

Fundamentals of Java

Session: 6

Classes and Objects





- ◆ Explain the process of creation of classes in Java
- ◆ Explain the instantiation of objects in Java
- ◆ Explain the purpose of instance variables and instance methods
- ◆ Explain constructors in Java
- ◆ Explain the memory management in Java
- ◆ Explain object initializers



◆ Class in Java:

- ◆ Is the prime unit of execution for object-oriented programming in Java.
- ◆ Is a logical structure that defines the shape and nature of an object.
- ◆ Is defined as a new data type that is used to create objects of its type.
- ◆ Defines attributes referred to as fields that represents the state of an object.

Conventions to be followed while naming a class

- Class declaration should begin with the keyword class followed by the name of the class.
- Class name should be a noun.
- Class name can be in mixed case, with the first letter of each internal word capitalized.
- Class name should be simple, descriptive, and meaningful.
- Class name cannot be Java keywords.
- Class name cannot begin with a digit. However, they can begin with a dollar (\$) symbol or an underscore character.

Declaring a Class 1-2



- ◆ The syntax to declare a class in Java is as follows:

Syntax

```
class <class_name> {  
    // class body  
}
```

- ◆ The body of the class is enclosed between the curly braces { }.
- ◆ In the class body, you can declare members, such as fields, methods, and constructors.
- ◆ Following figure shows the declaration of a sample class:

```
class Student {  
    String studName;  
    int studAge;  
  
    void initialize()  
    {  
        studName = "James Anderson";  
        studAge = 26;  
    }  
  
    void display()  
    {  
        System.out.println("Student Name: " + studName);  
        System.out.println("Student Age:" + studAge);  
    }  
  
    public static void main(String[] args)  
    {  
        Student objStudent = new Student();  
        objStudent.initialize();  
        objStudent.display();  
    }  
}
```

Fields or Instance Variables

Functions or Instance Methods



- ◆ Following code snippet shows the code for declaring a class **Customer**:

```
class Customer {  
  
    // body of class  
}
```

- ◆ In the code:
 - ◆ A class is declared that acts as a new data type with the name **Customer**.
 - ◆ It is just a template for creating multiple objects with similar features.
 - ◆ It does not occupy any memory.
- ◆ **Creating Objects:**
 - ◆ Objects are the actual instances of the class.

Declaring and Creating an Object 1-2



- ◆ An object is created using the `new` operator.
- ◆ On encountering the `new` operator:
 - ◆ JVM allocates memory for the object.
 - ◆ Returns a reference or memory address of the allocated object.
 - ◆ The reference or memory address is then stored in a variable called as reference variable.
- ◆ The syntax for creating an object is as follows:

Syntax

```
<class_name> <object_name> = new <class_name> ();
```

where,

`new`: Is an operator that allocates the memory for an object at runtime.

`object_name`: Is the variable that stores the reference of the object.

Declaring and Creating an Object 2-2



- ◆ Following code snippet demonstrates the creation of an object in a Java program:

```
Customer objCustomer = new Customer();
```

- ◆ The expression on the right side, **new Customer()** allocates the memory at runtime.
- ◆ After the memory is allocated for the object, it returns the reference or address of the allocated object, which is stored in the variable, **objCustomer**.



- ◆ Alternatively, an object can be created using two steps that are as follows:
 - ◆ Declaration of an object reference.
 - ◆ Dynamic memory allocation of an object.
- ◆ **Declaration of an object reference:**
 - ◆ The syntax for declaring the object reference is as follows:

Syntax

```
<class_name> <object_name>;
```

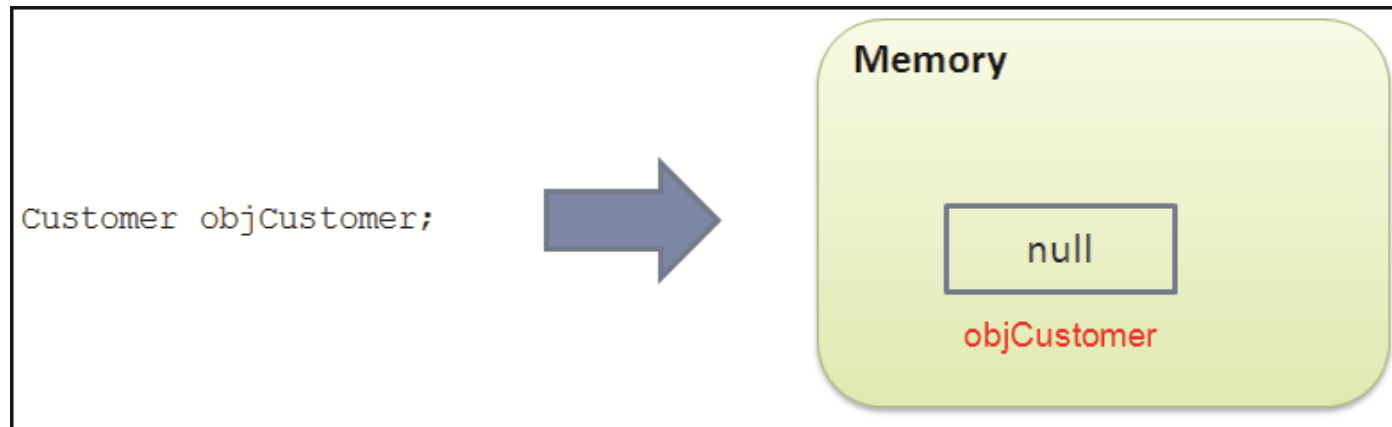
where,

`object_name`: Is just a variable that will not point to any memory location.

Creation of an Object: Two Stage Process 2-3



- ◆ Following figure shows the effect of the statement, **Customer objCustomer;** which declares a reference variable:

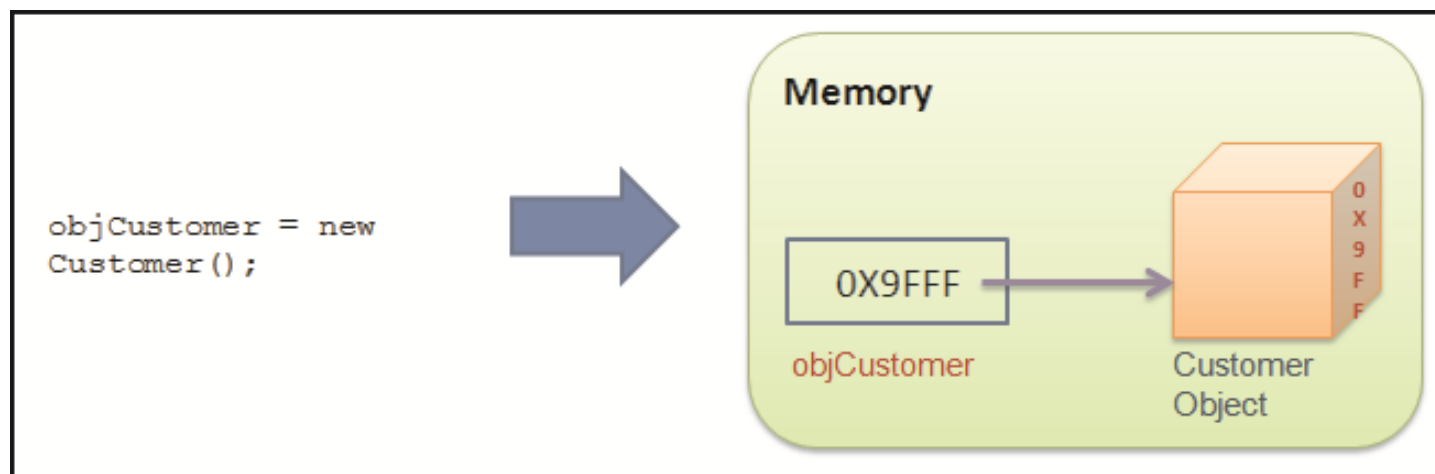


- ◆ By default, the value `null` is stored in the object's reference variable which means reference variable does not point to an actual object.
- ◆ If **objCustomer** is used at this point of time, without being instantiated, then the program will result in a compile time error.



◆ Dynamic memory allocation of an object:

- ◆ The object should be initialized using the `new` operator which dynamically allocates memory for an object.
- ◆ For example, the statement, `objCustomer = new Customer();` allocates memory for the object and memory address of the allocated object is stored in the variable `objCustomer`.
- ◆ Following figure shows the creation of object in the memory and storing of its reference in the variable, `objCustomer`:





- ◆ The members of a class are fields and methods.

Fields

- Define the state of an object created from the class.
- Referred to as instance variables.

