codecademy

Object-Oriented Java

Java objects' state and behavior

In Java, instances of a class are known as objects. Every object has state and behavior in the form of instance fields and methods respectively.

```
public class Person {
  // state of an object
  int age;
  String name;

  // behavior of an object
  public void set_value() {
    age = 20;
    name = "Robin";
  }
  public void get_value() {
    System.out.println("Age is " +
age);
    System.out.println("Name is " +
name);
  }

  // main method
  public static void main(String []
args) {
    // creates a new Person object
    Person p = new Person();

    // changes state through
behavior
    p.set_value();
  }
}
```

Java instance

Java instances are objects that are based on classes.

For example, `Bob` may be an instance of the class `Person`.

Every instance has access to its own set of variables which are known as *instance fields*, which are variables declared within the scope of the instance. Values for instance fields are assigned within the constructor method.

```
public class Person {
  int age;
  String name;

  // Constructor method
  public Person(int age, String
name) {
    this.age = age;
    this.name = name;
  }

  public static void main(String[]
args) {
    Person Bob = new Person(31,
"Bob");
    Person Alice = new Person(27,
"Alice");
  }
}
```

Java dot notation

In Java programming language, we use `.` to access the variables and methods of an object or a Class. This is known as *dot notation* and the structure looks like this-

`instanceOrClassName.fieldOrMethodName`

```
public class Person {
  int age;

  public static void main(String []
args) {
    Person p = new Person();

    // here we use dot notation to
set age
    p.age = 20;

    // here we use dot notation to
access age and print
    System.out.println("Age is " +
p.age);
    // Output: Age is 20
  }
}
```

Constructor Method in Java.

Java classes contain a *constructor* method which is used to create instances of the class.
The constructor is named after the class. If no constructor is defined, a default empty constructor is used.

```
public class Maths {
  public Maths() {
    System.out.println("I am
constructor");
  }
  public static void main(String []
args) {
    System.out.println("I am main");
    Maths obj1 = new Maths();
  }
}
```

Creating a new Class instance in Java

In Java, we use the `new` keyword followed by a call to the class constructor in order to create a new *instance* of a class.
The constructor can be used to provide initial values to instance fields.

```
public class Person {
  int age;

  public Person(int a) {
    age = a;
  }

  public static void main(String []
args) {
    // Here, we create a new
instance of the Person class
    Person p = new Person(20);

    System.out.println("Age is " +
p.age);
    // Output: Age is 20
  }
}
```

The body of a Java method

In Java, we use curly brackets `{}` to enclose the body of a method.

The statements written inside the `{}` are executed when a method is called.

```
public class Maths {
  public static void sum(int a, int
b) { // Start of sum
    int result = a + b;
    System.out.println("Sum is " +
result);
  } // End of sum


  public static void main(String []
args) {
    // Here, we call the sum method
    sum(10, 20);
    // Output: Sum is 30
  }
}
```

Method parameters in Java

In java, parameters are declared in a method definition. The parameters act as variables inside the method and hold the value that was passed in. They can be used inside a method for printing or calculation purposes.
In the example, a and b are two parameters which, when the method is called, hold the value 10 and 20 respectively.

```
public class Maths {
  public int sum(int a, int b) {
    int k = a + b;
    return k;
  }

  public static void main(String []
args) {
    Maths m = new Maths();
    int result = m.sum(10, 20);
    System.out.println("sum is " +
result);
    // prints - sum is 30
  }
}
```

Java Variables Inside a Method

Java variables defined inside a method cannot be used outside the scope of that method.

```
//For example, `i` and `j` variables
are available in the `main` method
only:

public class Maths {
  public static void main(String []
args) {
    int i, j;
    System.out.println("These two
variables are available in main
method only");
  }
}
```

Returning info from a Java method

A Java method can return any value that can be saved in a variable. The value returned must match with the return type specified in the method signature.

The value is returned using the `return` keyword.

```
public class Maths {

  // return type is int
  public int sum(int a, int b) {
    int k;
    k = a + b;

    // sum is returned using the
return keyword
    return k;
  }

  public static void main(String []
args) {
    Maths m = new Maths();
    int result;
    result = m.sum(10, 20);
    System.out.println("Sum is " +
result);
    // Output: Sum is 30
  }
}
```

Java method signature

In Java, methods are defined with a *method signature*, which specifies the scope (private or public), return type, name of the method, and any parameters it receives.

```
// Here is a public method named sum
whose return type is int and has two
parameters a and b
public int sum(int a, int b) {
  return(a + b);
}
```