Module 6

"Introducing Object-oriented Programming"

TEKNOLOGISK
INSTITUT

Agenda

- Introducing Object-Oriented Programming
- First Pillar of OOP: Encapsulation
- Creating Classes and Objects
- Access Modifiers
- Lab 6
- Discussion and Review

Object-Oriented Modeling

- Attempts to realistically reflect (part of) the real-world
- Introduced as a mechanism to ease modeling of simulation problems
- Slowly but steadily adopted into programming languages since 1973

- Abstraction is a crucial technique in this endeavor
  - Focus on important aspects
  - Disregard irrelevant aspects
  - "Selective ignorance"
  - Makes complex things simple!

- Main concepts include *Classes* and *Objects*

The Concept of Classes

- A class in effect classifies abstract or concrete things!

- Philosophers
  - Use artifacts of human classification
  - Classify concepts based upon common characteristics, behavior, and attributes
  - Create descriptions and names of such classifications

- Object-oriented programmers
  - Classify concepts using specific syntactic constructs describing behavior and attributes
  - Define data structures including both data and methods

The Concept of Objects

- Classes are “blueprints” for objects
  - An object is an instance of a class
- Objects have
  - Identity
    - Unique, Distinguishable
  - State
    - Setting, Data
  - Behavior
    - Performing operations modifying the state
- In (sloppy) everyday language the same vocabulary is often used for both the object and the class from which it originates

Examples of Classes and Objects





TEKNOLOGISK INSTITUT

Structs Vs. Classes

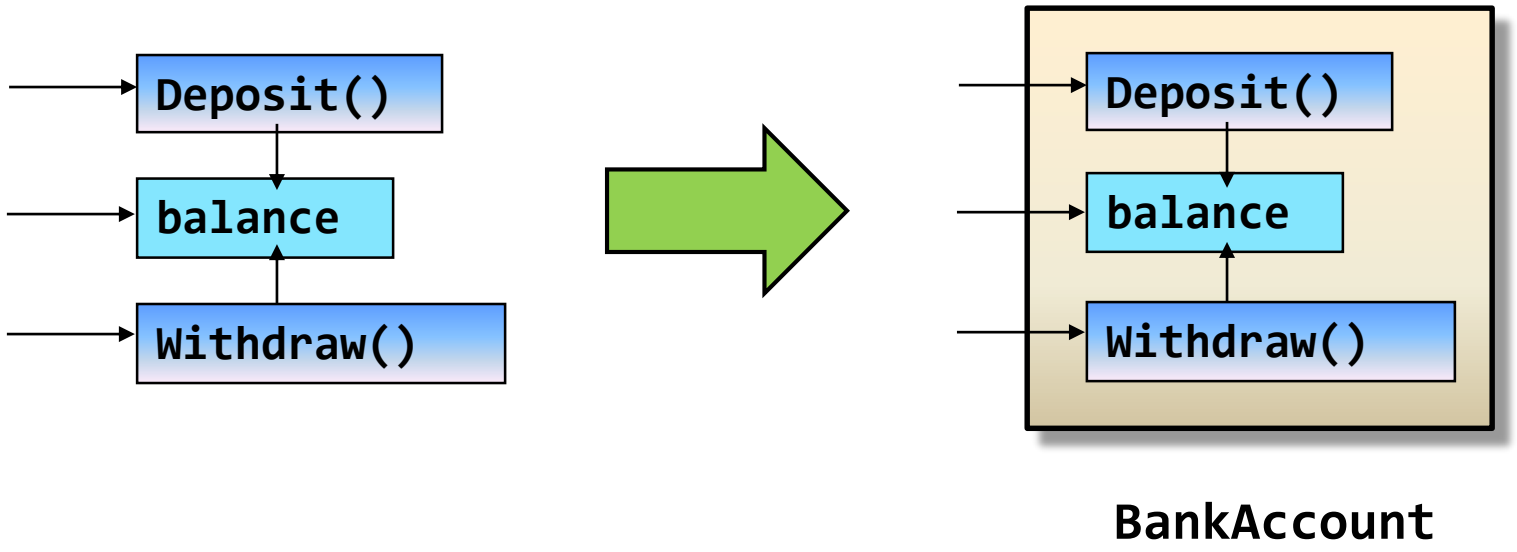
- Structs are “blueprints” for values
  - No distinguishable identity
  - No inaccessible state
  - No “behavior”
- Classes are “blueprints” for objects
  - Distinguishable identity
  - State can be inaccessible
  - Behavior central to object

