Notes and guidance: 8525/1 Marking guidance

Guidance for teachers on marking questions with a coded solution

This guidance has been produced to assist teachers and to ensure that students are treated as fairly as possible irrespective of the programming language they have used.

General principles

In the real-world minor syntax errors are often identified and flagged by the development environment. To reflect this, all responses in a high-level programming language will assess a student's ability to create an answer using precise programming commands/instructions but will avoid penalising them for minor errors in syntax.

When marking program code, you must take account of the different rules between the languages and only consider how the syntax affects the logic flow of the program. If the syntax is not perfect but the logic flow is unaffected then the response should not be penalised.

The case of all program code written by students is to be ignored for the purposes of marking. This is because it is not always clear which case has been used depending on the style and quality of handwriting used.

Additional spaces in code can be ignored except in identifier names. See the section on Meaningful Variable Names.

Teachers must ensure they follow the mark scheme instructions exactly.

If the intent of the student is clear from the code written then, where possible, marks should be awarded. This applies even if the methods used are unfamiliar to you as a teacher.

Writing a program on paper is hard when you have been used to the help offered by an IDE.

From the *Additional Practice Questions* document:

As detailed in the mark scheme, the case of text will be ignored and indentation will only be taken account in so far as the logic flow must be clear. Similarly, if punctuation is missing (eg semicolons, colons etc) marks can be awarded as long as the logic is clear.

It is expected that the majority of students will give responses that are simply and easily marked, but this guidance should help teachers to apply general principles if that is not the case.

Design marks

Most questions will have some design marks available, which may be awarded even if the code contains major errors. However, there must be some genuine and relevant attempt at writing a program that meets the requirements of the question, eg if there are design marks available for the use of both a selection construct and an iteration construct, merely writing:

```
Do
   If
   End If
Loop
```

should not be awarded any marks.

Marking syntax

This area will be very mark scheme dependent. It is important that the requirements of the mark scheme for the particular question are followed to the letter.

Question Example

Write a selection statement to output the message `"True"` if `a` contains a value greater than or equal to zero.

Example 1

If the question requires, and the mark scheme contains a mark for correctly using, a selection statement and the student includes a syntactically correct iteration statement instead, such as (in Python):

```
while a >= 0:
```

then the student cannot gain the mark even though the statement they have written is syntactically correct, as it does not match the requirement of the question/mark scheme combination.

Example 2

If the mark scheme contains a separate mark for testing whether `a >= 0` and the student writes (in Python):

```
while a >= 0:
```

then the mark would be awarded even though an incorrect program statement had been used.

However, if the student had written `while a > 0:` then neither of the marks in the two examples could be awarded.

Additional marking syntax guidance

1. If the student writes a statement that is syntactically incorrect, eg in Python

   `if a >= 0`   *Missing :*

   or in VB.NET

   `If a >= 0`   *Missing* `Then`

   or in C#

   `if a >= 0`   Missing *( )*

then any mark for the `IF` statement is still awarded as the logic and intent of the student is clear.

2. Missing spaces **within** a statement, eg `Ifa>=0Then` may be ignored — it may not be obvious from a student's writing that spaces are there.
3. Superfluous spaces and blank lines may be ignored.
4. Where there is a **Maximum *n* marks if any errors in code**, minor errors will not be penalised, so that, for example, missing a colon off the end of an `if` statement in Python, or missing the `;` off the end of a statement in C# will not be penalised.
5. Missing `import` (Python), `using` (C#) or `Imports` (VB.NET) statements should be ignored, eg a student using `ReadLine()` or `Write()`/`WriteLine()` without the `Imports System.Console`/`using System.Console;` statements (VB.NET/C#), or in Python `randrange` or `randint` without an `import random` statement.

Indentation and statement blocks

General rules for indicating block structure

For all languages, the first statement after an `if`/`while`/`do`/`else` etc should be marked as if they were 'under the control' of the statement. For Python, statements after the first one that are **not** indented correctly should be marked as not being under that control **unless** they are followed by `else`/`elif`/`case`.

For C# this rule also applies but instead of 'are not indented' is replaced by 'are not enclosed in { }".

For VB.NET this rule also applies (if a student misses off the `End If` for example).

Python

If a student has **consistently** used their own indentation guides, eg starting an indented block half way between indentation guides, then this should not be penalised as long as the indented levels match up.

