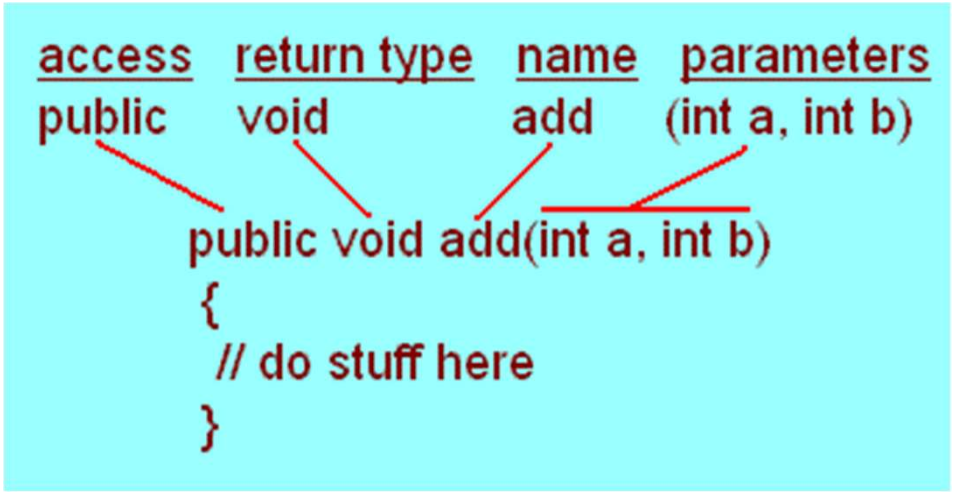
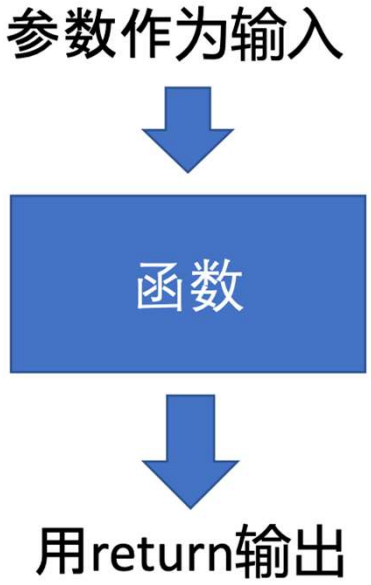
LightHouseIT

# 第5章
# メソッド

# Java 方法[メソッド]（method）

- Java方法[メソッド]是一个代码块，它只有被调用[呼び出す]（invoke，call）的时候才会被运行。
- 调用方法的时候需要传入这个方法需要的参数[引数]。可以将调用的方法[メソッド]返回的结果赋值给某个变量。
- 方法[メソッド]用于执行某些动作，它们也被称为函数（function）。



# 为什么需要方法[メソッド]

- 比如利用到目前为止学到的java语法计算2^3 + 5^4 + 8^5这个式子的结果

```
public static void main(String[] args) {
    int a = 2*2*2 + 5*5*5*5 + 8*8*8*8*8;
    System.out.println(a);
}
```

- 这样算完全没问题，可以得到正确结果
- 但是有很多重复的符号，写起来也很容易出错
- 而且如果幂数更高的话，要写得更长更久。

# 为什么需要方法[メソッド]

- 我们定义一个方法[メソッド]名字叫做power（乘方）
- 它有两个参数[引数]a和b，a表示底数，b表示幂数
- 这个方法[メソッド]返回的结果为a^b。

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

- 然后调用[呼び出す]这个方法[メソッド]再次计算2^3 + 5^4 + 8^5

```
public static void main(String[] args) {
    int a = power(2, 3) + power(5,4) + power(8, 5);
    System.out.println(a);
}
```

- 和刚刚的比对一下，方法[メソッド]就显得十分必要了

# 创建一个方法[メソッド]

- 方法[メソッド]必须在类[クラス]（class）中声明。用方法[メソッド]名称定义，后跟括号（）。Java提供了一些定义好了的方法[メソッド]，例如System.out.println()，但是你也可以创建自己的方法[メソッド]来执行某些操作：

```
public class MyClass {
  static void myMethod() {
    // code to be executed
  }
}
```

