Using Control Structures in Methods

Chapter 5

Chapter Contents

Objectives

- Give problem example requiring new control structures
- Take first look at basic control structures
  - sequential
  - selection
  - repetition
- Study the **`if`** statement used for selection

Objectives

- See use of `for` statement for counter-controlled repetitions
- See use of `for` statement used as "forever" loops
- Give applet example to generate output
- Brief indication of area of computability theory
- Describe use of numerical methods

5.1 Example: Improved Payroll Program

- Previous program (Figure 2.1) now must be upgraded
- Need capability of including overtime pay
- Desire for program to handle multiple employees, not just one

Additional Objects

Objects	Type	Kind	Name
In addition to previous objects …			
regular wages	`double`	variable	`regularPay`
overtime pay factor	`double`	constant	`OVERTIME_FACTOR`
overtime wages	`double`	variable	`overtimePay`
copmbined wages	`double`	variable	`wages`

Additional Operations

- Previous Operations …
  … plus …
- Compute **`regularPay`**, **`overtimePay`**, **`wages`**
- Display real values (**`wages`**)
- Repeat steps for each employee

Calculating Wages

More complicated than before:

```
if hoursWorked ≤ 40, calculate:
  regularPay = hoursWorked x
        hourlyRate;
  overtimePay = 0;
Otherwise, calculate:
  regularPay = 40 x hourlyRate
  overtimePay = OVERTIME_FACTOR  x
        (hoursWorked - 40) x
hourlyRate
wages = regularPay + overtimePay
```

Algorithm for New Payroll Program

- Construct **`Screen`** and **`Keyboard`** objects
- Display prompt for number of employees
- Read integer into **`numEmployees`**
- Loop from 1 through **`numEmployees`**
- For each employee …
  - Display prompts for hours, rate
  - Read doubles into **`hoursWorked`**, **`hourlyRate`**
  - Calculate **`wages`** according to previous algorithm
  - Display results with message

Coding and Testing

- Note source code Figure 5.1
  - looping structure

```
for( int count = 1 ;
         count <= numEmployees ;
         count++ )
    { … }
```

  - Selection structure

```
if( hours worked <= 40 )
  {   …   }
else
  {   …   }
```

  - Note sample runs

5.2 Methods That Use Selection

- Problem:
  Given two real values, return the minimum of the two values
- Behavior for our method
  - receive two real values from caller
  - if first less than second, return first
  - otherwise return second

Objects

Object	Type	Kind	Movement	Name
1st value	`double`	variable	received	*`first`*
2nd value	`double`	variable	received	*`second`*
minimum value	`double`	variable	returned	

Operations

- Receive two real values from method's caller
- Compare the two values to see if one is less than the other
- Do one (but not both of the following)
  - Return the first value
  - Return the second value

View Algorithm in Source Code

```
public static double minimum
(double first, double second)
{
  if (first < second)
      return first;
   else
      return second;
}
```

- Note driver program source code with sample runs, Figure 5.3

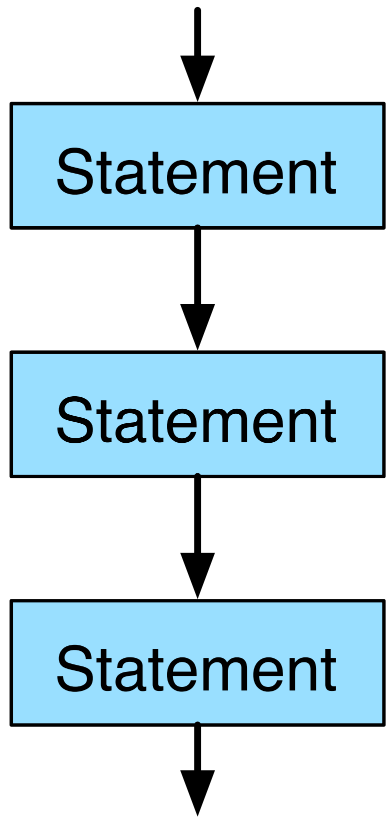
Programming Structures

- Sequential Execution
  - Like traveling down a straight road
  - Pass through a sequence of points or locations
- Selective Execution
  - Like coming to a fork in the road
  - We either take one direction or the other

