ES 1036 Programming Fundamentals

Unit 4: Modular Programming with Methods/Functions

*"There are no secrets to success. It is the result of preparation, **hard work**, and learning from failure" ~Colin Powell*

Dr. Quazi M. Rahman, Ph.D, P.Eng., SMIEEE
Office Location: TEB 361
Email: QRAHMAN@eng.uwo.ca
Phone: 519-661-2111 x81399

*"Genius is 1% talent and 99% percent **hard work**.." ~ Albert Einstein*

Outline

- **Java methods/functions**
  - **Method/Function characteristics/Definition/Call**
  - **Overloading functions/methods**
- **Scope of Objects/variables**
  - **Background info**
  - **Local Scope with examples**
  - **Demo on call stack**
- **FYI: Programming design/implementation revisited**

The Intro.: An example of a predefined function/method

```
public static void main(String[] args){
   double anyNumber = 4, y = 0;
   /*Note: in the following statement, the function sqrt has
   been called with an argument anyNumber. This function
   returns a value to the variable y.*/
   y = Math.sqrt(anyNumber);
   /*In the following statement pow method is called with
   two arguments in the println statement and the result is
   returned on the screen*/
   System.out.println(Math.pow(y,3));
   /*Question 1: Where have we defined these
   functions? Both sqrt and pow methods/functions are
   predefined methods which have been defined in the
   Math class (java.lang.Math)
   Question 2: How did we define these Methods?
         This is our learning goal in this section*/
}
```


The Intro: What is method, and types of Method/function

- **A Method (also, know as function) is a collection of statements that are grouped together to perform an operation.**

Function/Method Types:

- **Pre-defined**
  - **Stored in the standard Packages (example: java.lang.Math)**
  - **Functions are already defined**
  - **User needs to know how to use/call it**
    - **Modifier(s), Function name, return type, number & type of parameters**
  - **E.g.** `public static double sqrt(double number)`
- **Programmer defined (also known as: User defined)**
  - **programmer writes the entire function definition (modifier(s), name, return type, number & type of parameters and body)**

Example on Pre-defined Methods

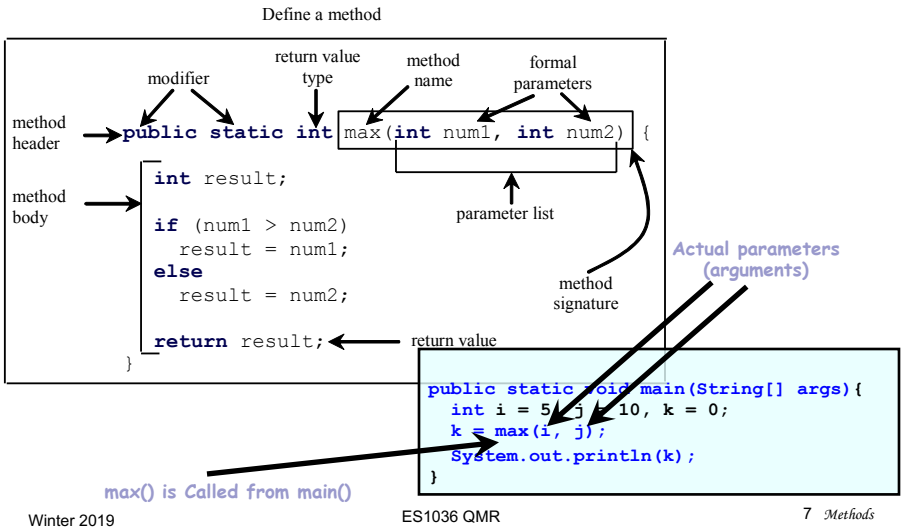
`Math.random()`	Returns a double type random number which is greater than or equal to 0.0 and less than 1.
`Math.abs(x)`	Returns the absolute value of x
`Math.sqrt(x)`	Returns the square root of x, where x >=0
`Math.pow(x,y)`	Returns x^y
`Math.ceil(x)`	Returns the nearest integer larger than x
`Math.floor(x)`	Returns the nearest integer smaller than x
`Math.exp(x)`	Returns e^x
`Math.log(x)`	Returns ln x, where x >0
`Math.log10(x)`	Returns $\log_{10}x$, where x>0
`Math.max(x,y)`	Returns the greater of the two arguments
`Math.min(x,y)`	Returns the smaller of the two arguments
`Math.floorMod(x,y)`	Returns the floor modulus of the arguments

Introducing Function: User defined method

- A user-defined (programmer-defined) method is defined outside the main() method (either above main() or below main()).
- It follows certain grammatical (Java) rules discussed later
- In any program, the 'first' user-defined method-call is made from the main() method.
  - Method call can also be made from any other called method (see the example code later).

Defining Methods

- Below, two methods named max() and main() have been defined as separate entities inside the same class, and max() method has been called/invoked from the main(). max() returns the greater of the two values.



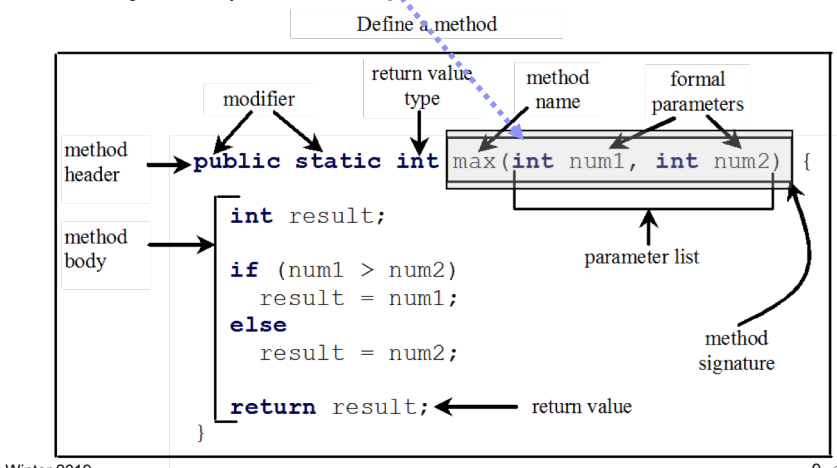
Where do we define Methods

- Every method is defined as a member of a class, and a method has to be defined as a separate entity (i.e., a method can NOT be defined inside another method (there are exceptions though!).)

```
public class MyClass {
        //Method A Definition
        //main() Method B Definition
        public static int max(int num1, int num2) {
        int result;
        if (num1 > num2)
                result = num1;
        else
                result = num2;
        return result;
        }
        //max() Method C Definition
        public static void main(String[] args){
        int i = 5, j = 10, k = 0;
        k = max(i, j);
        System.out.println(k);
        }
        //Method D Definition etc.
}
```

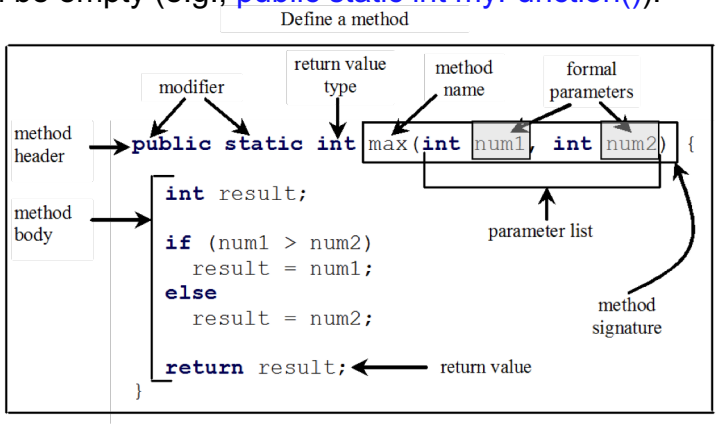
Terminology of Functions: Method Signature

- **Method signature** is the combination of the method name (a valid identifier which can not be a keyword) and the parameter list.



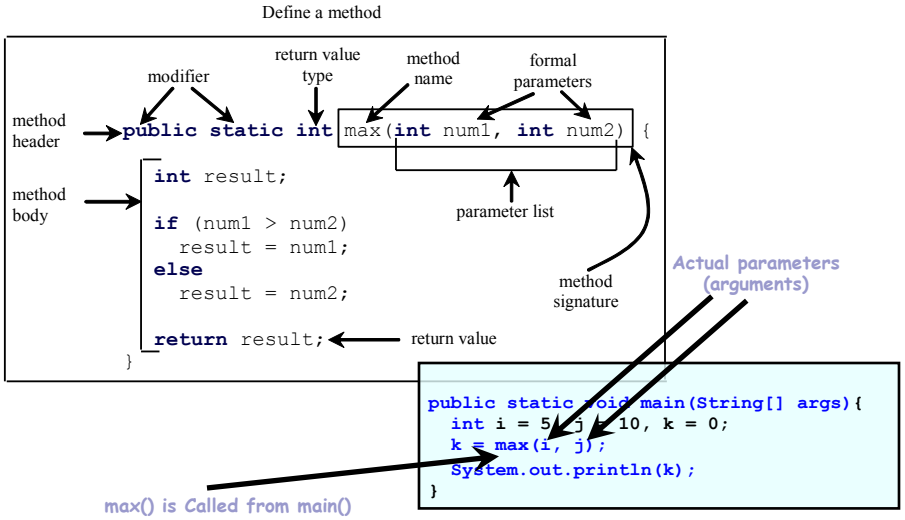
Terminology of Functions: Formal Parameters

- The variables declared/defined in the method header are known as **formal parameters**. This is a comma-separated list.
- Based on the code-requirement, the formal-parameter list can be empty (e.g., public static int myFunction()).