Agenda

- Introducing Object-Oriented Programming
- **First Pillar of OOP: Encapsulation**
- Creating Classes and Objects
- Access Modifiers
- Lab 6
- Discussion and Review

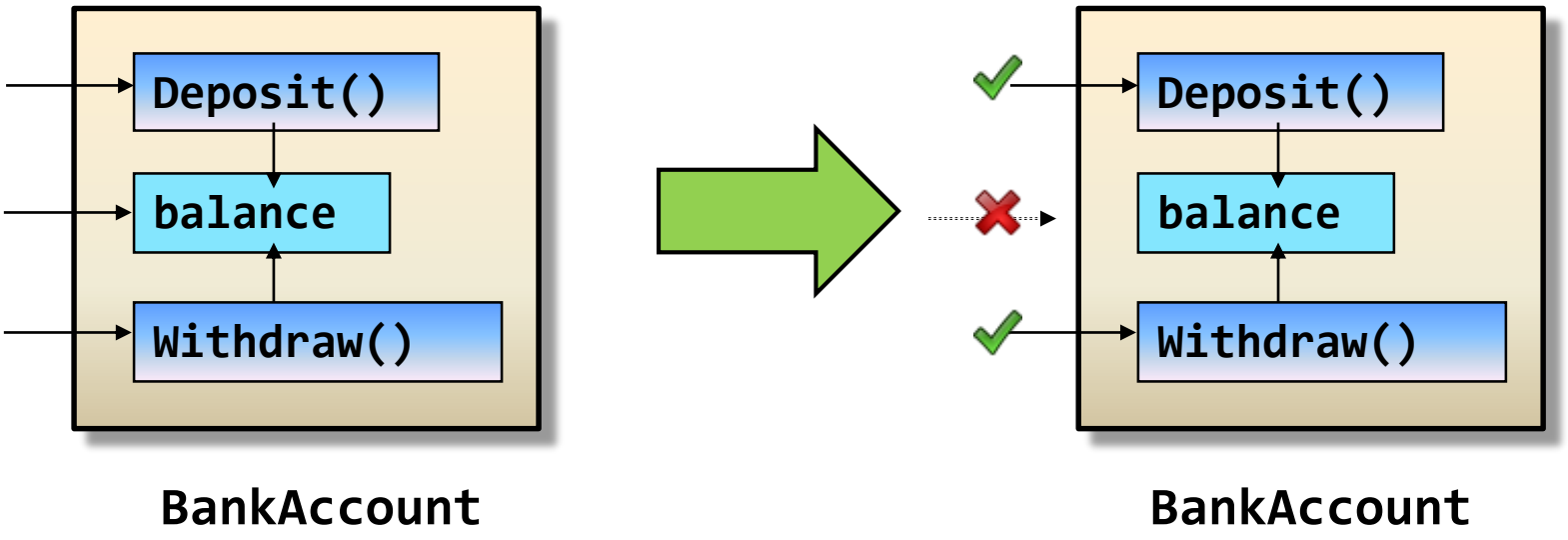
Introducing Encapsulation

- Grouping related ideas in a single unit



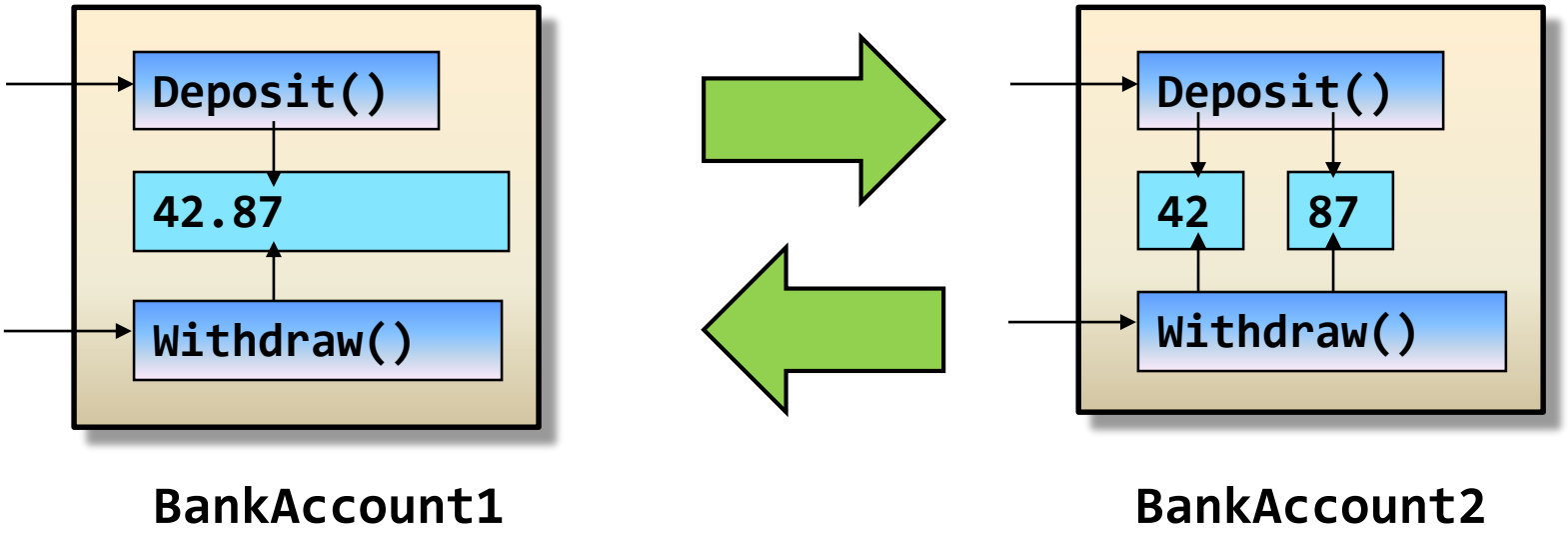
Introducing Encapsulation (2)

- The packaging of operations and attributes representing state into an object type so that state is accessible or modifiable only through the objects' interface



Introducing Encapsulation (3)

- The ability to hide internal detail to the outside
- Ability to reuse objects without internal representation



The Three Pillars of OOP

- Encapsulation
  - The grouping of related ideas in a single unit
  - The packaging of operations and attributes representing state into an object type so that state is accessible or modifiable only through the objects' interface
  - The ability to hide internal detail to the outside
  - Ability to reuse objects without internal representation
- Inheritance
- Polymorphism

Agenda

- Introducing Object-Oriented Programming
- First Pillar of OOP: Encapsulation
- **Creating Classes and Objects**
- Access Modifiers
- Lab 6
- Discussion and Review

Defining Classes

- Classes are defined using the **`class`** keyword

```
class Car
{
   public string petName;
   public int currentSpeed;

   public void PrintState()
   {
