OVERVIEW OF PROGRAMMING:
Growing Algorithms

Algorithm examples:
Change money (from a dollar)
Change tire (on a car)
Charge people (for admission)
Sort names (alphabetically)
Operate a device (camera)
Play a game (dice craps)
Compute statistics (averages)
Specify a trust (last will)

Algorithms are plans for doing actions on objects.
Plans are also called blueprints or contracts.
Actions are also called operations or processes.
Attributes are also called objects or data.

Algorithms need not always involve computers as shown in the given box of examples.

Programs are algorithms that involve computers.
Actions are operations (commands or instructions) which are often bundled and encased into one big action (called a method: function or routine, procedure).
Attributes (often called data, entities, state, fields) may involve numbers, characters, words, logic, or combinations of these.

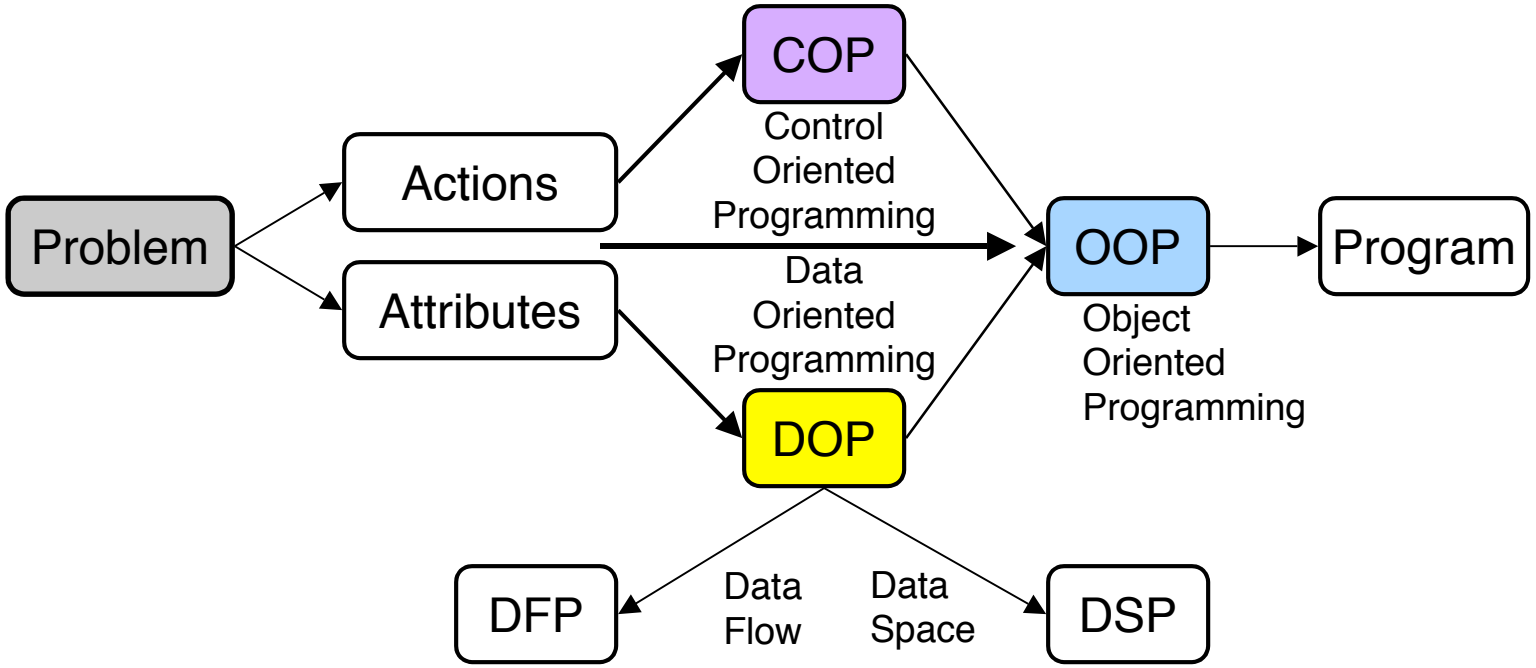
Paradigms are ways of viewing something; there is not just one way to view things!
Here we consider some views of programming; there are more.
COP or control-oriented programming emphasizes actions.
DOP or data-oriented programming emphasizes data, or attributes.
OOP or object oriented programming emphasizes both actions and data.
We will consider all these paradigms, for they are interrelated, but some (COP and OOP) done in much more detail than others.
Other paradigms, such as **SOP**, state oriented programming, are also possible.

Algorithms often begin small and simple, but can grow large and complex. We will use a payroll example to show how algorithms grow. This example gives a general overview; do not get caught up in the details.

Programming: Problem-Solving

In the beginning is a problem (system or situation), with its actions and attributes.
Problems can be solved by breaking them up into small parts, carefully.
(Like cooking an elephant; it must be done in pieces, properly)
There are many ways to do the break up of wholes into parts.
Some ways might be better than others, so it pays to know a few ways.
Some ways, or paradigms are shown in the figure below.

We can begin with emphasis on control (COP), or on data (DOP) or on both (OOP).
In this chapter we begin with algorithms, then COP, briefly DOP and finally OOP.
This provides some perspective which can be gained only by viewing at least two ways.
In the remaining chapters we will "spiral" through repeatedly coming back to others.
This can be done in various orders; there is much flexibility possible.



PROGRAMMING

Programming is a recent activity related to planning and problem-solving, but oriented to computing.
It involves communication with computers using a programming language.
The language is at a high level for communication between humans also.

Programming is a creative human activity; it is partly a science, partly an art, and partly a craft.
It borrows concepts and tools from many areas including mathematics, philosophy, engineering, business, psychology, linguistics and others.

Languages used for programming are many, but there are great similarities between them.
Here we use the Java language, actually a similar language called Jr, a sort of a "pseudo-code" which conveys fundamentals simply without distracting details.
PseudoCode can be converted to Java in a very quick and easy way.
This leads to an efficient way to learn programming and Java.
Jr also goes beyond Java in some ways involving invariants and Design by Contract.

Writing programs is closer to writing poetry than to writing prose.
The words are carefully chosen to reflect the proper meaning,
and they are arranged in particular patterns (with indenting, but usually without rhyming).

Modeling is a significant use of programming; it involves dealing with the “real” world by creating an “abstract” computer world, which has only the relevant "real" parts. A challenge of programming is to bridge the gap between the real and the abstract, but also to realize that the abstract is not the real.
A map is not reality, but it still can be very useful.

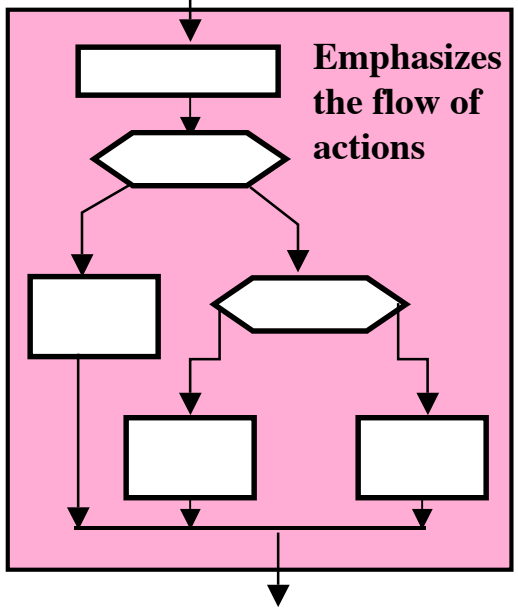
Many ways are possible to do programming, just as there are of building a house.
The extreme view is that of the the child’s fable of 3 pigs building houses using straw, sticks and bricks.
The computer equivalent is building out of bits, instructions, and then methods (routines and functions), and beyond to classes, packages, etc. This analogy extends the n pigs construction to walls, rooms, and prefabricated houses.

Many Ways

There are many ways to view programming! There is no one special, better, sacred way! Sorry.

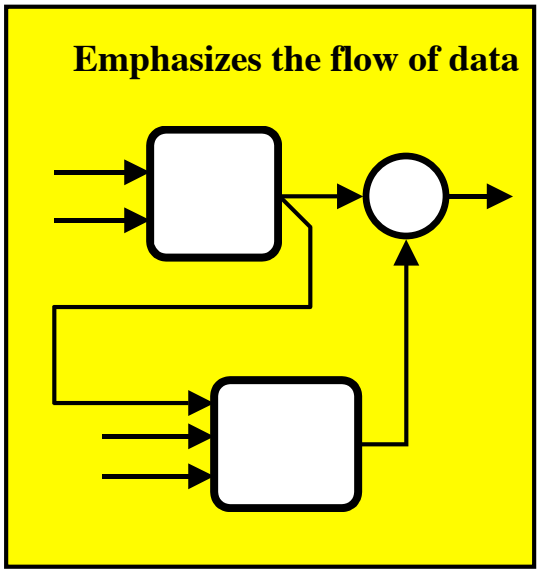
Here we consider three views or Programming Paradigms represented graphically in different ways as shown below. The details within the various boxes have been removed to avoid distraction.

Control Oriented View



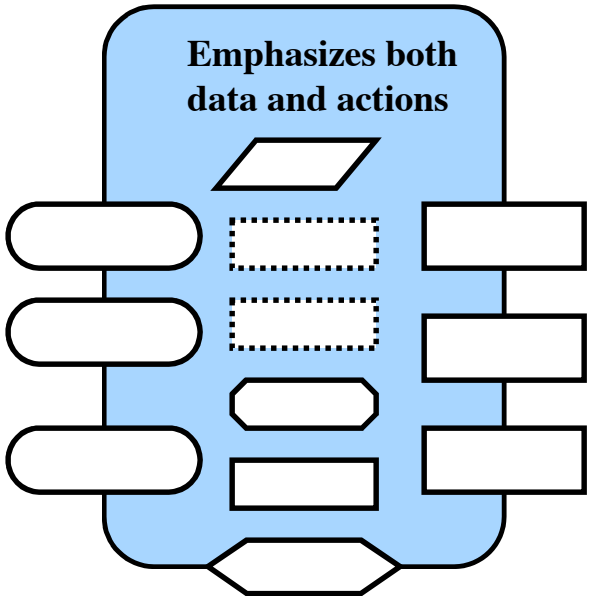
Oriented to actions
- **imperative**
- **procedural**

Data Oriented View



Oriented to data flow of data space, access

Object Oriented View



Oriented to "Class" integrated simulated

(If you just must see the details within the boxes now, you can look ahead to the end of this section)

Growing Pay

Weekly payroll is a simple example which can illustrate many concepts. Suppose that the pay rate is 10 dollars an hour, just for the convenience of easy calculation. So if you worked 25 hours your pay is 250 dollars; if you worked 50 hours your pay is $500. If you doubled your time you get double the pay; there is no extra pay complexity for working overtime. This could be expressed generally by the formula:

```
p = r x h
```

which is very short and confusing for "**x**" could be a quantity or a multiply operation. Usually in computing the asterisk "*" is used as the multiplication operation.

Names (or identifiers) of quantities (variables) in mathematics have traditionally been single letters, such as p, r and h. However, in programming, names could be longer to reflect the meaning. For example, alternatives for the name h could be hours, hoursWorked or even numberOfHoursWorked. Names of boxes (variable identifiers) begin with lower case letters, by convention, and may have upper case letters within them as shown in the previous example

Here we will usually prefer names of some intermediate length, as in the formula:

```
pay = rate * hours
```

However initially and in some drawings we will use shorter names, to save space and time. Then in programs we will use longer names for ease of reading the code.

Overtime Pay

Overtime is indicated by a given time (usually 40 hours) after which the pay rate is increased (often to 1.5 times the rate, called time-and-a-half). For example, in our previous case, if 50 hours are worked then the first 40 hours are paid at the rate of 10.00 dollars per hour (or $400), and the remaining 10 hours (50 - 40) overtime are paid at the rate of $15 per hour (or $150), for a total of $550. This is slightly more than the previous pay of just $10 * 50 = $500.

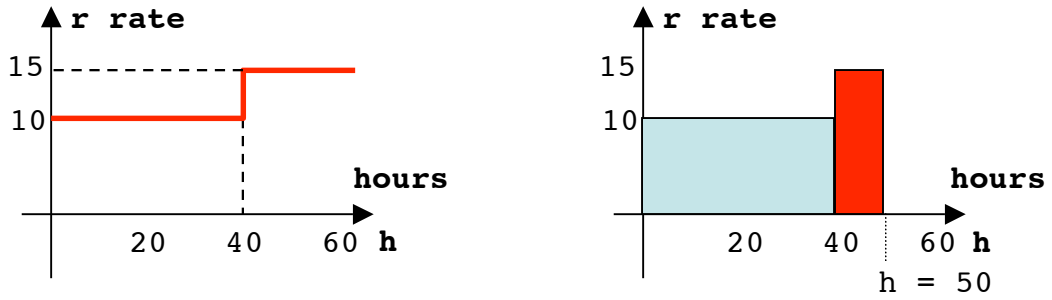
Graphs, or plots, of rate r vs hours h shown below are very useful.
They show that the pay is the area under this curve.

The graph at the right shows the total pay to be the sum of two rectangles:

```
p = 10*40 + 15*(h - 40)
```

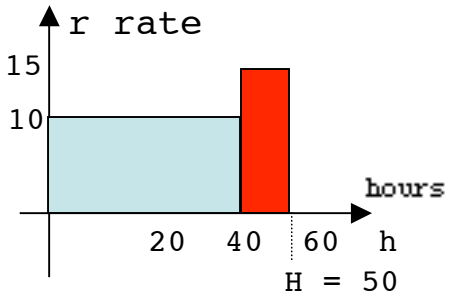
or in general

```
pay = rate*40 + 1.5*rate*(hours - 40)
```

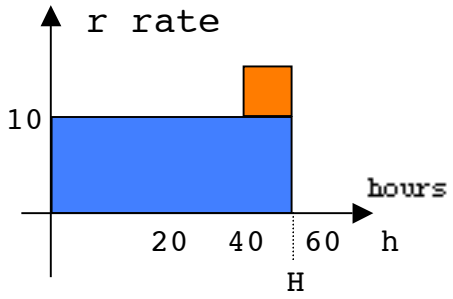


Pay as area:

Pay is the area under the rate vs hours curve can be computed in various ways as shown in the following two graphs.




```
p = 10*40 + 15*(h - 40)
  = 400   + 15*10
  = 550
```




```
p = 10*H  + 5*(h - 40)
  = 10*50 + 5*10
  = 550
```

Many ways are often possible to do anything; some may be better than others. Find another way to do it (using subtraction).

Another View: (optional)

Integrals, in mathematics, represent areas under curves. The pay p under the r vs h curve with h varying from 0 to H is represented by the following mathematical notation:

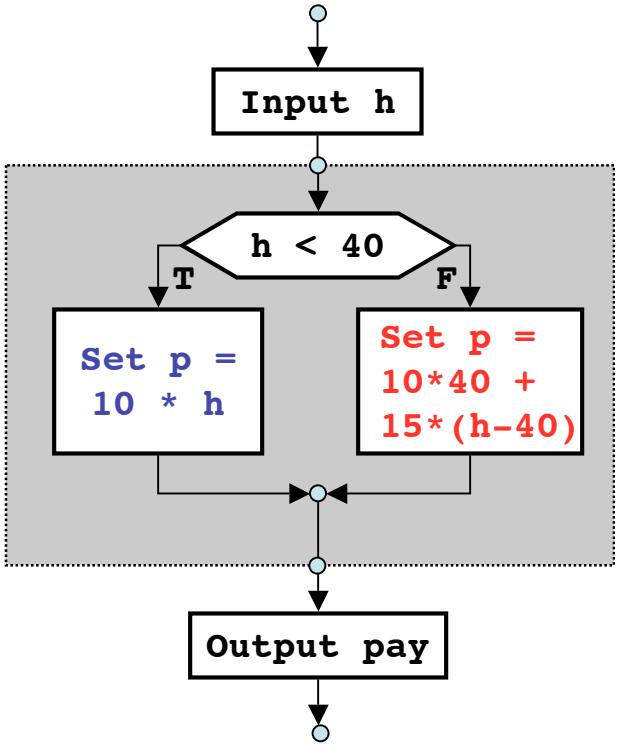
$$p = \int_0^H r \, dh$$

Flows: of Control and Data

Two useful notations for algorithms involve flow;
one describes the flow of control or actions, the other describes the flow of objects or data.