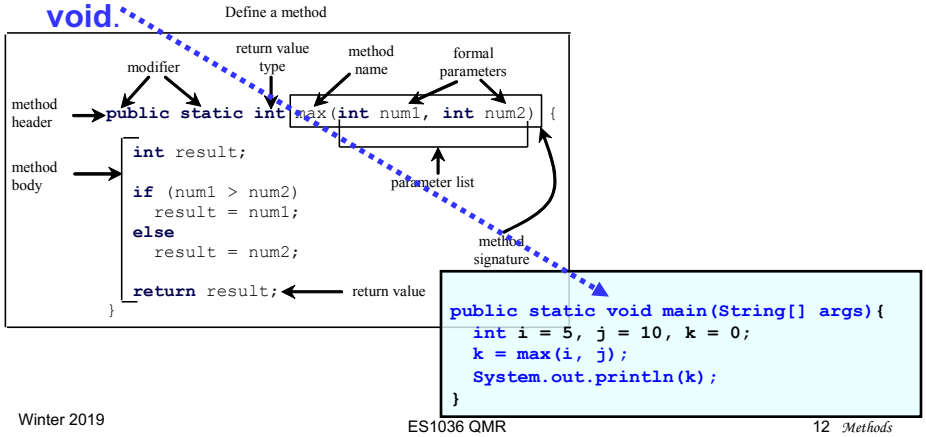
Terminology of Functions: Actual Parameters

- When a method is invoked/called, values are passed to the formal parameters based on the data-type and order. These values are **actual parameters or arguments of the method.**



Terminology of Functions: Return Value Type

- A method may return a value. The **returnValueType** is the data type of the value the method returns.
- Any method can NOT return more than one object.
- If the method does not return a value, the **returnValueType** is the keyword **void**.
- For example, the **returnValueType** in the **main** method is **void**.

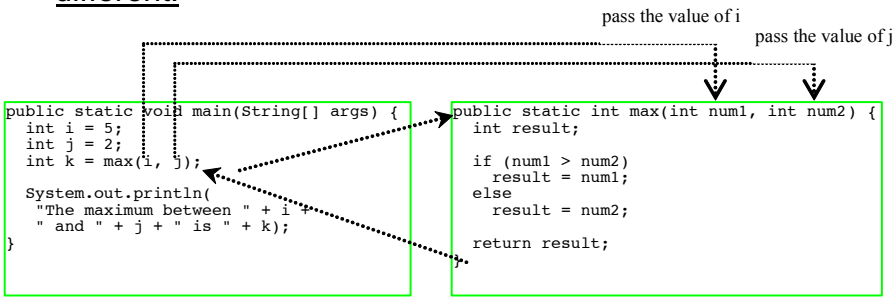


Terminology of Functions: Calling a method/Function

- When a function is called (invoked), the arguments must agree with the formal parameters in order, number and data type, but identifiers (variable names) in the function header and in the arguments in the function-call can be different (see the following slide).
- A function can be called by value or by reference.
- Pass by value
  - Formal parameter receives a copy of the argument (example on the next slide)
  - Changes to the formal parameter content do not affect the original object value that was passed via argument
- Pass by reference
  - Changing formal parameter content will change the original object value that was passed via argument!

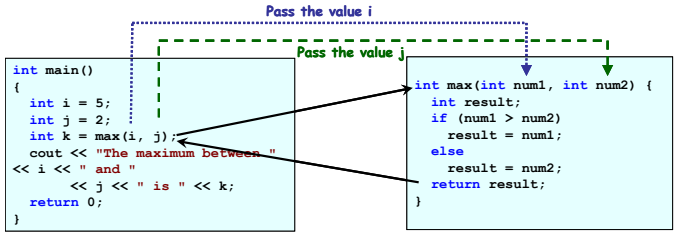
Example on Calling Methods

- The following is a 'Pass by value' example, where the max() method has been called from the main() method.
- Note: The identifiers (variable names) in the function parameter list (num1 and num2, below) and in the arguments (i and j, below) in the function-call can be different.



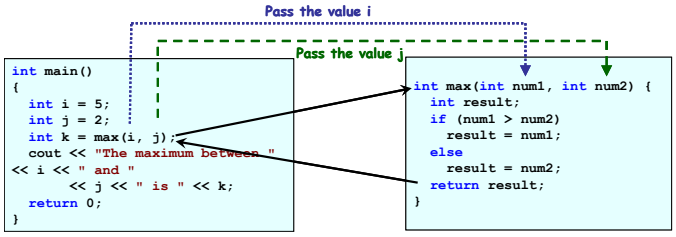
Using a Method/Function

- "Call" (or "Invoke") a function when you want to use it
- When you "call" (or "invoke") a function, flow of control is transferred to the header of the function body.
- Values of parameters are passed along
- Formal parameters must agree with arguments in order, number and data type, but parameter names can be different
- If you call a function from a println-statement, the function-call will have the higher precedence than the println-statement.



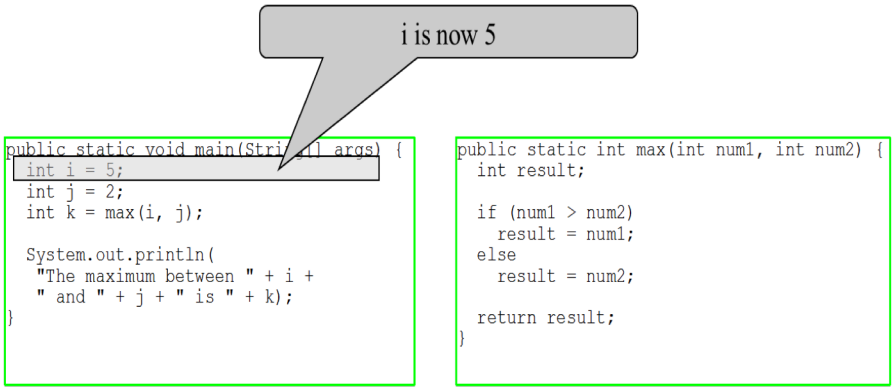
Using a Method/Function, *cont.*

- At the end of body (or with an explicit `return` statement) flow of control is transferred back to the calling function/method.
- A function can only return a single object to the calling method, if there is any return-value type in the function header.
  - The function header declares the type of object to be returned
  - A return statement is required in the body of the function

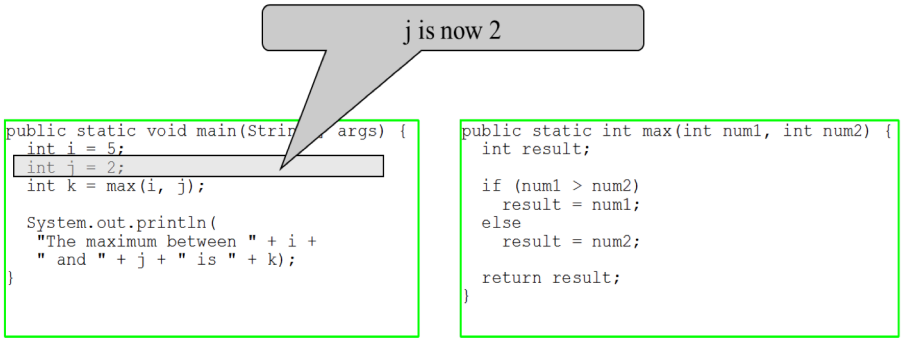


Tracing Method invocation/call (1 of 10)

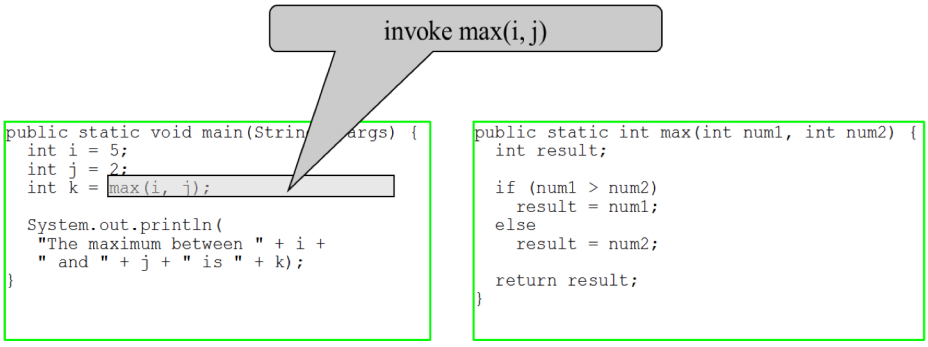
Draw the memory diagram as you trace the code



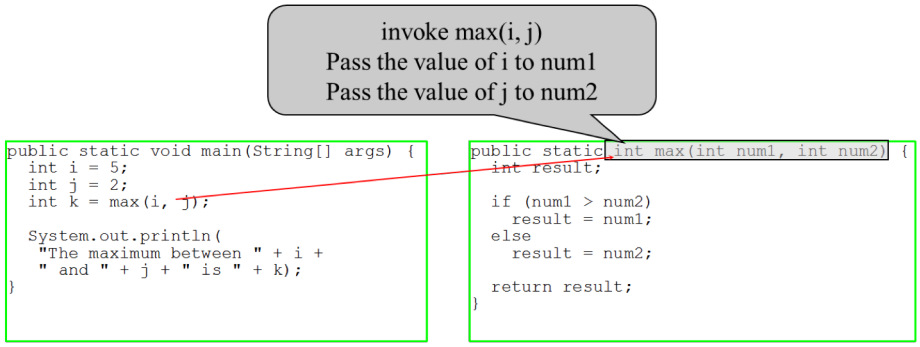
Tracing Method invocation/call (2 of 10)



