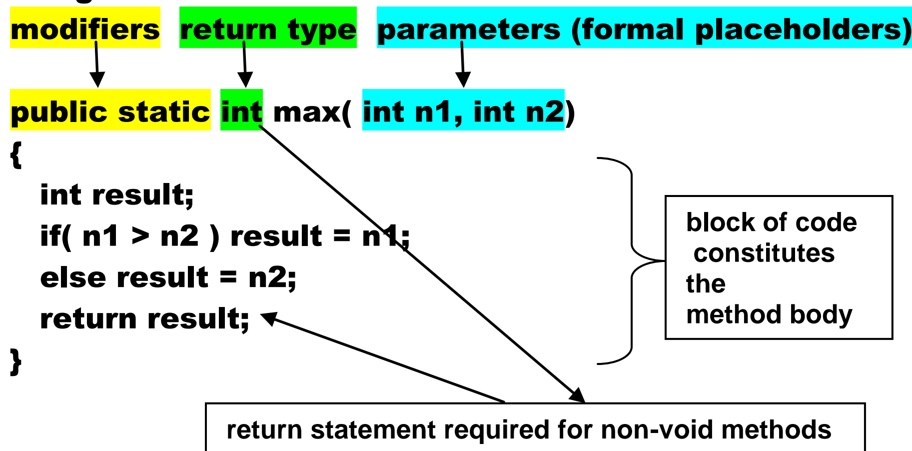
Lecture Notes
Chapter #5
Methods

1. **Method –**
   a. **group of statements, bundled together, designed to perform a specific function**
   b. **may be reused many times**
      i. **in a particular program or**
      ii. **in multiple programs**

2. **Examples – from the Java Library**
   a. **System.out.println( )**
   b. **JOptionPane.showMessageDialog()**
   c. **Integer.parseInt( )**
   d. **Math.random( )**

3. **Defining a Method**



Remark: Methods are defined outside of the main( ) program!

4. **Invoking, i.e., "calling", a Method**

```
public static void main( String [ ]args )
{
  int x, y;
  // input values for x and y;
  int z = max(x, y);
  // output the z value;
}
```

← **values of the actual parameters, i.e., arguments**

Remark: Methods are called inside of the main( ) program!

5. **Method Return Types**
   a. **Void Methods -- public static void main(String [ ]args)**
   b. **Value Returning Methods – Math.random, public static int max(int, int)**

return statements are required for value-returning methods

6. **Formal Parameters**

   parameters (formal placeholders)

   parameter list – type, order, number of parameters

   Method Signature: **int max(int, int);**
   i.e., method name + parameter list

   Method Header: **public static int max(int, int)**

   Remark:
   Each data type in the parameter list must be separately declared, i.e.,

   public static int max(int n1, int n2)
   not
   public static int max(int n1, n2)

7. **Alternative Names for Methods**
   a. **Value-Returning Method ←→ Function**
   b. **Void Method ←→ Procedure**

8. **Calling a Method -- Revisited**
   a. **Value-Returning Method ←→ Function**
   If the method returns a value, the call to the method is usually treated as a value, e.g.,

   ```
   int z = max(x, y);
   ```

   However, it is permissible to call value-returning method as a statement, if the return value is of no interest to the calling program.

   b. **Void Method ←→ Procedure**
   If the method returns a void, the call must be in the form of a statement, e.g.,

   ```
   System.out.println("This is the correct way");
   ```

9. Placement of Methods

If the method is to be used only in the current program, then place it in the same class, i.e., file, as the program, e.g.,

```
public class TestMax
{
  public static void main(String [ ] args)
  {
    int i = 5;
    int j = 2;
    int k = myMax(i, j);
    System.out.println("Max of  " + i + " and " + j + ": " +k);
  }
}
public static int myMax( int n1, int n2)
{
  int result;
  if( n1 > n2 ) result = n1;
  else result = n2;
  return result;
}
```

If the method is to be used in other programs, place it in another class, i.e., file, with a different descriptive name. For instance place the method,

```
public static int myMax( int n1, int n2)
{
    int result;
    if( n1 > n2 ) result = n1;
    else result = n2;
    return result;
}
```

in a class, i.e., file, MyMathStuff; the method call then be invoked by any other program in your account as

```
int z = MyMathStuff.myMax(x, y);
```

