HSBC Technology Graduate Training

Programming Fundamentals: Java

Day 2 (Afternoon)

Tuesday 27 October 2020 | 2pm

Contents

Static members

STATIC MEMBERS

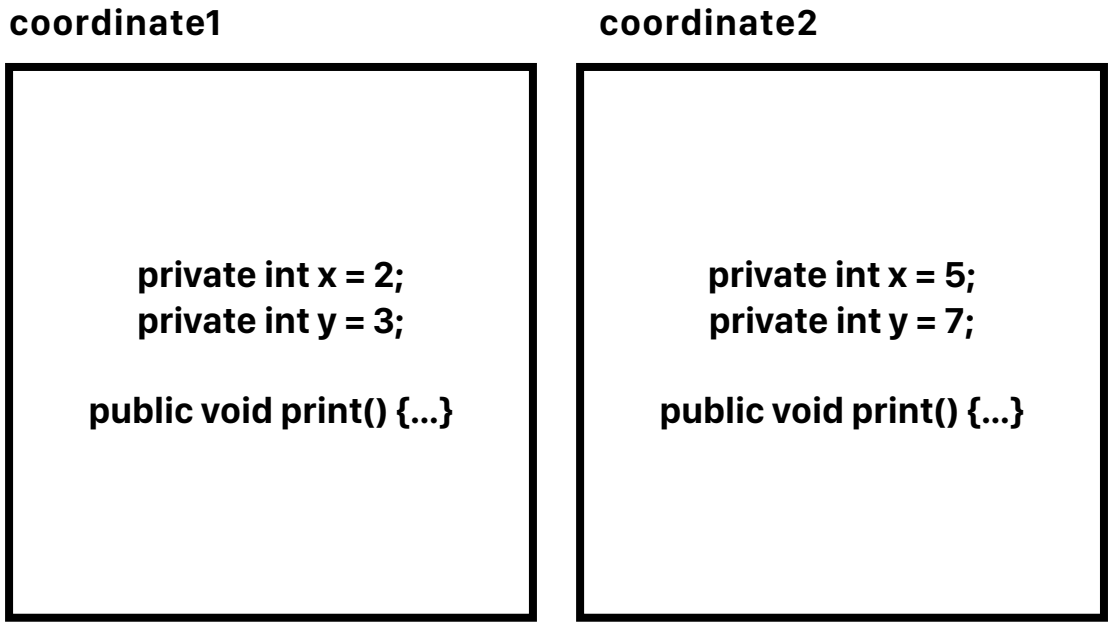
- Let's imagine a class Coordinate that stores two data members x and y.
- The Coordinate class has a constructor that sets the x and y values.
- The Coordinate class also defines a method that prints the coordinate.
- What happens when we create an object from this class?

```
public class Coordinate {

  private int x;
  private int y;

  public Coordinate(int a, int b) {
    x = a;
    y = b;
  }

  public void print() {
    System.out.println(x + ',' + y);
  }
}
```

STATIC MEMBERS

- Let's create 2 Coordinate objects.
- What is inside these objects? Simple: whatever we declared in the class.
- In this case, each object will have:
  - Its own x data member.
  - Its own y data member.
  - Its own print method.

```
public class Main {

    public static void main(String args[]) {

        Coordinate coordinate1 = new Coordinate(2, 3);

        Coordinate coordinate2 = new Coordinate(5, 7)'

    }

}
```



STATIC MEMBERS

- What do we see if we call print() on coordinate1 vs. print() on coordinate2?

```
public class Main {

    public static void main(String args[]) {

        Coordinate coordinate1 = new Coordinate(2, 3);

        Coordinate coordinate2 = new Coordinate(5, 7);

        coordinate1.print(); // Prints "2,3"

        coordinate2.print(); // Prints "5,7"

    }

}
```

- So coordinate1 has its own x and y values, and coordinate2 also has its own x and y value.
- This means that when you instantiate an object, you make a copy of the members declared within the relevant class.