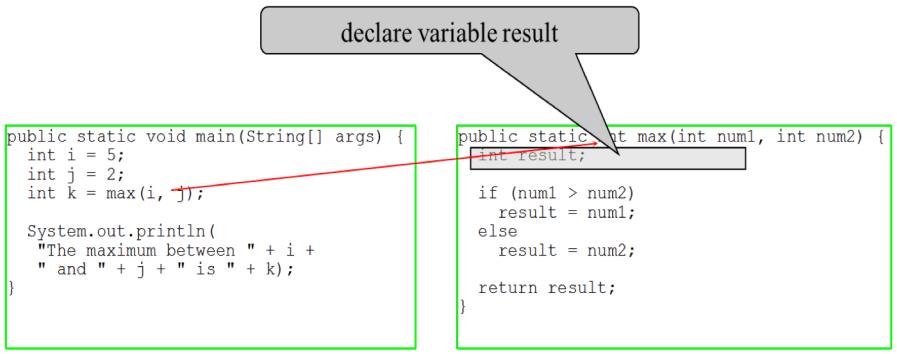
Tracing Method invocation/call (3 of 10)



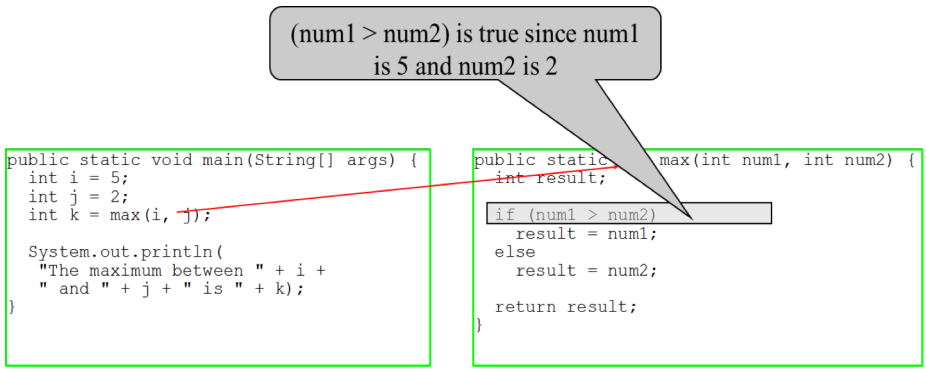
Tracing Method invocation/call (4 of 10)



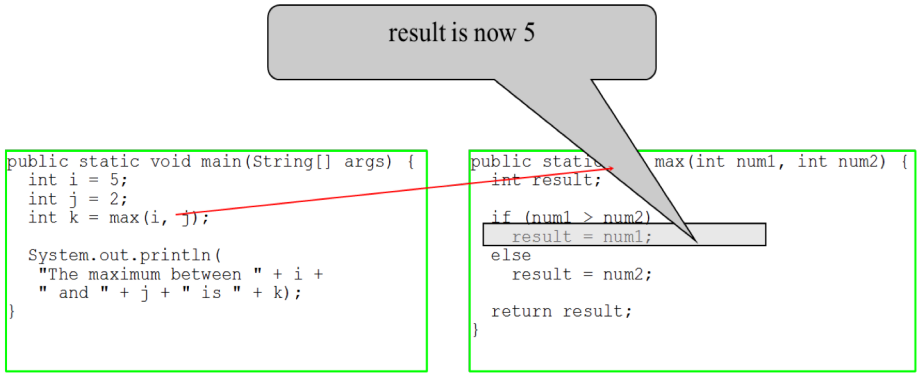
Tracing Method invocation/call (5 of 10)



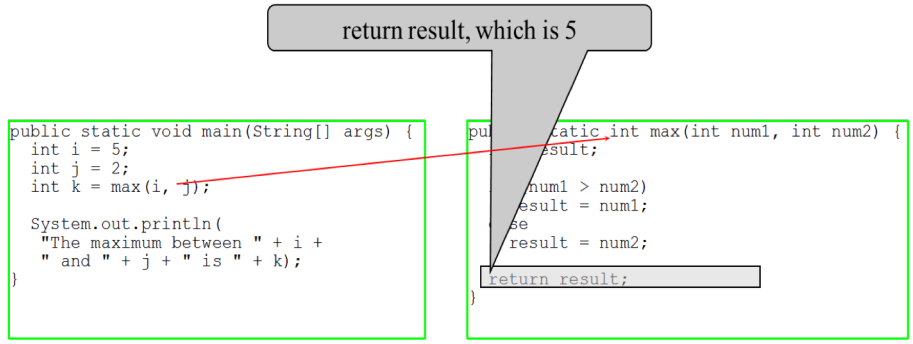
Tracing Method invocation/call (6 of 10)



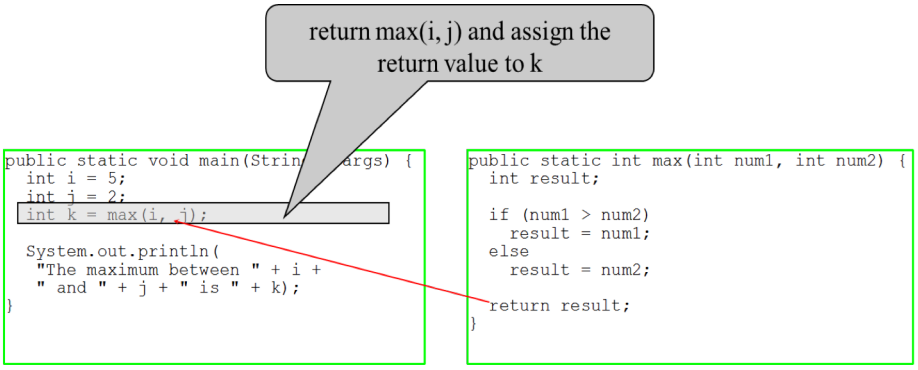
Tracing Method invocation/call (7 of 10)



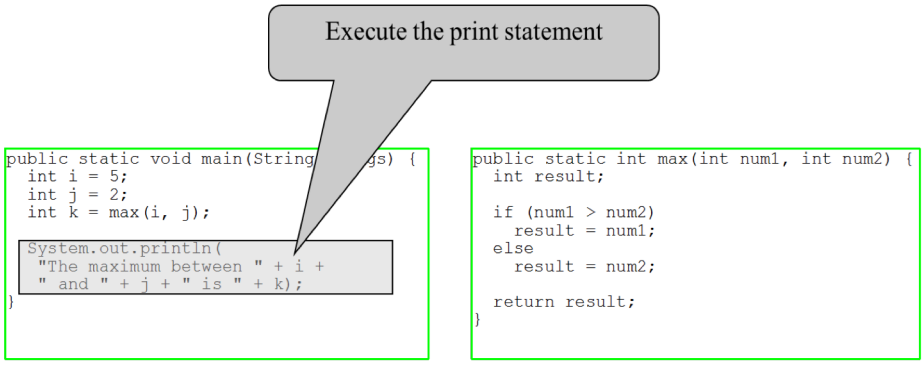
Tracing Method invocation/call (8 of 10)



Tracing Method invocation/call (9 of 10)



Tracing Method invocation/call (10 of 10)



Watch out!

- A **return** statement is required for a value-returning method.
- The method shown below in (a) is logically correct, but it has a compilation error because the Java compiler thinks, it possible that this method does not return any value.

```
public static int sign(int n) {
  if (n > 0)
    return 1;
  else if (n == 0)
    return 0;
  else if (n < 0)
    return -1;
}
```

(a)

Should be

```
public static int sign(int n) {
  if (n > 0)
    return 1;
  else if (n == 0)
    return 0;
  else
    return -1;
}
```

(b)

- To fix this problem, delete if (n < 0) in (a), so that the compiler will see a return statement to be reached regardless of how the if-statement is evaluated.

Why Functions/Methods?

- Complex program can be broken down into sub tasks, each of which is easy to manage and implement. This is called Modular programming approach
- The advantages of using functions, instead of writing the entire solution in main:
  - Multiple programmers can be involved
  - Testing/Debugging/Maintenance becomes easy
  - Code duplication is reduced
  - Information hiding: Hide the implementation from the user.

void Method/Function

- A method/function that does not return a value to the calling program is known as `void` method/function (Return type is void).
- Void method/function examples:
  - public static void main(String[] xyz);
  - public static void myFunction(int a, double b);
  - public static void anyFunction();
- In the function definition of a void function, `return` statement is optional
- If a return statement is used in the void function, it has the following form
  - `return;`

Review

56) The parameter list contains all objects (variables) used by the method/function?

a) True
b) False

Review

57) Which of the following is a valid definition for a method/function called "cube" ?

```
a) public static function cube(double x){ ...
   }
b) public static double cube(double x){ ... }
c) public static double cube(x){ ... }
d) public static cube(double x){ ... }
```


Review

58) Suppose your method does not return any value, which of the following keywords can be used as a return value-type?

a) **float**
b) **byte**
c) **int**
d) **double**
e) **void**
f) **return**

Review

59) **The signature of a method consists of _____.**

a) **parameter list**
b) **function name**
c) **function name and parameter list**
d) **return type, function name, and parameter list**

Review

60) **When you invoke a method with a parameter, the value in the argument is passed to the parameter. This is referred to as _____.**

a) **method invocation**
b) **pass by name**
c) **pass by reference**
d) **pass by value**

Review

61) What will be the output of the method-call `nPrint('a', 4)`?

```
public static void nPrint(char ch, int n)
{
  while (n > 0)
  {
    System.out.print(ch);
    n--;
  }
}
```

a) invalid call
b) aaa
c) aaaaa
d) aaaa

Review

62) Breaking a program in to a manageable sized functions/methods is called

a) modular programming
b) break up programming
c) high-level programming
d) functional programming
e) None of the above

Review

63) When a function is called, flow of control moves to the function's header.

a) True
b) False

Write a program to read 3 different integers and display 3 lines of stars '*', each representing the corresponding values of these integers.

