Einführung in die OOP mit C#

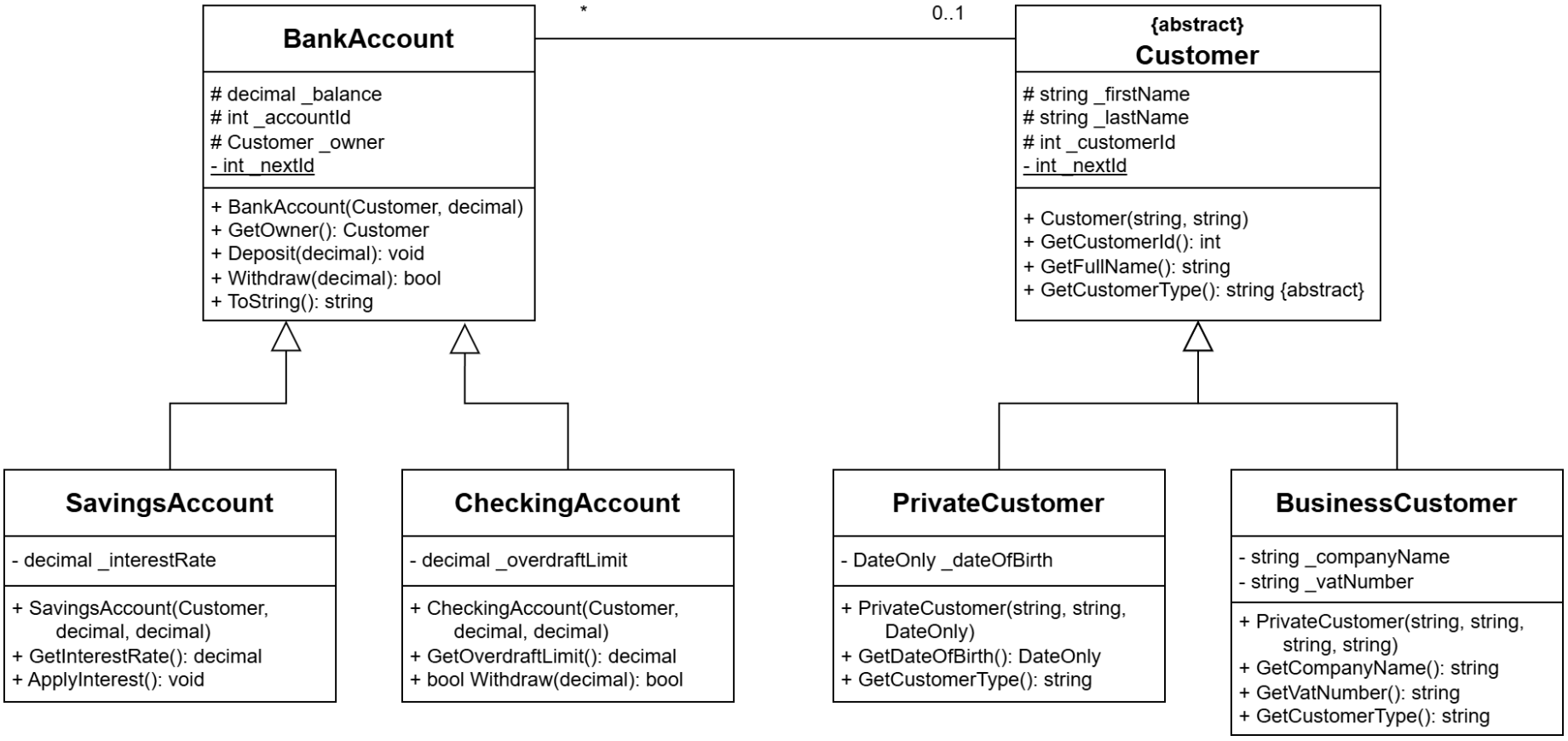
Teil 3: Vererbung

BITLC | Tom Selig
6. Januar 2026

Vererbung

- Die Vererbung (oder auch Generalisierung oder Spezialisierung) ist ein Alleinstellungsmerkmal der OOP.
- Bei der Vererbung kann eine Klasse die Attribute und Operationen einer anderen Klasse erben.
- Code, der in einer Klasse geschrieben wurde, kann so in anderen Klassen wiederverwendet werden.
- Die Klasse, die ihre Member weitergibt, wird Basisklasse genannt, manchmal auch Vater- oder Oberklasse.
- Die Klasse, die diese Member erbt, wird Subklasse genannt, manchmal auch Kind- oder Unterklasse.

