



Coding-bootcamps.com

Introduction to Java Programming Language

By Jim Sullivan
from [Coding Bootcamps](https://coding-bootcamps.com)



Writing Methods (Functions)

Session 5



Brief Recap

- Re-hash what we've covered in our last class

Objective - Discuss The Following

- Static vs. Dynamic Allocation
- Declaring Methods
- Declaring Methods with Multiple Parameters
- Method-Call Stack
- Scope of Declarations
- Argument Promotion and Casting
- Designing Methods for Reusability
- Method Overloading

Materials

- These Powerpoint Slides

Methods - Declaring Methods

- Last course we discussed how to declare methods.
 - Who remembers what the modifier types are?
 - What is the return type?
 - What is the parameter list?
 - Where does the body go?

Methods - Declaring Methods

- Suppose we want to modify a class where a method has multiple inputs.
 - The best way to accomplish this is to use parameters for passing additional information
 - To write a method that takes a parameter, you list the type and name of the parameter in parentheses.
 - If you want a method with multiple parameters, list each parameter's type and name in the method declaration's parentheses, separated by commas.
 -

Methods - Declaring Methods with Multiple Parameters

```
public class MyClass
{
    public void doSomething(int i)
    {
        ...
    }

    public void doSomething(int i, String s)
    {
        ...
    }

    public void doSomething(int i, String s, boolean b)
    {
        ...
    }
}
```


The Method-Call Stack

- The Stack keeps the state of all active methods (those that have been invoked but have not yet completed)
- When a method is called a new stack frame for it is added to the top of the stack, this stack frame is where space for the method's local variables and parameters are allocated
- When a method returns, its stack frame is removed from the top of the stack (space for its local vars and params is de-allocated)
- Space for local variables and parameters exist only while the method is active (it has a stack frame on the Stack)
- local variables and parameters are only in scope when they are in the top stack frame (when this method's code is being executed)

The Method-Call Stack (Example)

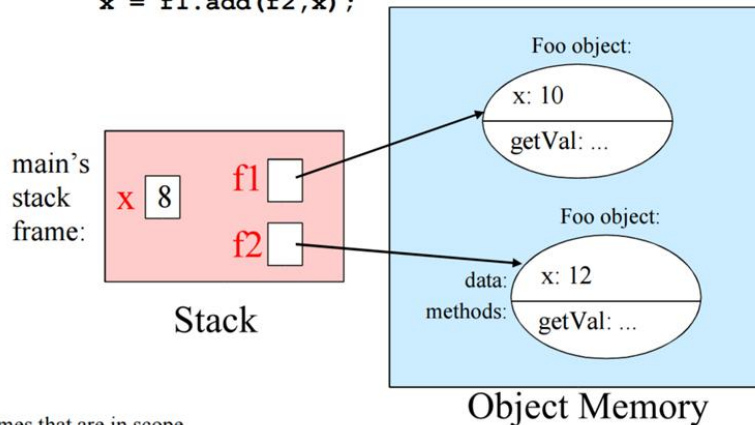
```
public class TestMethodCalls {  
    public static void main(String[] args) {  
        Foo f1, f2;  
        int x=8;  
        f1 = new Foo(10);  
        f2 = new Foo(12);  
        f1.setVal(x);  
        x = f1.add(f2, x);  
    }  
}
```

```
public class Foo {  
    private int x;  
    public Foo(int val) { x = val; }  
    public void setVal(int val) { x = val; }  
    public int getVal() { return x; }  
    public int plus(Foo f, int val) {  
        int result;  
        result = f.getVal() + x + val;  
        return result;  
    }  
}
```

The Method-Call Stack (What happens)

- We start executing code in main:
 - there is a single stack frame containing main's local variables

```
public class TestMethodCalls {  
    public static void main(String[] args) {  
        Foo f1, f2;  
        int x = 8;  
        f1 = new Foo(10);  
        f2 = new Foo(12);  
        f1.setVal(x);  
        x = f1.add(f2,x);  
    }  
}
```