Methods

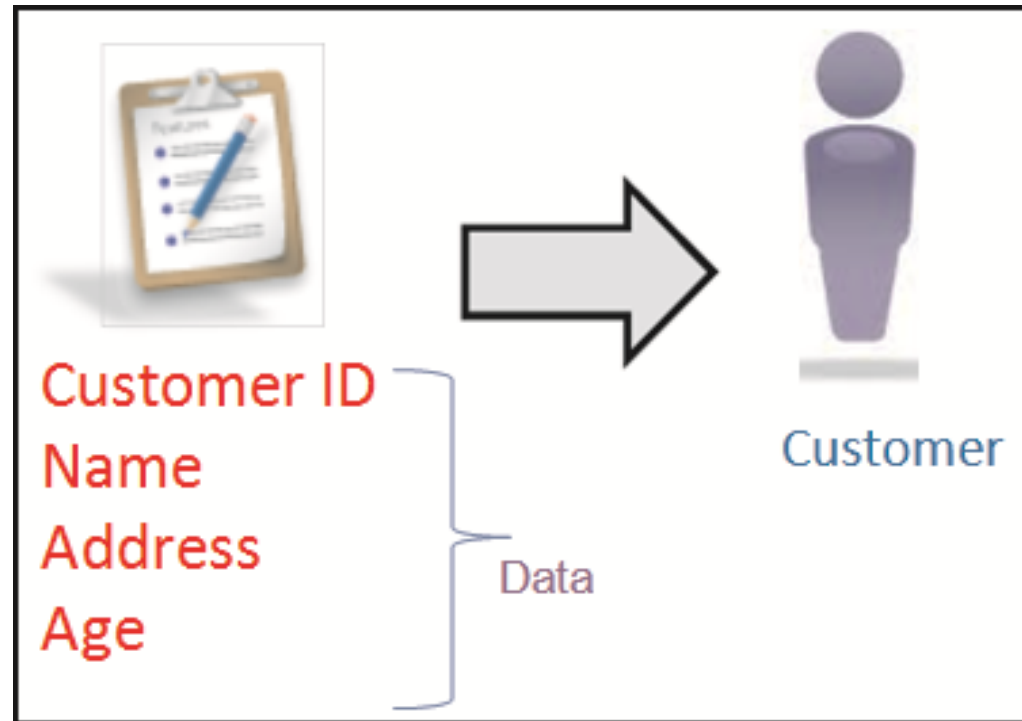
- Implement the behavior of the objects.
- Referred to as instance methods.



- ◆ They are used to store data in them.
- ◆ They are called instance variables because each instance of the class, that is, object of that class will have its own copy of the **instance variables**.
- ◆ They are declared similar to local variables.
- ◆ They are declared inside a class, but outside any method definitions.
- ◆ For example: Consider a scenario where the **Customer** class represents the details of customers holding accounts in a bank.
 - ◆ A typical question that can be asked is ‘What are the different data that are required to identify a customer in a banking domain and represent it as a single object?’.



- ◆ Following figure shows a **Customer** object with its data requirement:

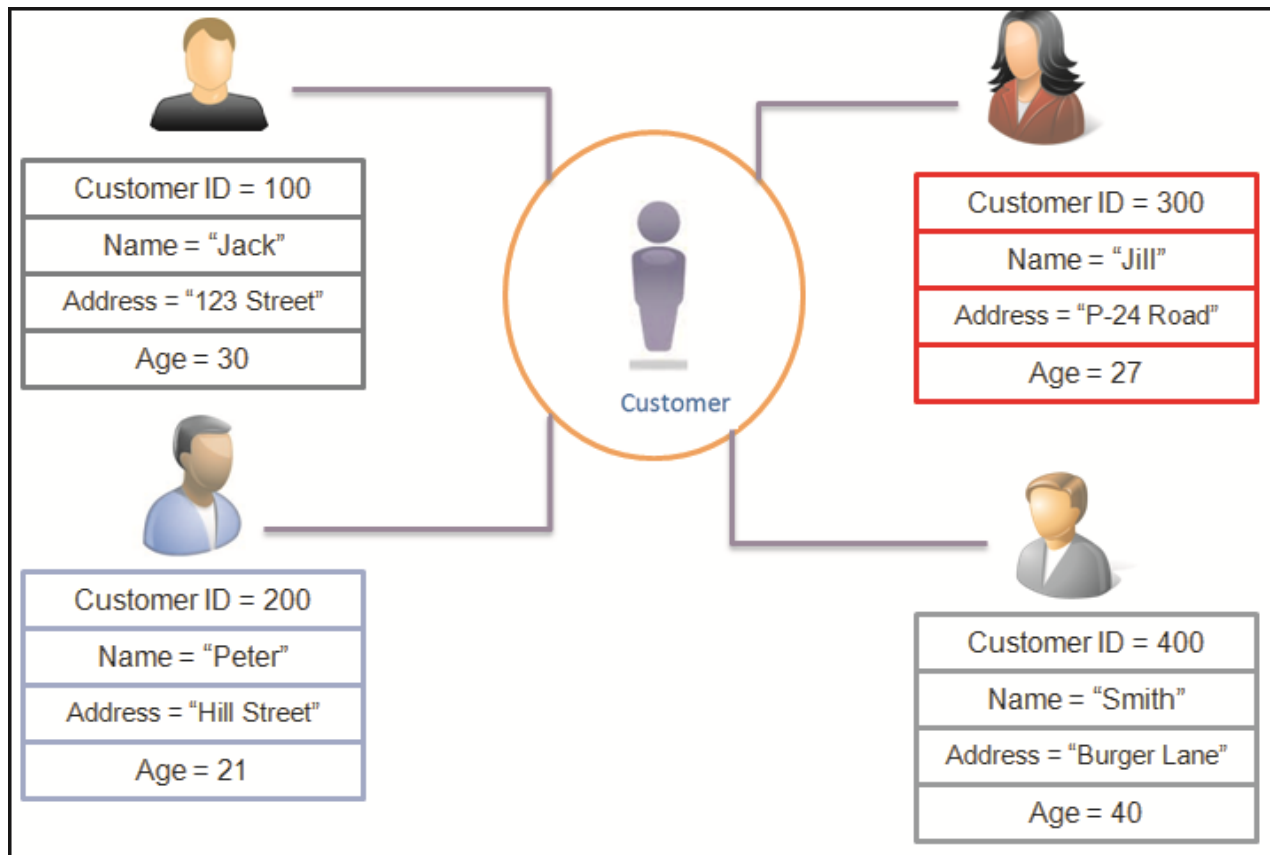


- ◆ The identified data requirements for a bank customer includes: Customer ID, Name, Address, and Age.
- ◆ To map these data requirements in a **Customer** class, **instance variables** are declared.

Instance Variables 3-7



- ◆ Each instance created from the **Customer** class will have its own copy of the instance variables.
- ◆ Following figure shows various instances of the class with their own copy of instance variables:





- ◆ The syntax to declare an instance variable within a class is as follows:

Syntax

```
[access_modifier] data_type instanceVariableName;
```

where,

`access_modifier`: Is an optional keyword specifying the access level of an instance variable. It could be `private`, `protected`, and `public`.

`data_type`: Specifies the data type of the variable.

`instanceVariableName`: Specifies the name of the variable.

- ◆ Instance variables are accessed by objects using the dot operator (.



- ◆ Following code snippet demonstrates the declaration of instance variables within a class in the Java program:

```
1: public class Customer {  
2: // Declare instance variables  
3: int customerID;  
4: String customerName;  
5: String customerAddress;  
6: int customerAge;  
  
7: /* As main() method is a member of class, so it can access other  
8: * members of the class */  
9: public static void main(String[] args) {  
10: // Declares and instantiates an object of type Customer  
11: Customer objCustomer1 = new Customer();  
}
```

- ◆ Lines 3 to 6 declares instance variables.
- ◆ Line 11 creates an object of type **Customer** and stores its reference in the variable, **objCustomer1**.

