Methods

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria

Wolfgang.Schreiner@risc.jku.at
http://www.risc.jku.at

Contents

Method Declarations

A method is a named piece of code that performs a certain task.

```
public static void name()
{
   ...
}
```

- Method *name*
  - Can be used in method calls.
- Method body
  - Instructions to be executed when method is called.

Unit of program development.

Method Calls

When a method is called, its body is executed.

```
name();
```

- Executes the body of method *name*.
- Continues execution with statement after the call.

Do not need to know how the method is implemented.

Example

```
public class Main
{
  public static void main(String[] args)
  {
    printHeader();
    System.out.println(1);
    printHeader();
    System.out.println(2);
  }
  public static void printHeader()
  {
     System.out.println("Number:");
     System.out.println("-------");
  }
}
```

```
Number:
-------
1
Number:
-------
2
```

Source Code Conventions

- Method naming:
  - A method performs an activity.
    - The method prints a header.
  - A method call should read like a command to perform the activity.
    - "Print the header!".
    - `printHeader`
- Method ordering:
  - Backward calls: declare all methods before they are called.
  - Forward calls: declare all methods after they are called.
  - Organize your source code in either way.
- Method header:
  - Write a comment at the beginning of the method that explains what the method does.
  - Will be discussed in more detail later.

Calling Methods from other Classes

Is a program always a single class?

- A program may consist of multiple classes:
  - Class *C* is stored in file *C*.`java` which is compiled to *C*.`class`.
  - Each class can contain multiple method definitions.
  - One class has the method `main`.
- Calling a method in another class:

  Name.*name*();

  - Class *Name*.
  - Method *name* in class *Name*.
  - Any method can call any method in any class.

Large programs are decomposed into multiple classes.

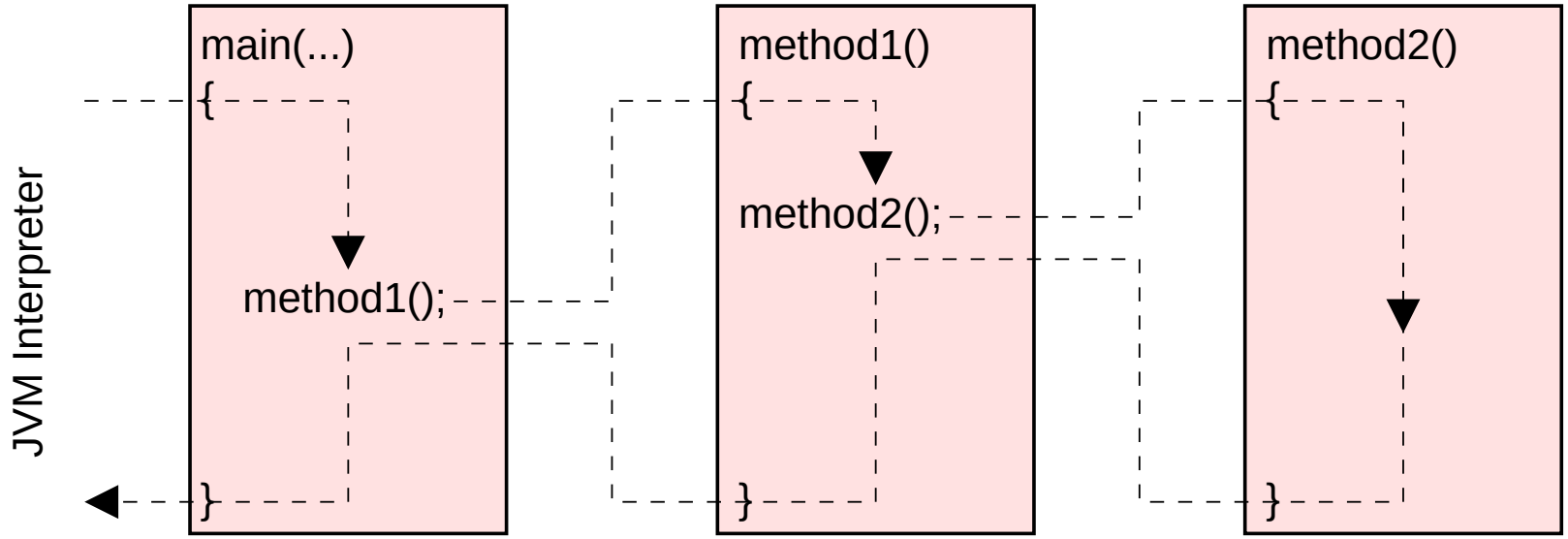
Example

```
public class Main                                   // file Main.java
{
  public static void main(String[] args)
  {
    Print.printHeader();
    System.out.println(1);
    Print.printHeader();
    System.out.println(2);
  }
}

public class Print                                  // file Print.java
{
  public static void printHeader()
  {
     System.out.println("Number:");
     System.out.println("-------");
  }
}
```