- myMethod() 是方法[メソッド]的名称
- static表示该方法[メソッド]属于MyClass类[クラス]，而不是MyClass类[クラス]的对象[オブジェクト]。之后会具体讲类[クラス]和对象[オブジェクト]。
- void表示此方法[メソッド]没有返回值[返り値]。

# 调用方法

```java
public class MyClass {
  static void myMethod() {
    System.out.println("I just got executed!");
  }

  public static void main(String[] args) {
    myMethod();
    myMethod();
    myMethod();
  }
}

// I just got executed!
// I just got executed!
// I just got executed!
```

# Java 方法[メソッド]参数[引数]

- 信息可以作为参数[引数]传递给方法[メソッド]。 参数[引数]在方法[メソッド]内部充当变量。
- 在方法[メソッド]名称后的括号内指定参数[引数]。 可以根据需要添加任意数量的参数[引数]，只需用逗号分隔即可。
- 添加参数[引数]需要定义参数[引数]的类型，参数[引数]的名称。
- 下面的示例包含一个使用String名为fname的参数[引数]的方法[メソッド]。 调用该方法[メソッド]时，要传递一个名字，该名字在方法[メソッド]内部用于打印全名：

```
public class MyClass {
  static void myMethod(String fname) {
    System.out.println(fname + " Refsnes");
  }

  public static void main(String[] args) {
    myMethod("Liam");
    myMethod("Jenny");
    myMethod("Anja");
  }
}
// Liam Refsnes
// Jenny Refsnes
// Anja Refsnes
```

当参数（parameter）传递给方法时，称为argument。因此，从上面的例子：fname是一个parameter，同时Liam，Jenny和Anja是argument。但这两者都翻译成参数。

# 多个参数[引数]

- 可以根据需要选择任意数量的参数[引数]：

```
public class MyClass {
  static void myMethod(String fname, int age) {
    System.out.println(fname + " is " + age);
  }

  public static void main(String[] args) {
    myMethod("Liam", 5);
    myMethod("Jenny", 8);
    myMethod("Anja", 31);
  }
}

// Liam is 5
// Jenny is 8
// Anja is 31
```

- 请注意，当使用多个参数[引数]时，方法[メソッド]调用[呼び出す]必须具有与parameters相同数量的arguments，并且必须以相同的顺序传递arguments。

# 返回值[返り値] Return

- 方法[メソッド]的返回值[返り値]就是你希望这个方法[メソッド]运行完返回到调用处的一个结果。
- 在之前示例中使用的关键字void表示该方法[メソッド]没有返回值[返り値]。如果希望方法[メソッド]返回一个值，则可以使用类型（如int， char，String等）代替void，并使用return关键字：

```
public class MyClass {
  static int myMethod(int x) {
    return 5 + x;
  }

  public static void main(String[] args) {
    System.out.println(myMethod(3));
  }
}
// Outputs 8 (5 + 3)
```

# 返回值[返り値] Return

- 还可以将返回结果存储在变量中（推荐，因为它更易于阅读和维护）：

```
public class MyClass {
  static int myMethod(int x, int y) {
    return x + y;
  }

  public static void main(String[] args) {
    int z = myMethod(5, 3);
    System.out.println(z);
  }
}
// Outputs 8 (5 + 3)
```

# 练习时间

- 写一个用来判断一个整数奇偶性的方法[メソッド]
- 方法名：checkParity
- 参数[引数]：类型：int 名称：num
- 返回值[返り値]：类型：String 值：“even” or “odd”
- 方法[メソッド]调用[呼び出す]：

```
public static void main(String[] args) {
    System.out.println(checkParity(2435));
    // Outputs "odd"
}
```

# Java 方法[メソッド]重载[オーバーロード]（Method Overload）

- 通过方法[メソッド]重载[オーバーロード]，多个方法[メソッド]可以使用不同的参数[引数]，相同的名称：

```
int myMethod(int x)
float myMethod(float x)
double myMethod(double x, double y)
```

