C#

Method Overloading Parameter Passing Variable Scope & Duration

01204111 Computer & Programming

Dr. Arnon Rungsawang
http://mike.cpe.ku.ac.th/01204111

Department of Computer Engineering Kasetsart University

MIKE LABORATORY

Review: Method

Method

- A **method's name** should provide a well-defined, easy-to-understand functionality.
  - A method takes input (parameters), performs some actions, and (sometime) returns a value.
- Writing a custom method
  - Header

    (modifier) Properties ReturnType MethodName (Param1, Param2, …)

  - Body
    - Contains the code of what the method does.
      - Local variables declaration
      - Statements
    - Contains the **return** value if necessary.
  - All methods must be defined inside of a class.

```
// MaximumValue.cs
// Finding the maximum of three doubles.

using System;

class MaximumValue
{
  // main entry point for application
  static void Main( string[] args )
  {
    // obtain user input and convert to double
    Console.Write( "Enter first floating-point value: " );
    double number1 = Double.Parse( Console.ReadLine() );

    Console.Write( "Enter second floating-point value: " );
    double number2 = Double.Parse( Console.ReadLine() );

    Console.Write( "Enter third floating-point value: " );
    double number3 = Double.Parse( Console.ReadLine() );

    // call method Maximum to determine largest value
    double max = Maximum( number1, number2, number3 );

    // display maximum value
    Console.WriteLine("\nmaximum is: " + max );

  } // end method Main
```

The program gets 3 values from the user

The three values are then passed to the Maximum method for use

```

  // Maximum method uses method Math.Max to help determine
  // the maximum value
  static double Maximum( double x, double y, double z )
  {
    return Math.Max( x, Math.Max( y, z ) );

  } // end method Maximum

} // end class MaximumValue
```

The Maximum method receives 3 variables and returns the largest one

The use of Math.Max uses the Max method in class Math. The dot operator is used to call it.

```
Enter first floating-point value: 37.3
Enter second floating-point value: 99.32
Enter third floating-point value: 27.1928

maximum is: 99.32
```

The Dual Roles of C# Classes

- Program modules:
  - A list of (static) method declarations and (static) data fields.
  - To make a method static, a programmer applies the **`static`** modifier to the method definition.
  - The result of each invocation of a class method is completely determined **by** the actual parameters (and static fields of the class)
  - To use a static method: **`ClassName.MethodName(…);`**
- Blueprints for generating objects:
  - Create an object
  - Call methods of the object: **`objectName.MethodName(…);`**

Dog and Cat class

```
using System;
class DogCat
{
 // object internal property or state
 static int NoPets = 0;
 int leg = 4, ear = 2, tail = 1;
 string color ="", cryingSound ="", name="";
 // constructor, run only once object is created
 DogCat ()
 {
   this.name = "toop"; this.cryingSound = "hong";
   NoPets++;
 }
 DogCat (string n, string c, string cs)
 {
   this.name = n; this.color = c;
   this.cryingSound = cs; NoPets++;
 }
```

Dog and Cat class (2)

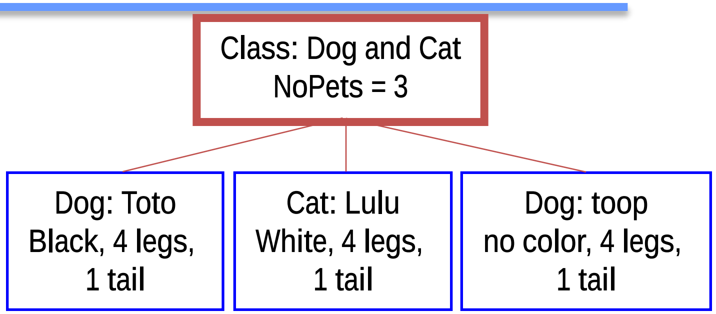
```
// ohter methods to transfer object's property to other state
void cutTail () {
  string n = this.isDogOrCat();
    if (this.tail == 0)
      Console.WriteLine("Your {0} already has no tail!", n);
    else
      {
        this.tail = 0;
        Console.WriteLine("OK, your {0} \'s tail has been cut. ", n);
      }
}
string isDogOrCat () {
  string s;
  if (this.cryingSound == "hong")  s = "dog, named \"";
  else  s = "cat, named \"";
  s += this.name; s += "\",";
  return s;
}
```