The solution in the main()

```
public static void main(String[] args) {
   int a , b , c; Scanner input = new Scanner(System.in);
   System.out.println("Enter a: "); a = input.nextInt();
   System.out.println("Enter b: "); b = input.nextInt();
   System.out.println("Enter c: "); c = input.nextInt();

   //Display the first line
   for (int i=0; i<a; i++)
       System.out.print("*") ;
    System.out.println() ;
   //Display the second line
   for (int i=0; i<b; i++)
        System.out.print("*") ;
    System.out.println() ;
   //Display the third line
   for (int i=0; i<c; i++)
        System.out.print("*") ;
    System.out.println() ;
    input.close();
}
```


Solution with programmer defined method: code duplication is omitted

```
public static void print_stars (int length){
   for (int i=0; i<length; i++)
        System.out.print("*") ;
   System.out.println() ;
}
public static void main(String[] args) {
   int a , b , c;
   Scanner input = new Scanner(System.in);
   System.out.println("Enter a: "); a = input.nextInt();
   System.out.println("Enter b: "); b = input.nextInt();
   System.out.println("Enter c: "); c = input.nextInt();
   //Display the first line
   print_stars (a);
   //Display the second line
   print_stars (b);
   //Display the third line
   print_stars (c);
   input.close();
}
```

Another Example with void function

```
//output formatted date
public static void print_date(int mo, int day, int year)
{
    string month;
    switch(mo)
    {
       case 1:  month = "January"; break;
       case 2:  month = "February"; break;
        …
       case 12: month = "December"; break;
       default: month = "January";
     }
    System.out.println(month +' '+day+','+year);
} //end print date
```

```
public static void main(String[] args) {
   /*call the function with today's date*/
}
```

Homework

Demo on modular programming and code-repetition elimination (1 of 4)

- Problem: Write a Java program to find the sum of integers from 1 to 10, from 20 to 30, and from 35 to 45, respectively.

Demo on modular programming and code-repetition elimination (2 of 4)

Solution inside main()

```
int sum = 0;
for (int i = 1; i <= 10; i++)
  sum += i;
System.out.println("Sum from 1 to 10 is " + sum);

sum = 0;
for (int i = 20; i <= 30; i++)
  sum += i;
System.out.println("Sum from 20 to 30 is " + sum);

sum = 0;
for (int i = 35; i <= 45; i++)
  sum += i;
System.out.println("Sum from 35 to 45 is " + sum);
```


Demo on modular programming and code-repetition elimination (3 of 4)

```
int sum = 0;
for (int i = 1; i <= 10; i++)
  sum += i;
System.out.println("Sum from 1 to 10 is " + sum);

sum = 0;
for (int i = 20; i <= 30; i++)
  sum += i;
System.out.println("Sum from 20 to 30 is " + sum);

sum = 0;
for (int i = 35; i <= 45; i++)
  sum += i;
System.out.println("Sum from 35 to 45 is " + sum);
```

Watch for code–repetitions!

Demo on modular programming and code-repetition elimination (4 of 4)

- *Note: the repetitive part is put in a separate method/function called sum(), and it is accessed 3 times from the main() method*

```
public static int sum(int i1, int i2) {
  int sum = 0;
  for (int i = i1; i <= i2; i++)
    sum += i;
  return sum;
}

public static void main(String[] args) {
  System.out.println("Sum from 1 to 10 is " + sum(1, 10));
  System.out.println("Sum from 20 to 30 is " + sum(20, 30));
  System.out.println("Sum from 35 to 45 is " + sum(35, 45));
}
```


Example: Write a program to add n different numbers entered from the keyboard (1 of 5)

Solution A: Written in main() method only

```
public static void main(String[] args)
{
   int n;
   double sum =0,num = 0; Scanner input = new Scanner(System.in);
   System.out.print("How many numbers do you want to add? ");
   n = input.nextInt();
   for(int i=1; i<=n; i++)
   {
      System.out.print("Input number " + i +": ");
      num = input.nextDouble();
      sum=sum+num;
   }
   System.out.println("The result is " + sum);
   input.close();
}
```

Example: Write a program to add n different numbers entered from the keyboard (2 of 5)

Solution: Here we are revisiting the solution with a second method public static double addNnumbers(int n_numbers) and calling it from the main().

```
public static void main(String[] args){
   int n; Scanner input = new Scanner(System.in);
   System.out.print("How many numbers do you want to add? ");
    n = input.nextInt();
    System.out.println("The result is " + addNnumbers(n));
   input.close();//check out the function call above
}
```

```
public static double addNnumbers(int n_numbers){
   double sum = 0, num = 0;
   Scanner input = new Scanner(System.in);
   for(int i=1; i<=n_numbers; i++){
        System.out.print("Input number " + i +": ");
        num = input.nextDouble();
        sum = sum + num;
   }
    input.close();
    return sum;
}
```


Example: Write a program to add n different numbers entered from the keyboard (3 of 5)

Solution: Here we are revisiting the solution with a new second method public static void addNnumbers(int n_numbers) and calling it from the main().

```
public static void main(String[] args){
   int n; Scanner input = new Scanner(System.in);
   System.out.print("How many numbers do you want to add? ");
   n = input.nextInt();
   addNnumbers(n); /*check out the function call*/
   input.close();
}
```

```
public static void addNnumbers(int n_numbers){
   double sum = 0, num = 0;
   Scanner input = new Scanner(System.in);
   for(int i=1; i<=n_numbers; i++){
        System.out.print("Input number " + i +": ");
        num = input.nextDouble();
        sum = sum + num;
   }
   System.out.println("The result is " + sum);
   input.close();
}
```

Example: Write a program to add n different numbers entered from the keyboard (4 of 5)

Solution: Here we are revisiting the solution with a different second method public static double addNnumbers() and calling it from the main().

```
public static void main(String[] args){
   System.out.println("The result is " + addNnumbers());
   //check out the function call above
}
```

```
public static double addNnumbers(){
   int n; Scanner input = new Scanner(System.in);
   System.out.print("How many numbers do you want to add? ");
   n = input.nextInt();
   double sum = 0, num = 0;
   for(int i=1; i<=n; i++){
        System.out.print("Input number " + i +": ");
        num = input.nextDouble();
        sum = sum + num;
   }
    input.close();
    return sum;
}
```


Example: Write a program to add n different numbers entered from the keyboard (5 of 5)

Solution: Here we are revisiting the solution with a different second method public static void addNnumbers() and calling it from the main().

```
public static void main(String[] args){
   addNnumbers();
   //check out the function call above
}
```

```
public static void addNnumbers(){
   int n; Scanner input = new Scanner(System.in);
   System.out.print("How many numbers do you want to add? ");
   n = input.nextInt();
   double sum = 0, num = 0;
   for(int i=1; i<=n; i++){
        System.out.print("Input number " + i +": ");
        num = input.nextDouble();
        sum = sum + num;
   }
   System.out.println("The result is " + sum);
   input.close();
}
```

Example: Calling a method from inside another method

```
/*Here the method multiplyBy2 is called from inside the
  method add3Numbers*/
public static double add3Numbers(){
   double sum=0, n; Scanner input= new Scanner(System.in);
   for(int i=1; i<=3; i++){
       System.out.println("Enter the next number: ";
       n = input.nextInt();
       sum=sum+multiplyBy2(Math.sqrt(n));
   }
   input.close();
   return sum;
}
public static double multiplyBy2(double a){
   return (2*a);
}
public static void main(String[] args){
   double x = add3Numbers();
   System.out.println("The result is: "+x);
}
```


Example: This program demonstrates that passing value does not change the content of the original variables

```
//Draw the memory diagram to find out the output
//Main method
public static void main(String[] args) {
     int a = 5, b = 6;
     swapValues(a,b);
     System.out.println(a+" and  "+b);
}
public static void swapValues(int x, int y){
     int temp; temp=x; x=y; y=temp;
     System.out.println(x+" and  "+y);
}
```

Example: Generating a random character

- As shown in Unit 2 (Slide #114), any character has a coded (Unicode and ASCII code; ASCII code is a subset of Unicode) value, which when casted into an integer number, can use all numeric operators.
- So, to generate a random character between any two characters ch1 and ch2 with ch2 > ch1 can be generated by following the guideline shown on slide #29 (unit 3) as follows:

  (char)(ch1+Math.random(range+1));

  where, range = (ch2-ch1);

Example: Generating a random character (Solution)

```
public static void main(String[] args) {
  System.out.println(getRandomCharacter('A',  'M'));
  System.out.println(getRandomLowerCaseLetter());
  System.out.println(getRandomDigitCharacter());
  System.out.println(getRandomCharacter());
}
/** Generate a random character between ch1 and ch2 */
  public static char getRandomCharacter(char ch1, char ch2) {
    return (char)(ch1 + Math.random() * (ch2 - ch1 + 1));
  }
  /** Generate a random lowercase letter */
  public static char getRandomLowerCaseLetter() {
    return getRandomCharacter('a', 'z');
  }
  /** Generate a random uppercase letter */
  public static char getRandomUpperCaseLetter() {
    return getRandomCharacter('A', 'Z');
  }
  /** Generate a random digit character */
  public static char getRandomDigitCharacter() {
    return getRandomCharacter('0', '9');
  }
  /** Generate a random character */
  public static char getRandomCharacter() {
    return getRandomCharacter('\u0000', '\uFFFF');
  }
```

Exercise Problems

- Rewrite the code given on slide #149/150 (Unit 3) on finding gcd by defining the method public static int gcd(int n1, int n2), and call this method from the main() method.
- Rewrite the solution for the problem given on slide #157 with the help of slide #155/156 (Unit 3) on finding first 50 prime numbers and printing them on the screen by defining the methods public static boolean isPrime(int number) and public static void printPrimeNumbers(int numberOfPrimes). The method isPrime() will check and decide whether a number is prime or not, while the method printPrimeNumbers() will print the numberOfPrimes passed to this function.

