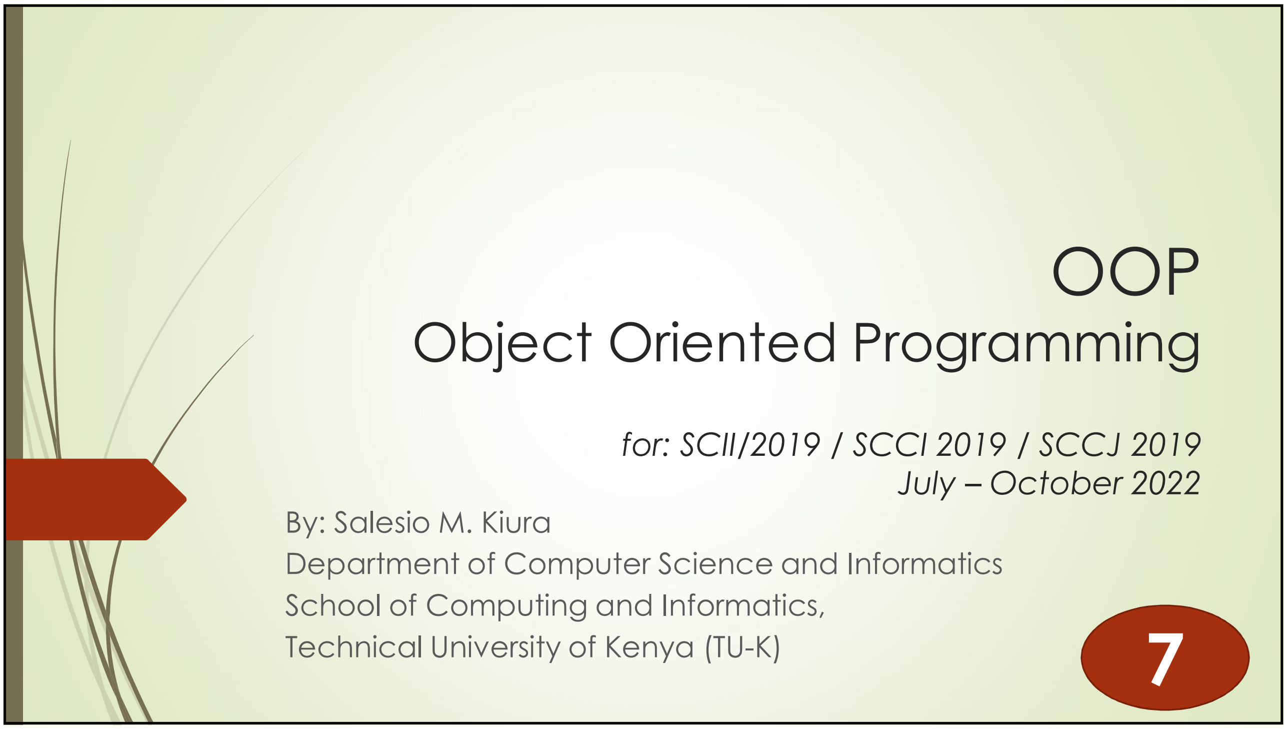
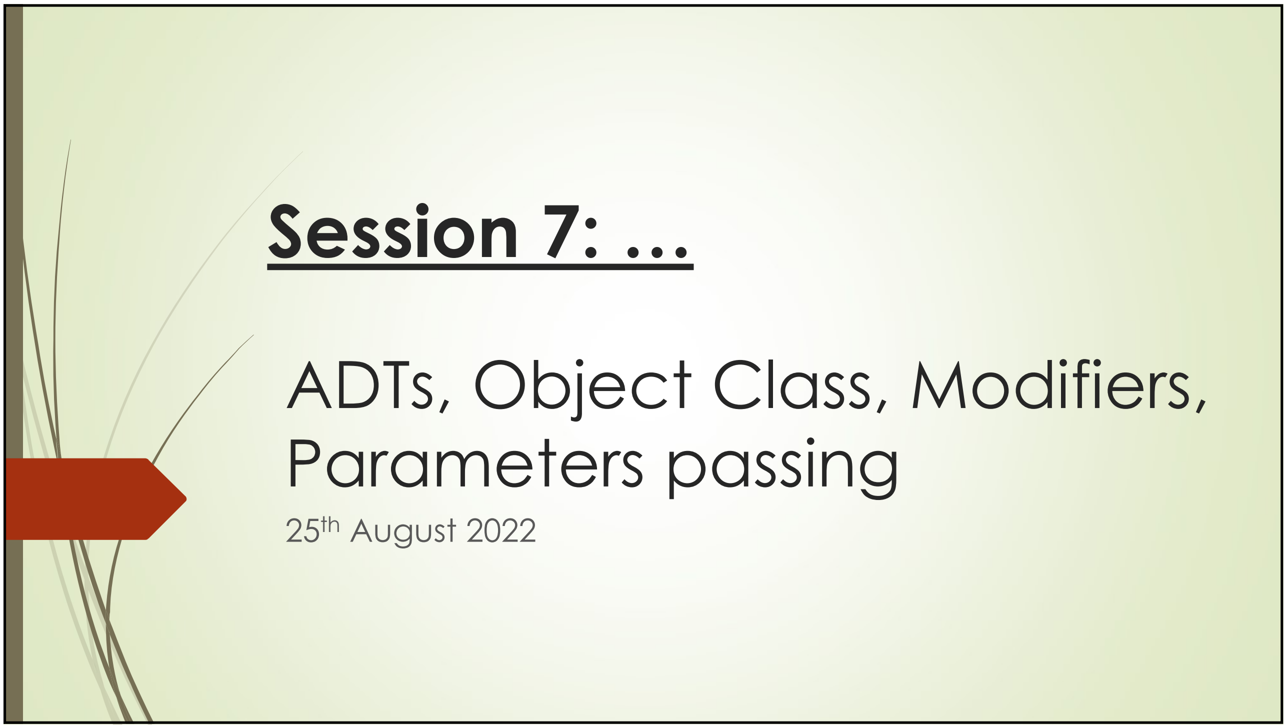


OOP
Object Oriented Programming

for: SCII/2019 / SCCI 2019 / SCCJ 2019
July – October 2022

By: Salesio M. Kiura
Department of Computer Science and Informatics
School of Computing and Informatics,
Technical University of Kenya (TU-K)

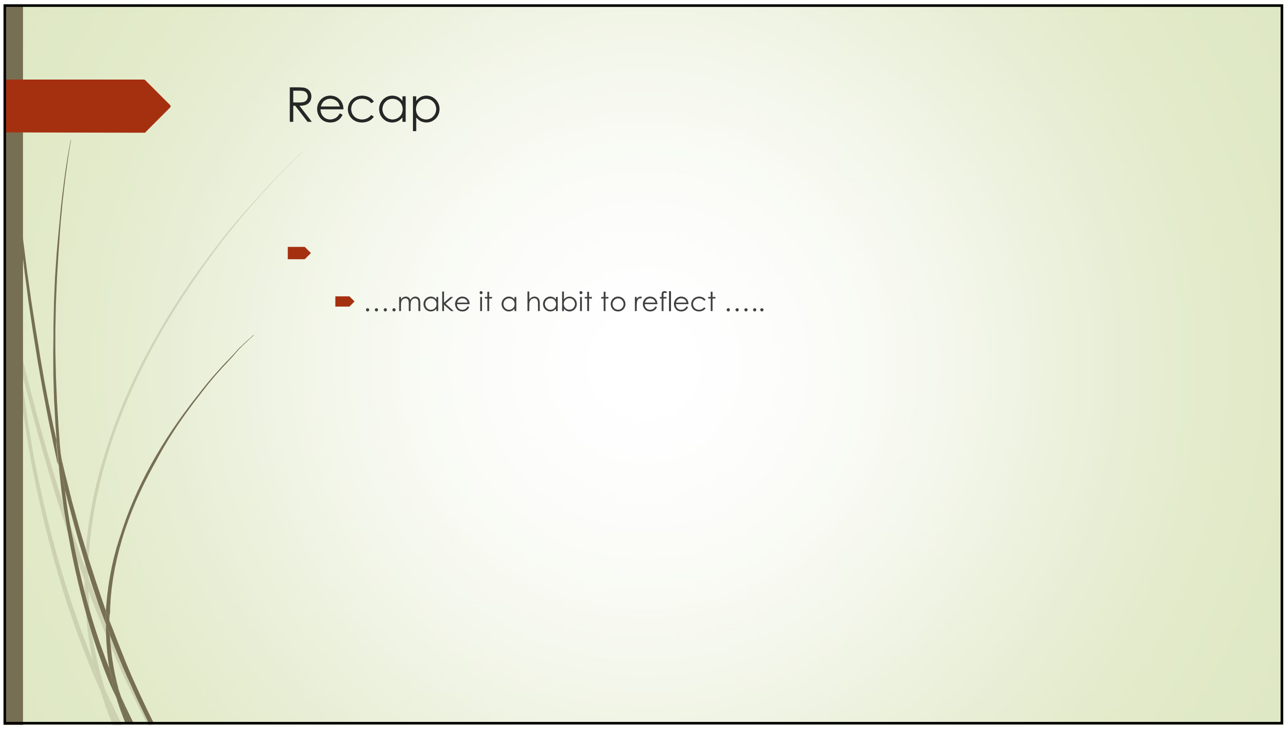
7



Session 7: …

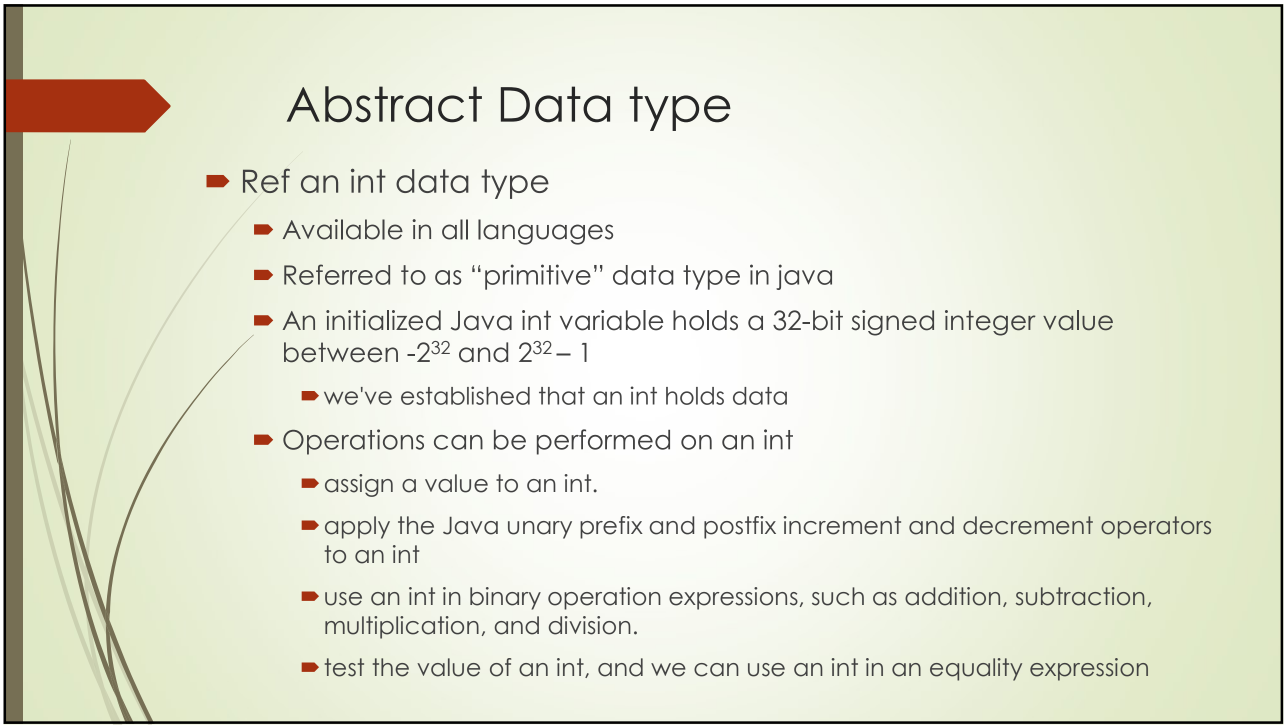
ADTs, Object Class, Modifiers, Parameters passing

25th August 2022



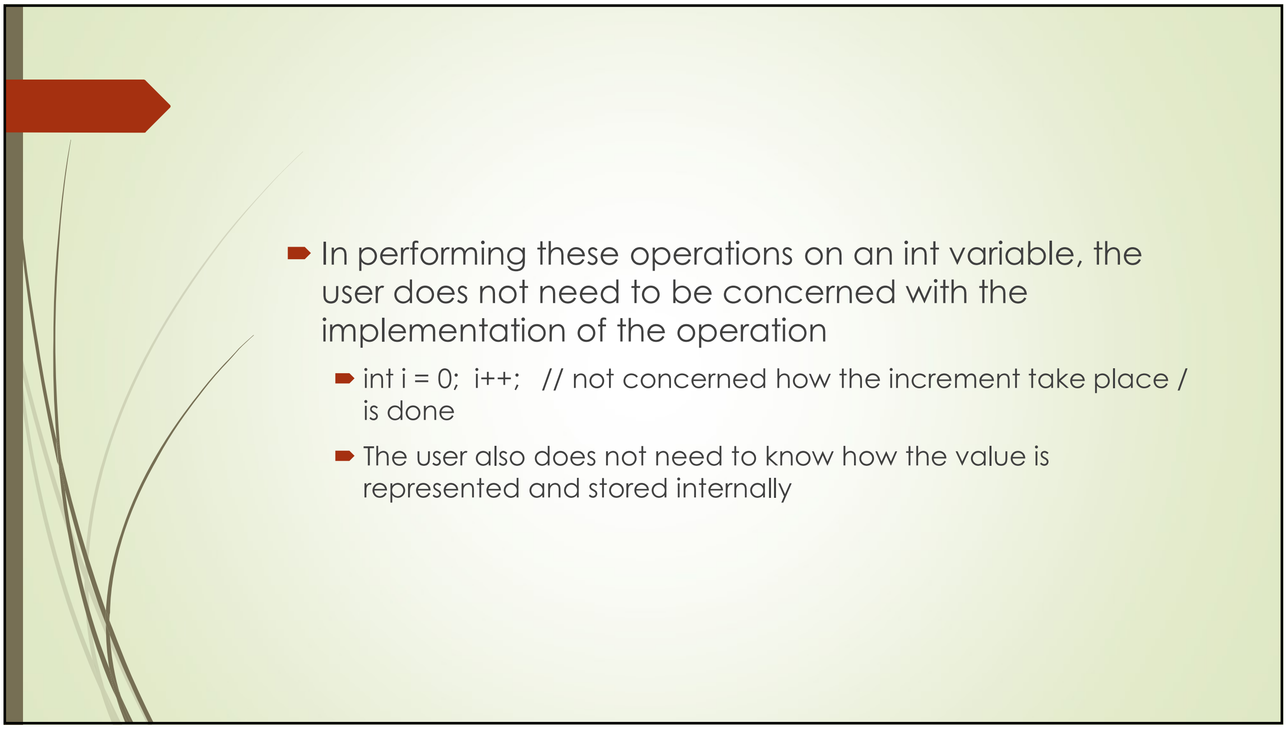
Recap

-
    - ….make it a habit to reflect …..

