

# Unit 2: Using Objects Methods

Adapted from:

- 1) Building Java Programs: A Back to Basics Approach  
by Stuart Reges and Marty Stepp
- 2) Runestone CSAwesome Curriculum

# Modularity

**modularity:** Writing code in smaller, more manageable components or modules. Then combining the modules into a cohesive system.

- Modularity with methods. Break complex code into smaller tasks and organize it using **methods**.

**Methods** define the behaviors or functions for objects.

An object's behavior refers to what the object can do (or what can be done to it). A method is simply a named group of statements.

# static vs non-static

Variables and methods can be classified as **static** or **nonstatic(instance)**.

**Non-static or instance:** Part of an object, rather than shared by the class. **Non-static methods are called using the dot operator along with the object variable name.**

**static:** Part of a class, rather than part of an object. Not copied into each object; shared by all objects of that class. **Static methods are called using the dot operator along with the class name unless they are defined in the enclosing class.**

We will further clarify this distinction in Unit 5 when we learn to write our own classes.

# Static Method Inside Driver Class

The **driver class** is the class with the main method. Note that the main method is the begin point of a run of any program. The driver class can contain other static methods. You can call a static method from another method in the **same class directly without referencing the name or object of the class.**

MyClass.java

```
public class MyClass{  
    public static void main(String[] args){  
        method1();  
        method2();  
    }  
    public static void method1(){  
        System.out.println("method1");  
    }  
    public static void method2(){  
        System.out.println("method2");  
    }  
}
```

**Output:**  
**method1**  
**method2**

# Static Method Inside Driver Class

The order of the methods in the driver class does not matter and does not affect the run or output of the program. The program below has the exact same output as the program from the previous slide. The main method is always the starting point of the run of any program.

MyClass.java

```
public class MyClass{  
    public static void method1(){  
        System.out.println("method1");  
    }  
    public static void main(String[] args){  
        method1();  
        method2();  
    }  
    public static void method2(){  
        System.out.println("method2");  
    }  
}
```

**Output:**  
**method1**  
**method2**

# Control flow

When a method is called, the program's execution...

- "jumps" into that method, executing its statements, then
- "jumps" back to the point where the method was called.

## What is the output?

```
public class MethodsExample {
```

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

```
        message1 () ;
```

```
        message2 () ;
```

```
    }
```

```
    ...
```

```
}
```

```
public static void message1() {  
    System.out.println("This is message1.");  
}
```

```
public static void message2() {  
    System.out.println("This is message2.");  
    message1 () ;  
    System.out.println("Done with message2.");  
}
```

```
public static void message1() {  
    System.out.println("This is message1.");  
}
```

Output:

This is message1.

This is message2.

This is message1.

Done with message2.

# Methods

**Non-static or instance** methods belong to individual objects. They are usually implemented inside of an object class rather than the driver class.

Methods in an object class are non-static by default unless explicitly labeled otherwise.

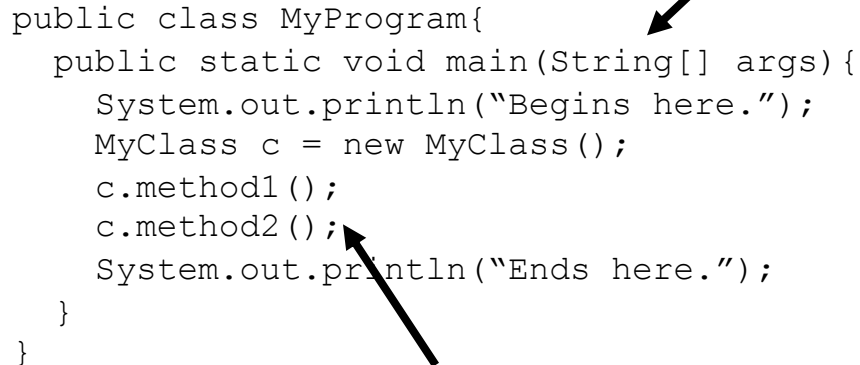
Non-static methods are called through objects of the class.

