# Methods in Java

# Program Modules in Java

- Modules in Java
  - Methods
  - Classes
- Java API provides several modules
- Programmers can also create modules
  - e.g., programmer-defined methods
- Methods
  - Invoked by a *method call*
  - Returns a result to *calling method* (*caller*)
  - Similar to a boss (caller) asking a worker (called method) to complete a task

# Math-Class Methods

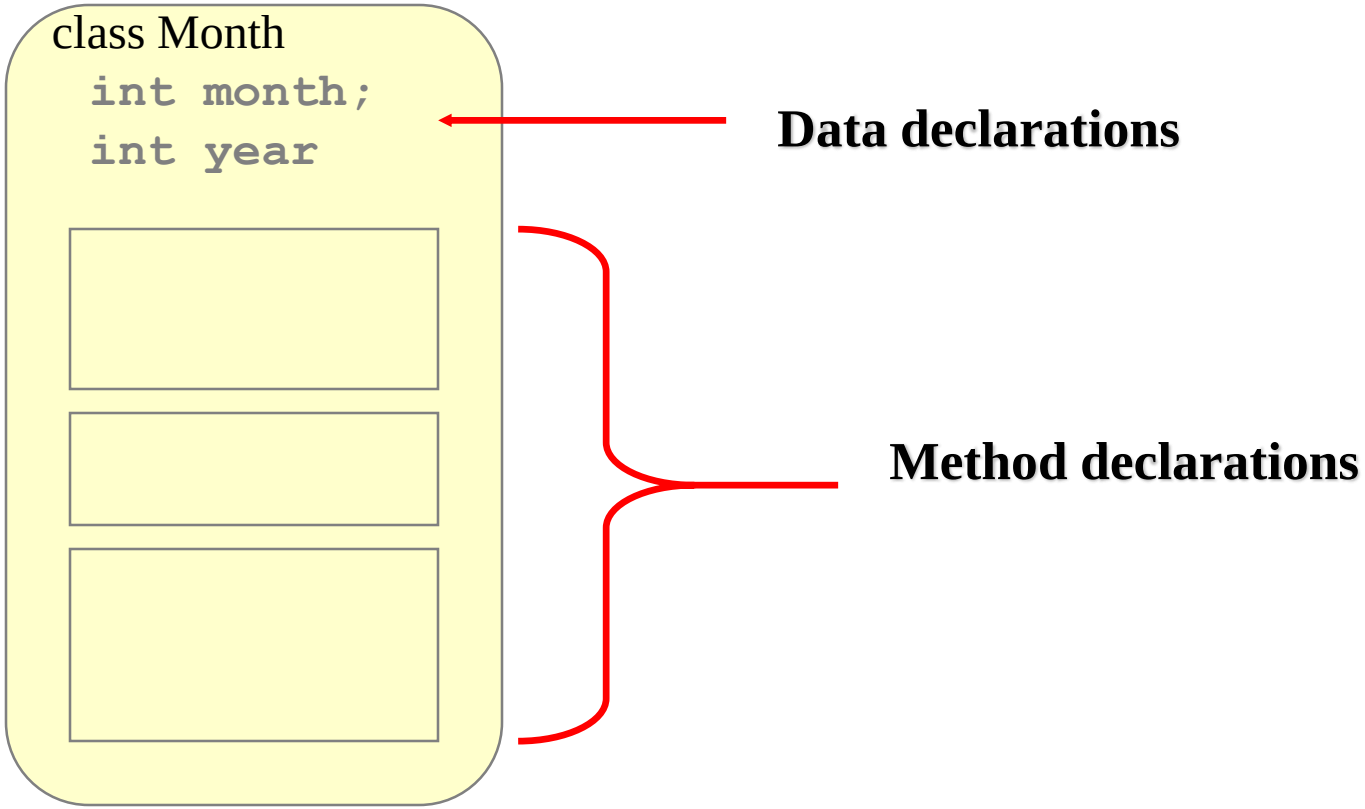
- Class `java.lang.Math`
  - Provides common mathematical calculations
  - Calculate the square root of `900.0`:
    - `Math.sqrt( 900.0 )`
      - Method `sqrt` belongs to class `Math`
        - » Dot (`.`) allows access to method `sqrt`
      - The *argument* `900.0` is located inside parentheses