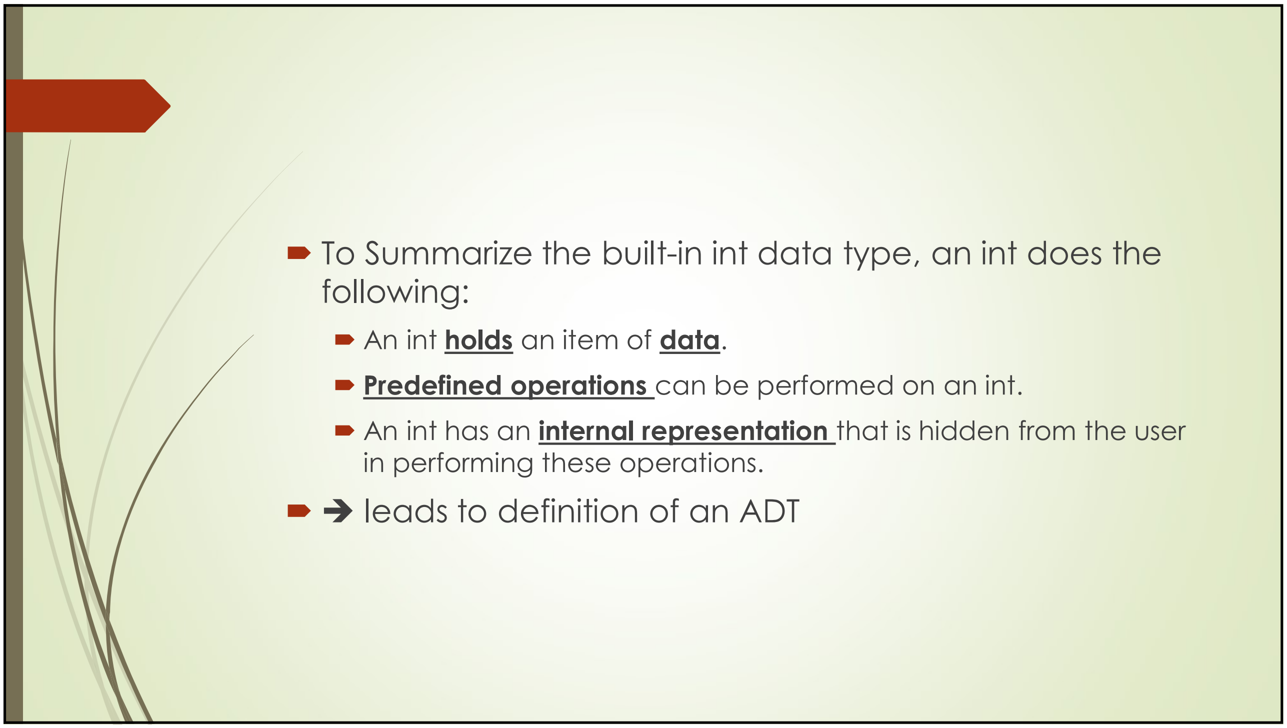


Abstract Data type

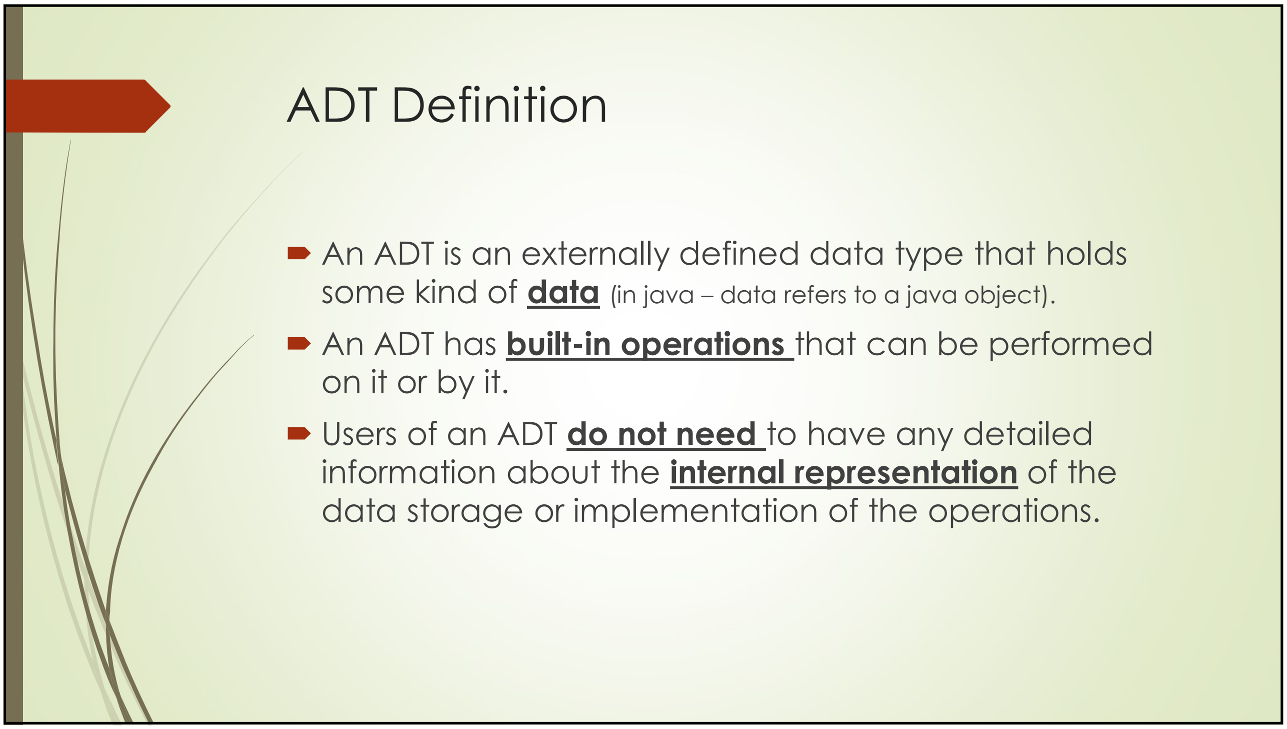
- Ref an int data type
    - Available in all languages
    - Referred to as “primitive” data type in java
    - An initialized Java int variable holds a 32-bit signed integer value between -2^{32} and 2^{32} – 1
        - we've established that an int holds data
    - Operations can be performed on an int
        - assign a value to an int.
        - apply the Java unary prefix and postfix increment and decrement operators to an int
        - use an int in binary operation expressions, such as addition, subtraction, multiplication, and division.
        - test the value of an int, and we can use an int in an equality expression



- In performing these operations on an int variable, the user does not need to be concerned with the implementation of the operation
    - int i = 0; i++;   // not concerned how the increment take place / is done
    - The user also does not need to know how the value is represented and stored internally

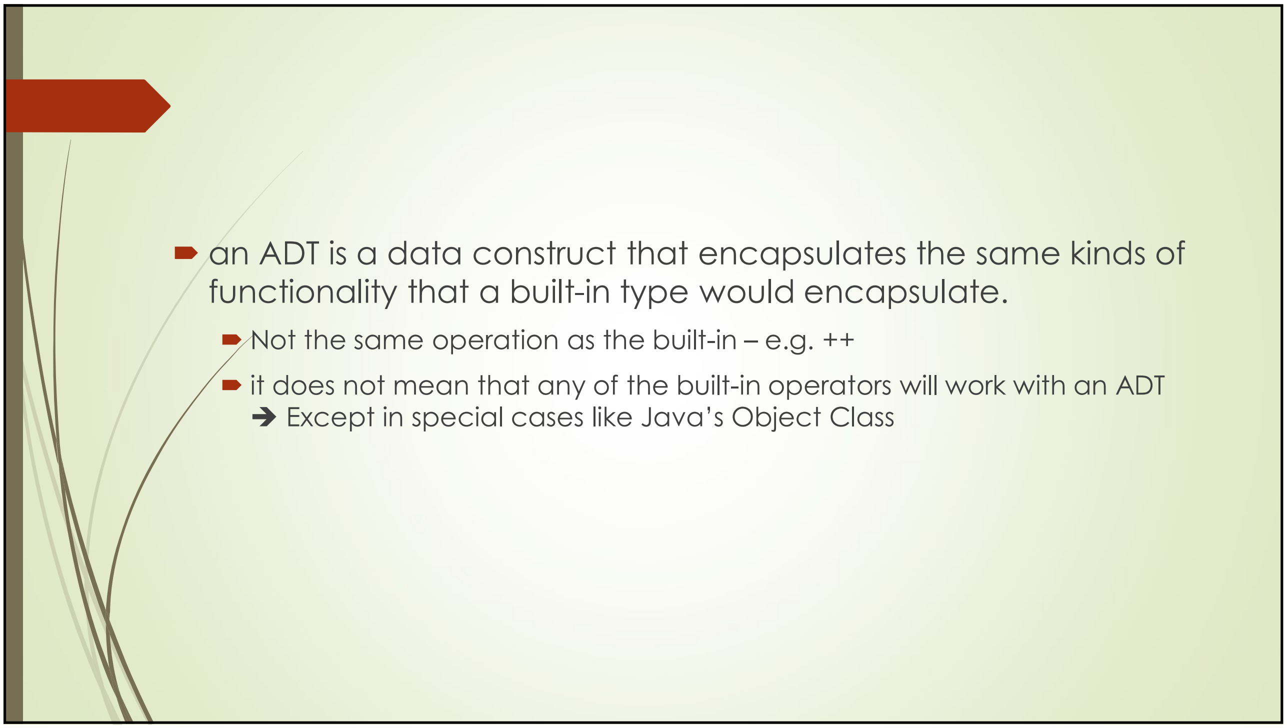


- To Summarize the built-in int data type, an int does the following:
    - An int **holds** an item of **data**.
    - **Predefined operations** can be performed on an int.
    - An int has an **internal representation** that is hidden from the user in performing these operations.
- ➔ leads to definition of an ADT

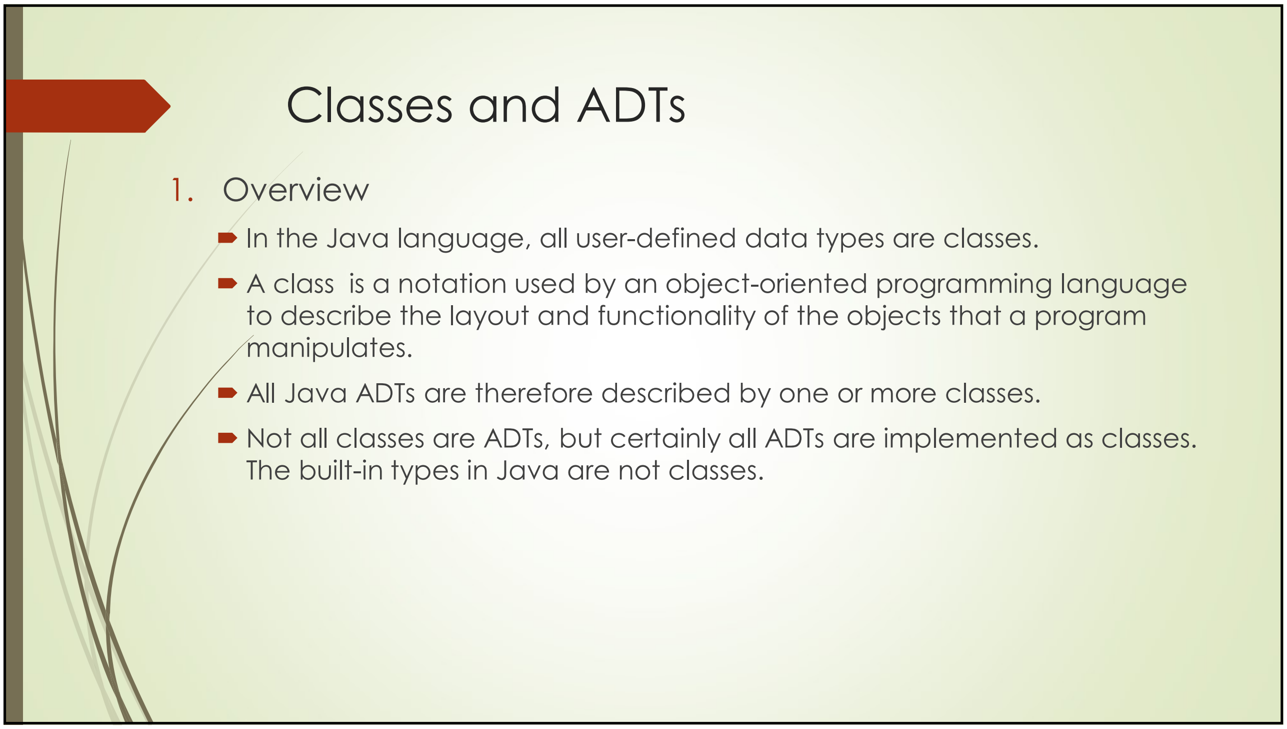


ADT Definition

- An ADT is an externally defined data type that holds some kind of **data** (in java – data refers to a java object).
- An ADT has **built-in operations** that can be performed on it or by it.
- Users of an ADT **do not need** to have any detailed information about the **internal representation** of the data storage or implementation of the operations.



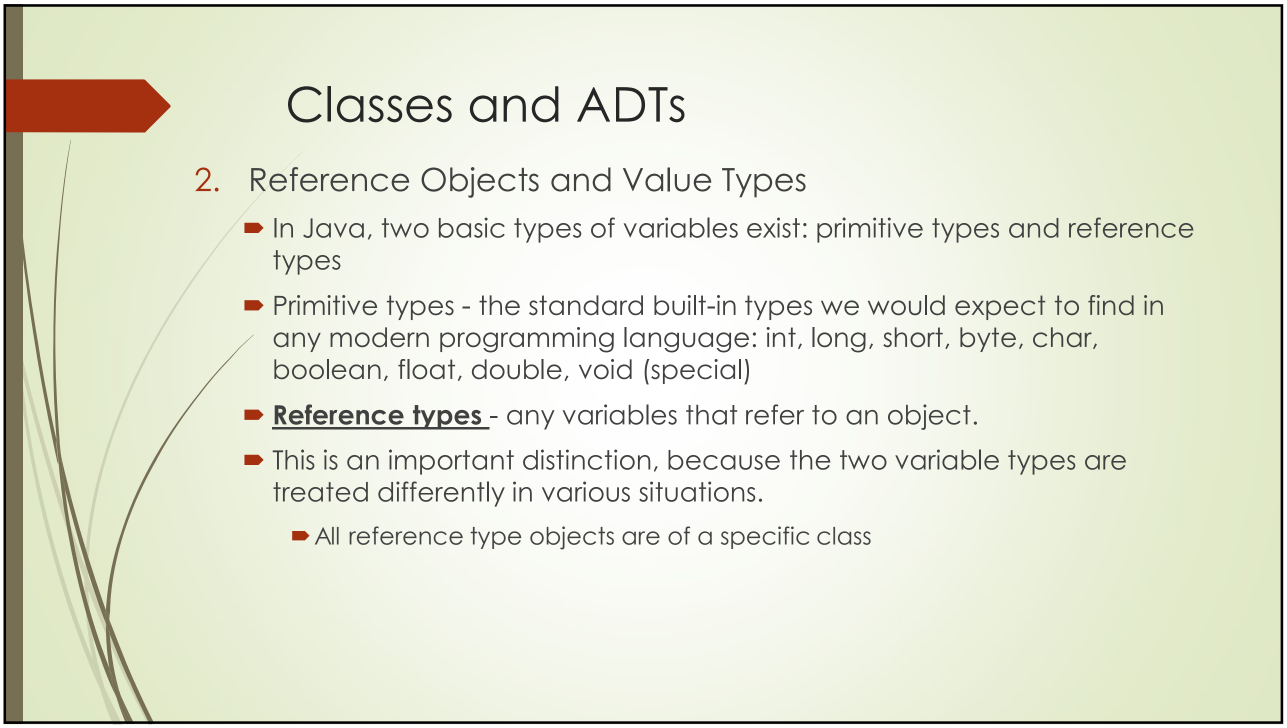
- an ADT is a data construct that encapsulates the same kinds of functionality that a built-in type would encapsulate.
  - Not the same operation as the built-in – e.g. ++
  - it does not mean that any of the built-in operators will work with an ADT
    ➔ Except in special cases like Java's Object Class



Classes and ADTs

1. Overview

- In the Java language, all user-defined data types are classes.
- A class is a notation used by an object-oriented programming language to describe the layout and functionality of the objects that a program manipulates.
- All Java ADTs are therefore described by one or more classes.
- Not all classes are ADTs, but certainly all ADTs are implemented as classes. The built-in types in Java are not classes.

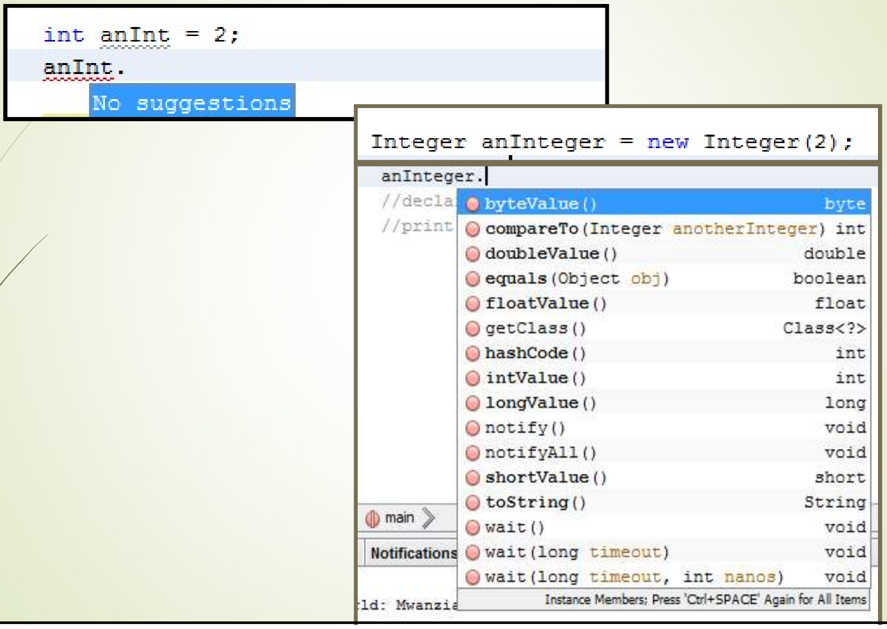


Classes and ADTs

2. Reference Objects and Value Types

- In Java, two basic types of variables exist: primitive types and reference types
- Primitive types - the standard built-in types we would expect to find in any modern programming language: int, long, short, byte, char, boolean, float, double, void (special)
- **Reference types** - any variables that refer to an object.
- This is an important distinction, because the two variable types are treated differently in various situations.
  - All reference type objects are of a specific class

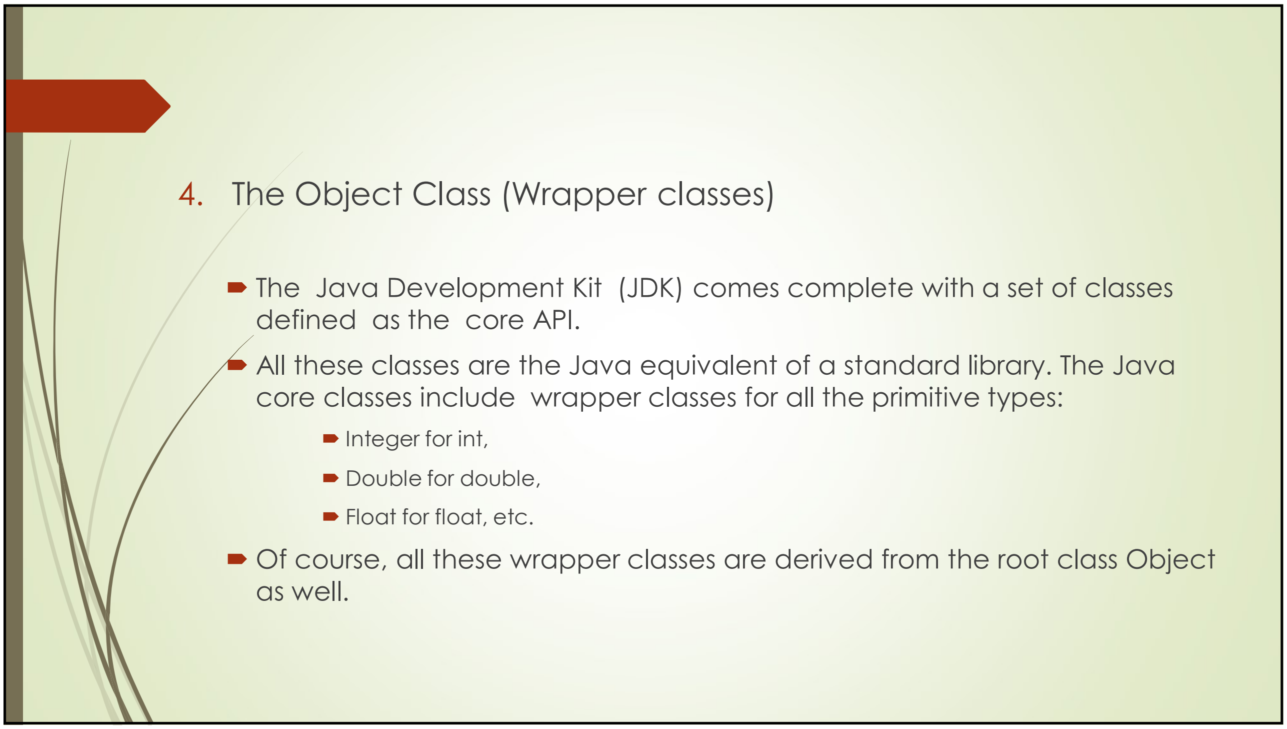
- Compare and Contrast the following:



```
int anInt = 2;
anInt.
    No suggestions
```

