Defining Classes

Classes, Fields, Constructors, Properties, Methods

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://about.softuni.bg/

Have a Question?

Software University

sli.do

#csharp-advanced

Table of Contents

Defining Simple Classes

Creating Class for an ADT

Defining Simple Classes

- Class is a **concrete implementation** of an ADT
- Classes provide **structure** for **describing** and **creating** objects

Keyword

Class name

```
class Dice
{
    …
}
```

Class body

Naming Classes

- Name classes with nouns using **PascalCasing**
- Use **descriptive nouns**
- **Avoid abbreviations** (except widely known, e.g. URL, HTTP, etc.)

```
class Dice { … }
class BankAccount { … }
```

```
class TPMF { … }
class bankaccount { … }
class intcalc { … }
```

Class Members

- **Members** are **declared** in the class and they have certain accessibility, which can be specified
- They can be:
  - Fields
  - Properties
  - Methods
  - Etc.

```
class Dice
{
  int sides;
  string Sides { get; }
  void Roll() { … }
}
```

Field

Property

Method

Creating an Object

- A class can have **many instances** (objects)

```
class Program
{
  public static void Main()
  {
    Dice diceD6 = new Dice();
    Dice diceD8 = new Dice();
  }
}
```

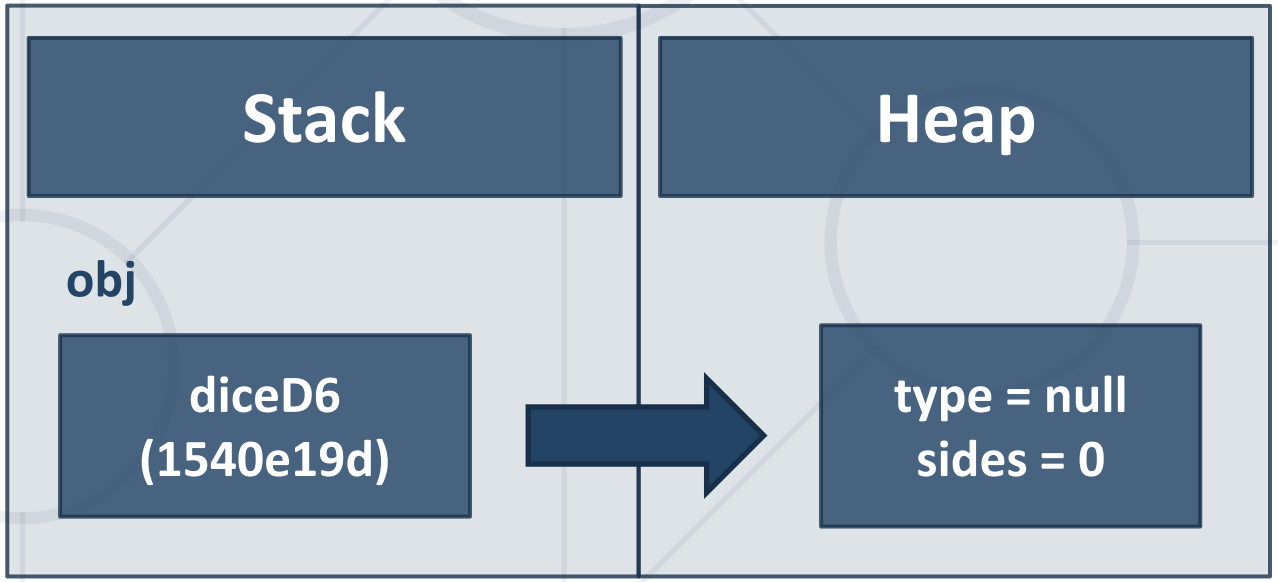
Use the **new** keyword

A variable holds an object **reference**

Object Reference

- Declaring a variable creates a **reference** in the stack
- The **new** keyword allocates memory on the heap

```
Dice diceD6 = new Dice();
```



Classes vs. Objects

- Classes provide **structure** for describing and creating objects
- An **object** is a **single instance of a class**

Dice ADT → Dice (Class) → D6 Dice (Object)