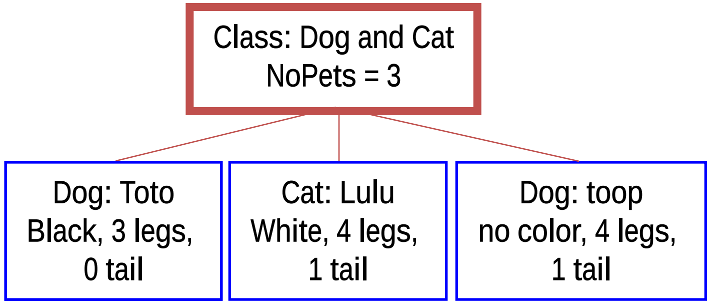
Dog and Cat class (3)

```
void hitByCar (int leg)
{
  if (this.leg > leg)
    this.leg = this.leg - leg;
}
string myPrint ()
{
  string s = this.isDogOrCat();
  s += String.Format(" has {0} leg(s), {1} ears, {2} tail,
            color = {3}", this.leg, this.ear, this.tail, this.color);
  return s;
}
static int NumberOfPets()
{
  return NoPets;
}
```

Dog and Cat class (4)

```
     // Main here
     public static void Main ()
     {
       DogCat a = new DogCat("Toto", "black", "hong");
       DogCat b = new DogCat("Lulu", "white", "miao");
       DogCat c = new DogCat();

       Console.WriteLine("\nNumber of pets are {0}.",
                       NumberOfPets());
       Console.WriteLine("My {0}.", a.myPrint());
       Console.WriteLine("My {0}.", b.myPrint());
       Console.WriteLine("My {0}.", c.myPrint());

       a.cutTail();
       a.hitByCar(1);
       Console.WriteLine("\nNumber of pets are {0}.", NumberOfPets());
       Console.WriteLine("My {0}.", a.myPrint());
       Console.WriteLine("My {0}.", b.myPrint());
       Console.WriteLine("My {0}.", c.myPrint());

       Console.ReadLine();
     }
   }
```





Explicitly Creating Objects

- A class name can be used as a type to declare an *object reference variable.*

```
String title;
Random myRandom;
```

  - An object reference variable holds the address of an object.
  - No object has been created with the above declaration.
  - The object itself must be created using the **new** keyword.

Creating and Accessing Objects

- We use the **new** operator to create an object

```
Random myRandom;
myRandom = new Random();
```

This calls the Random *constructor*, which is a special method that sets up the object.

- Creating an object is called **instantiation**.
  - An object is an *instance* of a particular class.
- To call an (*instance*) method on an object, we use the variable (not the class), e.g.,

```
Random generator1 = new Random();
int num = generate1.Next();
```

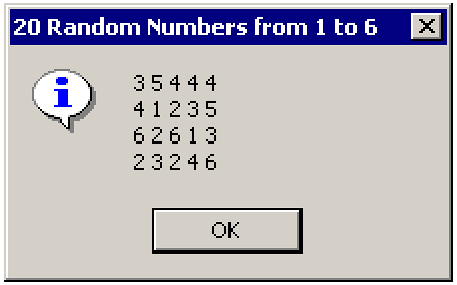
Example: the Random class

Some methods from the Random class

```
Random Random ()

 int   Next ()
         // returns an integer from 0 to Int32.MaxValue


 int   Next (int max)
         // returns an integer from 0 upto but not including max

int   Next (int min, int max)
        // returns an integer from min upto but not including max


double NextDouble ( )
        // returns a double number from 0 to 1
```

See RandomNumbers.cs click

```
// RandomInt.cs
// Random integers.

using System;
using System.Windows.Forms;

// calculates and displays 20 random integers
class RandomInt
{
   // main entry point for application
   static void Main( string[] args )
   {
      int value;
      string output = "";

      Random randomInteger = new Random();

      // loop 20 times
      for ( int i = 1; i <= 20; i++ )
      {
         // pick random integer between 1 and 6
         value = randomInteger.Next( 1, 7 );
         output += value + " "; // append value to output

         // if counter divisible by 5, append newline
         if ( i % 5 == 0 )
            output += "\n";

      } // end for structure
      MessageBox.Show( output, "20 Random Numbers from 1 to 6",
            MessageBoxButtons.OK,MessageBoxIcon.Information );
    }
}
```