Instance Variables 6-7



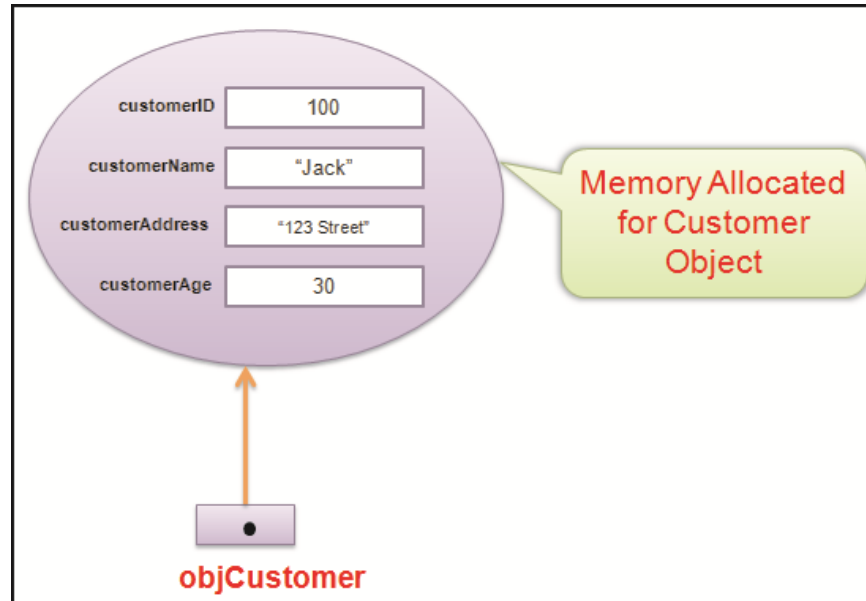
```
12: // Accesses the instance variables to store values
13: objCustomer1.customerID = 100;
14: objCustomer1.customerName = "John";
15: objCustomer1.customerAddress = "123 Street";
16: objCustomer1.customerAge = 30;

17: // Displays the objCustomer1 object details
18: System.out.println("Customer Identification Number: " +
objCustomer1.customerID);
19: System.out.println("Customer Name: " + objCustomer1.customerName);
20: System.out.println("Customer Address: " + objCustomer1.
customerAddress);
21: System.out.println("Customer Age: " + objCustomer1.customerAge);
    }
}
```

- ◆ Lines 13 to 16 accesses the instance variables and assigns them the values.
- ◆ Lines 18 to 21 display the values assigned to the instance variables for the object, **objCustomer1**.



- ◆ Following figure shows the allocation of **Customer** object in the memory:



- ◆ Following figure shows the output of the code:

```
Output - Session 6 (run)
run:
Customer Identification Number: 100
Customer Name: John
Customer Address: 123 Street
Customer Age: 30
BUILD SUCCESSFUL (total time: 1 second)
```



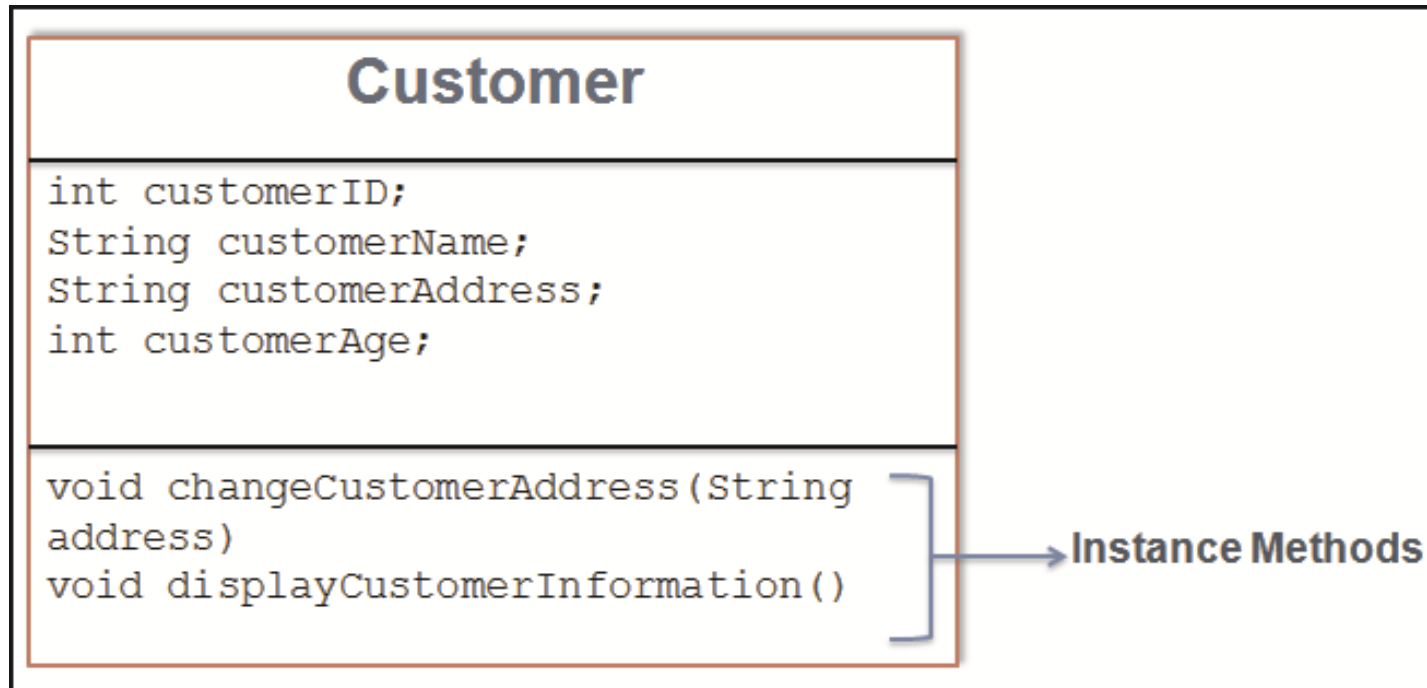
- ◆ They are functions declared in a class.
- ◆ They implement the behavior of an object.
- ◆ They are used to perform operations on the instance variables.
- ◆ They can be accessed by instantiating an object of the class in which it is defined and then, invoking the method.
- ◆ For example: the class **Car** can have a method `Brake ()` that represents the 'Apply Brake' action.
 - ◆ To perform the action, the method `Brake ()` will have to be invoked by an object of class **Car**.

Conventions to be followed while naming a method are as follows:

- Cannot be a Java keyword.
- Cannot contain spaces.
- Cannot begin with a digit.
- Can begin with a letter, underscore, or a '\$' symbol.
- Should be a verb in lowercase.
- Should be descriptive and meaningful.
- Should be a multi-word name that begins with a verb in lowercase, followed by adjectives, nouns, and so forth.



- ◆ Following figure shows the instance methods declared in the class, **Customer**:





- ◆ The syntax to declare an instance method in a class is as follows:

Syntax

```
[access_modifier] <return type> <method_name> ([list  
of parameters]) {  
  
// Body of the method  
  
}
```

where,

`access_modifier`: Is an optional keyword specifying the access level of an instance method. It could be `private`, `protected`, and `public`.

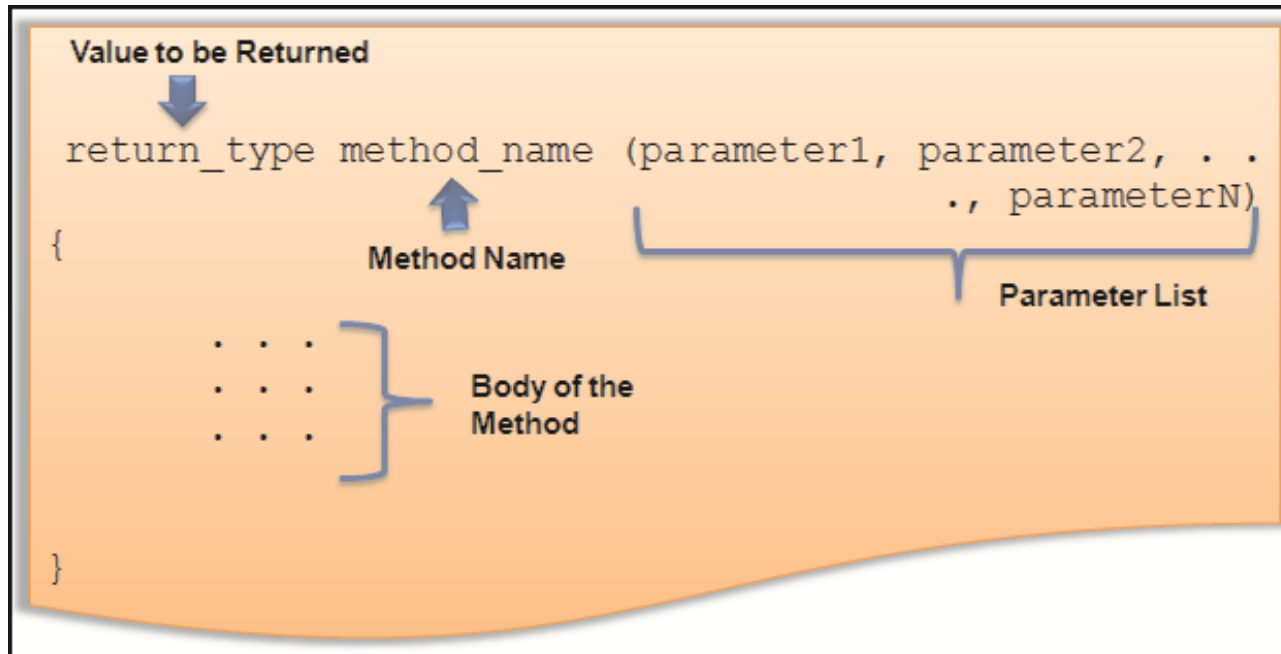
`returntype`: Specifies the data type of the value that is returned by the method.