# Methods Declarations

- Methods
  - Allow programmers to modularize programs
    - Makes program development more manageable
    - Software reusability
    - Avoid repeating code
  - Local variables
    - Declared in method declaration
  - Parameters
    - Communicates information between methods via method calls

# Defining Classes

- A class contains data declarations (static and instance variables) and method declarations (behaviors)



# Methods

- A program that provides some functionality can be long and contains many statements

- A method groups a sequence of statements and should provide a well-defined, easy-to-understand functionality

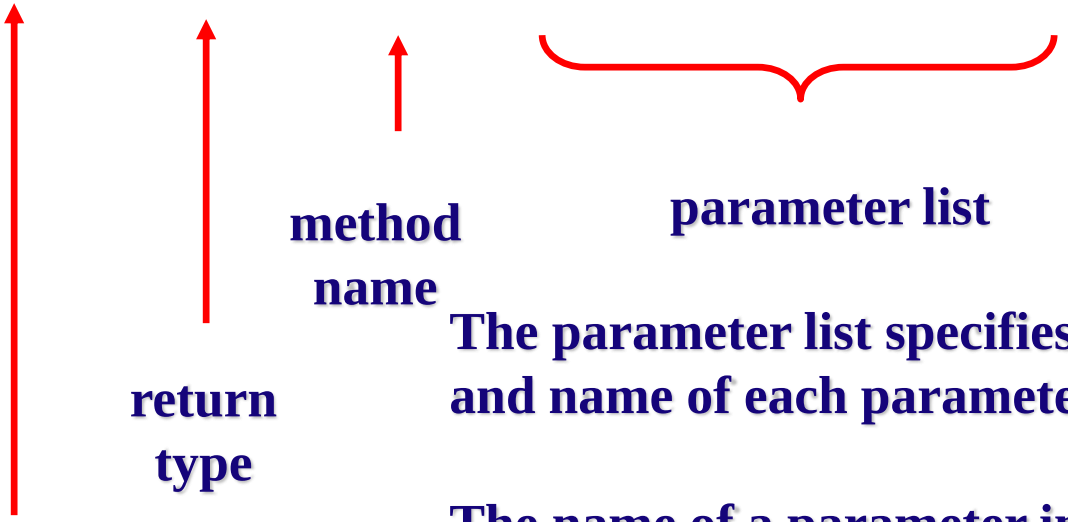
  – a method takes input, performs actions, and produces output

- In Java, each method is defined within specific class

# Method Declaration: Header

- A method declaration begins with a *method header*

```
class MyClass
{  …
   static  int    min ( int num1, int num2 )
```



**properties**

**return type**

**method name**

**parameter list**

**The parameter list specifies the type and name of each parameter**

**The name of a parameter in the method declaration is called a *formal argument***

# Method Declaration: Body

The header is followed by the *method body:*

```
class MyClass
{
    …
    static int min(int num1, int num2)
    {
        int minValue = num1 < num2 ? num1 : num2;
        return minValue;
    }


    …
 }
```

# The `return` Statement

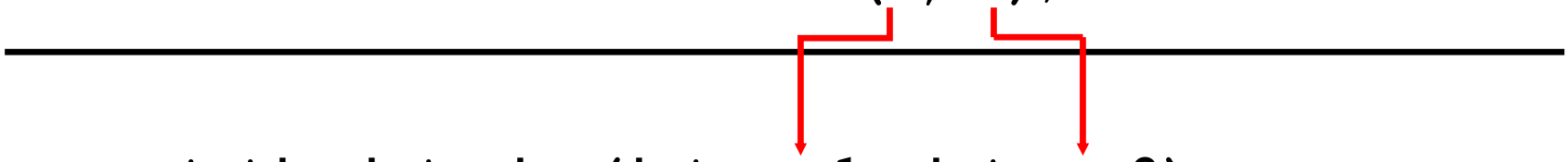
- The return type of a method indicates the type of value that the method sends back to the calling location
  - A method that does not return a value has a `void` return type

- The return statement specifies the value that will be returned
  - Its expression must conform to the return type

# Calling a Method

- Each time a method is called, the values of the *actual arguments* in the invocation are assigned to the *formal arguments*

