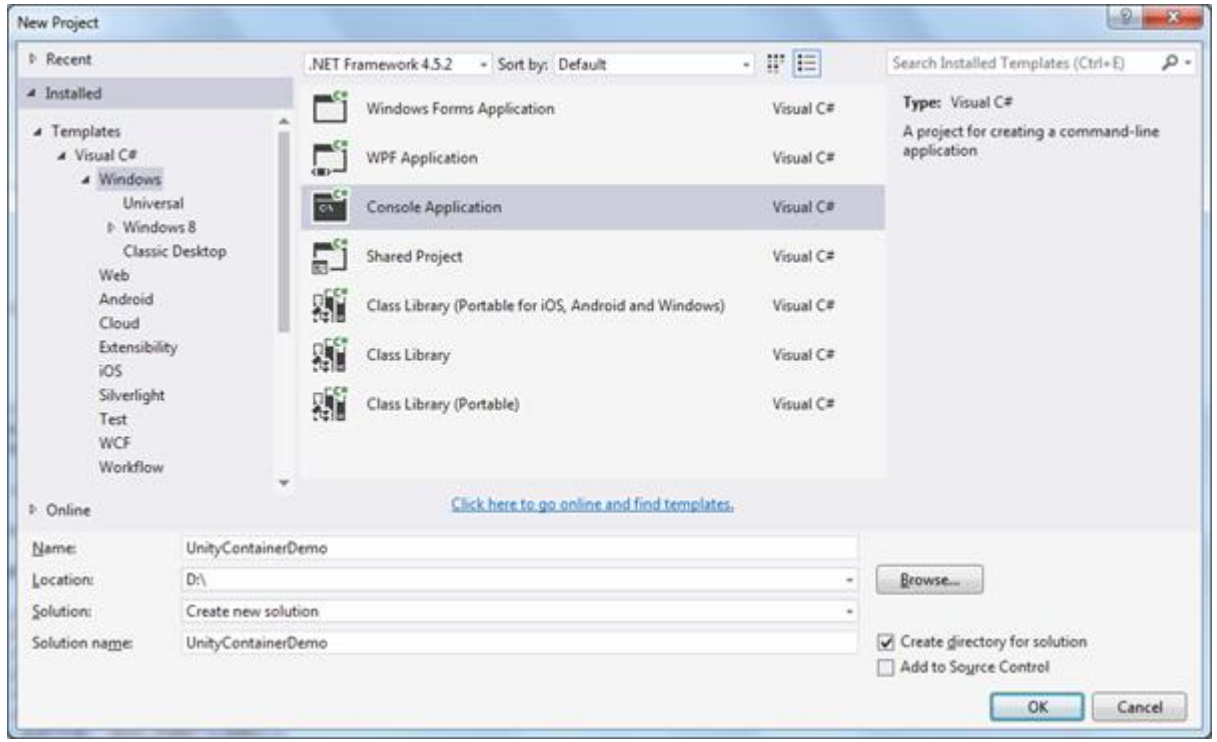
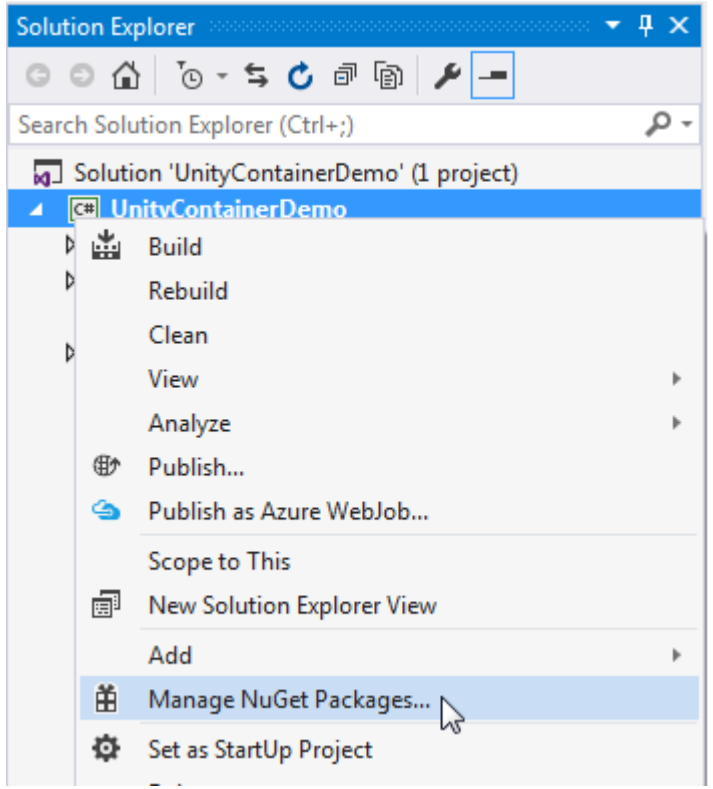
Използване на UNITY контейнер за Dependency injection