If a student indents too much this may be ignored as long as the indented levels match up.

If a student does not indent enough then it is possible that not all marks will be awardable as it may be unclear what is indented and what is not. Each case will need to be marked on its merit.

It is all about the logic flow and whether the intention of the logic flow is clear enough.

For example, a password checking program requires a message to be output if the entered password is not `Bananas` and the number of tries left decreased by one:

```
if password != "Bananas":
   print("Incorrect password.")
tries = tries - 1
```

would gain marks for the `if` statement, test, and `print` statement, but not gain marks for decrementing `tries` (unless the mark scheme has a mark for this without the usual requirement for the statement to be executed only under all correct circumstances)

A similar approach is taken with `while` and `for` statements.

However, when an `if … elif … else:` is used, then **apart from** the `else` block this rule can be relaxed.

```
if guess > target:
   print("Too high.")
tries = tries + 1
elif guess < target:
print("Too low.")
tries = tries + 1
else:
print("Well done.")
print(tries, "attempts.")
```

would gain any marks for the `if` statement, test, `print` statement, and incrementing `tries`, but not gain marks available for the `print(tries, "attempts.")` message within the `else` part (`print("Well done.")` would be marked correct under the general rule)

VB.NET

Since VB.NET uses keywords (eg `End If`, `Loop`), to indicate the end of a statement block, indentation (or the lack of it) can be ignored as long as the keyword(s) are not missing, but if indentation is present it may be used to compensate for mistakes elsewhere, eg a missing 'bracketing' keyword after `End`, or even missing `End`s.

```
If guess > target Then
Console.WriteLine("Too high.")
tries = tries + 1
Else If guess < target Then
Console.WriteLine("Too low.")
tries = tries + 1
Else
   Console.WriteLine("Well done.")
   Console.WriteLine(tries, "attempts.")
' Missing End If here
' Code continues here unindented
```

ignores missing indentation in the first and second `If` blocks, but the indentation in the `Else` block makes up for the lack of the `End If`.

If the code continued at the same indentation as the last two `Console.WriteLine`s any marks depending on them only being executed under correct circumstances could not be awarded.

C#

Since C# uses `{` and `}` braces to indicate the start and end of statement blocks, indentation (or the lack of it) can be ignored as long as the braces are not missing, but if indentation is present it may be used to compensate for mistakes elsewhere, eg a missing brace either at the start or the end of the block.

```
if guess > target {
Console.WriteLine("Too high.");
tries = tries + 1;
} else if (guess < target) {
Console.WriteLine("Too low.");
tries = tries + 1;
else
   Console.WriteLine("Well done.");
   Console.WriteLine(tries, "attempts.");
// Missing { after else and } missing here
// Code continues here unindented
```

ignores missing indentation in the first and second `if` blocks, but the occurrence of the `else` without the required closing `}` would not affect the marks and the indentation in the `else` block itself makes up for the lack of the `{}`.

If the code continued at the same indentation as the last two `Console.WriteLine`s any marks depending on them only being executed under correct circumstances could not be awarded.

Specific statement types

If a student uses an AQA pseudo-code construct that is not available in their chosen language, eg `REPEAT UNTIL`, then any design mark may be awarded but no mark for the code itself.

C#

`if`, `while`, and `do` statements:

> ignore missing parentheses around tests (this is also repeated as a similar rule in **Expressions** but it's worth repeating again here.)

`for` and `foreach` statements:

> Ignore omission of the type for the control variable, accept the use of `var`, and ignore missing parentheses around the sections that control the loop (initialisation, condition, iteration in `for` and iteration only in `foreach`.).
>
> In a `for` statement the initialisation, condition and iteration parts must all be present, although missing semi-colons may be ignored, eg `for i=0 i<10 i++` would gain all marks (assuming a loop from 0 to 9 or for 10 iterations were required by the question.)

VB.NET

`For` and `For Each` statements:

> Ignore omission of the type for the control variable.