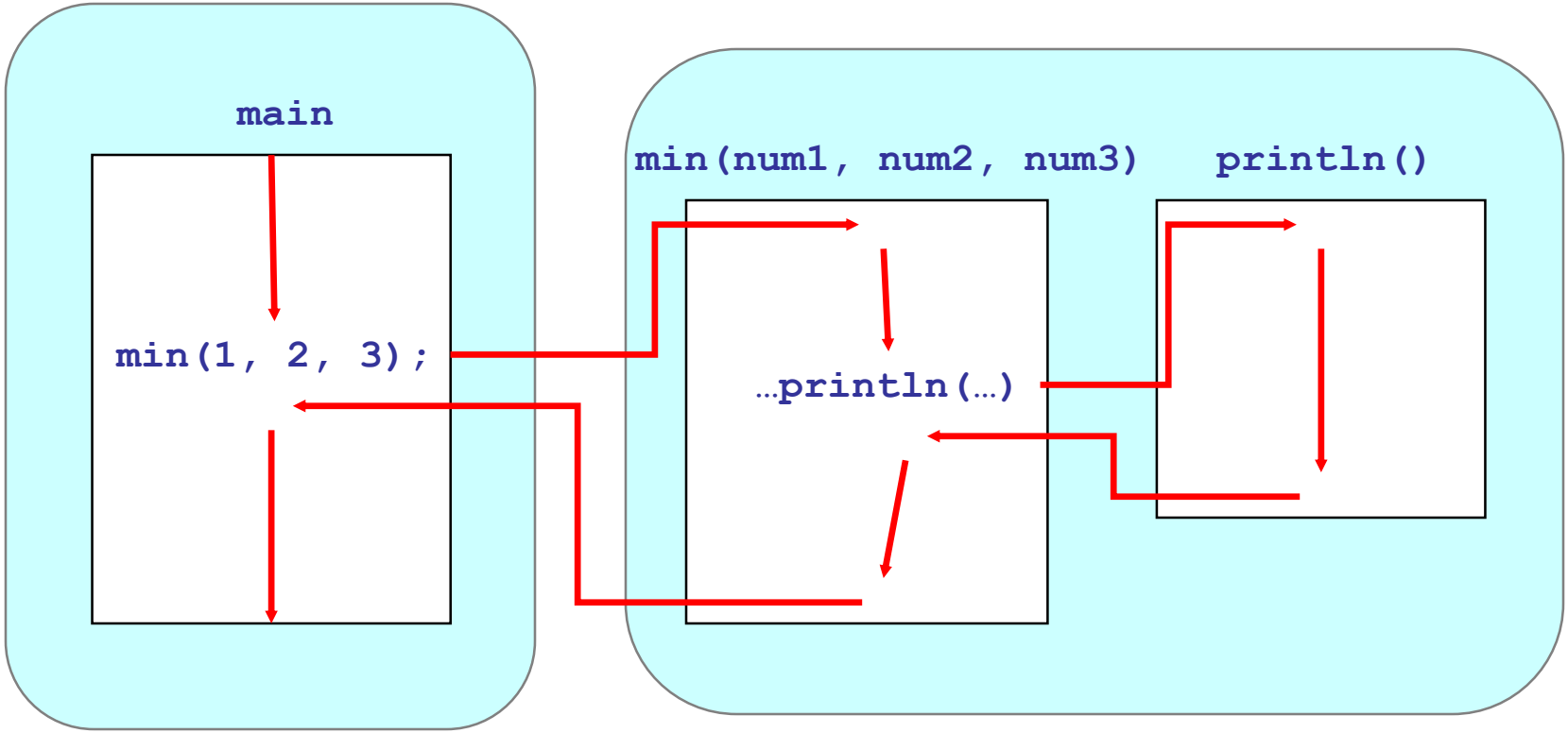
```
int  num = min (2, 3);
```



```
static int min (int num1, int num2)
{
   int minValue = (num1 < num2 ? num1 : num2);
   return minValue;
}
```

# Method Control Flow

- A method can call another method, who can call another method, …



# Method Overloading

- A class may define multiple methods with the same name---this is called method overloading
  - usually perform the same task on different data types

- Example: The PrintStream class defines multiple println methods, i.e., println is overloaded:

  println (String s)

  println (int i)

  println (double d)

  …

- The following lines use the System.out.print method for different data types:

```
System.out.println ("The total is:");
double total = 0;
System.out.println (total);
```

# Widening Primitive Conversions

- Widening primitive conversions are those that do not lose information about the overall magnitude of a numeric value
- Java defines 19 primitive conversions as widening primitive conversions

  byte → short, int, long, float, double

  short → int, long, float, double

  char → int, long, float, double

  int → long, float, double

  long → float, double

  float → double
- They are generally safe because they tend to go from a small data type to a larger one (such as a short to an int)
  - can problems happen in some of the cases?