10. Call Stacks

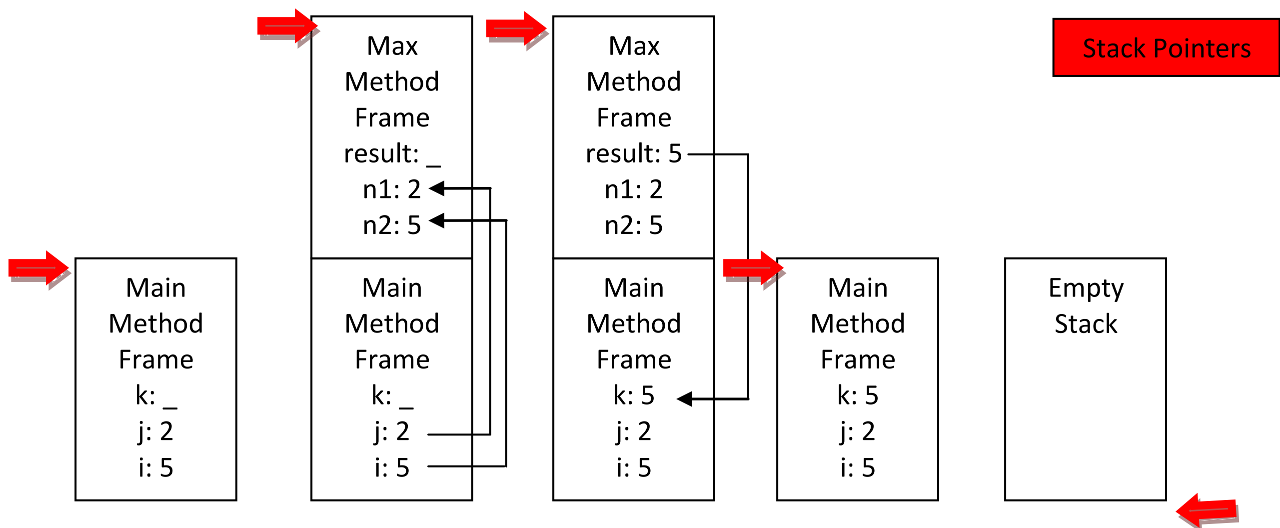
A stack is, simply, an area of memory in which information is stored and retrieved in a last-in, first-out, i.e., FIFO, manner.

The storage of dinner plates is normally accomplished by placing one plate on top of another. In this case, the set of plates are usually treated as a stack, i.e., a clean plate is placed on the top of the stack and when a plate is required for service, it is the plate on the top of the stack which is retrieved. It is the FIFO action and the storage structure which constitutes a stack.

When a method is called, the execution of the calling program must be suspended, and the called method is allowed to execute.

When any method is called, including the main method, the stack allocates an area at the top of the stack to store information relevant to that method. Thus as each executing method calls another method, the stack allocates additional area at the top of the stack for that method. The area allocated to each method is often referred to as a "frame" and common terminology is that a frame is "pushed" onto the stack.

When a method finishes executing, execution returns to the calling method and the area reserved for the completed method is deallocated, i.e., the frame is "popped off" the stack, or the stack is "popped".



11. Void Method

```
public class VoidTestMethod
{
   public static void main(String [ ] args)
   {
      double numberGrade;
       // input value for numberGrade
      System.out.print(“Number Grade: “ + numberGrade + “\tLetter Grade: “ );
      printGrade(numberGrade);
   }
}

public static void printGrade( double Score)
{
   if ((score < 0) || (score > 100))
   {
      System.out.println(“Invalid Score”);
      return;
   }
   if (score >= 90.0) System.out.println(‘A’);
   else if (score >= 80.0) System.out.println(‘B’);
   else if (score >= 70.0) System.out.println(‘C’);
   else if (score >= 60.0) System.out.println(‘D’);
   else System.out.println(‘F’);
}
```

12. Passing Parameters

a. Parameter Order Association – Pattern Matching

The arguments provided to a called method MUST be in the same order, be of a compatible type and, unless otherwise specified, be of the same number as that of the method's definition. Compatible type means passing without explicit casting!

