CHAPTER 6
METHODS

ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH INTRODUCTION TO JAVA PROGRAMMING, LIANG (PEARSON 2014)

OPENING PROBLEM

- Find the sum of integers from 1 to 10, from 20 to 30, and from 35 to 45, respectively.
- Compute the square root of a number over and over again
- Organize a large program into smaller components

PROBLEM

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

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

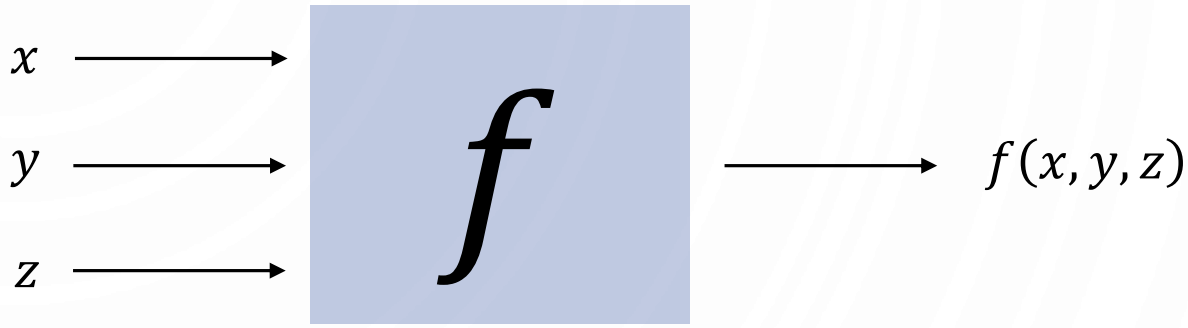
```
1. int sum = 0;
2. for (int i = 35; i <= 45; i++)
3.   sum += i;
4. System.out.println("Sum from 35 to 45 is " + sum);
```

SOLUTION

```
1.  public static int sum(int i1, int i2) {
2.    int sum = 0;
3.    for (int i = i1; i <= i2; i++)
4.      sum += i;
5.    return sum;
6.  }
7.
8.  public static void main(String[] args) {
9.    System.out.println("Sum from 1 to 10 is " + sum(1, 10));
10.   System.out.println("Sum from 20 to 30 is " + sum(20, 30));
11.   System.out.println("Sum from 35 to 45 is " + sum(35, 45));
12. }
```

DEFINING METHODS

- A **method** is a collection of statements that are grouped together to perform an operation.
  - "function"
  - "subroutine"
  - "algorithm"



Define a method

```
public static int max(int num1, int num2) {

   int result;

   if (num1 > num2)
     result = num1;
   else
     result = num2;

   return result;
}
```

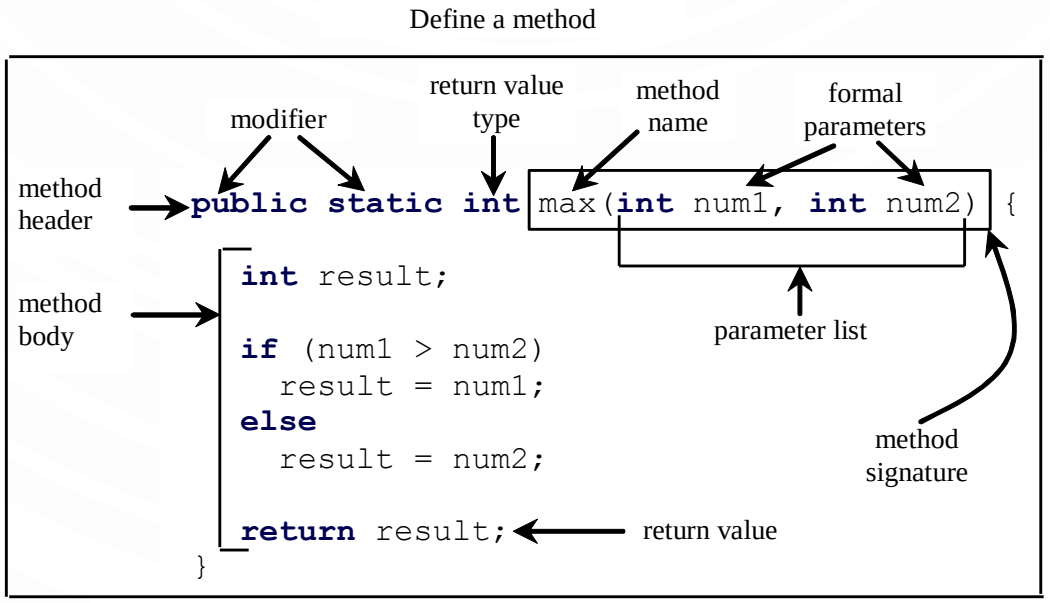
Invoke a method

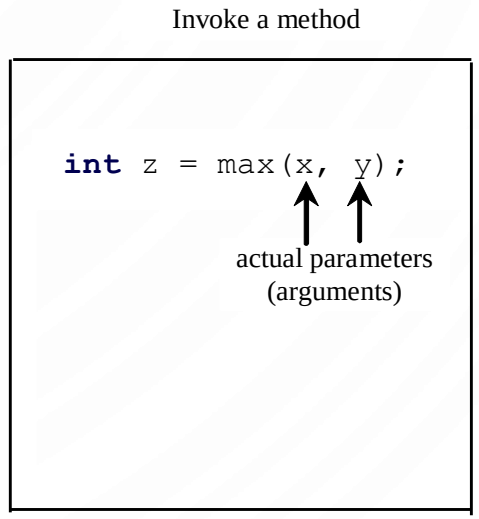
```
int z = max(x, y);
```

actual parameters (arguments)

