Methods in Java

Methods

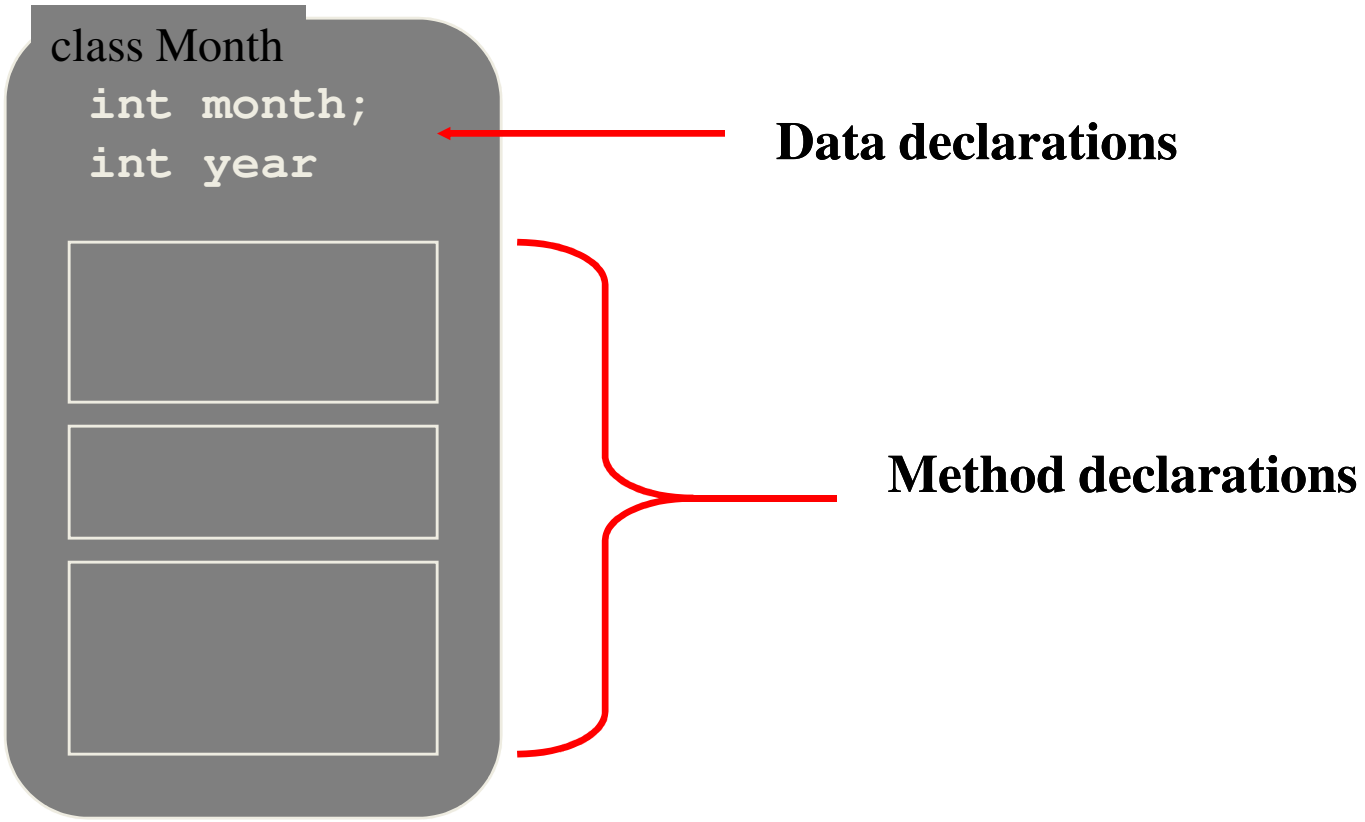
- A program that provides some functionality can be long and contains many statements