Classes vs. Objects

- **Classes** provide structure for creating objects

```
class
Dice
```

Class name

```
type: string
sides: int
```

Class data

```
Roll(…)
```

Class actions

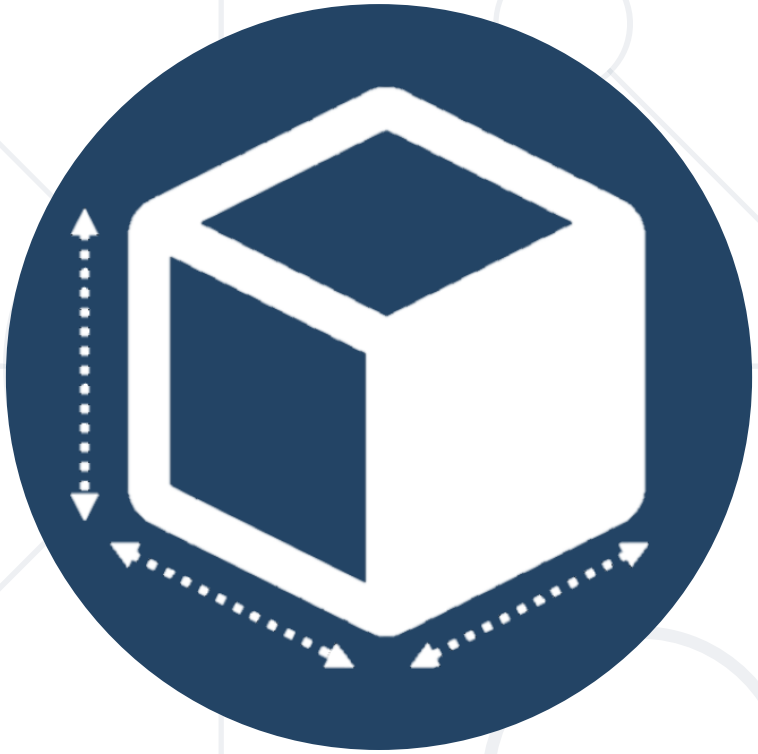
- An **object** is a single instance of a class

```
object
diceD6
```

Object name

```
type = "six sided"
sides = 6
```

Object data



Fields and Properties

Storing Data Inside a Class

Fields and Modifiers

- Class **fields** have type and name
- Access modifiers (like **public** / **private**) define accessibility

Class modifier

Fields should always be private

Fields can be of any type

```
public class Dice
{
  private string type;
  private int sides;
  private int[] rollFrequency;
  private Person owner;
  public void Roll () { … }
}
```

Properties

- Used to create **accessors** and **mutators** (**getters** and **setters**)

```
public class Dice
{
  private int sides;
  public int Sides
  {
    public get { return this.sides; }
    public set { this.sides = value; }
  }
}
```

The field is hidden

The getter provides access to the field

The setter provides field change

Problem: Car

- Create a class **Car**

```
Car
-make:string
-model:string
-year:int
(no actions)
```

```
private string make;
private string model;
private int year;
public string Make
{
  get { return this.make; }
  set { this.make = value; }
}
// TODO: Model and Year Getter & Setter
```

Methods

Defining a Class Behaviour

Methods

- Store **executable code** (an algorithm)

```
public class Dice
{
  private int sides;
  private Random rnd = new Random();
  public int Roll() {
      int rollResult = rnd.Next(1, this.sides + 1);
      return rollResult;
  }
}
```

this points to the current instance

Problem: Car Extension

- Create a class **Car**

Car
-make:string -model:string -year:int -fuelQuantity:double -fuelConsumption:double
+Drive(double distance):void +WhoAmI():string

Check your solution here: https://judge.softuni.bg/Contests/1478/Defining-Classes-Lab

