Like C, Python, and Javascript functions, a method may take parameters, and may return a single value. Of course, that single value may be a reference to a list, an object, or an array that contains other values. All value in Java must have a type, and method declarations must indicate the type of parameters and the return value.

Method parameters

Here is an example of the syntax to declare a method that takes parameters:

```
package com.github.akarazhev.jacademy.jprog.basics;

public final class MethodParameters {

    public static void printMyNumber(final int x) {
        System.out.println("My favorite number is " + x + ".");
    }

    public static void main(final String[] args) {
        printMyNumber(42);
    }
}
```

Parameters are local variables of the method. In the definition of `printMyNumber(int x)`, the code `int x` declares a new int variable, `x`. When the method is called, the value 42 is copied into the variable `x`, and then the code of `printMyNumber` is executed.

What happens if you call `printMyNumber` with a double value? The compiler will notice that you are passing in a double, and that the precision of the variable `x` is int. The compiler will give you an error message and halt. If you want to round a double down and pass it in, you are welcome to use the `(int)` cast: an example of explicit casting.

On the other hand, if you have a method that expects a double, you can pass in an int. Java will notice and do the conversion for you automatically: an implicit cast. For example:

```
package com.github.akarazhev.jacademy.jprog.basics;

public final class CastParameter {

    public static void main(final String[] args) {
        // This works, since the int 42 can be cast
        //  implicitly into a double:
        final double x = Math.sqrt(42);

        System.out.println(x);
    }
}
```

Method overloading and method signatures

Sometimes, you want a method that works for different types of parameters. Possibly, you'd even like the method to work slightly differently for different types of parameters.

In Java, you can write two different methods with the same name in the same class, as long as the methods take different parameter types. Here's an example:

```
package com.github.akarazhev.jacademy.jprog.basics;

public final class Overload {

    public static void printMyFavoriteNumber(final int x) {
        System.out.println("My favorite number is the integer " + x);
    }

    public static void printMyFavoriteNumber(final double x) {
        System.out.println("My favorite number is the double " + x);
    }

    public static void main(final String[] args) {
        printMyFavoriteNumber(42);
        printMyFavoriteNumber(3.1415);
    }
}
```

The name of a method, together with the types of its parameters in order, is called the **method signature**.

The names of the parameters are not part of the method signature, so you cannot define two methods with the same name that take the same types of parameters, even if the parameters have different names. Why? When the method is called, Java knows the types of the arguments being passed in, and can use that to determine which of the overloaded methods you intended. But Java can't tell the names of the intended formal parameters from argument values or types.

For similar reasons, the return type of the method, which we will see next, cannot be used for overloading.

Method return values

Unlike Python or Javascript, the type of value that a method returns must be stated in the method declaration. If you later try to return some other type of value, the Java compiler will indicate an invalid return type. This is very useful: coders who use the method know exactly what type of return value to expect, and that expectation can be enforced by the compiler.

An example:

```
package com.github.akarazhev.jacademy.jprog.basics;

public final class ReturnExample {

    public static double square(final double x) {
        return x * x;
    }

    public static boolean isEven(final int x) {
        return x % 2 == 0;
    }

    public static void main(final String[] args) {
        System.out.println(isEven(6));
        System.out.println(square(6));
    }
}
```

In the method declaration, the last word before the name of the method indicates the type of the return value. So we can see that `square` returns a double, `isEven` returns a boolean.

Every method must have its return type declared, even if the method has no return value. A function for drawing a smile, for example, might not return anything at all, so it should have a return type of `void`.

The `main` method returns `void`

The method `main` is declared with `public static void`. The `void` indicates that `main` does not return a value. In C or C++, the main method traditionally returns an integer, indicating to the operating system whether the program ran successfully or not.

Exiting a program with `System.exit()`

Java can exit prematurely and indicate success or failure to the operating system using `System.exit()`. Even if you don't care about the resulting value, `System.exit()` is occasionally useful for debugging, or for terminating a program early for other reasons.

```
package com.github.akarazhev.jacademy.jprog.basics;

public final class HardStop {

    public static void main(final String[] args) {
        System.out.println("Do or do not.");

        // force quit the program, making the code
        //  0 available to the operating system.
        //  (0 traditionally indicates success.)
        System.exit(0);

        System.out.println("There is no try.");
    }
}
```