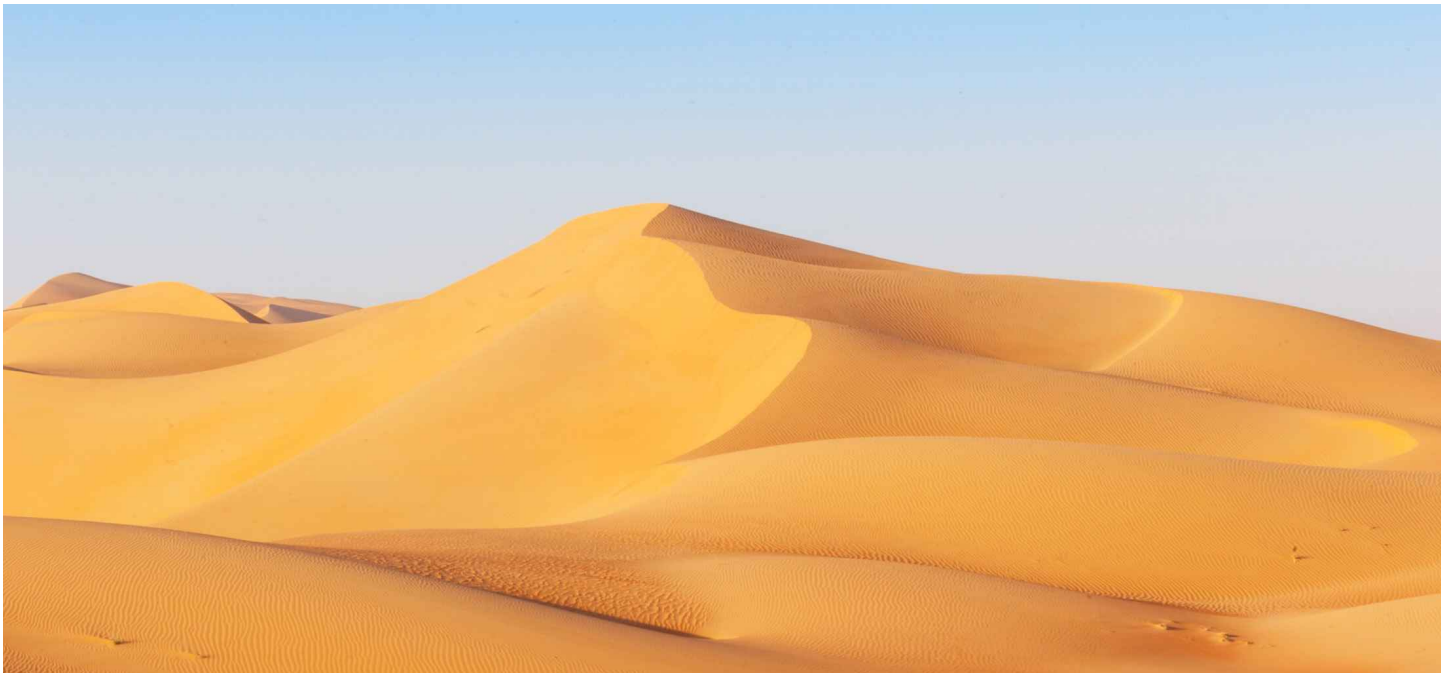


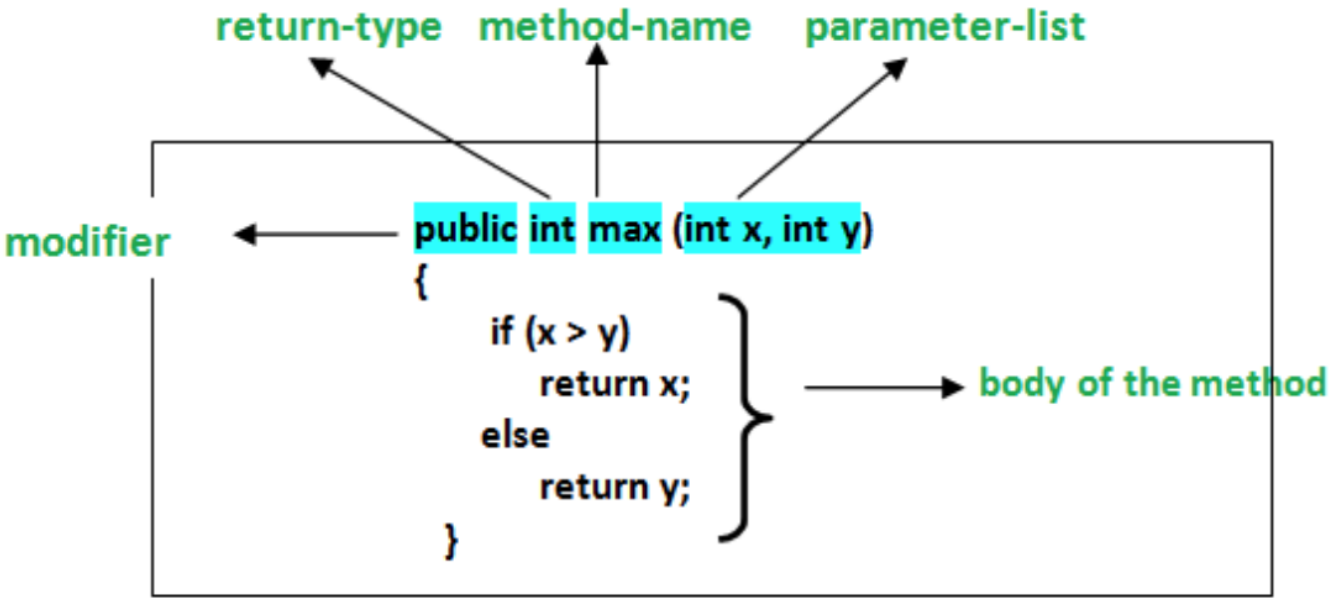
11-Instance Methods

Author: Vincent Lau

Learning Objectives

- List the features of a method
- Understand and use methods to encapsulate logic
- Understand the concept of Variable Arguments (varargs)
- Describe the circumstances under which a method can be terminated
- Understand the benefits of a method returning an interface

Methods



Defining Methods

- The only `required` elements of a method declaration are:
  - Return type
  - Method name
  - A pair of parentheses `()`
  - A body between braces `{}`
- Optional elements include:
  - Modifiers, such as `public` or `private`
  - The parameter list inside the parentheses
  - An exception list
- The method name should use **Lower Camel Case**

```
1  public String methodWithoutArgument() {
2      //do something
3      return someOutput;
4  }
5
6  public String methodWithArguments(int someInteger, String someString) {
7      //do something
8      return someOutput;
9  }
10
11 public double calculateAnswer(double wingSpan, int numberOfEngines,
12                              double length, double grossTons) {
13     //do the calculation here
```

```
14      return someAnswer;
15  }
```

Method Signature

- `Method Signature` = Method Name + Parameter Types
- Just a definition of what Method Signature is. ***`Method Overloading`*** has relations to *method signature*.

```
1  public void methodWithoutArgument()
2
3  public void methodWithArguments(int a, String b)
4
5  public void methodWithArguments(int x, String y) // duplicated method, compile
   error
6
7  calculateAnswer(double a, int b, double c, double d)
```