STATIC MEMBERS

- What if we don't want to make a copy of members?
- We can use the static keyword.
- A member that is declared as static will be tied to a class and not an object.
- In other words, when you create a new object of a class, the static members will not be copied into the object.
- Imagine now we wish that all Coordinate objects we work with has a unit, e.g. "miles", "km" etc.
- We want all objects of Coordinate to have the same unit.
- We can declare a unit member as static!

```
public class Coordinate {

  private int x;
  private int y;

  static String unit;

  public Coordinate(int a, int b) {
    x = a;
    y = b;
  }

  public void print() {
    System.out.println(x + ',' + y);
  }
}
```

- The example below shows the effect of declaring unit in Coordinate class as static.

```java
public class Main {
  public static void main(String args[]) {

    // Set Coordinate unit.
    Coordinate.unit = "cm";

    Coordinate coordinate1 = new Coordinate(2, 3);

    Coordinate coordinate2 = new Coordinate(5, 7);

    coordinate1.print(); // Prints "2,3"

    coordinate2.print(); // Prints "5,7"

    System.out.println(Coordinate.unit); // Prints "cm"

    System.out.println(coordinate1.unit); // Prints "cm"
    System.out.println(coordinate2.unit); // Prints "cm"

  }
}
```

Notice how we can set the value of unit before creating an object from the class Coordinate. This is because static members belong to classes, not objects.

Objects don't have their own copy of static members. In this case, the static member unit belongs to the Coordinate class, so ALL Coordinate objects will share the same unit member.

STATIC MEMBERS

- We can also create static methods.
- This means the method would belong to the class instead of the object.
- For example, imagine we add the static keyword to print in class Coordinate.
- Is this code valid?

```
public class Coordinate {

  private int x;
  private int y;

  static String unit

  public Coordinate(int a, int b) {
    x = a;
    y = b;
  }

  public static void print() {
    System.out.println(x + "," + y);
  }

}
```

STATIC MEMBERS

- This code is not valid.
- Data members x and y are NOT static. They belong to an object of type Coordinate.
- In other words, each Coordinate object has its own x and y values.
- However, the print() method is static, meaning it belongs to a class and is not copied to an object.

```
public class Coordinate {

  private int x;
  private int y;

  static String unit

  public Coordinate(int a, int b) {
    x = a;
    y = b;
  }

  public static void print() {
    System.out.println(x + "," + y);
  }

}
```

This code will not run.

Shadowing

- **What will be printed from the following?**

```java
public class A {

    int x = 10;

    public void foo() {
        int x = 20;
        System.out.println(x);
    }

    public void boo() {
        System.out.println(x);
    }

}
```

```java
public class Main {

    public static void main(String args[]) {

        A a = new A();

        a.foo();

        a.bar();

    }

}
```

SHADOWING

- Let's see what happens when we invoke the method a.foo()

```java
public class A {

    int x = 10;

    public void foo() {
        int x = 20;
        System.out.println(x);
    }

    public void bar() {
        System.out.println(x);
    }

}
```

- Notice that the class A has a data member x.
- However, in the method foo(), we declare a variable, also with the name of x.
- Which x will foo() use? Two options:
  - Global variable x
  - Local variable x
- Method will always prioritise local variables over global variables.
- In this case, foo() will print 20.

SHADOWING

- Let's see what happens when we invoke the method a.bar()

```
public class A {

    int x = 10;

    public void foo() {
        int x = 20;
        System.out.println(x);
    }

    public void bar() {
        System.out.println(x);
    }

}
```

- Notice that the class A has a data member x.
- Unlike foo(), bar() does not declare a local variable with the same name x.
- There is no conflict here, so bar() will print the value of the data member x in class A. In this case – 10.