      Console.WriteLine( "{0} is going {1} km/h",
         petName,
         currentSpeed );
   }
   public void SpeedUp( int delta )
   {
      currentSpeed += delta;
   }
}
```

Allocating Objects

- Objects are instantiated by the new keyword

```
Car myCar = new Car();
myCar.petName = "Goofy";

for( int i = 0; i < 5; i++ )
{
   myCar.SpeedUp( 10 );
   myCar.PrintState();
}
```

- Objects are not allocated in memory until they are “new’ed”

```
Car myCar;
myCar.petName = "Goofy";
```

Default Constructor

- Every class has a *default constructor* method supplied out-of-the-box
  - Takes no arguments and has no return type
  - Sets all field data to a default value
- The constructor is invoked when an object is allocated with **new**
- The default constructor can be redefined

```
class Car
{
    public string petName;
    public int currentSpeed;

    public Car()
    {
        petName = "Chuck";
        currentSpeed = 10;
    }
}
```

Custom Constructors

- Any set of overloaded custom constructors can be defined

```
class Car
{  ...
   public Car( string pt )
   {
      petName = pt;
   }
   public Car(string pn, int cs)
   {
      petName = pn;
      currentSpeed = cs;
   }
}
```

```
Car chuck = new Car( "Chuck" );
Car goofy = new Car( "Goofy", 87 );

chuck.PrintState();
goofy.PrintState();
```

- Note: When you define a custom constructor, the compiler silently removes the built-in default constructor!