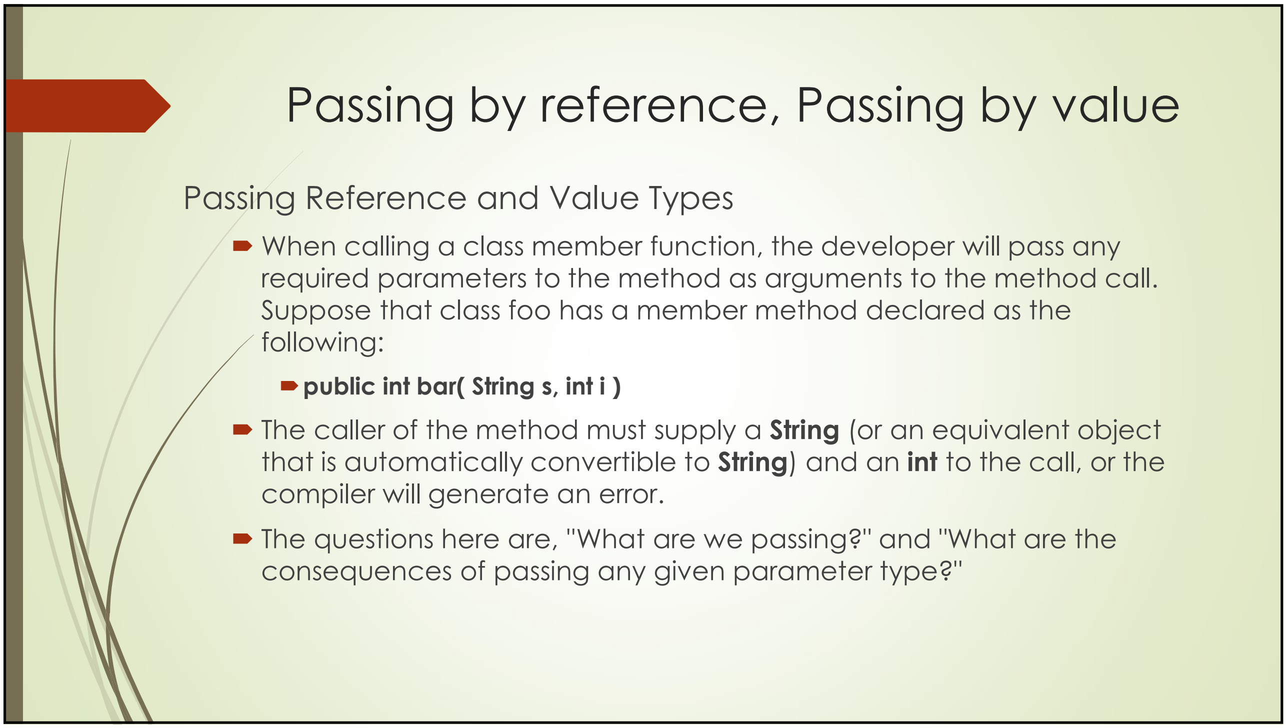
```
Integer anInteger = new Integer(2);
anInteger.
 byteValue()                                   byte
 compareTo(Integer anotherInteger)              int
 doubleValue()                               double
 equals(Object obj)                         boolean
 floatValue()                                 float
 getClass()                                Class<?>
 hashCode()                                     int
 intValue()                                     int
 longValue()                                   long
 notify()                                      void
 notifyAll()                                   void
 shortValue()                                 short
 toString()                                  String
 wait()                                        void
 wait(long timeout)                            void
 wait(long timeout, int nanos)                 void
Instance Members; Press 'Ctrl+SPACE' Again for All Items
```

3. The Object Class
   - All classes in Java are derived from the root class Object
   - So, given the rules of inheritance and polymorphism, an Object class variable can refer to any reference object of any class.
   - One of Java's strengths is the fact that it uses a polymorphic model wherein all classes are derived from a common root.
   - All objects share a common base Application Programming Interface (API)



4. The Object Class (Wrapper classes)

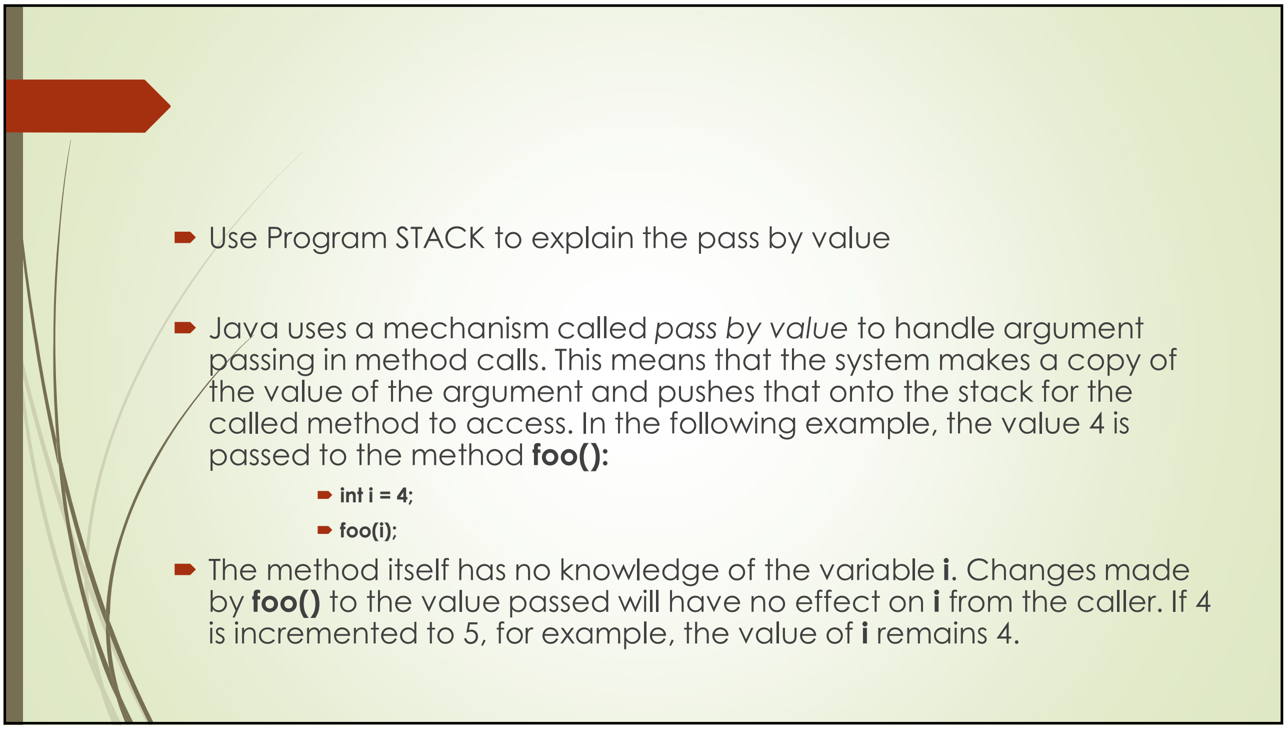
- The Java Development Kit (JDK) comes complete with a set of classes defined as the core API.
- All these classes are the Java equivalent of a standard library. The Java core classes include wrapper classes for all the primitive types:
    - Integer for int,
    - Double for double,
    - Float for float, etc.
- Of course, all these wrapper classes are derived from the root class Object as well.



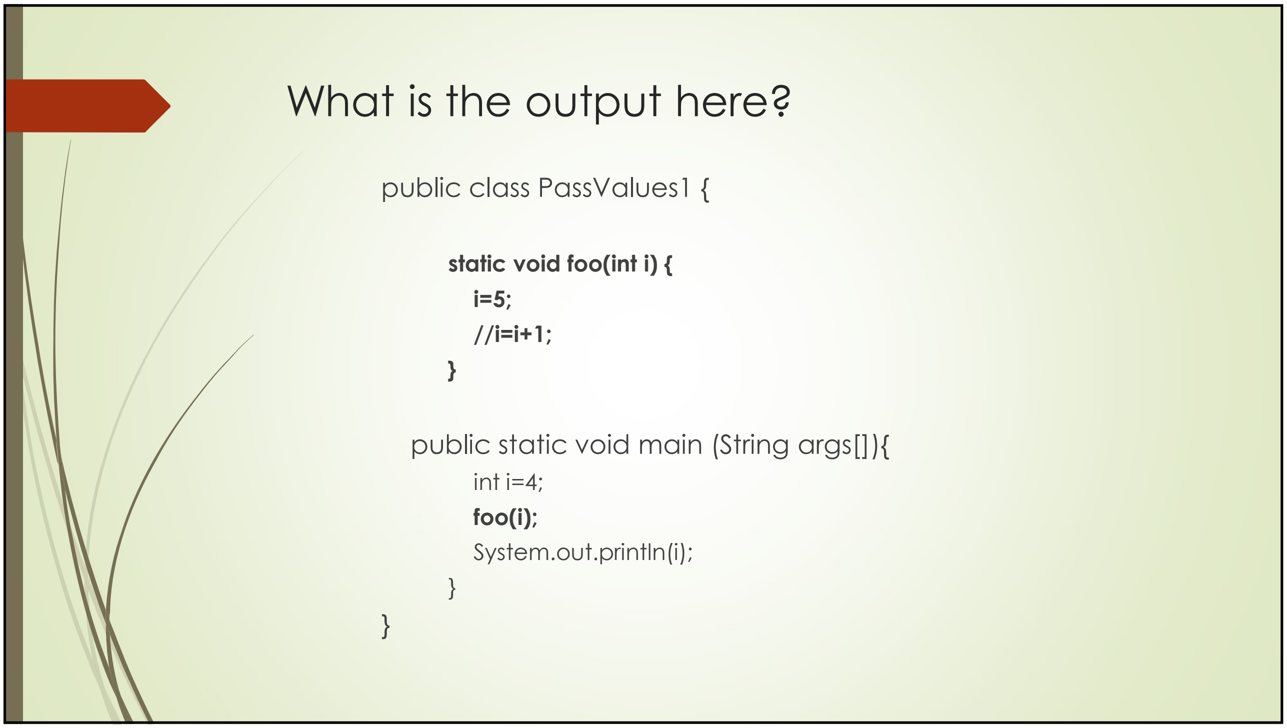
Passing by reference, Passing by value

Passing Reference and Value Types

- When calling a class member function, the developer will pass any required parameters to the method as arguments to the method call. Suppose that class foo has a member method declared as the following:
    - **public int bar(String s, int i)**
- The caller of the method must supply a **String** (or an equivalent object that is automatically convertible to **String**) and an **int** to the call, or the compiler will generate an error.
- The questions here are, "What are we passing?" and "What are the consequences of passing any given parameter type?"

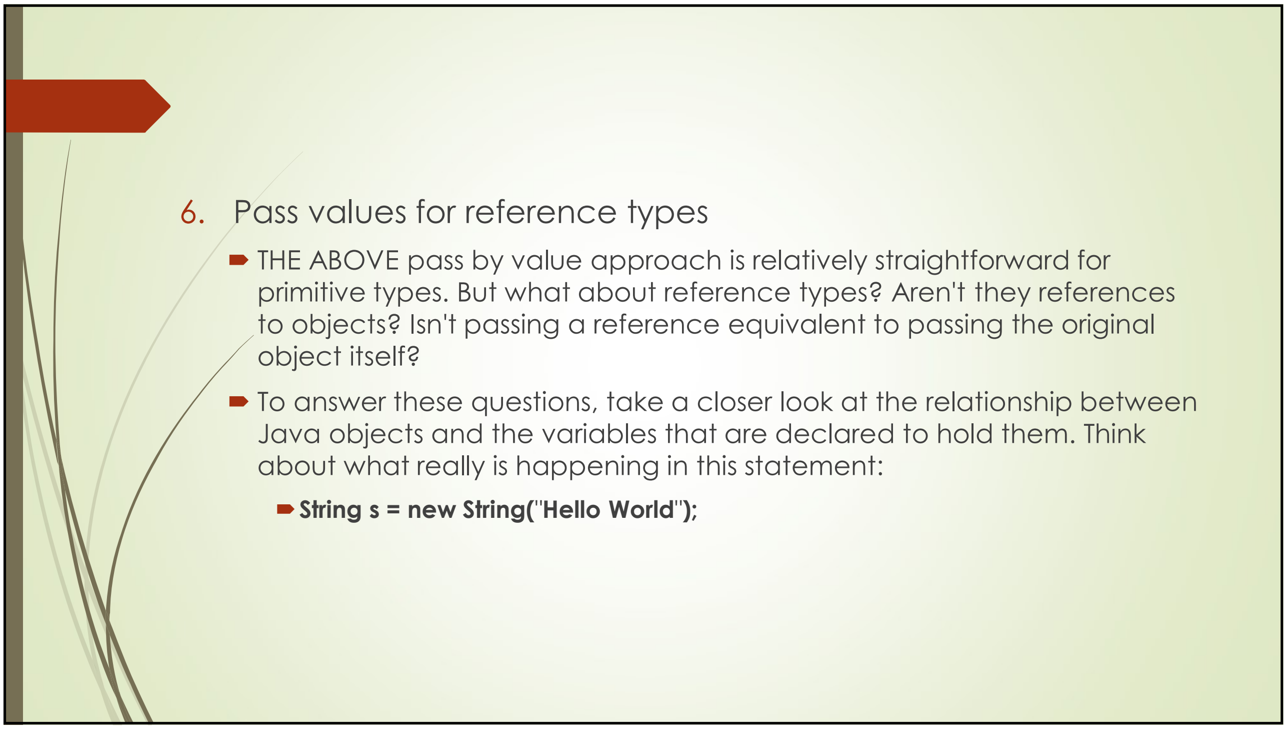


- Use Program STACK to explain the pass by value
- Java uses a mechanism called *pass by value* to handle argument passing in method calls. This means that the system makes a copy of the value of the argument and pushes that onto the stack for the called method to access. In the following example, the value 4 is passed to the method **foo():**
  - **int i = 4;**
  - **foo(i);**
- The method itself has no knowledge of the variable **i**. Changes made by **foo()** to the value passed will have no effect on **i** from the caller. If 4 is incremented to 5, for example, the value of **i** remains 4.



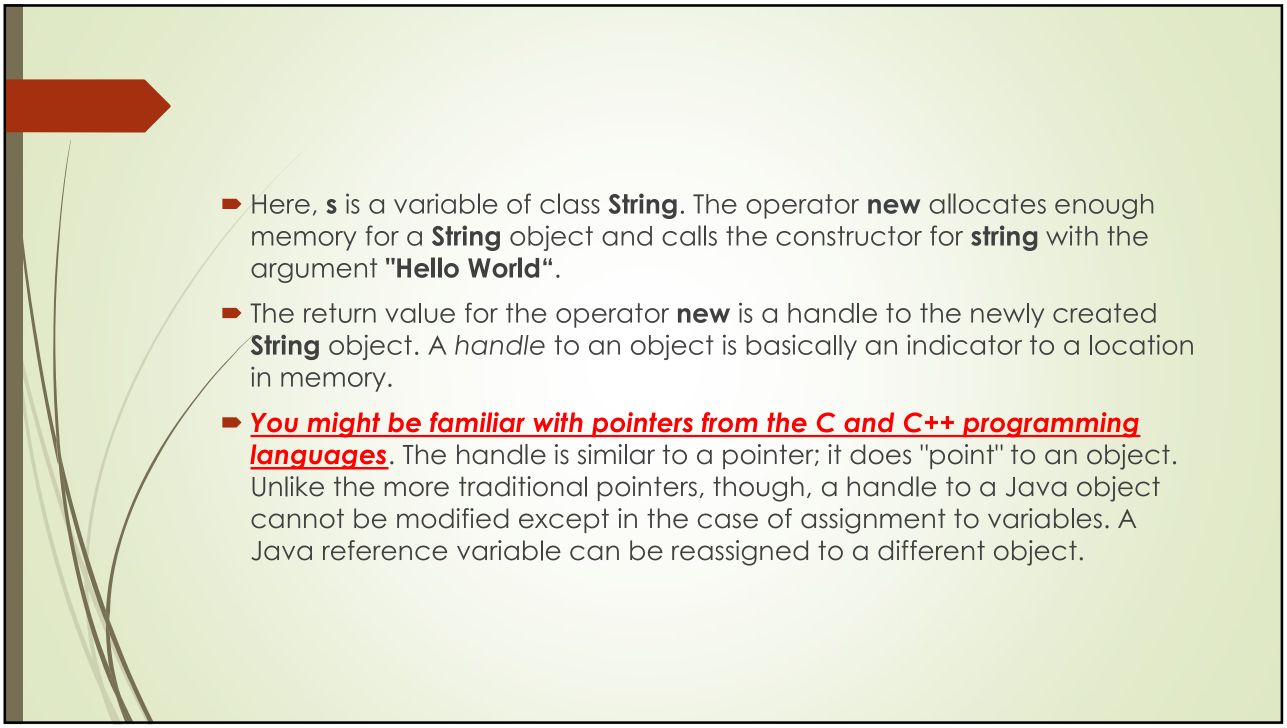
What is the output here?

```
public class PassValues1 {

      static void foo(int i) {
        i=5;
        //i=i+1;
      }

   public static void main (String args[]){
        int i=4;
        foo(i);
        System.out.println(i);
      }
}
```

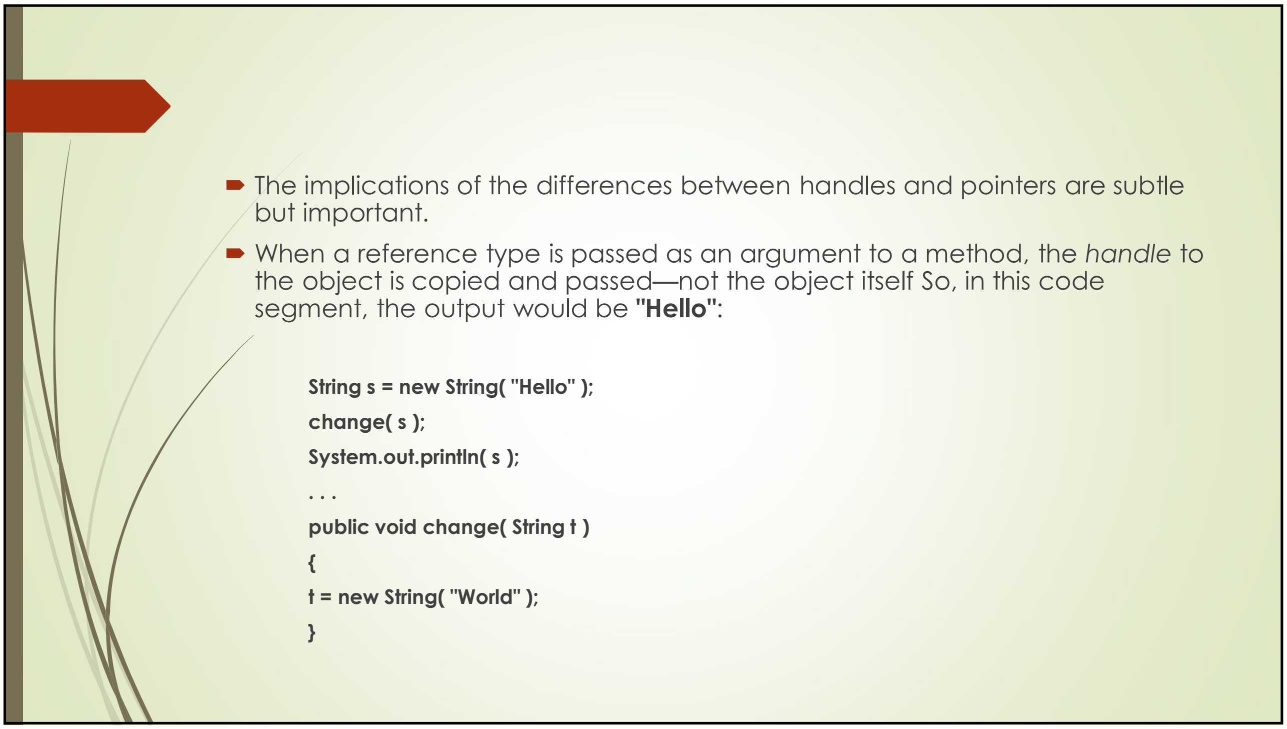


6. Pass values for reference types

- THE ABOVE pass by value approach is relatively straightforward for primitive types. But what about reference types? Aren't they references to objects? Isn't passing a reference equivalent to passing the original object itself?
- To answer these questions, take a closer look at the relationship between Java objects and the variables that are declared to hold them. Think about what really is happening in this statement:
  - **String s = new String("Hello World");**

