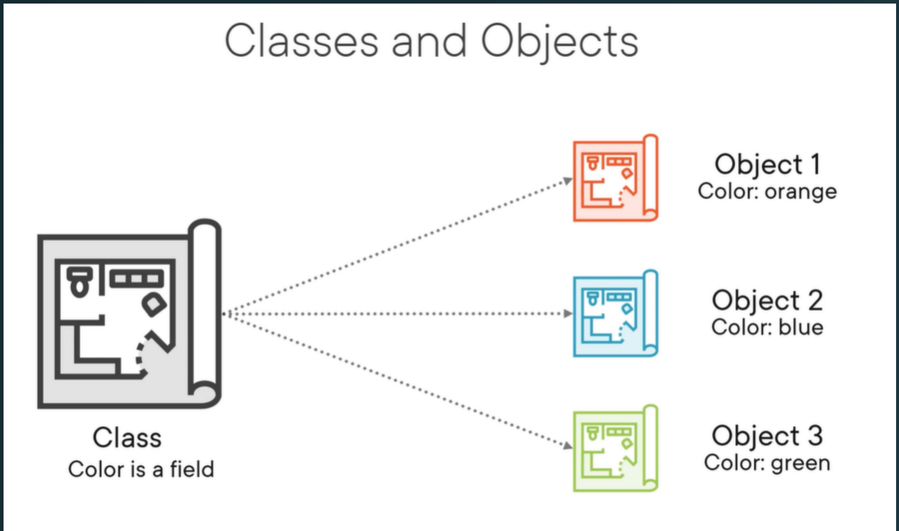
Types Inference

In C#, implicit types typically refer to the use of the var keyword when declaring variables. The var keyword allows the compiler to infer the type of a variable based on the assigned value. This is known as ”implicitly typed variables.”

Type inference

```
var population = 66_000_000; // 66 million in UK
var weight = 1.88; // in kilograms
var price = 4.99M; // in pounds sterling
var fruit = "Apples"; // strings use double-quotes
var letter = 'Z'; // chars use single-quotes
var happy = true; // Booleans have value of true or
false
```

What is a Class?



Is a blueprint of an object it defines data and functionality to work on its data.

It is used to represent concepts if the real world.

Uses the class keyword for its creation.

It is a reference type

An object is a instance of a class.

Building block of OOP

Declaring and using Classes

```
namespace Packt.Shared
{
  public class BankAccount
  {
    public string AccountName; // instance member
    public decimal Balance; // instance member
    public static decimal InterestRate; // shared
member
  }
}
```

Declaring a class

Class member

Static class member

```
var account = new BankAccount();

var newBankAccount = new BankAccount
{
  AccountName = "Personal_2025",
  Balance = 100000M,
};

account.AccountName = "Company_2020";
Console.WriteLine(newBankAccount.AccountName);
```

Instantiating a class using the default constructor

Instantiating a class and asigning values to it's members

Reasigning a value of a class member

Class members

Variables that are associated with the **class.**

Example :

Fields

```
public partial class BankAccount
{
        private string _accountNumber;
        private decimal _balance;
        private List<Transaction> _transactions;
}
```

Operators

Conversions and expression operators supported by the **class**.

Example :

Operators

```
public partial class BankAccount
{
    // Defines an implicit conversion operator from BankAccount
    // to decimal to get the account balance directly.
    public static implicit operator decimal(BankAccount account)
    {
        return account.Balance;
    }
}
```

Constructors

Actions required to initialize **instances of the class** or the **class itself.**

Example :

Constructors

```
public partial class BankAccount
{
        public BankAccount(string accountNumber)
        {
            _accountNumber = accountNumber;
            _balance = 0.0m;
            _transactions = new List<Transaction>();
        }
}
```

Finalizer

Actions done before instances of the **class** are permanently discarded.

