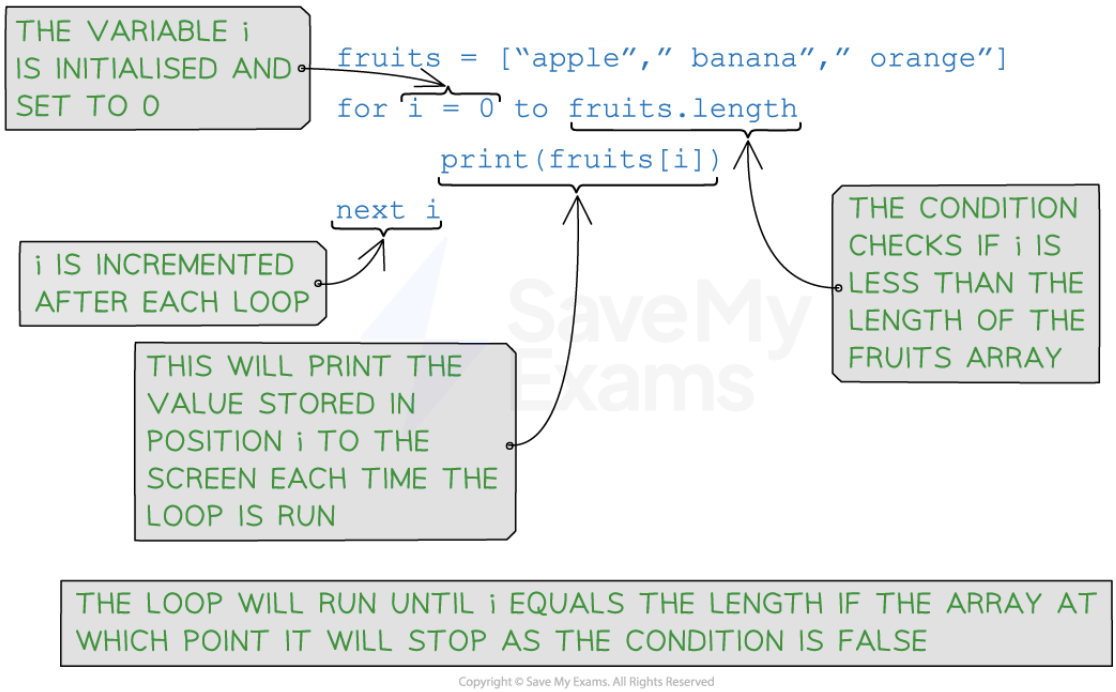
Java example

```
01 for (int i = 0; i < 6; i++) {
02     System.out.println(i);
03 }
```

Iterating over an array

Pseudocode example

Your notes



for loop iterating over an array and outputting each item

Python example

```
01 fruits = ["apple", "banana", "orange"]
02 for i in range(len(fruits)):
03     print(fruits[i])
```

Java example

```
01 String[] fruits = {"apple", "banana", "orange"};
02 for (int i = 0; i < fruits.length; i++) {
03     System.out.println(fruits[i]);
04 }
```

While Loops

- A while loop is a condition controlled loop that will repeat until a condition is met
- While loops provide a flexible and powerful way to handle situations **where the number of iterations is unknown in advance**

Syntax of a while loop

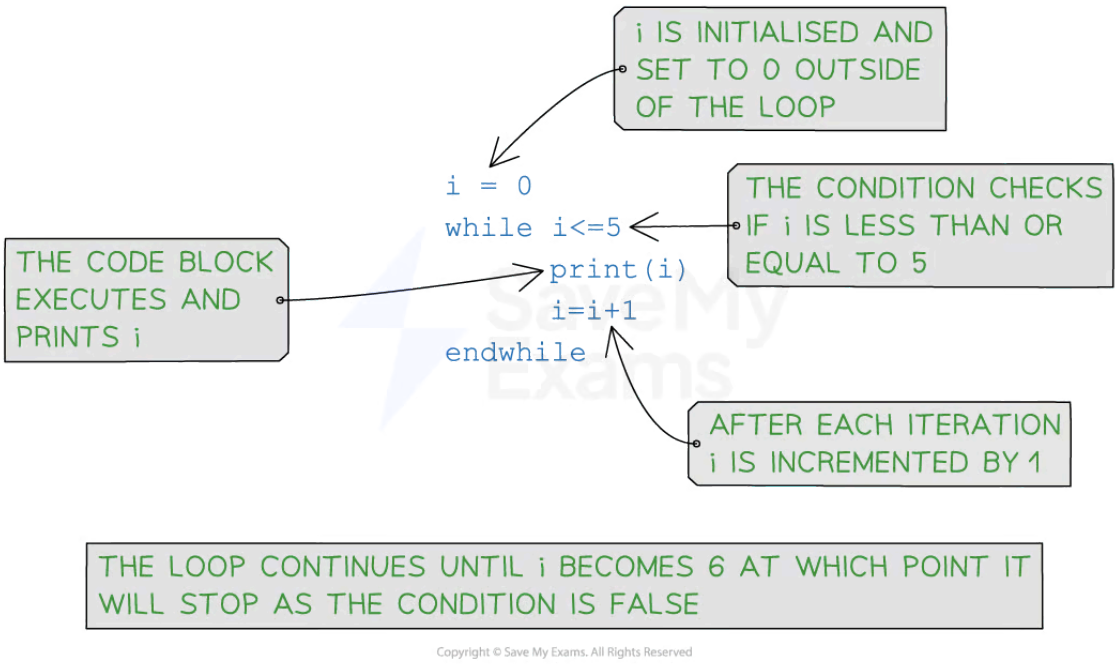
Your notes

The syntax of a while loop consists of a condition and a code block:

```
01 while condition
02   // Code to be executed as long as the condition is true
03 endwhile
```

The condition is evaluated **before** each iteration. If the condition evaluates to true, the code block is executed. If the condition evaluates to false, the loop terminates.

Pseudocode example



while loop example pseudocode

Python example

```
01 i = 0
02 while i <= 5:
03     print(i)
04     i = i + 1
```

Java example

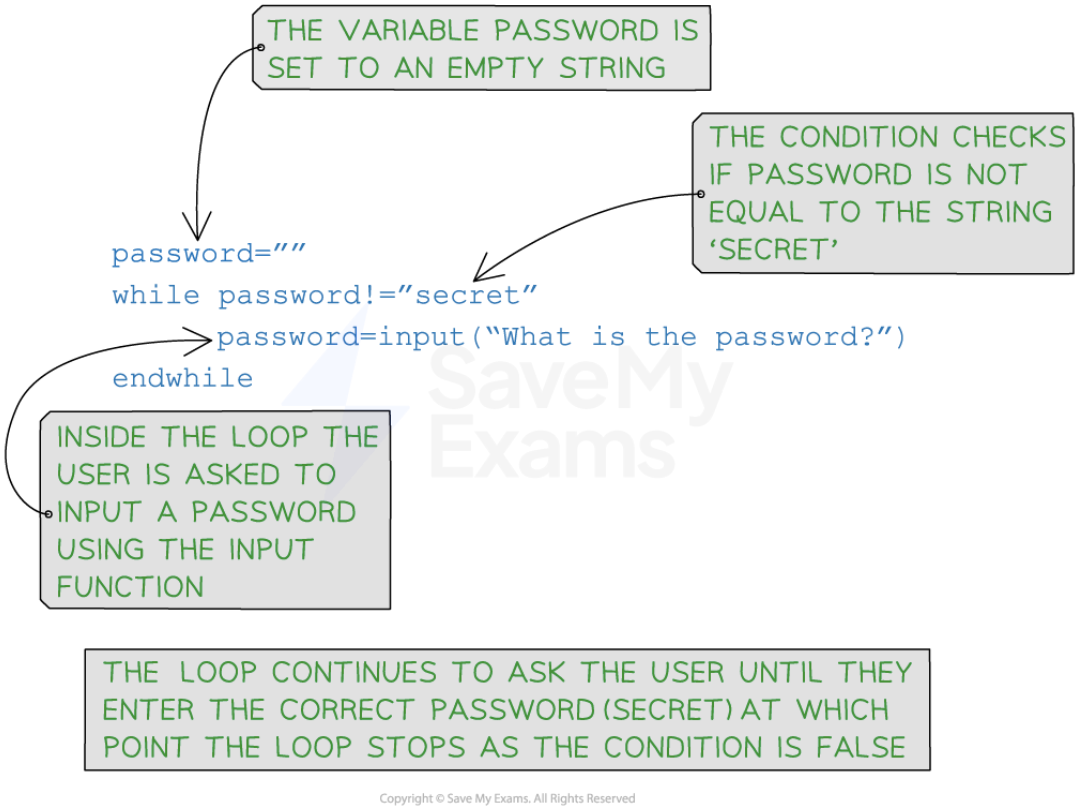
```
01 int i = 0;
02 while (i <= 5) {
```

```
03     System.out.println(i);
04     i = i + 1;
05 }
```

Your notes

Checking if the password is 'secret'

Pseudocode example



while loop checking if the password is correct

Python example

```
01 password = ""
02 while password != "secret":
03     password = input("What is the password? ")
```

Java example

```
01 import java.util.Scanner;
02 public class Main {
```

```
03   public static void main(String[] args) {
04      Scanner scanner = new Scanner(System.in);
05      String password = "";
06      while (!password.equals("secret")) {
07         System.out.print("What is the password? ");
08         password = scanner.nextLine();
09      }
10      scanner.close();
11   }
12 }
```

Your notes

Examiner Tips and Tricks

- Incrementing a variable can be done in different ways (depending on the language)
- For example:
  - i= i + 1
  - i + = 1
  - i++

Worked Example

A simple Python program is shown below.

```
01 //Program to calculate the number of times a number goes into 100
02
03 count = 0
04 num = int(input("Enter a number"))
05 while (count*num)<=100
06     count=count+1
07 endwhile
08 count=count-1 //Take one off as gone over
09 print(str(num) + " goes into 100 " + str(count) + " times.")
```

State the output of the program when the number 30 is entered.

[1]

How to answer this question:

- If 30 is entered this is saved in the variable num
- The while loop will run while count*num is less than or equal to 100

- count is 0, so 30 * 0 = 0
- As there is a loop to iterate through it would be useful to produce a trace table to help us keep track of the value of the different variables

count	num	count * num
0	30	
1		30
2		60
3		90
4		120
3		

- The loop will repeat and when count is 4, count*num is 120 which causes the condition to be false and the loop to stop
- Count is decremented
- The statement which is printed is 30 goes into 100 3 times

Answer:

30 goes into 100 3 times.