DEFINING METHODS

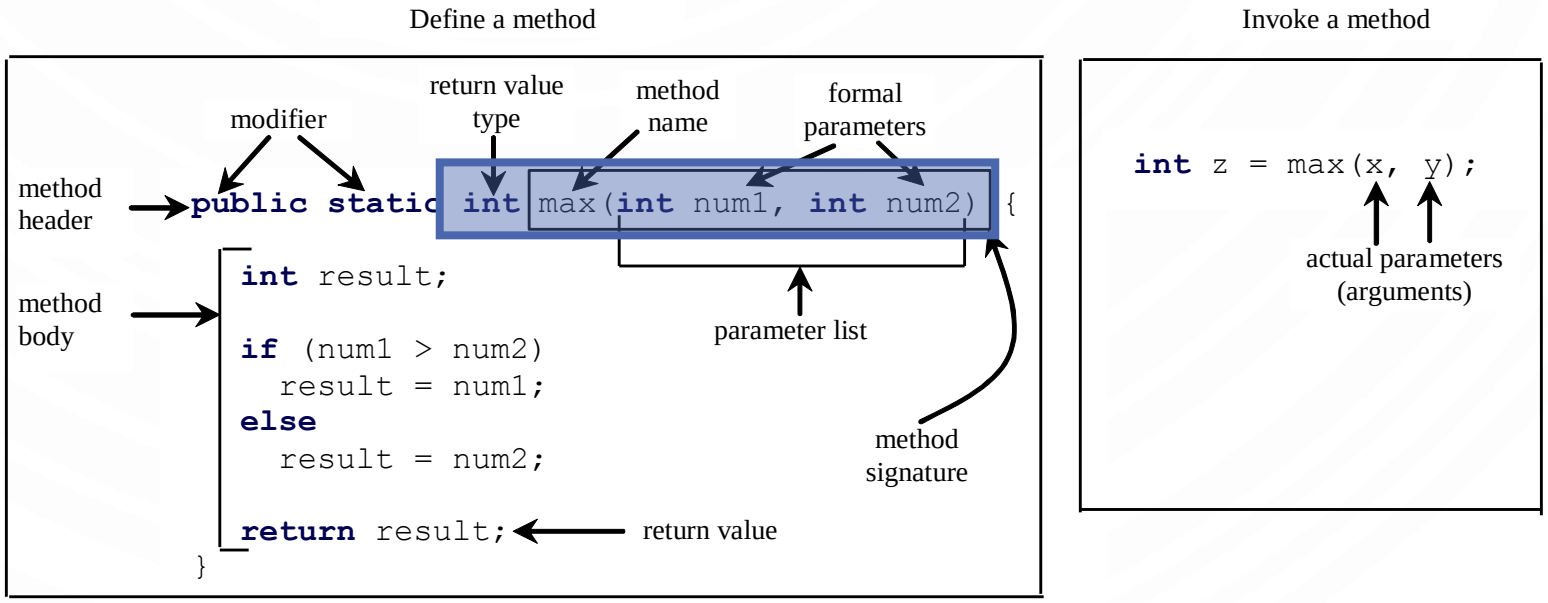
- A **method** is a collection of statements that are grouped together to perform an operation.
  - "function"
  - "subroutine"
  - "algorithm"





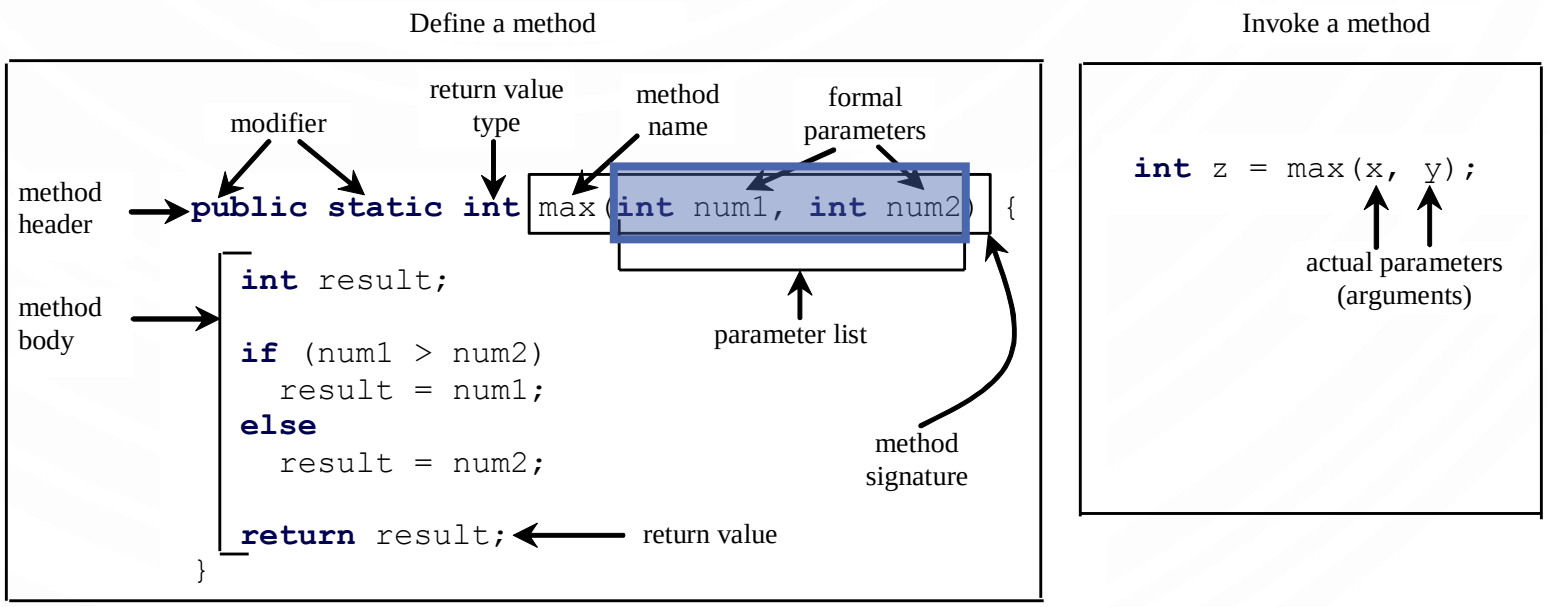
METHOD SIGNATURE

- **Method signature** is the combination of the method name and the parameter list.