Example :

Finalizer

```
public partial class BankAccount
{
        ~BankAccount()
        {
            // Cleanup code here
        }
}
```

This content was retrieved from this Awesome Post

Class members

Properties

Actions associated with reading and writing named properties of the **class.**

Example :

Properties

```
public partial class BankAccount
{
        // Allows reading and writing
        public string AccountNumber
        {
            get { return _accountNumber; }
            set { _accountNumber = value; }
        }

        public decimal Balance
        {
            get { return _balance; }
            private set { _balance = value; }
        }
}
```

Indexers

Actions associated with indexing instances of the **class** like an **array.**

Example :

Indexers

```
public partial class BankAccount
{
        // Allows access to the balance
        // using the square bracket notation.
        public decimal this[int index]
        {
            get => _balance;
            set => _balance = value;
        }
}
```

Events

Notifications that can be generated by the **class.**

Example :

Events

```
public partial class BankAccount
{
    public event EventHandler BalanceChanged;

    // It Is triggered whenever the balance of the account changes.
    protected virtual void OnBalanceChanged()
    {
        BalanceChanged?.Invoke(this, EventArgs.Empty);
    }
}
```

This content was retrieved from this Awesome Post

Class members

Methods

Actions that can be performed by the **class**.

Example :

Methods

```
public partial class BankAccount
{
    // Adds the specified amount to the account balance
    public void Deposit(decimal amount)
    {
        if (amount <= 0)
            throw new ArgumentException("Amount must be positive.");

        _balance += amount;
        _transactions.Add(new Transaction(amount, "Deposit"));
    }
}
```

Types

Nested types declared by the **class.**

Example :

Types

```
public partial class BankAccount
{
        public class Transaction
        {
            public decimal Amount { get; }
            public DateTime Date { get; }
            public string Type { get; }

            public Transaction(decimal amount, string type)
            {
                Amount = amount;
                Date = DateTime.Now;
                Type = type;
            }
        }
}
```

This content was retrieved from this Awesome Post

Understanding properties

```
// SImplified implementation
public class Product
{
    public string Name { get; set; }
}

// Behind the scenes implementation
public class Product
{
    string _name;

    public string Name
    {
      get
      {
        return _name;
      }
      set
      {
        _name = value;
      }
    }
}
```

How Getters and Setter work behind the scenes

```
public string Name { get; }

public string Name
{
    get { return name; }
}

public string Name { get; private set; }
```

Declaring only setters or getters and encapsulate functionality

Overview of method overloading

```
// Method overload
public void IncreaseDebt()
{
  AmountInDebt++;
}

public void IncreaseDebt(int amount)
{
  // implementation here
}
```

Method overload

```
// Constructor overload
public Product(int id) : this(id, string.Empty)
{ }

public Product(int id, string name)
{
  Id = id;
  Name = name;
}
```