Practice Example: Factorial of n (n!)

- Definition of Factorial of n
  - n! = n*(n-1)*(n-2)*…*1=1*2*3*…..*n (n>0)
  - n is a positive integer
  - 0! is 1 by definition
- Lets define a function/method in java that
  - Takes one integer parameter , say n
  - Calculate n!, say nFact
  - Returns nFact

Method for Factorial n (n!)

```
double factorial (int n)
{
    double nFact = 1;
    while (n > 1)
    {
       nFact = nFact * n;
       n--;
    }
    return (nFact);
}
```

Why should the result be placed in a double type variable?

Example Problem for homework

- Given a number n, display a nxn square as follows

```
********
.*******
..******
...*****
....****
.....***
......**
.......*
```

Solution with for-loops

```
public static void main(String[] args)
{
   int n, i, j, k;
   Scanner input = new Scanner(System.in);
   System.out.print("Input n: ");
   n = input.nextInt();
   System.out.println();
   for(i=1; i<=n; i++)
   {
      for(j=1; j<i; j++)
        System.out.print(".");
      for(k=j; k<=n; k++)
        System.out.print("*");
      System.out.println();
   }
   System.out.println();
   input.close();
}
```


Solution with function/Method

```
public static void main(String[] args) {
   int n;
   Scanner input = new Scanner(System.in);
   System.out.print("Input n: ");
   n = input.nextInt();
   for (int i=0 ; i < n ; i++)
   {
       PrintChar (i,'.');
       PrintChar (n-i, '*');
       System.out.println();
   }
   input.close();
}
public static void PrintChar(int m, char c)
{
   // Display the value of c m times
   for (int i=0 ; i<m; i++)
   {
       System.out.print(c);
   }
}
```

```
GetInput (the value of n)
Loop with i, n times
{
   PrintChars(i,'.');
   PrintChars( n-i, '*');
   Print newline
}
```

```
PrintChar(int m, char c)
{
   loop n time
  {
     print char;
  }
}
```

Method/Function overloading

- If two or more function signatures have the same name but different parameter lists, these functions are called overloaded functions, and this concept is known as function overloading.
- In this case, the return-data type can be same or different in the function header.
- Example:

```
public static void myFunction(int a){..}
public static int myFunction(int a, double b){..}
public static double myFunction(String s){..}
public static void myFunction(void){..}
```


Method/Function overloading example: What will be output?

```
//Function/Method definitions
public static int myFunction(int x, double y){
        return x%(int)y;
}
public static double myFunction(char k){
        System.out.print(k); return 10;
}
public static void myFunction(){
        System.out.println("Do nothing\n");
}
public static void myFunction(int x) {
        System.out.println(x);
}
public static void main(String[] args){//Method/Function calls
        myFunction(5);
        System.out.println(myFunction(4, 5.5));
        System.out.println(myFunction('c'));
        myFunction();
}
```

Example: Overloading a method called max() (1 of 2)

```
/** Return the max of two int values */
public static int max(int num1, int num2) {
	if (num1 > num2)
		return num1;
	else
		return num2;
}

/** Find the max of two double values */
public static double max(double num1, double num2) {
	if (num1 > num2)
		return num1;
	else
		return num2;
}

/** Return the max of three double values */
public static double max(double num1, double num2, double num3)
{
	return max(max(num1, num2), num3);
}
```


Example: Overloading a method called max() (2 of 2)

```
/** Return the max of two character values */
public static char max(char ch1, char ch2) {
	if (ch1 > ch2)
		return ch1;
	else
		return ch2;
}
/** Return the max of two String values */
public static String max(String s1, String s2) {
	if (s1.compareTo(s2)>0)
		return s1;
	else
		return s2;
}
public static void main(String[] args) {
	// Invoke the max method with different parameters
	System.out.println(max(3, 4));
	System.out.println(max(3.0, 5.4));
	System.out.println(max(3.0, 5.4, 10.14));
	System.out.println(max("Java", "Python"));
	System.out.println(max('E', 'S'));
}
```

Ambiguous Invocation in overloaded functions

- Sometimes there may be two or more possible matches for an invocation (call) of an overloaded function
- In this case, the compiler cannot determine the most specific match. This is referred to as *ambiguous invocation*.
- Ambiguous invocation results in a compilation error.

Example: Ambiguous Invocation

```
public static void main(String[] args)
{
  System.out.println(maxNumber(1, 2));//error
}
```

```
public static int maxNumber(int num1, double num2){
  if (num1 > num2)
    return num1;
  else
    return num2;
}
public static double maxNumber(double num1, int num2){
  if (num1 > num2)
    return num1;
  else
    return num2;
}
```

Common mistakes in code writing with methods/functions

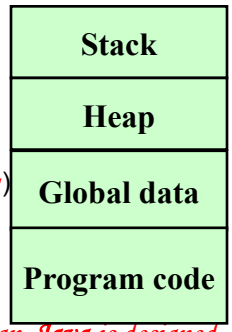
- Misspelled function name in the function call or function header.
- Calling a void-function from the println() statement.
- No return statement for a non-void function.
- Returning a value from a void-function.
- Using data-types in the function call.

Scope of Objects/Variables

Background info: Data storage and Memory

Four sections of the memory areas store four different types data:

- Stack
  - Stores Local variables and formal parameters
  - Managed by the compiler
- Heap
  - Stores dynamic variables
  - Managed by storage allocator
- Global Data (not used by java, see the note below)
  - Stores global variables
- Program code
  - Stores Other program data



- *Note: Java doesn't support global variables, by design. Java is designed with object-oriented principles in mind and as such, every variable in Java is either local or a member of a class.*

Scope of an Object/Variable

- Scope (visibility) refers to the region of the program in which an object/variable can be legally used or visible.
- It also, provides the information about the lifetime of a variable/object.
  - The lifetime of an object/variable is the time during which an identifier actually has memory allocated to it.
- Scopes are of two types:
  - Local scope associated to local variables
  - Global scope associated to global variables (Java does not use global variables; see the note on the previous slide)

Local variables having local scope/visibility

- Local variable/Object: The variables/object are declared inside a block or inside a method or in any method-header or for-header are the local variables/objects.

  Note: A local variable must be declared before it can be used.

- Local scope, also known as Block scope, extends from the point of the local variable/object-declaration to the end of the block.
- Can be accessed only within the block that declares it. This includes:
  - An object declared inside a block enclosed by '{' and '}'
  - An object declared inside a method
  - Formal Parameters of a function/method
  - Any parameter declared in the for-header

Example on Local Scope

```
public class MyClass {
  public static void main(String[] args) {
    int n = 10;
    Scanner input = new Scanner(System.in);
    for(int i = 1; i<=1; i++) {
      System.out.println(incrementBy2(n));
    }
    System.out.println(n);
    input.close();
  }
  public static int incrementBy2(int num) {
    int k = 2;
    return (num+k);
  }
}
```

Scope of n

Scope of i

Scope of input

Scope of k

Scope of num

Scope of Local variables (1 of 2)

- One can declare a local variable with the same name multiple times in different non-nesting blocks

```
public static void main(String[] args){
      System.out.println("Loop 1");
      for(int i=2;i<4;i++){//this i belongs to loop 1
            System.out.println("loop 1: "+i);
      }
      System.out.println("Loop 2");
      for(int i=4;i>2;i--){//this i belongs to loop 2
            System.out.println ("loop 2: "+i);
      }
}
```

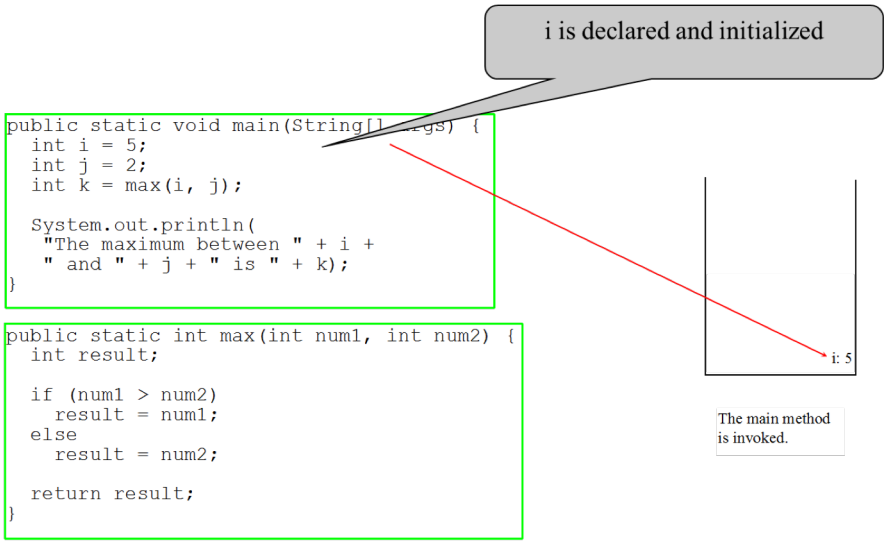

Scope of Local variables (2 of 2)

- One can NOT declare a local variable with the same name twice in the same block or in a nested blocks . In this case compilation error will be generated.

