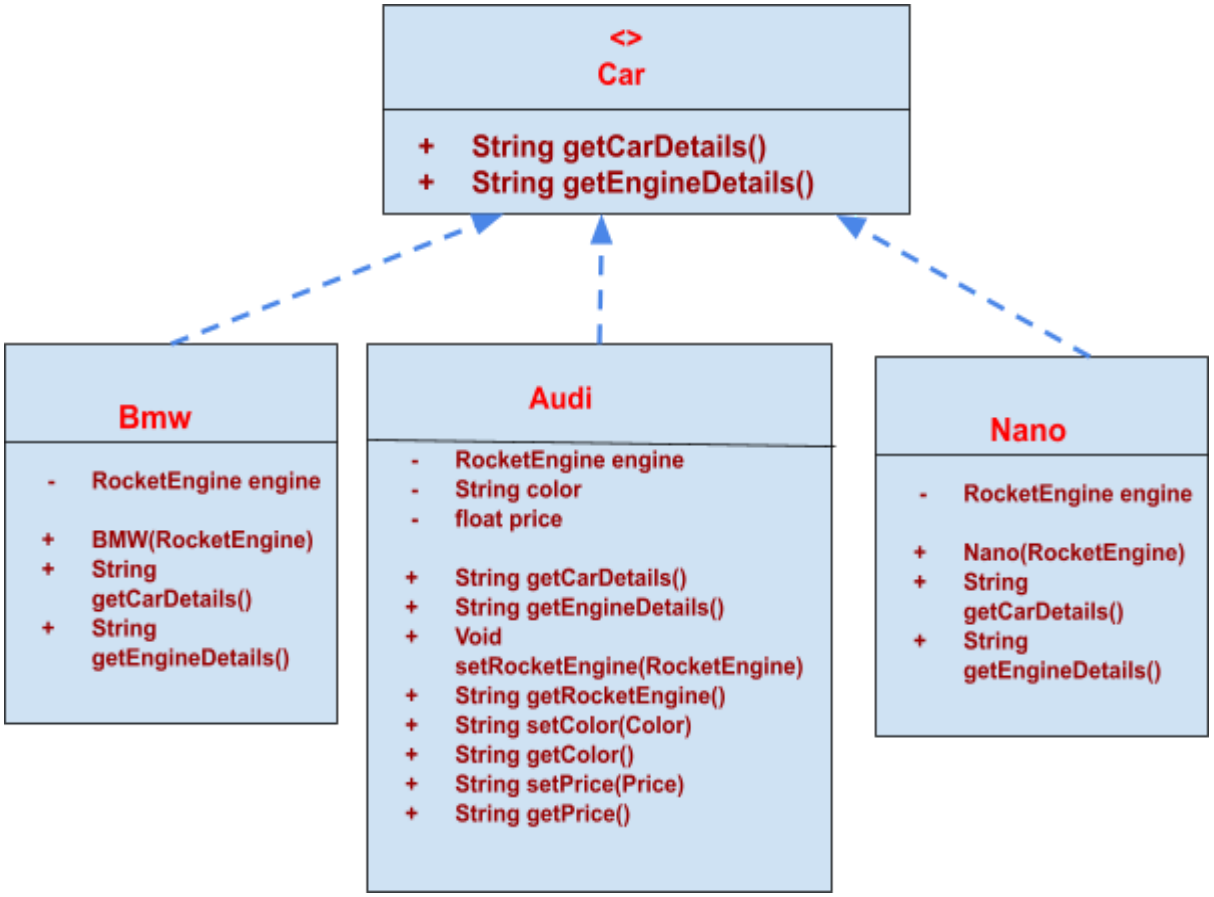
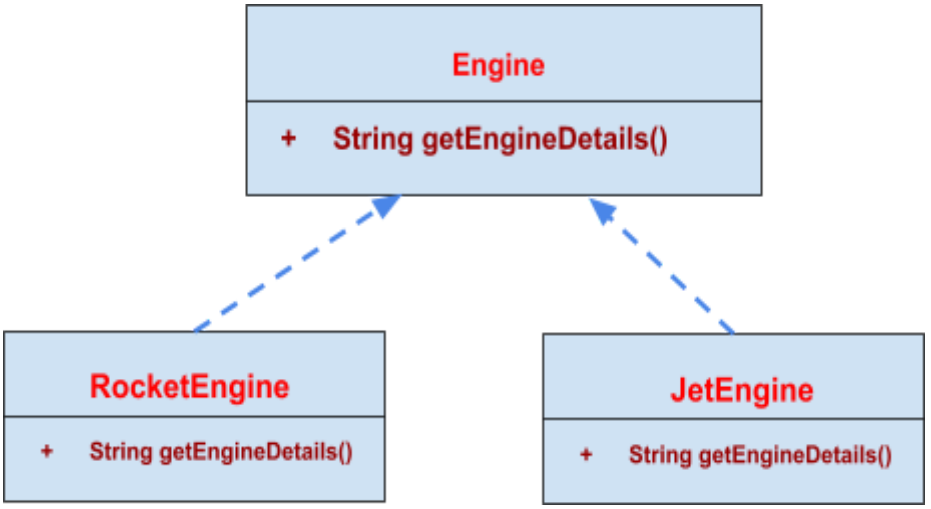
Constructor Injection for ambiguous situation