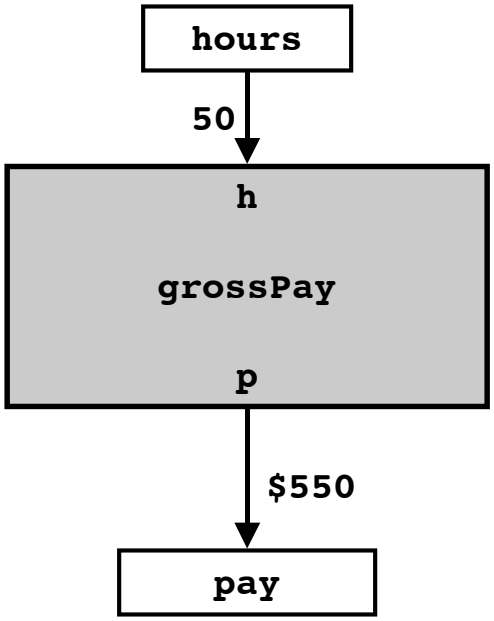
Flow of control, is shown as a flow chart,

where the action moves between circular points depending on the commands or decisions between the points.



Flow of data, is shown as a data flow diagram,

where the data moves into "black" boxes, is manipulated within them, and then flows back out. Here 50 hours flows in and $550 flows out.



Control flow emphasizes "**How**" something is done; Data flow emphasizes "**What**" is being done".
Control flow is the view of a worm; Data flow is that of a butterfly.
Both views are needed.

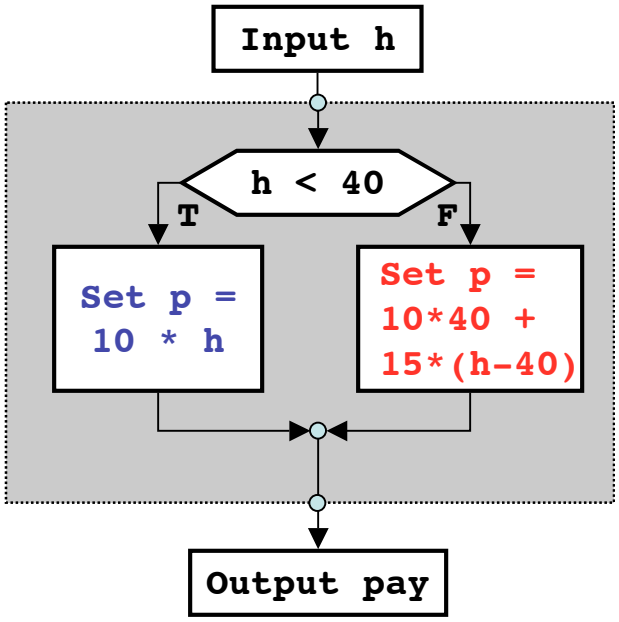
Pseudocode

Pseudo-Code, or pCode, is a very common notation to describe algorithms.
PseudoCode is a readable combination of arithmetic, logic, and natural language.
It looks like actual program code, but is not as precise as required by computers.

Notice the indentation of each of the two actions (the "If" part and the "Else" part).
The "out denting" of the If, Else, and EndIf gives a readable structure to the code.
The ElsePart has been split into three commands involving regular and overtime pay;
it could have been done also as one command even spread over two lines as:

```
Set pay = 10 * 40 +
          15 * (hours - 40)
```

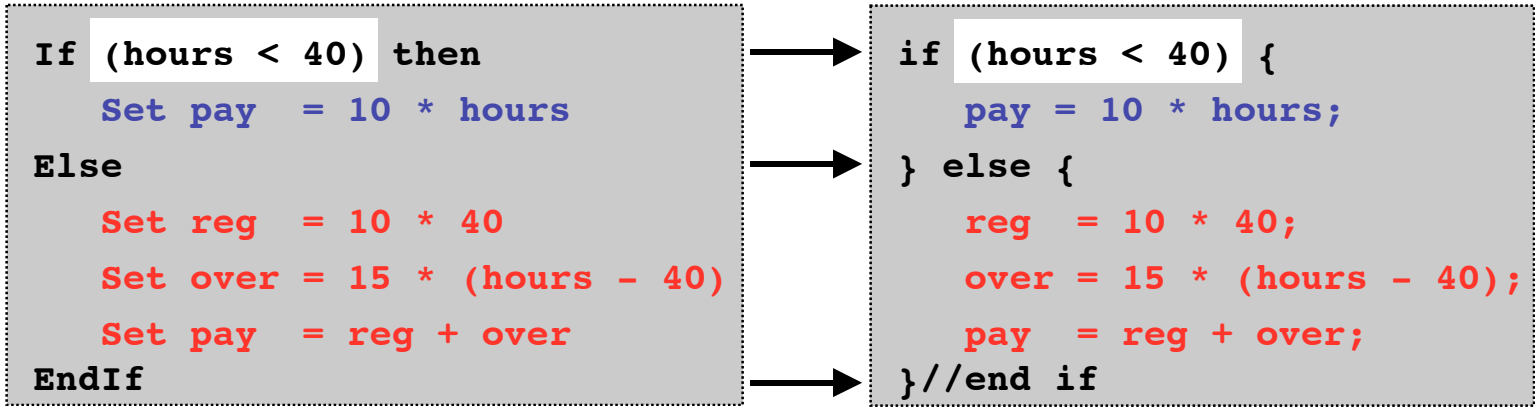
Growing the Pay




```
Input hours
If (hours < 40) then
   Set pay  = 10 * hours
Else
   Set reg  = 10 * 40
   Set over = 15 * (hours - 40)
   Set pay  = reg + over
EndIf
Output pay
```

Actual code: Java

Code in a language such as Java is very similar to pseudoCode.
For example, the Pay algorithm follows in pseudoCode and Java



Notice that each line of pseudoCode converts to exactly one line of Java;
This one-to-one correspondence is useful.

Note also other details in Java.

Case sensitivity means that capital letters differ from lower case ones.
So pay is not the same as PAY, which is not the same as Pay.

Semicolons are used in Java, to terminate all commands.

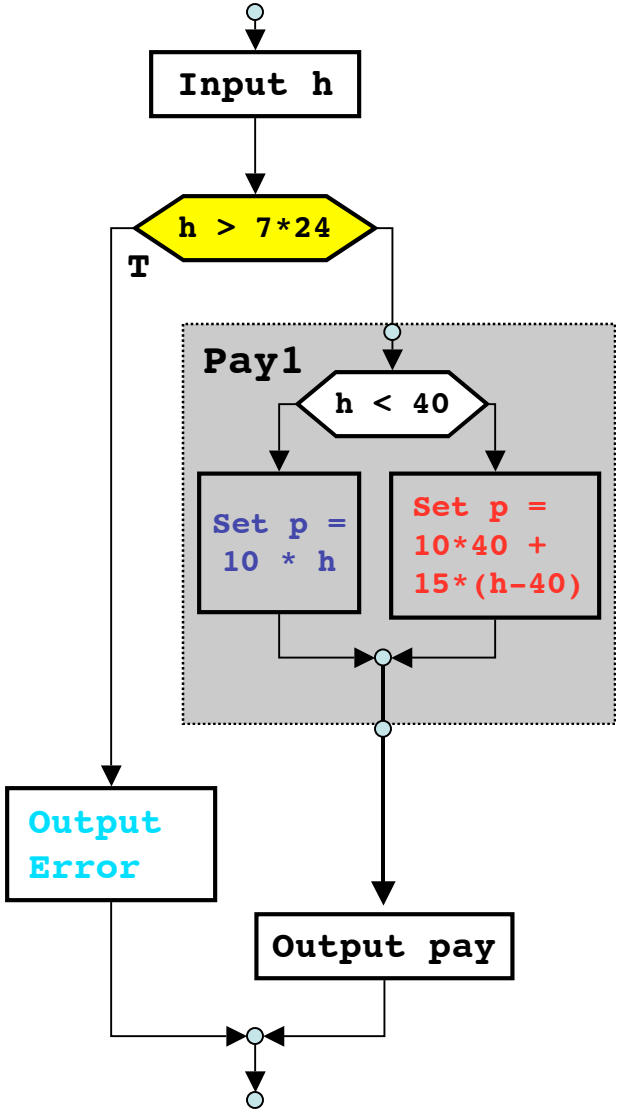
Curly braces '**{**' and '**}**' are used to group a number of commands.
Round braces '**(**' and '**)**' are used to group logical relations or arithmetic terms.

Double slashes '**//**' are used to begin a comment to the end of that line.

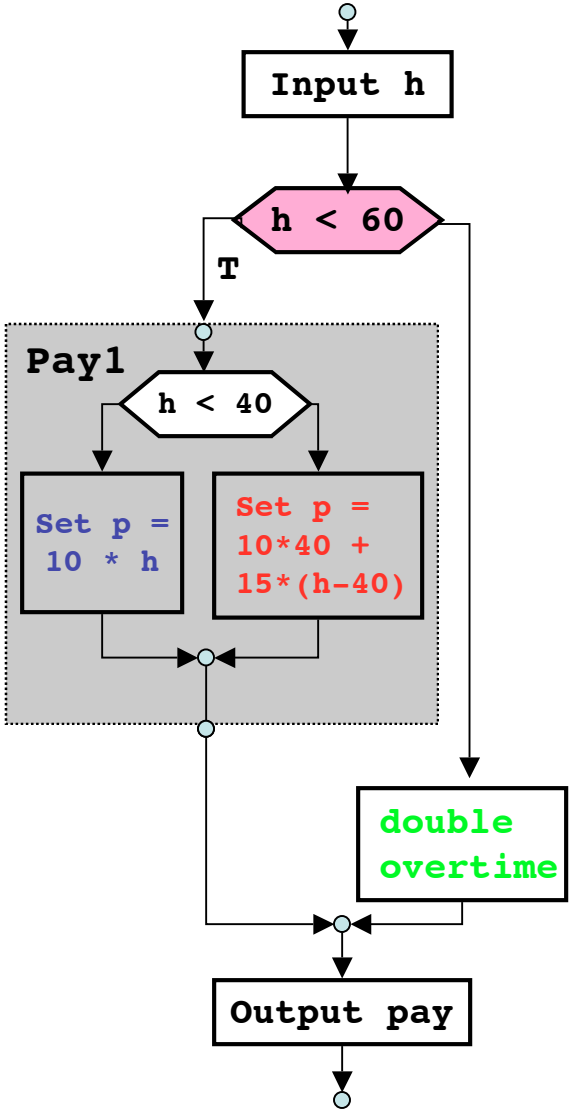
Growing of Pay can be done in many ways, and each will be described later.

The main thing to notice now is that each pay **reUses** the original pay, Pay1.

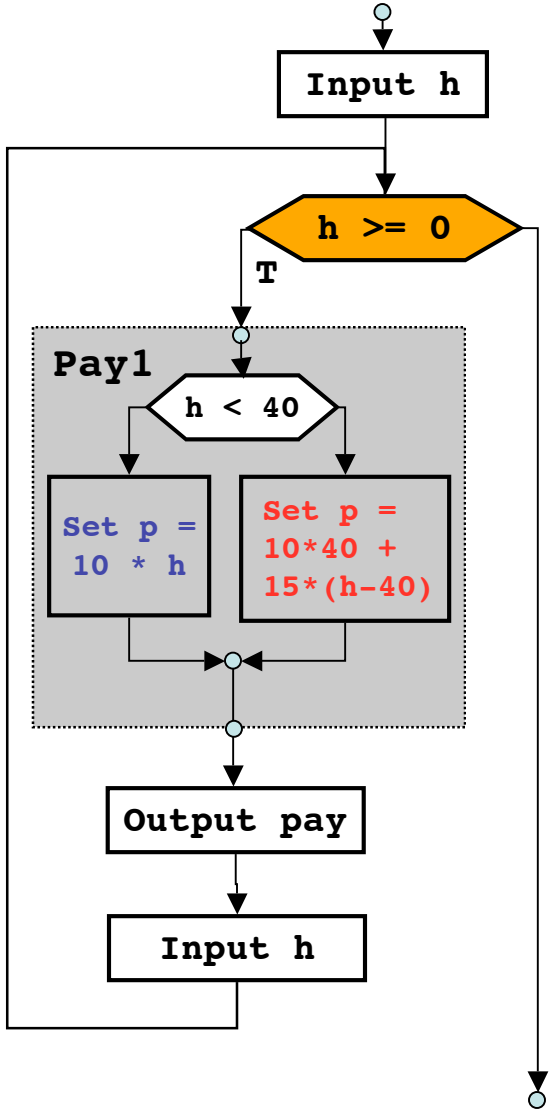
Pay2. FoolProofed



Pay3. Enhanced



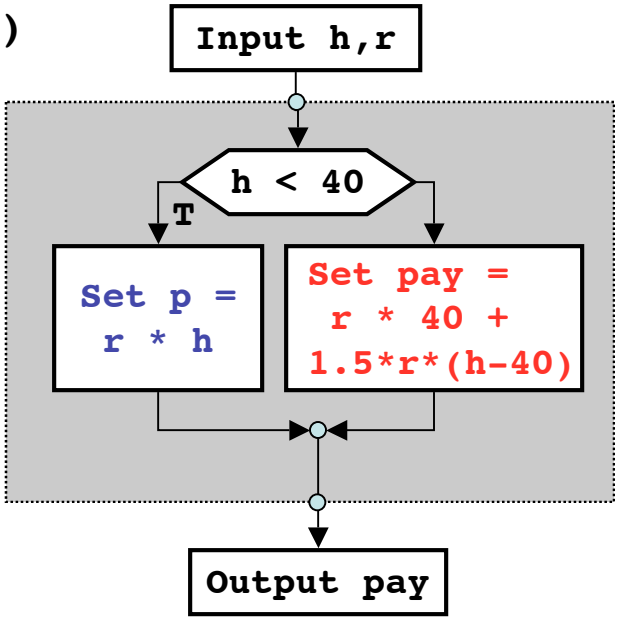
Pay4. Repeated



Next: on first reading go to the page (24) on **Combined algorithms**; then return here later.

Generalized Pay (optional at first)

Constant values, such as the rate should not be "built into" the algorithm. Instead, it is preferred to input the rate as a variable (box) which can have any value. Similarly, the overtime Limit of 40 hours should be in a named box so it can change (perhaps from 40 to 36) at this one place instead of the many (three) other places where it occurs. Such a named constant LIMIT is often written in all caps. The factor 1.5 could also be renamed as a constant, say RATE_FACTOR.