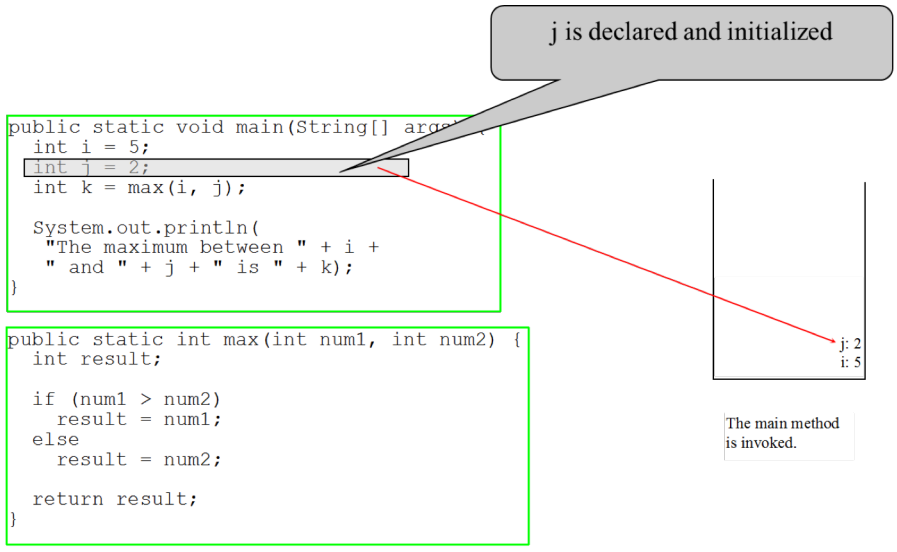
```
//With errors
public static void main(String[] args){
      int x = 10; System.out.println(x);
      double x = 3.14159; System.out.println(x);
}
```

```
// With errors
public static void incorrectMethod() {
  int x = 1;
  int y = 1;
  for (int i = 1; i < 10; i++) {
    int x = 0;
    x += i;
  }
}
```

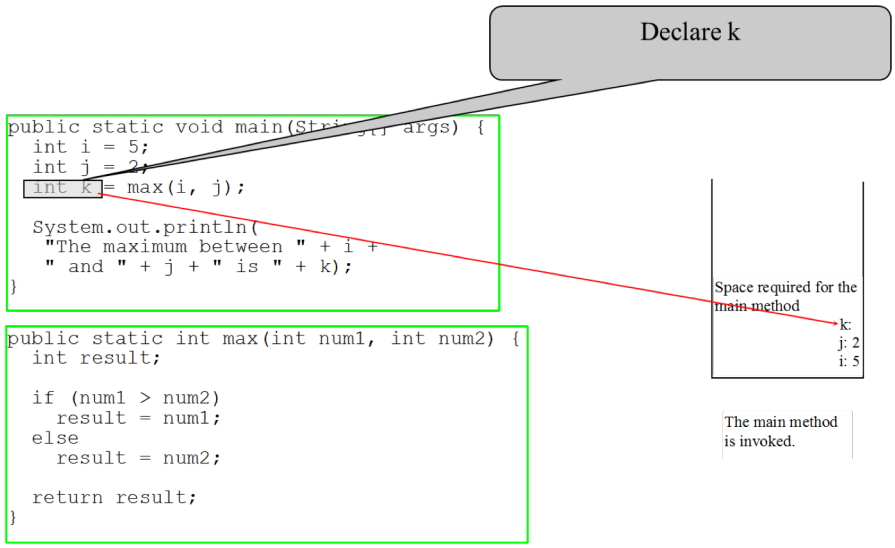
Demo on how a stack in the RAM is used (1 of 10)



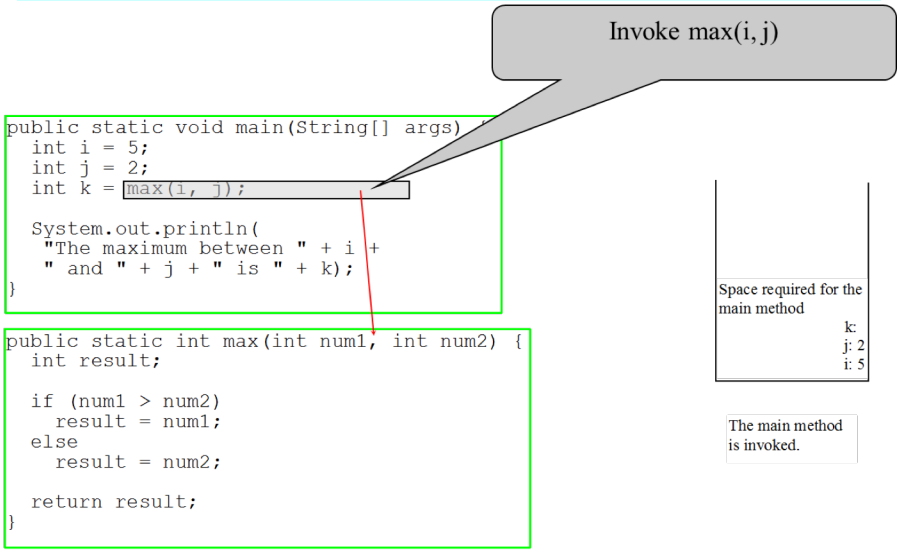
Demo on how a stack in the RAM is used (2 of 10)



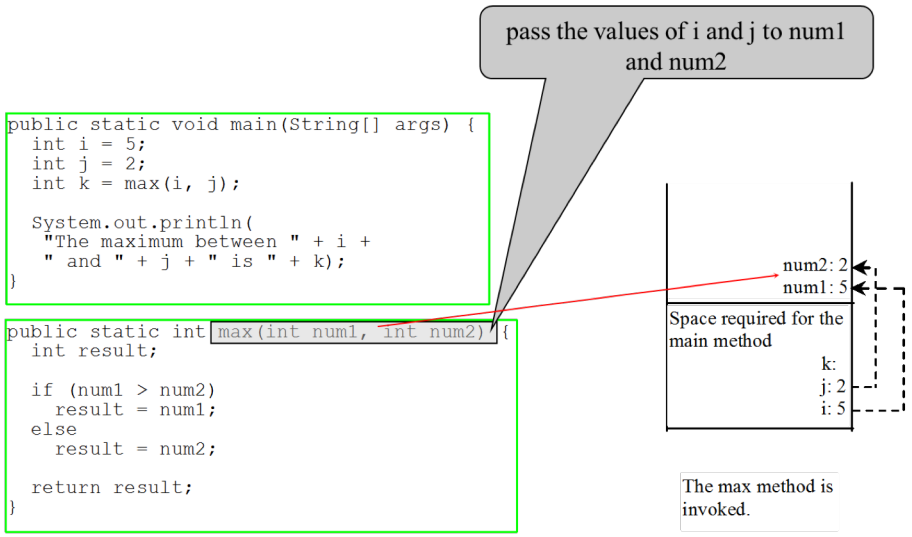
Demo on how a stack in the RAM is used (3 of 10)



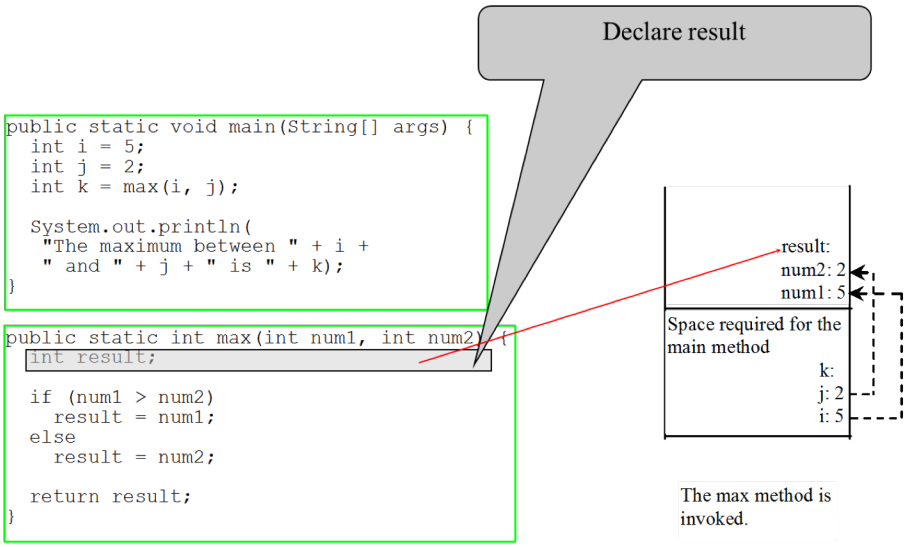
Demo on how a stack in the RAM is used (4 of 10)