Python

`REPEAT UNTIL` loops may be coded using a standard `while` loop as in the Python Coding Guide or as a `while True:` loop with a break at the bottom, eg

```
a = 1
while True:
   print(a)
   a = a + 1
   if a == 4:
      break
# outputs 1, 2, 3
```

which is equivalent to the pseudo code

```
a = 1
REPEAT
   OUTPUT a
   a = a + 1
UNTIL a = 4
# outputs 1, 2, 3
```

Python 3.10 `match` statement

Python versions before 3.10 use the `if … elif … elif … else`: … structure to implement a `switch` (C#) or `Select Case` (VB.NET) statement where a single value is tested against a number or possibilities, eg testing a character to see if which of the four compass directions it indicates:

```
if dir == "E":
   direction = "East"
elif dir == "N":
   direction = "North"
elif dir == "S":
   direction = "South"
elif dir == "W":
   direction = "West"
else:
   direction = "Invalid direction"
```

Starting with Python 3.10 the `match` statement can be used instead:

```
match dir:
   case "E":
      direction = "East"
   case "N":
      direction = "North"
   case "S":
      direction = "South"
   case "W":
      direction = "West"
   case _:  # _ matches anything (default: C# / Case Else VB.NET)
      direction = "Invalid direction"
```

Variable names

In all languages, once a variable is declared and/or given an initial value, accept uses of the variable name which differ in case (only relevant for Python and C#), and minor misspellings **where the name cannot be confused with another**.

For example, in Python

```
Total_Stock = 0
…
Totalstock = Total_stock + 1
```

Meaningful variable names

Where the question asks the student to *use meaningful variable names* consideration should be given to the context of the question. For example, if a password were to be input twice and the inputs compared for equality, meaningful variable names could be

`password1` and `password2`

`pwd1` and `pwd2`

`pw1` and `pw2`

`p1` and `p2`

In the last case the `1` and `2` help with adding meaning and would represent the minimum that was creditworthy.

A student could concentrate on the fact that the second input should be a verification of the first, and use

`password` and `verification`

Single letter variable names **may** be acceptable, for example in loops

```
for i in range(20):
```

or in a question involving coordinates, where `x` and/or `y` will be meaningful, but in the password example using

`a` and `b`

would not be acceptable as meaningful variable names (they could, however, be meaningful if the question was on Pythagoras' theorem!)

Treat hyphens as a valid break character in composite variable names, and so treat `braking-distance` as `braking_distance` unless `braking - distance` is a valid arithmetic expression.

Variable names must not contain noticeable spaces and if present they will be considered an error except when the mark is a Design mark.

Keywords

If a keyword/type is misspelled or incorrectly cased, but is clearly correct, eg `Whiel` instead of `While`/`flaot` instead of `float`, then the student's response should be marked as if it were correctly spelled.

Expressions

In an arithmetical/logical/string expression

- missing parentheses may be ignored provided the normal precedence rules give the correct answer.

  eg adding the cost of a number of items purchased to a running total with the statement

  `total = total + (price * quantity)` is correct

  as is `total = total + price * quantity`

  but obviously a statement such as `total = (total + price) * quantity` is incorrect.

- superfluous parentheses may be ignored as long as they do not alter the (correct) order of evaluation.

- the use of non-standard mathematical operators (eg x instead of * for multiplication) should not be penalised as long as the meaning is unambiguous and clear. The only potential exception to this rule would be if a variable name was the same as the operator used and had introduced ambiguity.

In string expressions ignore missing " around a string if the string is the only thing being output or assigned to a variable, eg `print("Hello)` or `name = "User1` gets any available mark but `print("Hello + name)` does not.

In function calls ignore missing parentheses, eg `name = input` or `name = input "What is your name"`

Logical tests

Irrespective of the language:

> allow <> , != or ≠ as the symbol for *not equal to*.
>
> allow = or == as the symbol for *equal to*.
>
> allow >= or ≥ as the symbol for *greater than or equal to*.
>
> allow <= or ≤ as the symbol for *less than or equal to*.

In C# only, allow `|` or `||` (short circuiting or conditional) for logical OR, and `&` or `&&` (short circuiting or conditional) for logical AND.

C# relational tests on strings

In C# allow <, <=, >, >= when comparing strings **or** characters.

To compare two **strings** in C# and determine their order, eg, to determine if `string1` comes 'before' `string2`, students may write code like:

```
int result = string1.CompareTo(string2);
```

or

```
int result = String.Compare(string1, string2);
```

and then compare `result` to `0`

```
if (result == 0)
  Console.WriteLine("Equal");
else if (result < 0)
  Console.WriteLine($"{string1} comes before {string2}");
else if (result > 0)
  Console.WriteLine($"{string1} comes after {string2}");
```

Note that using these methods `"A"` comes **after** `"a"`, ie both `"A".CompareTo("a")` and `String.Compare("A", "a")` return 1, whereas in both VB.NET and Python `"A" > "a"` returns `False`! This is because these methods do not just compare the codes for the individual characters in strings but use their lexical ordering.