Only through variables f1 and f2 can main access the objects' public members

■: names that are in scope

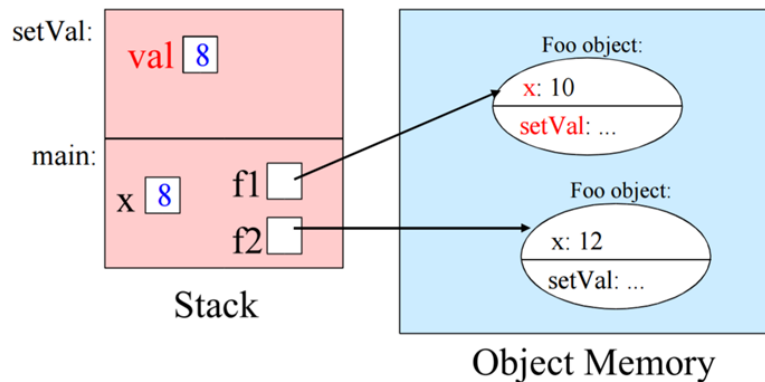
The Method-Call Stack (What happens)

- When main calls f1's setVal method a new stack frame is added that holds setVal's parameters and local variables

```
public class TestMethodCalls {  
    public static void main( ... ) {  
        ...  
        → f1.setVal(x);  
    }  
}
```

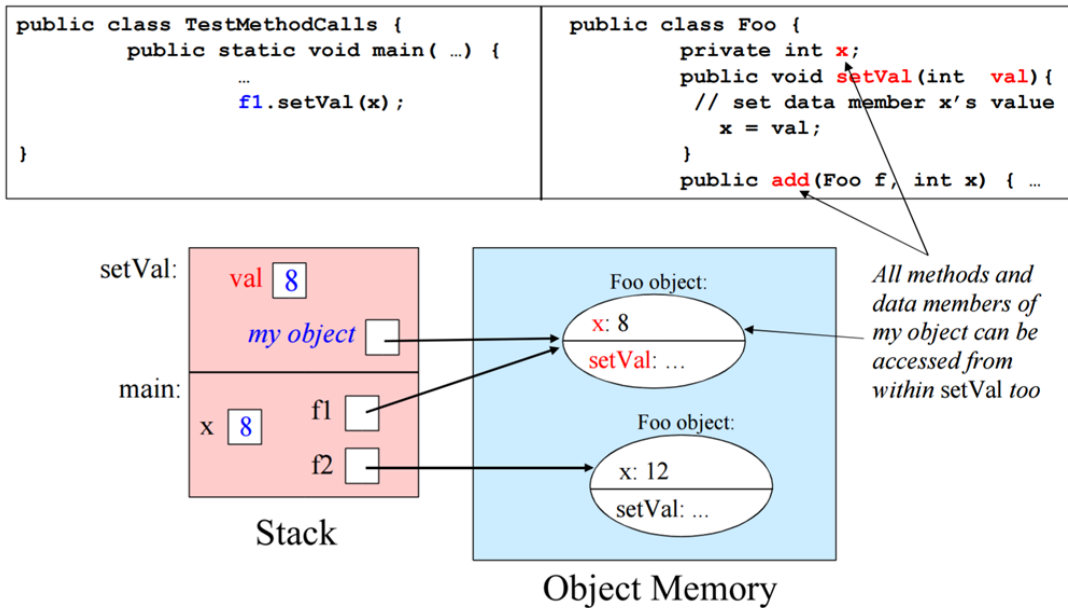
```
public class Foo {  
    private int x;  
    public void setVal(int val) {  
        x = val;  
    }  
}
```

Parameter **val** gets its value from its argument (the value of **x** in main)



The Method-Call Stack (What happens)