Vererbung


BankAccount
decimal _balance
int _accountId
Customer _owner
- int _nextId
+ BankAccount(Customer, decimal)
+ GetOwner(): Customer
+ Deposit(decimal): void
+ Withdraw(decimal): bool
+ ToString(): string
*
0..1
{abstract}
Customer
string _firstName
string _lastName
int _customerId
- int _nextId
+ Customer(string, string)
+ GetCustomerId(): int
+ GetFullName(): string
+ GetCustomerType(): string {abstract}
SavingsAccount
- decimal _interestRate
+ SavingsAccount(Customer, decimal, decimal)
+ GetInterestRate(): decimal
+ ApplyInterest(): void
CheckingAccount
- decimal _overdraftLimit
+ CheckingAccount(Customer, decimal, decimal)
+ GetOverdraftLimit(): decimal
+ bool Withdraw(decimal): bool
PrivateCustomer
- DateOnly _dateOfBirth
+ PrivateCustomer(string, string, DateOnly)
+ GetDateOfBirth(): DateOnly
+ GetCustomerType(): string
BusinessCustomer
- string _companyName
- string _vatNumber
+ PrivateCustomer(string, string, string, string)
+ GetCompanyName(): string
+ GetVatNumber(): string
+ GetCustomerType(): string

Vererbung

```
internal class BankAccount {
    protected int _accountId;
    ...

    public void Deposit(decimal a) {
        ...
    }
    public bool Withdraw(decimal a) {
        ...
    }
}
```

```
internal class SavingsAccount : BankAccount {
    private decimal _interestRate;
    ...

    public decimal GetInterestRate() {
        ...
    }
    public void ApplyInterest() {
        ...
    }
}
```

```
static void Main(string[] args) {
    var acc = new SavingsAccount(...);

    // Das geht natürlich:
    Console.WriteLine(acc.GetInterestRate());

    // Das geht auch:
    acc.Deposit(1_000);
}
```

Vererbung

- In C# ist es nur möglich von einer Klasse zu erben.
- Damit sollen die Probleme, die bei Mehrfachvererbung auftreten, vermieden werden (Diamond-Problem).
- Mehrfachvererbung ist aber in anderen Programmiersprachen grundsätzlich möglich (z. B. C++).
- Die erbende Klasse kann zusätzlich eigene Attribute und Methoden definieren.
- Jede Klasse erbt in C# implizit von der Klasse `Object`. D.h. jedes Objekt kann auf die Member der Klasse `Object` zugreifen.

Vererbung

- Jedes Objekt vom Typ der Subklasse ist immer auch ein Objekt vom Typ der Basisklasse.
- Das bedeutet, dass überall dort wo im Code ein Objekt vom Typ der Basisklasse erwartet wird, auch ein Objekt vom Typ einer Subklasse übergeben werden kann.
- Es kann dann aber nur auf die Member zugegriffen werden, die bereits in der Basisklasse definiert sind.
- Das übergebene Objekt kann aber später wieder auf den Typ der Subklasse gecastet werden.

Vererbung

```
internal class BankAccount {
    protected int _accountId;
    ...

    public void Deposit(decimal a) {
        ...
    }
    public bool Withdraw(decimal a) {
        ...
    }
}
```

```
internal class SavingsAccount : BankAccount {
    private decimal _interestRate;
    ...

    public decimal GetInterestRate() {
        ...
    }
    public void ApplyInterest() {
        ...
    }
}
```