- 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

Methods

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

General method declarations

- Header

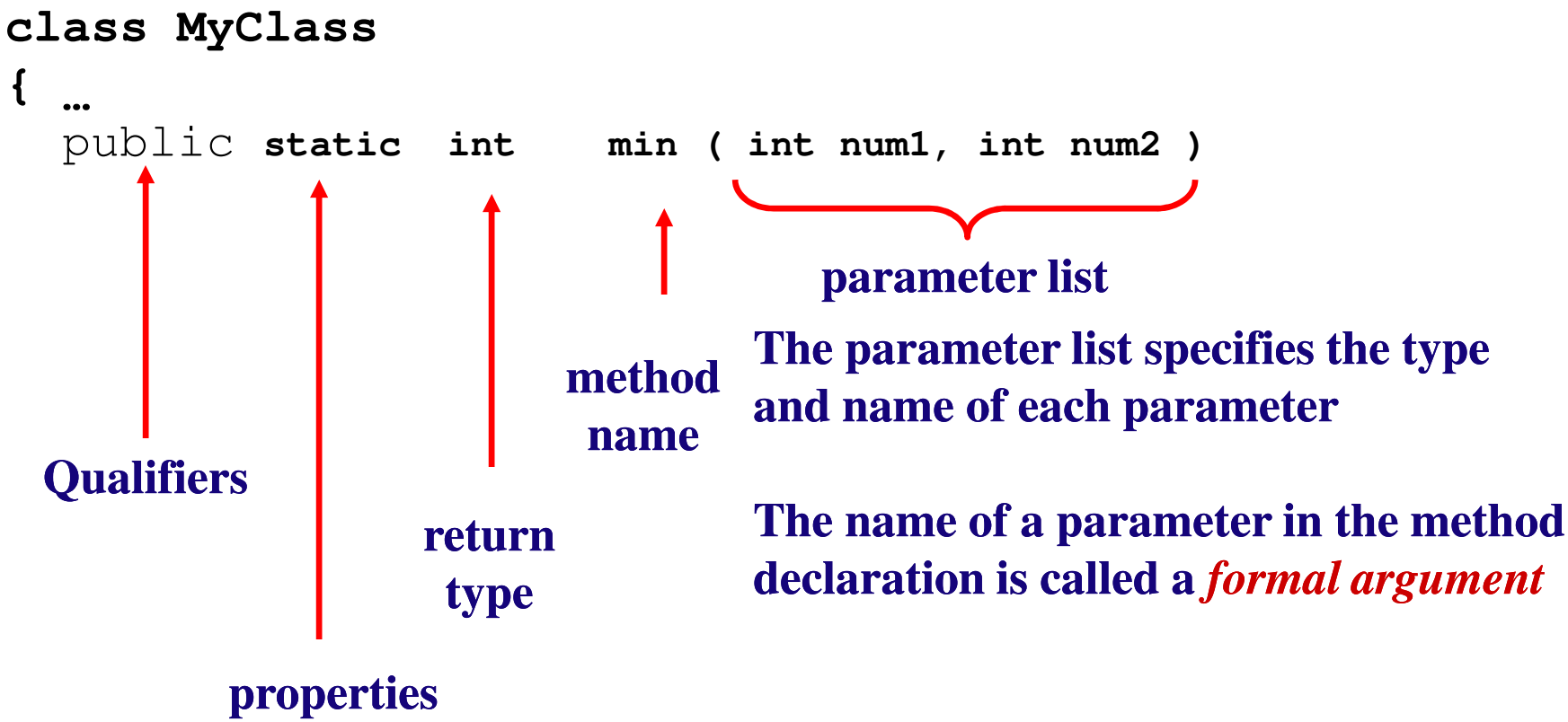
```
qualifiers properties return-type
   method-name ( formal-parameters )
   throws-clause
```

- Body

```
{
    Statements;
}
```

Method Declaration: Header

- A method declaration begins with a *method header*



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

- A method can be called by object of the class. Using dot operator.
  - objectName.methodName(parameter list);
- A static method can be called directly independent of object.
  - ClassName.methodName(parameter list);
- Within a class method can be called directly
  - mehtodName(parameter list);

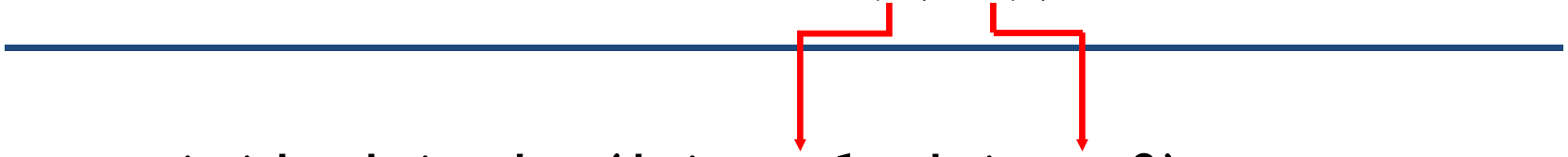
Multiple invocations or calling

- At any given point in time, a single method may have been invoked through multiple locations in a program.
- Every invocation gets its own copy of the methods local variables (including the formal parameters)
- A method may even call invoke itself. This is called *recursion*.