- Implicitly, a reference to the object referred to by f1 is passed as well:
 - setVal is called from within this object, so its members are in scope as well as all parameters and local variables of setVal

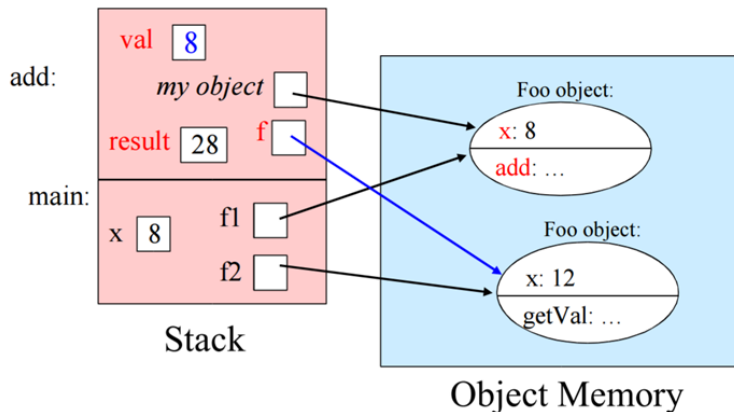


The Method-Call Stack (What happens)

- When main calls add, we are passing the value of object ref f2:
- add's parameter f refers to the same object as f2 does

```
public class TestMethodCalls {  
    public static void main( ... ) {  
        . . .  
        x = f1.add(f2, x);  
    }  
}
```

```
public class Foo {  
    private int x;  
    public int plus(Foo f, int val) {  
        int result;  
        result = f.getVal() + x + val;  
        return result;  
    }  
}
```

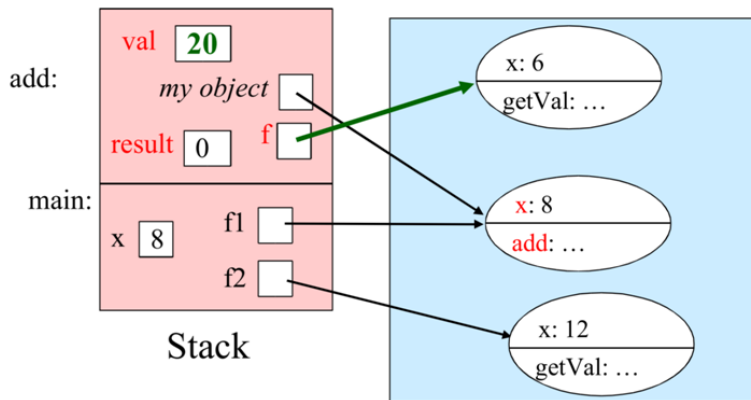


The Method-Call Stack (What happens)

- If a method changes the value of a parameter, it does not change the argument's value:

```
public class TestMethodCalls {  
    public static void main( ...) {  
        . . .  
        x = f1.add(f2, x);  
    }  
}
```

```
public class Foo {  
    private int x;  
    public int plus(Foo f, int val) {  
        int result = 0;  
        f = new Foo(6);  
        val = 20;  
        return result;  
    }  
}
```



The Method- Scope of Declarations

- Scope of a variable is the part of the program where the variable is accessible.
- All identifiers are lexically (or statically) scoped
 - i.e.scope of a variable can determined at compiler time and independent of function call stack.
- Java programs are organized in the form of classes. Every class is part of some package.