# Narrowing Primitive Conversions

- Java defines 23 primitive conversions as narrowing primitive conversions

  byte $\rightarrow$ char

  short $\rightarrow$ byte, char

  char $\rightarrow$ byte, short

  int $\rightarrow$ byte, short, char

  long $\rightarrow$ byte, short, char, int

  float $\rightarrow$ byte, short, char, int, long

  double$\rightarrow$ byte, short, char, int, long, float

- Narrowing primitive conversions may lose overall magnitude of a numeric value, or precision

# Method Overloading: Signature

- The compiler must be able to determine which version of the method is being invoked
- This is by analyzing the parameters, which form the *signature* of a method
  - the signature includes the type and order of the parameters
    - if multiple methods match a method call, the compiler picks the best match
    - if none matches exactly but some implicit conversion can be done to match a method, then the method is invoke with implicit conversion (widening).
  - the return type of the method is not part of the signature

# How Do Data Conversions Happen?

- Implicitly: arithmetic (numeric) promotion
  - occurs *automatically* when the operands of a binary arithmetic operator (note "=" is not one) are of different types
    - the promotion uses widening conversion, i.e.,
      - if either operand is double, the other is converted to double
      - otherwise, if either operand is float, the other is converted to float
      - otherwise, if either operand is long, the other is converted to long
      - otherwise, **both operands are converted to int**

Examples:

- 4.0 / 8 (which / is it: double/double, float/float, int/int)
- 4 / 8.0 (which / is it: double/double, float/float, int/int)
- 4 + 5 / 9 + 1.0 + 5 / 9 / 10.0 (what is the value?)

# Method Overloading

**Version 1**

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

**Version 2**

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

**Invocation**

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

# More Examples

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

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

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

Which tryMe will be called?

```
tryMe( 1 );

tryMe( 1.0 );

tryMe( 1.0, 2);

tryMe( 1, 2);

tryMe( 1.0, 2.0); //Error
```

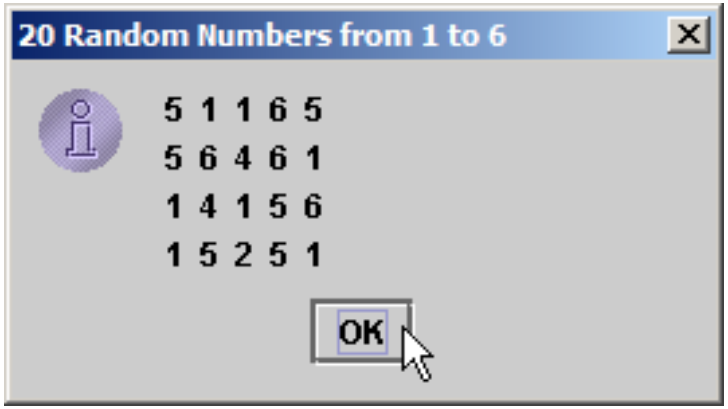
# Java API Packages

- Packages
  - Classes grouped into categories of related classes
  - Promotes software reuse
  - `import` statements specify classes used in Java programs
    - e.g., `import javax.swing.JApplet;`

# Random-Number Generation

- Java random-number generators
  - `Math.random()`
    - `( int ) ( Math.random() * 6 )`
      - Produces integers from 0 - 5
  - Use a *seed* for different random-number sequences

```
import javax.swing.JOptionPane;
public class RandomIntegers {
  public static void main( String args[] )
   {
      int value;
      String output = "";
     for ( int counter = 1; counter <= 20; counter++ ) {
       value = 1 + ( int ) ( Math.random() * 6 );
       output += value + "   ";
       if ( counter % 5 == 0 )
          output += "\n"; }
       JOptionPane.showMessageDialog
            ( null, output,  "20 Random Numbers from 1 to 6",
                          JOptionPane.INFORMATION_MESSAGE );
        System.exit( 0 );      }    }
```



# Variables scoping

- At a given point, the variables that a statement can access are determined by the scoping rule
  - the scope of a variable is the section of a program in which the variable can be accessed (also called visible or in scope)
- There are two types of scopes in Java
  - class scope
    - a variable defined in a class but not in any method
  - block scope
    - a variable defined in a block {} of a method; it is also called a local variable

# Java Scoping Rule

- A variable with a class scope
  - *class/static variable*: a variable defined in class scope and has the static property
    - it is associated with the class
    - and thus can be accessed (in scope) in all methods in the class
  - *instance variable*: a variable defined in class scope but not static
    - it is associated with an instance of an object of the class,
    - and thus can be accessed (in scope) only in instance methods, i.e., those non-static methods

- A variable with a block scope
  - can be accessed in the enclosing block; also called local variable
  - a local variable can shadow a variable in a class scope with the same name
- Do not confuse scope with duration
  - a variable may exist but is not accessible in a method,
    - e.g., method A calls method B, then the variables declared in method A exist but are not accessible in B.