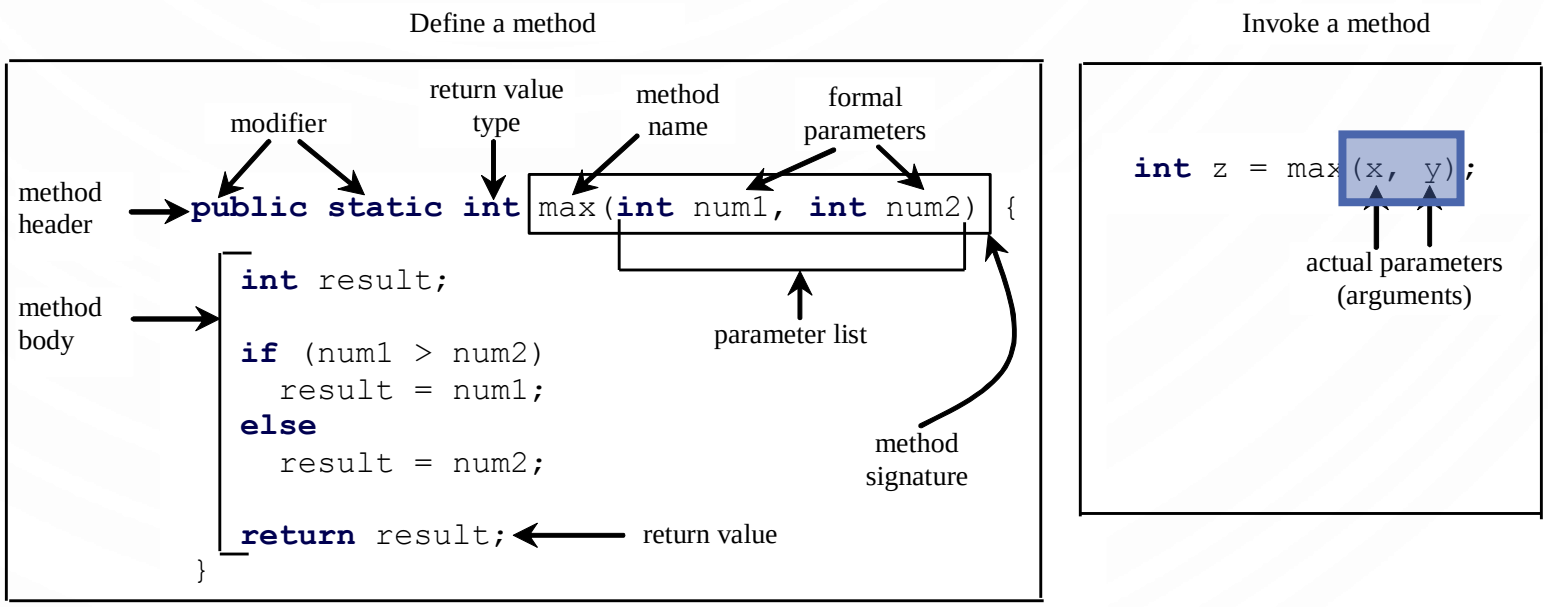
FORMAL PARAMETERS

- The variables defined in the method header are known as **formal parameters**.



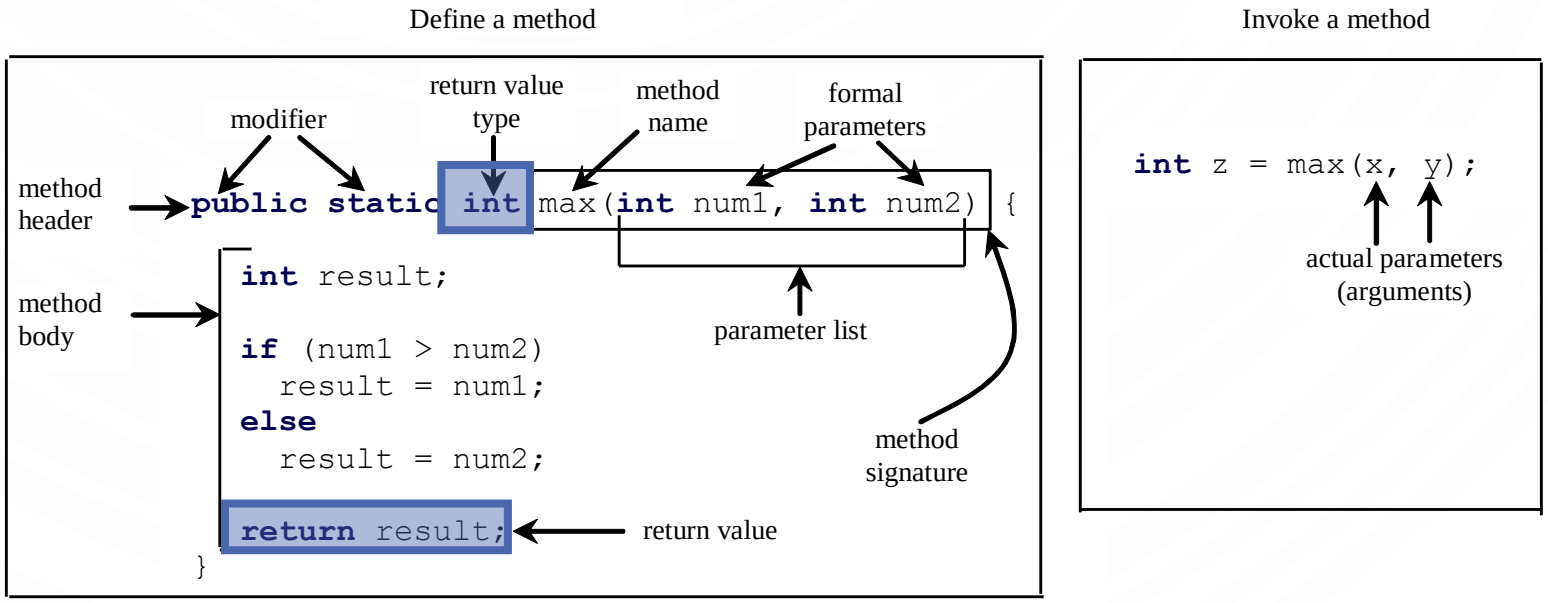
ACTUAL PARAMETERS

- When a method is invoked, you pass a value to the parameter. This value is referred to as **actual parameter** or **argument**.



RETURN VALUE TYPE

- A method may return a value. The **returnValueType** is the data type of the value the method returns. If the method does not return a value, the **returnValueType** is the keyword **void**.