The Method- Scope of Declarations

- Member Variables (Class Level Scope)
 - These variables must be declared inside class (outside any function). They can be directly accessed anywhere in class. Let's take a look at an example:

```
public class Test
{
    // All variables defined directly inside a class
    // are member variables
    int a;
    private String b
    void method1() {....}
    int method2() {....}
    char c;
}
```

The Method- Scope of Declarations

- We can declare class variables anywhere in class, but outside methods.
- Access specified of member variables doesn't effect scope of them within a class.
- Member variables can be accessed outside a class with following rules

The Method- Scope of Declarations

- Local Variables (Method Level Scope)
 - Variables declared inside a method have method level scope and can't be accessed outside the method.

```
public class Test
{
    void method1()
    {
        // Local variable (Method level scope)
        int x;
    }
}
```

The Method- Scope of Declarations

- Loop Variables (Block Scope)
 - A variable declared inside pair of brackets "{" and "}" in a method has scope withing the brackets only.

```
class Test
{
    public static void main(String args[])
    {
        for (int x = 0; x < 4; x++)
        {
            System.out.println(x);
        }

        // Will produce error
        System.out.println(x);
    }
}
```

The Method- Scope of Declarations

- Some Important Points about Variable scope in Java:
 - In general, a set of curly brackets { } defines a scope.
 - In Java we can usually access a variable as long as it was defined within the same set of brackets as the code we are writing or within any curly brackets inside of the curly brackets where the variable was defined.
 - Any variable defined in a class outside of any method can be used by all member methods.
 - When a method has same local variable as a member, this keyword can be used to reference the current class variable.
 - For a variable to be read after the termination of a loop, It must be declared before the body of the loop.

The Method- Argument Promotion and Casting

- Argument promotion—implicitly converting an argument's value to the type that the method expects to receive (if possible) in its corresponding parameter.
 - For example, an app can call Math method Sqrt with an integer argument even though the method expects to receive a double argument.

```
System.out.println(Math.sqrt(16));
```

The Method- Argument Promotion and Casting

- Such conversions may lead to compilation errors if promotion rules are not satisfied.
 - The promotion rules specify which conversions are allowed—that is, which conversions can be performed without losing data.
 - In the Sqrt example above, an int is converted to a double without changing its value.
 - However, converting a double to an int truncates the fractional part of the double value—thus, part of the value is lost.
 - Also, double variables can hold values much larger (and much smaller) than int variables, so assigning a double to an int can cause a loss of information when the double value doesn't fit in the int.
 - Converting large integer types to small integer types (e.g., long to int) also can produce incorrect results.

The Method- Argument Promotion and Casting

- The promotion rules apply to expressions containing values of two or more simple types and to simple-type values passed as arguments to methods.
- Each value is promoted to the appropriate type in the expression. (Actually, the expression uses a temporary copy of each promoted value—the types of the original values remain unchanged.)

The Method- Argument Promotion and Casting

Type	Conversion types
bool	no possible implicit conversions to other simple types
byte	ushort, short, uint, int, ulong, long, decimal, float or double
char	ushort, int, uint, long, ulong, decimal, float or double
decimal	no possible implicit conversions to other simple types
double	no possible implicit conversions to other simple types
float	double
int	long, decimal, float or double
long	decimal, float or double
sbyte	short, int, long, decimal, float or double
short	int, long, decimal, float or double
uint	ulong, long, decimal, float or double
ulong	decimal, float or double
ushort	uint, int, ulong, long, decimal, float or double

The Method- Method Overloading

- We previously discussed having the same method with multiple parameters
- Method Overloading is a feature that allows a class to have two or more methods having same name, if their argument lists are different.
- Argument lists could differ in
 - Number of parameters.
 - Data type of parameters.
 - Sequence of Data type of parameters.

Assignment

- Use what we've learned so far to create a simple application that can do the following
 - Create a car (With attributes)
 - A method to modify the car (with method overloading) to do things like paint the car, set the size of the tires, etc

Assignment

Reminder, this is how you read and accept keyboard input

- ```
Scanner keyboard = new Scanner(System.in);
System.out.println("enter an integer");
int myint = keyboard.nextInt();
```

# Summary

# Live private coaching sessions for Java

- [Private tutoring sessions for software design and engineering- Weekly and monthly plans](#)
- [Java programming language- Private tutoring sessions](#)



Coding-bootcamps.com

Thank You