SHADOWING

- **What will be printed from the following?**
- **The answer is 20 followed by 10.**

```java
public class A {

    int x = 10;

    public void foo() {
        int x = 20;
        System.out.println(x);
    }

    public void boo() {
        System.out.println(x);
    }

}
```

```java
public class Main {

    public static void main(String args[]) {

        A a = new A();

        a.foo();

        a.bar();

    }

}
```

SHADOWING

- **This demonstrates a concept called shadowing.**
- **Shadowing occurs where two variables with the same name are visible at some point within the code.**
- **The lower-level scoped variable is ALWAYS prioritised over high-scoped variables.**

```java
public class A {

    int x = 10;

    public void foo() {
        int x = 20;
        System.out.println(x);
    }

    public void boo() {
        System.out.println(x);
    }

}
```

```java
public class Main {

    public static void main(String args[]) {

        A a = new A();

        a.foo();

        a.bar();

    }

}
```

this

THIS

- Following from Shadowing, what if we wanted to use the global data member instead?
- We can explicitly state this using the “this” notation.
- foo() now prints 10 instead of 20.

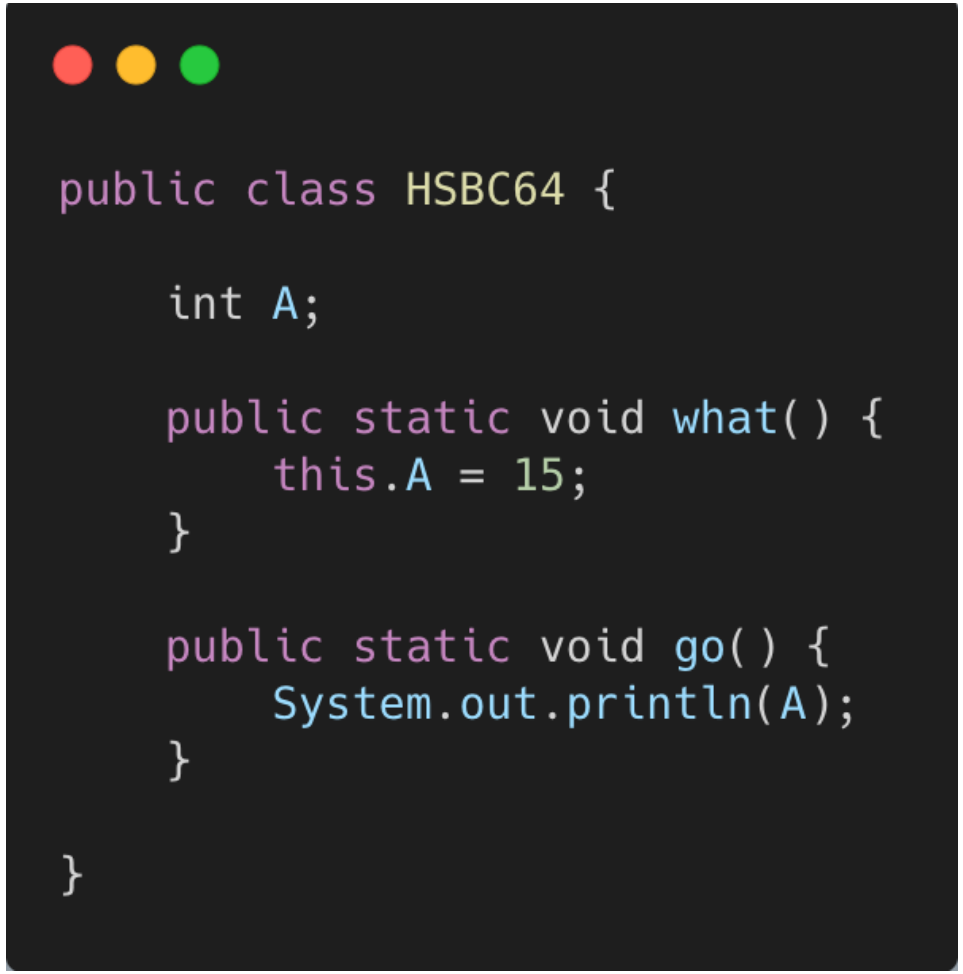
```java
public class A {

    int x = 10;

    public void foo() {
        int x = 20;
        System.out.println(this.x);
    }

    public void bar() {
        System.out.println(x);
    }

}
```

```java
public class Main {

    public static void main(String args[]) {

        A a = new A();

        a.foo();

        a.bar();

    }

}
```