CAUTION

- 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.
- 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.

```
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)

EXAMPLE

- Lets write a function to compute a random integer between $[a, b]$ (without using **`Random`**)
- Lets write a function to computes the square of a number (without using **`Math`**`.pow`)
- Lets write a function to output a telephone number in a specific format (assume input it 10 numbers as a **`String`**, output should be XXX-XXX-XXXX)

EXERCISE – WITH A PARTNER

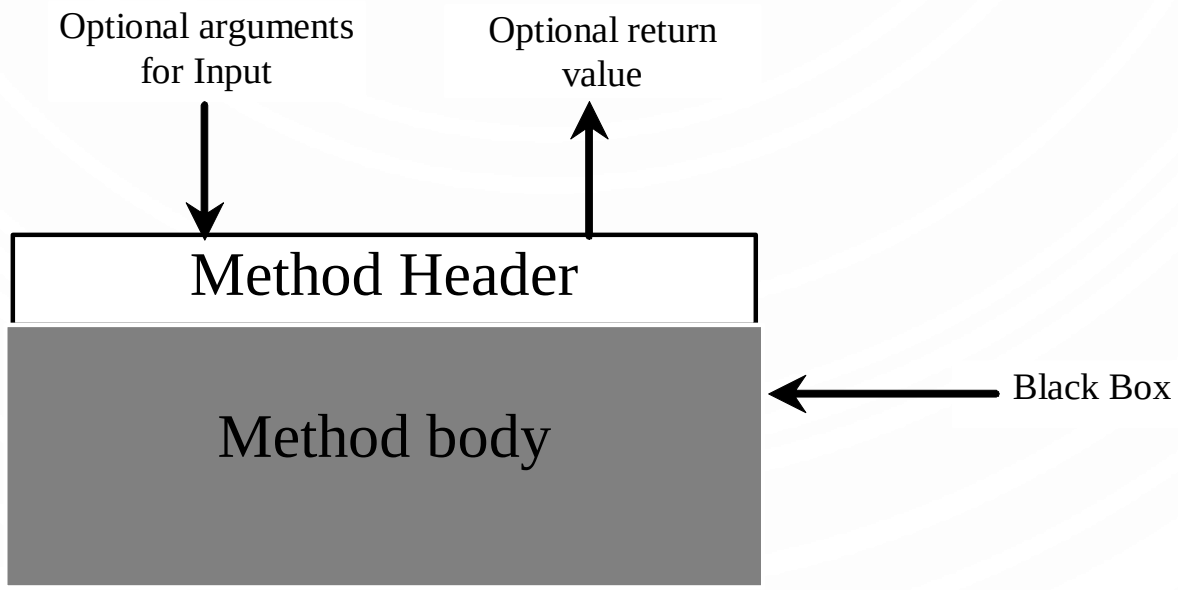
- Write a function to determine if two circles overlap
- Write a function that converts a number to binary (as a **`String`**)

REUSE METHODS FROM OTHER CLASSES

- NOTE: One of the benefits of methods is for reuse. The max method can be invoked from any class besides TestMax. If you create a new class Test, you can invoke the max method using ClassName.methodName (e.g., TestMax.max).
- Example
  - `Math.sqrt()`

METHOD ABSTRACTION

- You can think of the method body as a black box that contains the detailed implementation for the method.

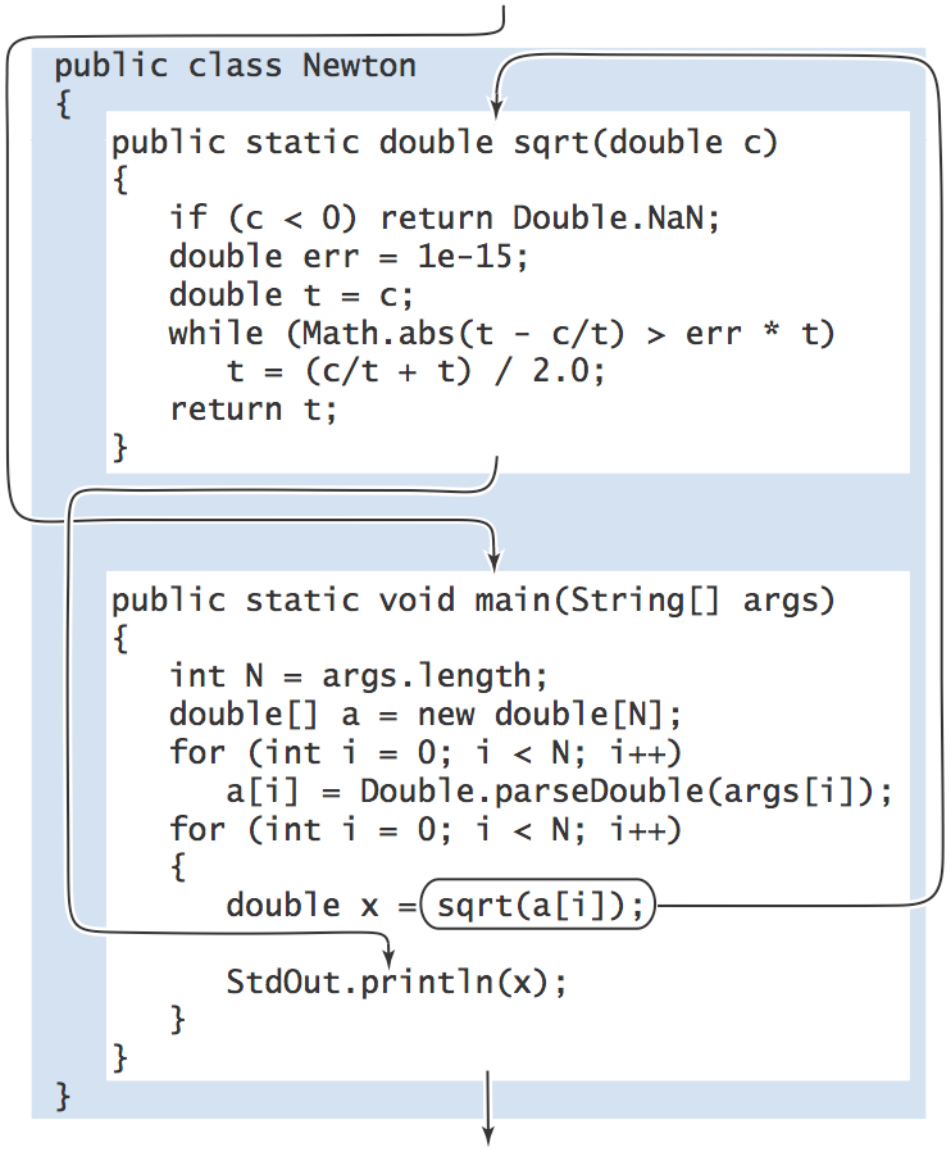


MODULARIZING CODE

- Methods can be used to reduce redundant coding and enable code reuse. Methods can also be used to modularize code and improve the quality of the program.

FLOW OF CONTROL

- Key point. Functions provide a new way to control the flow of execution.
- What happens when a function is called:
  - Control transfers to the function code.
  - Argument variables are assigned the values given in the call.
  - Function code is executed.
  - Return value is assigned in place of the function name in calling code.
  - Control transfers back to the calling code.



TRACE CALLING METHODS

```
1. public static void main(
       String[] args) {
2.
3.   int i = 0;
4.   int j = 2;
5.   int k = max(i, j);
6.
7.   System.out.println(
8.     "The maximum between " +
9.     i + " and " + j +
10.    " is" + k);
11.}
```

```
1. public static int max(
       int num1, int num2) {
2.   return num1 > num2 ?
3.     num1 : num2;
4. }
```

Memory

TRACE CALLING METHODS

```
1. public static void main(
       String[] args) {
2.
3.   int i = 0;
4.   int j = 2;
5.   int k = max(i, j);
6.
7.   System.out.println(
8.     "The maximum between " +
9.     i + " and " + j +
10.    " is" + k);
11.}
```

```
1. public static int max(
       int num1, int num2) {
2.   return num1 > num2 ?
3.     num1 : num2;
4. }
```

Memory

Function main

i 0

TRACE CALLING METHODS

```
1. public static void main(
       String[] args) {
2.
3.   int i = 0;
4.   int j = 2;
5.   int k = max(i, j);
6.
7.   System.out.println(
8.     "The maximum between " +
9.     i + " and " + j +
10.    " is" + k);
11.}
```

```
1. public static int max(
       int num1, int num2) {
2.   return num1 > num2 ?
3.     num1 : num2;
4. }
```

Memory

Function main

i 0 | j 2

TRACE CALLING METHODS