Chains of Method Calls

A method may call any other method.



Chain of method calls and returns.

The `return` Statement

```
return;
```

- Terminates current method and returns control to caller.
- If executed in `main`, returns control to JVM interpreter.
  - Same effect as `System.exit(0)`.

We may return control to caller before method has ended.

Example

```
public static void readPrintInt()
{
   System.out.print("Enter integer: ");
   int i = Input.readInt();
   if (!Input.isOkay())
   {
     System.out.println("Invalid input!");
     Input.clearError();
     return;
   }
   System.out.println(i);
}
```

Premature return to caller in case of error.

Example

```
public static void main(String[] args)
{
  for (int i = 0; i < 3; i++)
    readPrintInt();
  System.out.println("Done.");
}

Enter integer: 9
9
Enter integer: f
Invalid input!
Enter integer: -1
-1
Done.
```

Local Variables

If a variable is declared in a method, it is not visible to other methods.

```
public static void main(String[] args)
{
   int i = 1;
   method1();
   System.out.println(i);
}

public static void method1()
{
   int i = 2;
   System.out.println(i);
}
```

```
2
1
```

Same variable name in different methods denotes different variables!

Example

```
public static void main(String[] args)
{
   int i = 1;
   method1();
   System.out.println(i);
}
public static void method1()
{
   System.out.println(i);
}
```

```
Main.java:11: cannot resolve symbol
symbol  : variable i
location: public class Main
       System.out.println(i);
                          ^
1 error
```

Global Constant

How can we use a constant in multiple methods?

- Declare constant on the class level.
  - Add keyword `public static` to declaration (as for methods).

```
public class Main
{
  public static final int LENGTH = 10;
  ...
}
```

- Constant is visible to all methods with the class.
  - Constant is global to these methods.

A constant declared on the class level is global.

Example

```
public class Main
{
  public static final int LENGTH = 10;

  public static void main(String[] args) {
    print2TimesN();
    print3TimesN();
  }
  public static void print2TimesN() {
    System.out.println(2*LENGTH);
  }
  public static void print3TimesN() {
    System.out.println(3*LENGTH);
  }
}
```

```
20
30
```

Global Variables

Also variables may be declared globally.

```
public class Main
{
  public static int sum = 0;
  public static void main(String[] args)
  {
    add1(); multiply2(); add1();
    System.out.println(sum);
  }
  public static void add1() {
    sum = sum+1;
  }
  public static void multiply2() {
    sum = sum*2;
  }
}
```

```
Output:
3
```

Global Variables and Initialization

If global variable is not initialized, it receives a default value.

```
public class Main
{
  public static int i;
  public static void main(String[] args)
  {
     System.out.println(i);
  }
}
```

```
0
```

Default value is 0 for integer types, 0.0 for floating point types.

Parameters

A method may be parameterized.

```
public static void name(parameter1, parameter2, ...)
{
  ...
}
```

- Method head contains sequence of parameter declarations:

  type name
  `final` *type name*

  – Parameter *name*.
  – May be used in method like a local variable/constant.

A parameterized method may receive values from the caller.

Method Calls

The caller provides values for the method parameters.

```
name(expression1, expression2, ...)
```

- Method call contains sequence of arguments.
  - Arguments are expressions whose values are assigned to the parameters.
- A method call is executed by
  1. evaluating the arguments,
  2. assigning the argument values to the parameters of the method, and
  3. executing the body of the method.

The method parameters receive copies of the argument values.