# Non-static Method Call

MyProgram.java

**A program's run  
begins and ends at  
the main method.**

```
public class MyProgram{
    public static void main(String[] args){
        System.out.println("Begins here.");
        MyClass c = new MyClass();
        c.method1();
        c.method2();
        System.out.println("Ends here.");
    }
}
```



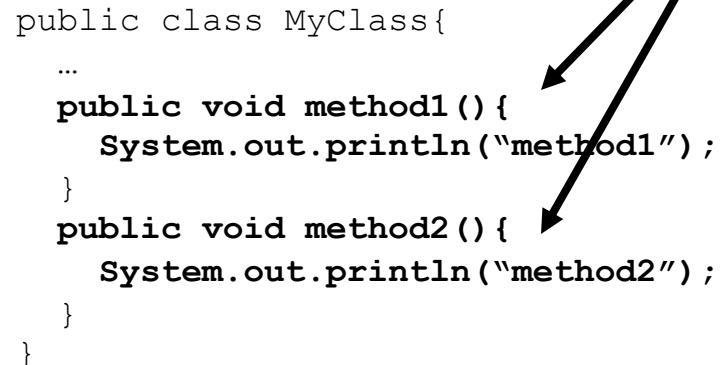
Output:  
Begins here.  
method1  
method2  
Ends here.

**non-static method  
is called through  
the name of an  
object using the dot  
notation**

MyClass.java

**non-static(instance)  
methods**

```
public class MyClass{
    ...
    public void method1(){
        System.out.println("method1");
    }
    public void method2(){
        System.out.println("method2");
    }
}
```





# Method Signature

A **method signature** for a method consists of the method name and the ordered, possibly empty, list of **parameter types**.

```
public void name(parameters) {  
    statements;  
}
```

Examples:

```
public void method1(){  
...  
}
```

**void: no value is returned when method ends.**

**no parameters**

```
public void method2(int x, double y){  
...  
}
```

The parameters in the method header are **formal parameters**.

# Parameters Example 1

When calling a method with parameters, values provided in the parameter list need to correspond to the order and type in the method signature.

```
public class MyProgram{
    public static void main(String[] args){
        mystery1(3, 4); // error, incompatible types!
        mystery1(); // missing actual parameters
        mystery1(3); // missing actual parameters
        mystery1(3, true); // correct
        mystery2(3.2, 3.0); // error, incompatible types!
        double a = 2.5;
        int b = 5;
        mystery2(double a, int b); // error, no type in actual parameters
        mystery2(a, b); // correct

    }
    public static void mystery1(int x, boolean y){
        ...
    }
    public static void mystery2(double x, int z){
        ...
    }
}
```

# Parameters Example 2

When calling a method with parameters, values provided in the parameter list need to correspond to the order and type in the method signature.

## MyProgram.java

```
public class MyProgram{
    public static void main(String[] args){
        MyClass c = new MyClass();
        c.method1(); // correct!
        c.method2(); // error! Missing actual parameters
        c.method2(3.5, 4.1); // error! Wrong types
        c.method2(2, 3.1); // correct!
        c.method2(3, 4); // correct, 4 is casted to a double 4.0
    }
}
```

## MyClass.java

```
public class MyClass{
    public void method1(){
        ...
    }
    public void method2(int x, double y){
        ...
    }
}
```

# Static Vs Non-static Method Calling

MyClass.java

```
public class MyClass{
```

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

```
        System.out.println(SomeClass.method1());
```

```
        SomeClass a = new SomeClass();
```

```
        System.out.println(a.method2());
```

```
    }
```

```
}
```

SomeClass.java

```
public class SomeClass{
```

```
    public SomeClass()
```

```
    {...}
```

```
    public static int method1() // static method
```

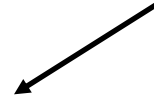
```
    {...}
```

```
    public int method2() // non-static or instance method
```

```
    {...}
```

```
}
```

call static method through  
name of class



call non-static method  
through name of an object



**Note that method1 and method2 both belong  
to a different class than the driver class  
where they are being called.**

# Method Returns

Methods in Java can have **return types**. Such **non-void** methods return values back that can be used by the program. A method can use the keyword “return” to return a value.

```
public type methodName(type var1,..., type var2) {  
...  
}
```