```
1. public static void main(
       String[] args) {
2.
3.   int i = 0;
4.   int j = 2;
5.   int k = max(i, j);
6.
7.   System.out.println(
8.     "The maximum between " +
9.     i + " and " + j +
10.    " is" + k);
11.}
```

```
1. public static int max(
       int num1, int num2) {
2.   return num1 > num2 ?
3.     num1 : num2;
4. }
```

Memory

Function max

num1 0 | num2 2

Function main

i 0 | j 2

TRACE CALLING METHODS

```
1. public static void main(
       String[] args) {
2.
3.   int i = 0;
4.   int j = 2;
5.   int k = max(i, j);
6.
7.   System.out.println(
8.     "The maximum between " +
9.     i + " and " + j +
10.    " is" + k);
11.}
```

```
1. public static int max(
       int num1, int num2) {
2.   return num1 > num2 ?
3.     num1 : num2;
4. }
```

Memory

Function max

num1 0 | num2 2

result 2

Function main

i 0 | j 2

k 2

TRACE CALLING METHODS

```
1.  public static void main(
        String[] args) {
2.
3.    int i = 0;
4.    int j = 2;
5.    int k = max(i, j);
6.
7.    System.out.println(
8.      "The maximum between " +
9.      i + " and " + j +
10.     " is" + k);
11. }
```

```
1.  public static int max(
        int num1, int num2) {
2.    return num1 > num2 ?
3.      num1 : num2;
4.  }
```

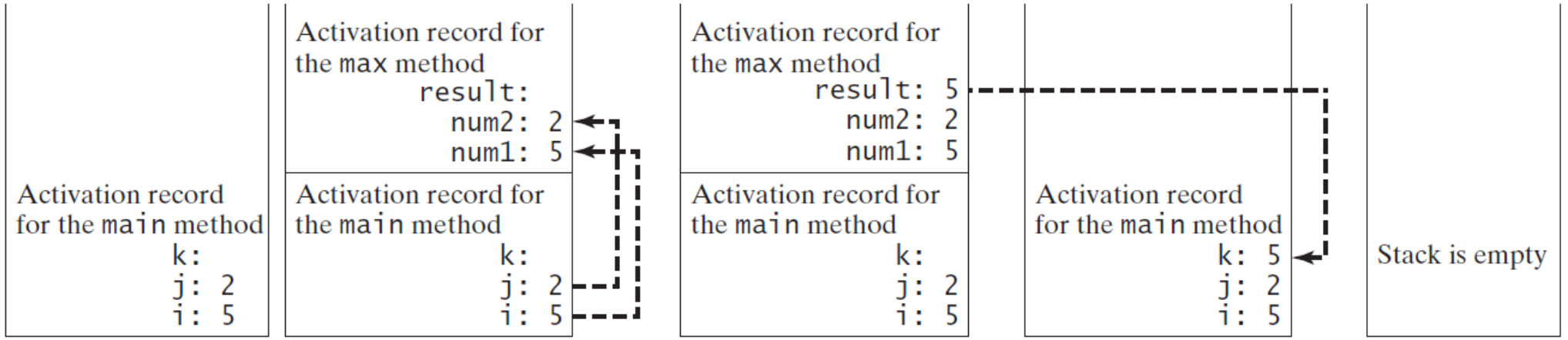
Memory

Function main

i 0 | j 2

k 2

CALL STACKS



(a) The `main` method is invoked.

(b) The `max` method is invoked.

(c) The `max` method is being executed.

(d) The `max` method is finished and the return value is sent to `k`.

(e) The `main` method is finished.

PASS-BY-VALUE

- All parameters in Java are passed by value (copied!)

```
1.  public static void nPrintln(
        String message, int n) {
2.    for (int i = 0; i < n; i++)
3.      System.out.println(message);
4.  }
```

"You're exactly the kind of applicant we're looking for."

- Suppose you invoke the method using `nPrintln("Welcome to Java", 5);`
- What is the output?

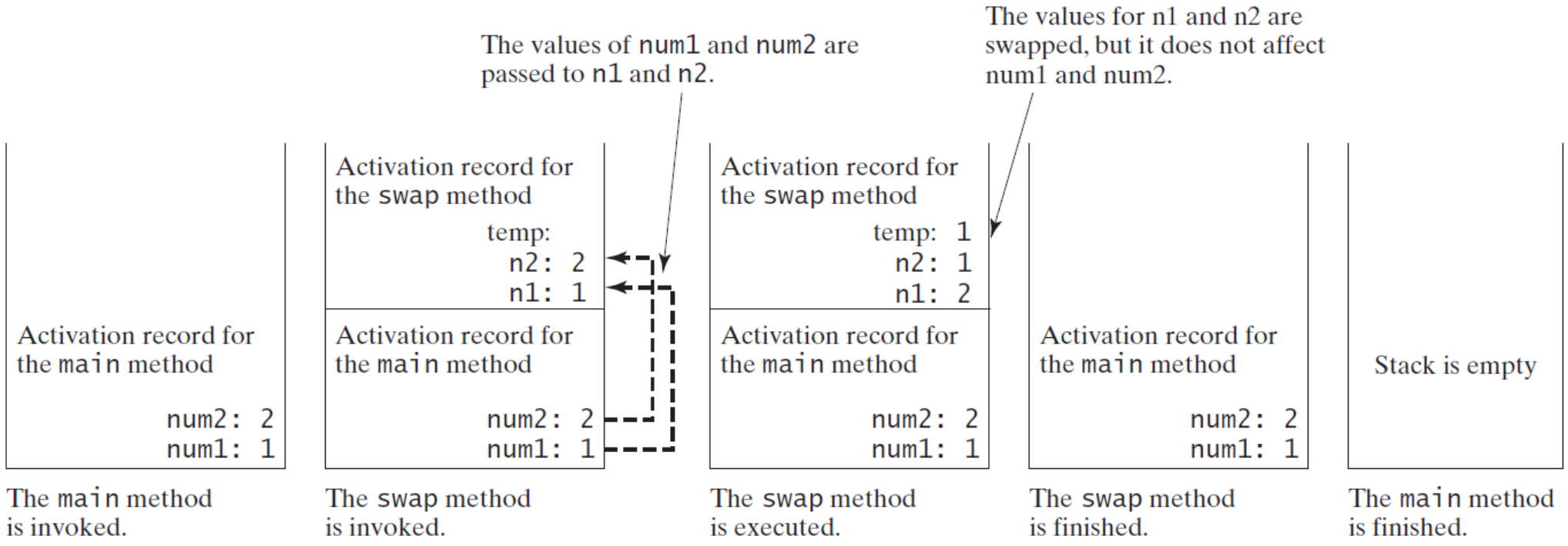
- Suppose you invoke the method using `nPrintln("Computer Science", 15);`
- What is the output?

- Can you invoke the method using `nPrintln(15, "Computer Science");`

EXAMPLE OF PASS-BY-VALUE

```
1.  public static void main(String[] args) {
2.    int num1 = 1;
3.    int num2 = 2;
4.    swap(num1, num2);
5.  }
6.
7.  public static void swap(int n1, int n2) {
8.    int temp = n1;
9.    n1 = n2;
10.   n2 = temp;
11. }
```

EXAMPLE OF PASS-BY-VALUE



FUNCTION CHALLENGE 1A

- Q. What happens when you compile and run the following code?