`method_name`: Is the method name.

`list of parameters`: Are the values passed to the method.



- ◆ Following figure shows the declaration of an instance method within the class:



- ◆ Each instance of the class has its own instance variables, but the instance methods are shared by all the instances of the class during execution.



- ◆ Following code snippet demonstrates the code that declares instance methods within the class, **Customer**:

```
public class Customer {  
    // Declare instance variables  
    int customerID;  
    String customerName;  
    String customerAddress;  
    int customerAge;  
  
    /**  
     * Declares an instance method changeCustomerAddress is created to  
     * change the address of the customer object  
     */  
    void changeCustomerAddress(String address) {  
        customerAddress = address;  
    }  
}
```

- ◆ The instance methods **changeCustomerAddress ()** method will accept a string value through parameter address.
- ◆ It then assigns the value of address variable to the **customerAddress** field.



```
/**
 * Declares an instance method displayCustomerInformation is created
 * to display the details of the customer object
 */
void displayCustomerInformation() {
    System.out.println("Customer Identification Number: " + customerID);
    System.out.println("Customer Name: " + customerName);
    System.out.println("Customer Address: " + customerAddress);
    System.out.println("Customer Age: " + customerAge);
}
```

- ◆ The method **displayCustomerInformation()** displays the details of the customer object.



- ◆ To invoke a method, the object name is followed by the dot operator (.) and the method name.
- ◆ A method is always invoked from another method.
- ◆ The method which invokes a method is referred to as the **calling method**.
- ◆ The invoked method is referred to as the **called method**.
- ◆ After execution of all the statements within the code block of the invoked method, the control returns back to the **calling method**.



- ◆ Following code snippet demonstrates a class with `main()` method which creates the instance of the class **Customer** and invokes the methods defined in the class:

```
public class TestCustomer {  
  
    /**  
     * @param args the command line arguments  
     * The main() method creates the instance of class Customer  
     * and invoke its methods  
     */  
    public static void main(String[] args) {  
        // Creates an object of the class  
        Customer objCustomer = new Customer();  
  
        // Initialize the object  
        objCustomer.customerID = 100;  
        objCustomer.customerName = "Jack";  
        objCustomer.customerAddress = "123 Street";  
        objCustomer.customerAge = 30;  
    }  
}
```

- ◆ The code instantiates an object **objCustomer** of type **Customer** class and initializes its instance variables.

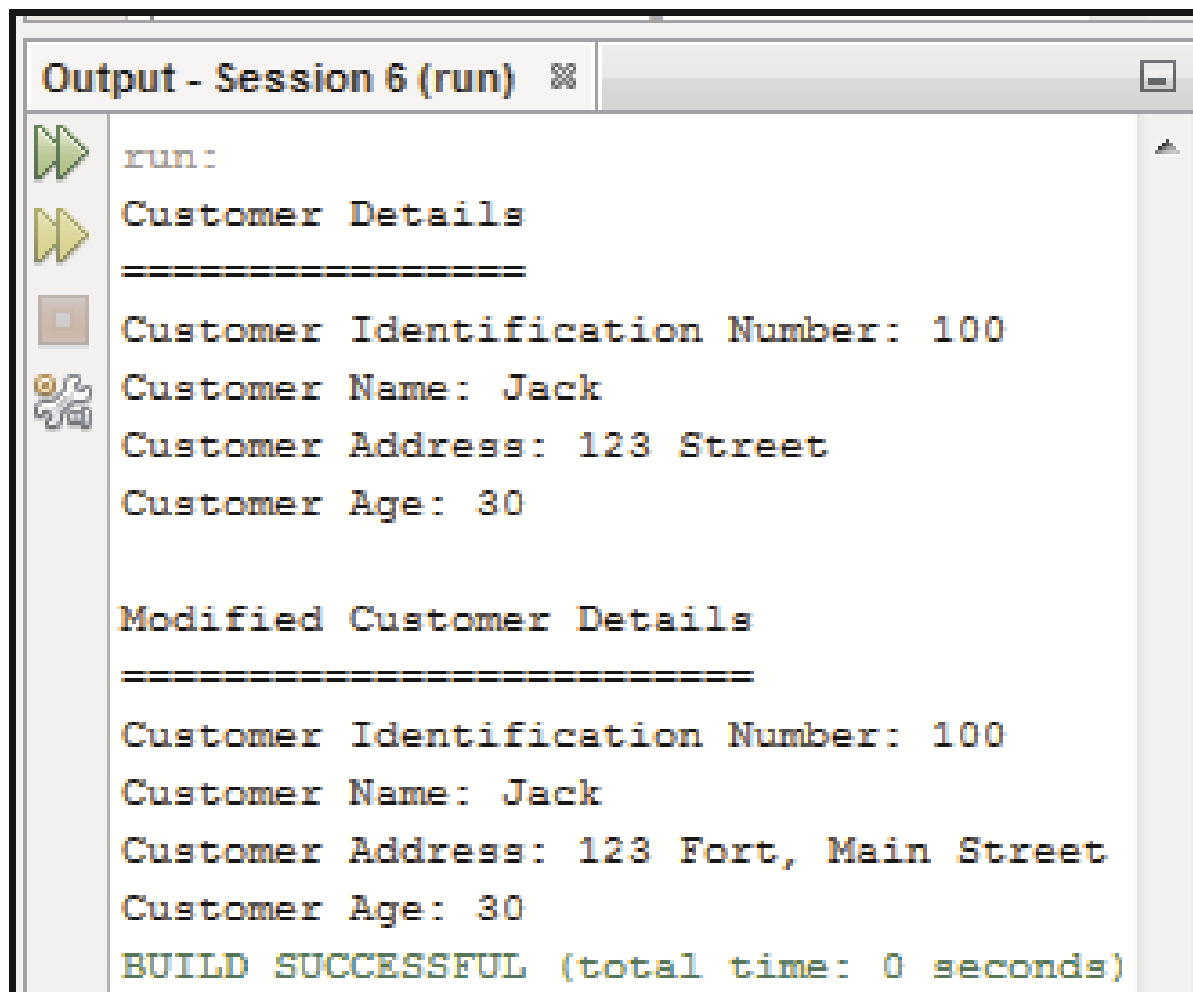


```
/*
 * Invokes the instance method to display the details
 * of objCustomer object
 */
objCustomer.displayCustomerInformation();
/*
 * Invokes the instance method to
 * change the address of the objCustomer object
 */
objCustomer.changeCustomerAddress("123 Fort, Main Street");
/*
 * Invokes the instance method after changing the address field
 * of objCustomer object
 */
objCustomer.displayCustomerInformation();
}
}
```

- ◆ The method **displayCustomerInformation()** is invoked using the object **objCustomer** and displays the values of the initialized instance variables on the console.
- ◆ Then, the method **changeCustomerAddress("123 Fort, Main Street")** is invoked to change the data of the **customerAddress** field.



- ◆ Following figure shows the output of the code:



```
run:
Customer Details
=====
Customer Identification Number: 100
Customer Name: Jack
Customer Address: 123 Street
Customer Age: 30

Modified Customer Details
=====
Customer Identification Number: 100
Customer Name: Jack
Customer Address: 123 Fort, Main Street
Customer Age: 30
BUILD SUCCESSFUL (total time: 0 seconds)
```



- ◆ It is a method having the same name as that of the class.
- ◆ It initializes the variables of a class or perform startup operations only once when the object of the class is instantiated.
- ◆ It is automatically executed whenever an instance of a class is created.
- ◆ It can accept parameters and do not have return types.
- ◆ It is of two types:
 - ◆ No-argument constructor
 - ◆ Parameterized constructor
- ◆ Following figure shows the constructor declaration:

```
class <ClassName>
{
    <ClassName> () ← Constructor
    {
        // Initialization code
    }
}
```



- ◆ The syntax for declaring constructor in a class is as follows:

Syntax

```
<classname>() {  
    // Initialization code  
}
```



◆ No-argument Constructor:

- ◆ Following code snippet demonstrates a class **Rectangle** with a constructor:

```
public class Rectangle {  
    int width;  
    int height;  
  
    /**  
     * Constructor for Rectangle class  
     */  
    Rectangle() {  
        width = 10;  
        height = 10;  
    }  
}
```

- ◆ The code declares a method named **Rectangle()** which is a constructor.
- ◆ This method is invoked by JVM to initialize the two instance variables, **width** and **height**, when the object of type **Rectangle** is constructed.
- ◆ The constructor does not have any parameters; hence, it is called as **no-argument constructor**.