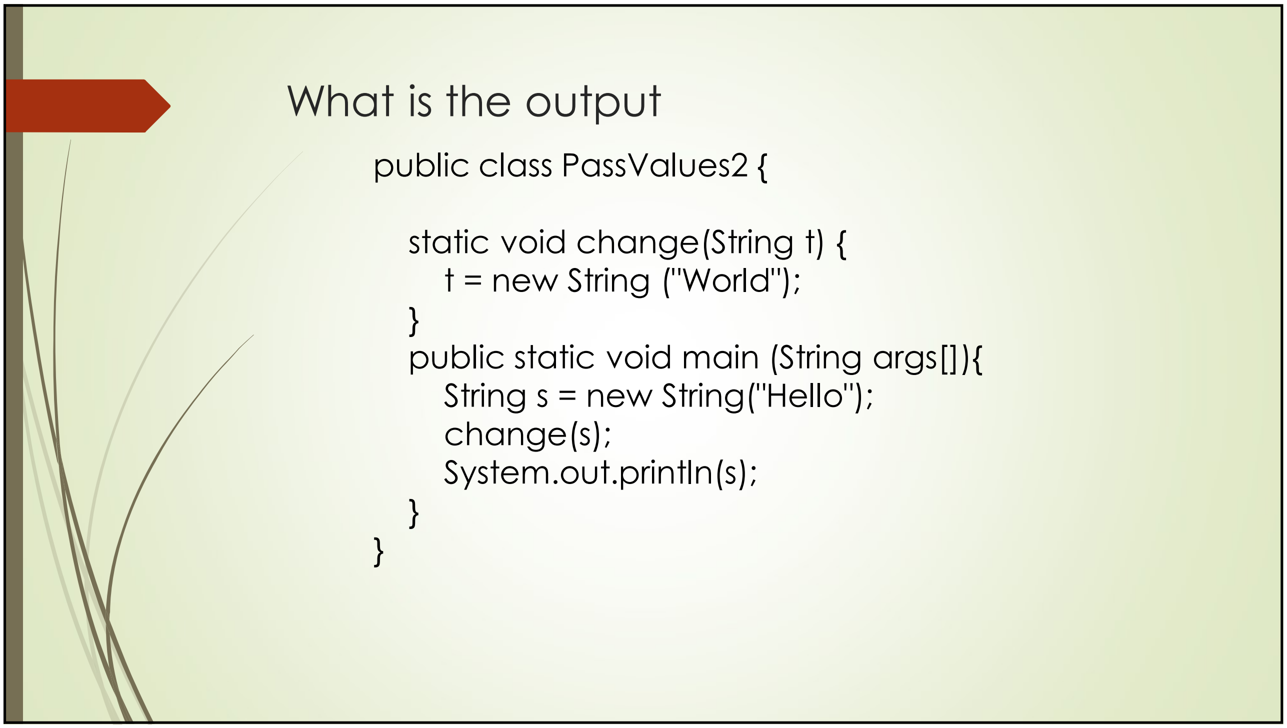


- Here, **s** is a variable of class **String**. The operator **new** allocates enough memory for a **String** object and calls the constructor for **string** with the argument **"Hello World“**.
- The return value for the operator **new** is a handle to the newly created **String** object. A *handle* to an object is basically an indicator to a location in memory.
- ***You might be familiar with pointers from the C and C++ programming languages***. The handle is similar to a pointer; it does "point" to an object. Unlike the more traditional pointers, though, a handle to a Java object cannot be modified except in the case of assignment to variables. A Java reference variable can be reassigned to a different object.



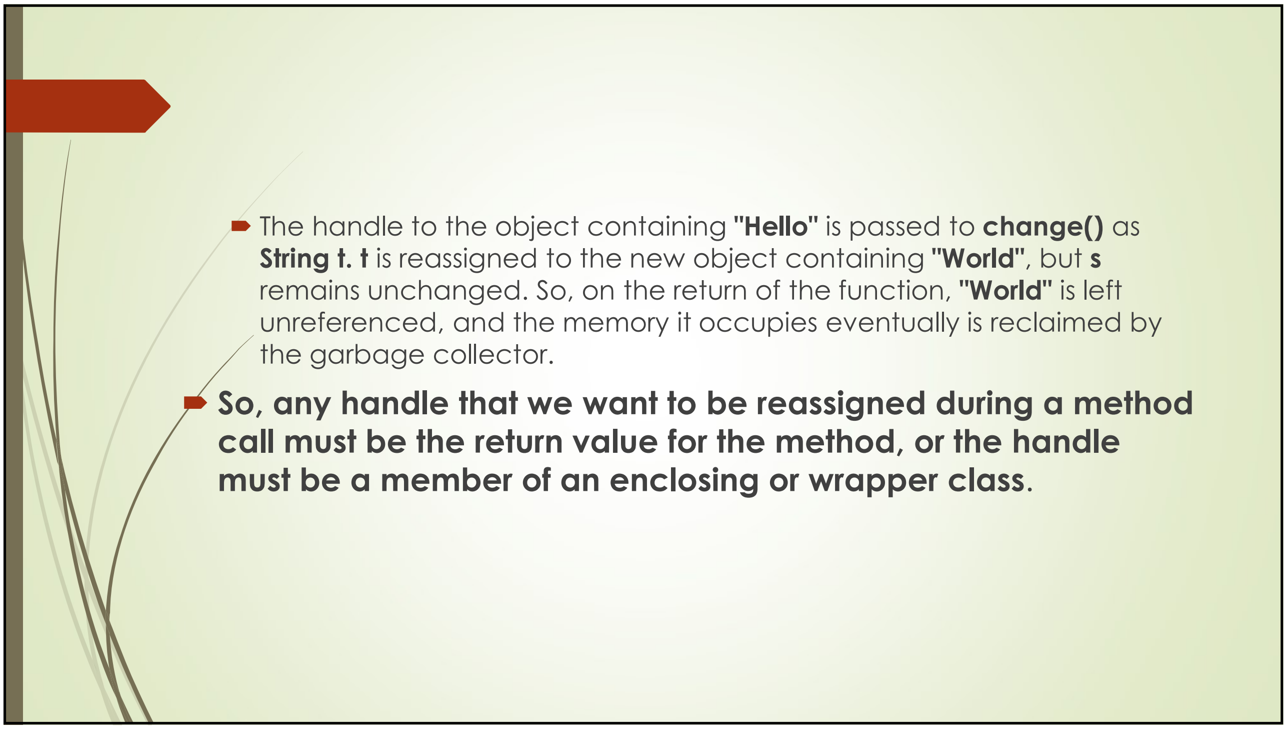
- The implications of the differences between handles and pointers are subtle but important.
- When a reference type is passed as an argument to a method, the *handle* to the object is copied and passed—not the object itself So, in this code segment, the output would be **"Hello"**:

```
String s = new String( "Hello" );
change( s );
System.out.println( s );
. . .
public void change( String t )
{
t = new String( "World" );
}
```



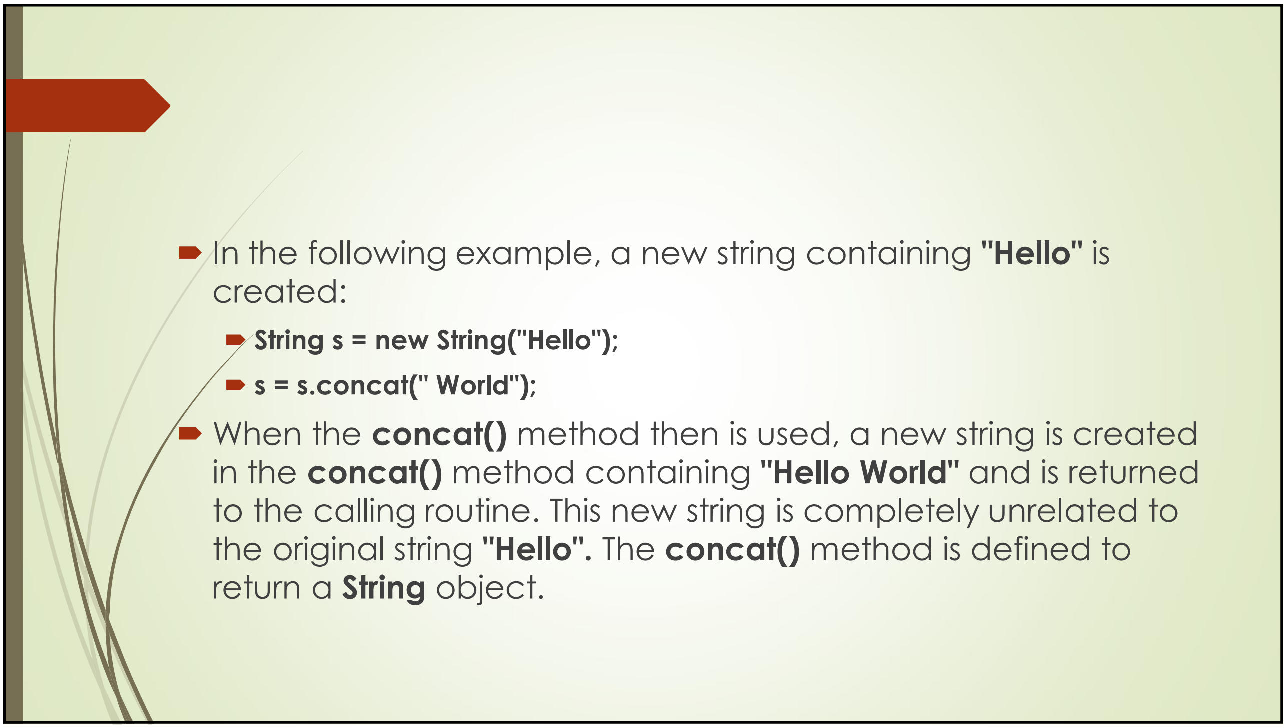
What is the output

```
public class PassValues2 {

   static void change(String t) {
      t = new String ("World");
   }
   public static void main (String args[]){
      String s = new String("Hello");
      change(s);
      System.out.println(s);
   }
}
```

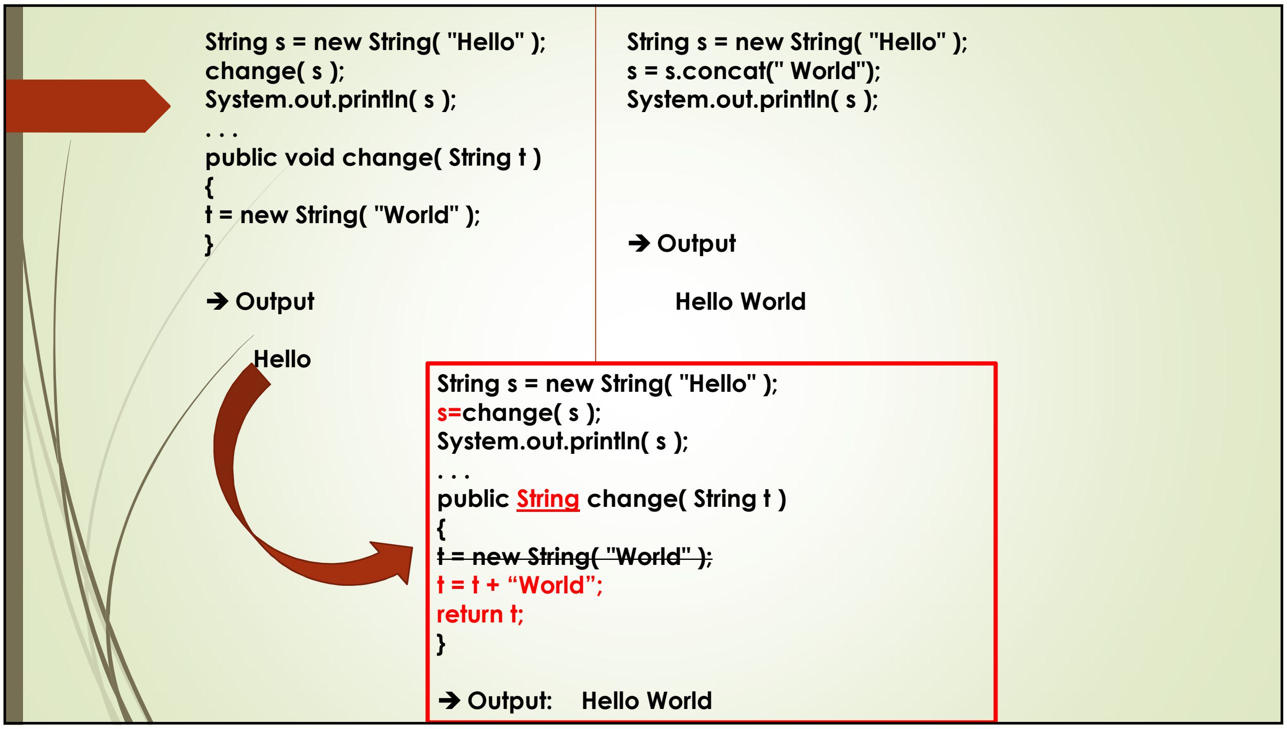


- The handle to the object containing **"Hello"** is passed to **change()** as **String t. t** is reassigned to the new object containing **"World"**, but **s** remains unchanged. So, on the return of the function, **"World"** is left unreferenced, and the memory it occupies eventually is reclaimed by the garbage collector.

- **So, any handle that we want to be reassigned during a method call must be the return value for the method, or the handle must be a member of an enclosing or wrapper class**.



- In the following example, a new string containing **"Hello"** is created:
  - **String s = new String("Hello");**
  - **s = s.concat(" World");**
- When the **concat()** method then is used, a new string is created in the **concat()** method containing **"Hello World"** and is returned to the calling routine. This new string is completely unrelated to the original string **"Hello".** The **concat()** method is defined to return a **String** object.



```
String s = new String( "Hello" );
change( s );
System.out.println( s );
. . .
public void change( String t )
{
t = new String( "World" );
}
```

➔ Output

```
Hello
```

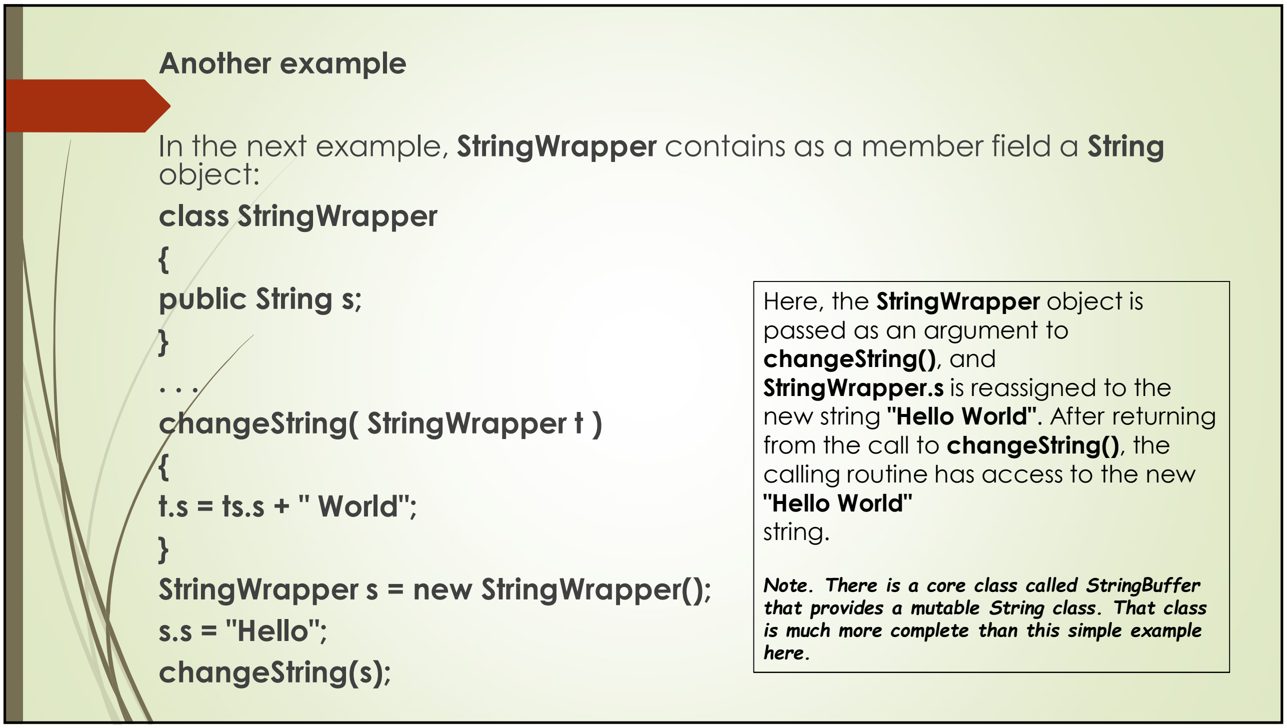
```
String s = new String( "Hello" );
s = s.concat(" World");
System.out.println( s );
```

➔ Output

```
Hello World
```

```
String s = new String( "Hello" );
s=change( s );
System.out.println( s );
. . .
public String change( String t )
{
~~t = new String( "World" );~~
t = t + "World";
return t;
}
```

➔ Output: Hello World



Another example

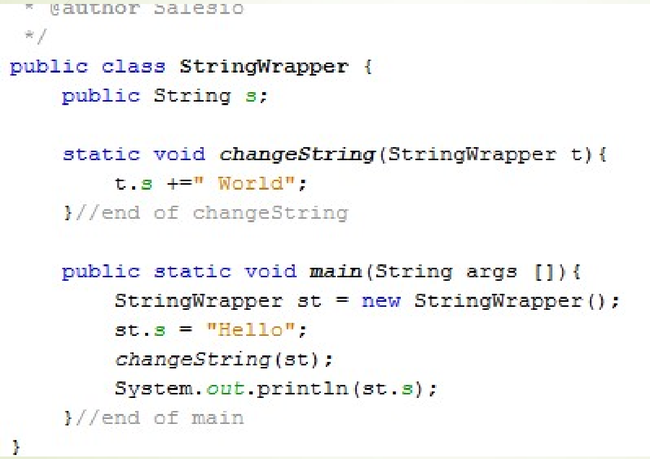
In the next example, **StringWrapper** contains as a member field a **String** object:

```
class StringWrapper
{
public String s;
}
. . .
changeString( StringWrapper t )
{
t.s = ts.s + " World";
}
StringWrapper s = new StringWrapper();
s.s = "Hello";
changeString(s);
```

Here, the **StringWrapper** object is passed as an argument to **changeString()**, and **StringWrapper.s** is reassigned to the new string **"Hello World"**. After returning from the call to **changeString()**, the calling routine has access to the new **"Hello World"** string.