20 Random Numbers from 1 to 6

```
3 5 4 4 4
4 1 2 3 5
6 2 6 1 3
2 3 2 4 6
```

Creates a new Random object

Will set value to a random number from1 up to but not including 7

Format the output to only have 5 numbers per line

Static vs. Instance Methods

- If a method is a static method
  - Call the method by `ClassName.MethodName(…);`

- If a method of a class is not a static method, then it is an instance method.
  - Create an object using the `new` operator.
  - Call methods of the object: `objectVariableName.MethodName(…);`

Method overloading

Method Overloading

- Using the `WriteLine` method for different data types:

```
Console.WriteLine ("The total is:");
double total = 0;
Console.WriteLine (total);
```

- **Method overloading** is the process of using the same method name for multiple methods (or usages).
  - Usually perform the same task on different data types.

- Example: The `WriteLine` method is overloaded:

```
WriteLine (String s)
WriteLine (int i)
WriteLine (double d)
 ...
```

Method Signature

- The compiler must be able to determine which version of the method is being invoked.
- This is done by analyzing the parameters, which form the signature of a method
  - The signature includes the number, type, and order of the parameters.
  - The return type of the method is not part of the signature.

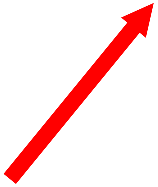
Method overloading example (1)

Version 1

```
double TryMe (int x)
{
   return x + .375;
}
```

Version 2

```
double TryMe (int x, double y)
{
   return x*y;
}
```



Invocation

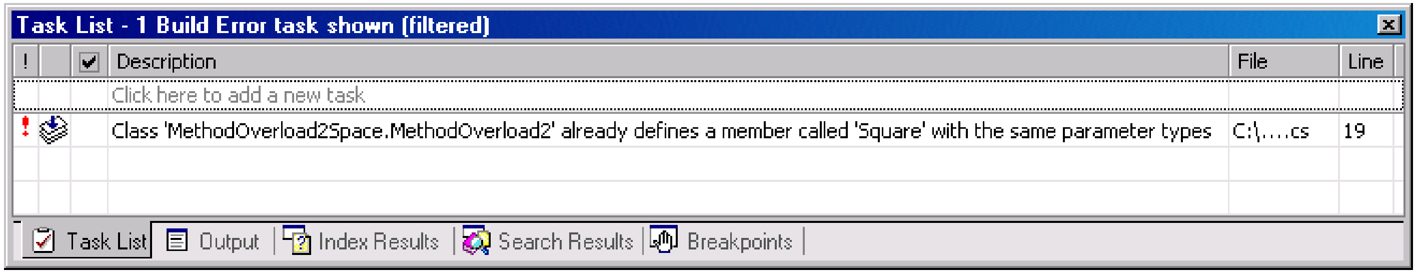
```
result = TryMe (25, 4.32)
```

```
// MethodOverload2.cs
// Overloaded methods with identical signatures and
// different return types.

using System;

class MethodOverload2
{
   static int Square( double x )
   {
      return x * x;
   }

   // second Square method takes same number,
   // order and type of arguments, error
   static double Square( double y )
   {
      return y * y;
   }

   // main entry point for application
   static void Main()
   {
      int squareValue = 2;
      Square( squareValue );
   }

} // end of class MethodOverload2
```

This method returns an integer

This method returns a double number

Since the compiler cannot tell which method to use based on passed values an error is generated



