# HSBC Technology Graduate Training

## Programming Fundamentals: Java

Day 2 (Morning)

Tuesday 27 October 2020 | 9am

# Contents

# Members

# MEMBERS

- A class contains <u>members</u>.
- There are 2 types of *members*.
  - Data member (fields)
  - Code (methods)

```
class Coordinate {

  int x;
  int y;

  public void setX(int value) {
    x = value;
  }

  public void setY(int value) {
    y = value;
  }

}
```

2 fields

2 methods

4 members

# Comments

# COMMENTS

- We can add text to our code that will be ignored by the computer.
- These are called <u>comments</u>.
- We use the following syntax to declare a comment: `// COMMENT HERE`
- We can use comments to make notes in English within our code.

```
class Main {

    public static void main(String args[]) {

      int x; // This is a comment.

    }


}
```

# Inheritance

# INHERITANCE

- A class can inherit another class.
- A class that inherits another class will contain the members of its parent class.
- We use the key word *extends* to inherit a class.

```
class One {

    int x;
    int y;

}
```

```
class Two extends One {

    int z;

}
```

- In this example, class Two <u>extends</u> class One.
- In other words, class Two <u>inherits</u> class One.
- Class Two inherits <u>members</u> from class One.
- Class One is the <u>parent</u>.
- Class Two is the <u>child</u>.
- Class One has 2 members.
- Class Two has 3 members (inherits 2 members from class One).

# INHERITANCE: CREATING AN OBJECT

- We can create an object from a class that inherits another class.
- This is just like any other class.
- Notice how we can set the data members ‘z’ and ‘x’. Class Two now contains x, y, and z.

```java
class One {

    int x;
    int y;

}
```

```java
class Two extends One {

    int z;

}
```

```java
class Main {

    public static void main(String args[]) {

      Two ref; // Declare variable named 'ref'.
      ref = new Two(); // Create new object from class Two.
      ref.z = 2; // We can see 'z' from class Two.
      ref.x = 2; // We can ALSO see 'x' from class One.

    }


}
```

# Inheritance: Extending Functionality

# INHERITANCE: A PRACTICAL EXAMPLE (PART 1 OF 2)

- When can inheritance be useful in the context of programming?
- Imagine we have a class Math that contains two methods: add and subtract.

```java
class Math {

  public void add(int x, int y) {
    System.out.println(x + y);
  }

  public void subtract(int x, int y) {
    System.out.println(x - y);
  }

}
```

# INHERITANCE: A PRACTICAL EXAMPLE (PART 2 OF 2)

- What if we wanted to write a **multiply** method but don't have access to this source code of the class **Math**? Two options:
  - 1. Rewrite the **Math** class with 3 methods: **add**, **subtract**, **multiply**.
  - 2. Write a new class containing one method **multiply** that inherits the **Math** class.
- Option 2 is better because we don't repeat code that has already been written.
- The class **NewMath** now has **add**, **subtract** and **multiply** methods.

```java
class Math {

  public void add(int x, int y) {
    System.out.println(x + y);
  }

  public void subtract(int x, int y) {
    System.out.println(x - y);
  }

}
```

```java
class NewMath extends Math {

    public void multiply(int x, int y) {
        System.out.println(x * y);
    }

}
```

# Inheritance: Overriding Functionality

# INHERITANCE: A PRACTICAL EXAMPLE (PART 1 OF 3)

- **Imagine we have a class Math that contains two methods: add and subtract.**

```java
class Math {

  public void add(int x, int y) {
    System.out.println(x + y);
  }

  public void subtract(int x, int y) {
    System.out.println(x - y);
  }

}
```

# INHERITANCE: A PRACTICAL EXAMPLE (PART 2 OF 3)

- What if we wanted to use the functionality of the Math class but wanted to re-write the add method? Two options again:
  - 1. Rewrite the Math class with the 2 methods: add and subtract.
  - 2. Write a new class containing one method add.
- Option 2 is better because we don't repeat code that has already been written.
- The class NewMath2 inherits class Math and contains one method: add.

```java
class Math {

  public void add(int x, int y) {
    System.out.println(x + y);
  }

  public void subtract(int x, int y) {
    System.out.println(x - y);
  }

}
```

```java
class NewMath2 extends Math {

    public void add(int x, int y) {
        System.out.println("The result is: " + x + y);
    }

}
```

# INHERITANCE: A PRACTICAL EXAMPLE (PART 3 OF 3)

- This is known as <u>overriding.</u>
- We are rewriting a method that exists in the <u>parent</u> class.
- When we create an object from class NewMath2 and call the add method, the overridden method is called instead of the one in the parent class.

```
class Main {

    public static void main(String args[]) {

        // Declare variable for class NewMath2.
        NewMath2 ref;

        // Create an object from class NewMath2.
        ref = new NewMath2();

        // Call add method from ref.
        ref.add(2, 5);

        // Prints "The result is: 7"

    }
}
```

# Abstract Classes

# ABSTRACT CLASSES

- The Bank of England (BoE) controls other banks.
- BoE will tell other banks what they need to do to operate as a bank in England.
- BoE will create an <u>abstract</u> class like below.

```
abstract public class Bank {

  abstract public void openAccount();
  abstract public void showBalance();

}
```

A class with abstract members must be declared as abstract.