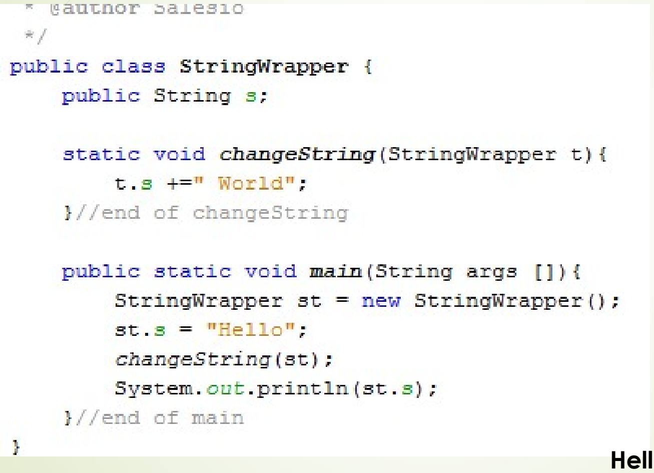
Note. There is a core class called StringBuffer that provides a mutable String class. That class is much more complete than this simple example here.

What’s the output here? (1)



```
 * @author Salesio
 */
public class StringWrapper {
    public String s;

    static void changeString(StringWrapper t){
        t.s +=" World";
    }//end of changeString

    public static void main(String args []){
        StringWrapper st = new StringWrapper();
        st.s = "Hello";
        changeString(st);
        System.out.println(st.s);
    }//end of main
}
```

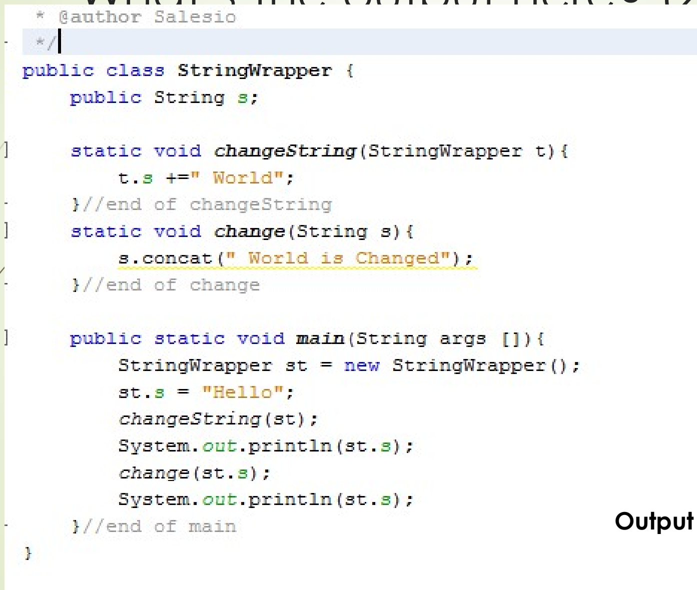
What’s the output here? (1_answer)



```
 * @author Salesio
 */
public class StringWrapper {
    public String s;

    static void changeString(StringWrapper t){
        t.s +=" World";
    }//end of changeString

    public static void main(String args []){
        StringWrapper st = new StringWrapper();
        st.s = "Hello";
        changeString(st);
        System.out.println(st.s);
    }//end of main
}
```

Hello World

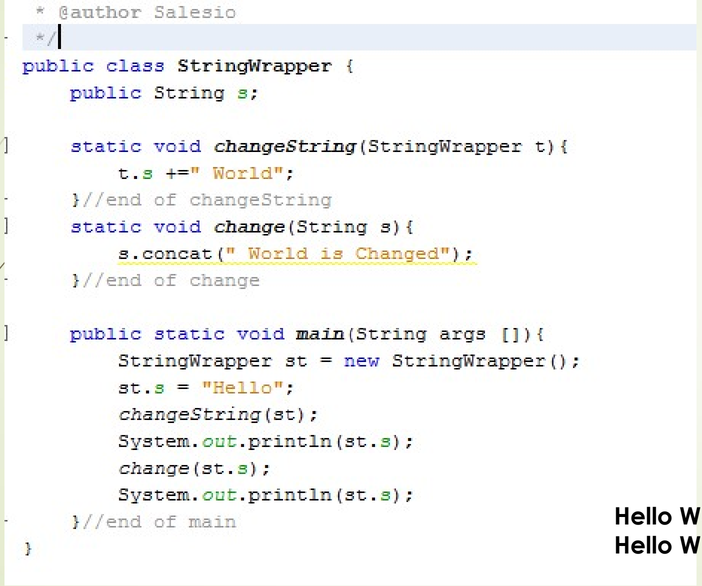
What's the output here? (2)



```
 * @author Salesio
 */
public class StringWrapper {
    public String s;

    static void changeString(StringWrapper t){
        t.s +=" World";
    }//end of changeString
    static void change(String s){
        s.concat(" World is Changed");
    }//end of change

    public static void main(String args []){
        StringWrapper st = new StringWrapper();
        st.s = "Hello";
        changeString(st);
        System.out.println(st.s);
        change(st.s);
        System.out.println(st.s);
    }//end of main
}
```

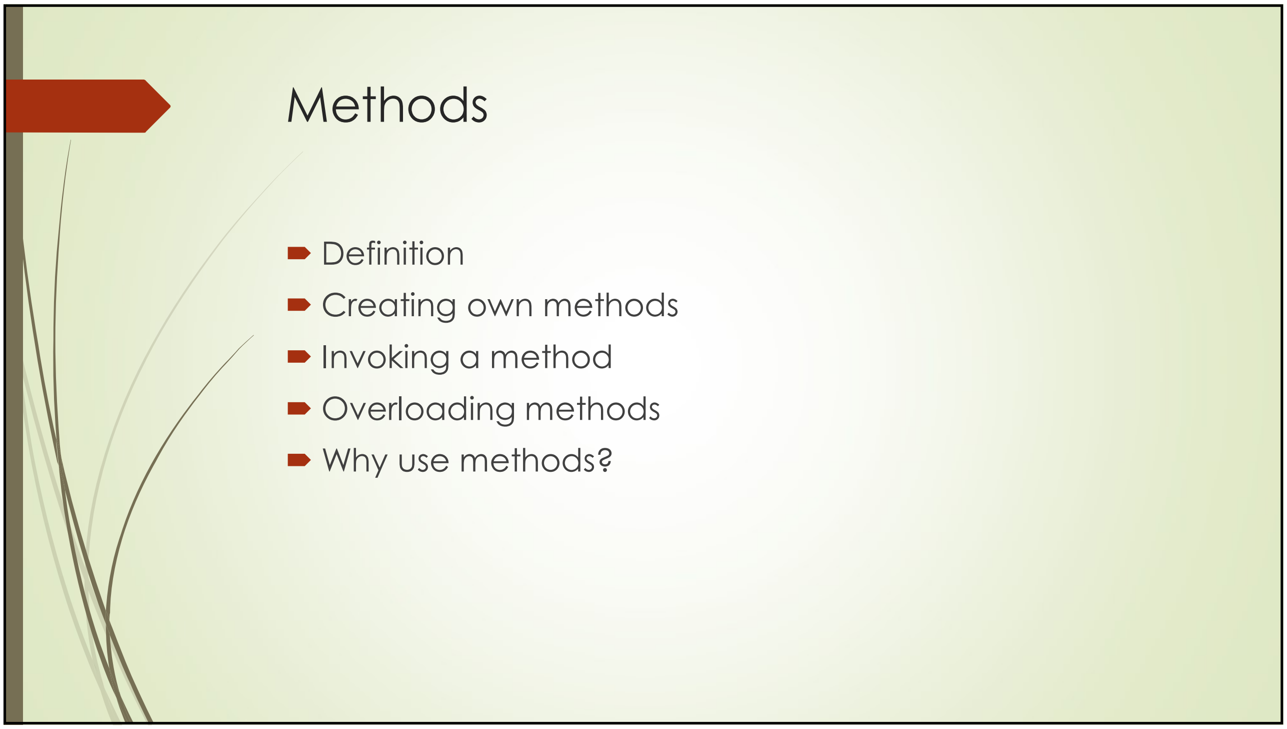
Output ?

What's the output here? (2_answer)



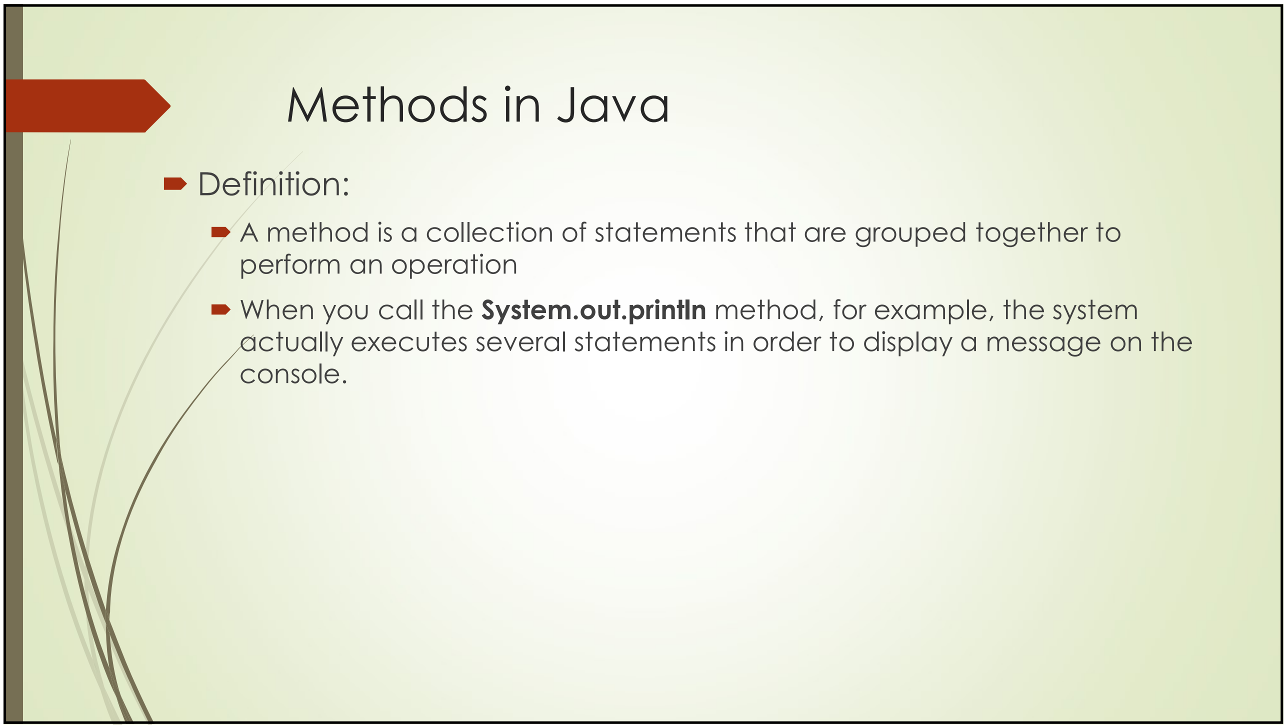
```
 * @author Salesio
 */
public class StringWrapper {
    public String s;

    static void changeString(StringWrapper t){
        t.s +=" World";
    }//end of changeString
    static void change(String s){
        s.concat(" World is Changed");
    }//end of change

    public static void main(String args []){
        StringWrapper st = new StringWrapper();
        st.s = "Hello";
        changeString(st);
        System.out.println(st.s);
        change(st.s);
        System.out.println(st.s);
    }//end of main
}
```

Hello World
Hello World



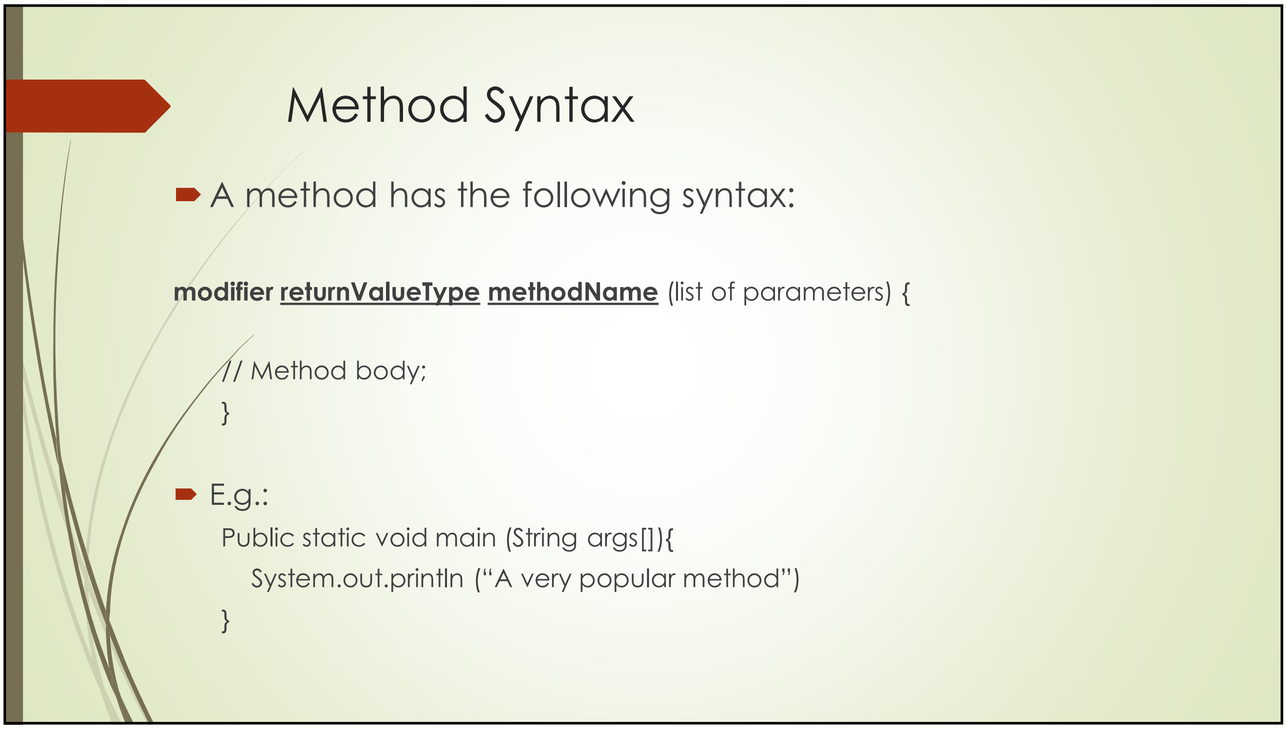
Methods

- Definition
- Creating own methods
- Invoking a method
- Overloading methods
- Why use methods?



Methods in Java

- Definition:
  - A method is a collection of statements that are grouped together to perform an operation
  - When you call the **System.out.println** method, for example, the system actually executes several statements in order to display a message on the console.