Constructor overload

Reference and value types

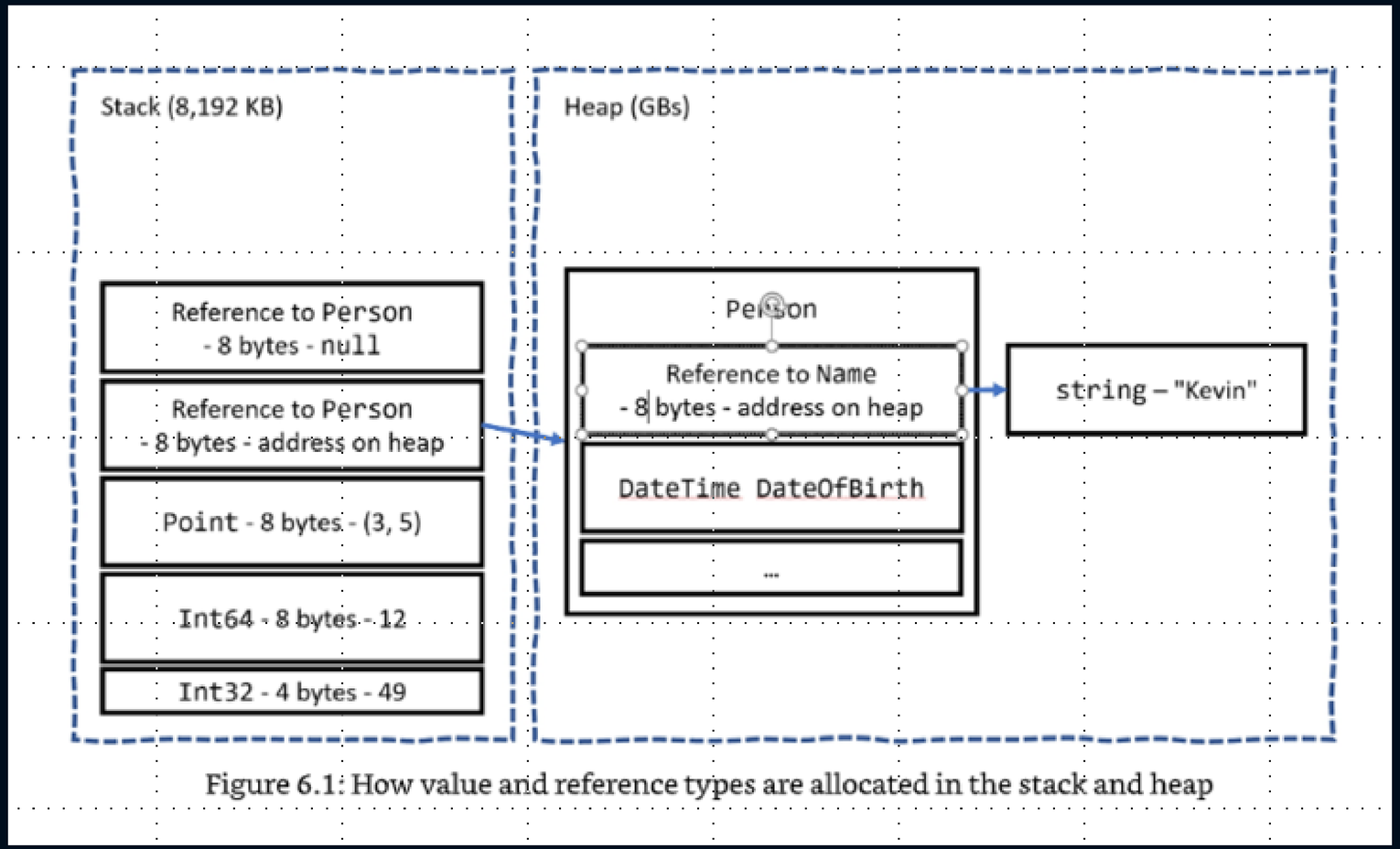


Figure 6.1: How value and reference types are allocated in the stack and heap

Partial classes

Product.cs

```
public partial class Product
{
}
```

Product2.cs

```
public partial class Product
{
}
```

Physically splitting up over multiple files
Classes, structs and interfaces support partial
Combined upon compilation
Most use cases are from code generators

Static members

```
public class Employee
{
  private static double bonusPercentage = 0.15;
  public static void IncreaseBonusPercentage(double newPercentage)
  {
    bonusPercentage = newPercentage;
  }
}

// Using static methods
Employee.IncreaseBonusPercentage(0.20);
```

Static members are useful when data is meant to be stored in a class level
Keep in mind that they are accessible through the class

Demo

We will check the following:

- Create a class
- Constructors
- Create properties and methods
- Static members
- Use class

Tasks

Exercise/ Homework

You are tasked with creating a calculator that can perform addition operations on two values of the same data type. The calculator should be able to handle different data types, such as numeric values and strings. The behavior of the calculator will depend on the data type of the input values.

Your task is to implement the calculator using the following guidelines:

1. If the data types are numeric (e.g., int, float, double), perform addition on the values and return the sum.
2. If the data types are strings, concatenate the values together and return the resulting string.
3. If the data types are different or not supported, throw an exception indicating that the operation is not supported.

```
int result1 = calculator.Add(5, 10);
// Expected output: 15

double result2 = calculator.Add(3.14, 2.56);
// Expected output: 5.7

string result3 = calculator.Add("Hello", "World");
// Expected output: "HelloWorld"
```