Methods

A methods defines a block of reusable code that can be used in the program.

`System.out.println` and `main` are methods.

Using methods saves the developer from duplicating code, which is time consuming and prone to errors.

Methods makes for easier debugging, better program design and improves readability.
The method definition syntax is:

```
modifier returnType name(parameter list)
{
}
```

**modifier**: defines **visibility**. For now use **public static**.

**returnType**: the type of what is returned by the method. It can be a primitive type, an object or the special type **void**, which indicates that nothing is returned by the method.

**name**: follows style rules of identifiers, alphanumeric with first character a lower case letter, descriptive and **camel case**.

**parameter list**: a comma separated list of parameters specified with type. The list is empty for methods that take no parameters.

The method body is always inside mandatory curly brackets.

You cannot have two methods in the same **scope** with the same name and parameter list.
Invocation

Methods are \textit{invoked} (or \textit{called}) by using the method name and the list of parameters enclosed in parenthesis.

The list of parameters must match the order, type and number of elements in the method definition.

If the method returns a value, the method can be used on the right hand side of an assignment or in expressions that expect a value, as long as it is of the correct type.
public static void noValue(int firstP, char secondP)
{
    // Method body
}

public static int aValue(int firstP, char secondP)
{
    // Method body
}

public static void main(String[] args)
{
    noValue(1, 'a');
    int init = aValue(3, 'b');
    init = noValue(2, 'c'); // Illegal void method
    init = aValue('d'); // Illegal wrong param

    while ( aValue(3, 'e') > 1 )
    {
    }

    while ( aValue(3, 'e') ) // Illegal wrong type
    {
    }
The return statement

The return statement terminates a method and returns control to the calling method.

A method that returns a value has to have at least one return statement which must return a value of the same type as in the method definition. At least one of the return statements must be reachable at all times.

Style note: whenever possible, a method that returns a value should have a single return statement at the end of the method.

Style note: void methods can have a return statement, but should be written not to use one whenever possible.
// Better implementation

boolean isEven(int number) {
    boolean result = false;
    if (number % 2 == 0)
        result = true;
    return result;
}

// Legal, but not as clean

boolean isEven(int number) {
    if (number % 2 == 0)
        return true;
    return false;
}

// Illegal, second return may not be reachable

boolean isEven(int number) {
    if (number % 2 == 0)
        return true;
    else if (number % 2 == 1)
        return false;
}
The call stack

When a method is invoked, an activation record containing the parameters and variables is created and placed on the call stack. This is an area of memory organized as LIFO (last in, first out) queue: the last item placed in the queue is the first executed. (The line at the cafeteria is a FIFO, first in, first out.)

Once the program returns from the method, the activation record is removed (popped) from the queue.
Overloaded methods

A class can have multiple methods with the same name, but with different parameter lists. The parameters must differ in type or number. These *overloaded* methods allow the programmer to invoke a method and let the compiler chose the appropriate implementation depending on the arguments.

```java
boolean isEqual(int num1, int num2)
{
    return num1 == num2;
}

boolean isEqual(String string1, String string2)
{
    return string1.equals(string2);
}
```
The compiler will first look for a method with the same exact signature, it will then look for a method whose signature matches the parameters after automatic conversion. This is not always possible and can lead to ambiguous invocations.

```java
double max(int lhs, double rhs)
{
    if ( lhs > rhs )
        return lhs;
    return rhs;
}
double max(double lhs, int rhs)
{
    if ( lhs > rhs )
        return lhs;
    return rhs;
}
double maxNum = max(1, 2); // Ambiguous
```

A method's modifier and return type cannot be used to create overloaded versions.

```java
long max(long lhs, long rhs) {}
int max(long lhs, long rhs) {} // Illegal
```
Variables' scope

The term **scope** refers to the section of the code where a variable is accessible. It begins with the variable's declaration and terminates at the end of the current block (indicated by the closing curly bracket).

Two variables in distinct scopes can have the same name. Two variables in nested scopes **cannot** have the same name.

The scope can be a class, a method or any block of code enclosed in curly brackets, such as the body of a loop.

The `for` loop is a special case:
- one scope starts with the initialization and terminates at the end condition
- the second scope is the loop body