Example

```
public static void main(String[] args)
{
  int x = 3;
  printSquare("this", x);
  printSquare("that", x+1);
}

public static void printSquare(String key, int val)
{
  System.out.println(key + ": " + val*val);
}
```

```
this: 9
that: 16
```

Example

```
public static void main(String[] args)
{
  int i=1;
  increment(i, 2);
  System.out.println("caller: " + i);
}

public static void increment(int i, int n)
{
  i += n;
  System.out.println("method: " + i);
}
```

```
method: 3
caller: 1
```

A modification of the parameter does not affect the argument!

Example

```
public static void main(String[] args)
{
   int x = 3; int y = 4;
   System.out.print("x = " + x + ", y = " + y);
   System.out.print(", (x+y)*(x+y) = ");
   System.out.println((x+y)*(x+y));
   int a = 5; int b = 6;
   System.out.print("a = " + a + ", b = " + b);
   System.out.print(", (a+b)*(a+b) = ");
   System.out.println((a+b)*(a+b));
}
```

```
x = 3, y = 4, (x+y)*(x+y) = 49
a = 5, b = 6, (a+b)*(a+b) = 121
```

Use parameterized methods rather than duplicating code!

Example

```
public static void main(String[] args) {
   printSumSquare("x", 3, "y", 4);
   printSumSquare("a", 5, "b", 6);
}

public static void
printSumSquare(String var1, int val1, String var2, int val2) {
   System.out.print(var1 + " = " + val1 + ", ");
   System.out.print(var2 + " = " + val2 + ", ");
   System.out.print("(" + var1 + "+" + var2 + ")*");
   System.out.print("(" + var1 + "+" + var2 + ") = ");
   System.out.println((val1+val2)*(val1+val2));
}
```

Extract reusable code into methods.

Functions

A function is a method that returns a value.

```
public static type name(parameters)
{
  ...
}
```

- A function has a particular return type.
  - A procedure is a method that does not return a value (return type `void`).
- A function must return a value of this type to the caller.

  ```
  return expression;
  ```

  - The last executed statement of function must be such a statement.
  - The value of *expression* is the value returned to the caller.
  - The type of *expression* must match the return type.

Function Application

A function application is not a statement but an expression.

```
name(arguments)
```

- A function application is an expression of the function's return type.
- Application is evaluated by invoking the corresponding function.
- The value returned by the function is the value of the application.

A function application returns a value.

Example

```
public static void main(String[] args) {
  int n = 3;
  int r = exp(n, 2);
  System.out.println(r);
  System.out.println(exp(n, 3));
  if (exp(n, 4) > 80)
    System.out.println("Yes.");
}

public static int exp(int a, int b) {
  int r = 1;
  for (int i = 0; i < b; i++)
    r = r*a;
  return r;
}
```

```
Output:
9
27
Yes.
```

Program Functions versus Mathematical Functions

There are two kinds of program functions.

- Functions that are procedures with return value.
  - Result depends on a global variable or execution causes a side effect.
  - Input/output, modification of a global variable.
- Functions that behave like mathematical functions.
  - Result is only determined by function arguments (and global constants).
  - Execution does not cause a side effect.
  - Example: `exp` (previous slide).

Both kinds have same syntax but let us keep them apart.

Procedures with Return Value

```
public static void main(String[] args) {
  while (true)
  {
    int n = askForNat();
    if (n < 0) break;
    System.out.println(n);
  }
}

public static int askForNat() {
   System.out.print("Enter natural: ");
   int i = Input.readInt();
   if (Input.isOkay() && i >= 0)
     return i;
   return -1;
}
```

Invoking Program Functions

We expect that an expression does not cause a side effect.

- Application of a function that behaves like a mathematical function:

```
... name(arguments) + expression ...
```

  – Application may occur in the context of another expression.

- Application of a procedure with return value:

```
type var = name(arguments);
var = name(arguments);
```

  – Application should occur only in an declaration or assignment.

The application of a procedure with return value should not occur within another expression.

Naming Program Functions

- Procedure with return value:
  - Application performs an activity (resulting in a value).
  - Name should express the activity.

    ```
    int i = Input.readInt();
    ```

  - "i is the result of reading an integer from the input stream".
- Program function that behaves like a mathematical function:
  - Application returns a result.
  - Name should describe the result.

    ```
    exp(a, b)
    ```

  - the result of the exponentiation of a by b.

Heuristics to choose good function names.