An abstract method does not have a body.

# ABSTRACT CLASSES

- So what's happening here?
- Bank of England has created an abstract class.
- We CANNOT create an object from an abstract class.
- We must inherit the abstract class and override all abstract members.
- Let's say we want to create a class HSBC.
- In order to be recognized by Bank of England as a "Bank", we must comply to the abstract class Bank defined by BoE.
- We do this by inheriting the Bank class defined by BoE.

```
public class HSBC extends Bank {

}
```

This code will not run.

- **Why does this code not run?**

```
public class HSBC extends Bank {

}
```

This code will not run.

- **Remember from the previous slide, when we inherit an abstract class, we must override all its abstract members.**
- **In this case, HSBC needs to override the following methods: openAccount and showBalance.**

- So let's override all abstract members from the inherited class Bank.

```java
abstract public class Bank {

  abstract public void openAccount();
  abstract public void showBalance();

}
```

Inherited by

```java
public class HSBC extends Bank {

  public void openAccount() {
    System.out.println("Opening an account");
  }

  public void showBalance() {
    System.out.println("Showing balance");
  }

}
```

- The class HSBC is now valid. We have <u>overridden</u> and defined all abstract members from its parent class.
- A class that can be instantiated is called a concrete class. HSBC is concrete.

# Final

# FINAL

- Classes may also have final members.
- A final member means that it cannot be overridden.
- In the example below, the method boom cannot be overridden by any class that extends the class Barclays.

```
public class Barclays {

  final public void boom() {
    System.out.println("Hello");
  }

}
```

# FINAL

- We can also have final members in abstract classes.
- The method showBalance cannot be overridden by any class that extends the class Bank.

```
abstract public class Bank {

  abstract public void openAccount();

  final public void showBalance() {
    System.out.println("Showing balance.");
  }

}
```

# FINAL

- **This is not valid as showBalance is a final method and cannot be overridden.**

```java
abstract public class Bank {

  abstract public void openAccount();

  final public void showBalance() {
    System.out.println("Showing balance.");
  }

}
```

Inherited by

```java
public class HSBC extends Bank {

  public void openAccount() {
    System.out.println("Opening an account");
  }

  public void showBalance() {
    System.out.println("Showing balance");
  }

}
```

# FINAL

- We can also declare a whole class as final.
- A class that is declared as final cannot be inherited, so cannot be extended or overridden.
- We can only use a final class.

```
final public class Barclays {

  public void boom() {
    System.out.println("Hello");
  }

}
```

# Constructors

- **Imagine we have a class Coordinate that stores an x and y value.**

```
public class Coordinate {

    int x;
    int y;

}
```

# CONSTRUCTORS

- **Remember how to create an object from a class?**

```
public class Main {

    public static void main(String args[]) {


      // Create variable and store object from class Coordinate.
      Coordinate coordinate = new Coordinate();

    }


}
```

# CONSTRUCTORS

- Notice the pair of brackets?
- This means we can pass some data into the brackets.
- In other words, we can pass some data to the Coordinate class when a new Coordinate object is created.

```
public class Main {

    public static void main(String args[]) {


      // Create variable and store object from class Coordinate.
      Coordinate coordinate = new Coordinate();

    }


}
```

# CONSTRUCTORS

- Let's say we want to create a Coordinate object with the x and y values of 2 and 5 respectively.
- We can pass 2 and 5 in the brackets when we create a new Coordinate object.

```
public class Main {

    public static void main(String args[]) {


      // Create variable and store object from class Coordinate.
      Coordinate coordinate = new Coordinate(2, 5);

    }


}
```

# CONSTRUCTORS

- How does the Coordinate class know what to do with the values passed in the brackets?
- We need to create a constructor.
- A constructor is a special method that exists in a class.
- The constructor method is called when you create a new object from that class.
- The example below shows an example of a constructor.

```java
public class Coordinate {

    int x;
    int y;

}
```

```java
public class Coordinate {

    int x;
    int y;

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

}
```

# CONSTRUCTORS

- A constructor omits the <u>void</u> notation and has the same name as the containing class.
- In this case, the constructor has a name of Coordinate.
- We want the users to pass two integers when they create a new Coordinate object, so we declare int a and int b as <u>parameters</u> to the constructor.

We can write any code in the constructor. In this case, we want to set the data members of the Coordinate object, so we assign local variable a to data member x and local variable b to data member y.

```
public class Coordinate {

    int x;
    int y;

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

}
```

We declare two input parameters. This means that when anyone creates an object from class Coordinate, they must pass two integers into the brackets as seen earlier.