Инсталиране на UNITY пакет в средата на Visual Studio

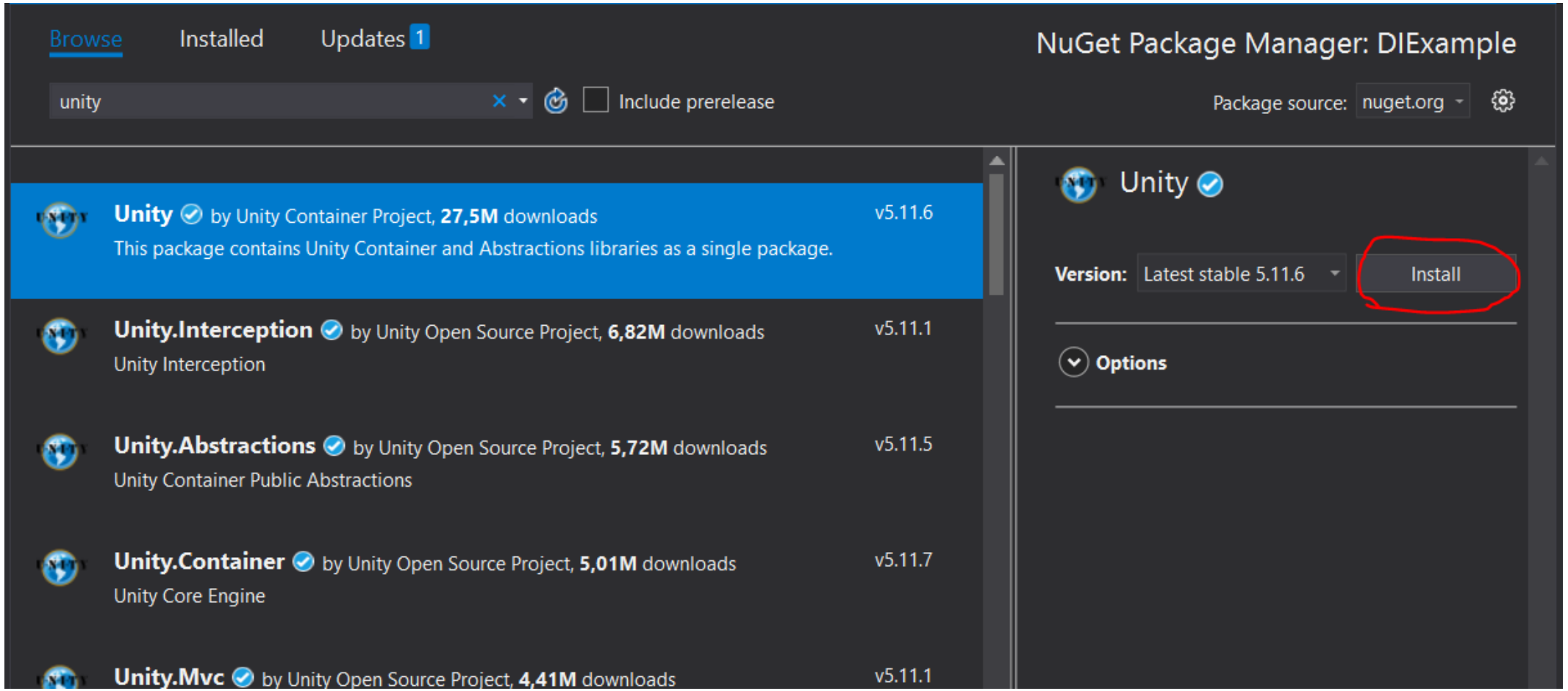
1. Създаваме нов проект



2. С десен клик върху проекта в Solution Explorer избираме Manage NuGet Packages



Инсталиране на UNITY пакет в средата на Visual Studio



Нека имаме следните класове

```
public interface ICar
{
    int Run();
}

public class BMW : ICar
{
    private int _miles = 0;

    public int Run()
    {
        return ++_miles;
    }
}

public class Ford : ICar
{
    private int _miles = 0;

    public int Run()
    {
        return ++_miles;
    }
}
```

```
public class Audi : ICar
    {
        private int _miles = 0;
        public int Run()
        {
            return ++_miles;
        }

    }
    public class Driver
    {
        private ICar _car = null;
        public Driver(ICar car)
        {
            _car = car;
        }

        public void RunCar()
        {
            Console.WriteLine("Running {0} - {1} mile ",
_car.GetType().Name, _car.Run());
        }
    }
```

