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

https://longbaonguyen.github.io

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.

- `main` is an example of a method

Example

Consider the following code which asks the user to enter two numbers and print out the average.

```
Scanner console = new Scanner(System.in);
System.out.print("Enter a number: ");
int num1 = console.nextInt();
System.out.print("Enter a number: ");
int num2 = console.nextInt();
System.out.println("The average is " + (num1 + num2)/2.0);
```

What if we need to do this again?

We don't want to repeat this code by copying and pasting as shown in the next slide.

Example

If we need to repeat this task, we do not want to simply copy and paste out code:

```
Scanner console = new Scanner(System.in);
int num1, num2;
System.out.print("Enter a number: ");
num1 = console.nextInt();
System.out.print("Enter a number: ");
num2 = console.nextInt();
System.out.println("The average is " + (num1 + num2)/2.0);

System.out.print("Enter a number: ");
num1 = console.nextInt();
System.out.print("Enter a number: ");
num2 = console.nextInt();
System.out.println("The average is " + (num1 + num2)/2.0)
```

Method

One way to organize code and to make it more readable and reusable is to factor out useful pieces into reusable *methods*.

A ***method*** is a named group of programming instructions that accomplish a specific task. If we want to perform the task, we simply "call" the method by its name. A method may be called as many times as we wish to redo the task.

The "30 seconds" button on the microwave is an example of a method. If we press it(call it by its name), it will run the microwave 30 seconds. Later, if we want to heat something else, we can press it again to run the microwave another 30 seconds.

In other programming languages, methods are also called **procedures** or **functions.**

Example

If we need to repeat this task, we do not want to simply copy and paste out code:

```
Scanner console = new Scanner(System.in);
int num1, num2;
System.out.print("Enter a number: ");
num1 = console.nextInt();
System.out.print("Enter a number: ");
num2 = console.nextInt();
System.out.println("The average is " + (num1 + num2)/2.0);

System.out.print("Enter a number: ");
num1 = console.nextInt();
System.out.print("Enter a number: ");
num2 = console.nextInt();
System.out.println("The average is " + (num1 + num2)/2.0)
```

Let's factor out this piece of code, convert it into a method by giving it a name!

Then we can call it repeatedly if we wish to run the code.

Method

A **method** is a group of code that has a name and can be called using parentheses.

```
public class Main{
  public static void main(String[] args){
     // calling it the first time
     average();
     // calling it again to repeat the task
     average();
  }
  public static void average(){
    Scanner console = new Scanner(System.in);
    int num1, num2;
    System.out.print("Enter a number: ");
    num1 = console.nextInt();
    System.out.print("Enter a number: ");
    num2 = console.nextInt();
    System.out.println("The average is " + (num1 + num2)/2.0);
  }}
```

```
Enter a number: 4
Enter a number: 6
The average is 5.0

Enter a number: 10
Enter a number: 11
The average is 10.5
```

static vs non-static

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

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.

```
double x = Math.pow(2, 3); // pow is a static method
                           // its code is in the Math class
                           // Note the dot notation.
```

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.

```
Scanner console = new Scanner(System.in);
// the code for accepting user input is found in the
// Scanner class

// the nextInt() method is non-static or instance, it is
// called through an object(console) rather than the class.
int num = console.nextInt();
```

```
// static methods, called through class name(Math)
double x = Math.pow(2, 3);
dount y = Math.sqrt(9);

// non static or instance methods, call through an object
Scanner console = new Scanner(System.in);
int num = console.nextInt();
```

Classify the `length` and `valueOf` methods below as static or non-static.

```
String name = "Smith";
System.out.println(name.length()); // 5
int i = 10;
String s = String.valueOf(i); // s = "10"
```

Answer: length is nonstatic and valueOf is static.

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 enclosing class directly without referencing the name or object of the class. No dot notation is needed.**

MyClass.java

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

Output:
running method2
running method1

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

Output:
running method2
running method1

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.

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 static void method1(){
…
}
```

void: no value is returned when method ends.

no parameters

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

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

Static Example

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){
  …
  }
}
```

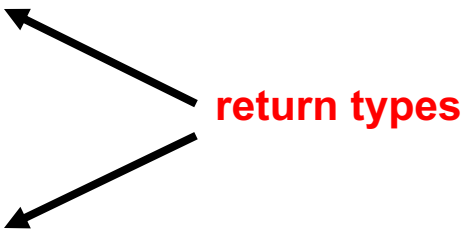
Method Returns

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 static int method1(){
…
}
```

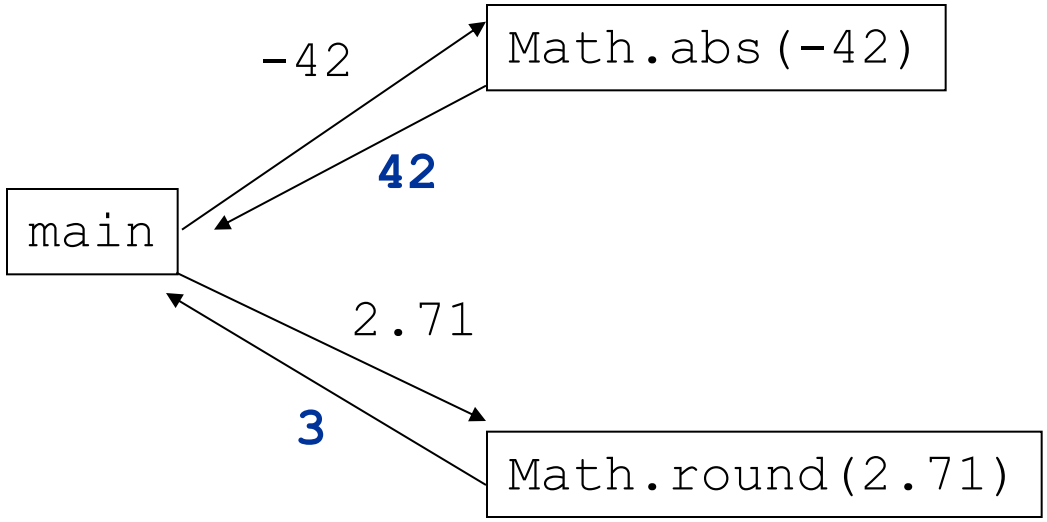