1. Generalizing Pay further (to any rate and constant LIMIT)

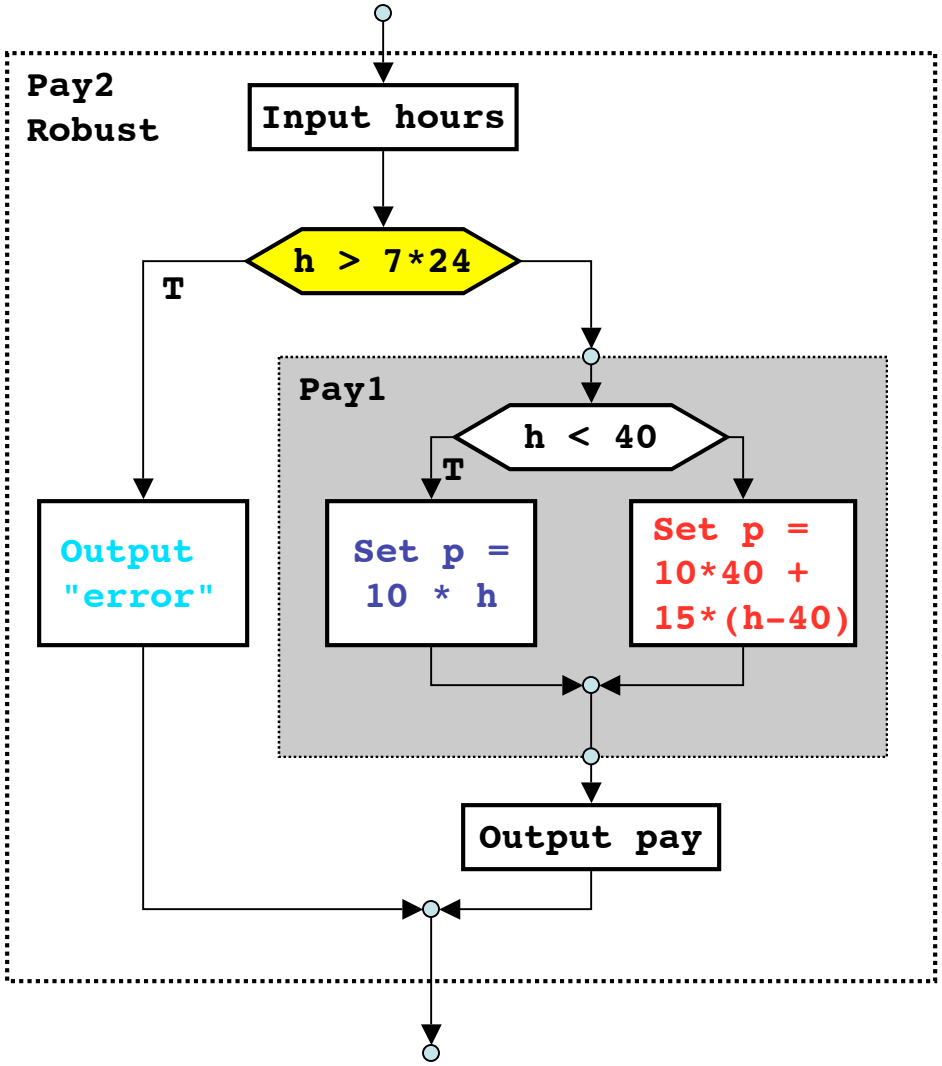
```
Set LIMIT = 40.0
Input hours
Input rate
If (hours < LIMIT) then
   Set grossPay = rate * hours
Else
   Set regPay   = rate * LIMIT
   Set overPay  = 1.5 * rate * (hours - LIMIT)
   Set grossPay = regPay + overPay
EndIf
```

Pay2. Robust Pay: Foolproofed

Foolproofing is the process of making an algorithm more reliable, fail-safe or robust, by anticipating problems such as bad input values. For example, an input to the previous weekly pay program of 200 hours is incorrect, for there are not that many hours in a week. The number of hours in a week is 7*24 which is 168 hours. If more hours are input an error message could be output.

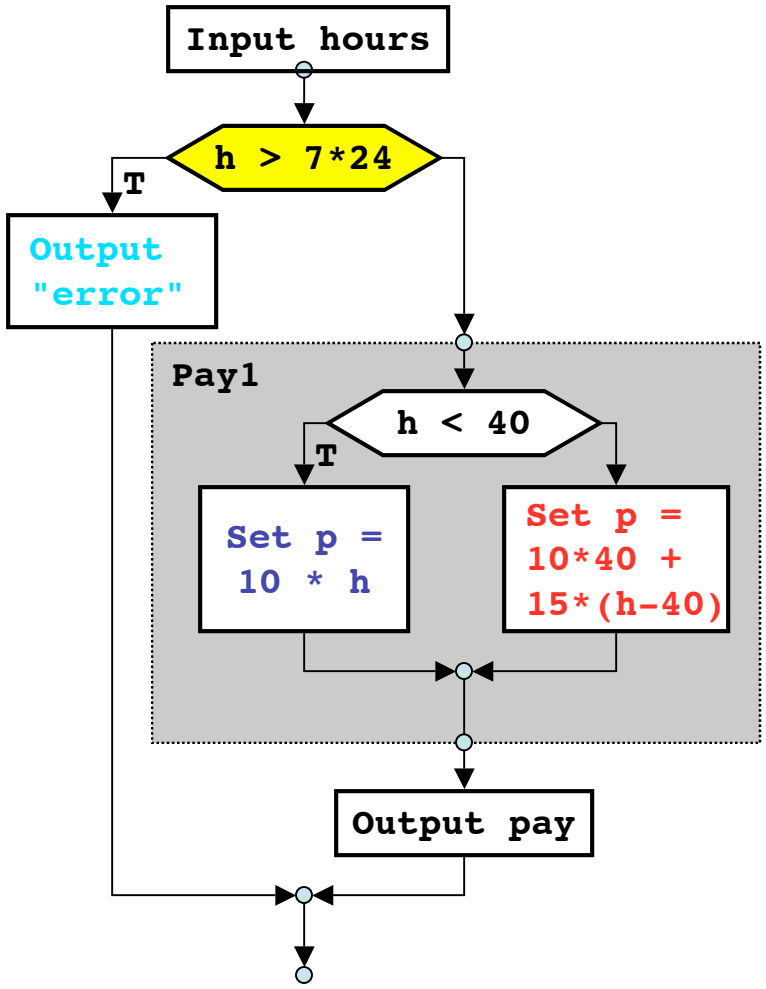
The given figure shows how the previous Pay1 algorithm (not the generalized one) is extended to be more foolproofed or robust. Note that the number 168 is not shown in the algorithm for it hides the two components of the product 7*24. Besides, humans should not need to do such arithmetic when computers can do it faster and better.

Notice that the original Pay1 is reused; it has not been modified.



Testing Robust Pay2

Testing an algorithm with a "hand" check may be very useful. For example, Pay2 may be traced for various test values as indicated on the table at the right.



Test Table

hours	output	range
0	0	h == 0
20	200	0 < h < 40
40	400	h == 40
50	550	40 < h < 168
168	1920	h is max
200	Error	h > 168
-50	-500?	h < 0 error!

This test reveals a problem with negative inputs; the algorithm needs to be further modified.

Pay3. Enhanced Pay: to double overtime rate

Enhancing algorithms to cover more cases is very common. For example, if the hours worked are more than 60 then the original pay rate could be doubled. This new algorithm extends the original Pay1, which is reused within it. The amount of double overtime pay is given by the formula:

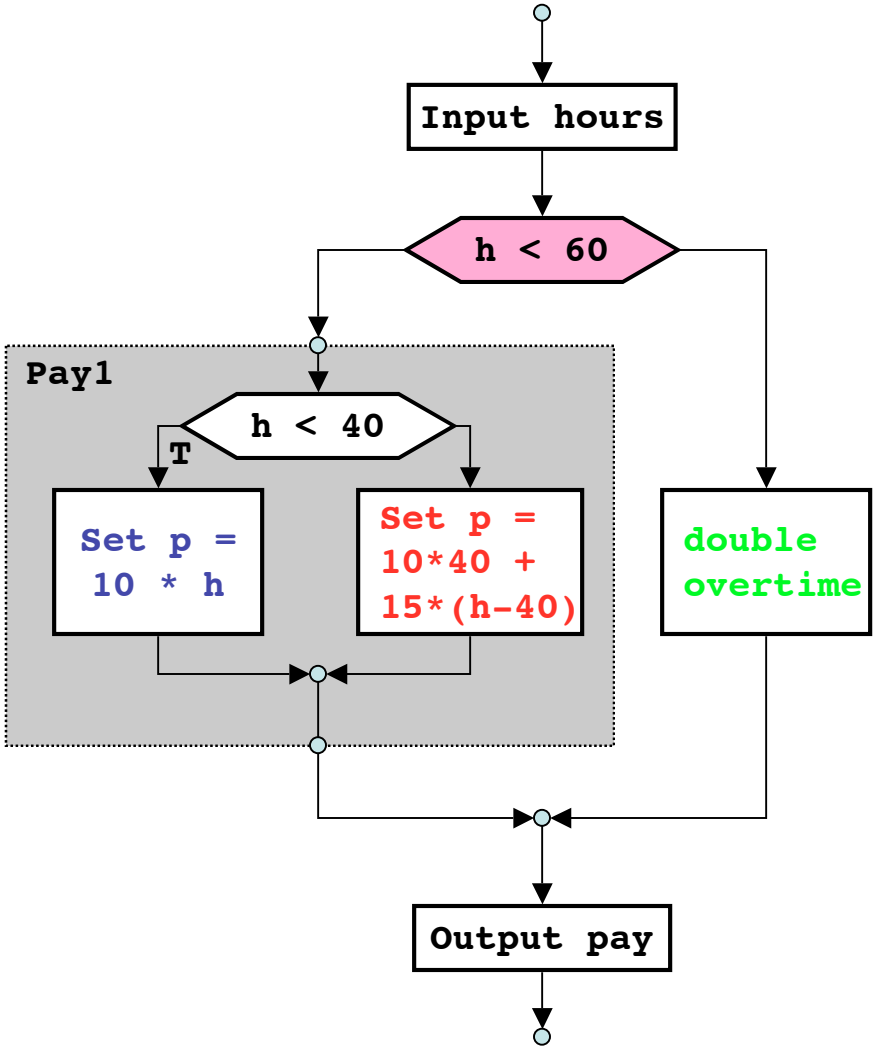
```
pay = 700 + 20*(hours - 60)
```

This formula can be derived from the area under the new rate vs hours graph or plot. Try it, before looking at the solution on the next page.

As an example, if the input is 100 hours, the output is:

```
pay = 700 + 20*(100 - 60)
    = $1500
```

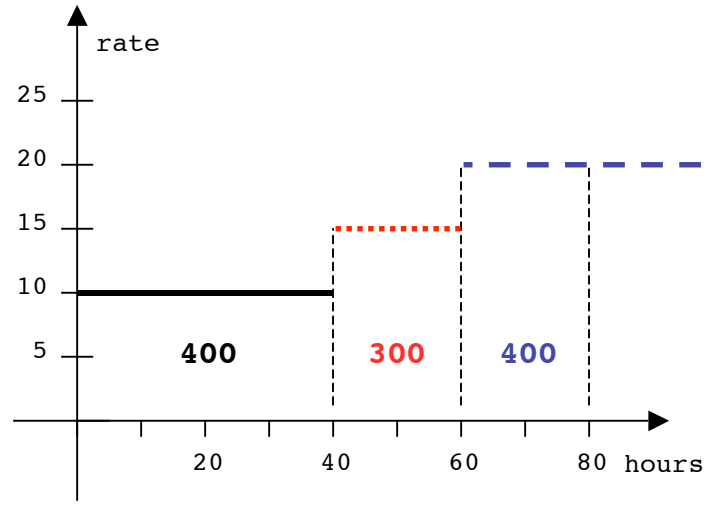
Note that the original pay example, with 25 hours yielded $250; when the hours are quadrupled to 100, the pay more than quadruples (to 1500 rather than only 1000).



GRAPHIC PAY (optional at first)

The extended gross pay algorithm can be viewed graphically as shown. Rate vs hours is a graph that has the shape of an increasing staircase; the rate of pay is more as hours increase. The gross pay is the area under this curve. For example, the pay for h = 80 hours consists of the three areas:

```
pay = 10*40 + 15*(60 - 40) + 20*(h - 60)
    =  400  + 300 + 20*(h - 60)
    =  700  + 20*(h - 60) in general
    =  $1100 (for h = 80) in particular
```



Pay vs hours is the second graph; it begins with a small slope at first, then increases in slope as the hours increase at two points 40 and 60 hours.

Only four points are needed to plot these three lines, as indicated by the small circles. This is known as a "piece-wise linear" graph.

The **first piece** (for hours from 0 to 40, shown solid) is described by the formula:

pay = 20 * rate

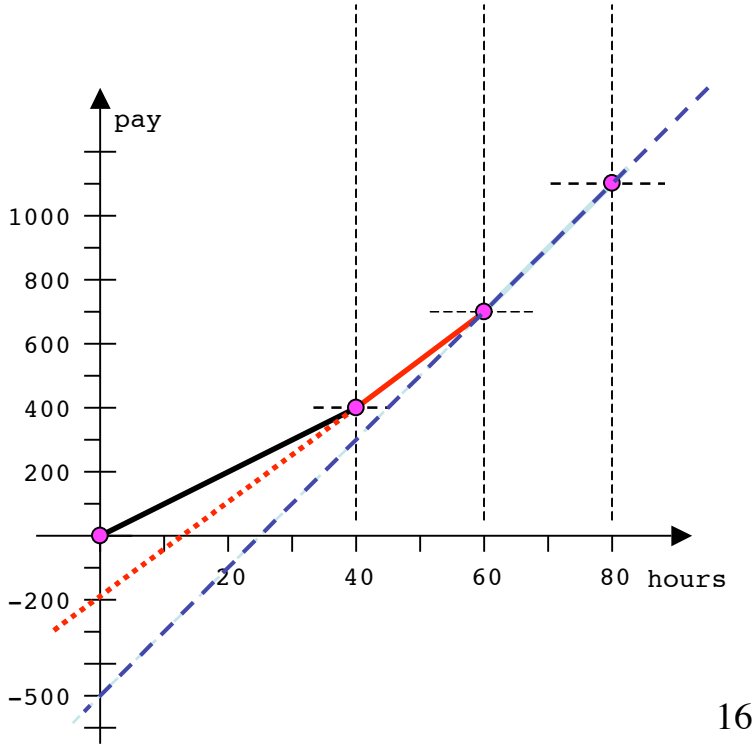
The **second piece** (for hours from 40 to 60, shown dotted) is described by the formula:

pay = 15 * rate - 200

The **third piece** (for hours greater than 60, shown dashed) is described by the formula:

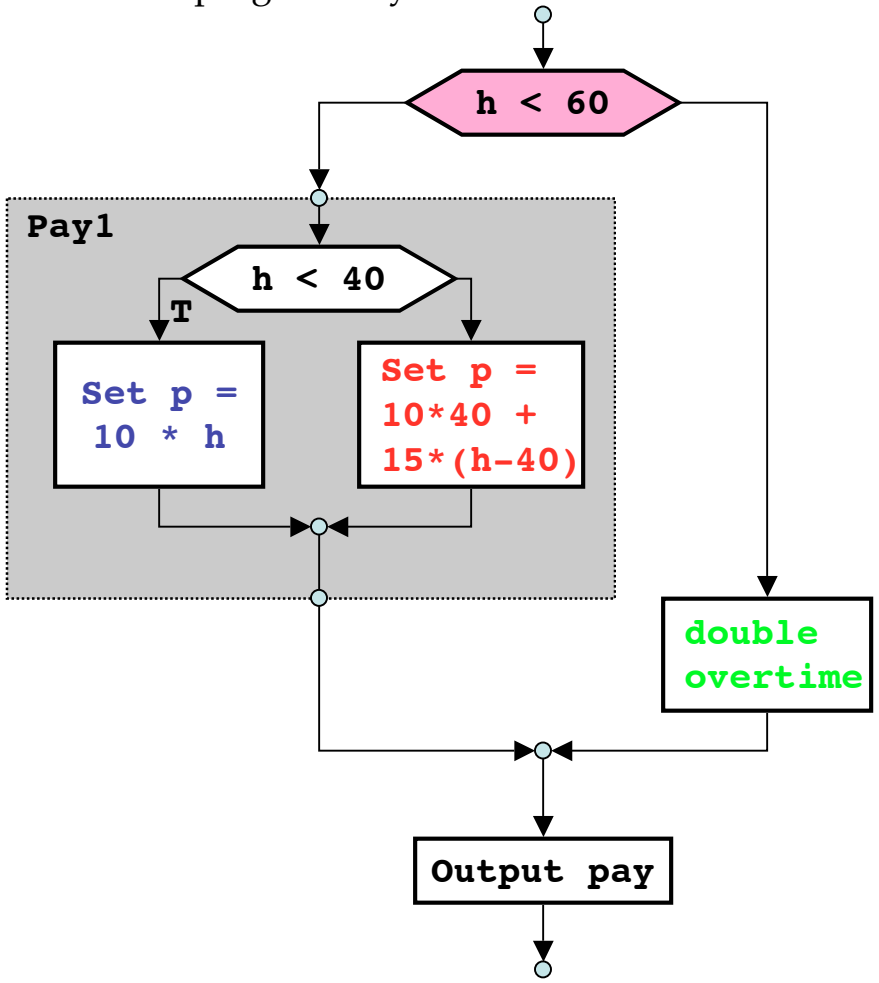
pay = 20 * rate - 500

Note that the constants in these equations are the points of intersection of the pay axis and note also that the slope is the rate.