# 为什么需要方法[メソッド]重载[オーバーロード]

- 考虑下面的示例，该示例有两种添加不同类型数字的方法[メソッド]：

```
static int plusMethodInt(int x, int y) {
  return x + y;
}

static double plusMethodDouble(double x, double y) {
  return x + y;
}

public static void main(String[] args) {
  int myNum1 = plusMethodInt(8, 5);
  double myNum2 = plusMethodDouble(4.3, 6.26);
  System.out.println("int: " + myNum1);
  System.out.println("double: " + myNum2);
}
```

# 为什么需要方法[メソッド]重载[オーバーロード]

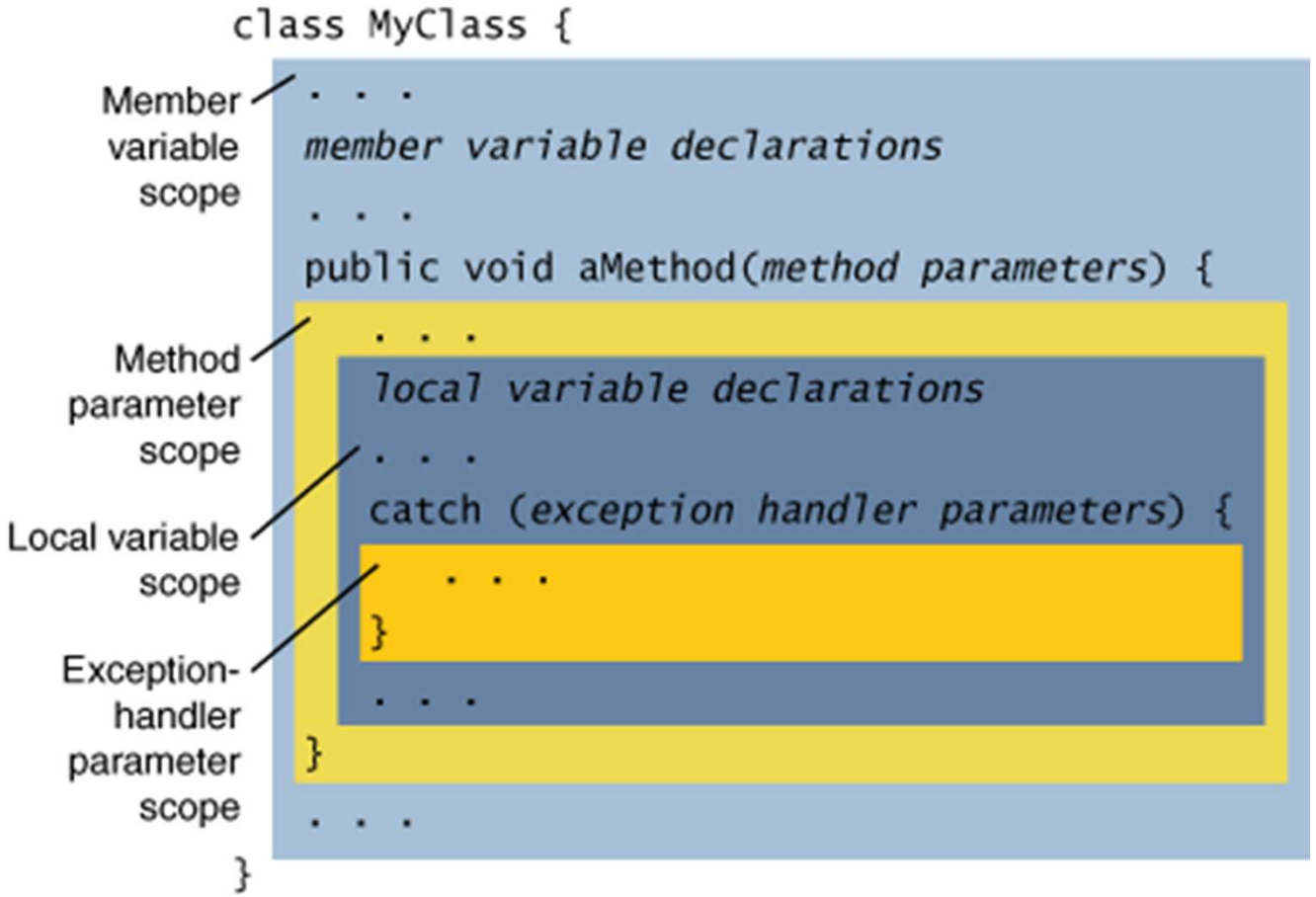
- 与其定义两个做同样事情的方法[メソッド]，不如重载[オーバーロード]一个方法[メソッド]。
- 在下面的示例中，我们重载[オーバーロード]了同时适用于int和double的plusMethod方法[メソッド]：

```
static int plusMethod(int x, int y) {
  return x + y;
}

static double plusMethod(double x, double y) {
  return x + y;
}

public static void main(String[] args) {
  int myNum1 = plusMethod(8, 5);
  double myNum2 = plusMethod(4.3, 6.26);
  System.out.println("int: " + myNum1);
  System.out.println("double: " + myNum2);
}
```