```
public static double method2(int x){
…
}
```

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 may not know how these methods were implemented but we can still use them.

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
                // (not a compiler/syntax error)
  // 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
  System.out.println(sum(3, 5)); // 8
  int result2 = sum(5, 7);  // 12
}
public static int sum(int a, int b){
  return a + b;
}
```

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

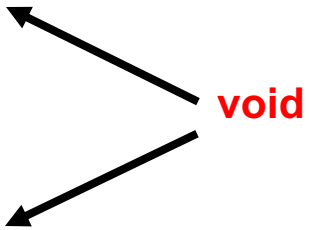
Void methods do not have return values. Once the execution of the method completes, the flow of control returns to the point immediately following where the method was called.

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

Examples:

```
public static void method1(){
…
}

public static void method2(int x){
…
}
```



Void Methods

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

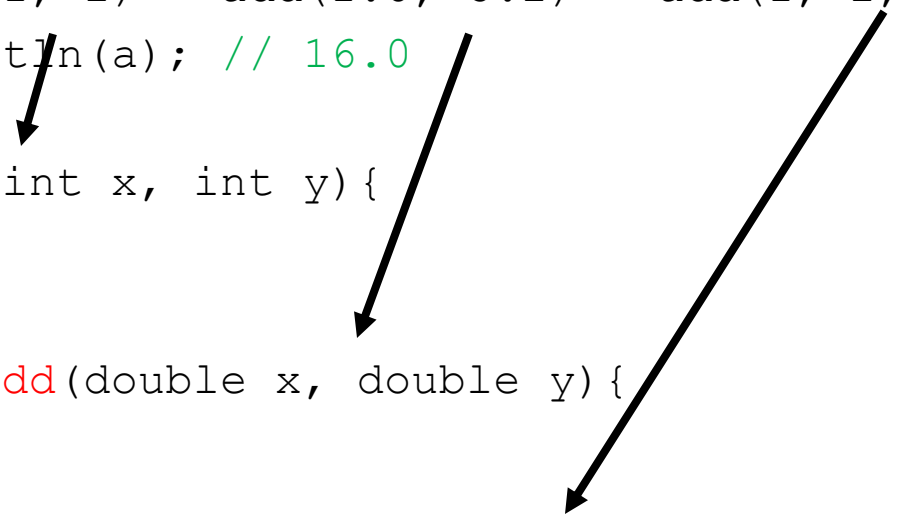
```
public class MyClass{
	public static void main(String[] args){
		int a = 3 + printX(5); //error! Does not return!
		int b = 5 * twiceX(3); // correct, b = 30
		printX(5); // correct
				// Output: The input x is 5
	}
	public static void printX(int x){
		System.out.println("The input x is" + x);
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

Three methods named "add".

```
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;
        }
}
```



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`, `double`, `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(5));
		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;
        }
}
```

Lab

Create a new repl on repl.it.

Write a driver class with the following five **static** methods.

```
// given two integers x and y, returns their average.
public static double average(int x, int y)
{…}


// given two points (x1, y1) and (x2,y2), returns
// the slope of the line through them. You may assume
// x1 is not equal to x2.
public static double slope(int x1,int y1,int x2,int y2)
{…}
```

Lab

```
// given two integers x and y, returns the difference x-y
public static int difference(int x, int y)
{…}

// given an integer x returns its square x*x.
public static int square(int x)
{…}

// given two points on the plane, returns the distance between them.
// You MUST CALL the methods difference and square above.
// In addition, you CANNOT use subtraction nor multiplication in this method.
// distance = √((x1 − x2)²+(y1 − y2)²)
public static double distance(int x1, int y1, int x2, int y2)
{…}
```

Lab

Write your main() method so that your program has an output similar to:

```
The average of 8 and 9 is 8.5
The slope of the line between (8,9) and (2,4) is 0.8333333333333334
The distance between (8,9) and (2,4) is 7.810249675906654
```

Notice the format of the points on the coordinate plane.

Lab 2

For this lab, please refer to optional lecture on **User Input** in Unit 2 on https://longbaonguyen.github.io/courses/apcsa/apjava.html

Create a new repl. Implement the driver class(Main.java on repl.it) to **ask the user to enter two different points(using a Scanner object)**on the plane and print out their midpoint and the distance between them.

For (x1, y1) and (x2, y2):

Midpoint: ((x1+x2)/2, (y1+y2)/2)

Lab 2

Write your program so that it has EXACTLY THE FOLLOWING OUTPUT.

Program Output: (underlined values are user-entered inputs)

Enter x1: 2
Enter y1: -1
Enter x2: 3
Enter y2: 5
The midpoint between (2,-1) and (3,5) is (2.5, 3.0)
The distance between (2,-1) and (3,5) is 6.082762530298219

Lab 2 Outline

I created a repl for this lab.

Click on the link below to go to the repl. Then "fork" it by either pressing on the "fork it" button or repl.it will fork it for you automatically if you begin editting the program.

Fill in the code as indicated by the comments.

https://repl.it/@LongNguyen18/userinputlab

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