Example

EXAMPLE

- **Is this code valid?**



```
public class HSBC64 {

    int A;

    public static void what() {
        this.A = 15;
    }

    public static void go() {
        System.out.println(A);
    }

}
```

EXAMPLE

- **No. Few reasons:**
  - **Data member A is not static yet the what() method is static.**
  - **We can't use this in a static method since it does not exist within a object.**

```
public class HSBC64 {

    int A;

    public static void what() {
        this.A = 15;
    }

    public static void go() {
        System.out.println(A);
    }

}
```

Constructors in Child Classes

CONSTRUCTORS IN CHILD CLASSES

- **Imagine the following two classes.**

```java
public class One {

  public One() {
    System.out.println("Running constructor in One");
  }

}
```

```java
public class Two extends One {

  public Two() {
    System.out.println("Running constructor in Two");
  }

}
```

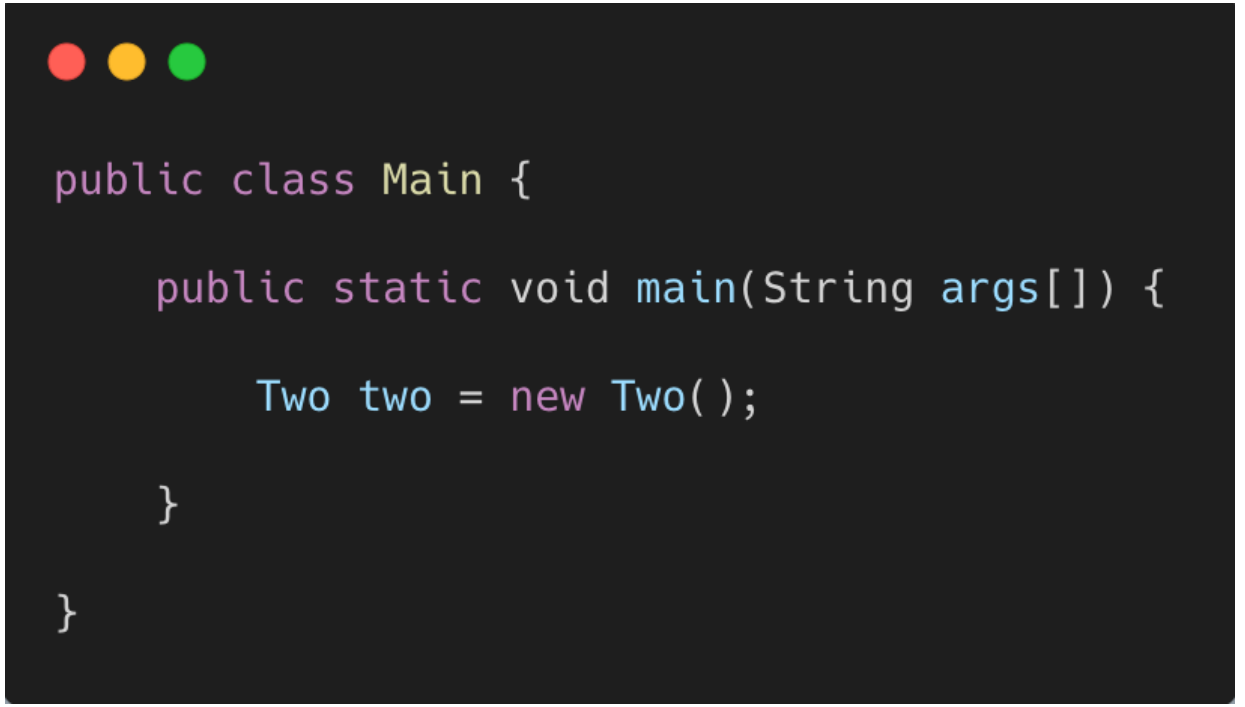
CONSTRUCTORS IN CHILD CLASSES

- **What will be printed if we run the following code?**

```
public class Main {

    public static void main(String args[]) {

        Two two = new Two();

    }

}
```