- ◆ The constructor is invoked immediately during the object creation which means:
 - ◆ Once the `new` operator is encountered, memory is allocated for the object.
 - ◆ Constructor method is invoked by the JVM to initialize the object.
- ◆ Following figure shows the use of `new` operator to understand the constructor invocation:

```
<class_name> <object_name> = new <class_name>();
```

The parenthesis after the class name indicates the invocation of the constructor.



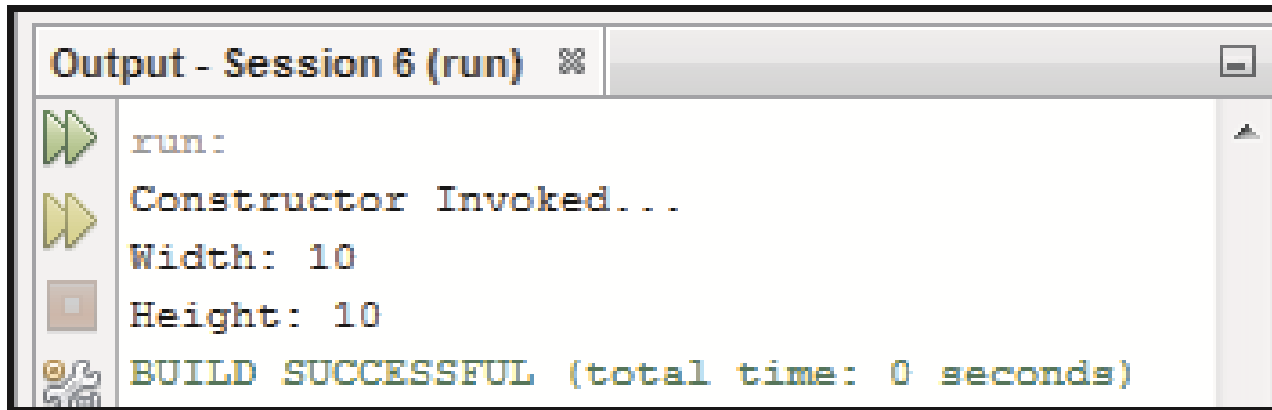
- ◆ Following code snippet demonstrates the code to invoke the constructor for the class **Rectangle**:

```
public class TestConstructor {  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
        // Instantiates an object of the Rectangle class  
        Rectangle objRec = new Rectangle();  
  
        // Accesses the instance variables using the object reference  
        System.out.println("Width: " + objRec.width);  
        System.out.println("Height: " + objRec.height);  
    }  
}
```

- ◆ The code does the following:
 - ◆ Creates an object, **objRec** of type **Rectangle**.
 - ◆ Then, the constructor is invoked.
 - ◆ The constructor initializes the instance variables of the newly created object, that is, **width** and **height** to 10.



- ◆ Following figure shows the output of the code:

A screenshot of an IDE's output window titled "Output - Session 6 (run)". The window contains the following text: "run:", "Constructor Invoked...", "Width: 10", "Height: 10", and "BUILD SUCCESSFUL (total time: 0 seconds)". On the left side of the window, there are four icons: a green double arrow, a yellow double arrow, a brown square, and a Java logo.

```
run:
Constructor Invoked...
Width: 10
Height: 10
BUILD SUCCESSFUL (total time: 0 seconds)
```



- ◆ Consider a situation, where the constructor method is not defined for a class.
- ◆ In such a scenario, an implicit constructor is invoked by the JVM for initializing the objects.
- ◆ This implicit constructor is also known as default constructor.
- ◆ **Default Constructor:**
 - ◆ Created for the classes where explicit constructors are not defined.
 - ◆ Initializes the instance variables of the newly created object to their default values.



- ◆ Following table lists the default values assigned to instance variables of the class depending on their data types:

Data Type	Default Value
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0
char	'\u0000'
boolean	False
String (any object)	Null



- ◆ Following code snippet demonstrates a class **Employee** for which no constructor has been defined:

```
public class Employee {  
    // Declares instance variables  
    String employeeName;  
    int employeeAge;  
    double employeeSalary;  
    boolean maritalStatus;  
    /**  
     * Accesses the instance variables and displays  
     * their values using the println() method  
     */  
    void displayEmployeeDetails() {  
        System.out.println("Employee Details");  
        System.out.println("=====");  
        System.out.println("Employee Name: " + employeeName);  
        System.out.println("Employee Age: " + employeeAge);  
        System.out.println("Employee Salary: " + employeeSalary);  
        System.out.println("Employee MaritalStatus:" + maritalStatus);  
    }  
}
```

- ◆ The code declares a class **Employee** with instance variables and an instance method **displayEmployeeDetails()** that prints the value of the instance variables.



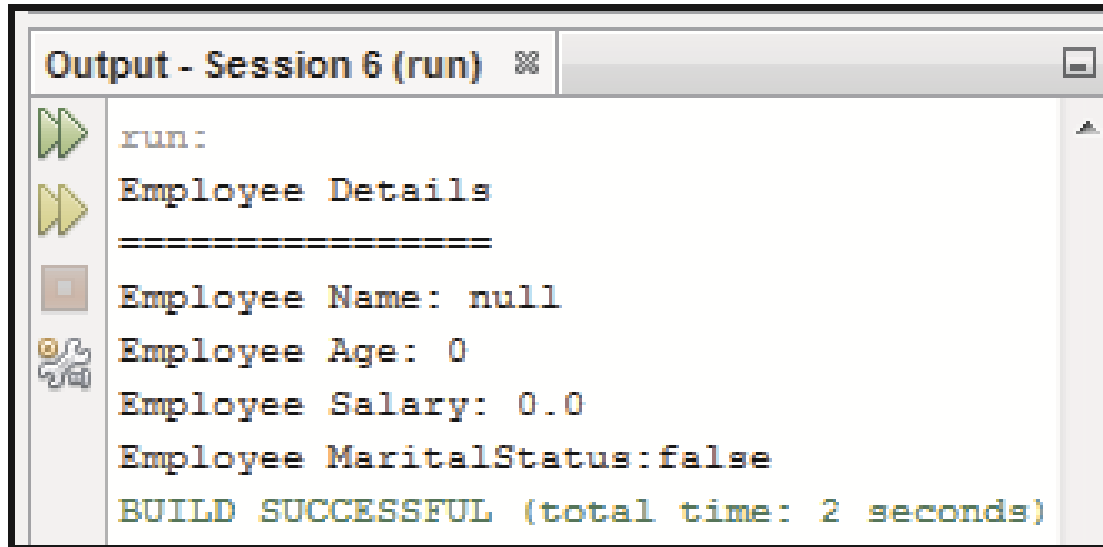
- ◆ Following code snippet demonstrates a class containing the `main()` method that creates an instance of the class **Employee** and invokes its methods:

```
public class TestEmployee {  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
        // Instantiates an Employee object and initializes it  
        Employee objEmp = new Employee();  
  
        // Invokes the displayEmployeeDetails() method  
        objEmp.displayEmployeeDetails();  
    }  
}
```

- ◆ As the **Employee** class does not have any constructor defined for itself, a **default constructor** is created for the class at runtime.
- ◆ When the statement **new Employee()** is executed, the object is allocated in memory and the instance variables are initialized to their default values by the constructor.
- ◆ Then, the method **displayEmployeeDetails()** is executed which displays the values of the instance variables referenced by the object, **objEmp**.