Example: Random Number Generation

```
public class Main {
   public static void main(String[] args)
   {
     for (int i=0; i < 1000; i++)
     {
       int r = Random.nextRandom(128);
       System.out.println(r);
     }
   }
}
```

```
26
18
38
101
...
```

`nextRandom` is a procedure with return value.

Example: Random Number Generation

```
public class Random
{
  public static int old = 314152;

  public static int nextRandom(int n)
  {
    int x = old;
    int high = x/127773;
    int low = x%127773;
    int val = 16807*low - 2836*high;
    if (val <= 0) val += Integer.MAX_VALUE;
    old = val;
    return val%n;
  }
}
```

`nextRandom` has a hidden state on which it depends.

Visibility of Variables

We have encountered various types of variables/constants.

- Variables/constants declared in the body of a class;
- Paremeters declared in a head of a method;
- Variables/constants declared in the body of a method.
- Variables/constants declared in a block.

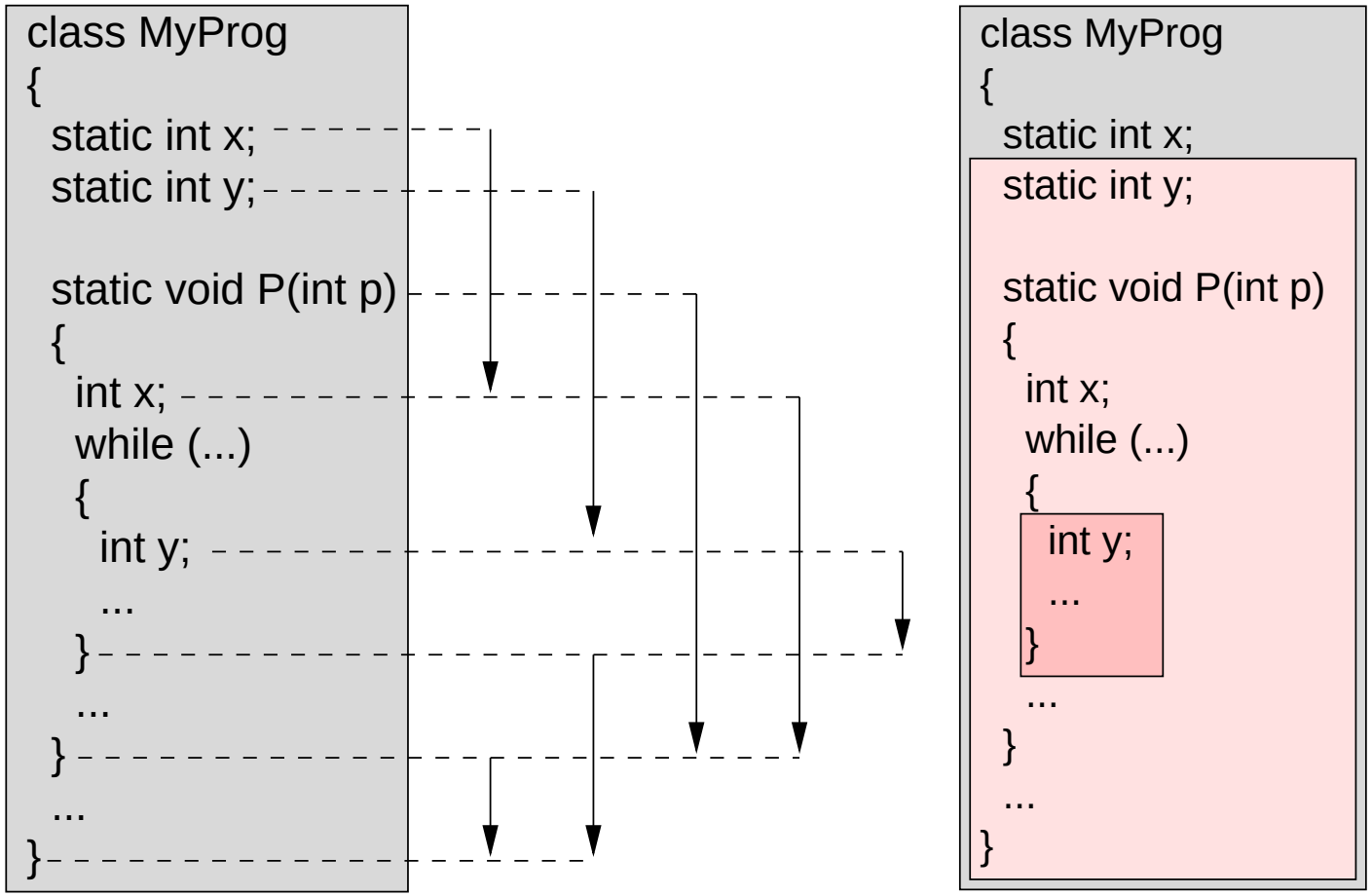
At which program position is which variable visible?