Output Layout

Unless the question asks for a specific output layout, eg *the name of the student on one line and their grade on the next*, or *the name of the student followed by a space followed by their class on the same line*, then ignore horizontal and vertical spaces in the answers.

So, in C# and VB.NET, whether they use `Console.Write` or `Console.Writeline` can be ignored unless specifically required by the question. Similarly, the use of `end=""` in Python can be ignored unless specifically required by the question.

Specific language type issues

General rules for typing

Python's approach complicates things, but, to align the languages as much as possible if there is a missing variable declaration in any of the languages which could make the code correct then mark as if the declaration was present. Assume that any such declaration initialises the variable's value to the 'zero' for that type, ie `0` or `0.0` for numbers, `""` for strings, `False` for Booleans.

So, in VB.NET

```
Do
  password = Console.ReadLine()
  If password <> "OpenSesame" Then
    tries = tries + 1
    If tries = 3 Then
      Console.WriteLine("Too many tries. Account locked")
      OK = False
      Break
    Else
      OK = True
      Exit Do
    End If
  End If
Loop
```

should be marked as if the declarations

```
Dim password As String = ""
Dim tries As Integer = 0
Dim OK As Boolean = False
```

had been made at the start of the program.

Python

Since Python does not type variables but only the values that variables contain, the type of the first value assigned to a variable should be taken as the variable type if there are any marks available for such typing, eg the following (initial) assignment to `numberOfStudents` 'declares' `numberOfStudents` to be of type `int`.

```
numberOfStudents = 47
```

Subsequent assignments to a variable must be of values that are compatible with the first, eg

```
numberOfStudents = "There are no students in the class"
```

means that any marks for the typing of the variable cannot be awarded.

In Python some students may use *type hints* when declaring variables, parameters and function return types. A type hint is a colon followed by the type of the values that a variable or parameter is expected to contain, eg

```
unitsInStock: int = 25
```

```
def Square(n: float) -> float:
                   # -> float gives the function's return type
   return n * n
```

If a student uses a type hint for a variable, parameter or return type the type specified by the hint should be taken as the type of the variable and, if this is correct, any type marks awarded. Any assignments of values incompatible with this type should be considered as errors and marked accordingly.

VB.NET

`Dim n As Integer: n = Console.ReadLine()` will implicitly convert from the string returned by `Console.ReadLine()` to the Integer type given to `n` and so is a valid way of entering an integer (similarly if `n` is defined as `Single`, `Double` or even `Boolean`.)

`Dim n = Console.ReadLine()` will give `n` a type of string (`Console.ReadLine()`'s return type) and so is **not** strictly a valid way of entering an integer, but if the student then goes on to write `n = n + 1` the general rule about typing and missing declarations comes into play.

`Dim n = Cint(Console.ReadLine())` or

`Dim n = Convert.ToInt32(Console.ReadLine())`

will give `n` a type of integer and are both valid ways of entering an integer.

C#

Students may use `var` to define variables which allows them to ignore giving a type to a variable if the variable is initialized to a value. So

```
var i = 25;
```

will declare a variable `i` of type `int`. The variable **must** be given an initial value in the var declaration, so

```
var i;
```

is an error.

```
var n = Console.ReadLine();
```

will give `n` a type of string (the return type of `Console.ReadLine()`). If a student then goes on to use the value of n as if it were an integer, eg, n = n + 1; then the general typing rule should be applied and the var n treated as if it were int n (and the implicit cast rule coming up applied). To get an integer typed `n` would normally require an explicit conversion, ie

```
var n = Convert.ToInt32(Console.ReadLine());
```

If a student has explicitly typed a variable, accept an assignment of `Console.ReadLine()` to that variable as performing an implicit cast to the variable type, eg accept

```
int a = Console.ReadLine();
```

as a valid way of entering an integer.

Students may add superfluous semi-colons after statements and statement parts, eg

if (a = 1) {;

};

These superfluous semi-colons can be ignored (for C# this is still a valid program unless the semi-colon appears after the closing brace of a function definition.)