Calling a Method

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

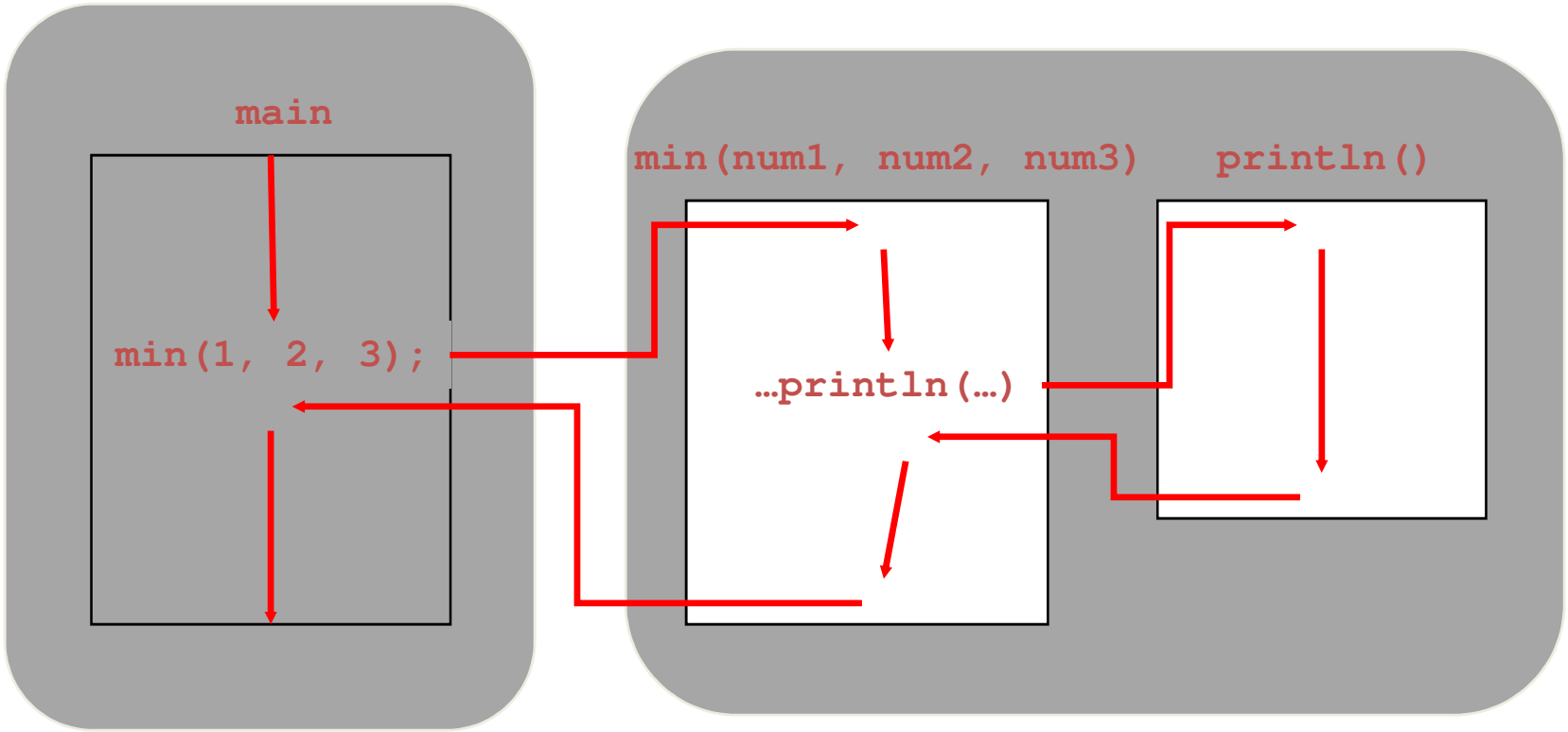
```
int  num = min (2, 3);
```



```
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 inputs

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

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.
  - the return type of the method is not part of the signature

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)
```

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;
{ … }
```

THANK YOU