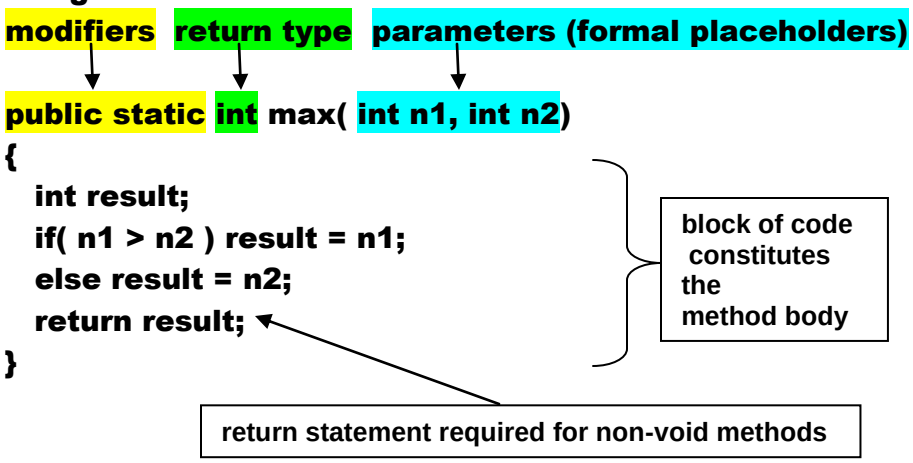
Lecture #6-7
Methods

1. **Method –**
   a. **group of statements 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 & y;
   int z = max(x, y); ← values of the actual parameters, i.e., arguments
   output the z value;
}
```

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)**
      Remark: return statement is required for value-returning method

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 = max(i, j);
    System.out.println("Max of  " + i + " and " + j + ": " +k);
  }
}
public static int max( 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 void main( String [ ]args )
{
      int x, y;
      input values for x & y;
      int z = max(x, y);
      output the z value;
}
```

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

```
int z = MyMathStuff.max(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".

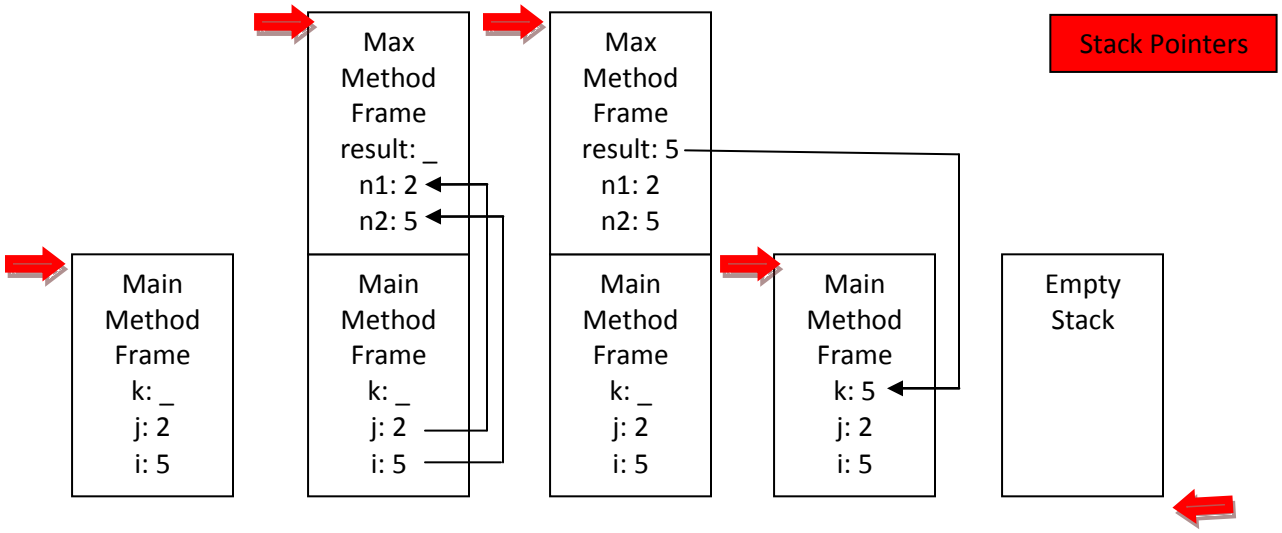


Diagram Liang page 145

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);
}
```