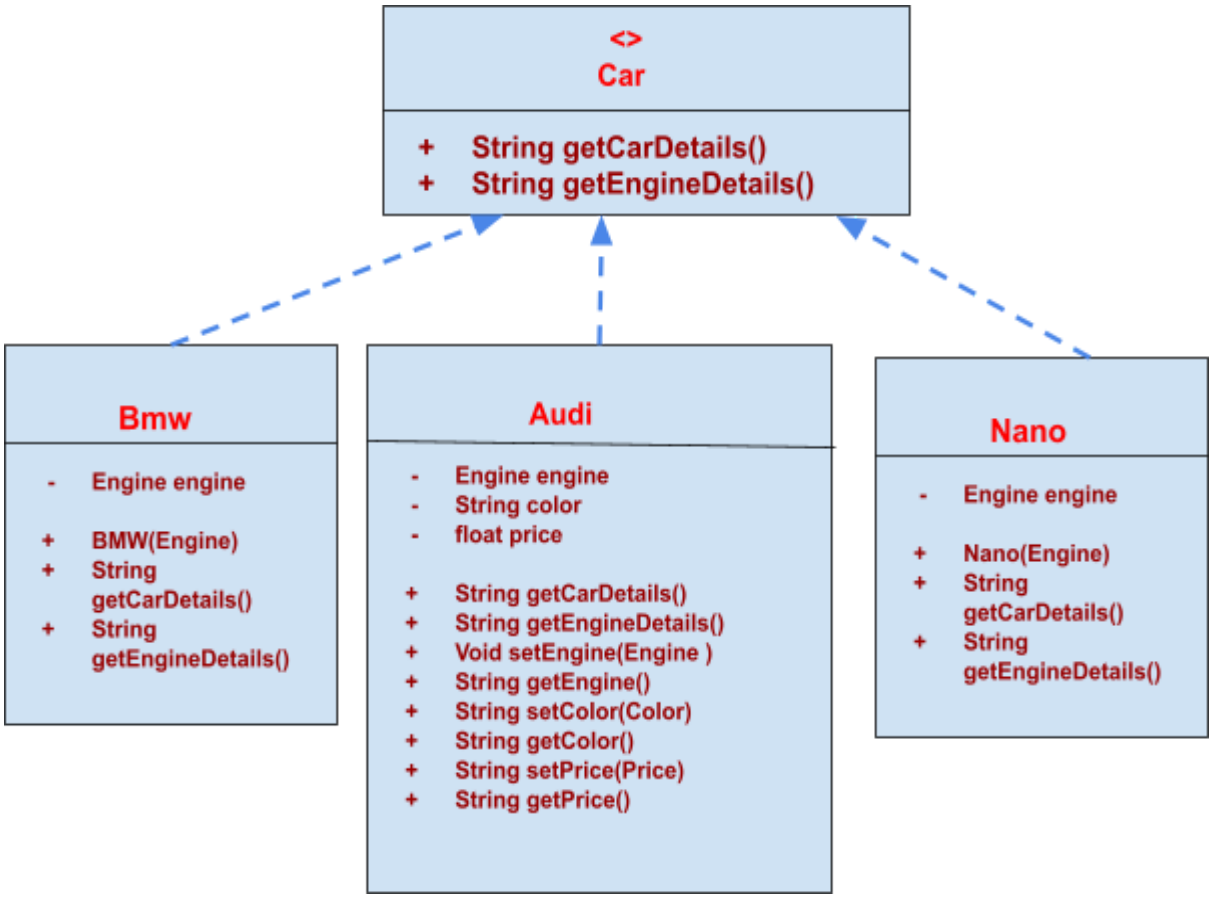
Scenario 1:

Now let's have two classes JetEngine and RocketEngine which implements engine interface.





Here if you observe there are two classes RocketEngine and JetEngine which implements Engine interface.
Now Bmw, Audi and Nano might use RocketEngine or JetEngine so you need to write generalised so instead of these two we need to mention Engine.



Now let’s try implementing and check how it works

Step 1: Create a class JetEngine() and implement getEngineDetails()

```
package com.tapacad.spring;

import org.springframework.stereotype.Component;

@Component("jetEngine")
public class JetEngine implements Engine{
		@Override
		public String getEngineDetail() {
			return "Jet Engine is amazing";
		}

}
```

Step 2: Now in Bmw, Audi and Nano change it to Engine instead of RocketEngine

Bmw.java

```
package com.tapacad.spring;

import org.springframework.stereotype.Component;

@Component("bmw")
public class BMW implements Car
{
	private Engine engine;

	public BMW(Engine engine) {
		super();
		this.engine = engine;
	}

	@Override
```

```
	public String getCarDetails() {
		return "BMW";
	}

	@Override
	public String getEngineDetail() {
		return engine.getEngineDetail();
	}

}
```

Nano.java

```
package com.tapacad.spring;
import org.springframework.stereotype.Component;

@Component("nano")
public class Nano implements Car
{
	private Engine engine;

	public Nano(Engine engine) {
		super();
		this.engine = engine;
	}

	@Override
	public String getCarDetails() {
		return "Nano";
	}

	@Override
	public String getEngineDetail() {
		return engine.getEngineDetail();
	}
}
```

Audi.java

```
package com.tapacad.spring;

import org.springframework.stereotype.Component;

@Component("audi")
public class Audi implements Car
{
	private Engine engine;
	private String colour;
	private int price;

	@Override
	public String getCarDetails() {
		return "Audi";
	}

	@Override
	public String getEngineDetail() {
		return engine.getEngineDetail();
	}

	public Engine getRocketEngine() {
		return engine;
	}

	public void setRocketEngine(Engine engine) {
		this.engine = engine;
	}

	public String getColour() {
		return colour;
	}

	public void setColour(String colour) {
```

```
            this.colour = colour;
        }

        public int getPrice() {
            return price;
        }

        public void setPrice(int price) {
            this.price = price;
        }

        void myStartUp(){
            System.out.println("Bean Created");
        }

        void myClosing(){
            System.out.println("Bean Destroyed");
        }

        void fun1(){
            System.out.println("Default Bean Created");
        }

        void fun2(){
            System.out.println("Default Bean Destroyed");
        }


}
```