- This code now works!
- When the Coordinate object is created, the constructor method is called within the Coordinate class.
- We have created a Coordinate object with the values of 2 and 5.

```
public class Main {

    public static void main(String args[]) {


      // Create variable and store object from class Coordinate.
      Coordinate coordinate = new Coordinate(2, 5);

    }


}
```

# Default Constructors

# DEFAULT CONSTRUCTORS

- If no constructor method is defined within a class, when we create an object of the class, a <u>default constructor</u> will be called instead.
- This default constructor is hidden.
- The default constructor would look like the method as seen on the right.
- It does nothing.

```
public class Coordinate {

    int x;
    int y;

}
```

```
public class Coordinate {

    int x;
    int y;

    public Coordinate() {
    }

}
```

# DEFAULT CONSTRUCTORS

- This means that a constructor method is ALWAYS called when you create a new object.
- This could be:
  - Default constructor
  - Custom defined constructor

# Constructor Overloading

# CONSTRUCTOR OVERLOADING

- **It is possible to have more than one constructor in a class, provided they all have different signatures (different number of parameters or different types).**
- **For example:**

```java
public class Main {

    public static void main(String args[]) {

        Coordinate coordinate1 = new Coordinate();

        Coordinate coordinate2 = new Coordinate(42);

        Coordinate coordinate3 = new Coordinate(34, 43);

        Coordinate coordinate4 = new Coordinate("59");

    }

}
```

```java
public class Coordinate {

    int x;
    int y;

    public Coordinate() {
    }

    public Coordinate(int a) {
        x = a;
    }

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

    public Coordinate(string a) {
        x = Integer.parseInt(a);
    }

}
```

# Polymorphism

- **Recall this example from earlier on.**
- **We can create an instance (object) from class One.**

```java
class One {

    int x;
    int y;

}
```

```java
class Two extends One {

    int z;

}
```

```java
public class Main {

    public static void main(String args[]) {

        One objectRef = new One();

    }

}
```

Type of object

Variable (reference to object in memory)

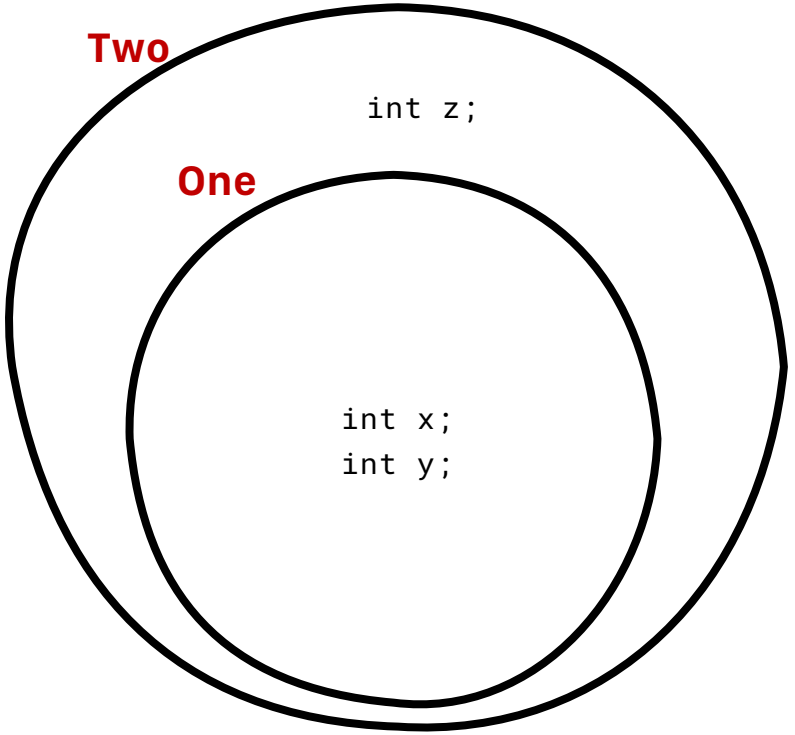
Instantiate object from class One.

# POLYMORPHISM

- Remember that class Two is a child of class One (class Two inherits class One).
- This means that an object of class Two IS-A object of class One.
- This is because an object of class Two can do everything an object of class One can.
- Here's a visual representation.

```
class One {

    int x;
    int y;

}
```

```
class Two extends One {

    int z;

}
```



- So if an object of class Two is also an object of class One.
- We say that Two IS-A One.
- We can do the following:

```java
class One {

    int x;
    int y;

}
```

```java
class Two extends One {

    int z;

}
```

```java
public class Main {

    public static void main(String args[]) {

        One objectRef = new Two();

    }

}
```

The type of reference variable is One,.

Instantiate object from class Two.

- **Because we've declared the variable objectRef as an object of type One, we can only access the members from class One.**

```
public class Main {

    public static void main(String args[]) {

        One objectRef = new Two();

        objectRef.x = 2; // Valid.
        objectRef.y = 3; // Valid.
        objectRef.z = 4; // INVALID.

    }

}
```

# POLYMORPHISM

- So what is Polymorphism?
- Polymorphism is the ability for an object to take many forms.
- In this case, the object of type Two is also an object of type One.

```
class One {

    int x;
    int y;

}
```

```
class Two extends One {

    int z;

}
```

| | | |
|---|---|---|
| Two | extends | One |
| One | extended by | Two |
| Two | inherits | One |
| One | inherited by | Two |
| Two | is a | One |