Method Syntax

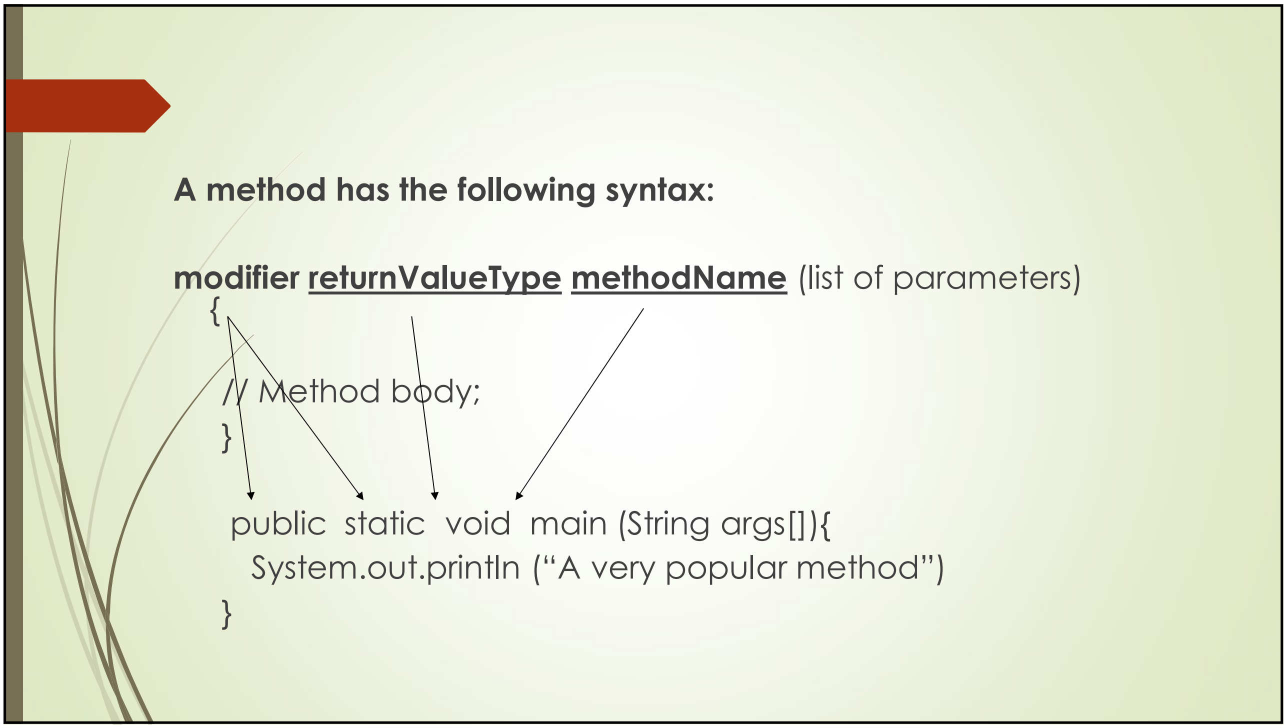
- A method has the following syntax:

modifier **<u>returnValueType</u>** **<u>methodName</u>** (list of parameters) {

```
    // Method body;
    }
```

- E.g.:

```
Public static void main (String args[]){
   System.out.println ("A very popular method")
}
```



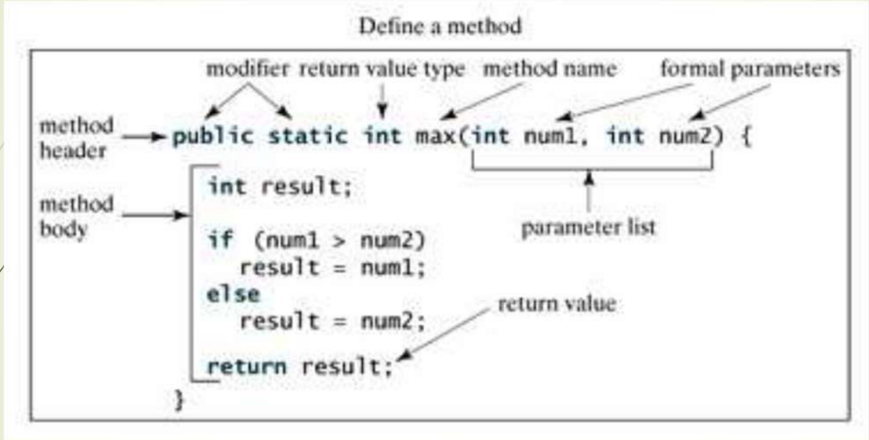
A method has the following syntax:

modifier **<u>returnValueType</u>** **<u>methodName</u>** (list of parameters)
{

```
    // Method body;
    }

     public  static  void  main (String args[]){
       System.out.println ("A very popular method")
    }
```

- Exercise
  - Write a method that given two integers, returns the greater one
  - Write a method that given two integers, returns the greater one, otherwise indicates they are equal

From the notes

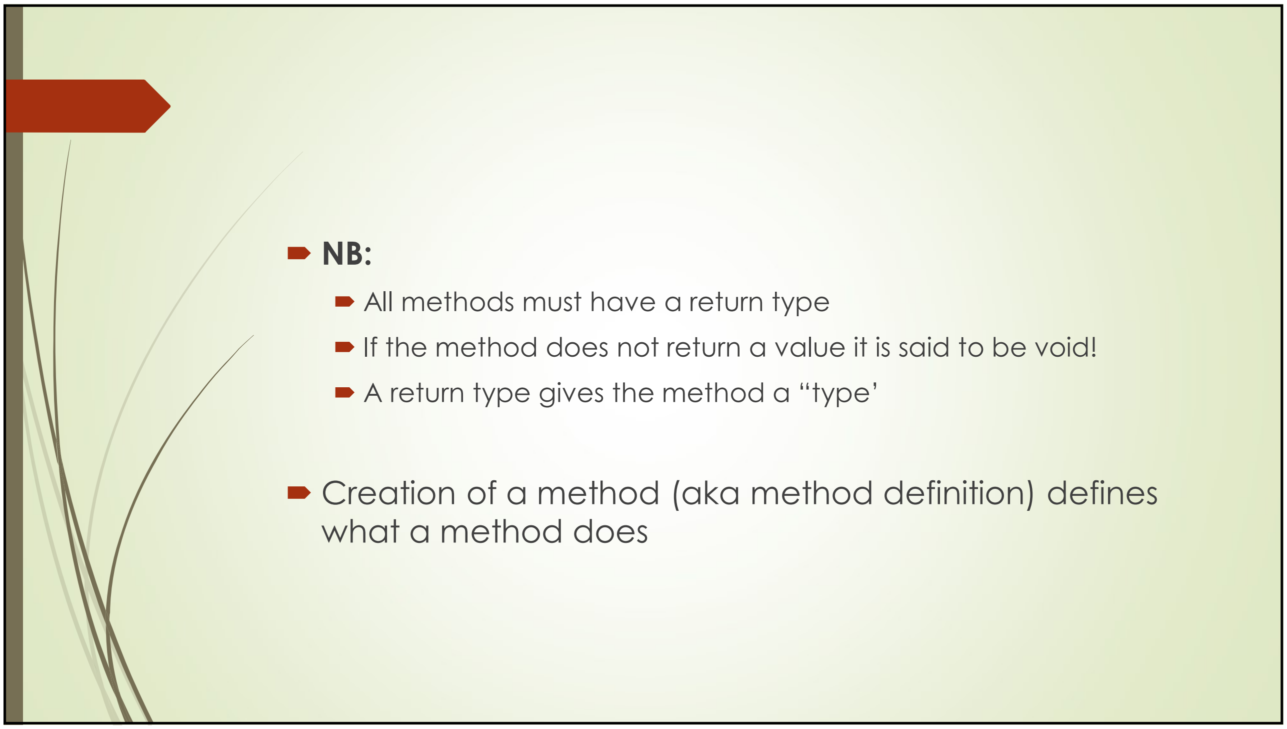


Define a method

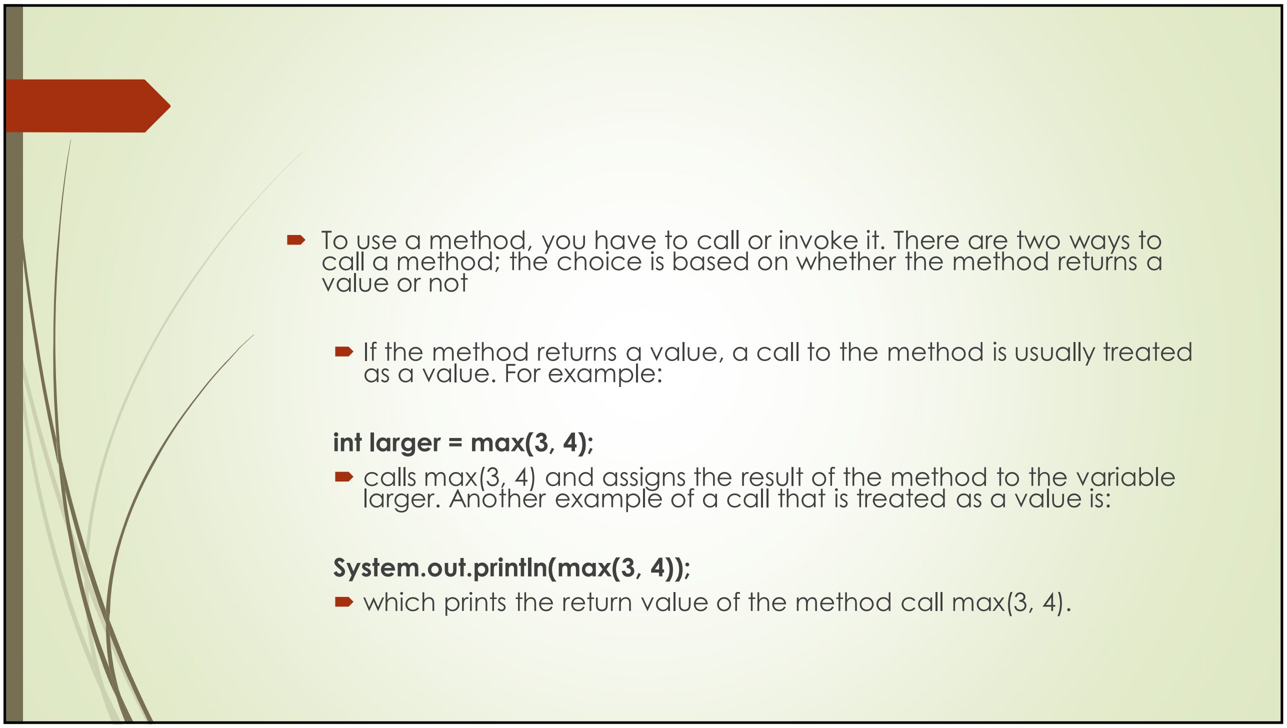
```
public static int max(int num1, int num2) {
   int result;

   if (num1 > num2)
     result = num1;
   else
     result = num2;

   return result;
}
```

Explanation:

Name: Max
Two parameters – of type integer
Checks which of the two is larger and returns it



- **NB:**
  - All methods must have a return type
  - If the method does not return a value it is said to be void!
  - A return type gives the method a “type’
- Creation of a method (aka method definition) defines what a method does



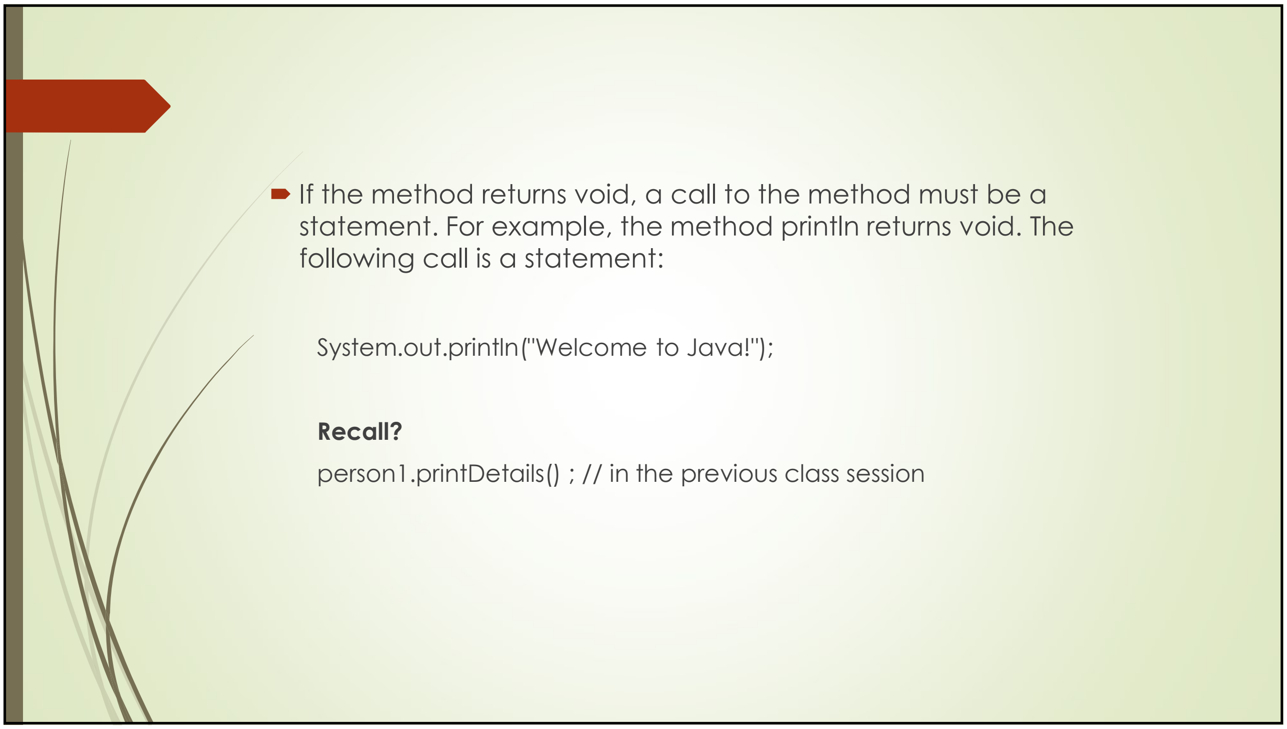
- To use a method, you have to call or invoke it. There are two ways to call a method; the choice is based on whether the method returns a value or not
  - If the method returns a value, a call to the method is usually treated as a value. For example:

int larger = max(3, 4);

- calls max(3, 4) and assigns the result of the method to the variable larger. Another example of a call that is treated as a value is:

System.out.println(max(3, 4));

- which prints the return value of the method call max(3, 4).

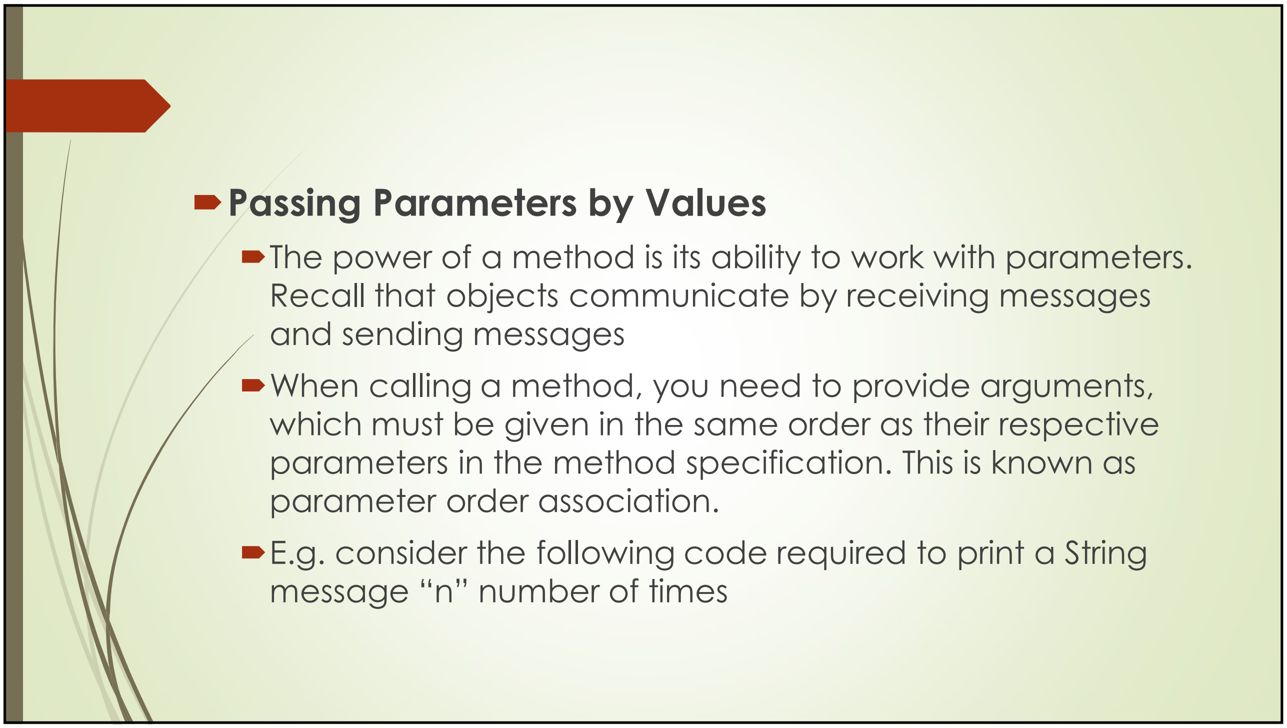


- If the method returns void, a call to the method must be a statement. For example, the method println returns void. The following call is a statement:

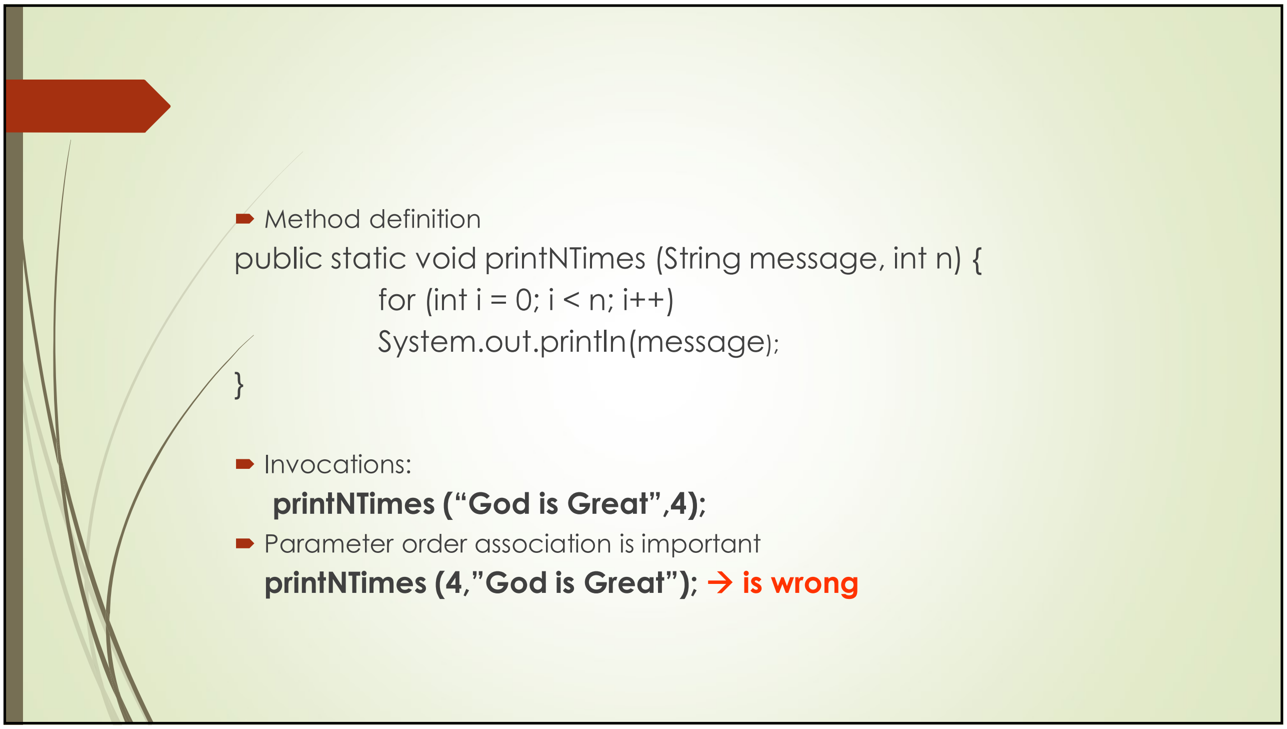
```
System.out.println("Welcome to Java!");
```

Recall?

```
person1.printDetails() ; // in the previous class session
```



- **Passing Parameters by Values**
  - The power of a method is its ability to work with parameters. Recall that objects communicate by receiving messages and sending messages
  - When calling a method, you need to provide arguments, which must be given in the same order as their respective parameters in the method specification. This is known as parameter order association.
  - E.g. consider the following code required to print a String message “n” number of times