Употреба на класовете в Main метода

```
class Program
   {
       static void Main(string[] args)
       {
           Driver driver = new Driver(new BMW());
           driver.RunCar();
           Console.ReadKey();

       }
   }
```

Output:
Running BMW - 1 mile

Създаване на UNITY контейнер

```
using Unity;


static void Main(string[] args)
        {
            IUnityContainer container = new UnityContainer();
            //or
            var container = new UnityContainer();

        }
```

Регистриране на Type-mapping

```
static void Main(string[] args)
        {
            IUnityContainer container = new UnityContainer();
            container.RegisterType<ICar, BMW>();
        }
```

- Извикването на метода container.RegisterType <ICar, BMW> () изисква Unity да създаде обект от клас BMW и да го инжектира чрез конструктор, когато инжектираме обект, имплементиращ ICar.
- https://msdn.microsoft.com/en-us/library/microsoft.practices.unity.iunitycontainer.registertype.aspx предоставя допълнителна информация за метода RegisterType

Resolve метод

- Unity създава обект от определения клас и автоматично инжектира зависимостите, използвайки метода Resolve(). Регистрирали сме BMW класа, като имплементиращ ICar. Сега можем да създадем обект от класа Driver, използвайки Unity, без да използваме ключова дума new, както е показано по-долу.

```
static void Main(string[] args)
        {
            IUnityContainer container = new UnityContainer();
            container.RegisterType<ICar, BMW>();// Map ICar with BMW

            //Resolves dependencies and returns the Driver object
            Driver drv = container.Resolve<Driver>();
            drv.RunCar();
        }
```

Output:
Running BMW - 1 mile

Resolve метод – две последователни извиквания

```
var container = new UnityContainer();
container.RegisterType<ICar, BMW>();

Driver driver1 = container.Resolve<Driver>();
driver1.RunCar();

Driver driver2 = container.Resolve<Driver>();
driver2.RunCar();
```

Output:
Running BMW - 1 mile
Running BMW - 1 mile

Множествена регистрация

```
IUnityContainer container = new UnityContainer();
container.RegisterType<ICar, BMW>();
container.RegisterType<ICar, Audi>();

Driver driver = container.Resolve<Driver>();
driver.RunCar();
```

Output:
Running Audi - 1 Mile

Регистриране на именован тип

```
IUnityContainer container = new UnityContainer();
container.RegisterType<ICar, BMW>();
container.RegisterType<ICar, Audi>("LuxuryCar");

ICar bmw = container.Resolve<ICar>();  // returns the BMW object
ICar audi = container.Resolve<ICar>("LuxuryCar"); // returns the Audi object
```

Както се вижда по-горе, сме регистрирали ICar както с BMW класа, така и с класа Audi. Особеното е, че сме именовали с "LuxuryCar" регистрацията на Audi класа. Сега методът Resolve () ще върне обект на Audi, ако посочим името, с което е регистраран.

Използване на InjectionConstructor

```
 static void Main(string[] args)
        {
            var container = new UnityContainer();
            container.RegisterType<ICar, BMW>();
            container.RegisterType<ICar, Audi>("LuxuryCar");

            // Registers Driver type
            container.RegisterType<Driver>("LuxuryCarDriver",
                                new
InjectionConstructor(container.Resolve<ICar>("LuxuryCar")));
            Driver driver1 = container.Resolve<Driver>();// injects BMW
            driver1.RunCar();
            Driver driver2 = container.Resolve<Driver>("LuxuryCarDriver");// injects Audi
            driver2.RunCar();
        }
```

Output:
Running BMW - 1 Mile
Running Audi - 1 Mile

В горния пример регистрирахме класа Driver с името "LuxuryCarDriver" и създадохме обект InjectionConstructor. Операторът new InjectionConstructor (container.Resolve <ICar> ("LuxuryCar")) инжектира класа Driver, като създава обект на класа Audi, тъй като container.Resolve ("LuxuryCar") връща обект на Audi класа. Сега можем да използваме container.Resolve <Driver> ("LuxuryCarDriver"), за да получим обект от Driver с инжектиран Audi обект, въпреки инжектирането по подразбиране на ICar с BMW класа