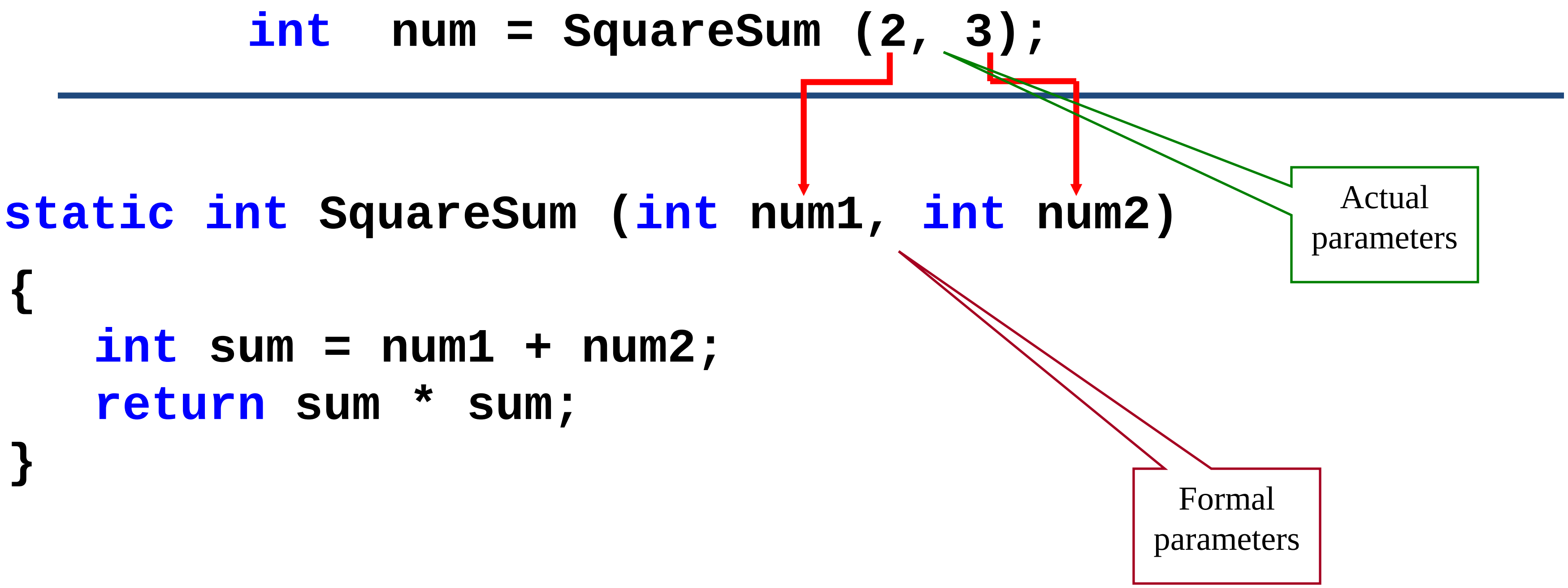
Task List - 1 Build Error task shown (filtered)

!		☑	Description	File	Line
			Click here to add a new task		
!			Class 'MethodOverload2Space.MethodOverload2' already defines a member called 'Square' with the same parameter types	C:\....cs	19

Paramethers Passing

Recall: Calling a Method

- Each time a method is called, the *actual arguments* in the invocation are copied into the *formal arguments.*



```
int  num = SquareSum (2, 3);
```

```
static int SquareSum (int num1, int num2)
{
    int sum = num1 + num2;
    return sum * sum;
}
```

Parameters: Modifying Formal Arguments

- You can use the formal arguments (parameters) as variables inside the method.
- Question: If a formal argument is modified inside a method, will the actual argument be changed?

```
static int Square ( int x )
{
   x = x * x;
   return x;
}
```

```
static void Main ( string[] args )
{
    int x = 8;
    int y = Square( x );
    Console.WriteLine ( x );
}
```

Parameter Passing

- If a modification on the formal argument has no effect on the actual argument,
  - it is call by value.

- If a modification on the formal argument can change the actual argument,
  - it is call by reference.

Call-By-Value vs. Call-By-Reference

- Depend on the type of the formal argument.
- For the simple data types, it is call-by-value.
- Change to call-by-reference
  - The `ref` keyword and the `out` keyword change a parameter to call-by-reference.
    - If a formal argument is modified in a method, the value is changed.
    - The `ref` or `out` keyword is required in both method declaration and method call.
    - `ref` requires that the parameter be initialized before enter a method,

      while `out` requires that the parameter be set before return from a method.

Example: ref

```
static void Foo( int p ) {++p;}
static void Main ( string[] args )
{
   int x = 8;
   Foo( x ); // a copy of x is made
   Console.WriteLine( x );
}
```

```
static void Foo( ref int p ) {++p;}
static void Main ( string[] args )
{
   int x = 8;
   Foo( ref x ); // x is ref
   Console.WriteLine( x );
}
```

See TestRef.cs

Example: out

```
static void Split( int timeLate,
                   out int days,
                   out int hours,
                   out int minutes )
{
    days = timeLate / 10000;
    hours = (timeLate / 100) % 100;
    minutes = timeLate % 100;
}

static void Main ( string[] args )
{
   int d, h, m;
   Split( 12345, out d, out h, out m );
   Console.WriteLine( “{0}d {1}h {2}m”, d, h, m );
}
```