# Scoping Rules (cont.): Variables in a method

- There are can be three types of variables accessible in a *method* :
  - class and instance variables
    - static and instance variables of the class
  - local variables
    - those declared in the method
  - formal arguments

# Example1:

```
public class Box
{
   private int length, width;
   …

   public int widen (int extra_width)
   {
        private int temp1;
        size += extra_width;
        …
   }
   public int lenghten (int extra_lenth)
   {
        private int temp2;
        size += extra_length;
        …
   }

   …

}
```

- instance variables
- formal arguments
- local variables

# Scope of Variables

```
public class Box
{
   private int length, width;
   …

   public int widen (int extra_width)
   {
        private int temp1;
        size += extra_width;
        …
   }
   public int lenghten (int extra_lenth)
   {
        private int temp2;
        size += extra_length;
        …
   }

   …

}
```

- Instance variables are accessible in all methods of the class
- formal arguments are valid within their methods
- Local variables are valid from the point of declaration to the end of the enclosing block

```
public class Test
{
    final static int NO_OF_TRIES = 3;
    static int i = 100;
    public static int square ( int x )
    {
        // NO_OF_TRIES, x, i in scope
        int mySquare = x * x;
        // NO_OF_TRIES, x, i, mySquare in scope
        return mySquare;
    }
    public static int askForAPositiveNumber ( int x )
    {
        // NO_OF_TRIES, x, i in scope
        for ( int i = 0; i < NO_OF_TRIES; i++ )
        {  // NO_OF_TRIES, x, i in scope; local i shadows class i
            System.out.print("Input: ");
            Scanner scan = new Scanner( System.in );
            String str = scan.nextLine();
            int temp = Integer.parseInt( str );
            // NO_OF_TRIES, x, i, scan, str, temp in scope
            if (temp > 0) return temp;
        }
        // NO_OF_TRIES, x, i in scope
        return 0;
    } // askForPositiveNumber
    public static void main( String[] args )
    {…}
```

# Two Types of Parameter Passing

- If a modification of the *formal argument* has no effect on the *actual argument*,
  - it is call by value
- If a modification of the *formal argument* can change the value of the *actual argument*,
  - it is call by reference

# Call-By-Value and Call-By-Reference in Java

- Depend on the type of the formal argument
- If a formal argument is a primitive data type, a modification on the formal argument has no effect on the actual argument
  - this is call by value, e.g. **`num1 = min(2, 3);`**
    **`num2 = min(x, y);`**
- This is because primitive data types variables *contain their values*, and procedure call trigger an assignment:

  <formal argument> = <actual argument>

```
int x = 2; int y = 3;
int num = min (x, y);
…
static int num( int num1, int num2)
{ … }
```

```
int x = 2;
int y = 3;
int num1 = x;
int num2 = y;
{ … }
```

# Call-By-Value and Call-By-Reference in Java

- If a formal argument is not a primitive data type, an operation on the formal argument can change the actual argument
  - this is call by reference
- This is because variables of object type *contain* ***pointers*** to the data that represents the object
- Since procedure call triggers an assignment

  <formal argument> = <actual argument>

  it is the pointer that is copied, not the object itself!

```
MyClass x = new MyClass();
MyClass y = new MyClass();
MyClass.swap( x, y);
…
static void swap( MyClass x1, MyClass x2)
{ … }
```

```
x = new MC();
y = new MC();
x1 = x;
x2 = y;
{ … }
```