Регистриране на инстанция

```
var container = new UnityContainer();
ICar audi = new Audi();
container.RegisterInstance<ICar>(audi);

Driver driver1 = container.Resolve<Driver>();
driver1.RunCar();
driver1.RunCar();

Driver driver2 = container.Resolve<Driver>();
driver2.RunCar();
```

Output:
Running Audi - 1 Mile
Running Audi - 2 Mile
Running Audi - 3 Mile

Инжектиране с конструктор с няколко параметъра

```
public interface ICarKey
{    }

public class BMWKey : ICarKey
{ }

public class AudiKey : ICarKey
{ }

public class FordKey : ICarKey
{ }
```

```
public class Driver
    {
        private ICar _car = null;
        private ICarKey _key = null;
        public Driver(ICar car, ICarKey key)
        {
            _car = car;
            _key = key;
        }
        public void RunCar()
        {
            Console.WriteLine("Running {0} with
{1} - {2} mile ", _car.GetType().Name,
_key.GetType().Name, _car.Run());
        }
    }
```

Инжектиране с конструктор с няколко параметъра

```
var container = new UnityContainer();

container.RegisterType<ICar, Audi>();
container.RegisterType<ICarKey, AudiKey>();

var driver = container.Resolve<Driver>();
driver.RunCar();
```

Output:
Running Audi with AudiKey - 1 mile

Наличие на повече от един конструктор, употреба на InjectionConstructor

```
public class Driver
    {
        private ICar _car = null;

        [InjectionConstructor]
        public Driver(ICar car)
        {
            _car = car;
        }

        public Driver(string name)
        {
        }

        public void RunCar()
        {
            Console.WriteLine("Running {0} - {1} mile ", _car.GetType().Name,
_car.Run());
        }
    }
```

Наличие на повече от един конструктор, динамична реализация без атрибут InjectionConstructor

```
static void Main(string[] args)
{
    var container = new UnityContainer();

    container.RegisterType<Driver>(new InjectionConstructor(new Ford()));

            //or

    container.RegisterType<ICar, Ford>();
    container.RegisterType<Driver>(new InjectionConstructor(container.Resolve<ICar>()));
 }
```

Подаване на допълнителни примитивни типове при извикване на конструктор

```
public class Driver
    {
        private ICar _car = null;
        private string _name = string.Empty;
        public Driver(ICar car, string driverName)
        {
            _car = car;
            _name = driverName;
        }
        public void RunCar()
        {
            Console.WriteLine("{0} is running {1} - {2} mile ",
                              _name, _car.GetType().Name, _car.Run());
        }
    }
```

Подаване на допълнителни примитивни типове при извикване на конструктор

```
static void Main(string[] args)
        {
            var container = new UnityContainer();

            container.RegisterType<Driver>(new InjectionConstructor(new
object[] { new Audi(), "Steve" }));

            var driver = container.Resolve<Driver>(); // Injects Audi and
Steve
            driver.RunCar();
        }
```

Output:
Steve is running Audi - 1 mile

Инжектиране на свойство с атрибута [Dependency]

```
public class Driver
    {

        public Driver()
        {
        }

        [Dependency]
        public ICar Car { get; set; }

        public void RunCar()
        {
            Console.WriteLine("Running {0} -
{1} mile ", this.Car.GetType().Name,
this.Car.Run());
        }
    }
```

```
 static void Main(string[] args)
        {
            var container = new
UnityContainer();
            container.RegisterType<ICar, BMW>();

            var driver =
container.Resolve<Driver>();
            driver.RunCar();
        }
```

Output:
Running BMW - 1 mile

Инжектиране на свойство с именоване на атрибута [Dependency]

```
public class Driver
    {

        public Driver()
        {
        }

        [Dependency(LuxuryCar)]
        public ICar Car { get; set; }

        public void RunCar()
        {
            Console.WriteLine("Running {0} -
{1} mile ", this.Car.GetType().Name,
this.Car.Run());
        }
    }
```

```
 static void Main(string[] args)
        {
            var container = new UnityContainer();
            container.RegisterType<ICar, BMW>();
            container.RegisterType<ICar,
Audi>("LuxuryCar");

            var driver =
container.Resolve<Driver>();
            driver.RunCar();



}
```

Output:
Running Audi - 1 mile

Инжектиране на свойство без атрибут, динамична реализация

```
public class Driver
    {

        public Driver()
        {
        }

        public ICar Car { get; set; }

        public void RunCar()
        {
            Console.WriteLine("Running {0} -
{1} mile ", this.Car.GetType().Name,
this.Car.Run());
        }
    }
```

```
 static void Main(string[] args)
        {
            var container = new UnityContainer();

            //run-time configuration
            container.RegisterType<Driver>(new
InjectionProperty("Car", new BMW()));

            var driver =
container.Resolve<Driver>();
            driver.RunCar();
        }
```