Solution: Car Extension

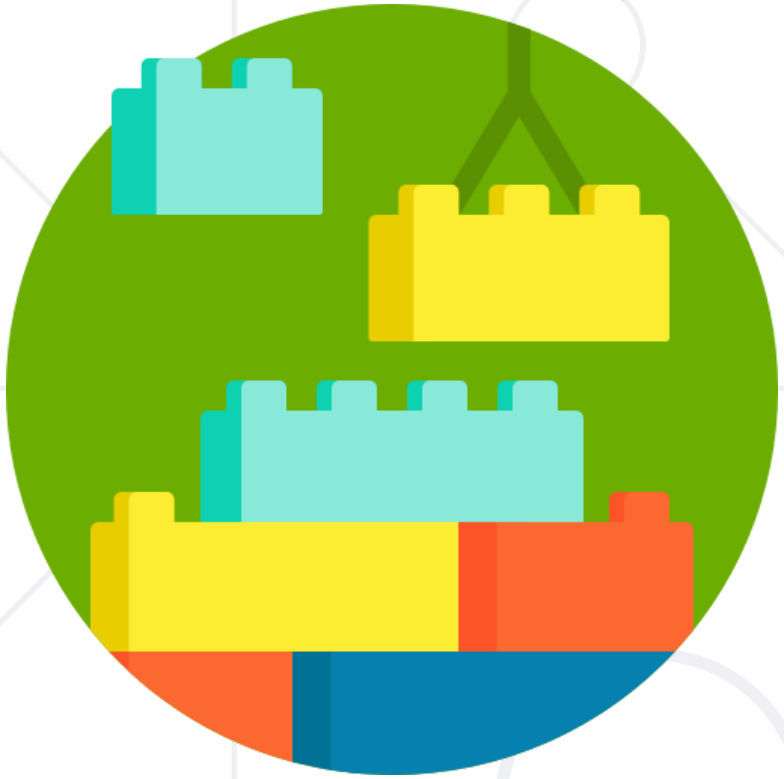
```
// TODO: Get the other fields from previous problem
private double fuelQuantity;
private double fuelConsumption;
// TODO: Get the other properties from previous problem
public double FuelQuantity {
  get { return this.fuelQuantity; }
  set { this.fuelQuantity = value; }}
public double FuelConsumption {
  get { return this.fuelConsumption; }
  set { this.fuelConsumption = value; }}
```

Solution: Car Extension

```
public void Drive(double distance)
{
  bool canContinue = this.FuelQuantity – (distance *
                                  this.FuelConsumption) >= 0;
  if (canContionue)
    this.FuelQuantity -= distance * this.FuelConsumption;
  else
    Console.WriteLine("Not enough fuel to perform this trip!");
}
```

Solution: Car Extension

```
public string WhoAmI()
{
    StringBuilder sb = new StringBuilder();
    sb.AppendLine($"Make: {this.Make}");
    sb.AppendLine($"Model: {this.Model}");
    sb.AppendLine($"Year: {this.Year}");
    sb.Append($"Fuel: {this.FuelQuantity:F2}L");
    return sb.ToString();
}
```



Constructors

Object Initialization

Constructors

- When a constructor is invoked, it creates an instance of its class and usually initializes its members
- Classes in C# are instantiated with the **keyword new**

```
public class Dice
{
  public Dice() { }
}
```

```
public class StartUp
{
  static void Main()
  {
    var dice = new Dice();
  }
}
```

Object Initial State

- Constructors **set object's initial state**

```
public class Dice
{
  int sides;
  int[] rollFrequency;
  public Dice(int sides) {
    this.sides = sides;
    this.rollFrequency = new int[sides];
  }
}
```

Always ensure **correct state**

Multiple Constructors

- You can have multiple constructors in the same class

```
public class Dice
{
  private int sides;
  public Dice() { }
  public Dice(int sides)
  {
    this.sides = sides;
  }
}
```

Constructor **without** parameters

Constructor **with** parameters

Constructor Chaining

- Constructors can call each other

```
public class Person {
  private string name;
  private int age;
  public Person() {
    this.age = 18;
  }
  public Person(string name) : this()
  {
    this.name = name;
  }
}
```

Calls default constructor

Problem: Car Constructors

- Extend the previous problem and **create 3 constructors**
- Default values are:
  - Make - VW
  - Model - Golf
  - Year - 2025
  - FuelQuantity = 200
  - FuelConsumption = 10

Car
+Car() +Car(string make, string model, int year) +Car(string make, string model, int year, double fuelQuantity, double fuelConsumption)

Check your solution here: https://judge.softuni.org/Contests/Practice/Index/1478#2

Solution: Car Constructors

```
public Car() {
  this.Make = "VW";
  this.Model = "Golf";
  this.Year = 2025;
  this.FuelQuantity = 200;
  this.FuelConsumption = 10;}
  public Car(string make, string model, int year) : this()
{
      this.Make = make;
      this.Model = model;
      this.Year = year;}
```

Solution: Car Constructors

```
public Car(string make, string model, int year,
double fuelQuantity, double fuelConsumption)
    : this(make, model, year)
{
    this.FuelQuantity = fuelQuantity;
    this.FuelConsumption = fuelConsumption;
}
```

