INHERITANCE

Fundamentals

- **Inheritance** allows a class to acquire all the attributes (fields) and behaviors (methods) of another class.
- A class that is derived from another class is called a **subclass**. It is also known as **derived class** or **child class**.
- The class from which the subclass is derived is called a **superclass**. It is also known as **base class** or **parent class**.
  Example:
```
//Person.java
public class Person {
  private String name;
  public void setName(String name) {
    this.name = name;
  }
  public String getName() {
    return name;
  }
}
//Student.java
public class Student extends Person {
  public static void main(String[] args) {
    System.out.println(getName());
 }
}
```
  Explanation: `Student` is the subclass while `Person` is the superclass. Assuming that both classes are in the same package, an *import* statement is not required in `Student`.java to access the `Person` class.
- Java supports multiple levels of inheritance where a class may extend another class, which in turn extends another class.
  Example:
```
public class Person { }
public class Student extends Person { }
public class FirstYear extends Student { }
```
- Java supports single inheritance where a class may inherit from only one (1) direct parent class. Extending multiple classes (multiple inheritance) are not allowed.
- In Java, all classes inherit from the **`Object`** class. It is the only class that does not have any parent classes.
```
//Class declaration 1
public class Student { }
//Class declaration 2
public class Student extends java.lang.Object { }
```
  Explanation: Both class declarations mean the same. A class that does not extend another class invisibly extends the `Object` class.
- To prevent a class from being extended, mark the class with the `final` modifier.
- Even though the superclass is public, its subclass cannot access its private fields.
  Example:
```
//Person.java
public class Person {
  private String name;
  public void setName(String name) {
    this.name = name;
  }
  public String getName() {
    return name;
  }
}
//Student.java
public class Student extends Person {
  public static void main(String[] args) {
    System.out.println(name); //does not compile
 }
}
```
  Explanation: The `name` variable in the `Person` class is marked as `private` and therefore not accessible from the subclass `Student`.

Rules in Defining Constructors

1. The first statement of every constructor is either a call to another constructor within the class using **`this()`** or a call to a constructor in the direct parent class using **`super()`**.

Example:

```
//Person.java
public class Person {
  private String name;
  public Person(String name) {
    this.name = "New student;
  }
}
//Student.java
public class Student extends Person {
  public Student(String name) { //1st constructor
    super(name);
  }
  public Student() { //2nd constructor
    this("No name yet.");
  }
}
```

Explanation: In the `Student` class, the first statement of the first constructor is a call to the constructor of the `Person` class. It also includes another no-argument constructor that calls the other constructor within the class using `this("No name yet.")`.

2. The **`super()`** command may only be used as the first statement of the constructor.

   Example:

```
public Person() {
  System.out.println("Person object created.");
  super(); //does not compile
}
```

   Explanation: The `super()` command is the second statement and therefore will lead to a compilation error.

3. If a **`super()`** call is not declared in a constructor, Java will insert a no-argument `super()` as the first statement of the constructor.

   Example:

```
//Empty Constructor
public Student() { }
//Equivalent to the empty constructor
public Student() {
  super();
}
```

   Explanation: The Java compiler invisibly adds a no-argument constructor `super()` if the first statement is not a call to the parent constructor.

4. If the parent class does not have a no-argument constructor and the child class does not define any constructors, the compiler will throw an error. This is because the child class has an invisible no-argument `super()` that tries to call the constructor of the parent class.

   Example:

```
//Person.java
public class Person {
  private String name;
  public Person(String name) {
    this.name = name;
  }
}
//Student.java
public class Student extends Person { }
//above does not compile
```

   Explanation: The compiler adds a no-argument constructor and a no-argument `super()`. However, the `Person` class does not have a no-argument constructor, leading to a compilation error.

5. If the parent class does not have a no-argument constructor, the compiler requires an explicit call to a parent constructor in each child constructor.

   Example:

```
//Person.java
public class Person {
  private String name;
  public Person(String name) {
    this.name = name;
  }
}
```

```
//Student.java
public class Student extends Person {
  public Student(){
    super("Nika Pena");
  }
}
```

Explanation: The constructor in `Person` can only be called if the `Student` specifies a `String` argument in the `super()` call of its constructor such as `super("Nika Pena");`

Calling Constructors and Overriding Methods

- The parent constructor is always executed before the child constructor.

Example:

```
//Person.java
public class Person {
  public Person() {
    System.out.println("Person");
  }
}
//Student.java
public class Student extends Person {
  public Student() {
    System.out.println("Student");
  }
}
//FirstYear.java
public class FirstYear extends Student {
  public static void main(String[] args) {
    FirstYear fy = new FirstYear();
  }
}
```

Output:
```
Person
Student
```

Explanation: The compiler first inserts invisible `super()` commands as the first statements of both `Person` and `Student` constructors. Next, the compiler inserts an invisible default no-argument constructor in the `FirstYear` class with an invisible `super()` as the first statement of the constructor. The code will execute with the parent constructors called first, producing the output below.

Output:
```
Person
Student
```

- You can define a new version of an existing method in a child class that makes use of the definition in the parent class. This ability is called **method overriding**.
- To override a method, declare a new method with the same name, parameter list, and return type as the method in the parent class. The method in the subclass must be at least as accessible as the method in the parent class.

Example:

```
//FirstClass
public class FirstClass {
  protected String getMessage() {
    return "Method";
  }
}
//SecondClass
public class SecondClass extends FirstClass {
  public String getMessage() {
    return super.getMessage() + " Overriding";
  }
  public static void main(String[] args) {
    System.out.println(new SecondClass().getMessage());
  }
}
```

Explanation: The `getMessage()` method of `SecondClass` overrides `getMessage()` of `FirstClass`. The `super` keyword can be used to access the method of the parent class being overridden.

Output:
```
Method Overriding
```

References:

Savitch, W. (2014). *Java: An introduction to problem solving and programming* (7th ed.). New Jersey: Pearson Education, Inc.

Oracle Docs (n.d.). *Citing sources.* Retrieved from https://docs.oracle.com/javase/tutorial/java/javaOO/index.html