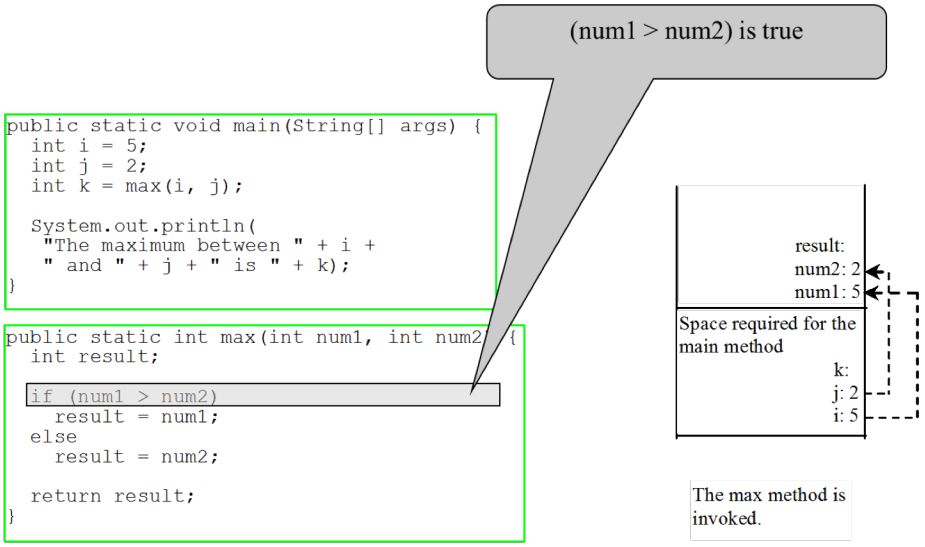
Demo on how a stack in the RAM is used (5 of 10)



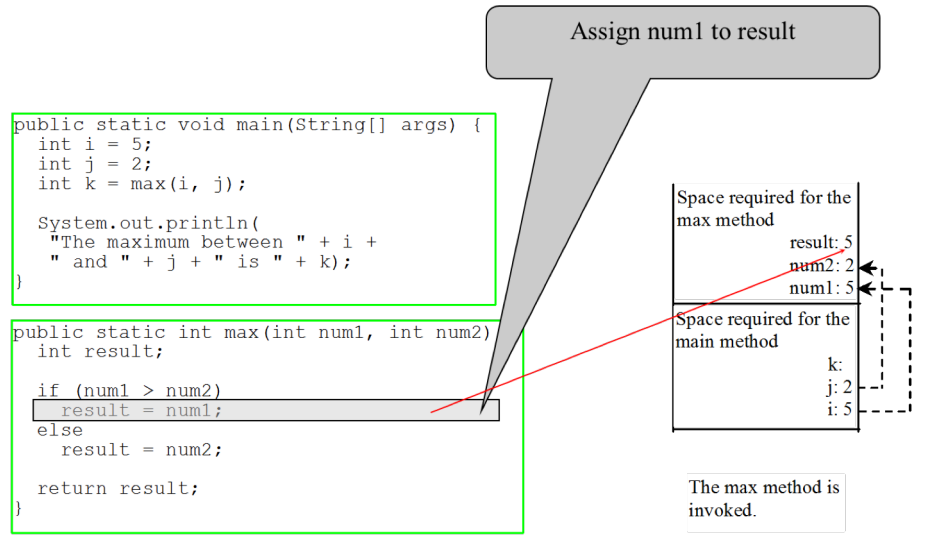
Demo on how a stack in the RAM is used (6 of 10)



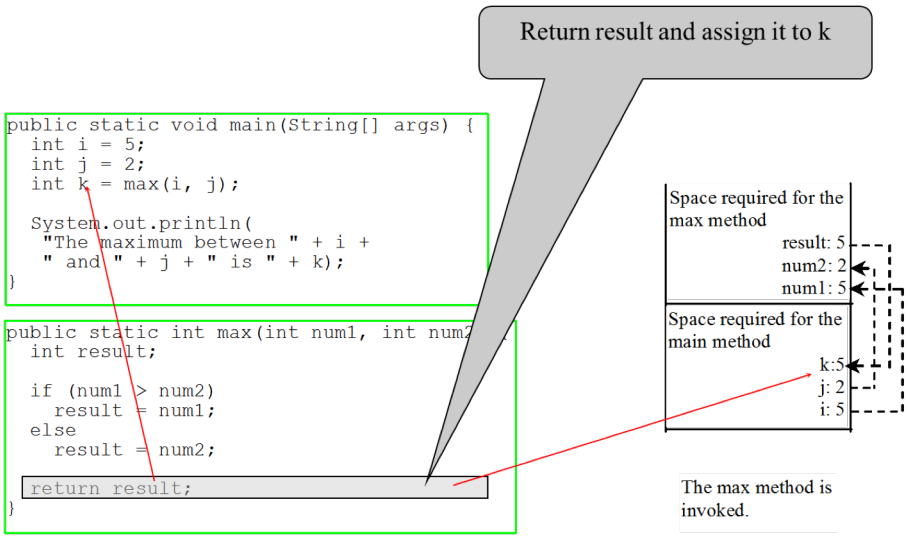
Demo on how a stack in the RAM is used (7 of 10)



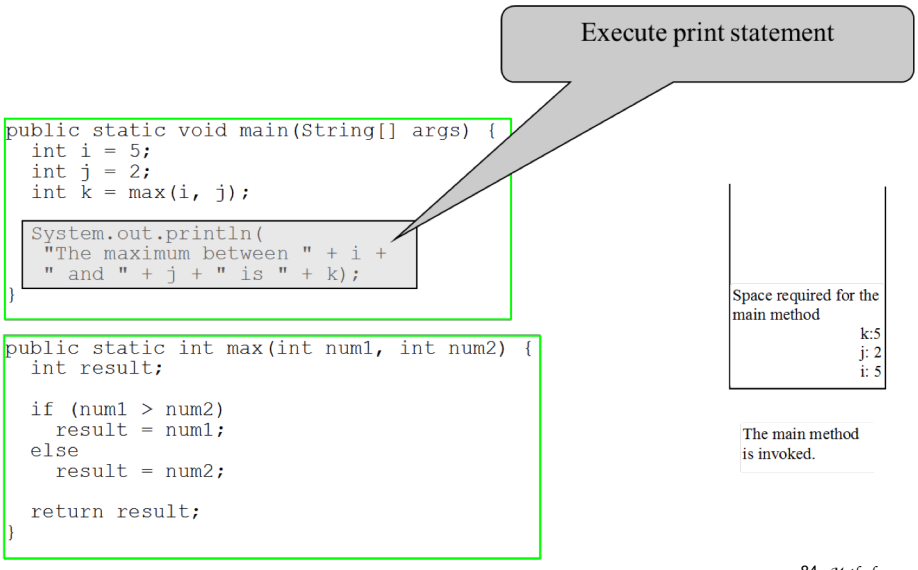
Demo on how a stack in the RAM is used (8 of 10)



Demo on how a stack in the RAM is used (9 of 10)



Demo on how a stack in the RAM is used (10 of 10)



Review

64) A _____ variable is defined inside the body of a function and is not accessible outside that function.

a) reference
b) counter
c) local
d) global
e) None of the above

Review

65) Functions are ideal for use in menu-driven programs. When a user selects a menu item, the program can _____ the appropriate function to carry out the user's choice.

a) define
b) declare
c) call
d) return
e) None of the above

Review

66) A function is executed when it is

a) defined
b) declared
c) prototyped
d) called
e) None of the above

Review

67) The group of the java statements in which an object/variable can be used is its

a) Storage class
b) Scope
c) Declaration
d) Formal argument

Review

68) Will the following code compile?

```
public class MyClass {
   public static void main(String[] args) {
   int n = 10;
   for(int i = 1; i<=1; i++)
       System.out.println(i);
   System.out.println(n+i);
  }
}
```

a) Yes

b) No

FYI: Design Issue Revisited (1 of 10)

- When writing a large program, you can use the "divide and conquer" strategy, also known as **stepwise refinement**, to decompose it into sub problems.
- The sub problems can be further decomposed into smaller, more manageable problems.

FYI: Design Issue Revisited (2 of 10)

- PrintCalender Case Study:
  - Let us use the PrintCalendar example to demonstrate the stepwise refinement approach.

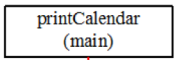
```
Enter full year (e.g., 2001): 2009
Enter month in number between 1 and 12: 4
         April 2009
-----------------------------
 Sun Mon Tue Wed Thu Fri Sat
                   1   2   3   4
   5   6   7   8   9  10  11
  12  13  14  15  16  17  18
  19  20  21  22  23  24  25
  26  27  28  29  30
```


FYI: Design Issue Revisited (3 of 10)

- When writing a large program, you can use the "divide and conquer" strategy, also known as **stepwise refinement**, to decompose it into sub problems.
- The sub problems can be further decomposed into smaller, more manageable problems.

