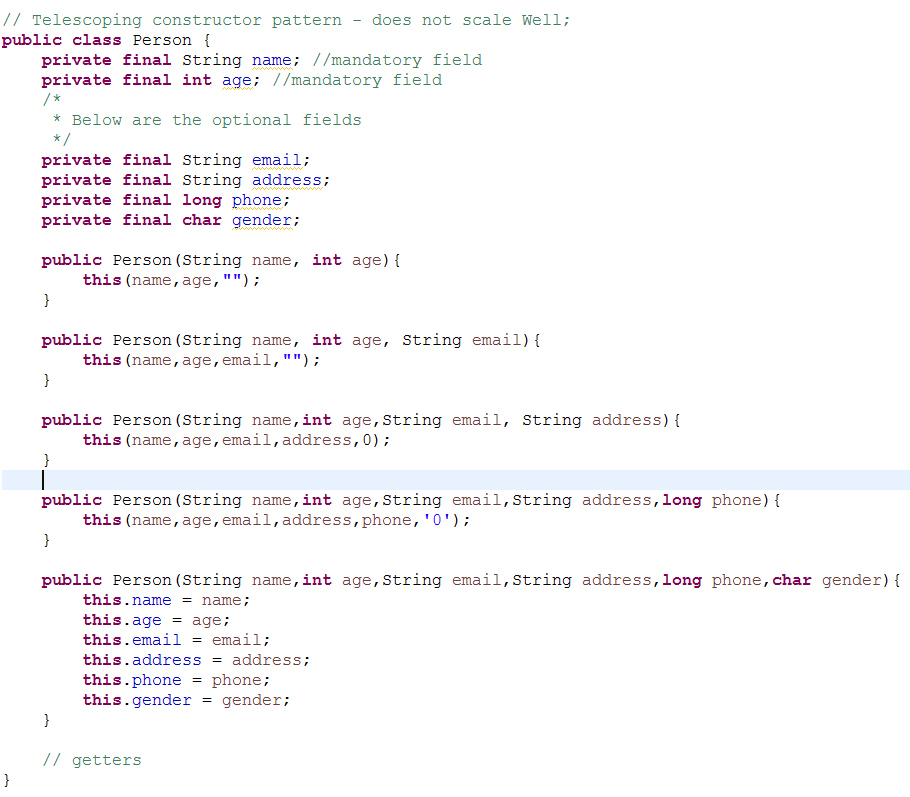
**Builder Design Pattern**

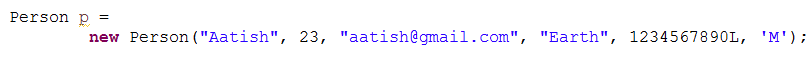
Builder Design Pattern is used when there are many constructor Parameter:

Before Learning Builder Design Patter, let’s first understand *Telescopic constructor Pattern* and *Java Beans Pattern.*

***Telescoping Design Pattern:***

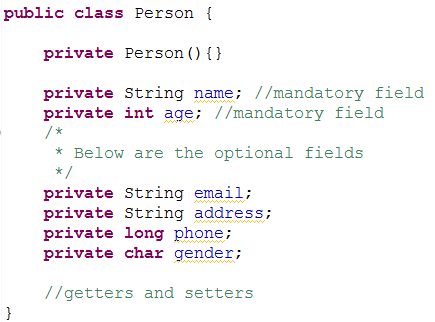


* Programmers have used *Telescopic constructor pattern,* in which you provide a constructor with only the required parameters, another with a single optional parameter, a third with two optional parameters, and so on, providing in a constructor with all the optional parameters.
* When you want an instance, you use the constructor with the shortest parameter list containing all the parameter you want to set:



* This constructor invocation will require many parameters that you won’t want to set, but you’re forced to pass a value for them anyway.
* **In short, the telescoping constructor pattern works, but it is hard to write client code when there are many parameters, and harder still to read it**.
* It is less readable for the users and Long sequences of identically typed parameters can cause subtle bugs. If the client accidentally reverses two such parameters, the compiler won’t complain, but the program will misbehave at runtime.

***Java Beans Pattern***

****

* A second alternative when you are faced with many constructor parameter is the *JavaBeans* pattern, in which you call a parameterless constructor to create the object and then call setter methods to set each required parameter and each optional parameter.
* This pattern has none of the disadvantages of the telescoping constructor pattern.
* Unfortunately, the JavaBeans pattern has serious disadvantages of its own. Because construction is split across multiple calls, **a JavaBean may be in an inconsistent state partway through its construction.** The class does not have the option of enforcing consistency merely by checking the validity of the constructor parameters.
* Attempting to use an object when it’s in an inconsistent state may cause failures that are far removed from the code containing the bug, hence difficult to debug.
* A related disadvantage is that **the JavaBeans pattern make it impossible of making a class immutable,** and requires added effort on the part of the programmer to ensure thread safety.

**Builder Design Pattern**

* Luckily, there is a third alternative that combines the safety of the telescoping constructor pattern with the readability of the JavaBeans pattern.
* It is a form of the *Builder* pattern. Instead of making the desired object directly, the client calls a constructor (or static factory) with all of the required parameters and gets a *builder object.*
* Then the client calls setter-like methods on the builder object to set each optional parameter.
* Finally, the client calls a parameterless *build* method to generate the object, which is immutable.
* The builder is a **static member class** of the class it builds.
* Here How it looks like:

**public** **class** Person {

**private** String name; //mandatory field

**private** **int** age; //mandatory field

**private** String email;

**private** String address;

**private** **long** phone;

**private** **char** gender;

**public** **static** **class** Builder {

//required params

**private** **final** String name;

**private** **final** **int** age;

//optional params

**private** String email = "";

**private** String address = "";

**private** **long** phone = 0;

**private** **char** gender = ' ';

**public** Builder(String name, **int** age){

**this**.name = name;

**this**.age = age;

}

**public** Builder email(String val){

email = val;**return** **this**;

}

**public** Builder address(String val){

address = val;**return** **this**;

}

**public** Builder phone(**long** val){

phone = val;**return** **this**;

}

**public** Builder gender(**char** val){

gender = val;

**return** **this**;

}

**public** Person build(){

**return** **new** Person(**this**);

}

}// end of Builder Class

**private** Person(Builder builder){

name = builder.name;

age = builder.age;

email = builder.email;

address = builder.address;

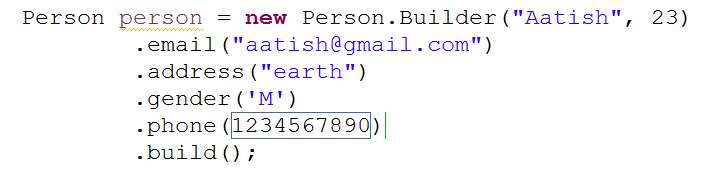
phone = builder.phone;

gender = builder.gender;

}

}

* Note that the Person is immutable, and that all parameter default values are in a single location.
* The builder’s setter methods return the builder itself so invocations can be chained. Here’s how the client code looks:



* The client code is easy to write and more importantly, to read.

**In Summary, the Builder pattern is a good choice when designing classes whose constructors or static factories would have more than handful of parameters, especially if most of those parameters are optional.**

**Client code is much easier to read and write with builders that with traditional telescoping constructor pattern and builder are much safer than JavaBeans.**