Given the method definition

```
public static void nPrintMessage(String message, int n)
{
  for (int i = 0; i < n; i++) System.out.println(message);
}
```

the calling statement nPrintMessage(v, w);
is valid only if v is of type String and w is of type int.

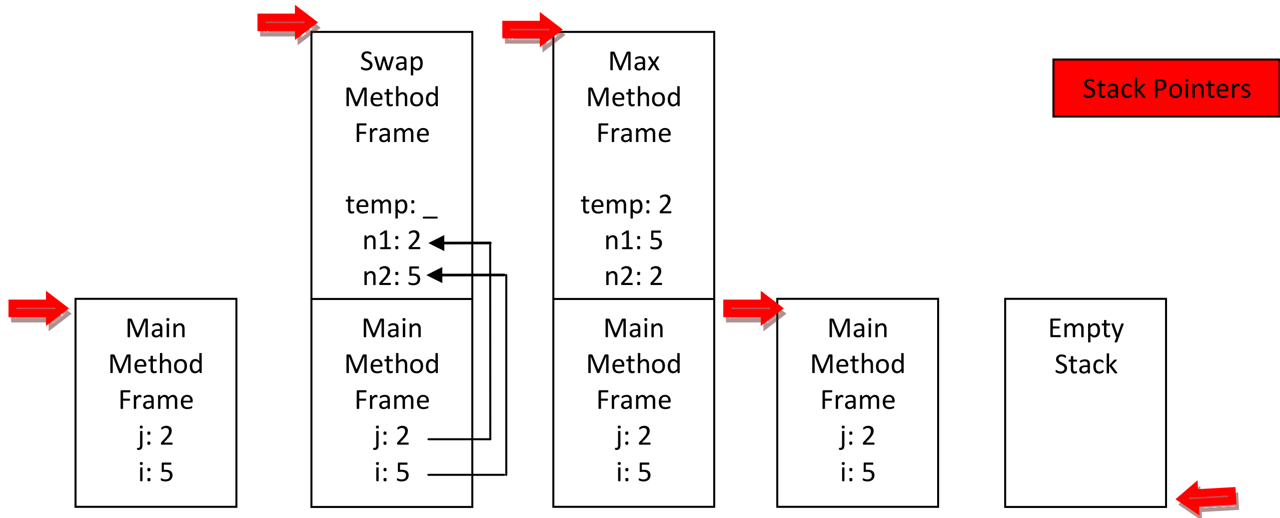
The statement nPrintMessage(15, "hello"); will fail!

b. Pass-by-Value

```
public class Increment
{
  public static void main(String [ ] args)
  {
     int x = 1;
     System.out.println(x);
     increment(x);
     System.out.println(x);
  }
     public static void increment(int n)
     {
       n++;
       System.out.println(n);
     }
}
```

Output: 1, 2, 1

```
public static void swap(n1, n2)
{
   int temp = n1;
   n1 = n2;
   n2 = temp;
}
```



13. Modular Code

a. **Reduce redundant code**
b. **Enable the reuse of code**
c. **Improve program quality**
d. **Enables the modification of the modular code independently from the various programs that use the modular code, e.g., Math.random( )**

```
public static int gcd(int n1, int n2)
{
   int gcd = 1;
   int k = 2;
   while( k <= n1 && k <= n2)
   {
     if( n1 % k == 0 && n2 % k == 0)
          gcd = k;
     k++;
   }
   return gcd;
}
```

```
public static boolean isPrime(int n)
{
   for( divisor = 2; divisor <= n/2;
     divisor++)
   {
     if( n % divisor == 0) return false;
   }
   return true;
}
```