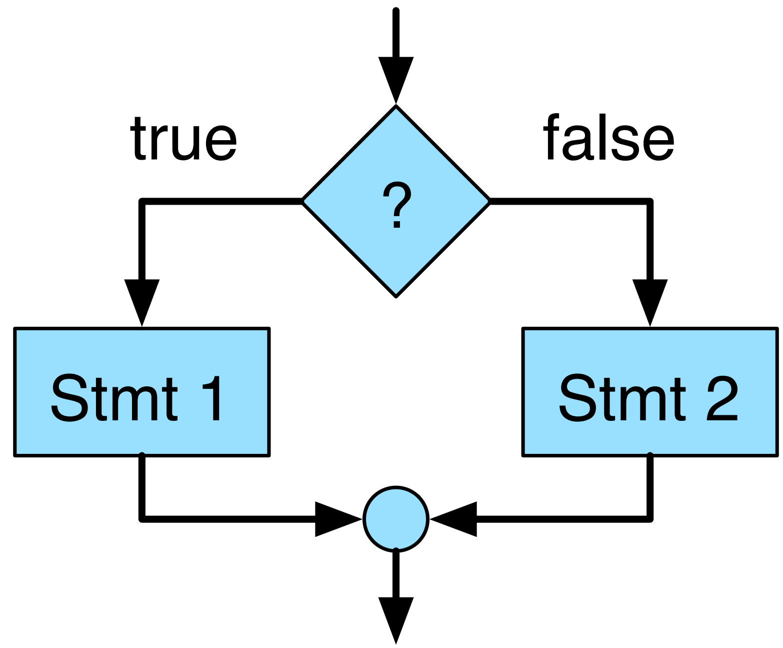
Programming Structures

Selective Execution

Sequential Execution

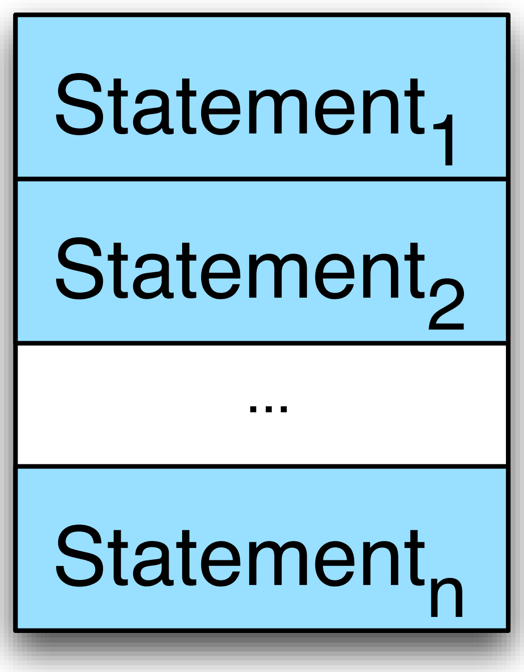


Selective Execution

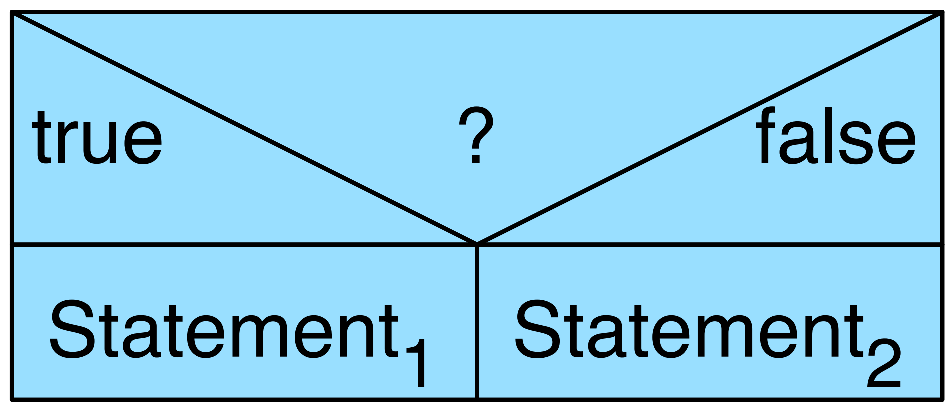


Alternate Graphical Representation

Sequential Execution



Selective Execution



IF Statement

- Two basic forms

```
if( boolean_expression )
    statement
```

Statement is only executed if **boolean_expression** is true

```
if( boolean_expression )
    statement1
  else
    statement2
```