- ◆ Following figure shows the output of the code:

A screenshot of an IDE's output window titled "Output - Session 6 (run)". The window contains the following text: "run:", "Employee Details", a separator line of equals signs, "Employee Name: null", "Employee Age: 0", "Employee Salary: 0.0", "Employee MaritalStatus:false", and "BUILD SUCCESSFUL (total time: 2 seconds)". On the left side of the window, there are several icons: a green double arrow, a yellow double arrow, a grey square, and a gear icon.

```
run:
Employee Details
=====
Employee Name: null
Employee Age: 0
Employee Salary: 0.0
Employee MaritalStatus:false
BUILD SUCCESSFUL (total time: 2 seconds)
```



- ◆ The parameterized constructor contains a list of parameters that initializes instance variables of an object.
- ◆ The value for the parameters is passed during the object creation.
- ◆ This means each object will be initialized with different set of values.
- ◆ Following code snippet demonstrates a code that declares parameterized constructor for the **Rectangle** class:

```
public class Rectangle {  
    int width;  
    int height;  
  
    /**  
     * A default constructor for Rectangle class  
     */  
  
    Rectangle() {  
        System.out.println("Constructor Invoked...");  
        width = 10;  
        height = 10;  
    }  
}
```


Parameterized Constructor 2-5



```
/**
 * A parameterized constructor with two parameters
 * @param wid will store the width of the rectangle
 * @param heig will store the height of the rectangle
 */
Rectangle (int wid, int heig) {
    System.out.println("Parameterized Constructor");
    width = wid;
    height = heig;
}

/**
 * This method displays the dimensions of the Rectangle object
 */
void displayDimensions() {
    System.out.println("Width: " + width);
    System.out.println("Height: " + height);
}
}
```

- ◆ The code declares a parameterized constructor, **Rectangle (int wid, int heig)**.
- ◆ During execution, the constructor will accept the values in two parameters and assigns them to **width** and **height** variable respectively.



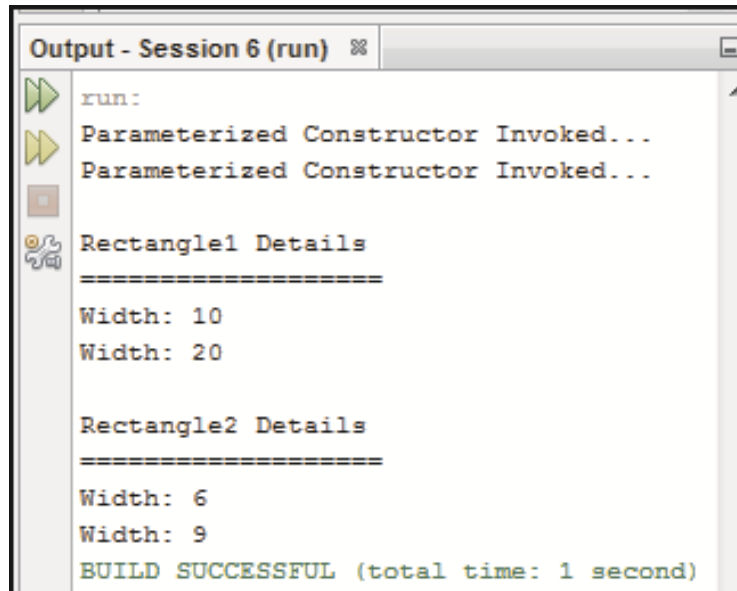
- ◆ Following code snippet demonstrates the code with `main()` method that creates objects of type **Rectangle** and initializes them with parameterized constructor:

```
public class RectangleInstances {  
  
    /**  
     * @param args the command line arguments  
     */  
  
    public static void main(String[] args) {  
        // Declare and initialize two objects for Rectangle class  
        Rectangle objRec1 = new Rectangle(10, 20);  
        Rectangle objRec2 = new Rectangle(6, 9);  
  
        // Invokes displayDimensions() method to display values  
        System.out.println("\nRectangle1 Details");  
        System.out.println("=====");  
        objRec1.displayDimensions();  
        System.out.println("\nRectangle2 Details");  
        System.out.println("=====");  
        objRec2.displayDimensions();  
    }  
}
```

Parameterized Constructor 4-5



- ◆ The statement `Rectangle objRect1 = new Rectangle(10, 20);` instantiates an object.
- ◆ During instantiation, the following things happen in a sequence:
 - ◆ Memory allocation is done for the new instance of the class.
 - ◆ Values 10 and 20 are passed to the parameterized constructor, `Rectangle(int wid, int heig)` which initializes the object's instance variables `width` and `height`.
 - ◆ Finally, the reference of the newly created instance is returned and stored in the object, `objRect1`.
- ◆ Following figure shows the output of the code:



```
Output - Session 6 (run) %
run:
Parameterized Constructor Invoked...
Parameterized Constructor Invoked...

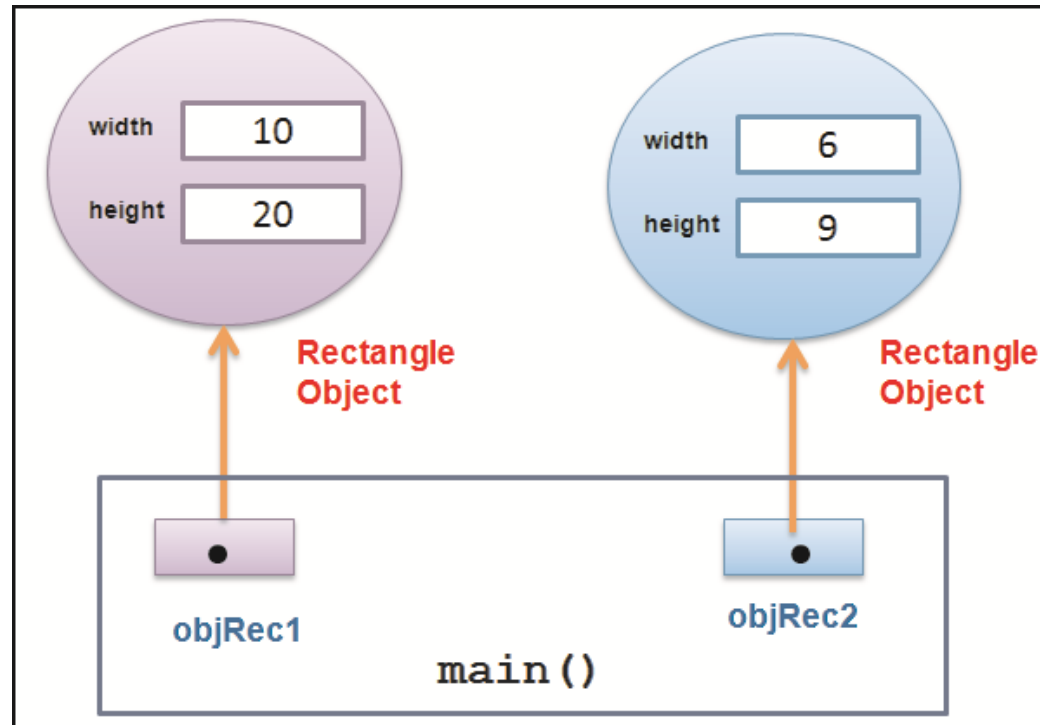
Rectangle1 Details
=====
Width: 10
Width: 20

Rectangle2 Details
=====
Width: 6
Width: 9
BUILD SUCCESSFUL (total time: 1 second)
```

Parameterized Constructor 5-5



- ◆ Following figure displays both the instance of the class **Rectangle**.
- ◆ Each object contains its own copy of instance variables that are initialized through constructor:





- ◆ The memory comprises two components namely, stack and heap.

Stack

- It is an area in the memory which stores object references and method information.
- It stores parameters of a method and local variables.