Your notes

Do While Loops

- A do while loop is another example of a condition controlled loop that will repeat until a condition is met.
- Do while loops provide a variation of the while loop with a slightly different behaviour
- The code within a do while loop will always run at least once, unlike a while loop which may not run at all if the condition is already met

Syntax of a do while loop

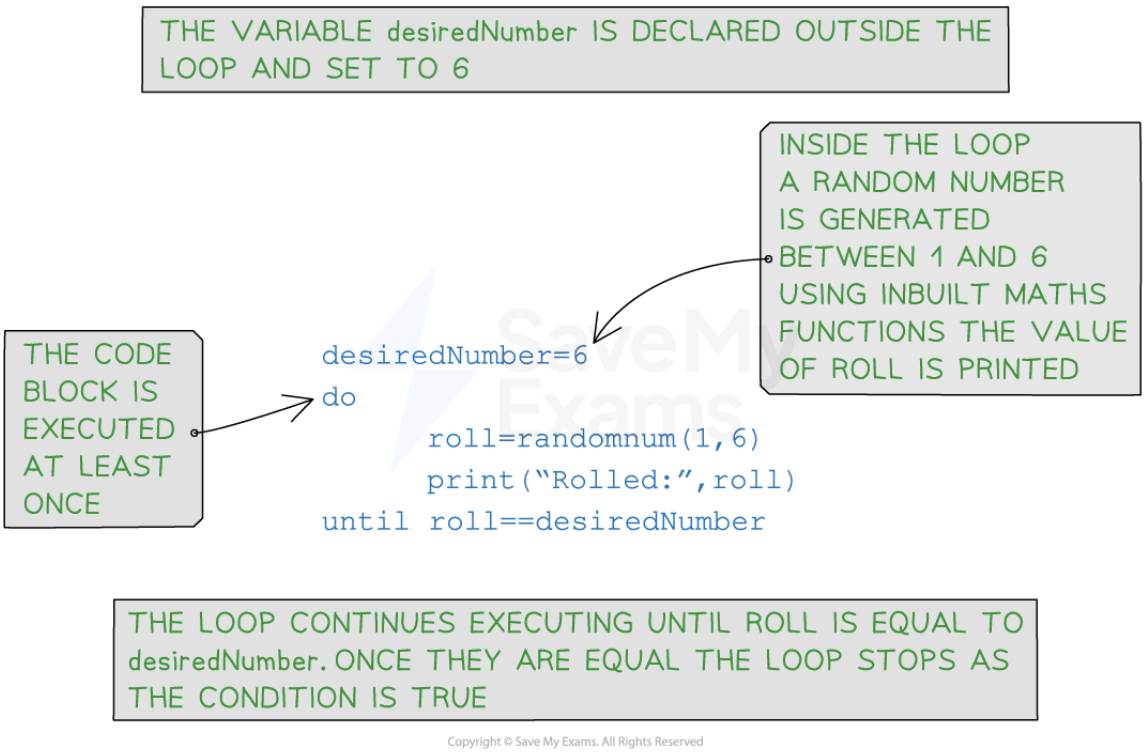
The syntax of a do while loop consists of a code block and a condition:

```
01 do
02   // Code to be executed at least once
03 until condition
```

- **Code Block**: The code block is executed first before evaluating the condition
- **Condition**: The condition is evaluated after executing the code block. If the condition evaluates to false, the loop continues executing. If the condition evaluates to true, the loop terminates

Pseudocode example

Your notes



Do while example pseudocode

Python example

- It isn't possible to use a do while loop in Python so the code would need to be written to use a while loop

```
01 import random

02 desired_number = 6
03 roll = 0

04 while roll != desired_number:
05     roll = random.randint(1, 6)
06     print("Rolled:", roll)
```

Java example

```
01 import java.util.Random;

02 public class Main {
03     public static void main(String[] args) {
```

```
04      Random random = new Random();
05      int desiredNumber = 6;
06      int roll;

07      do {
08          roll = random.nextInt(6) + 1;
09          System.out.println("Rolled: " + roll);
10      } while (roll != desiredNumber);
11  }
12 }
```

Examiner Tips and Tricks

You can use either a while loop or a do while loop but don't forget that a do while loop will always run once before checking if the condition is true

Your notes

Modularity, Functions & Procedures


Your notes

Modularity

What is Modularity?

- **Modularity** is a concept where problems are broken down into more manageable pieces
- Each piece of the problem should be carried out by one single **subroutine**
- **Subroutines**, also known as **modules** are **standalone blocks of code** and when called they will **complete a task**
- They promote code **reusability, modularity, and organisation**, enabling a programmer to write efficient and maintainable programs
- Functions and procedures are examples of subroutines and they are both very similar in nature

Difference between functions and procedures

- Functions **will return a value**. For example if a function completes a calculation, then the result of the calculation will be returned to the part of the programming code that called the function
- Procedures can also be called to complete a task, however, they **do not return a value** back to the part of the programming code that called the procedure
- Both functions and procedures may need a **parameter**. These are pieces of data which are **passed into the function** or **procedure** to allow it to complete its task

Considerations and best practices

- **Naming**: Choose descriptive and meaningful names for your functions and procedures that indicate their purpose
- **Parameter names**: Use clear and meaningful parameter names to improve code readability
- **Focus**: Aim for functions and procedures that are short and focused so they they complete a specific task
- **Return values**: Functions should have explicit return statements with meaningful return values, while procedures should not have return statements

Functions

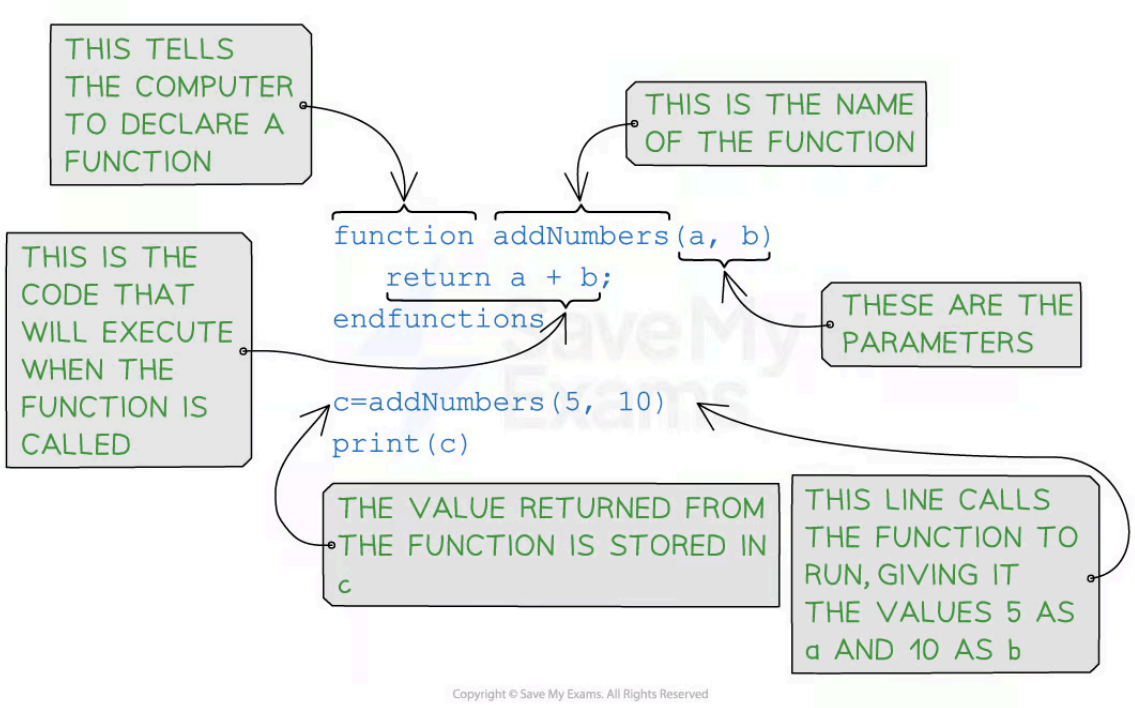
The syntax for defining a function is as follows:

```
function functionName(parameter1, parameter2)
  // Code block to perform the task
  // Return value;
endfunction
```

Your notes

Pseudocode example

In the example below two numbers are passed as parameters (a and b) into the function which adds them together and returns the result.



Function example in pseudocode

Python example

```
def add_numbers(a, b):
    return a + b
c = add_numbers(5, 10)
print(c)
```

Java example

```
public class Main {
    public static void main(String[] args) {
```

```
    int c = addNumbers(5, 10);
    System.out.println(c);
  }
  public static int addNumbers(int a, int b) {
    return a + b;
  }
}
```


Your notes

Procedures

- A procedure is similar to a function but **does not return a value**. Instead, it **performs a series of actions or operations** which could be anything the programmer wants the procedure to execute
- A procedure is essentially a function without a `return` statement or with a `return` statement that has no value to return

The syntax is the same as for functions:

```
procedure procedureName(parameter1, parameter2)
 // Code block to perform actions
 // No return statement or return with no value;
endprocedure
```

Pseudocode example

In the example below two numbers are passed as parameters (a and b) into the procedure which adds them together and prints the result. As this is a procedure, the result cannot be returned.

```
procedure addNumbers (a,b)
 total = a + b
 print (total) // The total is printed rather than returned
endprocedure

c = addNumbers(5,10)
print (c)
```


Your notes

Parameter Passing

What are Parameters?

- A parameter is a piece of data that is input into a code block such as a function or a procedure so that it can complete its task
- For example this could be a password that the user has entered which is passed into a code block as a parameter so it can check it meets password complexity rules
- When passing parameters into a code block, two standard methods are used:
  - Passing **by value** (also known as byVal)
  - Passing **by reference** (also known as byRef)
- The choice of which method to use depends on the specific requirements of the code block and how the programmer wants to handle data modifications inside and outside of this.

Parameter Passing by Value

- When a parameter is passed by value, **a copy** of the actual value is made and assigned to the function's parameter
- Any changes to the parameter within the function's scope do not affect the original value outside the function
- The function works with and changes its own **local copy of the value**
- This approach ensures that the **original data remains unchanged**, making it suitable when you want to protect the original data from being modified within the function
- When the function ends, the copy of the value is destroyed
- However, it can **lead to performance overhead** when working with large data structures since a copy is made

Pseudocode example

In this example pseudocode, a variable called `number` is assigned a random number between 1 and 10. This is then passed into the function `printValue` **by value**.

```
number = random (1,10)
function printValue(number : byVal) // value is passed by value
 return("The random value is:", number )
```

```
endfunction
printValue(number)
```

Your notes

Parameter Passing by Reference

- When a parameter is passed by reference, the function receives the **memory address of the value** rather than a copy of the value
- Any changes made to the parameter within the function **directly affect the original value** outside the function
- Both the function and the calling code share the same memory location
- This approach allows functions to **directly modify the original data**, making it **efficient** for working with large data structures
- However, it can also lead to **unexpected side effects** if not handled carefully, as changes made within the function affect other parts of the program

Pseudocode example

In this example pseudocode, a variable called `number` is assigned a random number between 1 and 10. This is then passed into the function `printValue` **by reference**.

```
number = random (1,10)
function printValue(number : byRef) // value is passed by reference
  return("The random value is:", number )
endfunction
printValue(number)
```

Worked Example

A pseudocode recursive function, `generate`, is shown.

```
function generate(num1 : byval)
if num1 > 10 then
  return 10
else
  return num1 + (generate(num1 + 1) DIV 2)
endif
endfunction
```

The parameter, `num1`, is passed by value. Explain why the parameter was passed by value instead of by reference.