MyApp8.java

```
package com.tapacad.spring;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
public class MyApp8 {
	public static void main(String[] args) {
//		Load Application context
		ClassPathXmlApplicationContext context =
				new
ClassPathXmlApplicationContext("annotateApplicationContext.xml");

//		Get bean
		Car car = context.getBean("bmw", Car.class);

//		call getCarDetails()
		System.out.println(car.getCarDetails());
		System.out.println(car.getEngineDetail());

//		close context
		context.close();

	}
}
```

Output:

```
WARNING: Exception encountered during context initialization -
cancelling refresh attempt:
org.springframework.beans.factory.UnsatisfiedDependencyException
```

Here while creating the bmw bean we have got an exception. Let's see why this exception occurred and how to fix this step by step.

```
@Component("bmw")
public class BMW implements Car
{
	private Engine engine;

	public BMW(Engine engine) {
		super();
		this.engine = engine;
	}
}
```

Here if you observe from the above code, in the constructor it is mentioned to create the bean foe engine but there are two types of engine **JetEngine and RocketEngine**. Now this is the ambiguous situation where it is confused which bean it has to create so you got the exception.

Let’s see now to overcome this exception.

@Autowired on constructor

It allows Spring to resolve and inject collaborating beans into our bean.
We'll see that an instance of Engine is injected by Spring as an argument to the Engine constructor:

```
@Component("bmw")
public class BMW implements Car
{
	private Engine engine;

	@Autowired
	public BMW(Engine engine) {
		super();
		this.engine = engine;
	}
}
```

By default, Spring resolves **@Autowired** entries by type. **If more than one bean of the same type is available in the container as there are two types of engine now(JetEngine, RocketEngine), the framework will throw a fatal exception**.
To resolve this conflict, we need to tell Spring explicitly which bean we want to inject using **@Qualifier**
Now there are two types of bean as shown below

RocketEngine.java

```
package com.tapacad.spring;
import org.springframework.stereotype.Component;

@Component("rocketEngine")
public class RocketEngine implements Engine
{
	@Override
	public String getEngineDetail() {
		return "Rocket Engine gives good performance";
	}
}
```

JetEngine.java

```
package com.tapacad.spring;
import org.springframework.stereotype.Component;

@Component("jetEngine")
public class JetEngine implements Engine{
		@Override
		public String getEngineDetail() {
			return "Jet Engine is amazing";
		}
}
```

Now let’s try to inject JetEngine bean into the BMW and Nano class

We can avoid this by narrowing the implementation using a *@Qualifier* annotation:

BMW.java

```
@Component("bmw")
public class BMW implements Car
{
	private Engine engine;

	@Autowired
	public BMW(@Qualifier("jetEngine") Engine engine) {
		super();
		this.engine = engine;
	}
.
.
}
```

Nano.java

```
@Component("nano")
public class Nano implements Car
{
	private Engine engine;

	@Autowired
	public Nano(@Qualifier("jetEngine") Engine engine) {
		super();
		this.engine = engine;
	}
.
.
}
```

When there are multiple beans of the same type, it's a good idea to **use *@Qualifier* to avoid ambiguity.**

Please note that the value of the **@Qualifier annotation** matches with the name declared in the **@Component annotation** of our JetEngine implementation.

MyApp8.java

```
package com.tapacad.spring;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
public class MyApp8 {
	public static void main(String[] args) {
//		Load Application context
		ClassPathXmlApplicationContext context =
				new
ClassPathXmlApplicationContext("annotateApplicationContext.xml");

//		Get bean
		Car car = context.getBean("bmw", Car.class);

//		call getCarDetails()
		System.out.println(car.getCarDetails());
		System.out.println(car.getEngineDetail());

//		close context
		context.close();

	}
}
```

Output:

```
BMW
Jet Engine is amazing
```

As you can see from the above output the **ambiguous bean injection** is resolved by making use of **@Autowired and @Qualifier.**