Code of Enhanced Pay3: double overtime

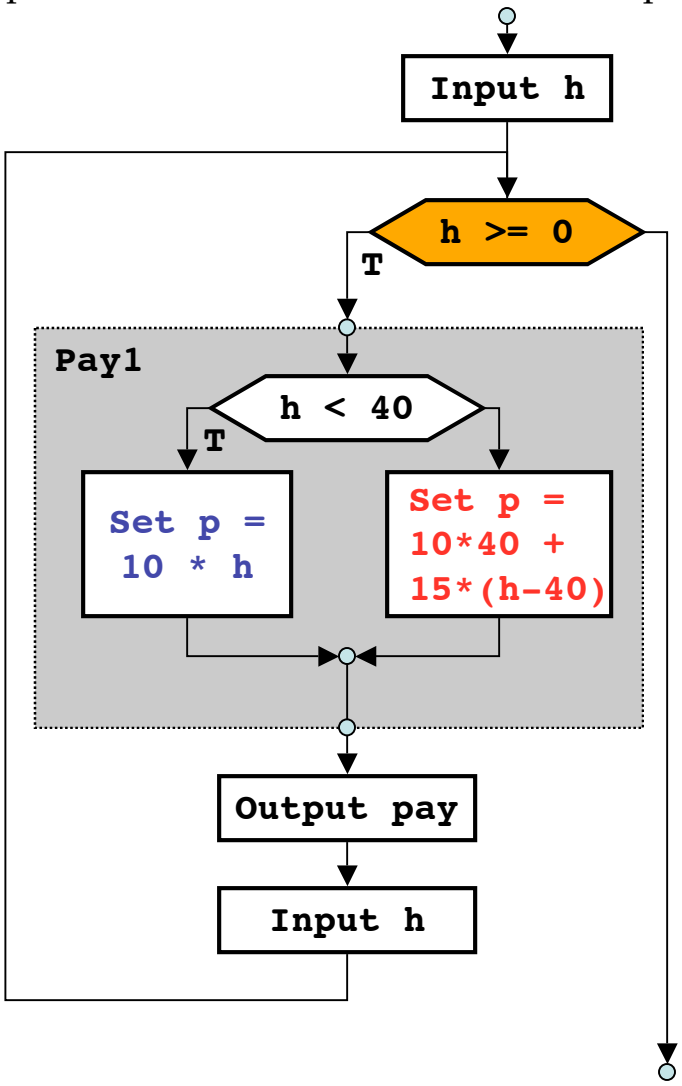
PseudoCode of the enhanced pay algorithm is shown to the right of the flow chart.
Notice that the original Pay1 is not modified; it becomes part of the new program.
The program Pay1 has been "reUsed" rather than modified. ReUse is the way to grow.




```
If (hours < 60) then
   If (hours < 40) then
      Set pay = 10 * hours
   Else
      Set reg  = 10 * 40
      Set over = 15*(hours - 40)
      Set pay  = reg + over
   EndIf
Else
   Set pay = 700 + 20*(hours - 60)
EndIf

Output pay
```

Pay4. Repeated Pay

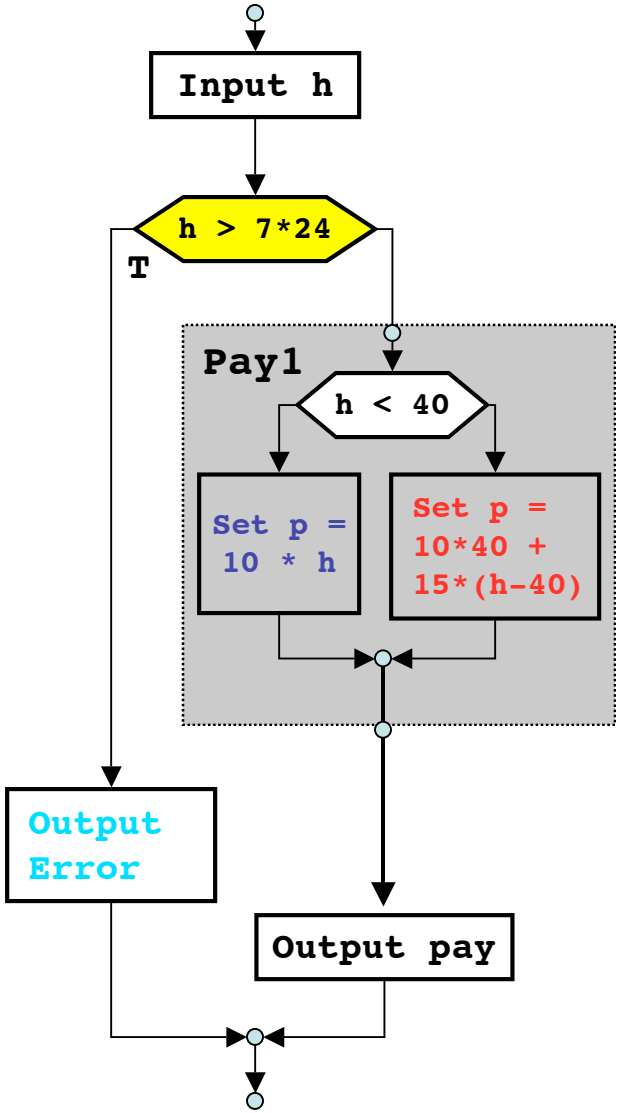
Repeating things is very common in computing. For example, the previous pay program did a computation for only one employee; it can be repeated for many employees by "nesting" the original Pay1 in a loop, which keeps repeating until a negative value is input. The adjacent pseudoCode shows how the While loop does the repetition. Notice the indentation.



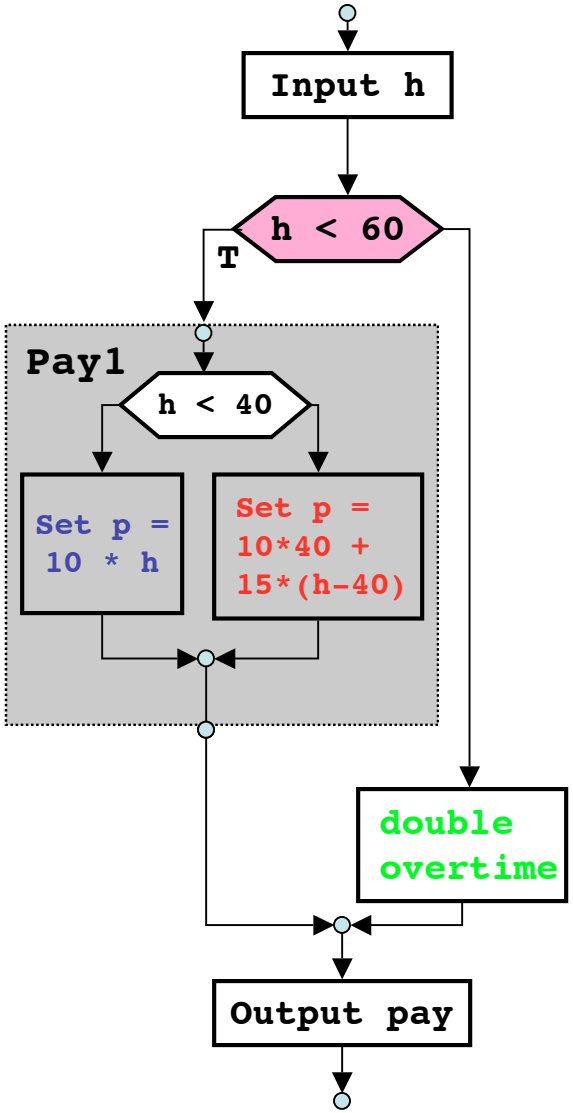

```
Input hours

While (hours >= 0)


   If (hours < 40) then
      Set pay = 10 * hours
   Else
      Set reg  = 10*40
      Set over = 15*(hours - 40)
      Set pay  = reg + over
   EndIf


   Output pay

   Input hours

EndWhile
```

ReUse of Pay (in gray) is clearer by the repeated figures below.

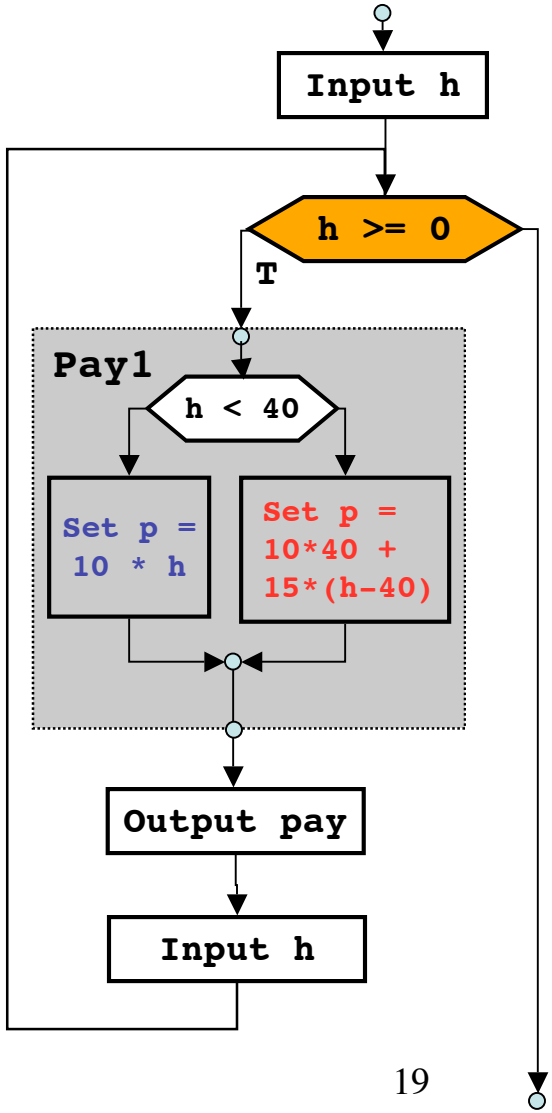
Pay2. Robust



Pay3. Enhanced



Pay4. Repeated



Growing by ReUse

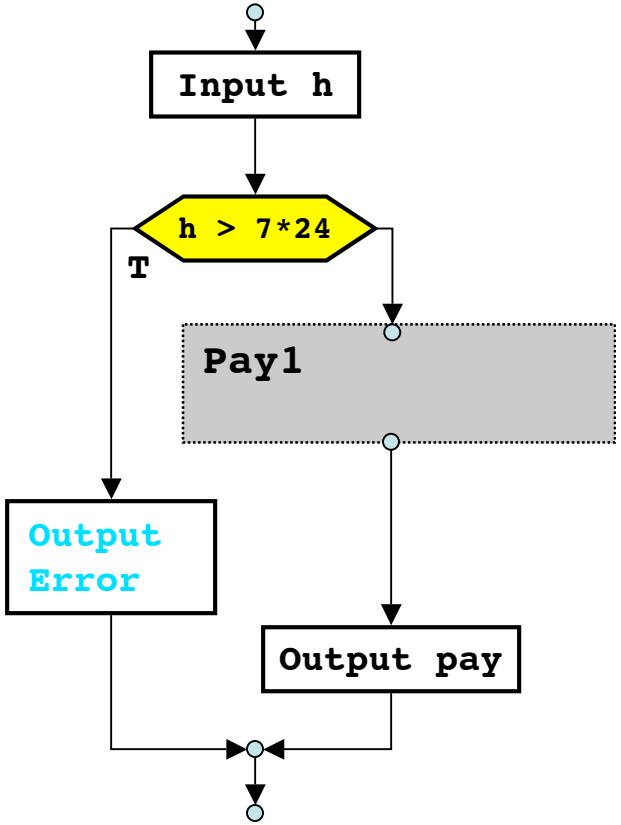
ReUse is the process of growing by making use of previously constructed parts (components).

In the Pay program it is important to realize that Pay1 is a component of all the three versions. So Pay1 can be viewed as an abstract black box.

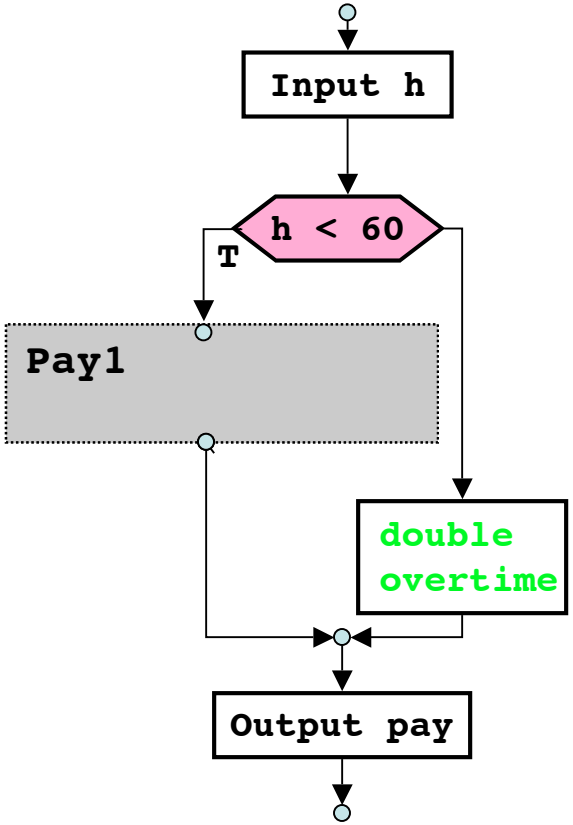
ReUse is also a common process often used in manufacture of cars, computers, cameras, etc.

When there is a shorter, cheaper, faster or better way to make the part then that part may be replaced simply without further changes to the rest of the system.