2 marks

Your notes

How to answer this question:

- You need to demonstrate your understanding of passing by parameter and by reference
- You must refer to the parameter in the question, num1

Answer:

If the value is passed by value, num1 will not be overridden as it is a copy of the parameter that is used, and this will produce the correct output.

Acceptable answers you could have given instead:

If the parameter had been passed by reference, it would not produce the correct result, as num1 would be overridden because it is a pointer to the address of the variable.

Comparison

Aspect	Passing by Value	Passing by Reference
Parameter Handling	A copy of the **argument's** value is assigned to the function's parameter. The function works with its own local copy of the value.	The function receives a reference or **memory** address of the argument, allowing direct access and modification of the original data.
Data Protection	Passing by value protects the original data from being modified within the function. Any changes made to the parameter do not affect the original argument outside the function.	Passing by reference allows functions to directly modify the original data, possibly leading to unintended changes outside the function if not handled carefully.
Performance Overhead	Copying data for passing by value can incur performance overhead, especially when working with large **data structures** or objects.	Passing by reference avoids data copying, making it efficient, especially for large data structures.
Memory Consumption	Passing by value can increase memory consumption due to the creation of a copy of the data for the function **call**.	Passing by reference does not duplicate data, improving memory efficiency, especially in memory-constrained environments.

Complex Data Structures	Suitable for passing primitive types (e.g. numbers, strings) or small data structures.	Well-suited for working with large data structures (e.g. arrays, lists) without creating copies, reducing memory usage.
Data Protection	Passing by value is useful when the original data remains unaltered, providing data protection.	Passing by reference is appropriate when the function needs to manipulate and update the original data directly.
Scope Limitation	Changes made to the parameter within the function do not spread to the calling code, preserving data integrity outside the function.	Changes made to the parameter directly affect the original data outside the function, which can lead to side effects and unintended behaviour.
Function Reusability	Passing by value can enhance function reusability since the function does not modify the original data.	Passing by reference may reduce function reusability as the function may unexpectedly affect the caller's data.
Convenience	It is easy to implement and understand, especially for simple data types.	It requires careful handling to prevent unintended changes to data outside the function, making it more complex to manage.


Your notes

Recursion


Your notes

Features of Recursion

What is Recursion?

- **Recursion** is a highly effective programming technique where a **function calls** itself to **solve a problem** or **execute a task**
- Recursion doesn't rely on iterative loops. Instead, it uses the idea of **self-reference** to **break down** complicated problems into **more manageable** subproblems
- A recursive algorithm has three features:
  - the function must **call itself**
  - a **base case** - this means that it can return a value without further recursive calls
  - a **stopping base** - this must be reachable after a finite number of times

How does recursion work?

- In a recursive function, **the function calls itself with a modified input parameter** until it reaches a base case — a condition that stops the recursion and provides the final result
- Each recursive call breaks down the problem into more minor instances until it reaches the base case

Example: Factorial calculation

```
def factorial(n):
# Base case
  if n == 0 or n == 1:
    return 1
  else:
    # Recursive call with a smaller instance of the problem
     return n * factorial(n - 1)
result = factorial(5)
print(result)
# Output: 120 (5! = 5 * 4 * 3 * 2 * 1)
```

- In this example, the factorial function calculates the **factorial** of a positive integer n
- It does so by breaking down the problem into smaller instances, multiplying n with the factorial of (n - 1) until it reaches the base case (n == 0 or n == 1)


Your notes

Importance of a proper stopping condition

- It is important to have a proper **stopping condition or base case** when using recursion to avoid **stack overflow** errors which result in program crashes
- If a recursive function does not have a stopping condition, it will continue to call itself indefinitely, which can use up excessive **memory** and cause the program to malfunction

Designing a stopping condition

- When creating a stopping condition, it's important to consider the problem being solved
- Identify the easiest scenario where the function can provide a direct result. This scenario should be defined as the base case, covering the simplest instances of the problem
- By doing so, the function will be able to stop the recursion when those conditions are met
- The difference between line 7 and the function declaration on line 1, is that num1 is replaced with result + 1 so we'll need to set num1 equal to result + 1

Recursion: Benefits & Drawbacks

- Programs can be written using either recursion or **iteration** - which one is used will depend on the problem being solved
- There are many benefits and drawbacks to using either, depending on the situation:

Recursion

Benefits	Drawbacks
Concise - can often be expressed in a more concise way, especially for structures like trees or fractals	**Performance** - repeated function calls can be CPU and Memory intensive, leading to slower execution
Simple - simply stating what needs to be done without having to focus on the how can make it more readable and maintainable	**Debugging** - recursive code can be much more difficult to track the state of the program

	Limited application - not all problems are suited to recursive solutions

Your notes

Iteration

Benefits	Drawbacks
Performance - more efficient that recursion, less memory usage.	**Complexity** - can get very complex and use more lines of code than recursive alternatives
Debugging - easier to understand and debug	**Less concise** - compared to recursive alternatives, making them harder to understand
Wider application - more suitable to a wider range of problems	

Writing Recursive Algorithms

- Here is a **recursive algorithm** for a simple countdown program written in **Python** to countdown from 10 to 0

Step 1

- Create the **subroutine** (in this example it will be a **function** as it will **return a value**) and identify any **parameters**

```
def countdown_rec(n): #n is the parameter passed when we call the subroutine
```

Step 2

- Create a **stopping condition** - when n is 0 the function will stop

```
def countdown_rec(n):

  print(n) #output the starting number
  if n == 0: #stopping condition
    return
```

Step 3

- Add a **recursive function** call

```
def countdown_rec(n):
  print(n)
```

```
    if n == 0:
        return
    countdown_rec(n -1) #recursive functional call
```

Step 4

- Call the **function**

```
def countdown_rec(n):
    print(n)
    if n == 0:
        return
    countdown_rec(n -1)

countdown_rec(10) #call the function and pass 10 as a starting value
```

Output to user

```
10
9
8
7
6
5
4
3
2
1
0
```

Tracing Recursive Algorithms

- Now lets **trace** the **recursive algorithm** we have just written to check what happens during the execution of the program
- Here is the completed program and we are going to start it using the command `countdown_rec(5)`

```
def countdown_rec(n):
    print(n)
    if n == 0:
        return
    countdown_rec(n -1)
```

- Using a simple **trace table** we can trace the **recursive** function call

Function call	print(n)	countdown_rec(n -1)

Your notes

countdown_rec(5)	5	4
countdown_rec(4)	4	3
countdown_rec(3)	3	2
countdown_rec(2)	2	1
countdown_rec(1)	1	**0 (return)**


Your notes

Translate Between Iteration & Recursion

- **Recursive** algorithms can be **translated** to use **iteration**, and **vice versa**
- Let's look at the previous example **recursive** program and see how it would change to **solve the same problem** but using an **iterative** approach

Recursive approach

```
01 def countdown_rec(n):
02    print(n)
03    if n == 0:
04        return
05    countdown_rec(n -1)
06 countdown_rec(10)
```

Iterative approach

```
01 def countdown_rec(n):
02   while n > 0:
03     print(n)
04     n = n-1
05   return
06 countdown_rec(10)
```

- The **recursive function** call on line **05** has been replaced with a **while loop** (line **02**) which checks if n > 0
- Using an **iterative** approach we use exactly the same amount of code (6 lines) **BUT**...
  - less memory would be used (**increased performance**)
  - **easier to debug**

Your notes

Worked Example

Hugh has written a recursive function called thisFunction() **using pseudocode.**

```
01 function thisFunction(theArray, num1, num2, num3)
02   result = num1 + ((num2 - num1) DIV 2)
03   if num2 < num1 then
04     return -1
05   else
06     if theArray[result] < num3 then
07       return thisFunction(theArray, result + 1, num2, num3)
08     elseif theArray[result] > num3 then
09       return thisFunction(theArray, num1, result - 1, num3)
10     else
11       return result
12     endif
13   endif
14 endfunction
```

Rewrite the function thisFunction() **so that it uses iteration instead of recursion.**
You should write your answer using pseudocode or program code.

6 marks

How to answer this question:

- The lines which call the function recursively are lines 07 and 09 so these are the lines which need to be changed
- To ensure the code is repeated, we can put lines 02 to 13 in a while loop:

```
while true
```

- The difference between line 07 and the function declaration on line 01, is that num1 is replaced with result + 1 so we'll need to set num1 equal to result + 1

```
num1=result+1
```

- The difference between line 9 and the function declaration on line 1, is that num2 is replaced with result - 1 so we'll need to set num2 equal to result - 1

```
num2=result-1
```

- All other lines of code remain the same

Answer:

```
function thisFunction(theArray, num1, num2, num3)

  while (true)

    result = num1 + ((num2 - num1) DIV 2)
```

```
    if num2 < num1 then
      return -1
    else
      if theArray[result] < num3 then
        num1 = result + 1
      elseif theArray[result] > num3 then
        num2 = result - 1
      else
        return result
      endif
    endif
  endwhile
endfunction
```

Examiner Tips and Tricks

Some questions will ask you to write an answer in **pseudocode**, whereas others will ask for pseudocode or program code. If it says you can write in program code, you can write in a language you choose, e.g. Python or Java. You won't be asked to name the language but do use the syntax from the language you're using if you're not using pseudocode

Your notes

Global & Local Variables

Your notes

Global Variables

What is a Global Variable?

- A global variable is a **variable** declared at the outermost level of a program. This means that they are declared outside any modules such as functions or procedures.
- Global variables have a global scope, which means they can be accessed and modified from any part of the program.

Python example

In this python code, you can see that the globalVariable (with the value 10) is declared **outside** of the function printValue. This means that this function and **any other modules** can access and change the value in the global variable.

```
globalVariable = 10 # Defines a global variable

def printValue():
    global globalVariable # Access the global variable inside a function
    print("The value into the variable is:", globalVariable)

printValue() # Call the function
```

Usage & need for global variables

- **Data Sharing**: Global variables facilitate data sharing between different parts of the program, allowing data to be passed between different modules easily without the use of parameter passing.
- **Persistent Storage**: Global variables retain their values throughout the program's execution, making them suitable for storing data that needs to persist across function calls
- **Global Configuration**: Global variables can be **used to store configuration settings or constants** that are relevant across the entire program

Benefits and drawbacks

Benefits	Drawbacks
The global variable only needs to be declared once.	The global variables are always stored in memory while the program is running which can use up memory.

You don't need to keep passing parameters between the different modules.	Makes the program difficult to maintain as it's difficult to understand where variables are changed in the program.
	Makes it difficult to test a single block of code as the programmer will need run the entire program to setup the global variables.