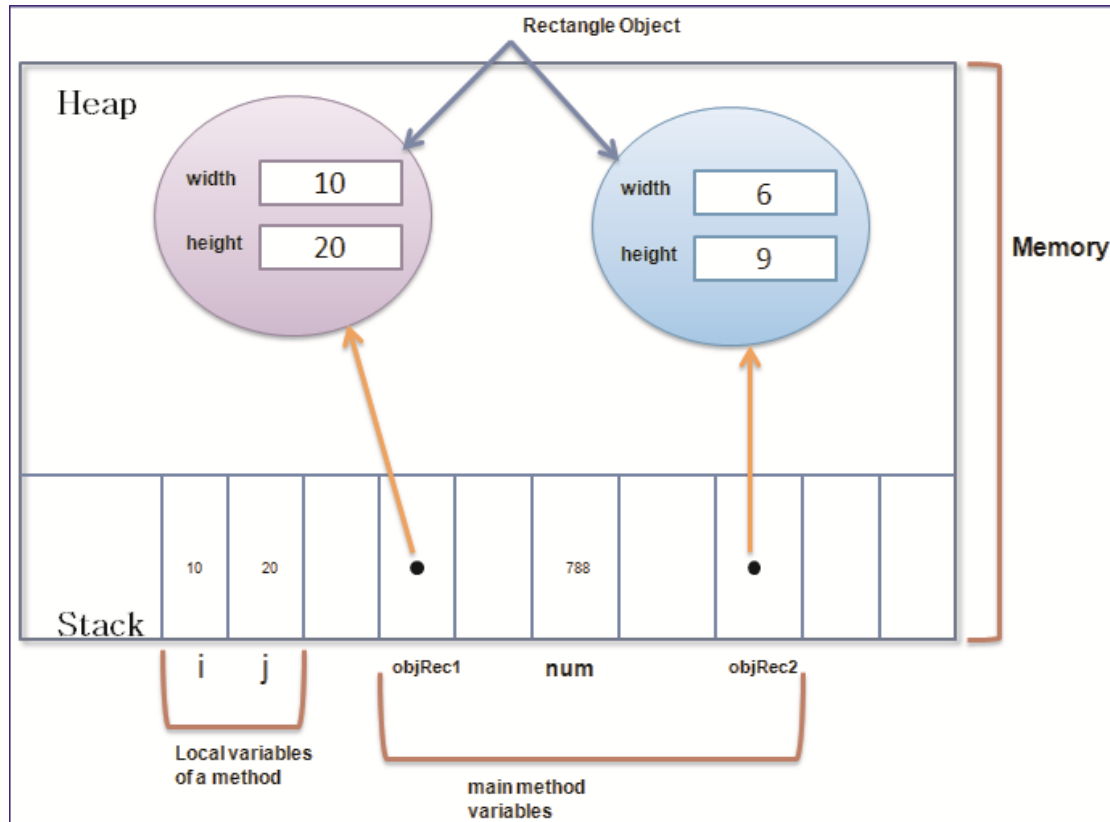
Heap

- It is area of memory deals with dynamic memory allocations.
- In Java, objects are allocated physical memory space on the heap at runtime, that is, whenever JVM executes the `new` operator.

Memory Management in Java 2-2



- ◆ Following figure shows the memory allocation for objects in stack and heap for **Rectangle** object:

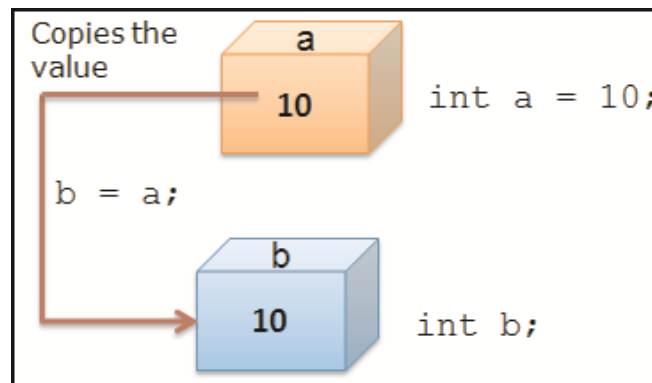


- ◆ The heap memory grows as and when the physical allocation is done for objects.
- ◆ Hence, JVM provides a garbage collection routine which frees the memory by destroying objects that are no longer required in Java program.



◆ Working with primitive data types:

- ◆ The value of one variable can be assigned to another variable using the assignment operator.
- ◆ For example, `int a = 10; int b = a;` copies the value from variable **a** and stores it in the variable **b**.
- ◆ Following figure shows assigning of a value from one variable to another:



◆ Working with object references:

- ◆ Similar to primitive data types, the value stored in an object reference variable can be copied into another reference variable.
- ◆ Both the reference variables must be of same type, that is, both the references must belong to the same class.

Assigning Object References 2-4



- ◆ Following code snippet demonstrates assigning the reference of one object into another object reference variable:

```
public class TestObjectReferences {  
  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
  
        /* Instantiates an object of type Rectangle and stores  
         * its reference in the object reference variable, objRec1  
         */  
        Rectangle objRec1 = new Rectangle(10, 20);  
  
        // Declares a reference variable of type Rectangle  
        Rectangle objRec2;
```

- ◆ The **objRec1** points to the object that has been allocated memory and initialized to 10 and 20.
- ◆ The **objRec2** is an object reference variable that does not point to any object.

Assigning Object References 3-4



```
// Assigns the value of objRec1 to objRec2
objRec2 = objRec1;
System.out.println("\nRectangle1 Details");
System.out.println("=====");

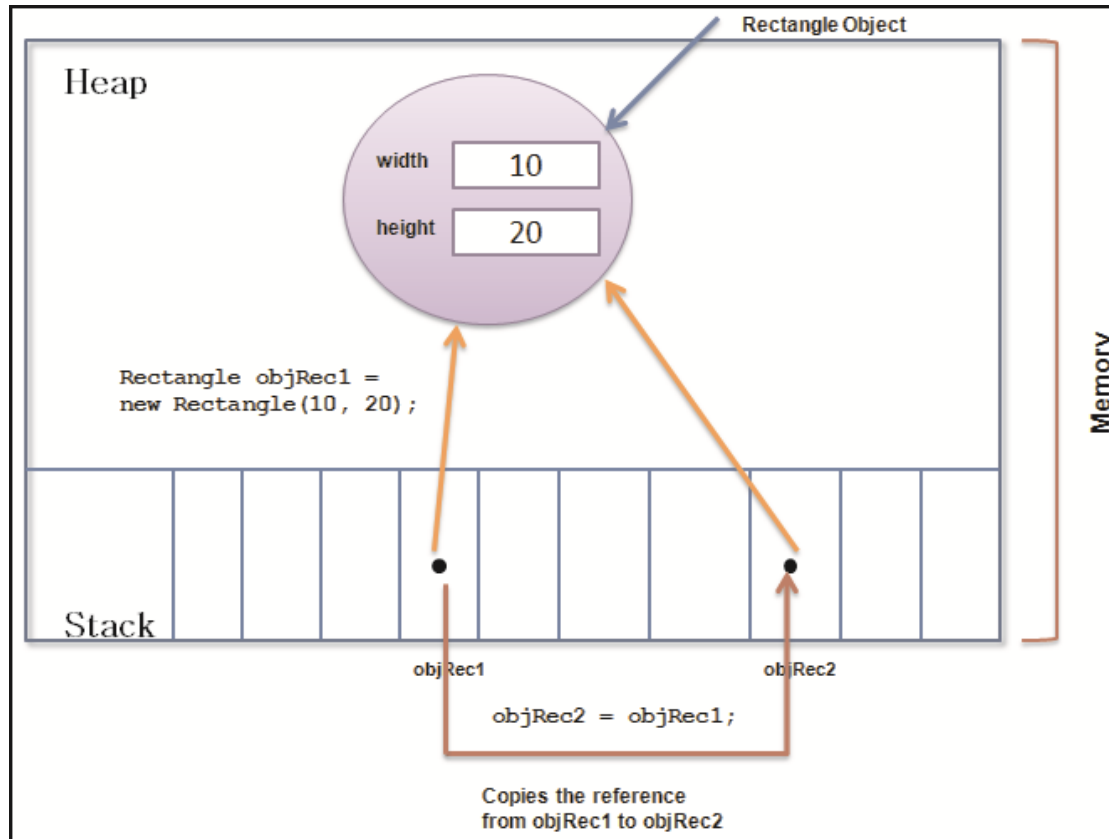
/* Invokes the method that displays values of the
 * instance variables for object, objRec1
 */
objRec1.displayDimensions();
System.out.println("\nRectangle2 Details");
System.out.println("=====");
objRec2.displayDimensions();
}
}
```

- ◆ The statement, **objRec2 = objRec1;** copies the address in **objRec1** into **objRec2**.
- ◆ Thus, the references are copied between the variables created on the stack without affecting the actual objects created on the heap.

Assigning Object References 4-4



- ◆ Following figure shows the assigning of reference for the statement, `objRec2 = objRec1;`





- ◆ In OOP languages, the concept of hiding implementation details of an object is achieved by applying the concept of encapsulation.

Encapsulation

- It is a mechanism which binds code and data together in a class.
- Its main purpose is to achieve data hiding within a class which means:
 - Implementation details of what a class contains need not be visible to other classes and objects that use it.
 - Instead, only specific information can be made visible to the other components of the application and the rest can be hidden.
- By hiding the implementation details about what is required to implement the specific operation in the class, the usage of operation becomes simple.



- ◆ In Java, the data hiding is achieved by using **access modifiers**.
- ◆ **Access Modifiers:**
 - ◆ Determine how members of a class, such as instance variable and methods are accessible from outside the class.
 - ◆ Decide the scope or visibility of the members.
 - ◆ Are of four types:

public

- Members declared as public can be accessed from anywhere in the class as well as from other classes.

private

- Members are accessible only from within the class in which they are declared.

protected

- Members to be accessible from within the class as well as from within the derived classes.

package (default)

- Allows only the public members of a class to be accessible to all the classes present within the same package.
- This is the default access level for all the members of the class.