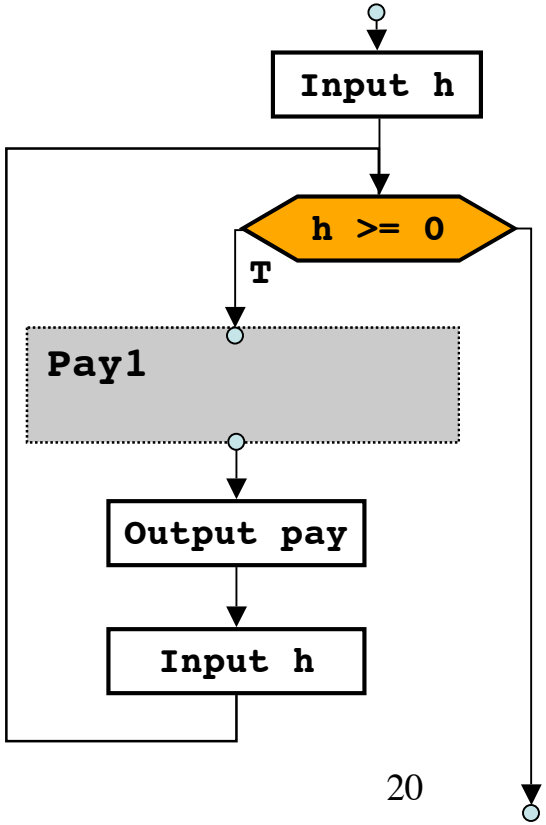
Pay2. Robust



Pay3. Enhanced



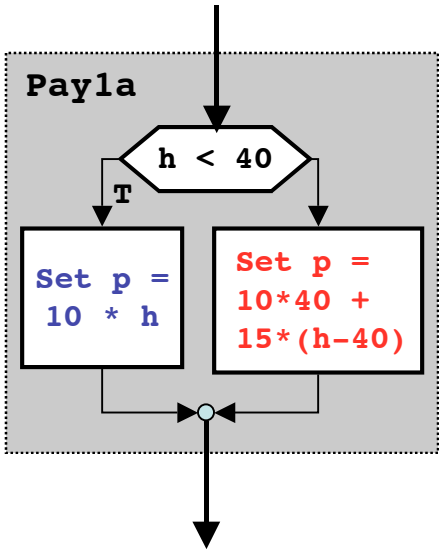
Pay4. Repeated



Equivalent Algorithms (optional at first)

Many ways are often possible to do anything. For example, Pay1 can be done in the following ways labeled Pay1a and Pay1b. These two ways differ in structure or form, but are equivalent in behavior or function.

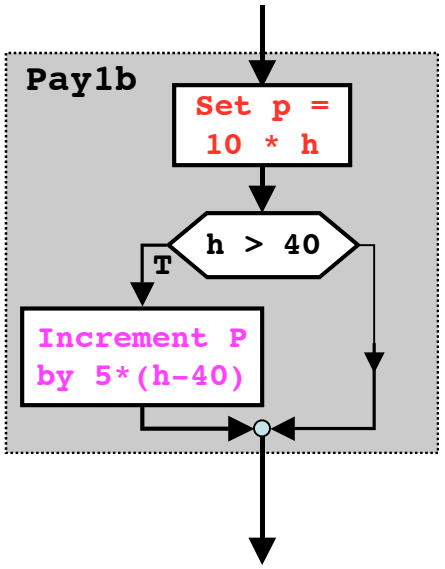
Proof of the equivalence can be done by testing the behavior for all different cases. In this system there are only three different cases, (h < 40), (h == 40) and (h > 40). The test table between the diagrams show the results of a trace of each of these cases for some typical valueof h in each range.



Test Table

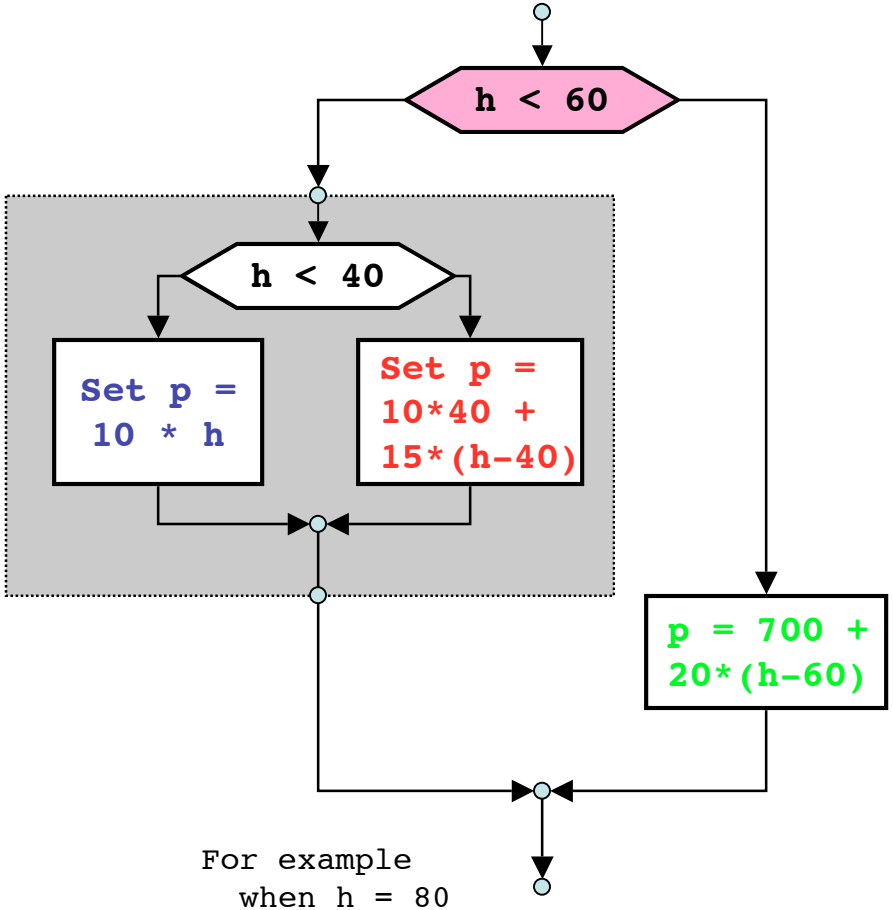
case hours	hours	h	outcomes pay1a	pay1b
(h < 40)	20	200	=	200
(h = 40)	40	400	=	400
(h > 40)	50	550	=	550

Equivalent



Equivalent Enhanced Pay: Optional

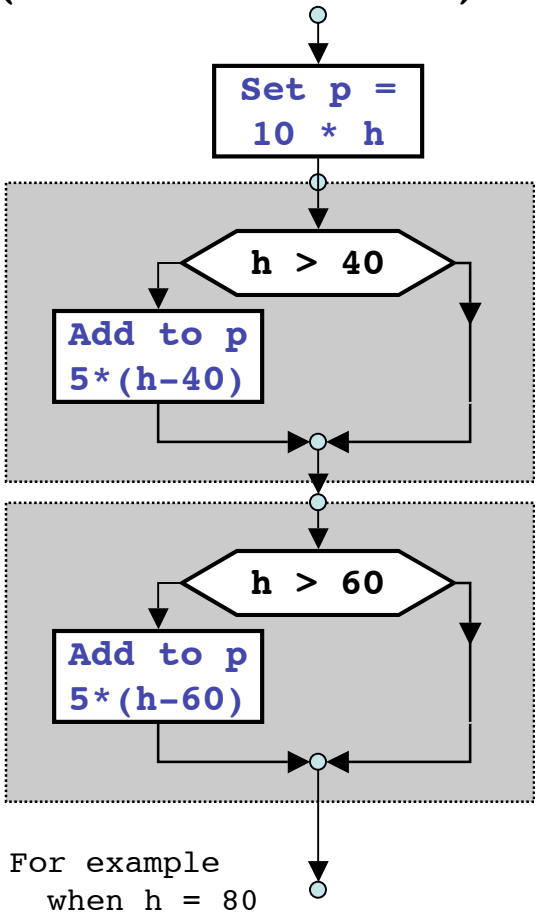
(Nest of Choices)



For example
when h = 80

```
p = 700 + 20*(h-60)
  = 700 + 20*(80-60)
  = 700 + 20*20
  = 1100
```

(Series of Choices)



For example
when h = 80

```
p = 10*h + 5*(h-40) + 5*(h-60)
  = 10*80+ 5*(80-40)+ 5*(80-60)
  = 800  +  200     +  100
  = 1100
```

Two Views of Enhanced Pay (optional)
(as vertical and horizontal bars)

The two previous equivalent ways to view the pay correspond to two ways to view the area under the curve.

The first algorithm, with nested choices, corresponds to the three vertical bars, having the total sum of:

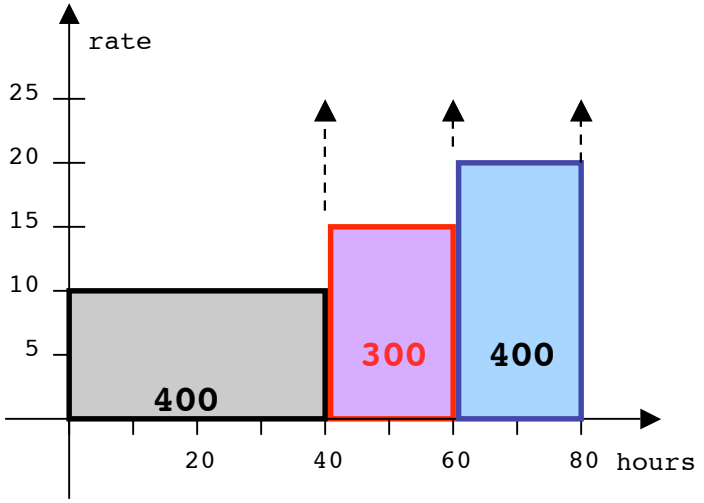
```
pay = 400 + 300 + 400 = $1100
```

and more generally, for h > 60

```
pay = 40*10 + 15*20 + 20*(h - 60)
```

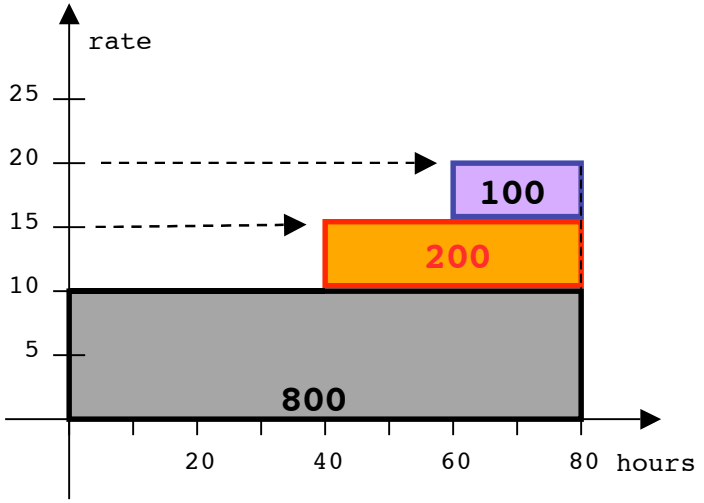
and even more generally as:

```
pay = 40*r + 1.5*r*(60-40) +
              2.0*r*(h - 60)
```



The second algorithm, with a series of choices, corresponds to the three horizontal bars, having the same total sum (for h > 60) of:

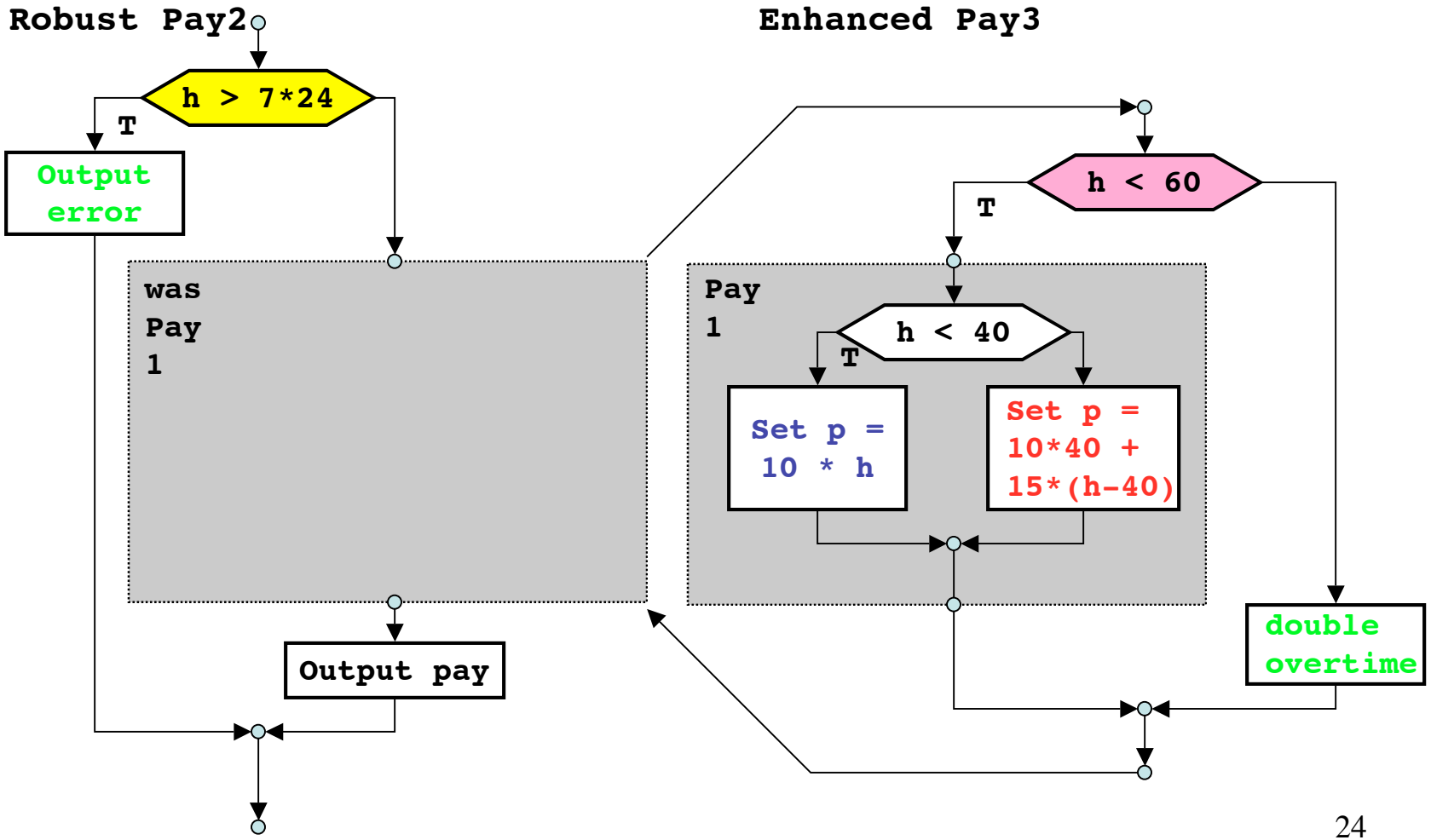
```
pay = h*r + 0.5*r*(h - 40) +
            0.5*r*(h - 60)
pay = 800 + 200 + 100 = $1100
```