Your notes

Local Variables

What is a Local Variable?

- A local variable is a variable declared within a specific scope, such as a function or a code block
- Local variables are accessible only within the block in which they are defined, and their lifetime is limited to that particular block
- Once the execution of the block ends, the local variable is destroyed, and its **memory** is released

Python example

In this python code, you can see that the localVariable (with the value 10) is declared **inside** of the function printValue. This means that **only** this function can access and change the value in the local variable. It **cannot be accessed** by other modules in the program.

```
def printValue():
    localVariable = 10  # Defines a local variable inside the function
    print("The value of the local variable is:", localVariable)

printValue()  # Call the function
```

Benefits and drawbacks

Benefits	Drawbacks
Local variables encapsulate data within a particular function or block, providing data hiding and preventing unintended access from other parts of the program.	Repeatedly creating and destroying local variables within a loop or **recursive** function can incur unnecessary memory overhead.

Local variables enable you to use the same variable name in different functions or blocks without causing conflicts, as each local variable is confined to its own scope.	Excessive use of local variables within deeply **nested** blocks can lead to code clutter and reduce code readability.
Local variables have a limited lifetime, and their memory is automatically reclaimed when the code block ends, making them memory efficient.	


Your notes

Converting Between Global and Local Variables

- In this Python code below, a global variable named global_string is **declared** and initialised with the string "Hello"
- Two functions are defined, add_world and add_name
- The add_world function **appends** the string ", world!" to global_string
- The add_name function takes a string name as an **argument** and appends it to global_string
- add_world is **called** and prints global_string, which results in "Hello, world!"
- After that, add_name is called with the argument "Alice" and prints global_string again, which results in "Hello, world! Alice"

```
# Global variable
global_string = "Hello"

def add_world():
    global global_string
    global_string += ", world!"

def add_name(name):
    global global_string
    global_string += " " + name

# Call the functions
add_world()
print(global_string)  # Outputs: Hello, world!

add_name("Alice")
print(global_string)  # Outputs: Hello, world! Alice
```

- In this revised Python code, a local variable named local_string is declared and initialised with the string "Hello"
- Two functions are defined, add_world and add_name

- The add_world function takes a string s as an argument, appends the string ", world!" to it and returns the modified string
- The add_name function takes two arguments: a string s and a string name. It appends name to s and returns the modified string
- add_world is called with local_string as an argument and assigns the result back to local_string. local_string is printed, which resulted in "Hello, world!"
- add_name is called with local_string and "Alice" as arguments and assigns the result back to local_string. local_string is printed again, which results in "Hello, world! Alice"

```
def add_world(s):
    s += ", world!"
    return s

def add_name(s, name):
    s += " " + name
    return s

# Initialize a local variable
local_string = "Hello"

# Call the functions
local_string = add_world(local_string)
print(local_string)  # Outputs: Hello, world!

local_string = add_name(local_string, "Alice")
print(local_string)  # Outputs: Hello, world! Alice
```


Your notes

Your notes

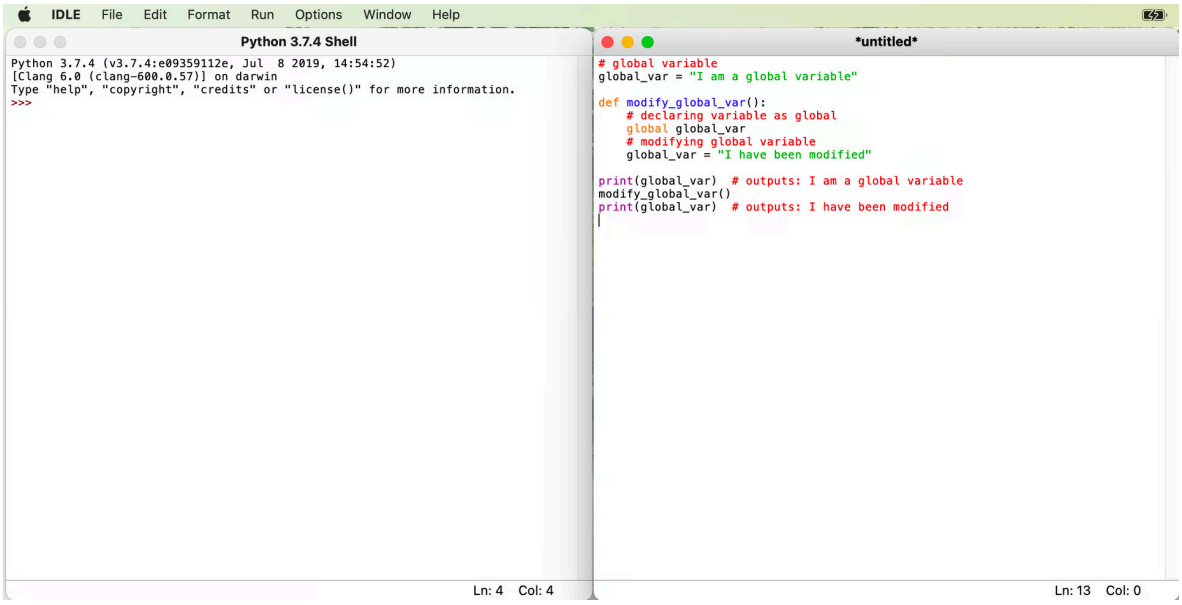
Integrated Development Environment (IDE)

What is an IDE?

- An Integrated Development Environment (IDE) is a software tool that provides programmers with a comprehensive and integrated platform to write, edit, **compile**, **debug**, and manage their code efficiently
- IDEs are designed to streamline the software development process by offering a wide range of features and tools that can help programmers
- IDEs can be an app to download or can be web-based

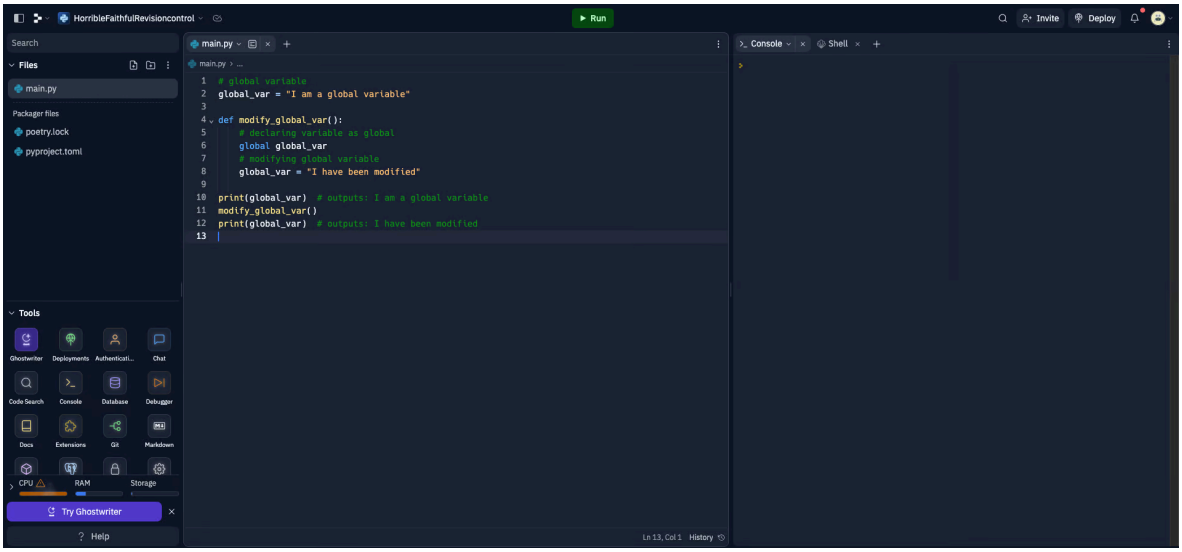
Example IDE's

IDLE is a basic IDE that is used for writing and running Python code.



IDLE software which is used to write and run Python code

Another example IDE is Replit. This allows programmers to write and run code as well as collaborate with other programmers online. It supports many programming languages such as Python, JavaScript and C++.



Replit which is web-based software used to write and run code in different languages

Examiner Tips and Tricks

Before answering questions on IDE's, make sure you read the question first.

Some questions may ask you to provide features of IDE's that can be used to:

- **write** programming code
- **debug** programming code

Some of the features that you have learnt cannot be used for both of these. For example autocomplete and auto indent can be used for writing programming code, however, are not useful for debugging programming code.

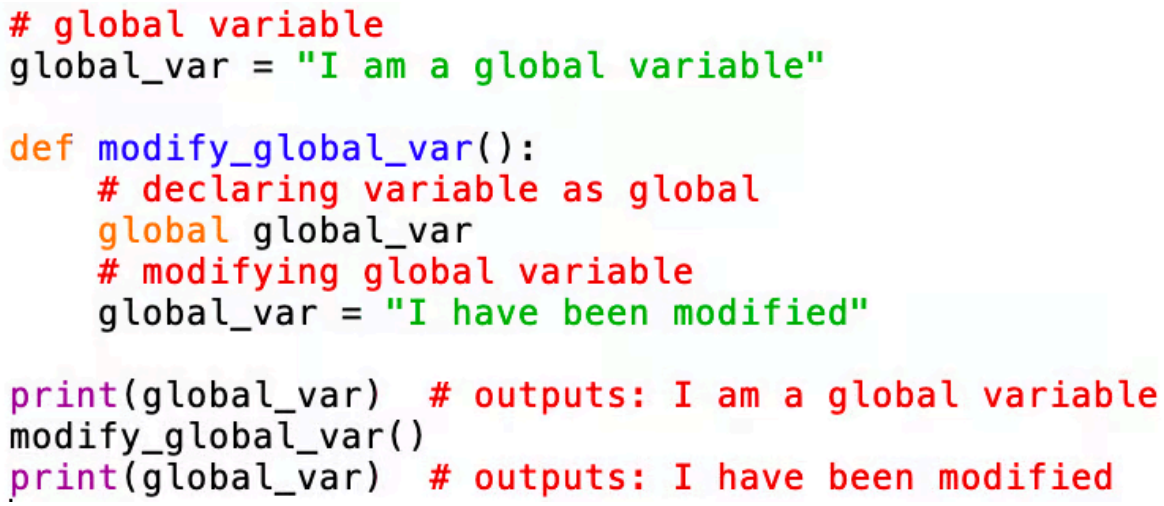
Features on an IDE

Syntax highlighting (pretty printing)

- IDEs use syntax highlighting to visually distinguish different elements of code based on their syntax
- The IDE assigns distinct colours or styles to keywords, strings, comments, variables, functions, and other language constructs
- Syntax highlighting helps programmers quickly identify and differentiate code components, reducing the chance of syntax errors and improving code comprehension

Your notes

- In this example Python code, you can see that the code comments are displayed in red, the strings are displayed in green and the print commands are displayed in purple