- ◆ As a general rule in Java, the details and implementation of a class is hidden from the other classes or external objects in the application.
- ◆ This is done by making instance variables as `private` and instance methods as `public`.
- ◆ Following code snippet demonstrates the use of the concept of encapsulation in the class **Rectangle**:

```
public class Rectangle {  
    // Declares instance variables  
    private int width;  
    private int height;  
    /*  
    * Declares a no-argument constructor  
    */  
    public Rectangle() {  
        System.out.println("Constructor Invoked...");  
        width = 10;  
        height = 10;  
    }  
}
```

- ◆ The access specifiers of the instance variables, **width** and **height** are changed from default to `private` which means that the class fields are not directly accessible from outside the class.



```
/**
 * Declares a parameterized constructor with two parameters
 * @param wid
 * @param heig
 */

public Rectangle (int wid, int heig) {
    System.out.println("Parameterized Constructor Invoked...");
    width = wid;
    height = heig;
}

/**
 * Displays the dimensions of the Rectangle object
 */
public void displayDimensions(){
    System.out.println("Width: " + width);
    System.out.println("Height: " + height);
}
}
```

- ◆ The access modifiers for the methods are changed to `public`.
- ◆ Thus, the users can access the class members through its methods without impacting the internal implementation of the class.



- ◆ They provide a way to create an object and initialize its fields.
- ◆ They complement the use of constructors to initialize objects.
- ◆ There are two approaches to initialize the fields or instance variables of the newly created objects:
 - ◆ Using instance variable initializers
 - ◆ Using initialization block
- ◆ **Instance Variable Initializers:**
 - ◆ In this approach, you specify the names of the fields and/or properties to be initialized, and give an initial value to each of them.
 - ◆ Following code snippet demonstrates a Java program that declares a class, **Person** and initializes its fields:

```
public class Person {  
    private String name = "John";  
    private int age = 12;  
  
    /**  
     * Displays the details of Person object  
     */  
}
```



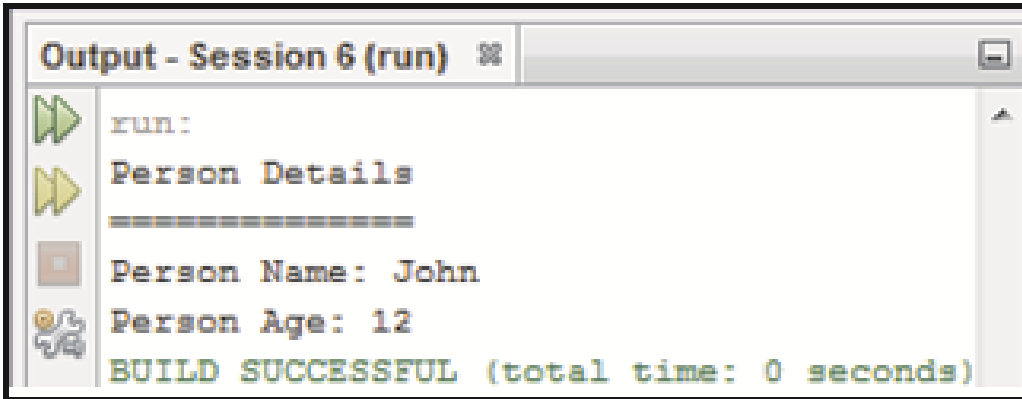
```
void displayDetails() {  
    System.out.println("Person Details");  
    System.out.println("=====");  
    System.out.println("Person Name: " + name);  
}  
}
```

- ◆ The instance variables **name** and **age** are initialized to values '**John**' and **12** respectively.
- ◆ Following code snippet shows the class with `main()` method that creates objects of type **Person**:

```
public class TestPerson {  
  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
        Person objPerson1 = new Person();  
        objPerson1.displayDetails();  
    }  
}
```




- ◆ The code creates an object of type **Person** and invokes the method to display the details.
- ◆ Following figure shows the output of the code:



```
Output - Session 6 (run)
run:
Person Details
=====
Person Name: John
Person Age: 12
BUILD SUCCESSFUL (total time: 0 seconds)
```

- ◆ **Initialization Block:**
 - ◆ In this approach, an initialization block is specified within the class.
 - ◆ The initialization block is executed before the execution of constructors during an object initialization.



- ◆ Following code snippet demonstrates the class **Account** with an initialization block:

```
public class Account {  
    private int accountID;  
    private String holderName;  
    private String accountType;  
  
    /**  
     * Initialization block  
     */  
    {  
        accountID = 100;  
        holderName = "John Anderson";  
        accountType = "Savings";  
    }  
}
```

- ◆ In the code, the initialization blocks initializes the instance variables or fields of the class.



```
/**
 * Displays the details of Account object
 */
public void displayAccountDetails() {
    System.out.println("Account Details");
    System.out.println("=====");
    System.out.println("Account ID: " + accountID + "\nAccount
Type: " + accountType);
}
```

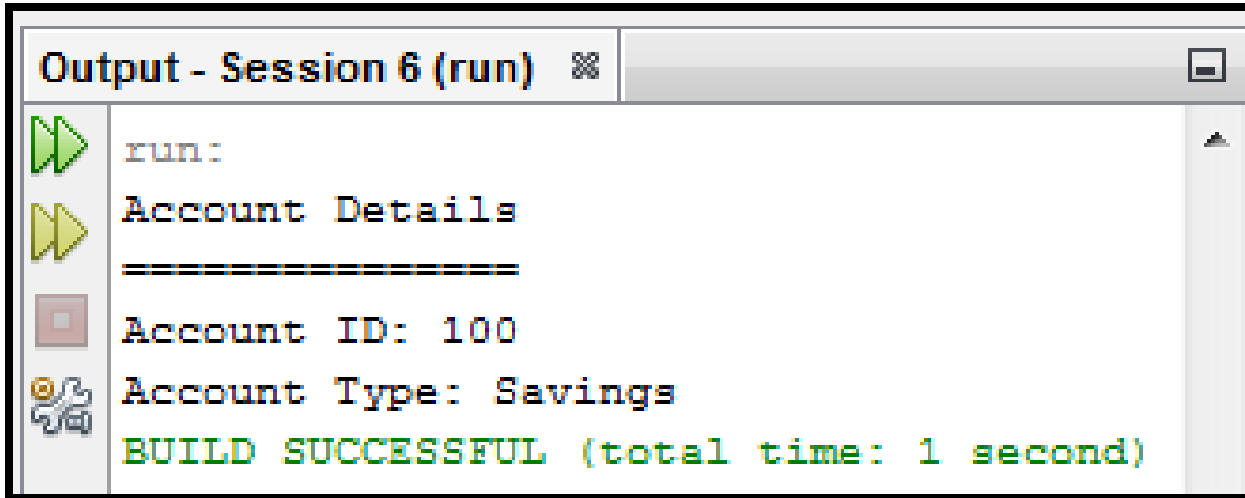
- ◆ Following code snippet shows the code with `main()` method to initialize the **Account** object through initialization block:

```
public class TestInitializationBlock {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Account objAccount = new Account();
        objAccount.displayAccountDetails();
    }
}
```

Object Initializers 6-6



- ◆ Following figure shows the output of the code:

A screenshot of an IDE's output window titled "Output - Session 6 (run)". The window contains the following text: "run:", "Account Details", a line of equals signs, "Account ID: 100", "Account Type: Savings", and "BUILD SUCCESSFUL (total time: 1 second)". On the left side of the window, there are four icons: a green play button, a yellow play button, a red stop button, and a blue icon representing a class or package structure.

```
run:
Account Details
=====
Account ID: 100
Account Type: Savings
BUILD SUCCESSFUL (total time: 1 second)
```



- ◆ The class is a logical construct that defines the shape and nature of an object.
- ◆ Objects are the actual instances of the class and are created using the new operator. The new operator instructs JVM to allocate the memory for the object.
- ◆ The members of a class are fields and methods. Fields define the state and are referred to as instance variables, whereas methods are used to implement the behavior of the objects and are referred to as instance methods.
- ◆ Each instance created from the class will have its own copy of the instance variables, whereas methods are common for all instances of the class.
- ◆ Constructors are methods that are used to initialize the fields or perform startup operations only once when the object of the class is instantiated.
- ◆ The heap area of memory deals with the dynamic memory allocations for objects, whereas the stack area holds the object references.
- ◆ Data encapsulation hides the instance variables that represents the state of an object through access modifiers. The only interaction or modification on objects is performed through the methods.