Statement1 is executed if **boolean_expression** is true; otherwise **statement2** is executed

Blocks

- An `if` statement may need to control several statements
- A group or "block" of statements can be specified with braces

```
{
    statement1
    statement2
      . . .
}
```

- Note use in wage calculation

Checking Preconditions

- Some algorithms work correctly only if certain conditions are true
  - no zero in a denominator
  - non negative value for square root
- `if` statement enables checking

```
public static double f(double x)
{ if (x >=0)
      return 3.5*Math.sqrt(x);
    else {
      System.err.println( "invalid x" );
      return 0.0;
    }
}
```

Style

- Key issue is how well humans (not computers) can read the source code
- Form for `if` statements
  - Align the `if` and the `else`
  - Use indentation to mark statements being selected (controlled) by the `if` and `else`

Nested `ifs`

- Note the syntax of the **`if`** statement
  - it controls whether a statement will be executed
  - this statement could be another **`if`**
- Referred to as a “nested” if

```
if( boolean_expression1 )
    statement1
else if( boolean_expression2 )
    statement2
```

Method Signature

- Signature (unique identification) of a method made up of
  - the name of the method
  - the list of the types of parameters
- This means we could have two methods with the same name but different types and/or numbers of parameters

```
public static double minimum
    (double first, double second) …
public static int minimum
    (int first, int second)
```

Method Overloading

- Two different methods with the same name are said to be “overloaded”
- The name of a method can be overloaded, provided no two definitions of the method have the same signature

5.3 Methods That Use Repetition

- Problem: Computing factorials

$$n! = \begin{cases} 1 & n = 0 \\ 1 \times 2 \times \ldots \times n & n > 0 \end{cases}$$

- Write a method that given an integer n >= 0, computes n factorial (n!)

Object-Centered Design

- Behavior- repeated multiplication
- Objects

Object	Type	Kind	Movement	Name
integer >=0	variable	`int`	received	`n`
running product	variable	`int`	returned	`product`
counter	variable	`int`	(local)	`count`

Operations

1. Check precondition `(n >= 0)`
2. Define, initialize two integer variables
   - `product` and `count`
3. Multiply `product x count`, assign result to `product`
4. Increment `count`
5. Repeat 3. and 4. so long as `count <= n`

Algorithm

- Receive **`n`** from caller, check precondition
- Initialize **`product`** to **`1`**
- Repeat following for each value of **`count`** in range **`2`** through **`n`**
  - Multiply **`product`** by **`count`**
- Return **`product`**

Coding

- Note `factorial ()` method, Figure 5.4 in text
- Note Driver for Method `factorial ()`, Figure 5.5 in text
- Note test runs
    - with legal arguments
    - with invalid argument