```
% java Cubes1
1 1
2 8
3 27
4 64
5 125
6 216
```

```
1.  public class Cubes1 {
2.    public static int cube(int i) {
3.      int j = i * i * i;
4.      return j;
5.    }
6.    public static void main(
7.        String[] args) {
8.      int N = 6;
9.      for (int i = 1; i <= N; i++)
10.       System.println(i + " " +
11.         cube(i));
12.   }
13. }
```

FUNCTION CHALLENGE 1B

- Q. What happens when you compile and run the following code?

Compile error!

```
1.  public class Cubes2 {
2.    public static int cube(int i) {
3.      int i = i * i * i;
4.      return i;
5.    }
6.    public static void main(
7.        String[] args) {
8.      int N = 6;
9.      for (int i = 1; i <= N; i++)
10.       System.println(i + " " +
11.         cube(i));
12.   }
13. }
```

FUNCTION CHALLENGE 1C

- Q. What happens when you compile and run the following code?

Compile error!

```
1.  public class Cubes3 {
2.    public static int cube(int i) {
3.      i = i * i * i;
4.    }
5.    public static void main(
6.        String[] args) {
7.      int N = 6;
8.      for (int i = 1; i <= N; i++)
9.        System.println(i + " " +
10.         cube(i));
11.   }
12. }
```

FUNCTION CHALLENGE 1D

- Q. What happens when you compile and run the following code?

```
% java Cubes4
1 1
2 8
3 27
4 64
5 125
6 216
```

```
1.  public class Cubes4 {
2.    public static int cube(int i) {
3.      i = i * i * i;
4.      return i;
5.    }
6.    public static void main(
7.        String[] args) {
8.      int N = 6;
9.      for (int i = 1; i <= N; i++)
10.       System.println(i + " " +
11.         cube(i));
12.   }
13. }
```

FUNCTION CHALLENGE 1E

- Q. What happens when you compile and run the following code?

```
% java Cubes5
1 1
2 8
3 27
4 64
5 125
6 216
```

```
1.  public class Cubes5 {
2.    public static int cube(int i) {
3.      return i * i * i;
4.    }
5.    public static void main(
6.        String[] args) {
7.      int N = 6;
8.      for (int i = 1; i <= N; i++)
9.        System.println(i + " " +
10.         cube(i));
11.   }
12. }
```

OVERLOADING METHODS

- Method **overloading** is creating more than one method with the same name but different signatures
- Example
  - `public static int abs(int a) {return a < 0 ? -a : a;}`
  - `public static double abs(double a) {return a < 0 ? -a : a;}`

AMBIGUOUS INVOCATION

- Sometimes there may be two or more possible matches for an invocation of a method, but the compiler cannot determine the most specific match. This is referred to as ambiguous invocation. Ambiguous invocation is a compile error.

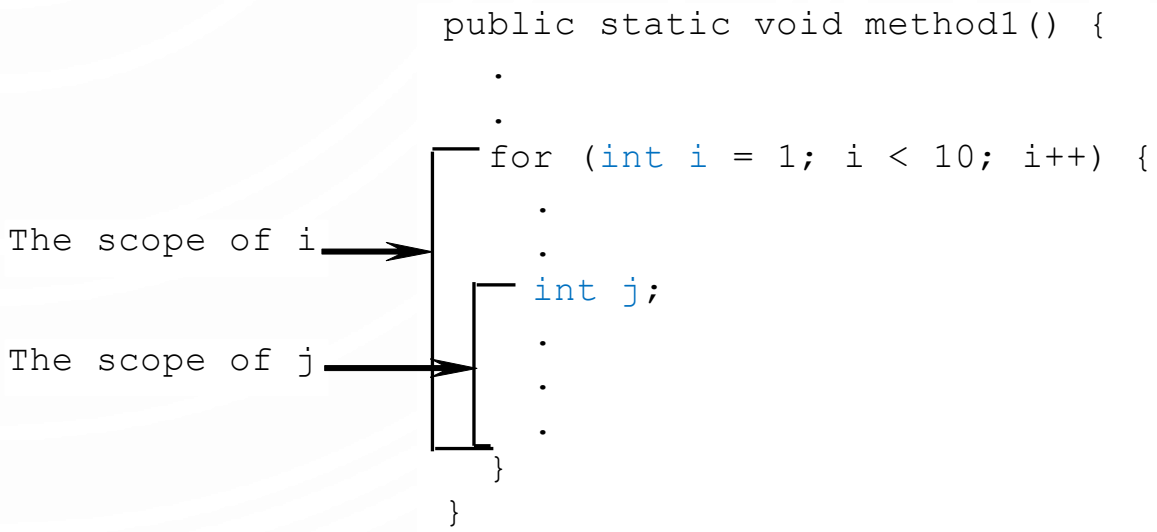
```
1.  public static void main(String[] args) {
2.    System.out.println(max(1, 2));
3.  }
4.
5.  public static double max(
6.      int num1, double num2) {
7.    if (num1 > num2)
8.      return num1;
9.    else
10.     return num2;
11. }
12.
13. public static double max(
14.     double num1, int num2) {
15.   if (num1 > num2)
16.     return num1;
17.   else
18.     return num2;
19. }
```

SCOPE OF LOCAL VARIABLES

- **A local variable** – a variable defined inside a method.
- **Scope** – the part of the program where the variable can be referenced.
- The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable. A local variable must be declared before it can be used.
- You can declare a local variable with the same name multiple times in different non-nesting blocks in a method, but you cannot declare a local variable twice in nested blocks.

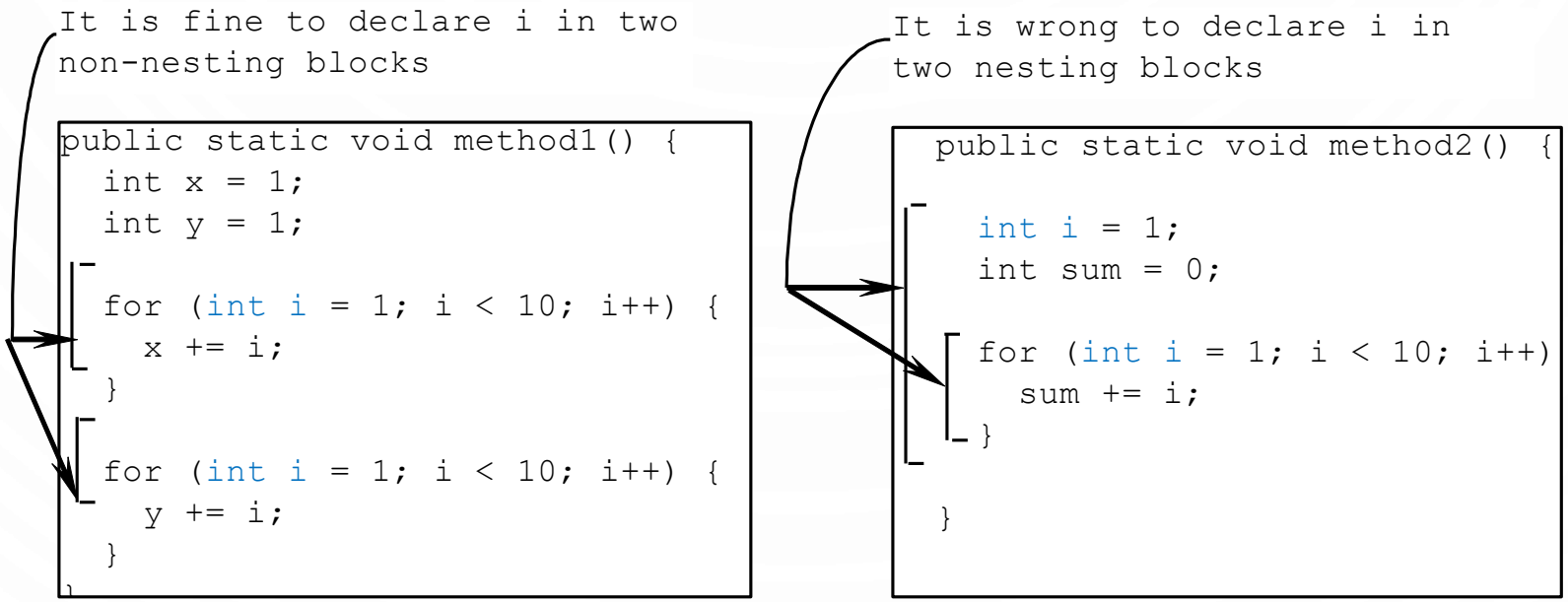
SCOPE OF LOCAL VARIABLES

- A variable declared in the initial action part of a for loop header has its scope in the entire loop. But a variable declared inside a for loop body has its scope limited in the loop body from its declaration and to the end of the block that contains the variable.



SCOPE OF LOCAL VARIABLES

- You can declare a local variable with the same name multiple times in different non-nesting blocks in a method, but you cannot declare a local variable twice in nested blocks.



QUESTION

- What happens in the following?