See TestOut.cs

```
// RefOutTest.cs
// Demonstrating ref and out parameters.

using System;
using System.Windows.Forms;

class RefOutTest {
   // x is passed as a ref int (original value will change)
  static void SquareRef( ref int x ) {
      x = x * x;
   }

   // original value can be changed and initialized
   static void SquareOut( out int x ) {
      x = 6;
      x = x * x;
   }

   // x is passed by value (original value not changed)
   static void Square( int x ) {
      x = x * x;
   }

   static void Main( string[] args ) {
      // create a new integer value, set it to 5
      int y = 5;
      int z;   // declare z, but do not initialize it

```

When passing a value by reference the value will be altered in the rest of the program as well

Since the methods are **void** they do not need a return value.

Since x is passed as **out** the variable can then be initialed in the method

Since not specified, this value is defaulted to being passed by value. The value of x will not be changed elsewhere in the program because a duplicate of the variable is created.

```
      // display original values of y and z
      string output1 = "The value of y begins as "
         + y + ", z begins uninitialized.\n\n\n";

      // values of y and z are passed by value
      RefOutTest.SquareRef( ref y );
      RefOutTest.SquareOut( out z );

      // display values of y and z after modified by methods
      // SquareRef and SquareOut
      string output2 = "After calling SquareRef with y as an " +
         "argument and SquareOut with z as an argument,\n" +
         "the values of y and z are:\n\n" +
         "y: " + y + "\nz: " + z + "\n\n\n";

      // values of y and z are passed by value
      RefOutTest.Square( y );
      RefOutTest.Square( z );

      // values of y and z will be same as before because Square
      // did not modify variables directly
      string output3 = "After calling Square on both x and y, " +
         "the values of y and z are:\n\n" +
         "y: " + y + "\nz: " + z + "\n\n";

      MessageBox.Show( output1 + output2 + output3,
         "Using ref and out Parameters", MessageBoxButtons.OK,
         MessageBoxIcon.Information );

   } // end method Main
} // end class RefOutTest
```

The calling of the SquareRef and SquareOut methods

The calling of the SquareRef and SquareOut methods by passing the variables by value

Variable Scope & Duration

Review: Method Overloading

- Method overloading is the process of using the same method name for multiple methods.
  - Usually perform the same task on different data types.
- The compiler determines which version of the method is being invoked by analyzing the parameters, which form the signature of a method.
  - If multiple methods match a method call, the compiler picks the best match.
  - If none matches exactly but some implicit conversion can be done to match a method, then the method is invoked with implicit conversion.

Method overloading example

```
double TryMe ( int x )
{
   return x + 5;
}
```

```
double TryMe ( double x )
{
   return x * .375;
}
```

```
double TryMe (double x, int y)
{
   return x + y;
}
```

```
TryMe( 1 );

TryMe( 1.0 );

TryMe( 1.0, 2);

TryMe( 1, 2);

TryMe( 1.0, 2.0);
```

Click here

Recap: Parameter Passing

- Two types of parameter passing:
  - Call by value: a modification on the formal argument has no effect on the actual argument.
  - Call by reference: a modification on the formal argument can change the actual argument.
  - Depend on the type of a formal argument.
- *For C# simple data types, it is call-by-value.*
- Change to call-by-reference: **`ref`** or **`out`**
  - The **`ref`** or **`out`** keyword is required in *both method declaration and method call.*
  - **`ref`** requires that the parameter be initialized before enter a method.
  - **`out`** requires that the parameter be set before return from a method.

Variable Duration and Scope

- Duration
  - Recall: a variable occupies some memory space.
  - The amount of time a variable exists in memory is called its duration.
- Scope
  - The section of a program in which a variable can be accessed (also called visible).
  - A variable can have two types of scopes;
    - Class scope
      - From when created in a class,
      - Until end of class (}).
      - Visible to all methods in that class.
    - Block scope

Local Variables

- Created when declared.
- Until end of block, e.g., }.
- Only used within that block.