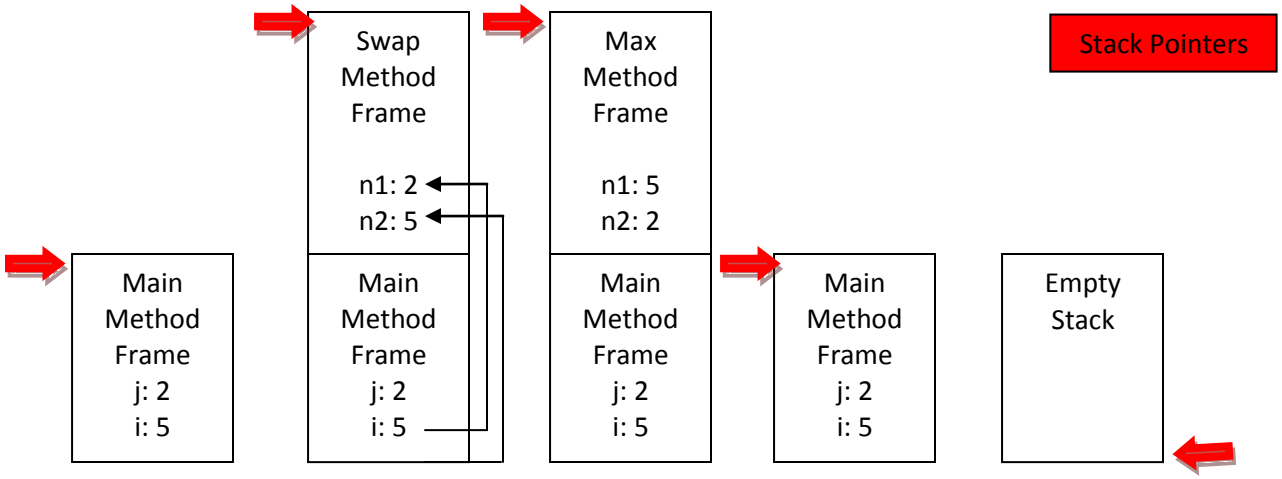
the calling statement `nPrintMessage(v, w);`
is valid only if v is of type String and w is of type int.
The statement `nPrintMessage(15, "hello");` will fail!

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

```
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**

```
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 should be defined in such a manner that such ambiguities cannot occur! Ambiguous overloads result in compiler errors!

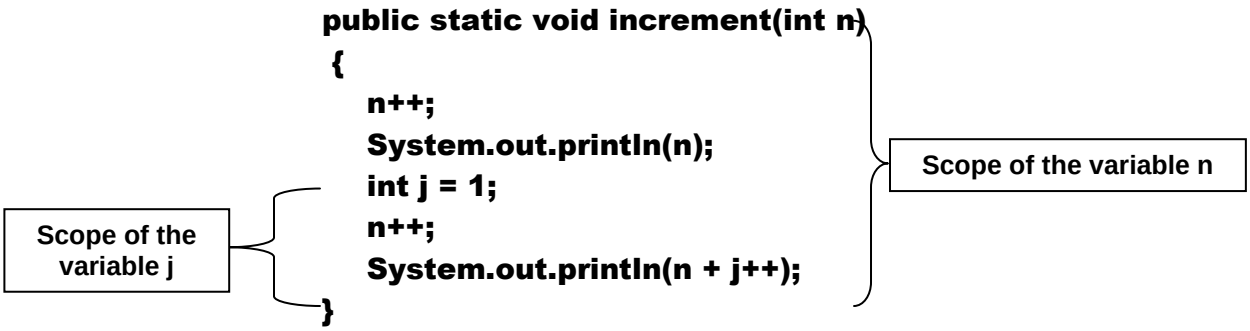
16. **Scope of Variables**

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!

```
public static void increment(int n)
{
  n++;
  System.out.println(n);
}
```

Scope of the variable n



17. **Math Class** http://java.sun.com/javase/6/docs/api/index.html
    a. **Math.PI**
    b. **Math.E**

       see Liang pages 154-157 for complete list

    c. **public static double ceil(double x) // rounded up to nearest neighbor**

       // ceil(4.1) → 5.0

    d. **public static double floor(double x) // rounded down to nearest neighbor**

       // floor(4.9) → 4.0

    e. **public static double rint(double x) // rounded to nearest neighbor,**

       if equally close, rounded to the even number, e.g.,

       // rint(4.5) → 4.0

       // rint(5.5) → 6.0

    f. **public static int round(float x)**

       // returns (int)Math.floor(x + 0.5)

       // (int)Math.floor(4.4 + 0.5) → (int)Math.floor(4.9) → 4

       // (int)Math.floor(4.5 + 0.5) → (int)Math.floor(5.0) → 5

    g. **public static long round(double x)**

       // returns (long)Math.floor(x + 0.5)

       // (long)Math.floor(4.4 + 0.5) → (long)Math.floor(4.9) → 4

       // (long)Math.floor(4.5 + 0.5) → (long)Math.floor(5.0) → 5

18. Random Numbers

a. 0.0 ≤ Math.random() < 1.0
b. 0.0 ≤ Math.random() * 10 < 10.0
c. 0.0 ≤ Math.random() * 100 < 100.0
d. 15.0 ≤ 15 + (Math.random() * 100) < 115.0
e. 0.0 ≤ Math.random() * (65535 +1) < 65536, i.e.,
   generates all numbers in the range [0000_{16} , $FFFF_{16}$] which encompasses the entire Unicode system.
f. (char)('a' + Math.random() * ('z' – 'a' + 1))
   generates all lower case characters
g. (char)('A' + Math.random() * ('Z' – 'A' + 1))
   generates all upper case characters
h. (char)(ch1 + Math.random() * (ch2 – ch1 + 1)) where ch1 < ch2
   generates characters between ch1 & ch2

19. Method Abstraction & Stepwise Refinement

Separate the Usage of a Method from its Implementation, i.e., the <u>implementation details are encapsulated in the method</u> and hidden from the user; the user does not need to know the implementation details to use the method.

Method Header – public -- users
Method Body – private – implementers only!!

E.g.,

```
System.out.print();
System.out.println();
Math.max();
JOptionPane.showInputDialog();
```