Method Overloading

- Java supports `overloading` methods, and Java can distinguish between methods with different *method signatures*. That means methods within a class can have the same name if they have different parameter lists.
- You **`CANNOT`** declare more than one method with **the same name and the same number and type of arguments**, because the compiler cannot tell them apart.
- The compiler **does not consider return type** when differentiating methods, so you **`CANNOT`** declare two methods with the same signature even if they have a different return type.
- Overloaded methods should be avoided because they reduce code readability. For example, the method process is supposed to have similar behavior even if they have different parameters. Everyone needs to read the *process* methods 4 times before deciding to use them.

```
1  public class SomeProcessor {
2      ...
3      public void process(String s) {
4          ...
5      }
6      public void process(int i) {
7          ...
```

```
    }
    // Compile error, due to same method signature
    public void process(int j) {
        ...
    }
    public void process(double a) {
        ...
    }
    // Compile error, due to same method signature, even return type is different
    public String process(double b) {
        ...
    }
    public void process(int i, double f) {
        ...
    }
}

// process(String i) and process(int i) are distinct and unique methods
// because they require different parameter types
```

Varargs - Variable Argument

- `Varargs` stands for Variable Arguments. In Java, an argument of a method can accept arbitrary number of values. This argument that can accept variable number of values is called *varargs*.
- A method can only have **one** varargs parameter.
- When defining method signature, **always keep varargs at last**. Otherwise, the compiler will not be able to distinguish between non-variable arguments and variable arguments.

```
// These three methods have the same method signature

public static void main(String[] args) { // corrent syntax
    // do something
}

public static void main(String... args) {  // corrent syntax, varargs
    // do something
}

// String[] is formal syntax of String array
```

```
// String args[] is not recommended, but compile will sucesss
public static void main(String args[]) {
    // do something
}
```

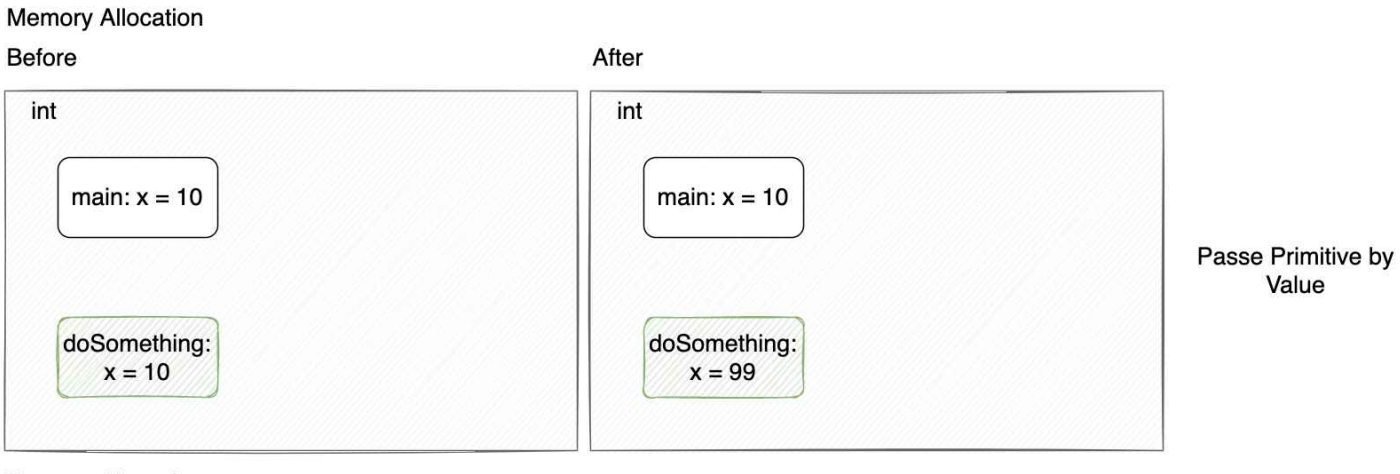
Example

```
public static int sum(int[] nums) {
    int sum = 0;
    for (int num : nums) {
        sum += num;
    }
    return sum;
}

public static int sum(int... nums) { // Variable Args cannot overloading with array
    int sum = 0;
    for (int num : nums) {
        sum += num;
    }
    return sum;
}

// main program: Invoke the method
System.out.println(sum(1, 2, 3, 4, 5)); // prints 15
System.out.println(sum(1, 2, 3)); // prints 6
System.out.println(sum()); // prints ?
```