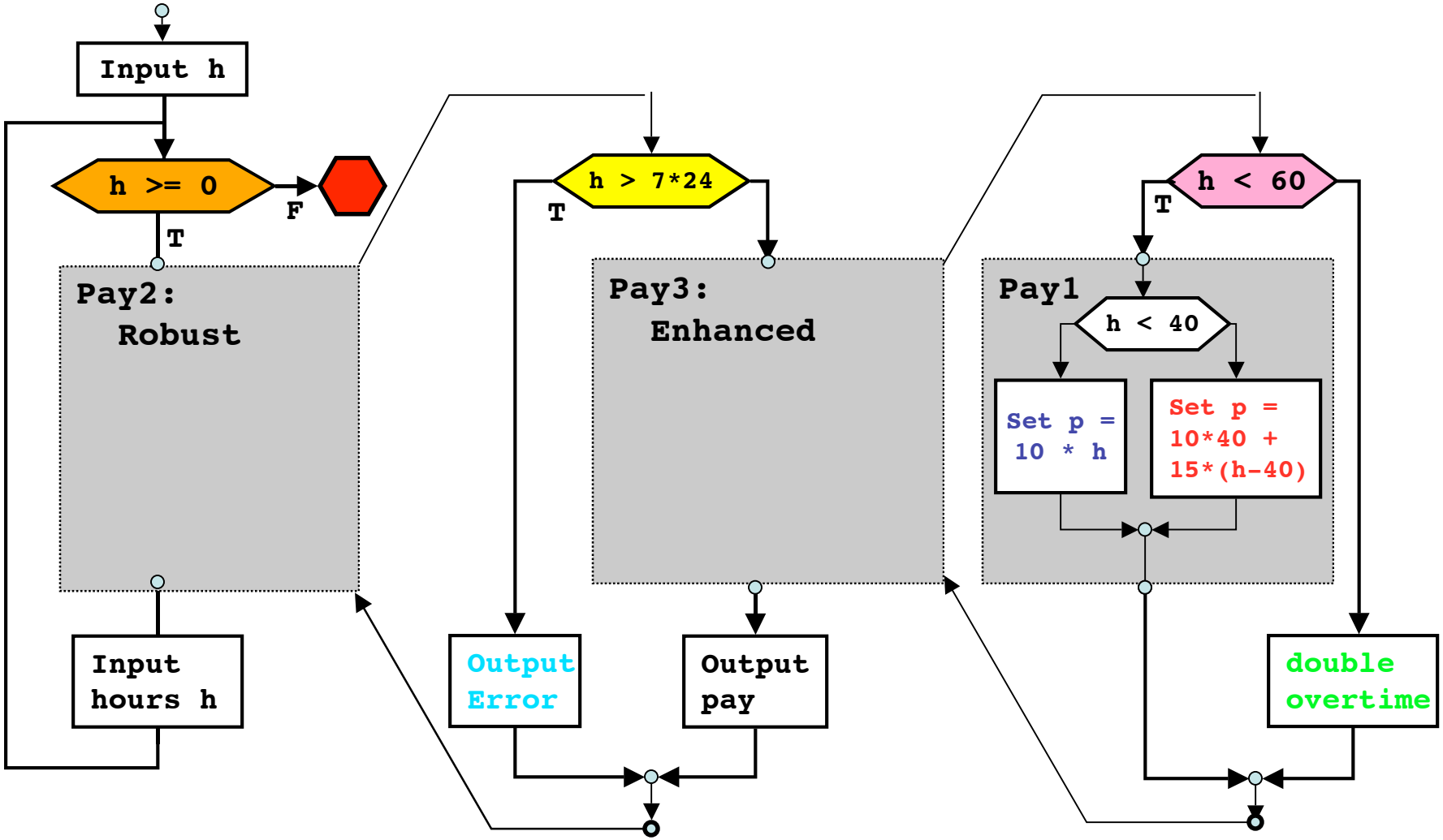
This shows again that there are often many ways to do anything; some ways may be better than others.

Combined Algorithms: Foolproofed (robust) and Enhanced Pay

Growing algorithms is often done by combining algorithms. For example, the previous robust Pay2 and Enhanced Pay3 both reused the original Pay1. But the robust Pay2 can reUse Pay3 (which in turn uses Pay1) as shown in the following figure. This figure can also be used to include the generalized pay and the repeated pay. Complexity grows quickly. Draw it.



BigPay: Including repeated, robust, and enhanced; in stages



Data Types:

The data values used in the pay example are numbers;
but numbers can be of two types: integer and real.

Integers (also called whole numbers) come from counting.
Examples are: `7, 11, 0, 40, 365, -7, 19683.`

Real numbers (also called floating point or double)
come from measuring. They involve decimal points.
Examples are: `1.5, 3.14159, 40.0, 0.2, -12.345, 365.54.`

Computers represent such numbers differently so we must make a distinction between them.
In other words, the integer 7 differs from the real number 7.0.

Consider, for example, the pay formula:

```
pay = 40*rate + 3/2*rate*(hours - 40)
```

For a rate of 11 dollars per hour, and 50 hours

```
pay = 40*11 + 3/2*11*(50 - 40)
    = 440   + 3/2*11*10
```

If the division by 2 were changed as shown below, in three different positions,
doing integer arithmetic (from left to right, with "chopping" each integer)
there are three different answers:

```
pay = 440 + 3/2*11*10 = 440 + (3/2)*11*10 = 440 + 1*11*10 = $550
pay = 440 + 3*11/2*10 = 440 + (33/2)*10   = 440 + 16*10   = $600
pay = 440 + 3*11*10/2 = 440 + (330/2)     = 440 + 165     = $605
```

Which is correct: the first one, that the employer likes; the last one that the employee likes;
or a compromise somewhere in the middle?
Beware; be aware.

Boxes, types, units, constants

Quantities involved here could be viewed as boxes with numeric contents.
Time would have a content which may vary each week.
The pay rate would have a content or value which is fixed at some constant value (such as $10 dollars per hour, abbreviated as dph).

Units or sizes also describe these quantities. For example the time could be measured as hours, and the rate given as 10 dollars per hour. Also, the time could be counted in minutes, and the rate computed in cents per minute (cpm) as:

```
rate = 10 dollars per hour

     = 10 dollars/hour * 1 hour/60 min * 100 cents/1 dollar

     = 16 2/3 cents/minute

     = 16.6666666666 .. cents per minute
```

Notice canceling of the units

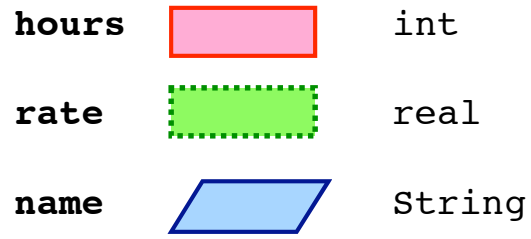
So the rate of 10 dollars per hour is equivalent to 16.6666 (unending) cents per minute.
In this case there would need to be an agreement (contract) which makes clear whether the resulting pay is rounded up to the next higher cent (17), chopped down to the next lower one (16), or rounded off to the closest tenth (16.7).

Data types can be shown as boxes of various shapes (or even colors):

Integers, such as the hours worked (or age or seniority) are shown in a rectangle,

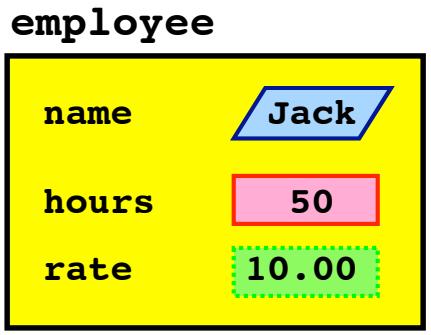
Reals, or doubles, such as the rate (payRate or taxRate) are in a dotted rectangle.

Strings, such as a name (or address or date) are in a parallelogram.



It is important to realize that (to a computer):
the integer `7` is not the same as the real value `7.0` which is not the same as the String `"7"`.

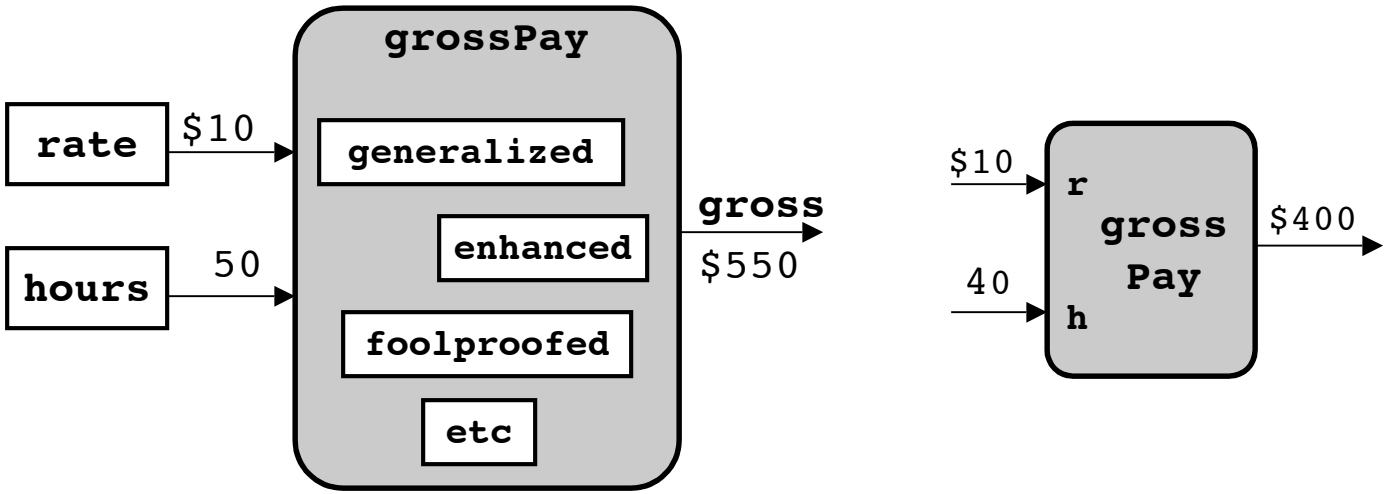
`Encasing data` of various parts into one whole is extremely useful.
For example,three parts may be viewed as one larger whole (an employee).



Encasing: Encapsulating control

Encasing, or encapsulation, is the process of simplifying a system by packaging many details into one box, and viewing that "black box" as one new component.

For example, all the details of computing the previous pay (called gross pay) can be put into a shaded box as shown, with two inputs (rate and hours) and one output (gross pay). The emphasis is not the internal **"how"** it does it, but the external **"what"** it does.



Wholes and Parts

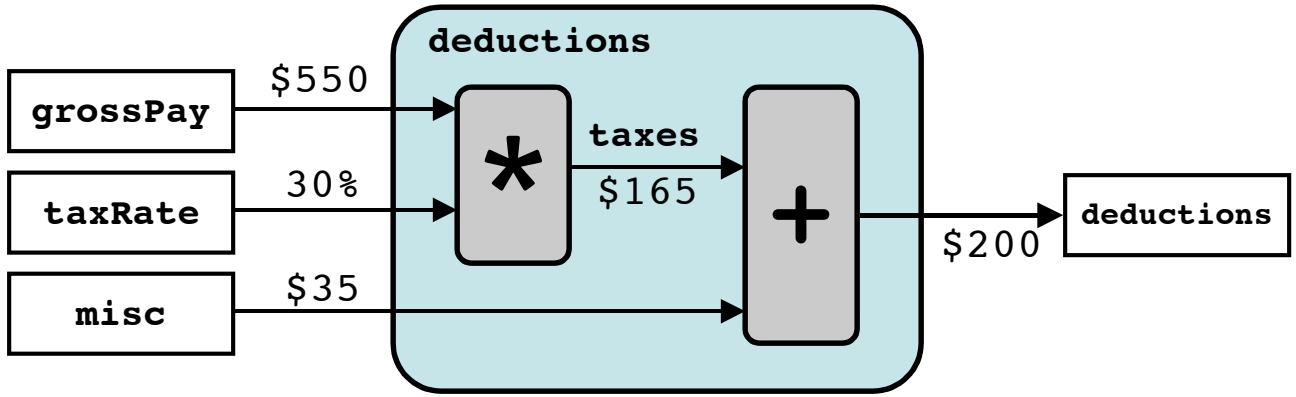
The generalized, enhanced and foolproofed parts make up a gross pay whole. Then this gross pay whole can be a part of a larger whole, the net pay.

DOP: Data Flow Programming

DFP or data flow programming emphasizes the flow of data between "boxes". For example, in a pay program taxes could be computed, as a given percent of the gross pay.

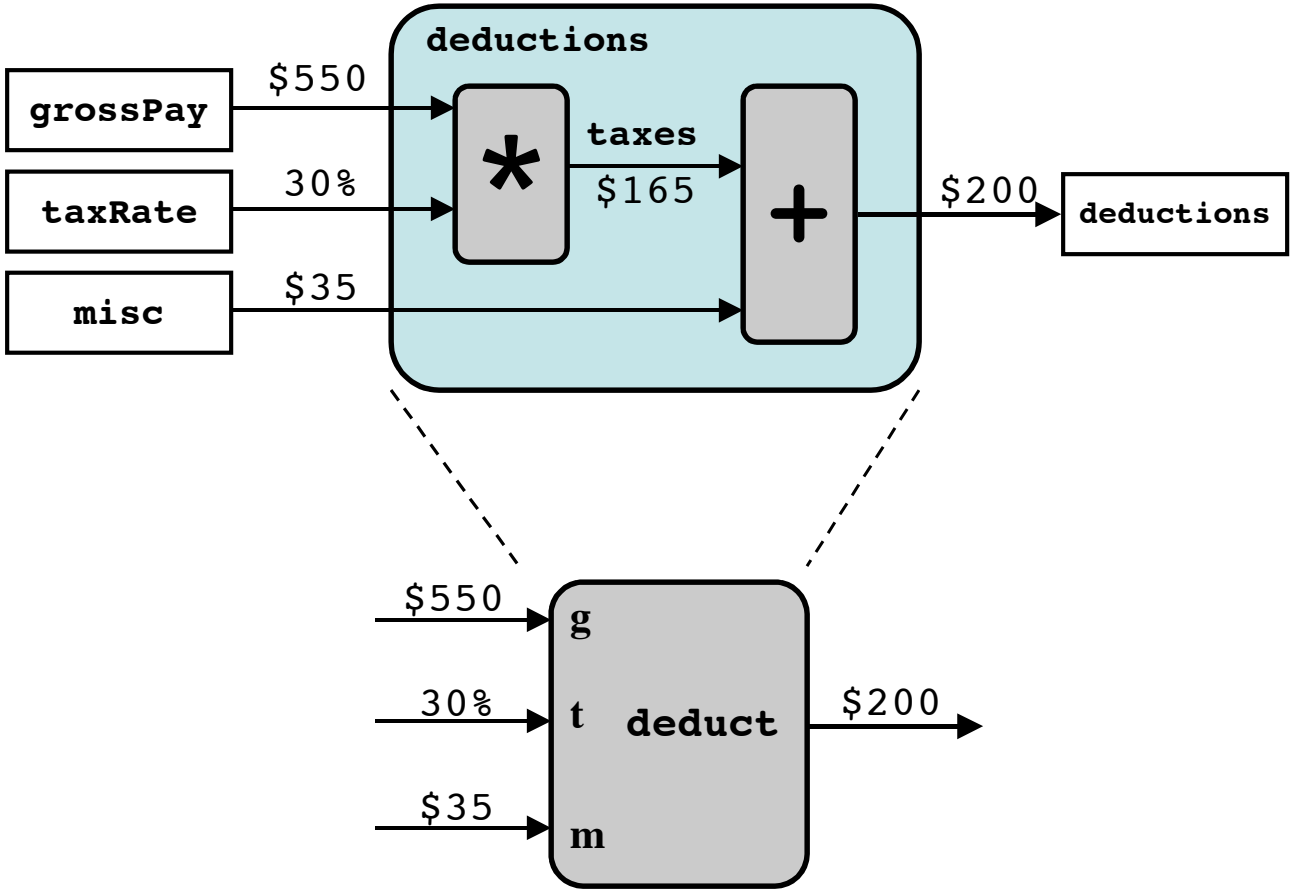
The previous Pay program did not compute the take-home pay (called the net pay). The previous pay called grossPay, is reduced by various deductions for taxes, dues, and other miscellaneous amounts.