Visibility of Variables

1. The visibility range of a variable/constant is from the point of its declaration to end of the entity (class or method or block) in which it was declared unless
2. there is an inner declaration of a variable/constant of the same name; in this case, the inner declaration shadows the outer declaration, i.e., the variable of the outer declaration is not visible.

These two rules determine the visibility.

Example


class MyProg
{
static int x;
static int y;
static void P(int p)
{
int x;
while (...)
{
int y;
...
}
...
}
...
}
class MyProg
{
static int x;
static int y;
static void P(int p)
{
int x;
while (...)
{
int y;
...
}
...
}
...
}

Method Overloading

Generally all methods in a class must have different names.

- Exception: different parameter lists.
  - Number of parameters or their types are different.
- Method name is overloaded with different methods.
  - `public static void printLine(int i)`
  - `public static void printLine(String s)`
- On application, method is chosen from argument types:

```
printLine("hi");
```

It is a good strategy to give methods the same name if they have similar functionality.

Documenting Methods

Comment header of method should describe methods such that it can be used without looking into the body of the method.

- Input condition
  - What condition must the method parameters (and the relevant global variables) satisfy such that it is legal to call the method?
  - It is the responsibility of the caller to make sure that this condition holds.
- Output condition
  - What condition do the method's return value (and the values of the modified global variables) satisfy after the call?
  - It is the responsibility of the method to make sure that this condition holds.
- External effect
  - If the method performs input/output, also document the external effect of the method.

Example

```
// ------------------------------------------------------------
// r = method(input, transient)
// One sentence that explains how above statement is to be read.
//
// Input condition:
//   A statement involving the 'input' parameters and the
//   'transient' parameters and the used global variables.
// Output condition:
//   A statement involving the return value 'r' and the
//   'transient' parameters and the used global variables.
// Effect:
//   A statement describing the external effect of the method.
// ------------------------------------------------------------
public static type method(type input, type transient)
```

Example

```
// ------------------------------------------------------
// q = div(m, n)
// 'q' becomes the truncated quotient of the natural numbers
// 'm' and 'n'.
//
// Input condition:
//   'm' is not negative, 'n' is positive.
// Output condition:
//   there exists a remainder 'r' less than 'n' such that
//   'm = n*q + r'.
// ------------------------------------------------------
public static int div(int m, int n)
```

Example

```
// ------------------------------------------------------------
// p = position(s, c)
// 'p' is the index of the first occurrence of character 'c'
// in string 's'.
//
// Input condition:
//   's' is not null.
// Output condition:
//   'p' is greater than or equal to -1 and less than
//     the length of 's'.
//   If 'p' is equal to -1, then 'c' does not occur in 's'.
//   If 'p' is greater than -1, then
//     * 's' has at position 'p' character 'c' and
//     * 's' does not have at any position less than 'p'
//       character 'c'.
// ------------------------------------------------------------
public static int position(String s, char c)
```

Example

```
// ------------------------------------------------------------
// t = cut(s, n)
// t is copy of string 's' cut at position 'n'
//
// Input condition:
//   's' is not null, 'n' is not negative.
// Output condition:
//   let 'l' be the length of 's'.
//   if 'n' is greater than or equal 'l',
//   then 't' is a copy of 's'.
//   otherwise, 't' has length 'n' and has in all positions
//   the same characters as 's'
// ------------------------------------------------------------
public static String cut(String s, int n)
```

Example

```
// -------------------------------------------------------
// n = readNat()
// read natural number 'n' from the input stream.
//
// Output condition:
//   'n' is greater than or equal to -1.
// Effect:
//   Asks the user for a non-negative integer number.
//   If such a number is read, it is returned as 'n'.
//   If the input stream has ended, 'n' is -1.
//   Otherwise, the process is repeated.
// -------------------------------------------------------
public static int readNat()
```