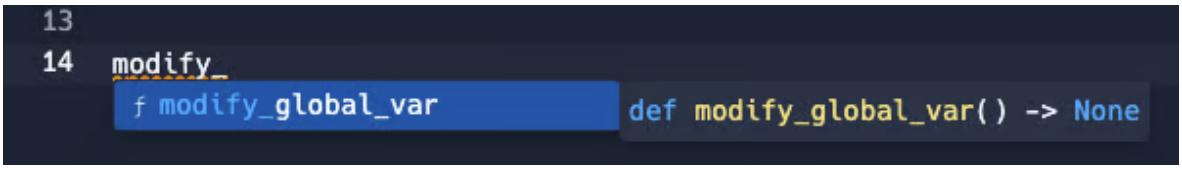


```
# global variable
global_var = "I am a global variable"

def modify_global_var():
    # declaring variable as global
    global global_var
    # modifying global variable
    global_var = "I have been modified"

print(global_var)  # outputs: I am a global variable
modify_global_var()
print(global_var)  # outputs: I have been modified
```

Syntax highlighting in an IDE

Autocomplete

- Autocomplete will suggest code completions as programmers type
- For example, as developers start typing a variable, function, or method name, the IDE displays a list of available options
- Another example is when programmers use opening brackets and the autocomplete facility will automatically add closing brackets
- Code completion **improves productivity** by reducing typing effort and preventing typing errors and naming inconsistencies
- IDEs can automatically correct common mistakes or typing errors made during coding. When an error-prone sequence is detected, **the IDE can suggest a correction**, helping developers fix minor errors as they type their code



Code completion in an IDE

Your notes

Auto indent

- Auto indent will automatically indent the programming code when you start a new line of code within a code block
- They are commonly used in selection and iteration programming constructs
- In the example below, the indentation is automatic after starting to define the code block on line 04 and is clearly marked to show which lines of code are included within the block.

```
4 def modify_global_var():
5     # declaring variable as global
6     global global_var
7     # modifying global variable
8     global_var = "I have been modified"
9
```

Code formatting with automatic indentation

Code comments

- Comments are lines of text within the code that are not executed but serve as documentation and explanations for programmers
- IDEs enable developers to add single-line or multi-line comments to describe the code's purpose, algorithmic steps, and any relevant information that will be useful to the maintenance of the program
- Comments enhance code understanding, making it easier for developers to revisit and modify code at a later stage
- Comments are in green on the image above

Syntax error highlighting

- IDEs can automatically highlight syntax errors in the code editor as developers type
- This feature helps catch mistakes instantly, allowing programmers to address syntax-related issues promptly
- Syntax errors are highlighted with distinct colours, making them easily noticeable, even before executing the code

Variable watch window

- The watch window is a debugging tool that allows developers to monitor the values of specific variables during **runtime**

Your notes

- Developers can add variables to the watch window and observe their values change as the program executes
- This tool is particularly valuable for understanding how variables evolve throughout the program's execution and diagnosing issues related to variable values


Your notes

Breakpoints

- Breakpoints are markers that developers can set at specific lines of code to halt program execution during **runtime**
- When the program reaches a breakpoint, it pauses, allowing developers to inspect the program's state and variable values and identify the cause of potential bugs
- Breakpoints provide a controlled way to analyse code step-by-step

Error message list

- The error message list displays a collection of errors, warnings, and messages generated during the compilation and execution of the program
- IDEs present detailed error descriptions and relevant code locations, enabling programmers to address and correct issues systematically

Stepping mode

- Stepping mode is a debugging mode that allows developers to execute the program one line at a time
- Developers can choose between different step options, such as step into, step over, and step out, to examine function calls and control flow
- Step mode offers a granular view of the program's execution, making it easier to identify the exact location of potential bugs

Crash dump/post-mortem report

- When a program crashes, IDEs can generate a crash dump or post-mortem report
- This report captures the program's state and **memory** contents at the time of the crash, helping developers diagnose the root cause of the crash after the program has terminated unexpectedly

Stack contents

- The stack contents tool allows developers to inspect the contents of the program's call stack during debugging
- It shows the order in which functions were called and the parameters and local variables of each function

- Understanding the call stack, aids in understanding the program's flow and diagnosing issues related to function calls


Your notes

Programming Classes, Objects, Methods & Attributes


Your notes

Programming Classes

- To program classes, you should already have a solid understanding of what classes in Object Oriented Programming (OOP) are.

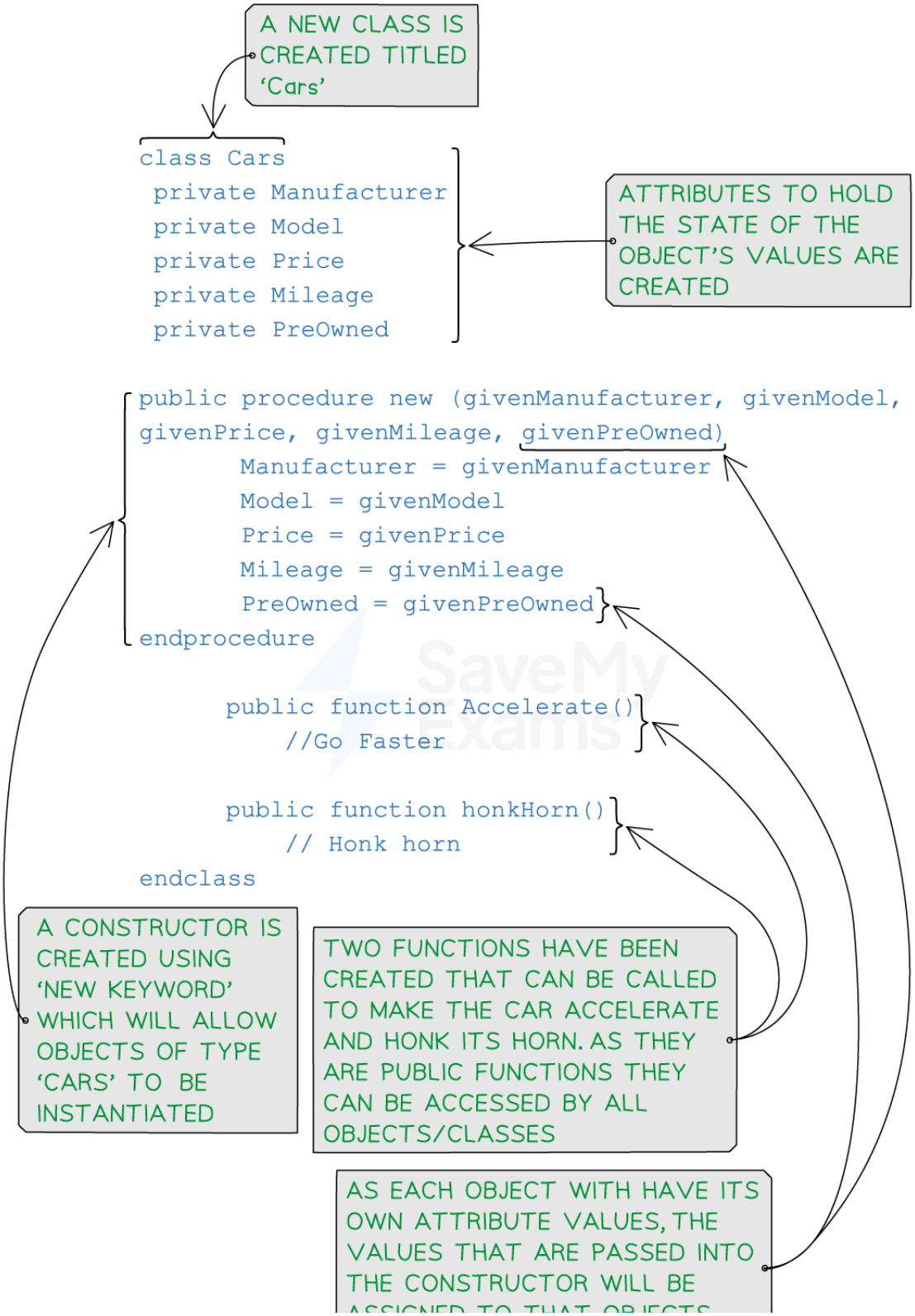
How do you Define a Class?

In this example we will:

- Define a class for a car sales program where the following is required:
  - Name of Manufacturer
  - Model of Car
  - Price
  - Mileage
  - Whether the car is preowned
- Instantiate two objects for the car sales program ensuring that they have appropriate identifiers
- List two methods that would be useful for a car object

Pseudocode

Your notes



Your notes

Pseudocode for creating classes, a constructor, and functions

Java

```
//creating a class called cars
public class Cars {
//creating class attributes for a car
   private String Manufacturer;
   private String Model;
   private int Price;
   private int Mileage;
   private boolean PreOwned;
   //Constructor - This is used to create the objects. More on this in the object chapter)
   public Cars(String manufacturer, String model, int price, int mileage, boolean preOwned) {
      this.Manufacturer = manufacturer;
      this.Model = model;
      this.Price = price;
      this.Mileage = mileage;
      this.PreOwned = preOwned;
   }
   //method to make the car go faster.
   public void Accelerate(){
      //code to be executed which will make the car go faster
   }
   //Method to honk horn
   public void honkHorn(){
      // code to be executed which will honk the car horn
```

```
  }
}
```

Python

```
#creating a class called cars
class Cars:
  # constructor - This is used to create the objects. More on this in the object chapter)
  def __init__(self, manufacturer, model, price, mileage, preOwned):
    self.Manufacturer = manufacturer
    self.Model = model
    self.Price = price
    self.Mileage = mileage
    self.PreOwned = preOwned

  #method to make the car go faster.
  def Accelerate(self):
    # code to be executed which will make the car go faster

  # method to honk horn
  def honkHorn(self):
    # code to be executed which will honk the car horn
```