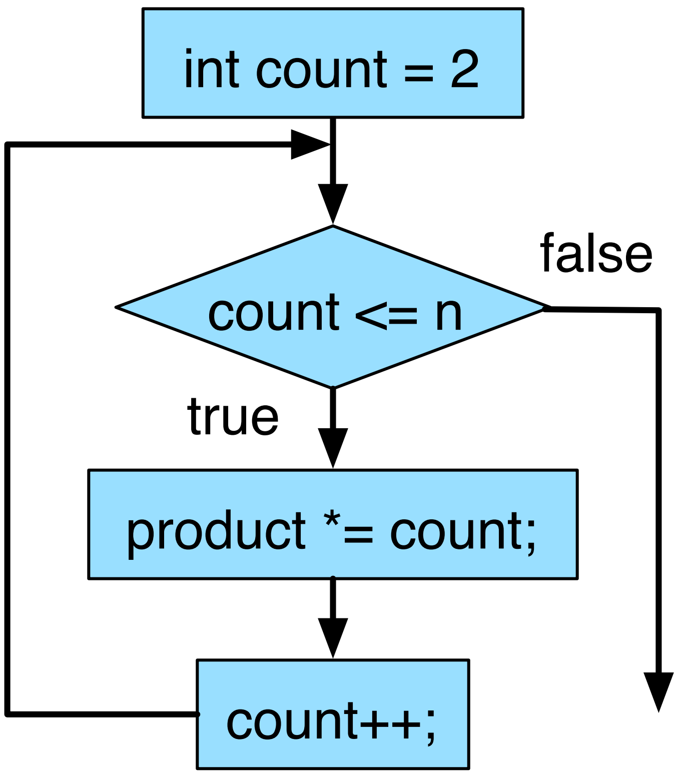
Repeated Execution: The `for` Statement

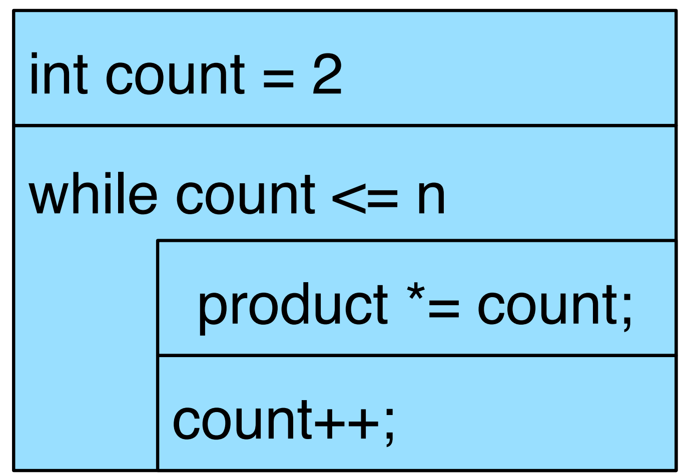
- Make analogy to a roadway
  - Think of a race track
  - Enter the track
  - Circle for a set number of times
  - Leave the track
- Three parts to the repetition mechanism
  - Initialization
  - Repeated execution
  - Termination

Flow Graph Example

Definite iteration

```
/* given */
 for (int count=2; count <= n; count++)
     product *= count
```





`for` Statement Syntax

- ```
  for (initExpression;
       booleanExpression;
       stepExpression)
    statement;
  ```
- **`for`** is a keyword
- **`initExpression`**: usually an assignment
- **`booleanExpression`**: usually a comparison (think "**`while`**")
- **`stepExpression`**: usually an increment
  ```

Typical `for` Execution

```
for (initExpression;
         booleanExpression
         stepExpression)
    statement;
```

1. Loop control variable given initial value
2. **`booleanExpression`** checked
   1. If it is **`true`**, statement executed
   2. If **`false`**, loop terminates
3. Increment of loop control variable
4. Back to step 2

Alternate Version of `for`

- Specifications inside the parentheses are not required
  - only the two semicolons

```
for ( ; ; )
 {. . .
   if ( … ) break;
   }
```

Termination or exit condition

Indefinite iteration

- **`break`** statement jumps flow of control out of for loop (See Figure 5.6 in text)

Sentinel Based Loop

- Often user asked to enter a sentinel value
- When sentinel value found in **`if ( )`**, loop terminates