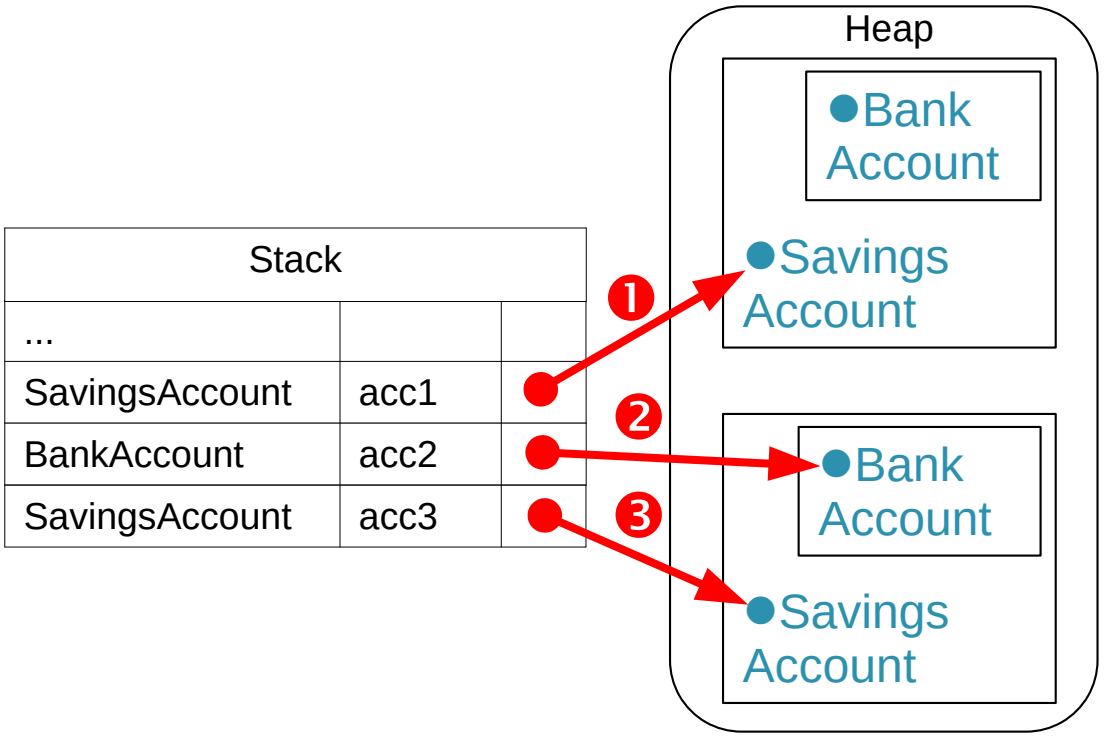
```
static void Main(string[] args) {
    BankAccount acc = new SavingsAccount(...);

    // Das geht weiterhin:
    acc.Deposit(1_000);

    // Das geht aber nicht mehr:
    Console.WriteLine(acc.GetInterestRate());
}
```

Vererbung

```
❶
// Ein Objekt vom Typ SavingsAccount
// wird in einer Variablen vom Typ
// SavingsAccount gespeichert.
SavingsAccount acc1 =
    new SavingsAccount(...);
❷
// Ein Objekt vom Typ SavingsAccount
// wird in einer Variablen vom Typ
// BankAccount gespeichert.
BankAccount acc2 =
    new SavingsAccount(...);
❸
// Das zuletzt erzeugte Objekt wird zu
// einem SavingsAccount gecastet.
// Das geht nur, wenn acc2 tatsächlich
// von diesem Typ ist.
SavingsAccount acc3 =
    (SavingsAccount)acc2;
```



Vererbung

❶ Collection vom Typ BankAccount wird erzeugt.

❷ Die Collection kann Subtypen von BankAccount aufnehmen.

❸ Zugriff ist auf Member aus BankAccount beschränkt.

❹ Elemente der Collection können wieder auf den Subtyp gecastet werden.

❺ Zugriff auf Member der Subtypen ist wieder möglich.

```
static void Main(string[] args) {
    var coll = new BankAccount[2];

    coll[0] = new SavingsAccount();
    coll[1] = new CheckingAccount();

    foreach (BankAccount ba in coll) {
        ba.Deposit(1_000);
    }

    SavingsAccount acc1 =
        (SavingsAccount)coll[0];
    acc1.GetInterestRate();

    CheckingAccount acc2 =
        (CheckingAccount)coll[1];
    acc2.GetOverdraftLimit();
}
```

Zugriffsmodifizierer

- Member, die in der Basisklasse `private` deklariert wurden, sind in der abgeleiteten Klasse nicht sichtbar.
- Member, die in der Basisklasse `public` deklariert wurden, sind in allen Klassen sichtbar.
- Daher gibt es im Rahmen der Vererbung einen weiteren wichtigen Zugriffsmodifizierer:

  `protected`:
  Das Feld oder die Methode ist innerhalb der eigenen Klasse sichtbar, und in Klassen die von dieser Klasse abgeleitet sind.

- Damit sind jetzt alle Zugriffsmodifizierer in C# bekannt.