Programming Objects

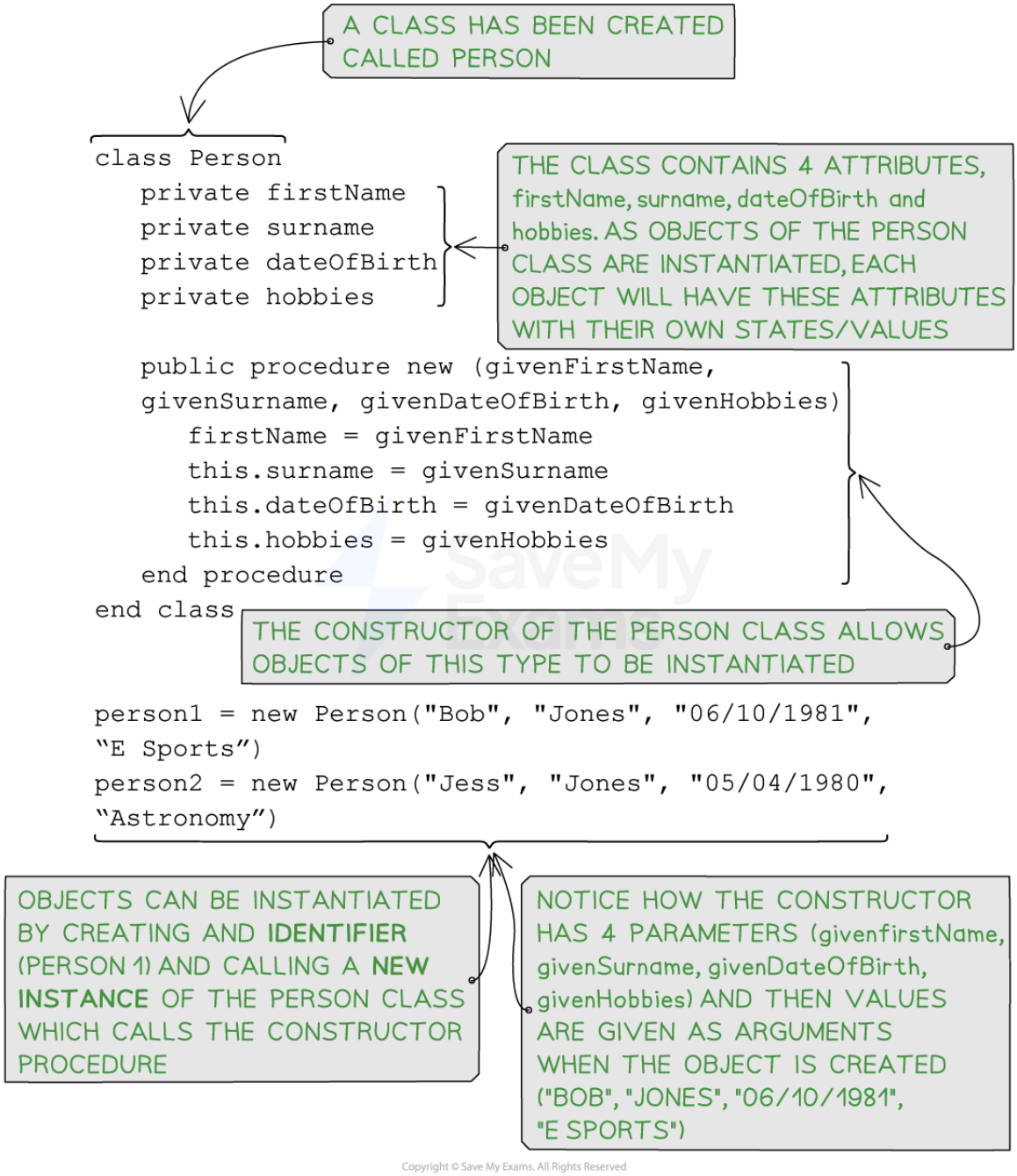
- To program objects, you should already have a solid understanding of what objects in Object Oriented Programming (OOP) are.

Defining an Object

Pseudocode

Your notes

Your notes



Pseudocode for the class 'person' and instantiating two objects

Java

Your notes

```
//creating the person class
public class Person {
        // creating 4 attributes for the person class
    private String firstName;
    private String surname;
    private String dateOfBirth;
    private String hobbies;

    // Constructor -This creates objects of the person class
    public Person(String firstName, String surname, String dateOfBirth, String hobbies) {
        this.firstName = firstName;
        this.surname = surname;
        this.dateOfBirth = dateOfBirth;
        this.hobbies = hobbies;
    }
//Creating Objects (Instances) of the person class
Person person1 = new Person("Bob", "Jones", "06/10/1981", “E Sports”);
Person person2 = new Person("Jess", "Jones", "05/04/1980", “Astronomy”);
```

Python

```
#creating the person class
class Person:
      #Constructor -This creates objects of the person class
    def __init__(self, firstName, surname, dateOfBirth, hobbies):
        self.firstName = firstName
        self.surname = surname
        self.dateOfBirth = dateOfBirth
        self.hobbies = hobbies
```

Your notes

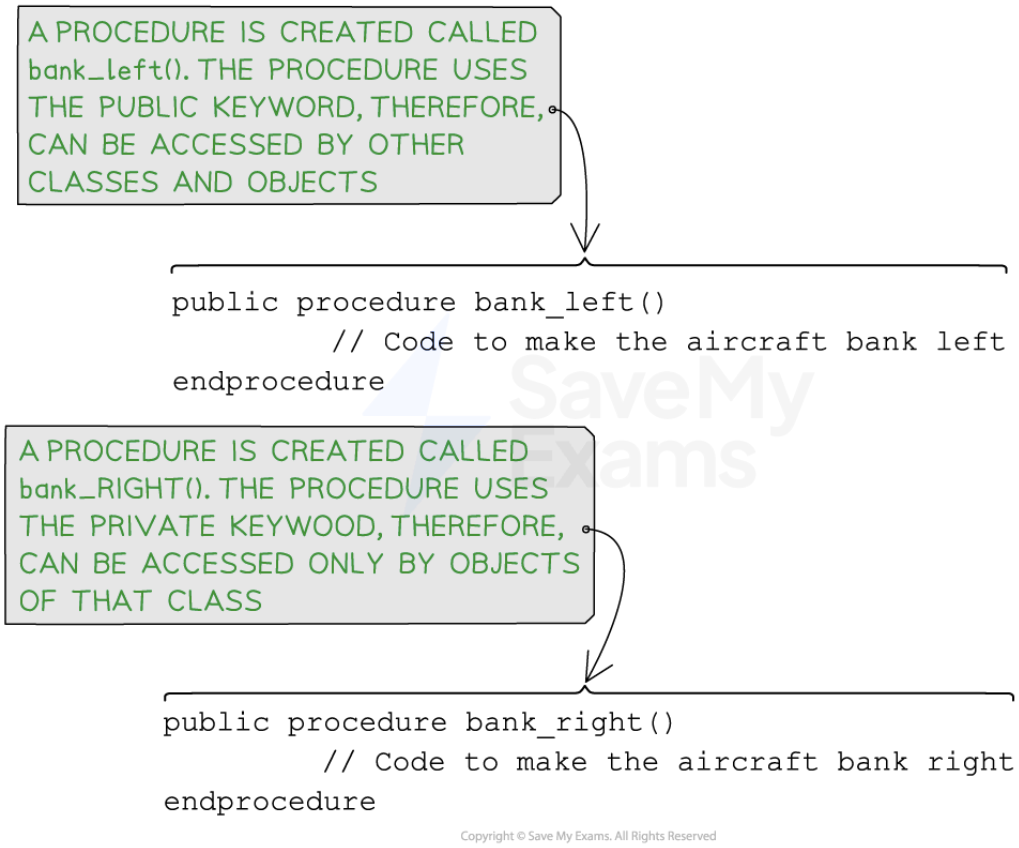
```
#Creating Objects (Instances) of the person class

person1 = Person("Bob", "Jones", "06/10/1981", "E Sports")

person2 = Person("Jess", "Jones", "05/04/1980", "Astronomy")
```

Programming Methods

- To program methods, you should already have a solid understanding of what methods in Object Oriented Programming (OOP) are.

Defining Methods

Pseudocode



Pseudocode for creating methods

Your notes

Java

```
//Creating a public method called BankLeft

public void bankLeft() {

 // Code to make the aircraft bank left

   }

//Creating a private Method called BankLeft

public void bankLeft() {

 // Code to make the aircraft bank left

   }
```

Python

```
#Creating a public method called BankLeft

def bank_left(self):

      # Code to make the aircraft bank left

#Creating a private method called BankLeft
```

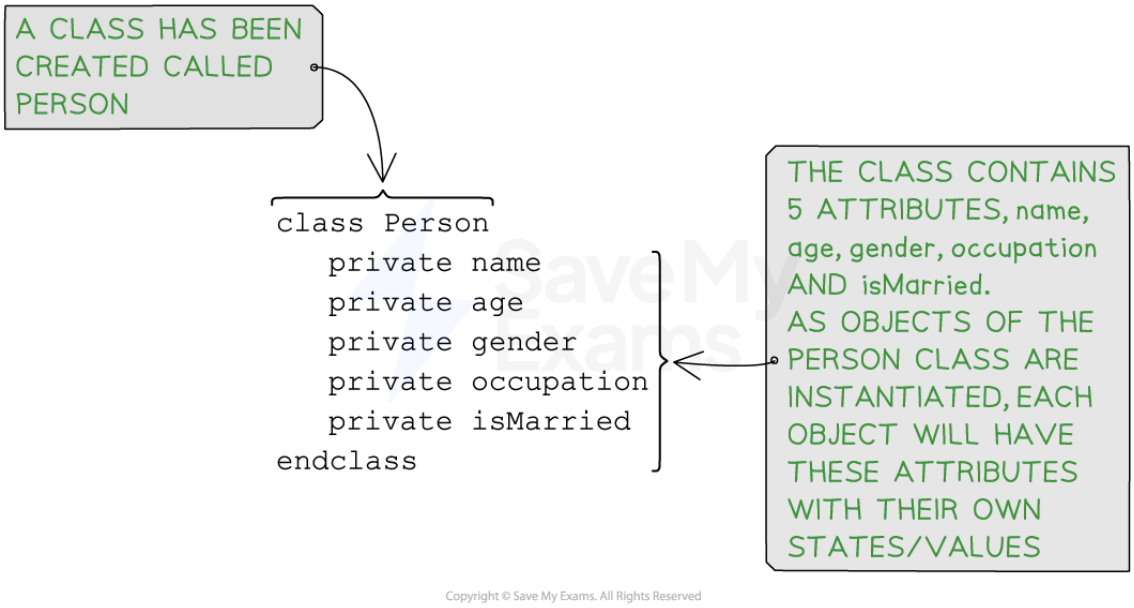
- In Python, there is no explicit **access modifier** for methods like "private" in Java.
- By convention, methods with a single leading underscore _ are considered private and should not be accessed directly from outside the class.

```
_def _bank_left(self):

      # Code to make the aircraft bank left
```

- It's important to note that even though the method is marked as private, it can still be accessed and **invoked** from outside the class. However, as a rule, other developers should treat it as private and avoid accessing it directly.

Programming Attributes

Pseudocode



Example of a created class, "Person", containing several attributes

Your notes

Java

```
public class Person {

   // Attributes for the person class

   private String name;

   private int age;

   private String gender;

   private String occupation;

   private boolean isMarried;

}
```

Python

In Python attributes are defined using the self keyword followed by the attribute name and its initial value.

```
class MyClass:

  def __init__(self, attribute1, attribute2):
```

```
# Define attributes

self.attribute1 = attribute1

self.attribute2 = attribute2
```

Worked Example

Below is a sample of code for the class "Lizard".

```
class Lizard
  private speed
  private mass
  private size
  public procedure new(givenSpeed, givenMass, givenSize)
    speed=givenSpeed
    mass=givenMass
    size=givenSize
  endprocedure
  public function breakBlock(brick)
    if speed*mass>=brick.getStrength() then
      speed=((speed*mass)-brick.getStrength())/mass;
      return true
    else
      return false
    endif
  endfunction

...
...
...
endclass
```

Identify an attribute in the Lizard class.