Examples:

```
public int method1() {  
...  
}
```

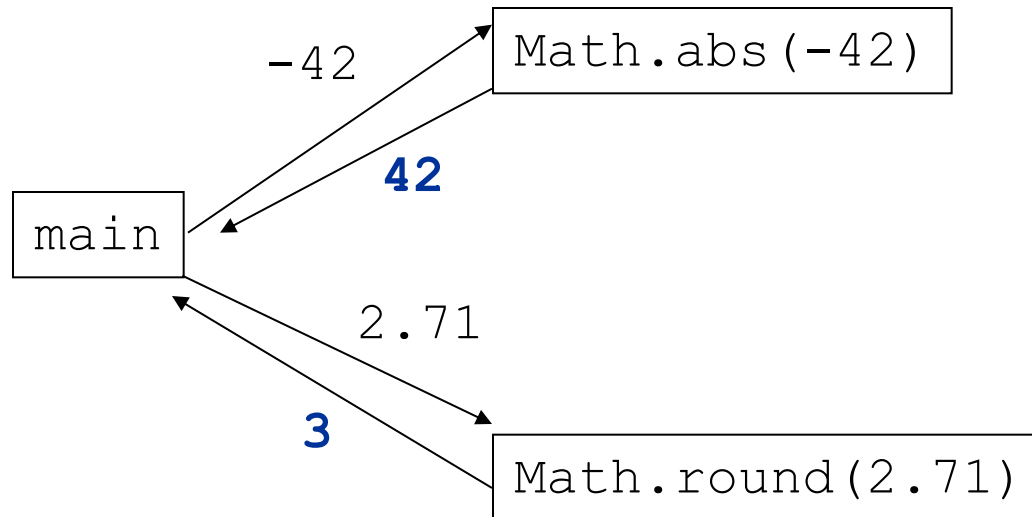
**return types**

```
public double method2(int x) {  
...  
}
```

**Note: Method parameters are its inputs and method returns are its outputs.**

# Return

- **return:** To send out a value as the result of a method.
  - The opposite of a parameter:
    - Parameters send information *in* from the caller to the method.
    - Return values send information *out* from a method to its caller.
      - A call to the method can be used as part of an expression.



# Return

Non-void methods return a value that is the same type as the return type in the signature.

To use the return value when calling a non-void method, it must be stored in a variable or used as part of an expression.

**Procedural abstraction** allows a programmer to use a method by knowing what the method does even if they do not know how the method was written.


For example, the Math library, part of the java.lang package contains many useful mathematical methods. We will use these methods to understand how to use return values.

# Common error: Not storing

Many students forget to store the result of a method call.

```
public static void main(String[] args) {  
    Math.abs(-4); // error! Returned value not stored nor used  
  
    // corrected  
    int result = Math.abs(-4);  
    System.out.println(result); // 4  
  
    System.out.println("the square root of 4 is " + Math.sqrt(4));  
    // the square root of 4 is 2.0  
}
```

**returned value is concatenated with a string**





# NullPointerException

Using a null reference to call a method or access an instance variable causes a **NullPointerException** to be thrown.

```
public static void main(String[] args) {  
  
    Sprite a = null; //currently the variable a references no object  
    a.display(); // NullPointerException, can't call method on  
                // a reference to nothing!  
    System.out.println(a.center_x); // NullPointerException  
}
```

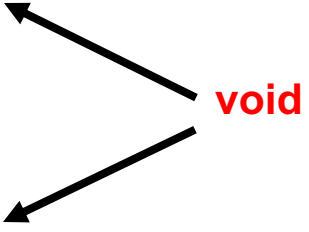
# Void Methods

Void methods do not have return values. Once the execution of the method completes, the flow of control returns to the point immediately following where the method was called.

```
public void methodName (type var1, ..., type var2) {  
...  
}
```

Examples:

```
public void method1 () {  
...  
}  
  
public void method2 (int x) {  
...  
}
```



The diagram consists of two black arrows. The first arrow originates from the word 'void' (highlighted in red) and points to the 'void' keyword in the signature of 'method1'. The second arrow originates from the same red 'void' and points to the 'void' keyword in the signature of 'method2'.