```
for ( ; ; )
 {. . .
   if ( value is sentinel ) break;
 }
```

- Called “sentinel-based” input processing

Forever Loops

- Using `for ( )`
```
for ( ; ;)
 { . . .
   if (booleanExpression) break;
  . . . }
```
- Using `while ( )`
```
  while ( true )
 { . . .
   if (booleanExpression) break;
  . . . }
```
- Note: something in the loop must cause booleanExpression to evaluate to true
  - Otherwise the loop does go forever

Testing, Maintaining `factorial()`

- Method works correct for values 1 – 12
- Incorrect value for 13!
  - Algorithm is correct
  - Problem is use if type **`int`**
  - 13! exceeds maximum **`int`** value
- Solution is to change type returned (and received) by the method to
- Note new version and test runs, Figure 5.7 of text

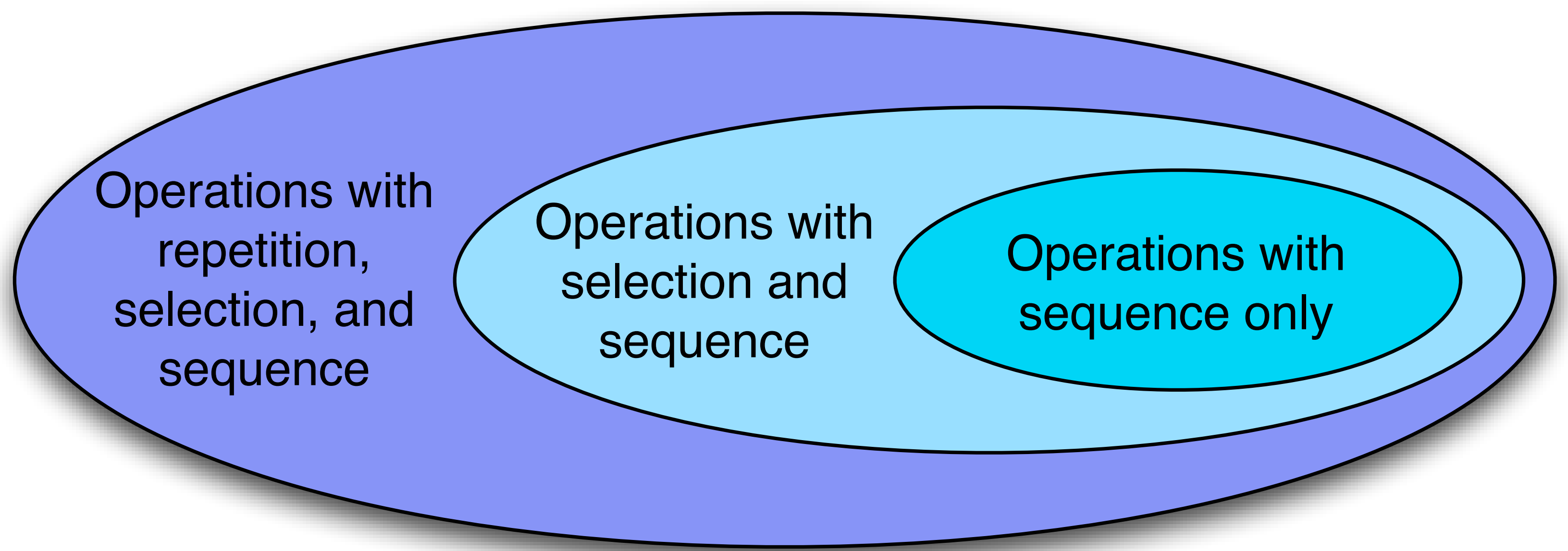
5.4 Graphical/Internet Java: Old MacDonald… Applet Revisited

- Write versions of the applet using more flexible structure
- Write for ( ) loop to receive inputs from user
  - name of animal
  - sound of animal
- See source code Figure 5.8, Text

Part of the Picture: Computability Theory

- Note the capabilities now available to us
  - sequential execution
  - selection (branching)
  - repetition (looping)

provide more capability



Computability Theory Considerations

- What kinds of operations can/cannot be computed?
- How can be operations be classified