CONSTRUCTORS IN CHILD CLASSES

- **The following will be printed:**
  - `Running constructor in One`
  - `Running constructor in Two`



```java
public class Main {

    public static void main(String args[]) {

        Two two = new Two();

    }

}
```

CONSTRUCTORS IN CHILD CLASSES

- When we create an object from a class that inherits another, the constructors of the parent classes will be called too.
- In the earlier example, we created an object from class Two.
- Class Two inherits class One.
- So when we create the new object from class Two, the constructor of class One is called followed by the constructor of class Two.

```
public class One {

  public One() {
    System.out.println("Running constructor in One");
  }

}
```

```
public class Two extends One {

  public Two() {
    System.out.println("Running constructor in Two");
  }

}
```

Super

SUPER

- Remember how classes may have more than one constructor provided they have different signatures?
- Let's apply that to class One from the previous example.

```
public class One {

  public One() {
    System.out.println("Running constructor in One");
  }

  public One(int x) {
    System.out.println("Running constructor with parameter " + x + " in One");
  }

}
```

```
public class Two extends One {

  public Two() {
    System.out.println("Running constructor in Two");
  }

}
```

- How does the constructor in Two know which parent constructor to call?
- Before we can answer that question, we must understand what Super() will do.

SUPER

- **super()** is a special method that invokes a constructor of the parent class.
- By default, Java adds **super()** to any child constructors if none is defined.
- For example, the class **Two** does not call **super()** in its constructor.
- So Java adds (in the background) **super()** as the first line in the constructor of **Two**.

```
public class Two {

    public Two() {
        super(); // Added by Java in the background.
        System.out.println("Running constructor in Two");
    }

}
```

SUPER

- So in this case, **super()** (with no arguments) will call the corresponding constructor in the parent class **One** with no arguments.
- Note: this will work without **super()** too since Java will invoke the default constructor of the parent class inherently.

Calling super() will run this constructor in class One.

```java
public class One {

  public One() {
    System.out.println("Running constructor in One");
  }

  public One(int x) {
    System.out.println("Running constructor with parameter " + x + " in One");
  }

}
```

```java
public class Two {

    public Two() {
        super(); // Added by Java in the background.
        System.out.println("Running constructor in Two");
    }

}
```

SUPER

- If we call super with one integer as an argument, it will call the corresponding constructor in the parent.

```java
public class One {

  public One() {
    System.out.println("Running constructor in One");
  }

  public One(int x) {
    System.out.println("Running constructor with parameter " + x + " in One");
  }

}
```

```java
public class Two {

    public Two() {
        super(3); // Added by Java in the background.
        System.out.println("Running constructor in Two");
    }

}
```

Calling super(3) will run this constructor in class One.

SUPER

- If we call super(), it must be the first instruction of the constructor.
- The following code is not valid since super(2) is the second instruction in the constructor.

```java
public class One {

  public One() {
    System.out.println("Running constructor in One");
  }

  public One(int x) {
    System.out.println("Running constructor with parameter " + x + " in One");
  }

}
```

```java
public class Two {

    public Two() {
        System.out.println("Running constructor in Two");
        super(2);
    }

}
```

this()

THIS()

- **Super()** calls the constructor of a parent class.
- **This()** calls a constructor within the same class.
- Imagine the following class.

```java
public class One {

  public One() {
    System.out.println("First constructor.");
  }

  public One(int x) {
    System.out.println("Second constructor.");
  }

  public One(int x, int y) {
    System.out.println("Third constructor.");
  }

}
```

THIS()

- What if we wanted to call another constructor within the same class?
- We can use this().
- In this example, we call the constructor One(int x) from the constructor One().
- So when we create an object from class One, the following will be printed:
  - `First constructor.`
  - `Second constructor.`

```java
public class One {

  public One() {
    System.out.println("First constructor.");
    this(35);
  }

  public One(int x) {
    System.out.println("Second constructor.");
  }

  public One(int x, int y) {
    System.out.println("Third constructor.");
  }

}
```