Pass by Value

Pass by Value (Scenario 1 - Primitive Type)

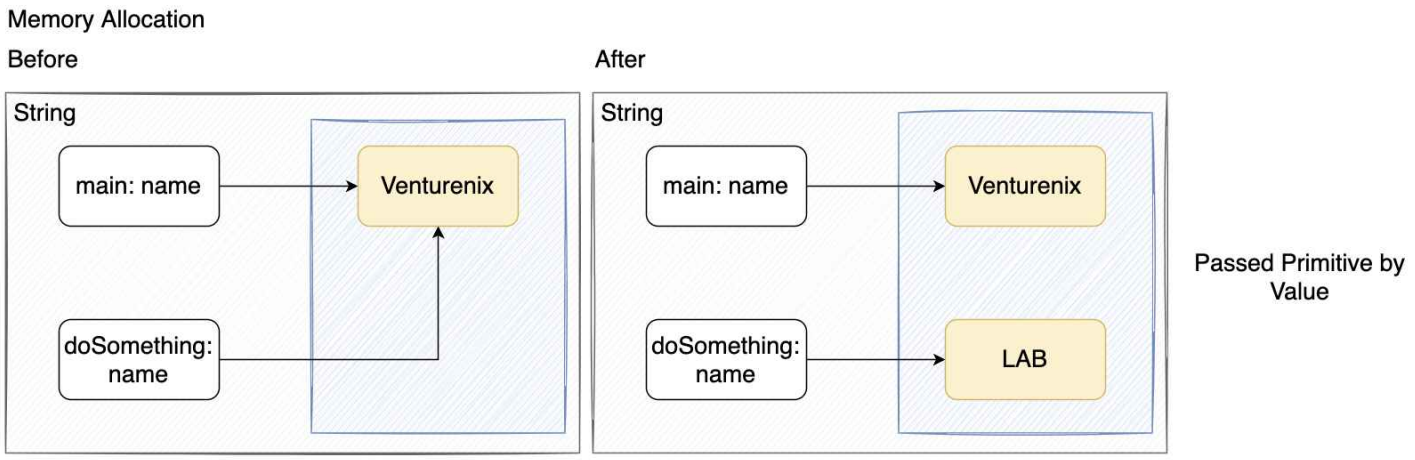
- In Java, primitive arguments, such as *int* or *double,* are passed into methods *by value*. This means that any changes to the values of the arguments exist only within the scope of the method. When the method returns, the parameters are gone and any changes to them are lost.




```java
public class PassPrimitiveByValue {

    public static void doSomething(int x) {
        // x = 10
        x = 99; // this value assignment only apply to parameter x in this method
        // x = 99
        System.out.println(x); // print 99
    }

    public static void main(String[] args) {
        int x = 10; // int x; x = 10;
        doSomething(x); // pass value 10 to the method
        System.out.println("x is " + x); // print 10
    } // Output: x is 10
}
```