14. Overloading Methods

```
public static int max( int n1, int n2)
{
  int result;
  if( n1 > n2 ) result = n1;
  else result = n2;
  return result;
}
```

```
public static double max( double n1, double n2)
{
  double result;
  if( n1 > n2 ) result = n1;
  else result = n2;
  return result;
}
```

```
public static long max( long n1, long n2)
{
  long result;
  if( n1 > n2 ) result = n1;
  else result = n2;
  return result;
}
```

```
public static float max( float n1, float n2)
{
  float result;
  if( n1 > n2 ) result = n1;
  else result = n2;
  return result;
}
```

```
public static char max( charn1, char n2)
{
  char result;
  if( n1 > n2 ) result = n1;
  else result = n2;
  return result;
}
```

```
public static double max( double n1, double n2, double n3)
{
  return max(max(n1, n2), n3);
}
```

---

double x = max(3, 3.5); invokes double max(double, double)
double x = max(3, 5); invokes int max(int, int)

15. **Ambiguous Overloads – compiler cannot determine which of two or more possible matches should be used for an invocation of a method.**

For example, given

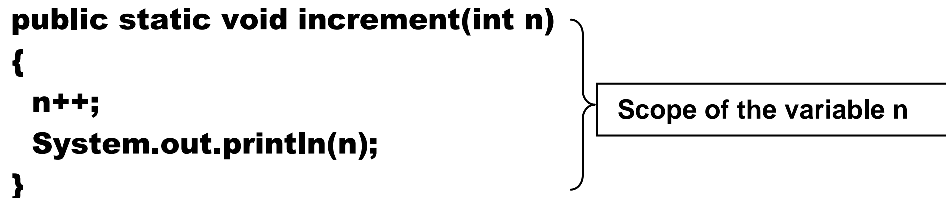
```
public static double max( int n1, double n2)
{
   double result;
   if( n1 > n2 ) result = n1;
   else result = n2;
   return result;
}
public static double max( double n1, int n2)
{
   double result;
   if( n1 > n2 ) result = n1;
   else result = n2;
   return result;
}
```

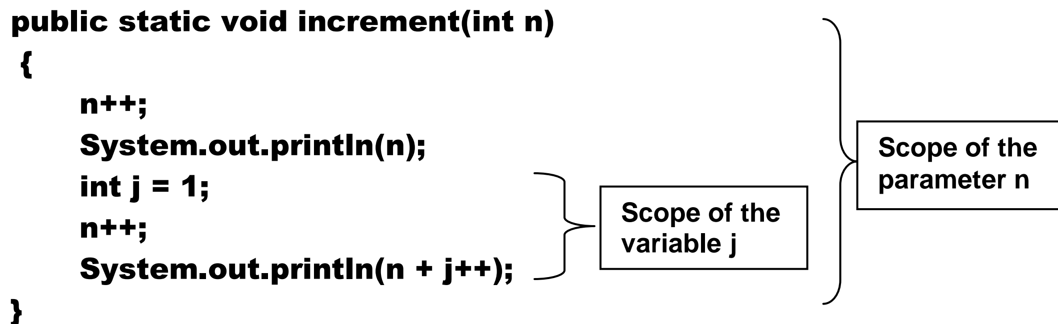
Overloaded methods must be defined in such a manner that such ambiguities cannot occur! Ambiguous overloads result in compiler errors!