Taxes, for example, are deductions which are computed at some rate (such as 30 percent) of the gross pay. This computation is shown below as two values (gross and payRate) going into a multiply box and the product ($165) leaving the box. Similarly, some miscellaneous values (of $35) can be added to compute the total amount of deductions (of $200).



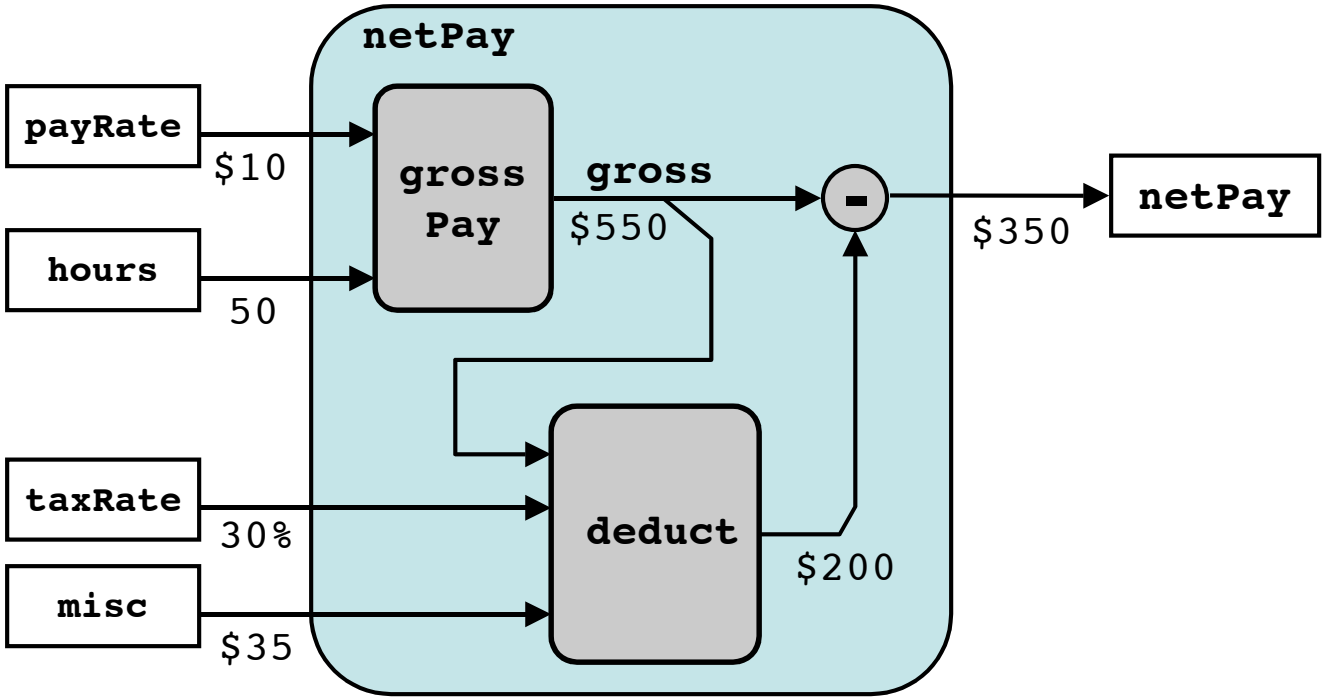
Hiding details

Details of the deductions can now be hidden in a new "black box" and this new box can now be a part of a larger pay program, as shown next.



DOP: Data Oriented Programming

Net pay is the gross pay minus the deductions. This is shown by the following data flow diagram, which combines the previous two diagrams.

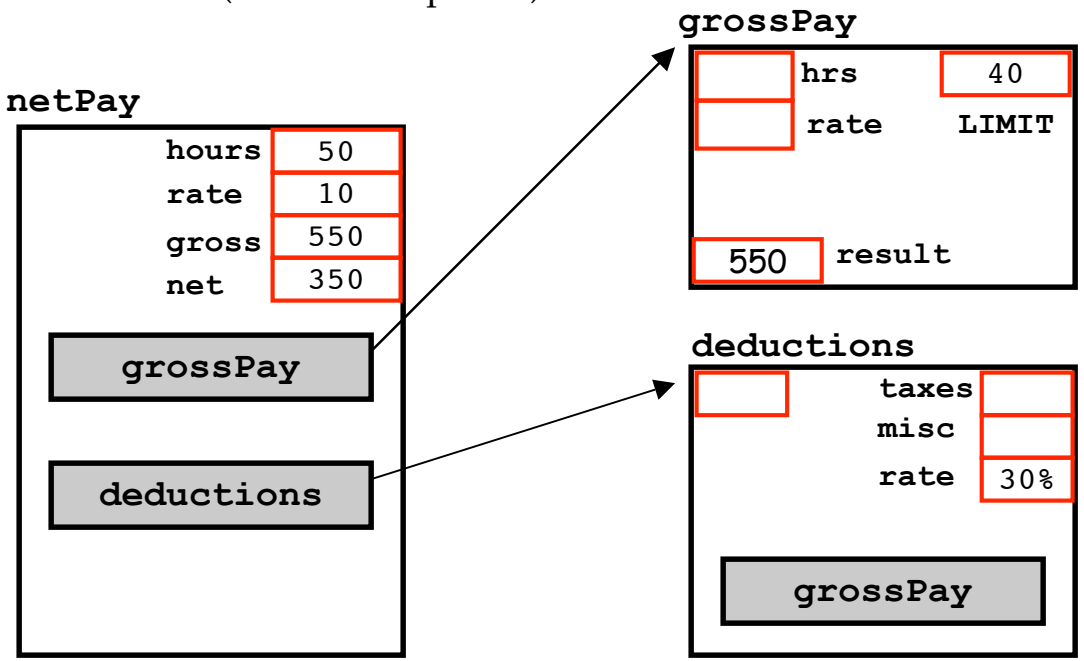


The grossPay and deduction subs can be coded as methods (functions or procedures), and the netPay can also be coded as one method which uses (or calls, invokes) the two other methods.

DSP: Data Space Programming (vs Data Flow Programming)

Data spaces are distributed between the various sub programs (or methods); all data is not stored in one place. Access to data is communicated in various ways.

Hiding and sharing of data values is critical; there are many ways to do it. Some data values are constant (such as the work hour LIMIT of 40 or the tax Rate). Other values may be passed as parameters, arguments, slots (such as the hours worked or pay rate) from one sub to another. Some values are **private** (or hidden), others **public** (or shared), and some restricted in other ways. Note that there are 3 boxes marked rate; this is not a problem for each is within a method hiding from the others. So if three different programmers decided to use the same name in three methods, they could do so without checking if the name is used by others. It is a design challenge to know when to hide values (make them private) and when to share values (make them public).



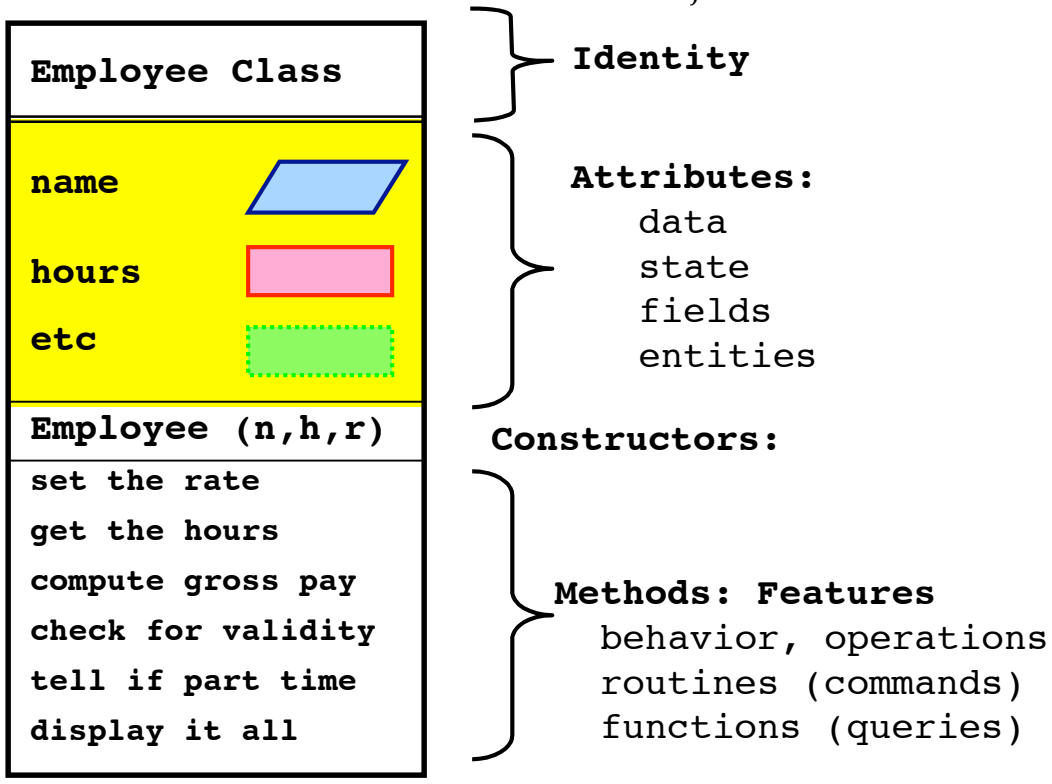
Class (Object) Oriented Paradigm: OOP

Objects consist of three parts as shown on the given diagram:

1. **identity**, given by a name such Employee (or Student, Bank Account, etc)
2. **attributes** (data, state, fields, entities) such as name, hours, payRate
3. **methods** (behavior, operations, features) such as setRate, getHours.

Classes specify the form or structure of a type of object.

Constructors set or initialize the attributes of an object.



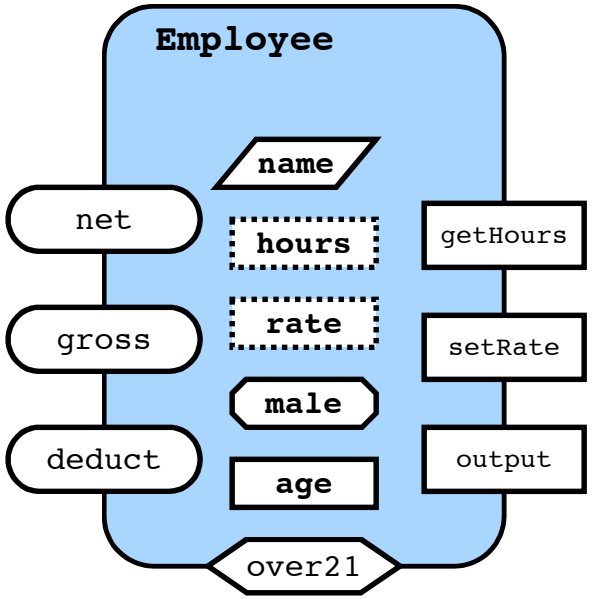
More Ways: to graphically show Classes

System Model

Employee	
Data (fields)	**Methods (actions)**
name	netPay
hours	grossPay
rate	deduction
male	getHours
age	setRate
seniority	over21?
children	married?
address	newName
phone	stateTax

Lists many aspects:
(reality is complex)
some may be irrelevant

Class Diagram



Hides attributes inside
accessed only by methods
(typed, boolean or void)

UML interface

Employee

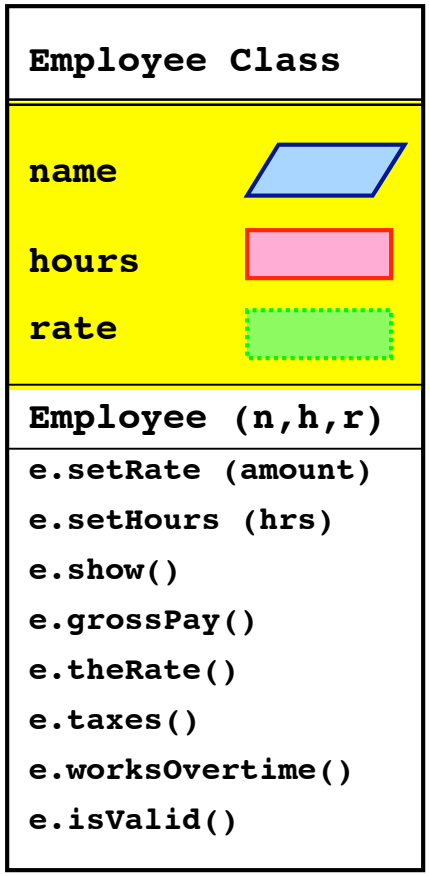
- **name**
- **hours**
- **rate**
- **male**
- **age**

+ **netPay**
+ **grossPay**
+ **deductions**
+ **getHours**
+ **setRate**
+ **over21**

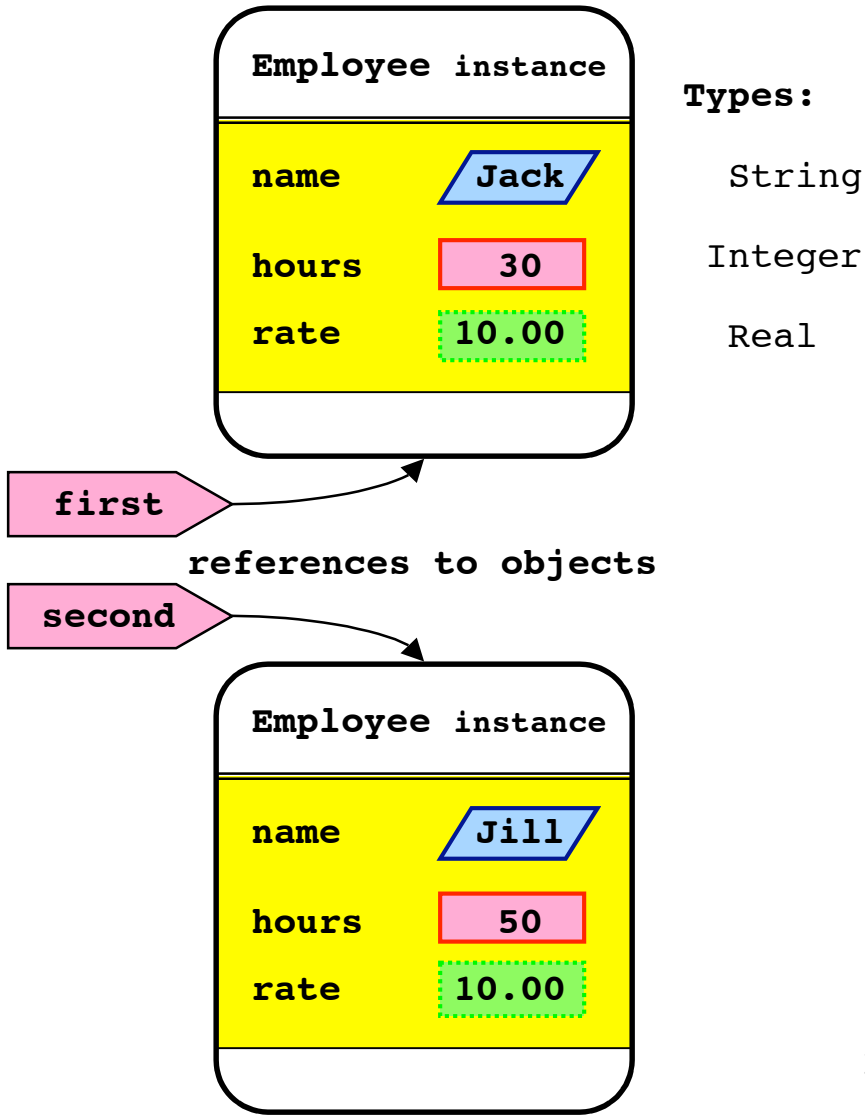
Universal Model Language
lists fields and methods
(- is private; + public)