Problem: Car Engine and Tires

- Create the two classes and extend the Car class

Engine
-horsePower:int -cubicCapacity:double
+Engine(int horsePower, double cubicCapacity)

Tire
-year:int -pressure:double
+Tire(int year, double pressure)

Car
+Car(string make, string model, int year, double fuelQuantity, double fuelConsumption, Engine engine, Tire[] tires)

Check your solution here: https://judge.softuni.org/Contests/Practice/Index/1478#3

Solution: Car Engine and Tires

```
private int horsePower;
private double cubicCapacity;
public Engine(int horsePower, double cubicCapacity) {
    this.HorsePower = horsePower;
    this.CubicCapacity = cubicCapacity; }
public int HorsePower {
    get { return this.horsePower; }
    set { this.horsePower = value; }}
public double CubicCapacity {
    get { return this.cubicCapacity; }
    set { this.cubicCapacity = value; }}
```

Solution: Car Engine and Tires

```
private int year;
private double pressure;
public Tire(int year, double pressure) {
    this.Year = year;
    this.Pressure = pressure; }
public int Year {
    get { return this.year; }
    set { this.year = value; }}
public double Pressure {
    get { return this.pressure; }
    set { this.pressure = value; }}
```

Solution: Car Engine and Tires

```
public Car(string make, string model, int year,
double fuelQuantity, double fuelConsumption, Engine engine,
Tire[] tires)
    : this(make, model, year, fuelQuantity, fuelConsumption)
{
    this.Engine = engine;
    this.Tires = tires;
}
```

Enumerations

Syntax and Usage

Enumerations

- Represent a numeric value from a fixed set as a text
- We can use them to pass **arguments** to **methods** without making code confusing

```
enum Day { Mon, Tue, Wed, Thu, Fri, Sat, Sun }
```

```
GetDailySchedule(0)
```

➡

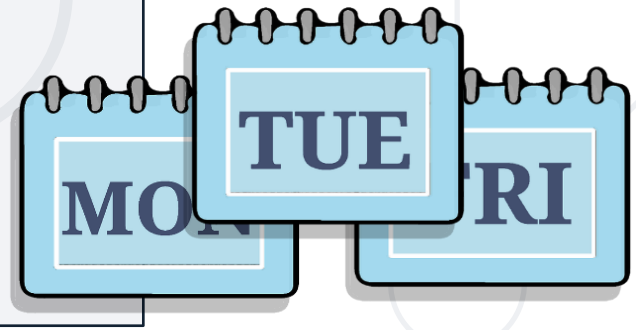
```
GetDailySchedule(Day.Mon)
```

- By default **enums** start at 0
- Every next value is incremented by 1

Enumerations

- We can **customize** enum **values**

```
enum Day {
   Mon = 1,
   Tue,   // 2
   Wed,   // 3
   Thu,   // 4
   Fri,   // 5
   Sat,   // 6
   Sun    // 7
}
```




```
enum CoffeeSize
{
   Small = 100,
   Normal = 150,
   Double = 300
}
```

Static Classes

Static Class Members

Static Class

- A static class is declared by the **static** keyword
- It **cannot** be **instantiated**
- You **cannot declare** variables from its **type**
- You access its **members** by using the **its name**

```
double roundedNumber = Math.Round(num);
int absoluteValue = Math.Abs(num);
int pi = Math.PI;
```

Static Members

- Both **static** and **non-static** classes can contain **static** members:
  - Methods, fields, properties, etc.
- A **static member** is **callable** on a class even when no instance of the class has been created
- Accessed by the **class'** name, not the **instance** name
- Only **one copy** of a static member **exists**, regardless of how many **instances** of the class are created

Static Members

- Static methods can be overloaded but not overridden
- A **const field** is essentially **static** in its **behavior** and it belongs to the **type**, **not** the **instance**
- Static members are initialized **before** the static member is **accessed** for the **first time** and **before** the static **constructor**

```
Bus.Drive();
int wheels = Human.NumberOfWheels;
```

Example: Static Members

```
public class Engine
{
  public static void Run() {
    Console.WriteLine("This is a static method"); }
}
```

```
public static void Main() {
  Engine.Run();
}
// Output: This is a static method
```