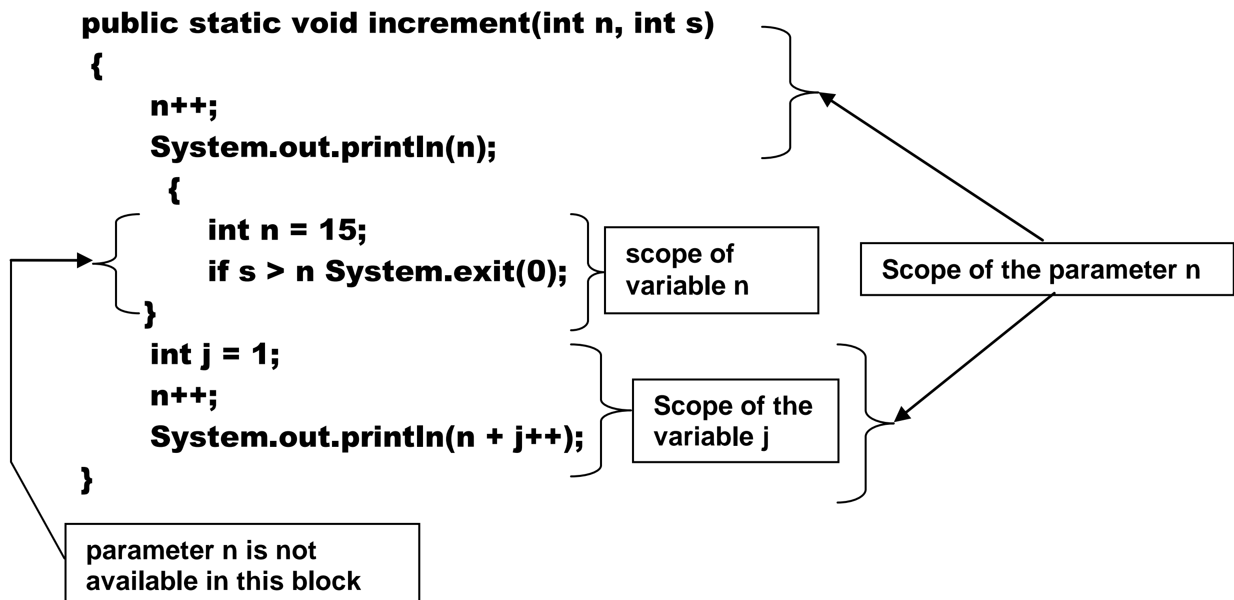
16. Scope of Variables

The scope of a particular variable is the part of the program where the variable can be referenced.

A variable defined inside of a method is referred to as a LOCAL variable. In the method below, n is a local variable with a scope limited by the enclosing block; n cannot be referenced outside of the block!







17. Math Class http://java.sun.com/javase/6/docs/api/index.html

- **does not contain a main method**
- **serves as a container class, i.e., holds a selection of methods**

a. **Math.PI → 3.14159**
b. **Math.E → 2.71828**
c. **Math.random( ) double $0.0 \le r < 1.0$**
d. **Trigonometric Methods**
   sin, cos, tan, etc,
e. **Exponent Methods**
   i. **double exp(double x) → e^x**
   ii. **double log(double x) → $\log_e(x)$**
   iii. **double log10(double x) → $\log_{10}(x)$**
   iv. **double pow(double a, b) → a^b**
   v. **double sqrt(double x) → $x^{1/2}$**

f. **Rounding Methods**
   i. **double ceil(double x) → round up to nearest integer**
   ii. **double floor(double x) → round down to nearest integer**
   iii. **double rint(double x) → round to nearest integer;**
   if equally close to both integers, the even integer is returned
   iv. **int round(float x) → (int)Math.floor(x + 0.5)**
   v. **long round (double x) → (long)Math.floor(x + 0.5)**

g. **Minimum, Maximum, Absolute**
   - **methods are overloaded for int, long, float, & double**

   i. **min(a, b)**
   ii. **max(a, b)**
   iii. **abs(a)**

18. Random Characters

a. **Unicode → 0 ≤ u ≤ 65535**

Math.random( ) → 0.0 ≤ r < 1.0 → [0.0, 1.0)

thus

0 ≤ (int)(Math.random( ) * (65535 + 1)) < 65536

hence

Unicode characters can be generated by (int)(Math.random( ) * (65535 + 1))

b. **Random Letters**

- **random integer between (int)'a' & (int)'z'**
  (int)((int)'a' = Math.random( ) * ((int) 'z' – (int)'a' + 1)
  or
  'a' + Math.random( ) * ('z' – 'a' + 1)

- **random lowercase letter**
  (char)('a' + Math.random( ) * ('z' – 'a' + 1))

- **random uppercase letter**
  (char)('A' + Math.random( ) * ('Z' – 'A' + 1))