# Void Methods

Void methods do not have return values and are therefore not called as part of an expression.

```
public class MyClass{
    public static void main(String[] args){
        int a = 3 + printX(5); //error! Does not return!
        int b = 5 * twiceX(3); // correct, b = 30
    }
    public static void printX(int x){
        System.out.println("The input x is" + x);
    }
    public static int twiceX(int x){
        return 2 * x;
    }
}
```

# Overloaded Methods

Methods are said to be **overloaded** when there are multiple methods with the same name but a different signature.

```
public class MyClass{
    public static void main(String[] args){
        double a = add(1, 2) + add(1.8, 5.2) + add(1, 2, 3);
        System.out.println(a); // 16.0
    }
    public static int add(int x, int y){
        return x + y;
    }
    public static double add(double x, double y){
        return x + y;
    }
    public static int add(int x, int y, int z){
        return x + y + z;
    }
}
```

Three methods named "add".

# Value Semantics

Parameters are passed using **call by value or value semantics**. Call by value initializes the formal parameters with copies of the actual parameters. When primitive variables (`int`, `float`, `boolean`) and `String` (the only object class that does this) are passed as parameters, **their values are copied**.

- Modifying the parameter will not affect the variable passed in.

```
public class MyClass{
    public static void main(String[] args){
        int x = 23;
        strange(x);
        System.out.println("2. x = " + x);
    }
    public static void strange(int x){
        x = x + 1;
        System.out.println("1. x = " + x);
    }
}
```

The `x` variable in `main` is different than the `x` variable in `strange`.

Output:

1. x = 24  
2. x = 23

Note: The value of `x` in `main` did not change.

# Value semantics

**Value semantics:** methods cannot change the values of primitive types(int, boolean, float) and String.

```
public class MyClass{
    public static void main(String[] args){
        int x = 5;
        doubleMyNumber(x);
        System.out.println("My number is" + x); //My number is 5
    }
    public static void doubleMyNumber(int x){
        x = x * 2;
    }
}
```

Note: The value of x in main did not change.

# Find all errors.

```
public class MyClass{
    public static void main(String[] args){
        printX();
        add();
        add(3, 5);
        System.out.println(printX());
        System.out.println("3 + 5 = " + add(3, 5));
        int y = 3 + add(4, 6.0);
    }
    public static void printX(int x){
        System.out.println("The input x is" + x);
    }
    public static int add(int x, int y){
        return x + y;
    }
}
```

# Answers

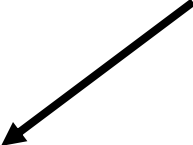
```
public class MyClass{
    public static void main(String[] args){
        printX(); // missing actual parameter.
        add(); // missing actual parameters.
        add(3, 5); // returned value not stored
                    // but not a syntax error.
        System.out.println(printX(5)); // error!
                                    //no returned value!
        System.out.println("3 + 5 = " + add(3, 5)); //correct!
        int y = 3 + add(4, 6.0); // incompatible types!
    }
    public static void printX(int x){
        System.out.println("The input x is" + x);
    }
    public static int add(int x, int y){
        return x + y;
    }
}
```



# Nonstatic vs Static

Let's do one example of a object class to understand when to make a method static vs. non-static.

```
class Student{  
    int id;  
    public Student(int new_id){  
        id = new_id;  
    }  
    public void printMyID(){  
        System.out.println("My ID is " + id);  
    }  
    public static void printWelcomeMessage(){  
        System.out.println("Welcome all students!");  
    }  
}
```



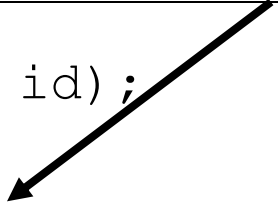
**printMyID is a non-static method and belong to individual student objects. E.g. if there are 5 student objects, there are 5 different copies of printMyID, one for each student.**

# Nonstatic vs Static

Let's do one example of a object class to understand when to make a method static vs. non-static.

```
class Student{
    int id;
    public Student(int new_id){
        id = new_id;
    }
    public void printMyID(){
        System.out.println("My ID is " + id);
    }
    public static void printWelcomeMessage(){
        System.out.println("Welcome all students!");
    }
}
```

**printWelcomeMessage is a static(class) method. It belongs to the class rather than individual objects. If there are 5 student objects, there is only ONE shared printWelcomeMessage.**



# References

1) Building Java Programs: A Back to Basics Approach by Stuart Reges and Marty Stepp

2) Runestone CSAwesome Curriculum:

<https://runestone.academy/runestone/books/published/csawesome/index.html>

For more tutorials/lecture notes in Java, Python, game programming, artificial intelligence with neural networks:

<https://longbaonguyen.github.io>