



U9L2: Writing Constructors for Subclasses

Constructors are not inherited

Inherited

1. All `public` and `protected` attributes and methods.
2. `default` attributes and methods as long as the subclass is in the same package as the superclass.

Not Inherited

1. Constructors
2. All `private` attributes and methods.
3. `default` attributes and methods if the subclass is in a different package than the superclass.

As we saw in the previous lab, we can still invoke the superclass constructor by calling `super()`, but technically this doesn't count as being *inherited*. **Watch out for this on multiple choice problems.**

Eclipse Error Flag


Suppose we define the following class named `Parent` that we will extend to create a subclass named `Child`:

```
1 public class Parent{
2     protected String name;
3     public Parent(String name){
4         this.name = name;
5     }
6 }
7 /*-----*/
8 public class Child extends Parent{
9     // TODO: Implement Class
10 }
```


Eclipse flags the `Child` class right away with the following error:

```
public class Child extends Parent{
```

```
}
```

 Implicit super constructor Parent() is undefined for default constructor. Must define a no-arg constructor

1 quick fix available:

 [Add constructor 'Child\(String\)'](#)

What does that mean?

Implicit super constructor Parent() is undefined for default constructor. Must define an explicit constructor.



How did we fix this in the lab? By defining an explicit constructor:

```
1 public Child(String name){  
2     super(name);  
3 }
```

What does that mean?

To fully understand why this is necessary and what the error message is saying, we need to understand the following 3 concepts.

1. Default constructors.
2. Why calling the super constructor is necessary.
3. Implicit vs. Explicit super constructor calls.

Default Constructors (Review)

A **default constructor** is a no argument constructor generated by the compiler when no explicit constructors have been provided. For example if we define the following class with out a constructor:

```
1 class Pizza{
2     int num;
3     String str;
4
5     public void printInfo(){
6         System.out.println("num: " + num + ", str: " + str);
7     }
8 }
```

We are still able to instantiate this class thanks to the *default constructor*

```
1 Pizza pizza = new Pizza();
2 pizza.printInfo();
```

No Default Constructor here

If we define at least 1 constructor, the default constructor will **not** be generated. For example if the following constructor were in the `Pizza` class

```
1 public class Pizza{
2     int num;
3     String str;
4
5     public Pizza(int num, String str){
6         this.num = num;
7         this.str = str;
8     }
9
10    public void printInfo(){
11        System.out.println("num: " + num + ", str: " + str);
12    }
13 }
```

Then we couldn't create an object using the no argument constructor `Pizza pizza = new Pizza()` without explicitly defining one.

Why call the super constructor?

When we inherit a class in Java we say the inherited class `extends` the superclass. The `extends` keyword is well-chosen since the subclass should only include the following code:

1. Specific attributes not relevant to the superclass.
2. Specific methods not relevant to the superclass.
3. Methods that need to be customized to work with the subclass. (Overrides)

All the common code defined in the superclass is not written in the subclass. It isn't possible to construct a subclass object without constructing a superclass object first and adding the new features to it.

Implicit vs. Explicit Calls

For this reason, the Java compiler will look to see if you made a call to `super()` on the first line of your constructor(s) (Explicit Super Constructor Call). If you did not, it will attempt to implicitly call `super()` with no arguments before running any of your other code.

Implicit Call: Even super classes include this implicit call. This is because all classes are inherited from the `Object` class.

```
1 public class Parent{
2 public Parent(String name){
3     // super(); <- will call new Object() automatically before setting name
4     this.name = name;
5 }
```

Error Explained

When you do not write a constructor, the compiler attempts to call `super()` when executing the default constructor. This is the source of the error.

```
1 public class Parent{
2     protected String name;
3     public Parent(String name){
4         this.name = name;
5     }
6 }
7 /*-----*/
8 public class Child extends Parent{
9     // TODO: Implement Class
10 }
```

Child has no constructor yet. When this code compiles the following will happen.

1. The compiler generates a default constructor `Child()`
2. This constructor implicitly calls `Parent()` however `Parent()` doesn't exist, only `Parent(String name)`.

Error Explained

1. The compiler generates a default constructor `Child()`
2. This constructor implicitly calls `Parent()` however `Parent()` doesn't exist, only `Parent(String name)`.

In other words:

Implicit super constructor `Parent()` is undefined for default constructor. Must define an explicit constructor.

Demonstrate Implicit Calls

To verify this is happening we can insert `println` statements in the beginning of our constructors to verify the order in which they executed.

Takeaways

- We only need to explicitly call `super()` if we are passing parameters to the super constructor.
- Explicit calls to `super()` must occur on the first line of the constructor.
- Implicit calls to `super()` will not work if a no argument constructor doesn't exist.

Practice Problems