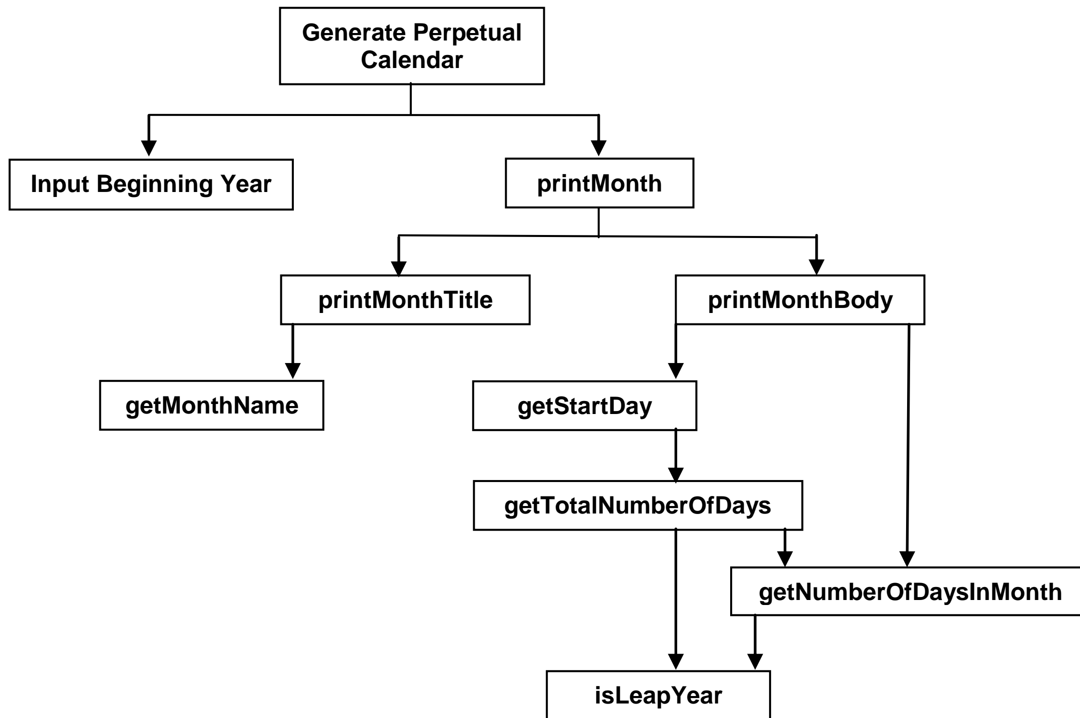
- **random character between ch1 & ch2 with ch1 < ch2**
  (char)(ch1 + Math.random( ) * (ch2 – ch1 + 1))

c. **Listing 5.8 & Listing 5.9 pages 158-159**

19. Method Abstraction & Stepwise Refinement

- **divide problem into smaller sub-problems**
- **stepwise refinement**
- **divide & conquer**

a. Top-Down Design



b. Top-Down Implementation

- **stubs, e.g., printMonthBody{ }**

c. Bottom-Up Implementation

i. **Implement <u>isLeapYear( )</u> method and create a test program for the <u>isLeapYear( )</u> method**

ii. **Implement <u>getNumberOfDaysInMonth( )</u> method and create a test program for the <u>getNumberOfDaysInMonth( )</u> method**

iii. **Etc.**

20. Planning versus Experimental Development

a. It is only viable to plan a project when you know what to expect in the nature of major aspects of the development

b. When you are unsure of your knowledge, engage in experimental development

> **Pottery Rule**
> **Make one, throw it away!**
> **Make another to keep.**

c. After you have made one, you are more qualified to use Top-Design to plan the project

d. Given a design problem that you don't know how to solve, resort to a concrete example. If you can solve a concrete example, you are better prepared to solve the more general, i.e., abstract, design problem. Go from the concrete to the abstract; if you try to start with the general, you may never finish.

e. Start your problem solution at a desk, not at the computer terminal.

f. For some types of problems, you might try to start your experimental development by using Data Tables to give a concrete problem to solve.

g. Data Tables combined with a detailed Analysis will often lead to the correct Algorithms to solve the general design problem.

h. Before you announce a public solution, however, you must do a test implementation of the algorithms. Make sure to test the algorithms with a sufficiently large Data Set so as to ensure that the algorithms are, indeed, correct.