- 注意：只要参数[引数]的数量和/或类型不同，多个方法[メソッド]就可以具有相同的名称。

# Java 作用域[スコープ]（Scope）

- 在Java中，变量只能在创建它们的区域内被访问。这称为变量的作用域[スコープ]。



# 方法[メソッド]作用域[スコープ]（Method Scope）

- 直接在方法[メソッド]内部声明的变量，在声明它们的代码行之后的方法[メソッド]中的任何位置都可用。

```
public class MyClass {
  public static void main(String[] args) {

    // Code here cannot use x

    int x = 100;

    // Code here can use x
    System.out.println(x);
  }
}
```

# 代码块作用域[スコープ]（Block Scope）

- 代码块是指花括号{}之间的所有代码。
- 在代码块内部声明的变量只能由在声明该变量的行之后的花括号之间的代码访问。
- 一个代码块可能属于它自己，或者属于一个if，while或for语句。对于for语句，语句本身小括号（）中声明的变量也可以在块的作用域[スコープ]内使用。

```
public class MyClass {
  public static void main(String[] args) {

    // Code here CANNOT use x

    { // This is a block

      // Code here CANNOT use x

      int x = 100;

      // Code here CAN use x
      System.out.println(x);

    } // The block ends here

    // Code here CANNOT use x
  }
}
```

尝试代码：Scope.java

# 方法[メソッド]的递归[再帰]（Recursion）

- 递归[再帰]是使指方法[メソッド]里面调用自身方法[メソッド]的一种技术，该技术提供了一种将复杂问题分解为易于解决的简单问题的算法。
- 第一次遇到递归[再帰]可能会有点难以理解，弄清楚递归[再帰]的最佳方法是尝试一下。
- 例：
  - 将两个数字加在一起很容易，但是将一个范围里的数字加起来则比较复杂。在以下示例中，递归[再帰]用于将一个范围内的数字加在一起，方法[メソッド]是将其分解为简单的两个数字相加的任务：

```
public class MyClass {
  public static void main(String[] args) {
    int result = sum(10);
    System.out.println(result);
  } } } }
  public static int sum(int k) {
    if (k > 0) {
      return k + sum(k - 1);
    } else {
      return 0;
```

尝试代码： Recursion.java

# 方法[メソッド]的递归[再帰]（Recursion）

- 示例说明：
  - 调用sum()时，它将参数[引数]k加到所有小于k的数字之和上，并作为返回值[返り値]返回。当k变为0时，该函数仅返回0以终止递归[再帰]。运行时，程序执行以下步骤：

```
10 + sum(9)
10 + ( 9 + sum(8) )
10 + ( 9 + ( 8 + sum(7) ) )
...
10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + sum(0)
10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + 0
```

  - 由于在k=0时函数不会自行调用[呼び出す]，因此程序会在这里停止并返回结果。

# 递归[再帰]的停止条件

- 就像循环会遇到无限循环的问题一样，递归[再帰]函数也会遇到无限递归[再帰]的问题。
- 无限递归[再帰]是函数永不停止调用[呼び出す]自身的情况。
- 每个递归[再帰]函数都应具有停止条件，即该函数停止自行调用[呼び出す]的条件。在前面的示例中，停止条件是参数[引数]k变为0。

- 尝试代码： HaltingCondition.java