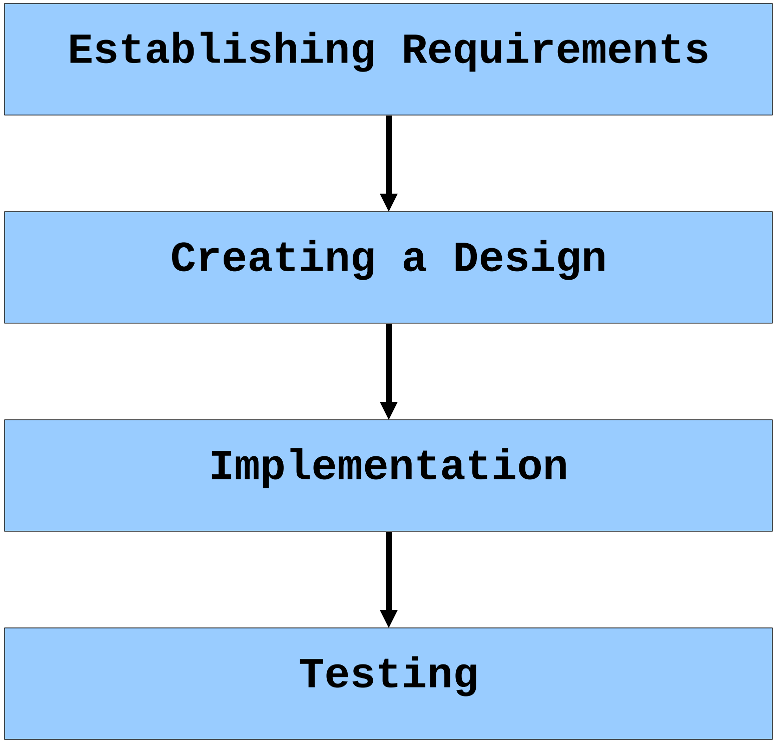
```
class Test
{
    const int NoOfTries = 3; // class scope
    static int Square ( int x ) // formal arg.
    {
       // NoOfTries and x in scope
       int square = x * x; // square local var.
       // NoOfTries, x and square in scope
       return square;
    }
    static int AskForAPositiveNumber ( int x )
    {
       // NoOfTries and x in scope
       for ( int i = 0; i < NoOfTries; i++ )
       {  // NoOfTries, i, and x in scope
          string str = Console.ReadLine();
          // NoOfTries, i, x, and str in scope
          int temp = Int32.Parse( str );
          // NoOfTries, i, x, str and temp in scope
          if (temp > 0) return temp;
       }
       // now only x and NoOfTries in scope
       return 0;
    } // AskForPositiveNumber
    static void Main( string[] args )
    {…}
}
```

Scope & Duration : What's matter?

- Scope
  - – A local variable is accessible after it is declared and before the end of the block.
  - – A class variable is accessible in the whole class.
  - – Parameter passing with `ref` and `out` makes some variables aliases of others.
- Duration
  - – A local variable may exist but is not accessible in a method,
    - e.g., method A calls method B, then the local variables in method A exist but are not accessible in B.

Program Development Process

Program Development Process



The development process is much more involved than this, but these basic steps are a good starting point.

Requirements

- *Requirements* specify the tasks a program must accomplish
  - what to do, not how to do it!
- A requirement often includes a description of user interface.
- An initial set of requirements are often provided, but usually must be critiqued, modified, and expanded.
  - It is often difficult to establish detailed, unambiguous, complete requirements.
  - Users do not know what they need – they will know when they see it – prototype to help.

Design

- Design methodology:
  - The top-down or stepwise methodology
    - Use methods (also called functions) to divide a large programming problem into smaller pieces that are individually easy to understand and reusable.
    - Also called decomposition.
  - Object-oriented design
    - Establishes the classes, objects, and methods that are required.
- Many ways to represent design
  - Pseudocode
  - Flow chart

Implementation

- *Implementation* is the process of translating a design into source code.
  - This is actually the least creative step -- almost all important decisions are made during requirements and design.
  - Many tools can help to convert a design to an implementation.
- Implementation should focus on coding details, including style guidelines and documentation.

Testing

- A program should be executed multiple times with various input in an attempt to find errors
- A testing methodology:
  - combine implementation with testing
    - write a piece, test a piece.

- Debugging is the process of discovering the cause of a problem and fixing it.