The `this` Keyword

- In any class the **`this`** keyword is a reference to the current object
- It can be used to e.g. resolve naming conflicts

```
class Car
{
   public string petName;

   public Car( string petName )
   {
      this.petName = petName;
   }
}
```

- Local variables overshadow member variables
- Useful with IntelliSense

Chaining Constructors

- Constructors can be chained using **`this`**
- In this way the core construction code can be kept non-duplicated
  - Often there is a central initialization method of sorts

```
public Car() : this( "Chuck" )
{
}
public Car( string petName ) : this( petName, 0 )
{
}
public Car( string petName, int currentSpeed )
{
   // This is the central initialization code
   this.petName = petName;
   this.currentSpeed = currentSpeed;
}
```

Revisiting Optional Arguments

- The optional arguments of Module 5 can also be applied for constructors

```
public Car( string petName = "Chuck", int
currentSpeed = 0 )
{
   // This is the central initialization code
   this.petName = petName;
   this.currentSpeed = currentSpeed;
}
```

```
Car alice = new Car( "Alice", 30 );
Car bob = new Car( "Bob" );
Car chuck = new Car( currentSpeed: 50 );
```

- Carefully chosen default values usually reduce the number of necessary constructors

Partial Classes

- The implementation of a class can be divided into multiple `.cs`-files

```
// Car.Constructors.cs
partial class Car
{
   public Car( string pt )
   {
      petName = pt;
   }
   public Car(string pn, int cs)
   {
      petName = pn;
      currentSpeed = cs;
   }
}
```

```
// Car.cs
partial class Car
{
   public string petName;
   public int currentSpeed;

   public void SpeedUp( int delta )
   {
      currentSpeed += delta;
   }
}
```

Rules of Thumb

"Nouns are classes.
Verbs are their methods.
Adjectives are their properties".

Agenda

- Introducing Object-Oriented Programming
- First Pillar of OOP: Encapsulation
- Creating Classes and Objects
- **Access Modifiers**
- Lab 6
- Discussion and Review

Access Modifiers

Access Modifier	Meaning…
`public`	No access restrictions
`private`	Can only be accessed by the defining type
`protected`	Can only be accessed by the defining type and its derived types
`internal`	Accessible only within the current assembly defining the type
`protected internal`	`Protected` + `Internal`; Accessible only within the current assembly defining the type as well as in derived types

Default Access Modifiers

- Members are implicitly private
- Types are implicitly internal

```
namespace Devices
{
   class Radio    // internal class
   {
      Radio()     // private constructor
      {
      }
   }
}
```

- Good style to declare access modifier explicitly (even if default)

Access Modifiers and Nested Types

- Nested types can be access-modified as well

```
public class Tv
{
   private enum Encoding { Mpeg2, Mpeg4 }; // Only visible
                                           // inside Tv class

   public Tv()
   {
   }
}
```

- Top-level types cannot be private!

A Matter of Style and Taste

- There are no mandatory rules for the nomenclature of classes, members etc.
- Best approach is to follow Microsoft ☺
  - Classes and other Types are PascalCase
  - Methods and Properties are PascalCase
  - Public member variables are PascalCase
  - Parameters are camelCase
- Religious issues
  - Private member variables are _camelCase
  - Member variables at top of class definition
    - Except… ☺
  - …

```
class Car
{
    public string PetName;
    private int _currentSpeed;

    public void SpeedUp(int delta)
    {
        ...
    }
}
```

TEKNOLOGISK INSTITUT

Quiz: Classes – Right or Wrong?

```
class Car
{
   public string PetName;
   public int CurrentSpeed;
}
```

```
Car c;
c.PetName = "Beardyman";
```

```
Car c = new Car();
c.PetName = "Beardyman";
```

```
class Person
{
   string Name;

   public void Person(string name)
   {
      this.Name = name;
   }
}
```

```
Person p = new Person("Dude");
```

```
Person p = new Person();
```

```
Person p = new Person("Dude");
p.Name = "Homie";
```

TEKNOLOGISK
INSTITUT

Lab 6: Creating Classes

Discussion and Review

- Introducing Object-Oriented Programming
- First Pillar of OOP: Encapsulation
- Creating Classes and Objects
- Access Modifiers


WINCUBATE
Jesper Gulmann Henriksen
PhD, MCT, MCSD, MCPD
Phone : +45 22 12 36 31
Email : jgh@wincubate.net
WWW : http://www.wincubate.net
Hasselvangen 243
8355 Solbjerg
Denmark