Namespaces

Definition and Usage

Namespaces

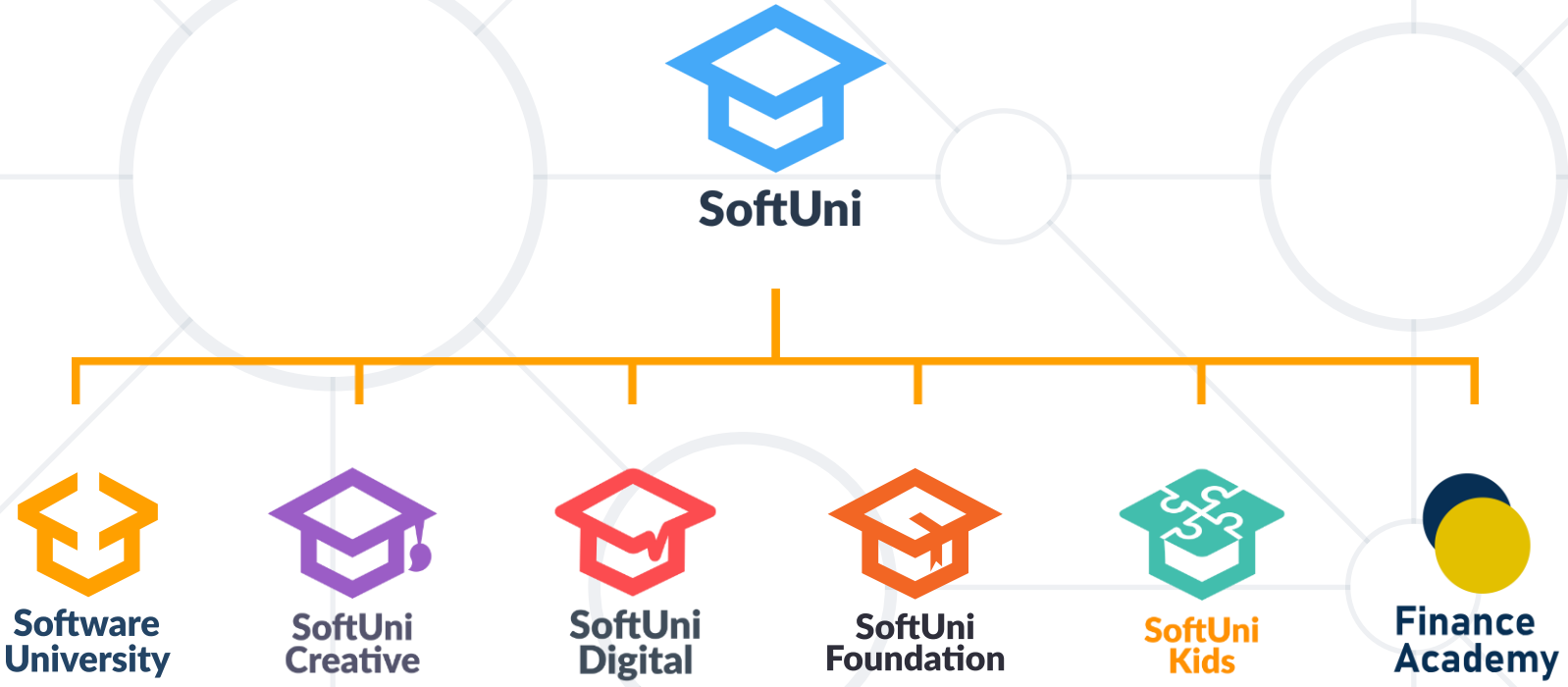
- Used to organize classes
- The **using** keyword allows us not to write their names
- Declaring your own namespaces can help you control the scope of class and method names

```
System.Console.WriteLine("Hello world!");
var list = new
  System.Collections.Generic.List<int>();
```

Summary

- Classes define **structure** for objects
- Objects are **instances of a class**
- Classes define **fields**, **methods**, **constructors** and other members
- Constructors:
  - **Invoked** when creating **new instances**
  - **Initialize** the **object's state**

Questions?



SoftUni Diamond Partners

Software University

SmartIT

Coca-Cola HBC Bulgaria

SOFTWARE GROUP

DXC TECHNOLOGY

createX

Postbank
Решения за твоето утре

SUPER HOSTING .BG

DRAFT KINGS

VIVACOM

INDEAVR
Serving the high achievers

AMBITIONED

PHAR VISION

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, about.softuni.bg
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg

License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – https://about.softuni.bg/
- © Software University – https://softuni.bg