- Method definition

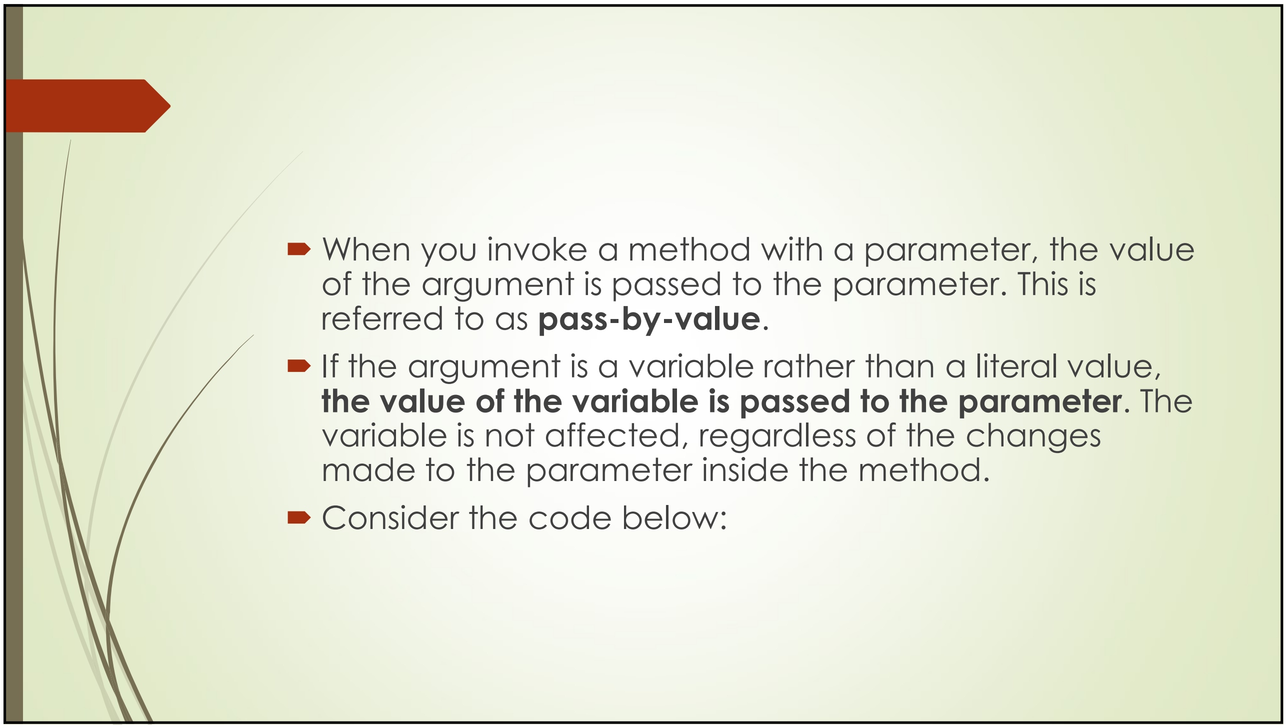
```
public static void printNTimes (String message, int n) {
            for (int i = 0; i < n; i++)
            System.out.println(message);
}
```

- Invocations:

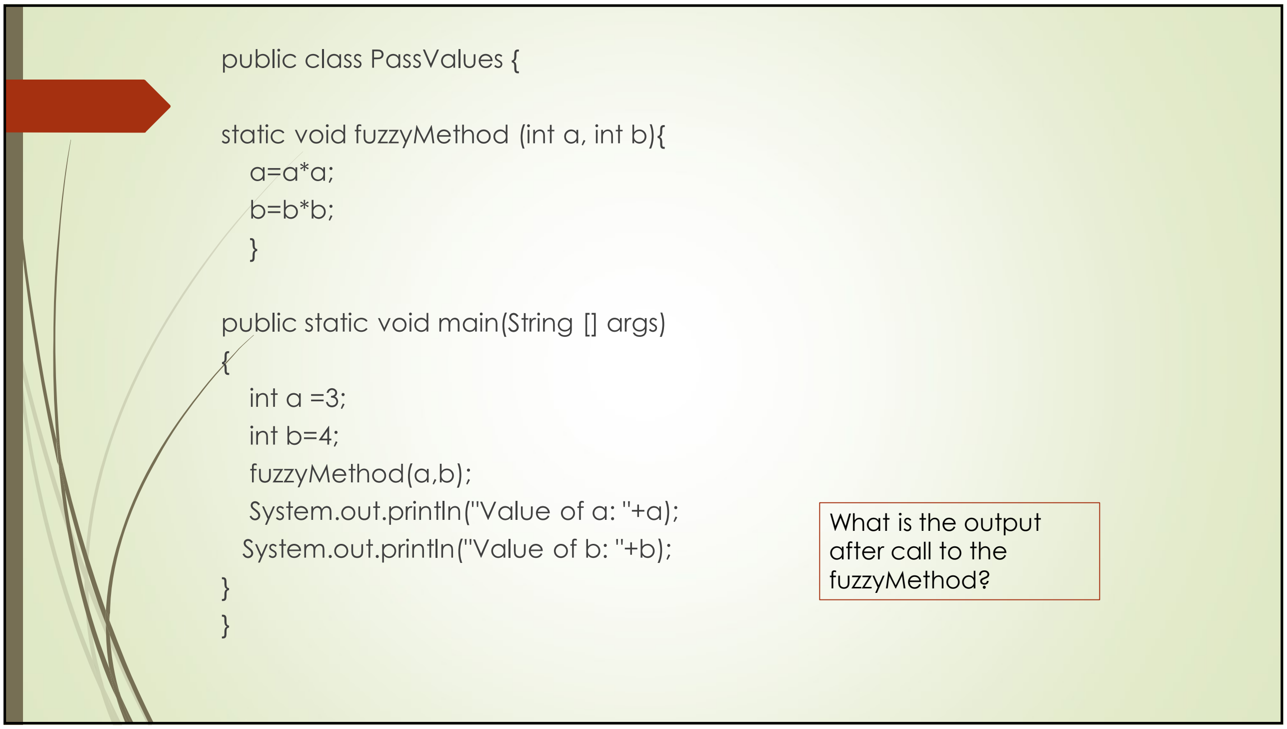
  printNTimes ("God is Great",4);

- Parameter order association is important

  printNTimes (4,"God is Great"); → **is wrong**

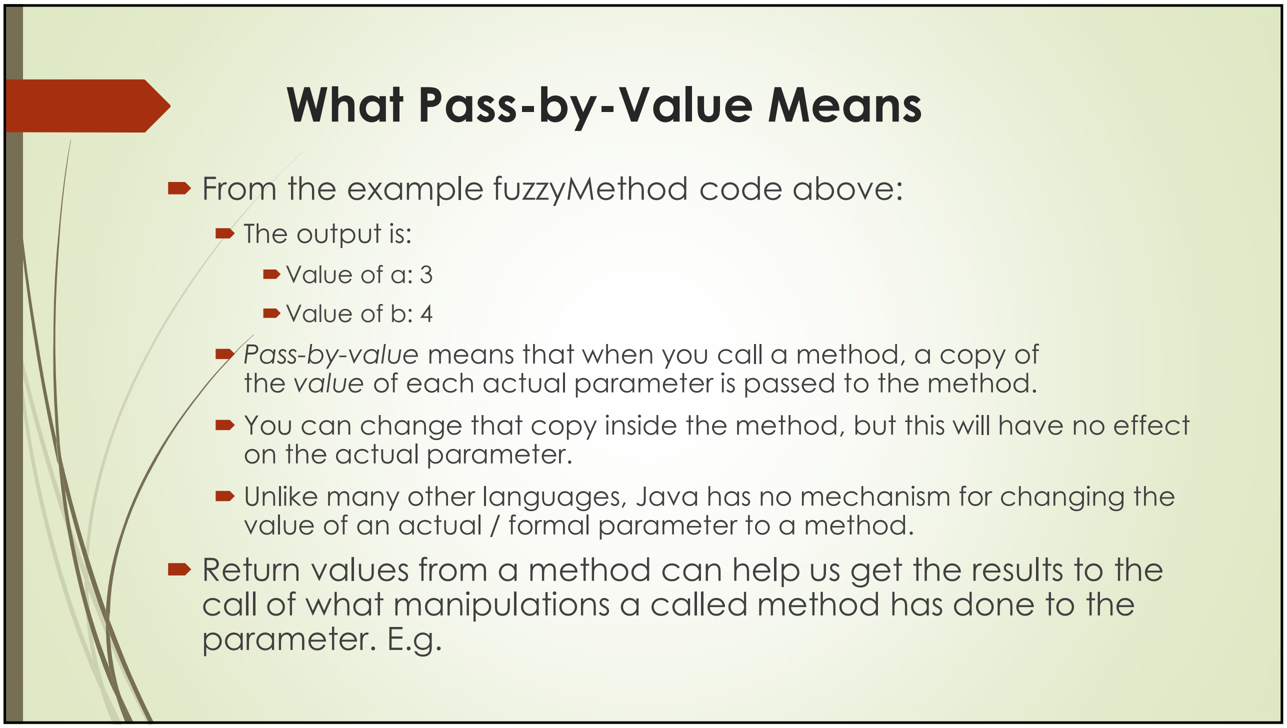


- When you invoke a method with a parameter, the value of the argument is passed to the parameter. This is referred to as **pass-by-value**.
- If the argument is a variable rather than a literal value, **the value of the variable is passed to the parameter**. The variable is not affected, regardless of the changes made to the parameter inside the method.
- Consider the code below:



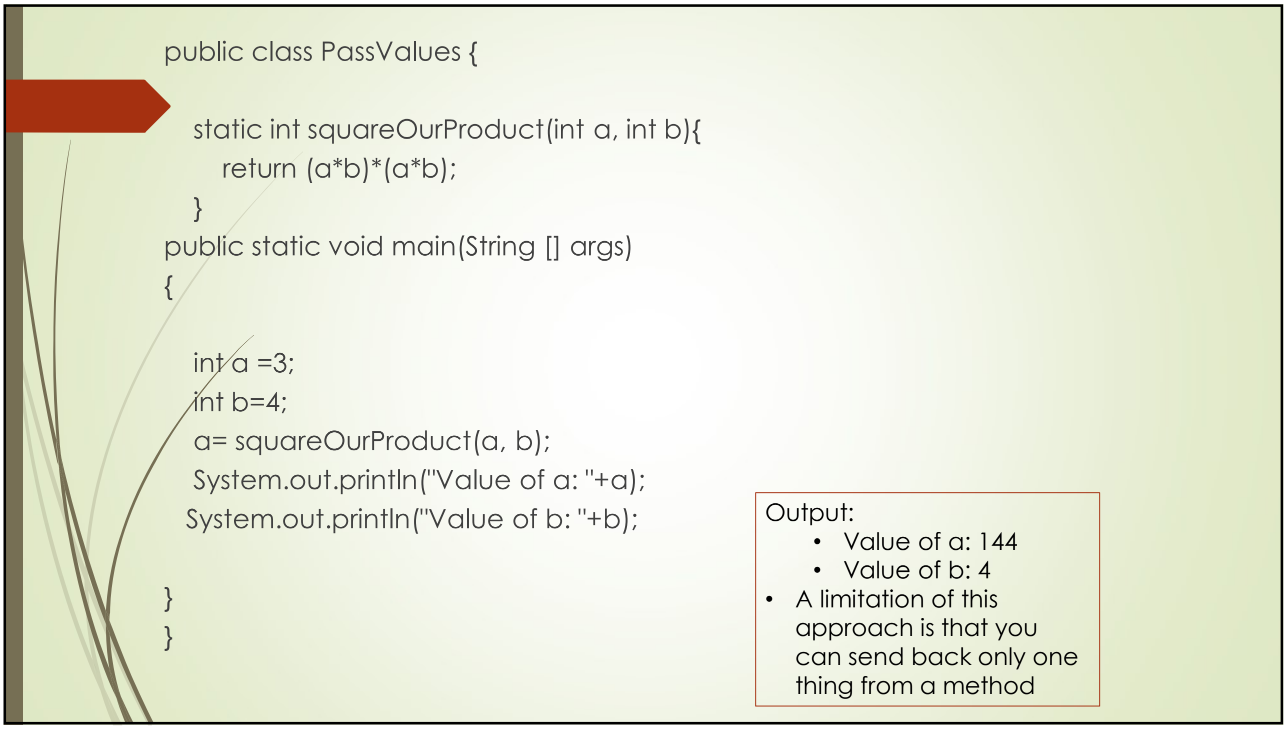
```
public class PassValues {

static void fuzzyMethod (int a, int b){
   a=a*a;
   b=b*b;
   }

public static void main(String [] args)
{
   int a =3;
   int b=4;
   fuzzyMethod(a,b);
   System.out.println("Value of a: "+a);
  System.out.println("Value of b: "+b);
}
}
```

What is the output after call to the fuzzyMethod?



What Pass-by-Value Means

- From the example fuzzyMethod code above:
  - The output is:
    - Value of a: 3
    - Value of b: 4
  - *Pass-by-value* means that when you call a method, a copy of the *value* of each actual parameter is passed to the method.
  - You can change that copy inside the method, but this will have no effect on the actual parameter.
  - Unlike many other languages, Java has no mechanism for changing the value of an actual / formal parameter to a method.
- Return values from a method can help us get the results to the call of what manipulations a called method has done to the parameter. E.g.

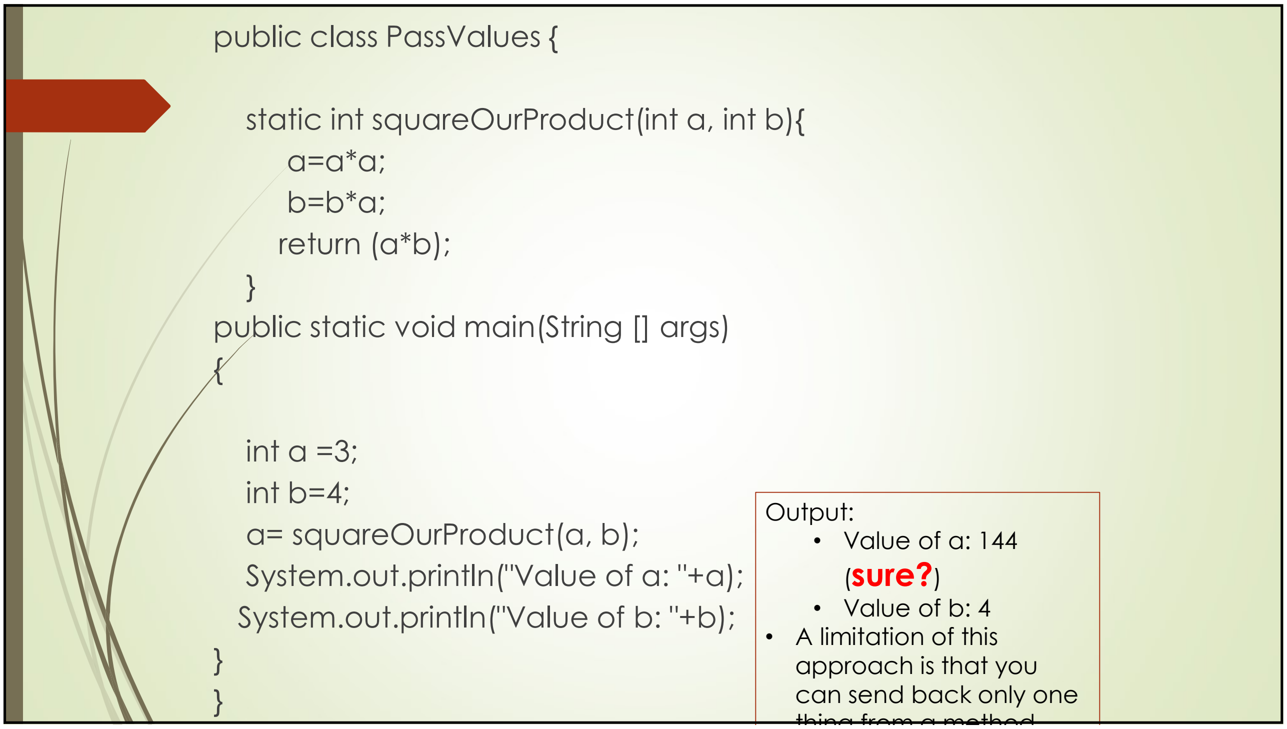


```
public class PassValues {

    static int squareOurProduct(int a, int b){
        return (a*b)*(a*b);
    }
public static void main(String [] args)
{

    int a =3;
    int b=4;
    a= squareOurProduct(a, b);
    System.out.println("Value of a: "+a);
   System.out.println("Value of b: "+b);

}
}
```

Output:

- Value of a: 144
- Value of b: 4

- A limitation of this approach is that you can send back only one thing from a method



```
public class PassValues {

    static int squareOurProduct(int a, int b){
        a=a*a;
        b=b*a;
       return (a*b);
    }
public static void main(String [] args)
{

    int a =3;
    int b=4;
    a= squareOurProduct(a, b);
    System.out.println("Value of a: "+a);
   System.out.println("Value of b: "+b);
}
}
```

Output:

- Value of a: 144 (**sure?**)
- Value of b: 4

- A limitation of this approach is that you can send back only one thing from a method

```
public class PassValues {

static int squareOurProduct(MyNumbers myNums){
      myNums.a *= myNums.a;
      myNums.b *= myNums.b;
      return myNums.a*myNums.b;
   }

public static void main(String [] args)
{
   MyNumbers twoNums = new MyNumbers(3, 4);
       int a= squareOurProduct(twoNums);
       int b=twoNums.b;
       System.out.println("Value of a: "+a);
      System.out.println("Value of b: "+b);
}
}
```

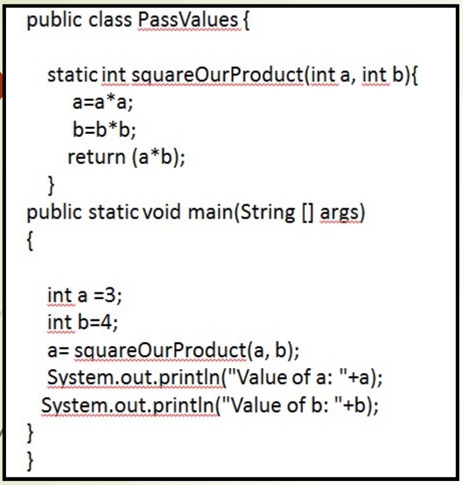
```
public class MyNumbers {
   int a;
   int b;
   public MyNumbers(int num1, int num2)
    {
      a=num1;
      b=num2;
    }
}
```

Output:

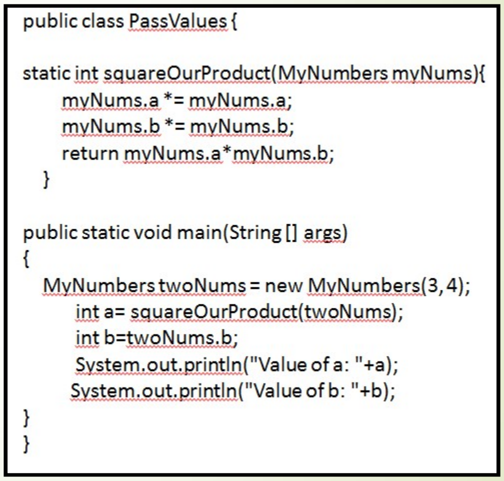
  - Value of a: 144
  - Value of b: 16

- A limitation of this approach is that you can send back only one thing from a method