Output:
Running BMW - 1 Mile

Инжектиране чрез метод и атрибута InjectionMethod

```
public class Driver
    {
        private ICar _car = null;

        public Driver()
        {
        }
        [InjectionMethod]
        public void UseCar(ICar car)
        {
            _car = car;
        }

        public void RunCar()
        {
            Console.WriteLine("Running {0} -
{1} mile ", _car.GetType().Name, _car.Run());
        }
    }
```

```
static void Main(string[] args)
        {
            var container = new
UnityContainer();
            container.RegisterType<ICar,
BMW>();

            var driver =
container.Resolve<Driver>();
            driver.RunCar();
        }
```

Output:
Running BMW - 1 mile

Инжектиране чрез метод без атрибута InjectionMethod (динамично)

```
public class Driver
    {
        private ICar _car = null;

        public Driver()
        {
        }
       public void UseCar(ICar car)
        {
            _car = car;
        }

        public void RunCar()
        {
            Console.WriteLine("Running {0} - {1} mile ", _car.GetType().Name,
_car.Run());
        }
    }
```

Инжектиране чрез метод без атрибута InjectionMethod (динамично)

```
 static void Main(string[] args)
        {
            var container = new UnityContainer();

            //run-time configuration
            container.RegisterType<Driver>(new InjectionMethod("UseCar", new
Audi()));

            //to specify multiple parameters values
            container.RegisterType<Driver>(new InjectionMethod("UseCar", new
object[] { new Audi() }));

            var driver = container.Resolve<Driver>();
            driver.RunCar();
        }
```

Output:
Running Audi - 1 Mile

Време на живот на инжектираните обекти, мениджъри, управляващи времето на живот

```
var container = new UnityContainer().RegisterType<ICar, BMW>(new TransientLifetimeManager());
```

Lifetime Manager	Description
TransientLifetimeManager	Creates a new object of the requested type every time you call the Resolve or ResolveAll method.
ContainerControlledLifetimeManager	Creates a singleton object first time you call the Resolve or ResolveAll method and then returns the same object on subsequent Resolve or ResolveAll calls.
HierarchicalLifetimeManager	Same as the ContainerControlledLifetimeManager, the only difference is that the child container can create its own singleton object. The parent and child containers do not share the same singleton object.
PerResolveLifetimeManager	Similar to the TransientLifetimeManager, but it reuses the same object of registered type in the recursive object graph.
PerThreadLifetimeManager	Creates a singleton object per thread. It returns different objects from the container on different threads.
ExternallyControlledLifetimeManager	It maintains only a weak reference of the objects it creates when you call the Resolve or ResolveAll method. It does not maintain the lifetime of the strong objects it creates, and allows you or the garbage collector to control the lifetime of the objects. It enables you to create your own custom lifetime manager.

TransientLifetimeManager – по подразбиране, създава всеки път нов обект за инжектиране

```
var container = new UnityContainer().RegisterType<ICar, BMW>();

var driver1 = container.Resolve<Driver>();
driver1.RunCar();

var driver2 = container.Resolve<Driver>();
driver2.RunCar();
```

Output:
Running BMW - 1 Mile
Running BMW - 1 Mile

ContainerControlledLifetimeManager – създава по една инстанция на инжектирания обект

```
var container = new UnityContainer()
                .RegisterType<ICar, BMW>(new
ContainerControlledLifetimeManager());

var driver1 = container.Resolve<Driver>();
driver1.RunCar();

var driver2 = container.Resolve<Driver>();
driver2.RunCar();
```

Output:
Running BMW - 1 mile
Running BMW - 2 mile

HierarchicalLifetimeManager – подобен на ContainerControlledLifetimeManager, но всеки Child контейнер, поддържа собствена инстанция на инжектирания обект

```
var container = new UnityContainer()
                .RegisterType<ICar, BMW>(new
HierarchicalLifetimeManager());

var childContainer = container.CreateChildContainer();

var driver1 = container.Resolve<Driver>();
driver1.RunCar();
var driver2 = container.Resolve<Driver>();
driver2.RunCar();
var driver3 = childContainer.Resolve<Driver>();
driver3.RunCar();
var driver4 = childContainer.Resolve<Driver>();
driver4.RunCar();
```

Output:
Running BMW - 1 mile
Running BMW - 2 mile
Running BMW - 1 Mile
Running BMW - 2 Mile