Python Boolean expressions

Python's approach to Boolean expressions is that, when needed, a value, eg a number, a string, a list, has a 'falsy or a 'truthy' interpretation which can be used as if it were a Boolean value (`False` or `True`). These values can be used as the conditions in indefinite iteration (`while`) statements and selection (`if`) statements.

If an integer is `0` or a float is `0.0` then it can be used as if it were `False` but any other non-zero value can be used as if it were `True`. For example, instead of avoiding a division by zero error like this:

```
if b != 0:
   print("a / b =", a / b)
else:
   print("Cannot divide by zero!")
```

you can write:

```
if b:
   print("a / b =", a / b)
else:
   print("Cannot divide by zero!")
```

For strings the 'falsy' value is the empty string (`""`) and anything else is 'truthy':

```
if s:
   # rather than len(s) > 0, or s != ""
   print("The first character is", s[0])
else:
   print("The string is empty")
# avoids a string index out of range error
```

For lists, the 'falsy' value is the empty list and anything else is 'truthy':

```
if l:
   # rather than len(l) > 0, or l != []
   print("The first element is", l[0])
else:
   print("The list is empty")
# avoids a list index out of range error
```

Relational operators such as `and`, `or`, and `not` can be used with simple values.

eg `if a and b:`

Unlikely/Unusual programming syntax

Teachers should note that students may present unusual looking code that may in fact work perfectly well in their chosen language.

This could be as simple as placing multiple statements on a single line such as

```
if a == 0: print("Hello"); print("Goodbye")
```

or the use of statements like

```
a = b = 1 # Valid in Python and C# but not in VB.NET
          # In VB.NET b has the value 1, but a the integer
          # equivalent of the Boolean value b = 1 which is 0
```

More unusual code might include the use of functional programming techniques or recursion.

To ensure consistency of marking where unusual/unlikely coding statements (not covered by the mark scheme) are seen you should discuss the statement with your colleagues.

Arrays

VB.NET

When defining an array, students using VB.NET may allocate space for one more element than is asked for since the number in an array declaration gives the highest index value, not the count of elements in the array. For example, if a question requires counts to be kept for 1000 throws of a die then a student may declare an array like this:

```
Dim counts(6) As Integer
```

creating an array with elements `counts(0)`, `counts(1)`, ..., `counts(6)`, i.e., with 7 elements. They could then use the result of a random throw (`random.Next(1, 7)`) as an index directly into the element of `counts` to be incremented.

If the student defined the array like this:

```
Dim counts(5) As Integer
```

creating an array with elements `counts(0)`, `counts(1)`, ..., `counts(5)`, i.e., with 6 elements. They would then need to subtract 1 from the result of a random throw to use this as an index but would only create the required number of array elements.

Both methods of declaring an array should be allowed, but if a student then goes on to write code that could access a non-existent element, e.g., in the second case not subtracting 1 from the random throw, then the answer is not completely correct and should be marked accordingly. In the first case **not** accessing the first element (using index values of 1 to 6) or **not** accessing the last element (using index values of 0 to 5) would be perfectly acceptable.

Accessing individual characters or substrings in strings

A string can be treated as an array of characters (with indices running from 0 to one less than the length of the string), so that in VB.NET

```
Dim s As String = "Hello"
Console.Writeln(s(2))
```

outputs a lower-case `l`. Python and C# can do the same thing. Some VB.NET students may have been taught to use the `Left`, `Mid`, and `Right` functions, which are all acceptable, but the first character is at position 1 not 0, so to get `ell` from the string `s` defined above would require `Mid(s, 2, 3)`.

Records

It is possible that students may use unusual structures to implement records, particularly in Python. We have suggested that In C# and VB.NET `struct`s and `Structure`s are used, while in Python `class`es should be used, eg given the car example in the Teaching guide we would expect something like:

VB.NET

```
Structure Car
  Dim make As String
  Dim model As String
  Dim price As Single
  Dim doors As Integer

  Sub New(make As String, model As String,
                        price As Double, doors As Integer)
    Me.make = make
    Me.model = model
    Me.price = price
    Me.doors = doors
  End Sub

End Structure
```