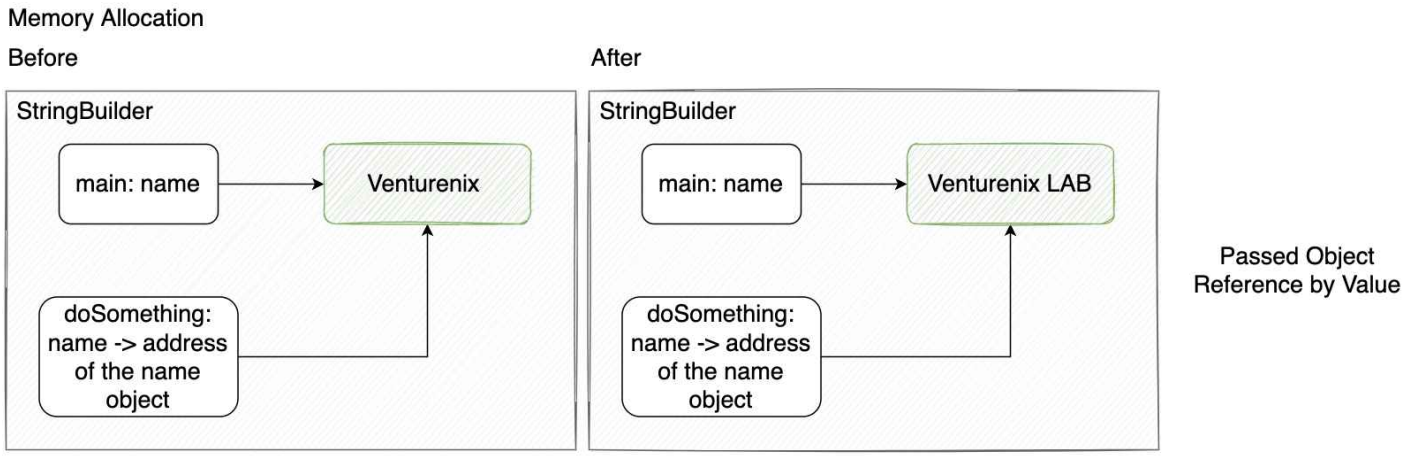
Pass by Value (Scenario 2 - Wrapper Class)

- Object reference parameters are also passed into methods *by value*. This means that when the method returns, the passed-in reference still **points to the same object as before**.
- However, the value of the object's fields *can *****be changed in the method.




```java
// Scenario 1
public class PassReferenceByValue {
    public static void doSomething(String name) { // String is immutable, as well as all other wrapper class
        name = "LAB";
    }

    public static void main(String[] args) {
        String name = new String("Venturenix");
        doSomething(name);
        System.out.println(name); // print "Venturenix"
    } // Output: "Venturenix"
}
```

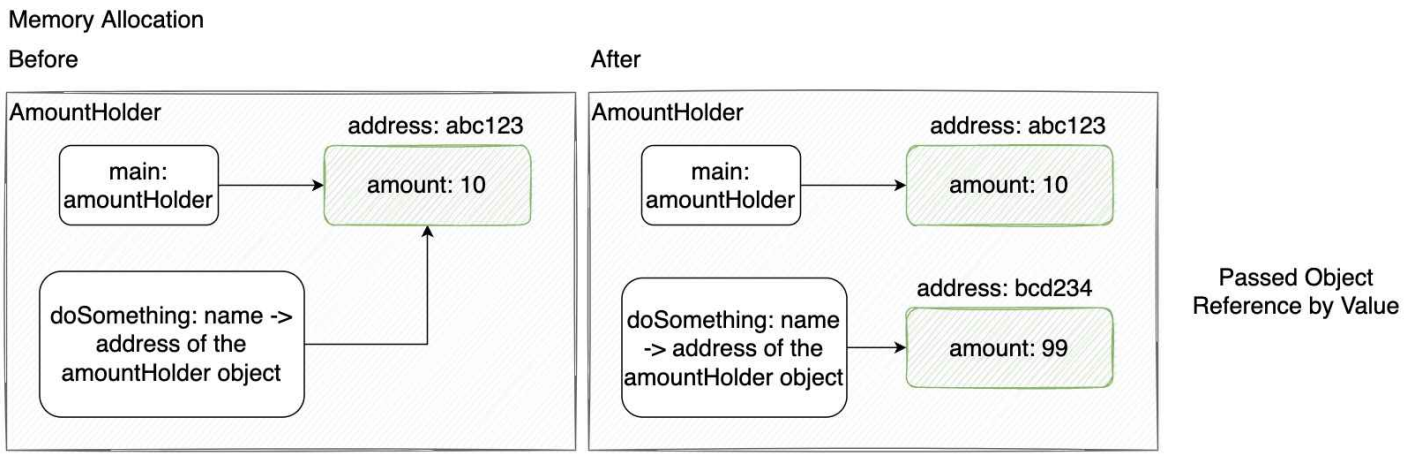
Pass by Reference (All Classes, except Wrapper class & Primitive)




```java
// Scenario 2
public class PassReferenceByValue {
    public static void addLab(StringBuilder name) { // StringBuilder is mutable
      name.append("LAB"); // Veturenix LAB
```

```
    }

    public static void main(String[] args) {
      StringBuilder name = new StringBuilder("Venturenix ");
      addLab(name);
      System.out.println(name); // print "Venturenix LAB"
    }
  } // Output: "Venturenix LAB"
```