1 marks

How to answer this question to get full marks:

- There are 3 attributes in the Lizard class which are speed, mass and size

Answer:

Speed

Your notes

Your notes

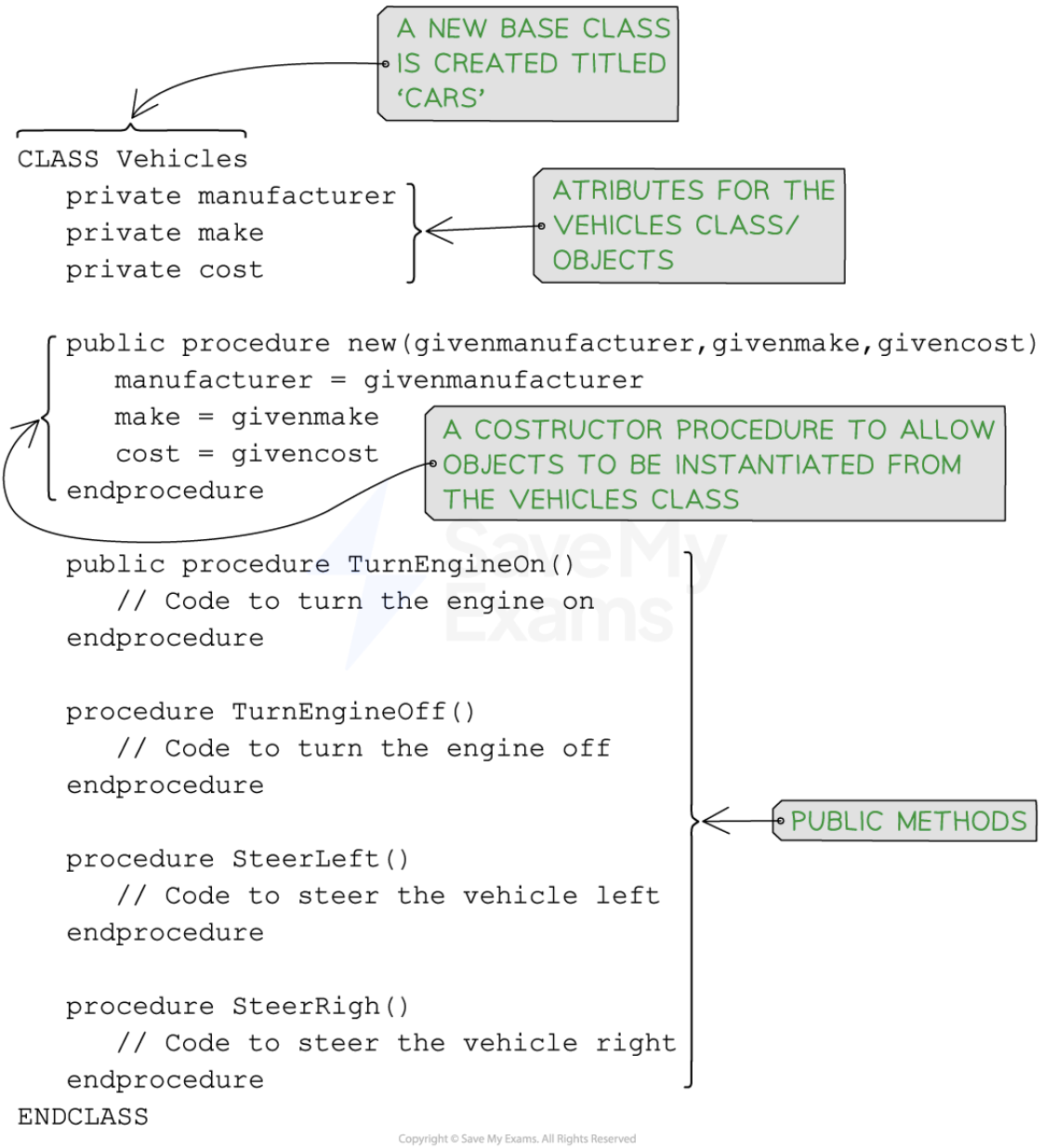
Programming Inheritance

- To program inheritance, you should already have a solid understanding of what inheritance in Object Oriented Programming (OOP) is.

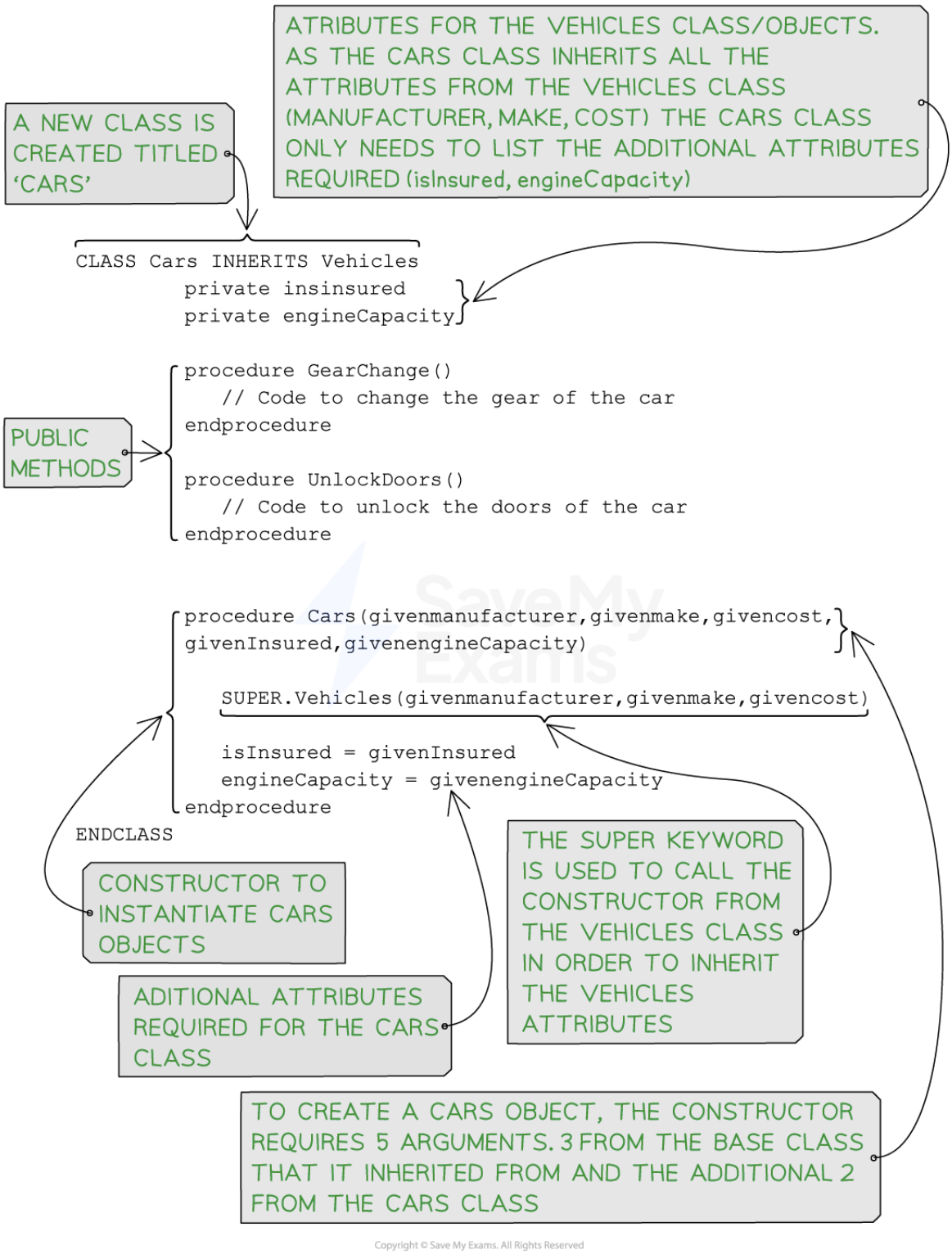
How do you Define Inheritance?

Pseudocode

Your notes



Your notes



Pseudocode for a class created using inheritance

Your notes

Java

```
//vehicle class created
public class Vehicle {
    //attributes created for the vehicle class
   private String manufacturer;
   private String make;
   private double cost;

   //constructor to create objects of type vehicle
   public Vehicle(String manufacturer, String make, double cost) {
      this.manufacturer = manufacturer;
      this.make = make;
      this.cost = cost;
   }

   //method to start the engine of the car
   public void turnEngineOn() {
      //code to turn the engine on
   }
   //method to stop the engine of the car
   public void turnEngineOff() {
      //code to turn the engine off
}
   //method to steer the car to the left
   public void steerLeft() {
      // code to steer the vehicle left
```

```
    }
    //method to steer the car to the right
    public void steerRight() {
        // code to steer the vehicle right
    }
}

//Derived Class (Car)
//the Car class uses the keyword 'extends' to inherit from the Vehicle class
public class Car extends Vehicle {
    private boolean isInsured;
    private double engineCapacity;

    // Constructor to create objects of type Car
    public Car(String manufacturer, String make, double cost, boolean isInsured, double   engineCapacity) {
        //the super keyword is used to inherit the attributes from the base class
        super(manufacturer, make, cost);
        this.isInsured = isInsured;
        this.engineCapacity = engineCapacity;
    }
    //method to change gear
    public void gearChange() {
        //code to change the gear of the car
    }
    //method to unlock the vehicle doors
    public void unlockDoors() {
        //code to unlock the doors of the car
```

Your notes

```
  }
}
```

Your notes

Python

```
#Base class (Vehicle)

#Vehicle class created

class Vehicle:

    #attributes created for the Vehicle class

 def __init__(self, manufacturer, make, cost):

      self.manufacturer = manufacturer

      self.make = make

      self.cost = cost


    #method to start the engine of the car

 def turn_engine_on(self):

      # Code to turn the engine on


   #method to stop the engine of the car

   def turn_engine_off(self):

      # Code to turn the engine off


    #method to steer the car to the left

   def steer_left(self):

      # Code to steer the vehicle left


    #method to steer the car to the right

   def steer_right(self):

      # Code to steer the vehicle right
```

Your notes

```
#Derived Class (Car)
#the Car class uses the keyword parent class name in parenthesis to inherit from the Vehicle class
class Car(Vehicle):
  def __init__(self, manufacturer, make, cost, is_insured, engine_capacity):
    #the super keyword is used to inherit the attributes from the base class
    super().__init__(manufacturer, make, cost)
    self.is_insured = is_insured
    self.engine_capacity = engine_capacity
  #method to change gear
  def gear_change(self):
    # Code to change the gear of the car

   #method to steer the car to the right
  def unlock_doors(self):
    # Code to unlock the doors of the car
```

Your notes

Programming Encapsulation

- To program encapsulation, you should already have a solid understanding of what encapsulation in Object Oriented Programming (OOP) is.