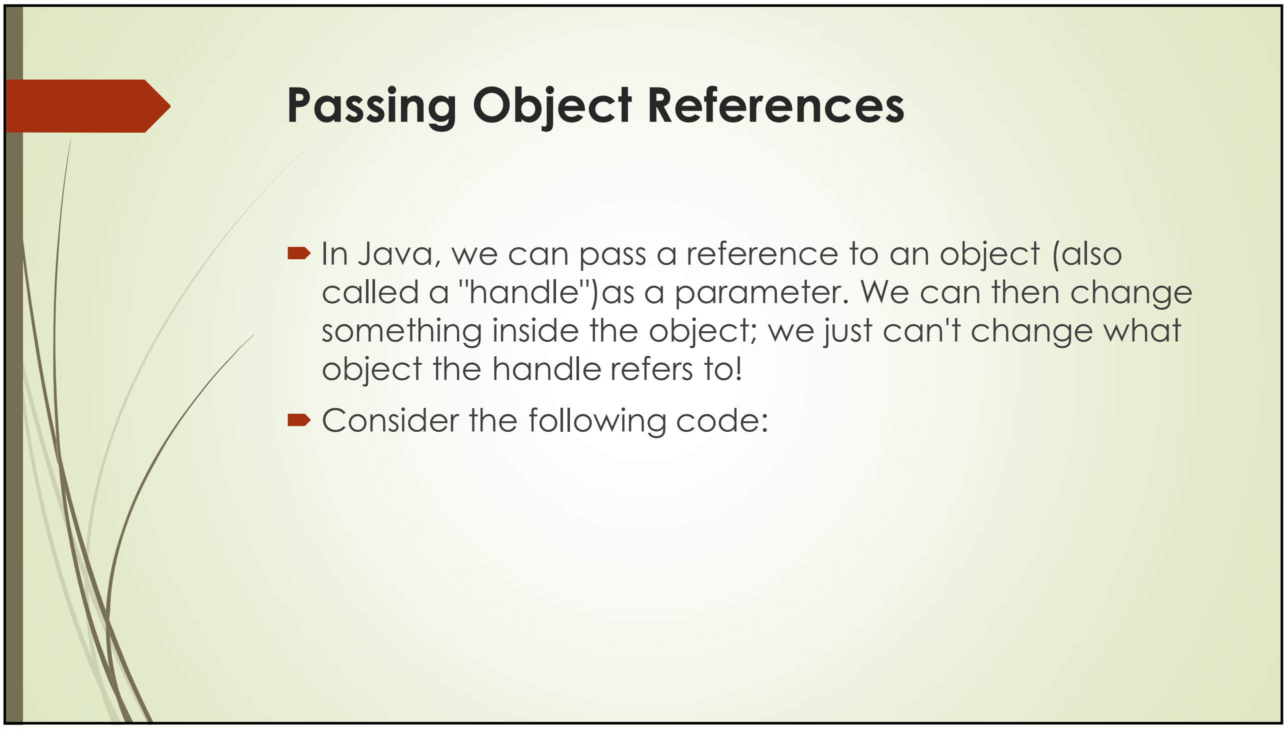
The two previous codes together for Comparison



```
public class PassValues {

  static int squareOurProduct(int a, int b){
     a=a*a;
     b=b*b;
    return (a*b);
  }
public static void main(String [] args)
{

  int a =3;
  int b=4;
  a= squareOurProduct(a, b);
  System.out.println("Value of a: "+a);
 System.out.println("Value of b: "+b);
}
}
```

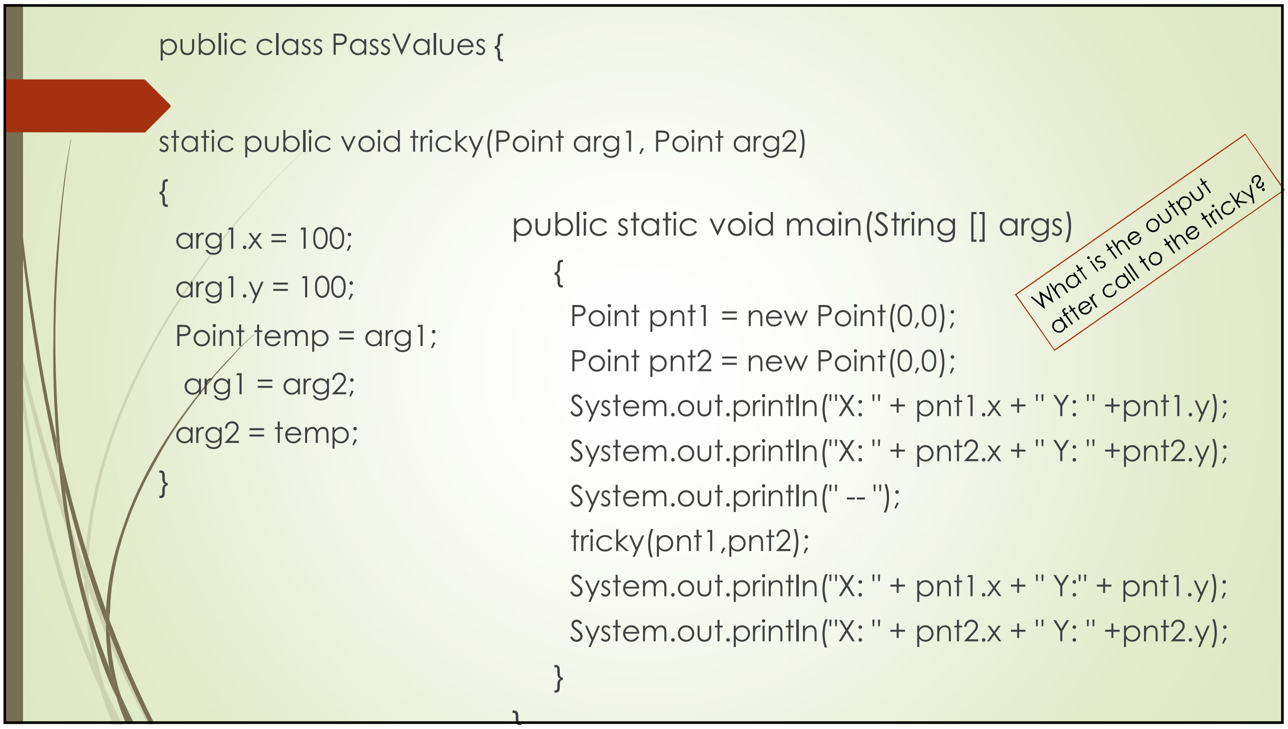


```
public class PassValues {

static int squareOurProduct(MyNumbers myNums){
     myNums.a *= myNums.a;
     myNums.b *= myNums.b;
     return myNums.a*myNums.b;
  }

public static void main(String [] args)
{
   MyNumbers twoNums = new MyNumbers(3, 4);
       int a= squareOurProduct(twoNums);
       int b=twoNums.b;
       System.out.println("Value of a: "+a);
      System.out.println("Value of b: "+b);
}
}
```



Passing Object References

- In Java, we can pass a reference to an object (also called a "handle")as a parameter. We can then change something inside the object; we just can't change what object the handle refers to!
- Consider the following code:



```
public class PassValues {

static public void tricky(Point arg1, Point arg2)
{
  arg1.x = 100;
  arg1.y = 100;
  Point temp = arg1;
   arg1 = arg2;
  arg2 = temp;
}

public static void main(String [] args)
   {
     Point pnt1 = new Point(0,0);
     Point pnt2 = new Point(0,0);
     System.out.println("X: " + pnt1.x + " Y: " +pnt1.y);
     System.out.println("X: " + pnt2.x + " Y: " +pnt2.y);
     System.out.println(" -- ");
     tricky(pnt1,pnt2);
     System.out.println("X: " + pnt1.x + " Y:" + pnt1.y);
     System.out.println("X: " + pnt2.x + " Y: " +pnt2.y);
   }
}
```

What is the output after call to the tricky?

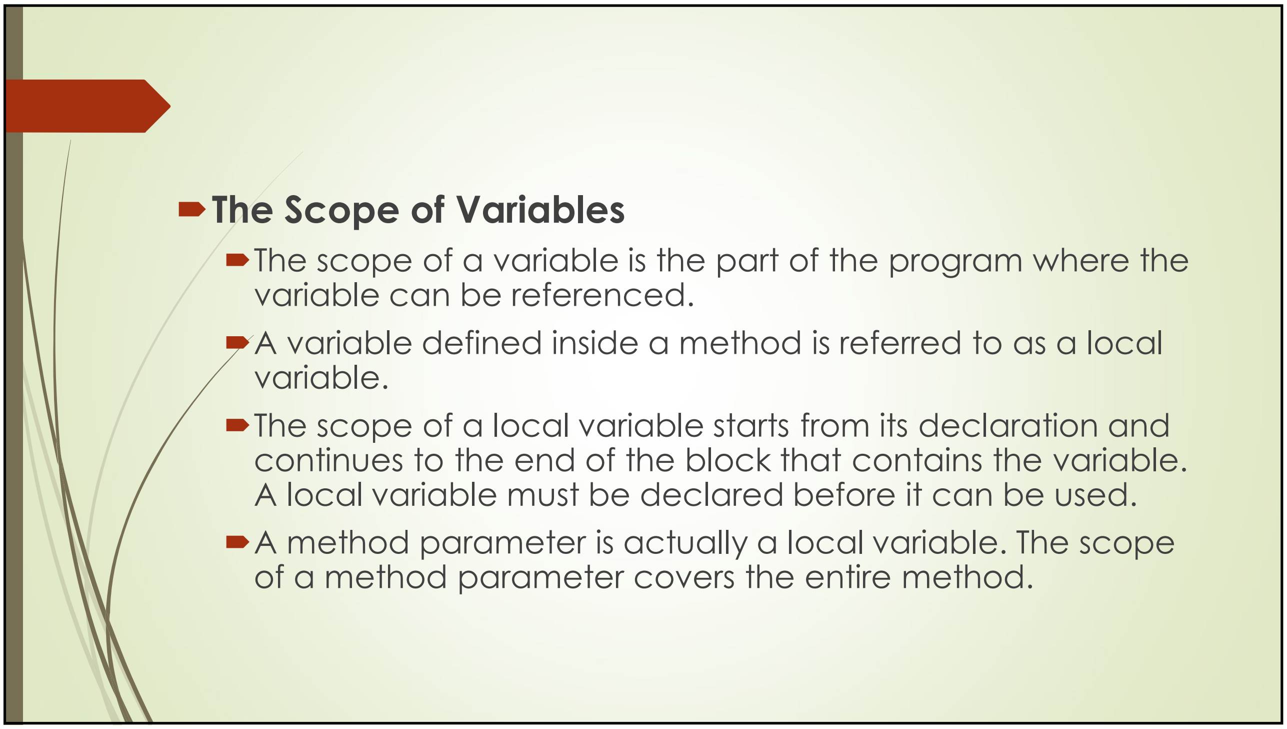
What passing object reference means

- The output:

```
X: 0 Y: 0
X: 0 Y: 0
 --
X: 100 Y:100
X: 0 Y: 0
```

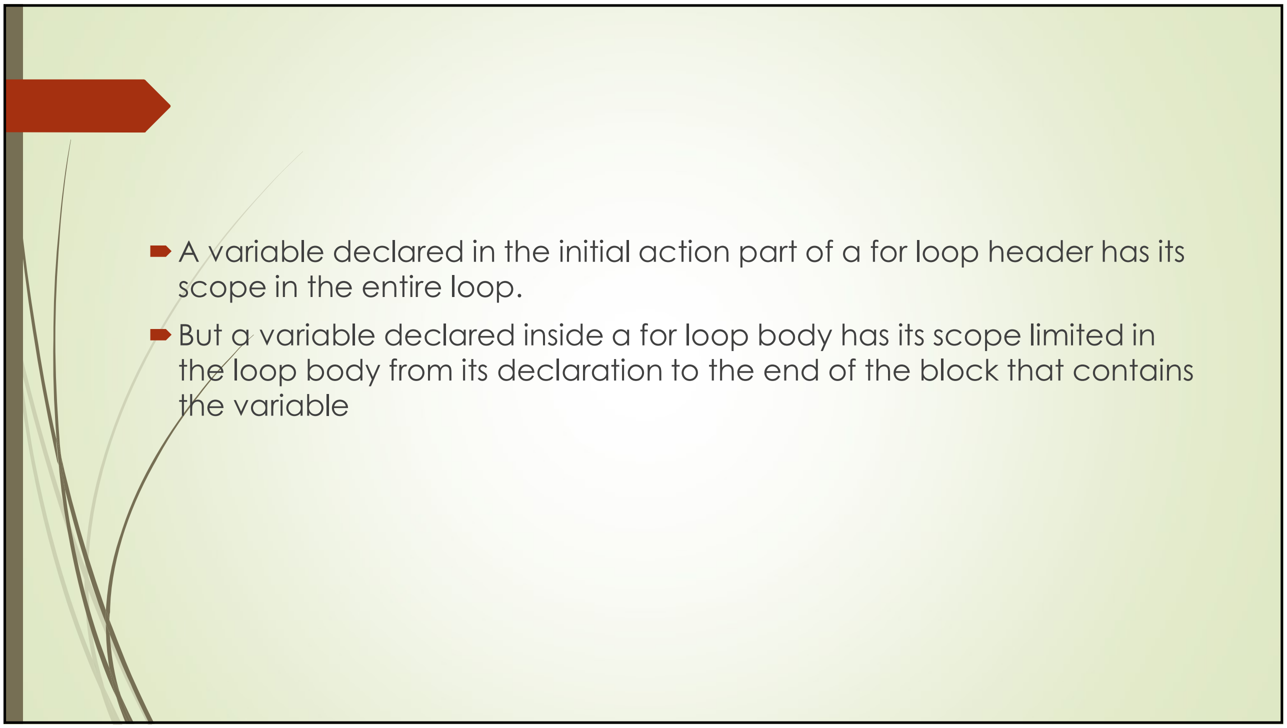
- The method successfully alters the value of pnt1, even though it is passed by value; however, a swap of pnt1 and pnt2 fails!
- In the main() method, pnt1 and pnt2 are nothing more than object references. When you pass pnt1 and pnt2 to the tricky() method, Java passes the references by value just like any other parameter. This means the references passed to the method are actually *copies* of the original references.
- Java copies and passes the *reference* by value, not the object. Thus, method manipulation will alter the objects, since the references point to the original objects. But **since the references are copies, swaps will fai**l

- Reflections and own notes / research !

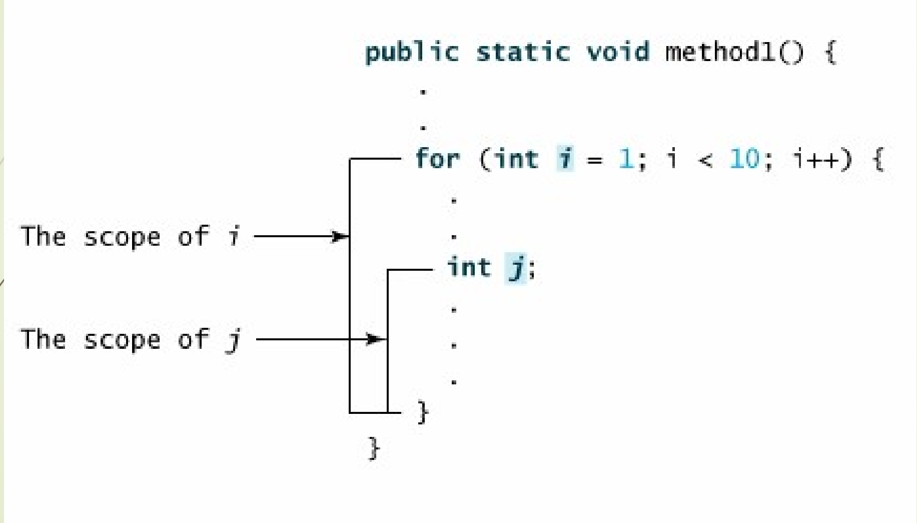


The Scope of Variables

- The scope of a variable is the part of the program where the variable can be referenced.
- A variable defined inside a method is referred to as a local variable.
- The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable. A local variable must be declared before it can be used.
- A method parameter is actually a local variable. The scope of a method parameter covers the entire method.

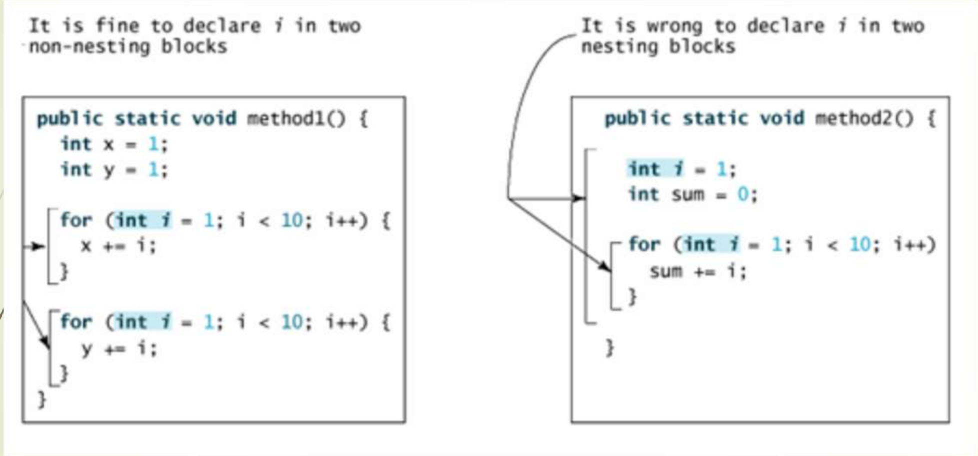


- A variable declared in the initial action part of a for loop header has its scope in the entire loop.
- But a variable declared inside a for loop body has its scope limited in the loop body from its declaration to the end of the block that contains the variable



```
public static void method1() {
    .
    .
    for (int i = 1; i < 10; i++) {
      .
      .
      int j;
      .
      .
      .
    }
}
```

- You can declare a local variable with the same name multiple times in different non nesting blocks in a method, but you cannot declare a local variable twice in nested blocks, as shown in the figure below.



It is fine to declare *i* in two non-nesting blocks

```
public static void method1() {
  int x = 1;
  int y = 1;

  for (int i = 1; i < 10; i++) {
    x += i;
  }

  for (int i = 1; i < 10; i++) {
    y += i;
  }
}
```

It is wrong to declare *i* in two nesting blocks

```
public static void method2() {

  int i = 1;
  int sum = 0;

  for (int i = 1; i < 10; i++)
    sum += i;
  }

}
```

- Thanks!