C#

```
struct Car
{
  public string make;
  public string model;
  public double price;
  public int doors;

  public Car(String make, string model,
             double price, int doors)
  {
    this.make = make;
    this.model = model;
    this.price = price;
    this.doors = doors;
   }
}
```

Python

```
class Car:
    def __init__(self, make, model, price, doors):
        self.make = make
        self.model = model
        self.price = price
        self.doors = doors
```

When marking such implementations, the constructor (`New`, `Car` and `_init_`) may be omitted as long as any initialisation is done field by field, eg

VB.NET

```
Structure Car
  Dim make As String
  Dim model As String
  Dim price As Single
  Dim doors As Integer
End Structure

Dim myCar As Car

myCar.make = "Ford"
myCar.model = "Anglia"
myCar.price = 453
myCar.doors = 4
```

C#

```
struct Car
{
  public string make;
  public string model;
  public double price;
  public int doors;
}

Car myCar;

myCar.make = "Ford";
myCar.model = "Anglia";
myCar.price = 453;
myCar.doors = 4;
```

Python

```
class Car:
  pass

myCar = Car()

myCar.make = "Ford"
myCar.model = "Anglia"
myCar.price = 453
myCar.doors = 4
```

This simplified syntax will work but does mean that any instances of `Car` that are created start off without any fields being defined, and it is only the assignment of a value to a field that creates that field.

Thus, the following would be valid

```
yourCar = Car()
yourCar.mark = "Vauxhall      # mark should have been make
yourCar.model = "Astra"
yourCar.price = 8600
yourCar.doors = 5
```

but would result in an error on the following statement

```
print(myCar.make, yourCar.make)
```

Without the declarations of the variables holding a record of type `Car` (`Dim myCar As Car`, `Car myCar;`, and `myCar = Car()`) none of these examples work correctly and any marks available for creating the record itself (but not its initialization to the values given) should not be awarded.

Alternative implementations

It is possible (but unlikely) that C# and VB.NET students could use classes instead of structures, but it is far more likely that students may use alternative methods for structures in Python (the following examples are taken from a discussion on the Computing At School Community - which unfortunately no longer seems to be available on the site):

```
import types

car = types.SimpleNamespace()

car.make = "Ford"
car.model = "Anglia"
car.price = 453
car.doors = 4

print(f"{car.make}, {car.model}, {car.price}, {car.doors}")
```

or

```
from dataclasses import dataclass

@dataclass
class Car:
    make: str = ""       # Initialising variables is not
                         # essential but the type hints are
    model: str = ""
    price: float = 0.0
    doors: int = 0

car = Car()

car.make = "Ford"
car.model = "Galaxy"
car.price = 12000.0
car.no_of_doors = 5

print(car)
```

Examples

C#

```
Int speed;                          // Ignore case of Int
double breaking-distance;           // Ignore - rather than _
string IsWet;
speed = console.readline();         // Ignore case and no cast to
                                    // int
while speed < 10 | spead > 50 {     // Ignore missing ( ), accept |
                                    // or ||, ignore misspelling
                                    // of spead
     speed = console.readline()     // Ignore missing ; and no cast
}
Breaking_distants = speed / 5;
Iswet = console.readline();         // Ignore misspelling of IsWet
if (iswet = "yes")                  // As above and accept = for ==
speed *= 1.5;                       // No indentation but valid
                                    // Compound assignment valid
Console.writeline(breaking_distance)
```

Python

```
Odd = 1
Number = int(input("Enter an integer? "))
While odd <> number       # Ignore case of While, missing : and <>
                          # used instead of !=
   Print(odd)             # Ignore case of Print and odd
   If number < 0          # Ignore case of If and missing :
   Odd -= 2               # Compound assignments valid, ignore
                          # lack of indentation
   Else:
   Odd += 2
```

VB.NET

```
dim distance as integer         ' Ignore keywords/subroutine case
dim passengers as integral      ' Clearly Integer so treat as such
Console.Write("What is the distance? ")
distants = console.readline()             ' Clearly distance
console.write(How many passengers? )      ' Ignore missing quotes
passengers=console.readline()
dim fare = 2 * passengers       ' fare will be typed as an integer
                                ' although 2.0 * passengers would
                                ' give correct type!
far = far + 1.5 * diustance     ' Clearly fare and distance again
                                ' but since fare is an integer the
                                ' calculation result drops any
                                ' fractional part
console.writeline(fare)
```