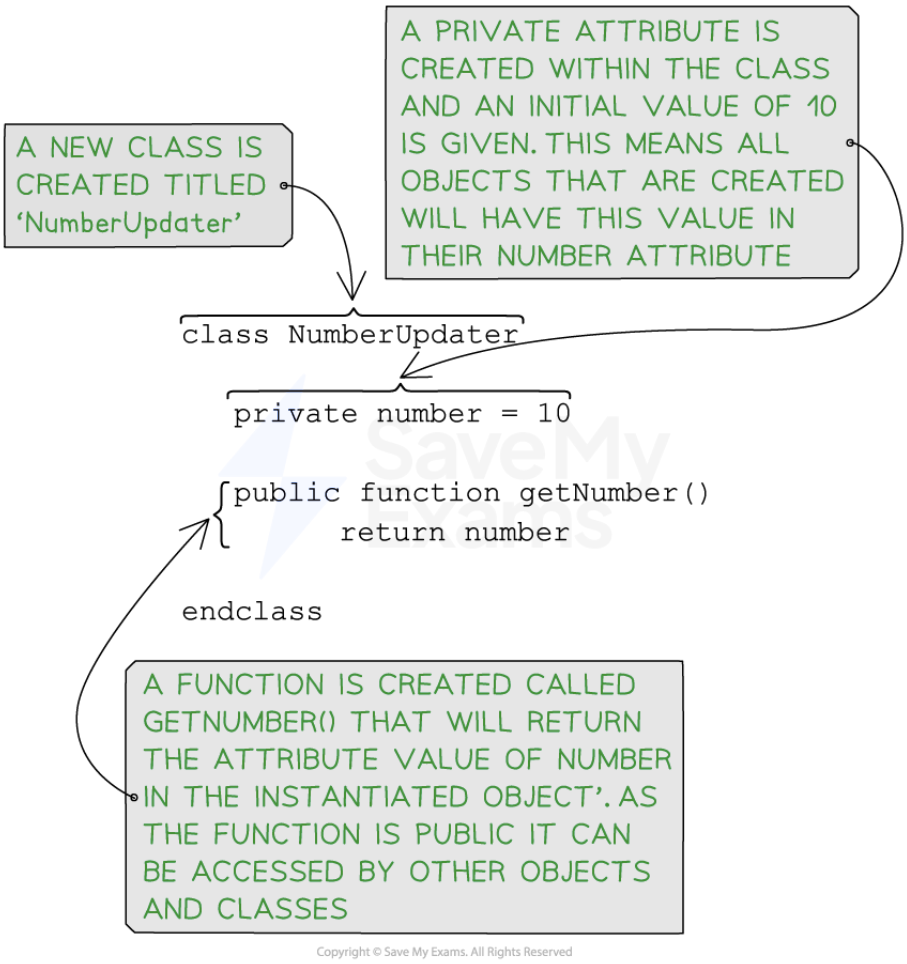
How do you Define Encapsulation?

Get Methods

- The "get" **method** is a common naming convention for a type of method that is used to retrieve the value of an object's private **attributes** (instance variables).
- These methods are also known as "getter" methods or "accessor" methods.
- The main purpose of a get method is to provide controlled access to the internal state (data) of an object without allowing direct modification.
- **Attributes** are often declared as private to achieve **encapsulation**, so cannot be accessed directly from outside the class.
- External code can use the public get methods to read the values of these private attributes (also known as instance variables).

Pseudocode

Your notes



Pseudocode for a get method

Java

```
//creating a class called NumberUpdater
public class NumberUpdater {
   //setting the private attribute to the value of 10
   private int number = 10;
   //creating a get method that will return the value store in the number attribute
   public int getNumber() {
```

```
    return number;
  }
}
```

Your notes

Python

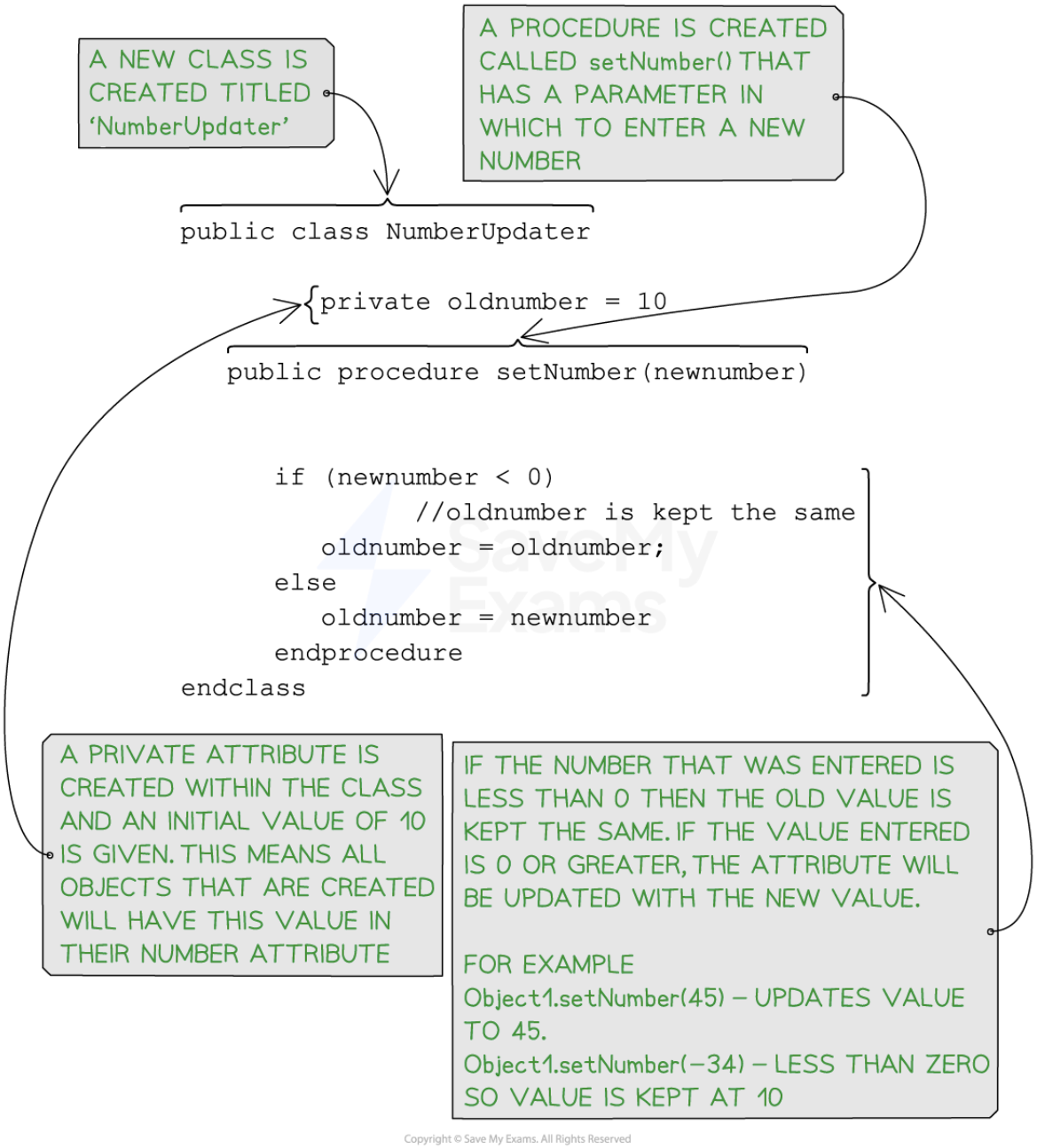
```
#creating a class called NumberUpdater
class NumberUpdater:
  #setting the private attribute to the value of 10
 def __init__(self):
    self.__number = 10
  #creating a get method that will return the value stored in the number attribute
  @property
  def number(self):
    return self.__number
```

Set Methods

- The "set" method is a common naming convention for a type of method that is used to set the value of object's private instance variables.
- Setter methods, also known as “setter” methods or “mutator” methods
- Generally additional code is written within the method to allow controlled access by enforcing certain conditions or validations before updating the attribute

Pseudocode

Your notes



Pseudocode for a "setter" method

Java


Your notes

```
//creating a class called NumberUpdater
public class NumberUpdater
    //setting attribute value to 10
   private int oldnumber = 10;
   //creating a set method
   public void setNumber(int newnumber) {
       //if the new number is less than zero
    if (newnumber < 0) {
          //oldnumber is kept the same
      oldnumber = oldnumber;
       } else {
        //oldnumber is updated with the new number value
            oldnumber = newnumber;
       }
   }
}
```

Python

```
//creating a class called NumberUpdater
class NumberUpdater:
    //setting attribute value to 10
  def __init__(self):
      self.__oldnumber = 10
```


Your notes

```
  //creating a set method
def setNumber(self, newnumber):
    //if the new number is less than zero
    if newnumber < 0:
         //oldnumber is kept the same
        self.__oldnumber = self.__oldnumber
     else:
        //oldnumber is updated with the new number value
         self.__oldnumber = newnumber
```

Your notes

Programming Polymorphism

- To program polymorphism, you should already have a solid understanding of what polymorphism in Object Oriented Programming (OOP) is.

How do you Define Polymorphism?

Pseudocode

1. Create a blueprint for a general animal:

- Define a template called Animal
- Give Animal a placeholder action called speak() that does nothing yet

2. Create a blueprint for a dog:

- Define a template called Dog based on the Animal template
- Override the speak() action for Dog to make it print "Woof"

3. Create a blueprint for a cat:

- Define a template called Cat based on the Animal template
- Override the speak() action for Cat to make it print "Meow"

4. Create a function to make animals speak:

- Define a function called make_sound(animal)
- Inside the function, tell the given animal to perform its speak() action

5. Create a dog and a cat:

- Build a Dog object and put it in a box labeled "dog"
- Build a Cat object and put it in a box labeled "cat"

6. Ask the dog and cat to speak:

- Call the make_sound() function with the dog object
- Call the make_sound() function with the cat object

Python

```
class Animal:
    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        print("Woof")

class Cat(Animal):
    def speak(self):
        print("Meow")

def make_sound(animal):
    # calling the speak method of the Animal class
    animal.speak()

dog = Dog()
cat = Cat()

make_sound(dog)
make_sound(cat)
```

Java

```
class Animal {
    public void speak() {
        // Empty default implementation
    }
}

class Dog extends Animal {
    @Override
    public void speak() {
        System.out.println("Woof");
    }
}

class Cat extends Animal {
    @Override
    public void speak() {
        System.out.println("Meow");
    }
}

public class Main {
    public static void makeSound(Animal animal) {
```


Your notes

```
    animal.speak();
  }

  public static void main(String[] args) {
    Dog dog = new Dog();
    Cat cat = new Cat();

    makeSound(dog);
    makeSound(cat);
  }
}
```


Your notes