Zugriffsmodifizierer

Modifizierer	Beschreibung	Klassen	Member
`private` (-)	Der Zugriff ist nur innerhalb des gleichen Typs möglich.	X	X default
`private protected`	Der Zugriff ist nur innerhalb des gleichen Typs oder innerhalb davon abgeleiteter Typen in der gleichen (oder befreundeten) Assembly möglich.	-	X
`protected` (#)	Der Zugriff ist nur innerhalb des gleichen Typs oder innerhalb davon abgeleiteter Typen möglich.	-	X
`internal` (~)	Der Zugriff ist nur innerhalb der gleichen (oder befreundeten) Assembly möglich.	X default	X
`protected internal`	Der Zugriff ist nur innerhalb der gleichen (oder befreundeten) Assembly oder innerhalb davon abgeleiteter Typen außerhalb der Assembly möglich.	-	X
`public` (+)	Der Zugriff ist nicht eingeschränkt.	X	X

Zugriffsmodifizierer

`private`

Type	Assembly: same	Assembly: other
same	✓	
derived	⊗	⊗
other	⊗	⊗

`private protected`

Type	Assembly: same	Assembly: other
same	✓	
derived	✓	⊗
other	⊗	⊗

`protected`

Type	Assembly: same	Assembly: other
same	✓	
derived	✓	✓
other	⊗	⊗

`internal`

Type	Assembly: same	Assembly: other
same	✓	
derived	✓	⊗
other	✓	⊗

`protected internal`

Type	Assembly: same	Assembly: other
same	✓	
derived	✓	✓
other	✓	⊗

`public`

Type	Assembly: same	Assembly: other
same	✓	
derived	✓	✓
other	✓	✓

Up- und Downcasting

- Objekte einer Subklasse können auf den Typ der Basisklasse gecastet werden (**Upcasting**).
- Anschließend können diese wieder auf den ursprünglichen Typ der Subklasse gecastet werden (**Downcasting**).
- Upcasting funktioniert immer, Downcasting nur dann, wenn der Typ passt (sonst Exception).

```
internal class Vehicle {
    // Baseclass
}

internal class Car : Vehicle {
    // Subclass
}

internal class Program {
    static void Main(string[] args) {

        // New object of subclass
        Car car1 = new Car();

        // Upcasting to baseclass
        Vehicle vehicle = car1;

        // Downcasting back to subclass
        Car car2 = (Car)vehicle;
    }
}
```

`is`-Operator

- Der `is`-Operator prüft, ob ein Objekt von einem bestimmten Typ ist.
- Damit kann beim Downcasting eine `InvalidCastException` vermieden werden.
- Der `is`-Operator kann auch für andere Prüfungen eingesetzt werden wie Konstanten- und Variablenmuster.

```
internal class Program {
    static void Main(string[] args) {

        Vehicle[] vs = {
            new Car(),
            new Bike()
        };

        if (vs[0] is Bike) {
            Bike b = (Bike)vs[0];
        }
        else {
            Car c = (Car)vs[0];
        }
    }
}
internal class Vehicle {
}
internal class Car : Vehicle {
}
internal class Bike : Vehicle {
}
```

`as`-Operator

- Der `as`-Operator führt einen Downcast durch, der `null` ergibt wenn er fehlschlägt.
- Damit kann beim Downcasting eine `InvalidCastException` vermieden werden.
- In einem zweiten Schritt muss dann allerdings auf die `null`-Referenz geprüft werden.

```
internal class Program {
    static void Main(string[] args) {
        Vehicle[] vs = {
            new Car(),
            new Bike()
        };

        Bike b = vs[0] as Bike;
        if (b != null) {
            // ...
        }
        else {
            // ...
        }
    }
}
internal class Vehicle {
}
internal class Car : Vehicle {
}
internal class Bike : Vehicle {
}
```