Objects (many) are instances of a class (One)

Class
(general)



Objects (instances)
(particular)



Reference (or access) to objects is by a dot notation as in:

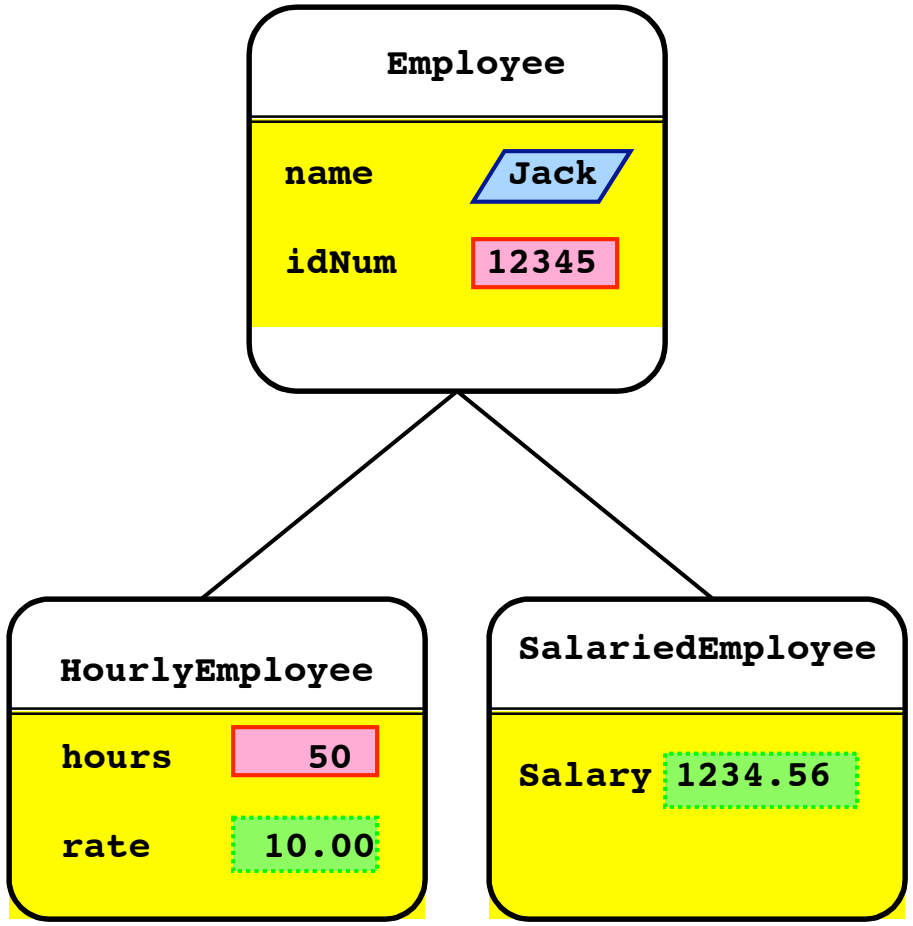
```
first.name = "Jack";
time = second.hours;
payRate = first.rate;
```

Inheritance
is a way of growing algorithms, by beginning generally, and then proceeding by specializing to specific details.

An Employee class, for example, begins generally with only those attributes and methods that are common to all kinds of employees (a name and id number).

From this "super" class other sub classes (such as an hourly employee or a salaried employee) can be extended as shown in the following figures.

The extended classes have all the general properties of the super class and also may have special other properties.



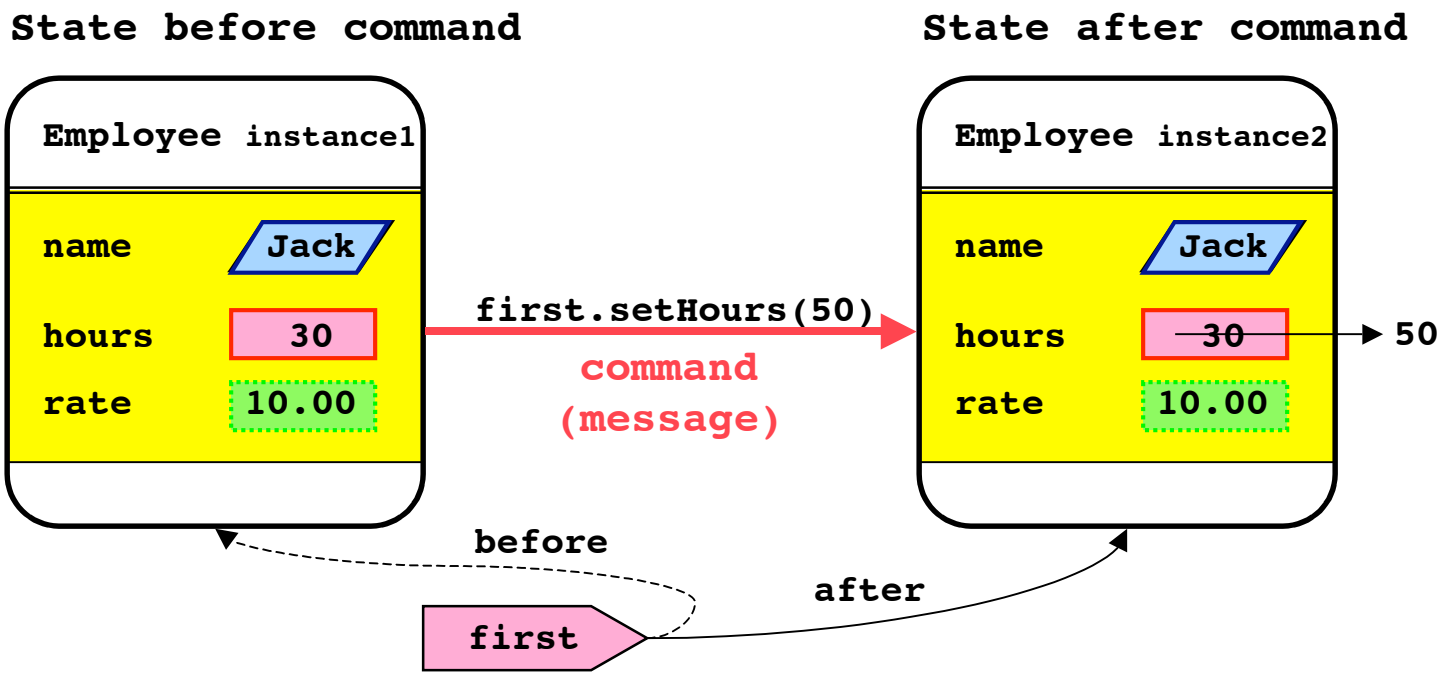
State Oriented Paradigm (Optional)

States describe the attributes of an object, and these states may change (make a transition) depending on various commands. For example, the state of an employee (named first) is given by the three values of the data fields. When receiving a command (or message) of the form:

```
first.setHours(50)
```

the **hours** part of the data of the **first** object is **set** to 50, destroying the previous value of 30.

The picture of the first object before and after this command (or message) is shown below.

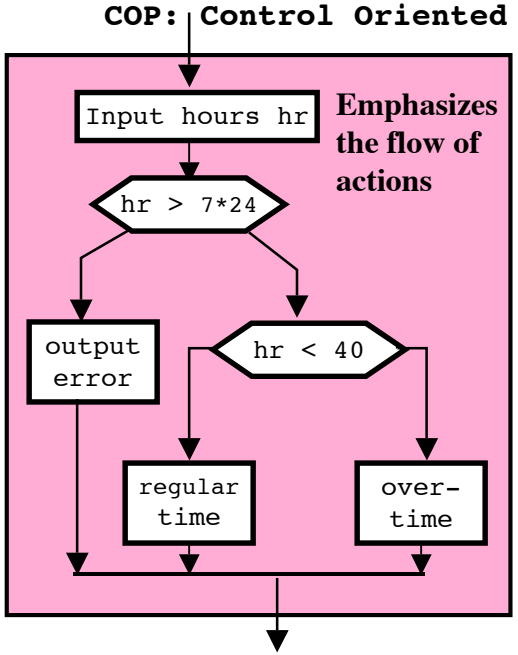


Summary:

There are many ways to view programming!

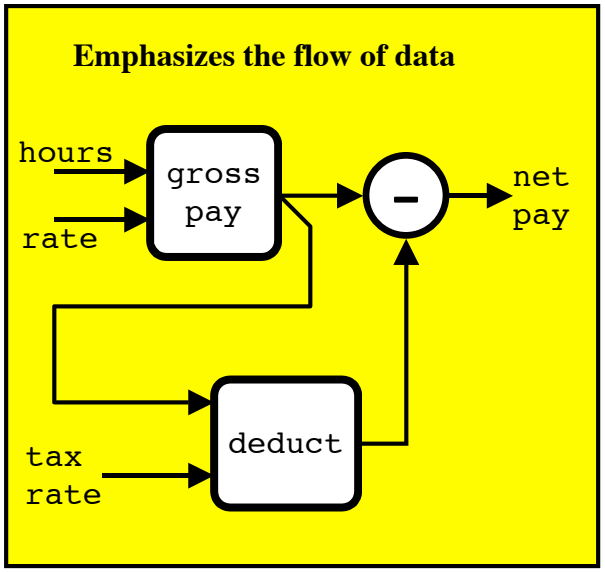
Three Principal Programming Paradigms are:

COP: Control Oriented



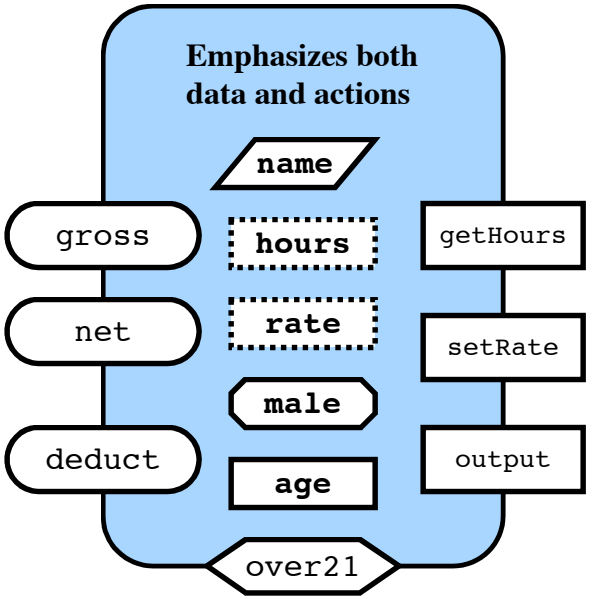
Concentrates on actions
- **imperative**
- **procedural**

DOP: Data Oriented



Concentrates on data
- **flow, hide & share**
- **store space access**

OOP: Object Oriented



Concentrates on "Class"
- **integrated**
- **simulated**

Pay Problems

1. More Pay

If the condition for overtime in Pay1 were changed
from `(h < 40) to (h <= 40)`,
does this change the amount of pay?
Explain your answer.

2. Negative Input

If the hour h is input as a negative amount (by mistake),
is the output correct except for its sign?

3. More Robust

Extend the robust pay2 diagram to indicate a proper error comment
when a negative value is entered.

4. Triple Pay

Extend the extended pay3 diagram for triple pay
when over 80 hours are worked.

5. Big Pay

Draw a flow chart which includes Pay1 (generalized), Pay2, Pay3, and Pay4
all on one page.

6. Bigger Pay

Draw a flow chart which includes the Pay1 to Pay4 above,
and also the More Robust pay and the Triple pay of problems 3 and 4.

More Rate Problems

1. Ideal Weight

A man should weigt 106 pounds for the first 5 feet, and 7 pounds for every inch above that; a woman should weigt 100 pounds for the first 5 feet and 6 pounds for each inch over it.

a.create an algorithm in flowchart form which outputs the ideal weight when input the sex and height in inches.

b. Represent this algorithm as two tables from 5 feet to 6 foot 6 inches.

c. Represent this algorithm in a graphic form, with 2 graphs on one grid.

2. Dogs Life

An algorithm which relates a dog's age to the corresponding human age follows.

A one year old dog is equivalent to a 15 year old human.
In the second year the dog grows 10 human years older, and
each year after that it grows 5 human years.

Create an algorithm with input of dog age and output human age

as a control flow diagram (or flow chart)

as a table, with dog age varying from 1, 2, … 10

as a graph (on a grid) of dog age vs human age

as a graph (on a grid) of human age vs dog age.

Price Break Problems

The selling price p of some item depends on the quantity q that is purchased.
The item may be discrete or digital (number of pens or puppies such as 7)
or it could be continuous or divisible (fractions of pounds of peanuts, such as 7.7).
If the quantity is less than or equal to 3 then the price is 4 dollars for each unit;
if the quantity is over 3 then the price is 3 dollars for each. For example:
when q is 4 then p is 3 and the total price t is simply p*q = 3*4 = 12 dollars.

1. Represent: Show

Draw a graph of p vs q and a flow chart or pseudo-code which
outputs the total cost t for any input quantity q from 0 to 8.

2. Analyze: Observe

Compute the total t when quantity q varies from 0 to 8.
Draw this as a table of 3 columns q, p and t.
Then draw a graph of t vs q; observe it for a surprise.
(Hint: how many items can be purchased for 12 dollars?)

3. Extend: Grow

Modify this algorithm if the price is further reduced to 2 dollars a unit
when the quantity purchased is more than 5; draw the extended flow chart.
Draw also the graph of t vs q.

4. FoolProof: Robust

Modify the above extended flow chart to include any foolproofing necessary.

5. Repeat

Modify the above flowchart to continue repeating until a negative input of q.

Energy Rate Problems

1. Encouraging excessive use

Suppose that electrical energy rates decreased with higher energy use as follows: The rate is a constant $4 per unit for the first 4 units, then drops to $2 per unit for more than 4 units. So if 6 units are used then the cost is 4*4 for the first 4 units plus 2*2 for the next 2 units for a total cost of $20. Also, if more than 8 units are used then the price per unit drops to $1 for those units over 8.

Make a table showing the cost c depending on the number of units u (for u = 0, 2, 4, 6, … 12). Draw a graph of the price per unit p vs the number of units u. Then draw also the graph of cost vs units. Finally make an algorithm which outputs the cost for any given value of units input.

2. Discouraging excessive use of energy

Suppose that the energy rates increase with higher energy use as follows:

The rate is a constant $4 per unit for the first 4 units, then increases to $8 per unit for the next 4 units. So if 6 units are used the cost is 4*4 + 2*8 = 32. Also if more than 8 units are used then the rate increases again to $10 per unit for those units over 8. Make the table, diagrams, and algorithm as in the previous problem

3. Severely discouraging excessive use of energy

Suppose that the energy rates increase more with higher energy use as follows:

The rate is a constant $4 per unit for the first 4 units, but increases to $8 per unit for all the units after that. So if 6 units were used then all 6 would be at the higher rate, costing a total amount of 6*8 = 48. Also, if more than 8 units were used then the rate increases to $10 per unit for all units. Make the table, diagrams, and algorithm.

4. Compare Rate Structures

Put all three graphs of cost vs units onto one graph.