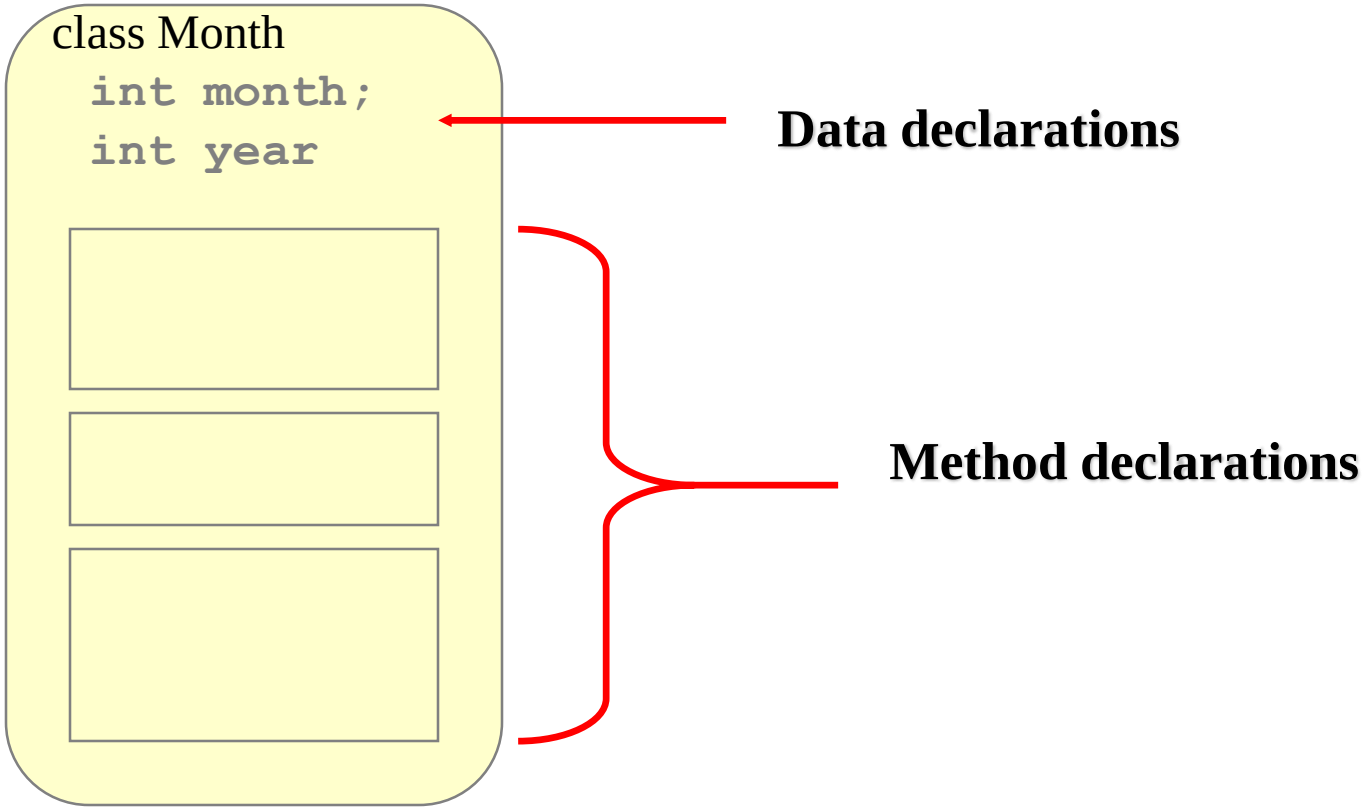
# Classes define Objects

- In object-oriented design, we group methods together according to objects on which they operate
- An object has:
  - *state* - descriptive characteristics
  - *behaviors* - what it can do (or be done to it), may depend on the state, and can change the state

- For example, a calendar program needs Month objects:
  - the state of a Month object is a (month,year) pair of numbers
  - these are stored as *instance variables* of the Month class
  - the Month class can also have *class variables,* e.g. the day of the week that Jan 1, 2000 falls on…
  - some behaviors of a month object are:
    - get name, get first weekday, get number of days,
    - print
    - set month index, set year

# Defining Classes

- A class contains data declarations (state) and method declarations (behaviors)



# Method Types

- There can be various types of methods (behavior declarations)
  - access methods : read or display states (or those that can be derived)
  - predicate methods : test the truth of some conditions
  - action methods, e.g., print
  - constructors: a special type of methods
    - they have the same name as the class
      - there may be more then one constructor per class (overloaded constructors)
    - they do not return any value
      - it has no return type, not even void
    - they initialize objects of the class, using the new construct:
      - e.g. m1 = new Month();
    - you do not have to define a constructor
      - the value of the state variables have default value