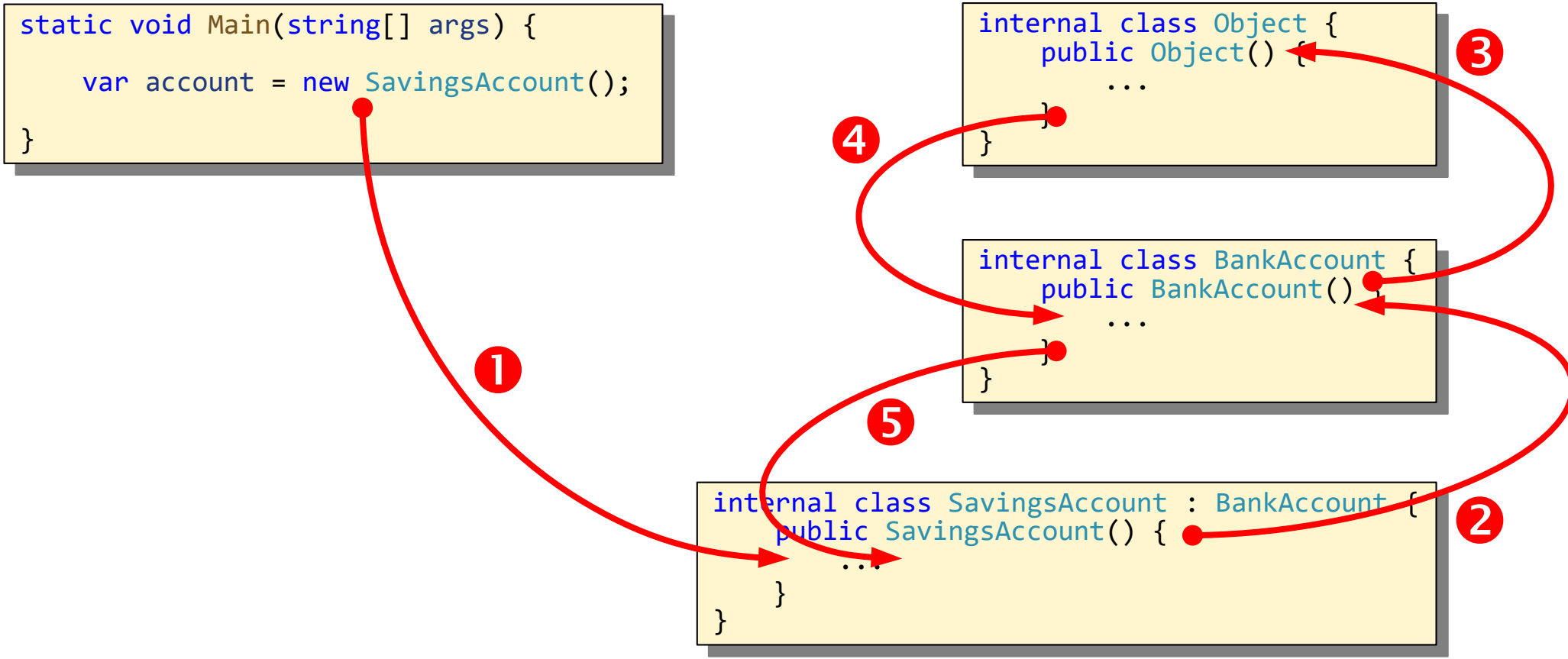
Konstruktoren bei Vererbung

- Konstruktoren werden nicht vererbt. Subklassen besitzen einen eigenen Konstruktor.

- Bei der Instanziierung eines Objekts ruft der Konstruktor zunächst den Konstruktor seiner Basisklasse auf. Dieser ruft dann ebenfalls wieder den Konstruktor seiner Basisklasse auf, bis die Kette beim Konstruktor der Klasse `Object` angekommen ist.

- Der Code der Konstruktor-Methoden wird dann, beginnend bei der Klasse `Object`, bis hinunter zum Konstruktor der ursprünglich aufgerufen wurde, ausgeführt.

Konstruktoren bei Vererbung



Konstruktoren bei Vererbung

- Implizit wird immer der parameterlose Konstruktor der Basisklasse aufgerufen. Wenn kein Konstruktor programmiert wurde, wird der Default-Konstruktor aufgerufen.

- Sollte es keinen parameterlosen Konstruktor in der Basisklasse geben, muss mit dem Schlüsselwort `base` explizit ein Konstruktor aufgerufen werden.

- Auch wenn ein spezieller Konstruktor der Basisklasse aufgerufen werden soll, kann dies mit dem Schlüsselwort `base` erfolgen.

Konstruktoren bei Vererbung

```
static void Main(string[] args) {

    var account =
        new SavingsAccount("John Doe", 1_000);

}
```

```
internal class Object {
    public Object() {
        ...
    }
}
```

```
internal class BankAccount {
    private decimal _balance;

    public BankAccount(decimal bal) {
        _balance = bal;
    }
}
```

```
internal class SavingsAccount : BankAccount {
    private string _name;

    public SavingsAccount(string name, decimal bal) : base(bal) {
        _name = name;
    }
}
```