- Note that reassigning the reference within a method will not change the original reference to the object.




```
public class PassReferenceByValue {
    public static void doSomething(AmountHolder a) {
        // the object reference of "holder" is abc123 (abc123 is Object Address)
        a = new AmountHolder(99); // will not change the original reference
        // the object reference of "holder" become bcd234  (bcd234 is Object Address)
    }

    public static void doOtherThing(AmountHolder a) {
        // find the object in heap AND set the amount
        a.setAmount(100);
    }

    public static void main(String[] args) {
        AmountHolder holder = new AmountHolder(10);
        doSomething(holder);
        System.out.println("Amount is " + holder.getAmount()); // 10, why not 99?
        doOtherThing(holder);
        System.out.println("Amount is " + holder.getAmount()); // 100
```

```
    }
} // Output: Amount is 10
```

Termination of a Method

- A method terminates (returns to the code that invoked it) under three circumstances, whichever happens first:
  a. `Completes all the statements` in the method, i.e. *void* method
  b. Reaches a `return` statement
  c. `Throws` an exception

Returning a Value

```
public class ReturningValue {
    public static void main(String[] args) {
        int c = sum(1, 2);
        System.out.println("c is " + c);
    }

    public static int sum(int a, int b) {
        return a + b;
    }
} // Output: c is 3
```

Return Type as a Custom Class

```
// Remember the *AmountHolder* class
public class AmountHolder {
    private int amount;

    public AmountHolder(int amount) {
        this.amount = amount;
    }

    public int getAmount() {
        return amount;
    }

```

```
    public void setAmount(int amount) {
        this.amount = amount;
    }
}

public class ReturningClass {
    public static void main(String[] args) {
        int amount = 10;
        AmountHolder holder = getAmountHolder(amount);
//        AmountHolder holder = new AmountHolder(amount);
        System.out.println("holder has amount " + holder.getAmount());
    }

    public static AmountHolder getAmountHolder(int amount) {
        AmountHolder amountHolder = new AmountHolder(amount);
        return amountHolder;
    }
}
```

Returning an Interface

```
// Assume we have an Animal interface
public interface Animal {
    String makeSound();
}

// And a Cow class that implements the Animal interface
public class Cow implements Animal {
    @Override
    public String makeSound() {
        return "Moooooooooo";
    }
}

public class Pig implements Animal {
    @Override
    public String makeSound() {
        return "abc";
    }
}


public class Pig implements Animal {
    @Override
    public String makeSound() {
```

```
25          return "mmm";
26      }
27  }
28
29  // We can return an interface and still be able to
30  public class ReturningInterface {
31      public static void main(String[] args) {
32          int type = 4;
33          Animal animal = getAnimal(type);
34          System.out.println(animal.makeSound());
35      }
36
37      // Cow can be return type also.
38      public static Animal getAnimal() { // just because Cow implements Animal
39          if (type == 1) {
40              return new Cow();
41          } else if (type == 2) {
42              return new Rabbit();
43          }
44          return new Pig();
45      }
46  }
```

Questions

- What are the features of a method?
- How can using methods help encapsulate logic?
- What is the purpose of Variable Arguments (varargs)?
- What are the circumstances under which a method can be terminated?
- What are the benefits of a method returning an interface?