```
1. public static void incorrectMethod() {
2.    int x = 1;
3.    int y = 1;
4.    for (int i = 1; i < 10; i++) {
5.       int x = 0;
6.       x += i;
7.    }
8. }
```

BENEFITS OF METHODS

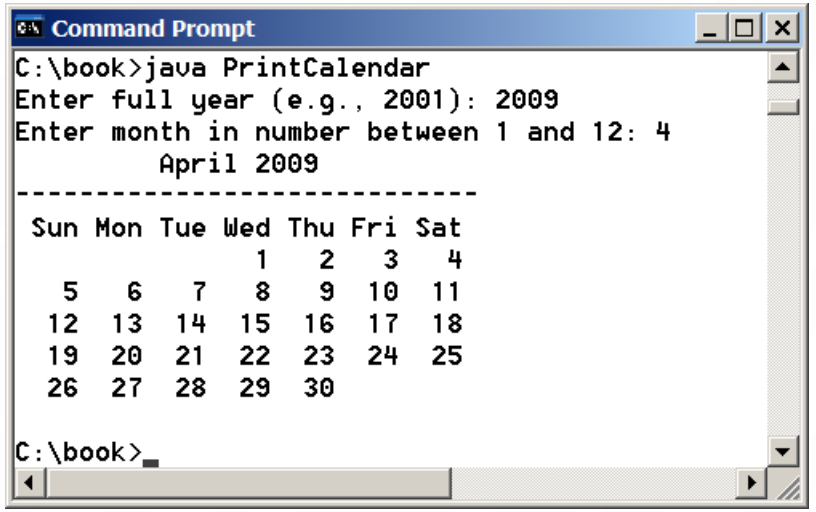
- Write a method once and reuse it anywhere.
- Information hiding. Hide the implementation from the user.
- Reduce complexity.

STEPWISE REFINEMENT

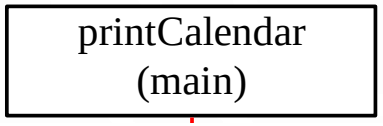
- The concept of method abstraction can be applied to the process of developing programs. When writing a large program, you can use the “divide and conquer” strategy, also known as stepwise refinement, to decompose it into subproblems. The subproblems can be further decomposed into smaller, more manageable problems.

PRINTCALENDER CASE STUDY

- Let us use the PrintCalendar example to demonstrate the stepwise refinement approach.



DESIGN DIAGRAM



IMPLEMENTATION: TOP-DOWN

- 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. Thus, your program may begin like this:

IMPLEMENTATION: BOTTOM-UP

- 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.

BENEFITS OF STEPWISE REFINEMENT

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

UNIT TESTING

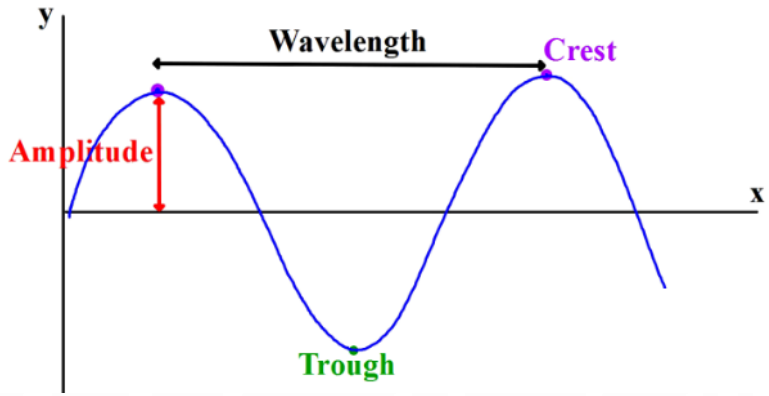
- **Unit test** – Automated piece of code that invoked a "unit" of work and then checks a single assumption of its behavior. Use main() to test each library.
- Follows SETT – unit testing paradigm
  - Setup – create data for input and predetermine the output
  - Execute – call the function in question
  - Test – analyze correctness and determine true/false for test
  - Teardown – cleanup any data, close buffers, etc

- Take a function to compute the square of a number. Here is a good unit test:

```
1.  public static boolean testSquare() {
2.     //Setup - predetermined values!
3.     int x = 5, ans = 25;
4.
5.     //Execute - call the function
6.     int sqr = square(x);
7.
8.     //Test
9.     return sqr == ans;
10.
11.    //Empty teardown
12. }
```

EXERCISE – EVERYONE CODE ALONG

EXERCISE



- Lets make a simple program to study the effects of a wave
- Equation

$$f(x) = A * \cos(Bx - C)$$

- A is the amplitude/height of the wave
- B determines the period of the wave (wavelength)
- C determines the phase shift of the wave (where first crest occurs)

EXERCISE

- Will make a plotting program to plot n points on the curve
- Assume x values range from -10 to 10
- Break the problem into manageable pieces!
  - Input – A, B, C, n
  - Need to be able to compute an interval distance between each of the points
  - Need to be able to evaluate the function
  - Output – Plot of the curve
- Make two files
  - Wave.java – handles the math (library)
  - DrawWave.java – handles the graphics (client)

EXERCISE WAVE.JAVA

- Wave needs to be able to calculate the interval between points, determine the position of the ith interval, and grab a specific value of the function
- Have a main function for unit testing

```
1.  public class Wave {
2.
3.    /// @param n Number of points
4.    /// @return x interval between each point
5.    public static double GetInterval(int n) {
6.    }
7.
8.    /// @param i ith inteval number
9.    /// @param interval x interval
10.   ///        between each point
11.   /// @return ith interval x value
12.   public static double GetT(int i, double
    interval) {
13.   }
15.   /// @param amplitude A in A*cos(Bt - C)
16.   /// @param period B in A*cos(Bt - C),
17.   ///        period is 2pi/B
18.   /// @param phase C in A*cos(Bt - C),
19.   ///        phase shift is B/C
20.   /// @param t time in parametric equation
21.   /// @return A*cos(Bt - C)
22.   public static double GetValue(double
    amplitude, double period, double phase,
23.       double t) {
24.   }
25.
26.   public static void main(String[] args) {
27.     //Simple test of wave module
28.   }
29. }
```

EXERCISE WAVE.JAVA

- Wave needs to be able to calculate the interval between points, determine the position of the ith interval, and grab a specific value of the function
- Have a main function for unit testing

```
1.  public class Wave {
2.
3.    /// @param n Number of points
4.    /// @return x interval between each point
5.    public static double GetInterval(int n) {
6.      return (10. - -10.) / n; //[-10, 10] is hardcoded
7.                               //interval of values
8.    }
9.
10.   /// @param i ith inteval number
11.   /// @param interval x interval
12.   ///        between each point
13.   /// @return ith interval x value
14.   public static double GetT(int i, double interval) {
15.     return -10 + i * interval;
16.   }
```

```
15.   /// @param amplitude A in A*cos(Bt - C)
16.   /// @param period B in A*cos(Bt - C),
17.   ///        period is 2pi/B
18.   /// @param phase C in A*cos(Bt - C),
19.   ///        phase shift is B/C
20.   /// @param t time in parametric equation
21.   /// @return A*cos(Bt - C)
22.   public static double GetValue(double amplitude,
    double period, double phase,
23.       double t) {
24.     return amplitude * Math.cos(period*t - phase);
25.   }
26.
27.   public static void main(String[] args) {
28.     Scanner in = new Scanner(System.in);
29.     System.out.println("Enter amplitude period phase n:
    ");
30.     double amplitude = in.nextDouble()
31.     double period = in.nextDouble();
32.     double phase = in.nextDouble();
33.     int n = in.nextInt();
34.
35.     //Simple test of wave module
36.     double interval = GetInterval(n);
37.     for(int i = 0; i < n; ++i)
38.       System.out.println(GetValue(amplitude, period,
    phase, GetT(i, interval)));
39.   }
40. }
```

EXERCISE DRAWWAVE.JAVA

- DrawWave should be able to initialize, draw axis, draw wave

```
1. public class DrawWave {
2.
3.   //Set up X scale, Y scale, and canvas size
4.   public static void Initialize() {
5.   }
6.
7.   //Draw axis
8.   public static void DrawAxis() {
9.   }
18.  /// @brief Draw wave
19.  /// @param interval Interval between points
20.  /// @param wave Wave values
21.  public static void DrawWave(double amplitude, double period, double phase, int n) {
22.  }
23.
24.  public static void main(String args[]) {
25.    //Read in wave parameters
26.
27.    //Draw wave
28.  }
29.}
```

EXERCISE DRAWWAVE.JAVA

- DrawWave should be able to initialize, draw axis, draw wave

```
1.  public class DrawWave {
2.
3.    //Set up X scale, Y scale, and canvas size
4.    public static void Initialize() {
5.      StdDraw.setCanvasSize(500, 500);
6.      StdDraw.setXscale(-11, 11);
7.      StdDraw.setYscale(-11, 11);
8.    }
9.
10.   //Draw axis
11.   public static void DrawAxis() {
12.     StdDraw.setPenColor(StdDraw.RED);
13.     StdDraw.line(-10, 0, 10, 0);
14.     StdDraw.textLeft(9.5, -0.5, "x");
15.     StdDraw.line(0, -10, 0, 10);
16.     StdDraw.textLeft(0.5, 9.5, "y");
17.   }
```

```
18.   /// @brief Draw wave
19.   /// @param interval Interval between points
20.   /// @param wave Wave values
21.   public static void DrawWave(double amplitude, double
      period, double phase, int n) {
22.     StdDraw.setPenColor(StdDraw.BLACK);
23.     double interval = Wave.GetInterval(n);
24.     for(int i = 0; i < n; ++i) {
25.       double t = Wave.GetT(i, interval);
26.       StdDraw.point(t, Wave.GetValue(amplitude, period,
      phase, t));
27.     }
28.   }
29.
30.   public static void main(String args[]) {
15.     Scanner in = new Scanner(System.in);
16.     System.out.println("Enter amplitude period phase n:
      ");
17.     double amplitude = in.nextDouble()
18.     double period = in.nextDouble();
19.     double phase = in.nextDouble();
20.     int n = in.nextInt();
18.
19.     //Draw wave
20.     Initialize();
21.     DrawAxis();
22.     DrawWave(amplitude, period, phase, n);
23.   }
24. }
```

EXERCISES

- Alter the API and implementation of your program to allow for an arbitrary range of data $[min, max]$
  - Need to modify GetInterval, GetT, initialize, draw axis, draw wave
- Allow keyboard input to alter the initial wave parameters
  - Example: ‘w’ increases amplitude, ‘s’ decreases amplitude, etc
- Continually animate the drawing of the wave!
- Augment function to also allow vertical shift by D
- Allow multiple waves