  - What relationships exist among classes
- What is most efficient algorithm for solving a particular problem

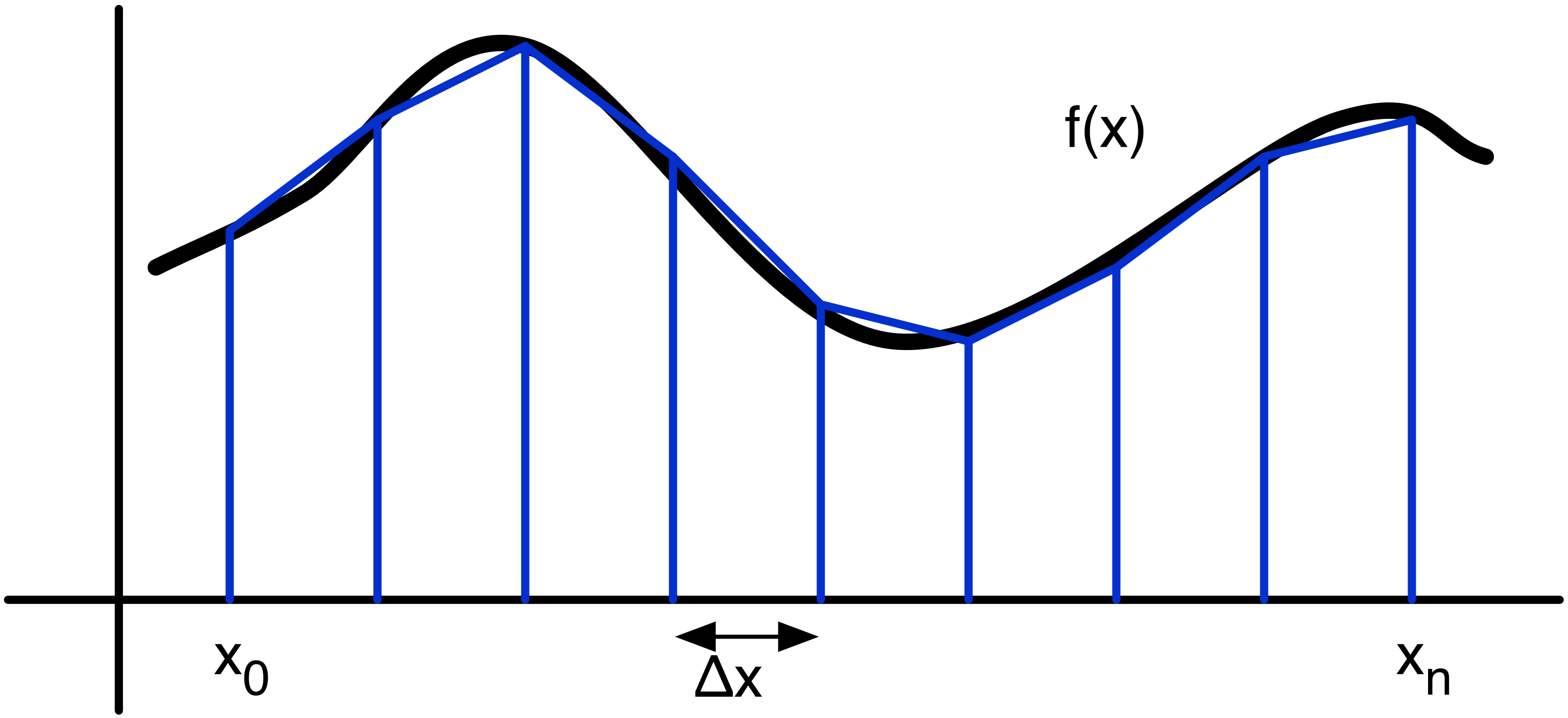
Computability Theory

- Represent programs abstractly
  - use mathematical model
- Provides language and hardware independence
  - gives theory with timelessness

Part of the Picture: Numerical Methods

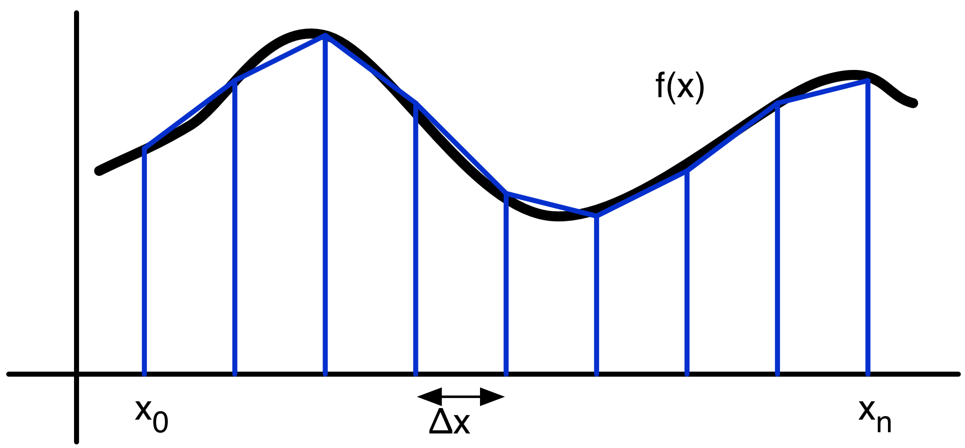
- Mathematical models used to solve variety of problems
  - Often involve solutions to different kinds of equations
- Examples
  - Curve fitting
  - Equation solving
  - Integration
  - Differential equations
  - Solving linear systems

Trapezoid Method for Approximating Areas



The sum of the areas of these trapezoids is approximately the area under the graph of f(x) between the points x_0 and x_n. The approximation improves as dx gets smaller

Trapezoidal Method



Use this formula as an algorithm for calculating approximation of area.

$$\text{area} = \Delta x \left(\frac{f(x_0) + f(x_n)}{2} + \sum_{i=1}^{n-1} f(x_i) \right)$$

`TrapezoidalArea()` Method

- Note source code Figure 5.9 in text
- Tasks
  - screen prompts for y values
  - inside **`for()`** loop sums the successive f(x) values
  - calculates and returns total area under curve
- Method applied to road construction
  - determine total volume of dirt removed for highway (Figure 5.10, text)
  - cross section is trapezoid