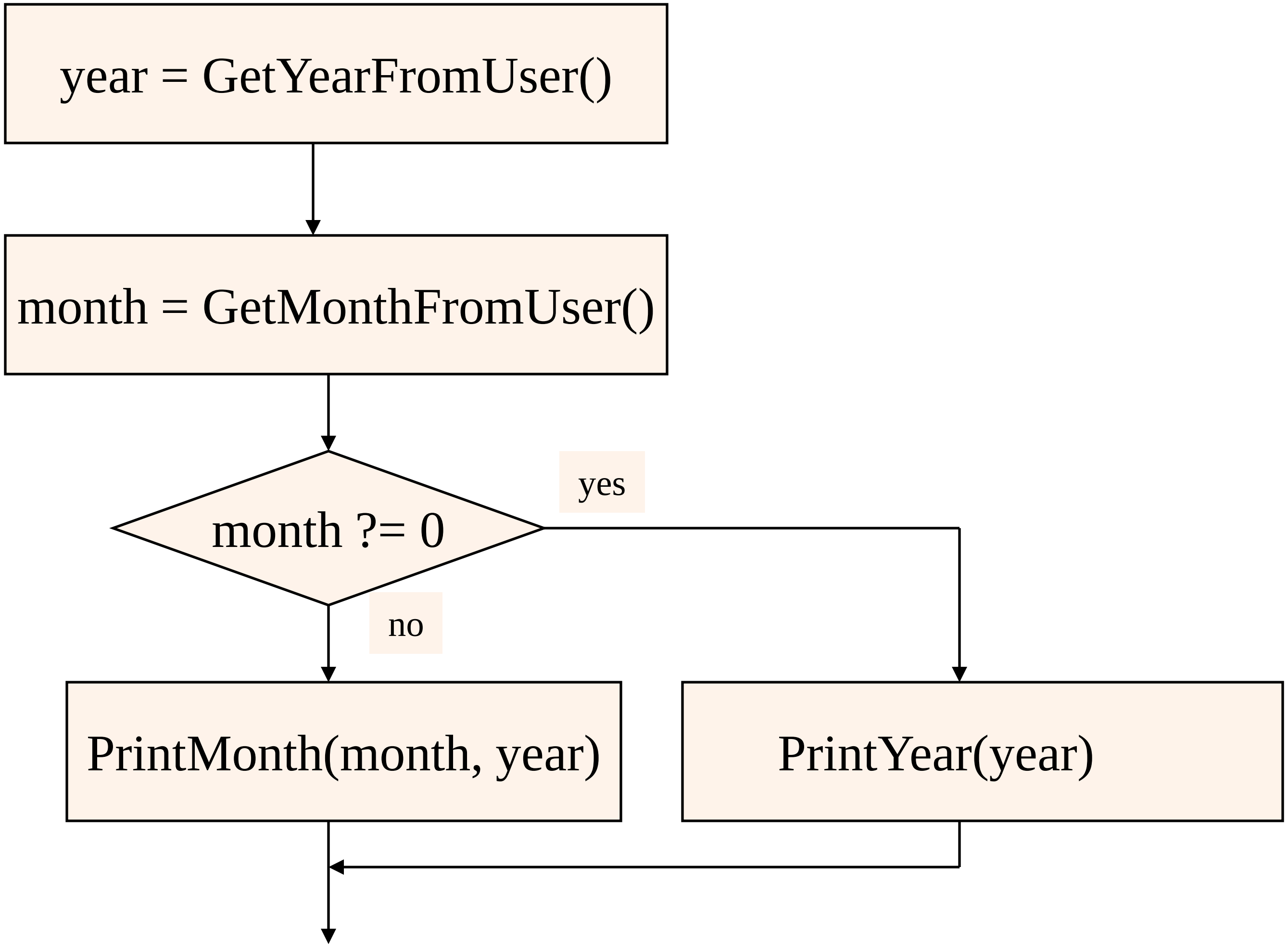
Calendar: Requirements

- Get a year from the user, the earliest year should be 1900.
- Get a month from the user, the input should be from 0 to 12.
  - If 0, print calendar for all 12 months.
  - Otherwise, print the calendar of the month of the year.

Design: Stepwise Refinement

- *Stepwise refinement* (or top-down design)
  - Start with the main program.
  - Think about the problem as a whole and identify the major pieces of the entire task.
  - Work on each of these pieces ***one by one.***
  - For each piece, think what is its major sub-pieces, and repeat this process.

Design


year = GetYearFromUser()

month = GetMonthFromUser()

month ?= 0

yes

no

PrintMonth(month, year)

PrintYear(year)

PrintMonth(month, year)

```
         January 1900
Sun Mon Tue Wed Thu Fri Sat
      1   2   3   4   5   6
  7   8   9  10  11  12  13
 14  15  16  17  18  19  20
 21  22  23  24  25  26  27
 28  29  30  31
```