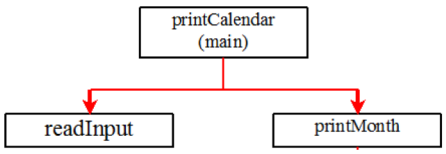
FYI: Design Issue Revisited (4 of 10)

- Design Flow diagram (1 of 7)



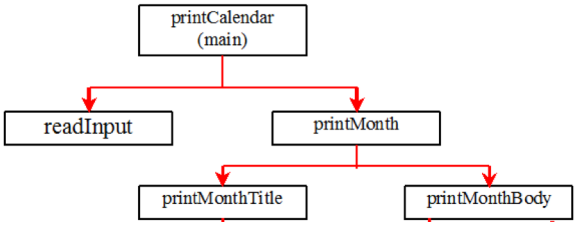
FYI: Design Issue Revisited (5 of 10)

- Design Flow diagram (2 of 7)



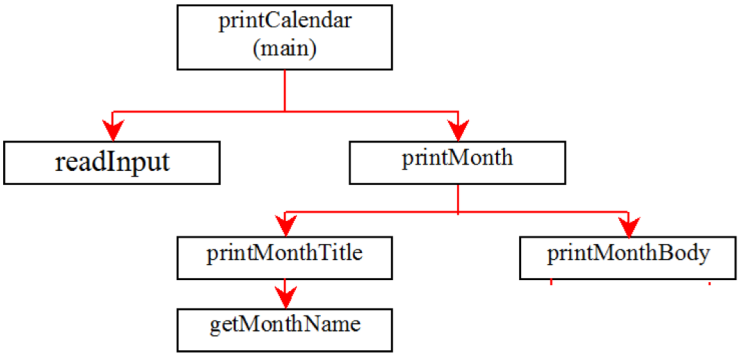
FYI: Design Issue Revisited (6 of 10)

- Design Flow diagram (3 of 7)



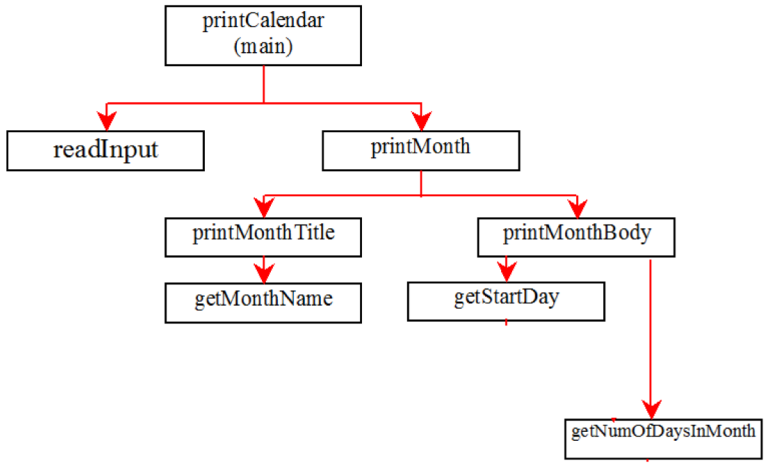
FYI: Design Issue Revisited (7 of 10)

- Design Flow diagram (4 of 7)



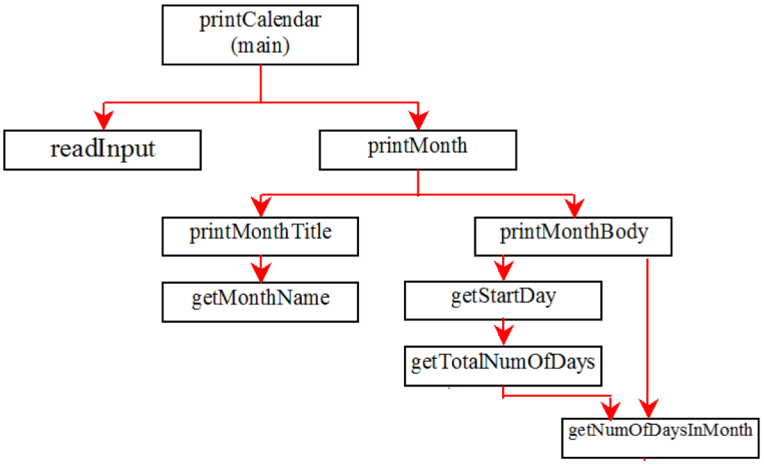
FYI: Design Issue Revisited (8 of 10)

- Design Flow diagram (5 of 7)



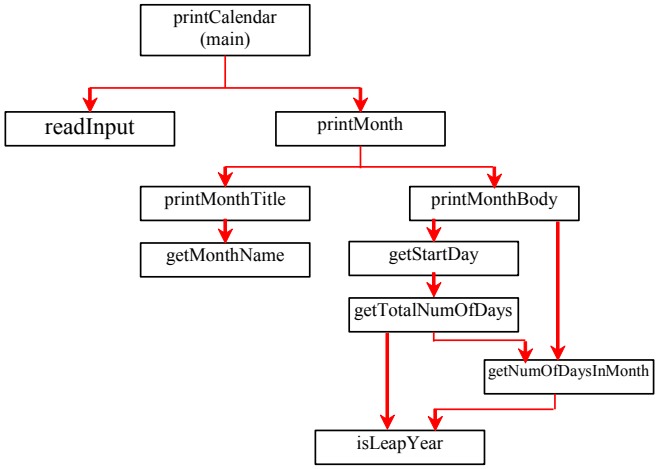
FYI: Design Issue Revisited (9 of 10)

- Design Flow diagram (6 of 7)



FYI: Design Issue Revisited (10 of 10)

- Design Flow diagram (7 of 7)



FYI: Advantage of Stepwise refinement

- Simpler Program
- Reusing Methods
- Easier Developing, Debugging, and Testing
- Better Facilitating Teamwork

FYI: Implementation Approach

- Top-down Approach:
  - Top-down approach is to implement one method in the structure chart at a time from the top to the bottom.
  - Stubs can be used for the methods waiting to be implemented. A stub is a simple but incomplete version of a method.
  - The use of stubs enables you to test invoking the method from a caller. Implement the main method first and then use a stub for the printMonth method.
  - For example, let printMonth display the year and the month in the stub.

FYI: Implementation Approach

- Bottom-Up Approach:
  - Bottom-up approach is to implement one method in the structure chart at a time from the bottom to the top.
  - For each method implemented, write a test program to test it.
  - Both top-down and bottom-up methods are fine. Both approaches implement the methods incrementally and help to isolate programming errors and makes debugging easy. Sometimes, they can be used together.

FYI: Implementation Approach

- Bottom-Up Approach:
  - Bottom-up approach is to implement one method in the structure chart at a time from the bottom to the top.
  - For each method implemented, write a test program to test it.
  - Both top-down and bottom-up methods are fine. Both approaches implement the methods incrementally and help to isolate programming errors and makes debugging easy. Sometimes, they can be used together.

FYI: Complete Implementation (1 of 4)

```
/** Main method */
public static void main(String[] args) {
  Scanner input = new Scanner(System.in);
  // Prompt the user to enter year
  System.out.print("Enter full year (e.g., 2001): ");
  int year = input.nextInt();
  // Prompt the user to enter month
  System.out.print("Enter month in number between 1 and 12: ");
  int month = input.nextInt();
  // Print calendar for the month of the year
  printMonth(year, month);
  input.close();
}
/** Print the calendar for a month in a year */
public static void printMonth(int year, int month) {
  // Print the headings of the calendar
  printMonthTitle(year, month);
  // Print the body of the calendar
  printMonthBody(year, month);
}
```

FYI: Complete Implementation (2 of 4)

```
/** Print the month title, e.g., May, 1999 */
public static void printMonthTitle(int year, int month) {
  System.out.println("         " + getMonthName(month) + " " + year);
  System.out.println("-----------------------------");
  System.out.println(" Sun Mon Tue Wed Thu Fri Sat");
}
/** Get the English name for the month */
public static String getMonthName(int month) {
  String monthName = "";
  switch (month) {
    case 1: monthName = "January"; break;
    case 2: monthName = "February"; break;
    case 3: monthName = "March"; break;
    case 4: monthName = "April"; break;
    case 5: monthName = "May"; break;
    case 6: monthName = "June"; break;
    case 7: monthName = "July"; break;
    case 8: monthName = "August"; break;
    case 9: monthName = "September"; break;
    case 10: monthName = "October"; break;
    case 11: monthName = "November"; break;
    case 12: monthName = "December";
  }
return monthName;
}
```


FYI: Complete Implementation (3 of 4)

```
/** Print month body */
public static void printMonthBody(int year, int month) {
  // Get start day of the week for the first date in the month
  int startDay = getStartDay(year, month);
  // Get number of days in the month
  int numberOfDaysInMonth = getNumberOfDaysInMonth(year, month);
  // Pad space before the first day of the month
  int i = 0;
  for (i = 0; i < startDay; i++)
    System.out.print("    ");
  for (i = 1; i <= numberOfDaysInMonth; i++) {
    System.out.printf("%4d", i);
    if ((i + startDay) % 7 == 0)
      System.out.println();
  }
 System.out.println();
}
/** Get the start day of month/1/year */
public static int getStartDay(int year, int month) {
  final int START_DAY_FOR_JAN_1_1800 = 3;
  // Get total number of days from 1/1/1800 to month/1/year
  int totalNumberOfDays = getTotalNumberOfDays(year, month);
  // Return the start day for month/1/year
  return (totalNumberOfDays + START_DAY_FOR_JAN_1_1800) % 7;
}
```

FYI: Complete Implementation (4 of 4)

```
/** Get the total number of days since January 1, 1800 */
public static int getTotalNumberOfDays(int year, int month) {
  int total = 0;
  // Get the total days from 1800 to 1/1/year
  for (int i = 1800; i < year; i++)
    if (isLeapYear(i))
      total = total + 366;
    else
      total = total + 365;
// Add days from Jan to the month prior to the calendar month
  for (int i = 1; i < month; i++)
    total = total + getNumberOfDaysInMonth(year, i);
return total;
}
/** Get the number of days in a month */
public static int getNumberOfDaysInMonth(int year, int month) {
  if (month == 1 || month == 3 || month == 5 || month == 7 ||
      month == 8 || month == 10 || month == 12)
      return 31;
  if (month == 4 || month == 6 || month == 9 || month == 11)
      return 30;
  if (month == 2) return isLeapYear(year) ? 29 : 28;
  return 0; // If month is incorrect
}
/** Determine if it is a leap year */
public static boolean isLeapYear(int year) {
  return year % 400 == 0 || (year % 4 == 0 && year % 100 != 0);
}
```


Answer Key

56. B
57. B
58. E
59. C
60. D
61. D
62. A
63. A
64. C
65. C
66. D
67. B
68. B