OBJECT-ORIENTED PROGRAMMING WITH C#

OBJECT CONCEPTS

issntt@nus.edu.sg

Objectives

At the end of the course, students will be able to

- Distinguish value-type from reference-type variables
- Differentiate the different effects when assigning value-type and reference-type variables
- Understand how C# garbage collection works to write more efficient code
- Differentiate the different effects when passing value-type and reference-type variables as method arguments
- Use *this* reference properly
- Design and implement static and non-static variables/methods properly

Topics

- **Data Types**
  - **Value Types vs Reference Types**
- Assignment on variables
- Garbage Collection (Self-Study)
- Passing Arguments to Methods
- The *this* reference
- Static variables
- Static methods

Data Types

Every variable has a **data type**, which determines

- The **values** that the variable can contain, and
- The **operations** that can be performed on it

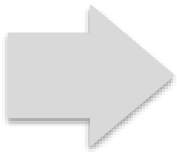
```
int number = 10;
number++;

string course = "oopcs";
course = course.ToUpper();
```

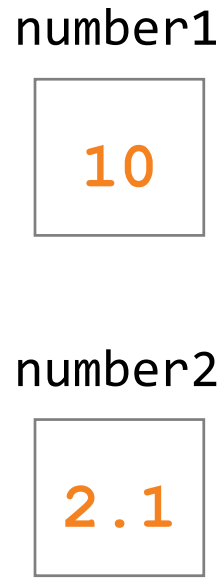
Can we assign *oopcs* to *number*? Can we call *course++*?

2 Categories of Data Types

A **value-type** variable stores **real values**

```
int number1 = 10;
double number2 = 2.1;
```



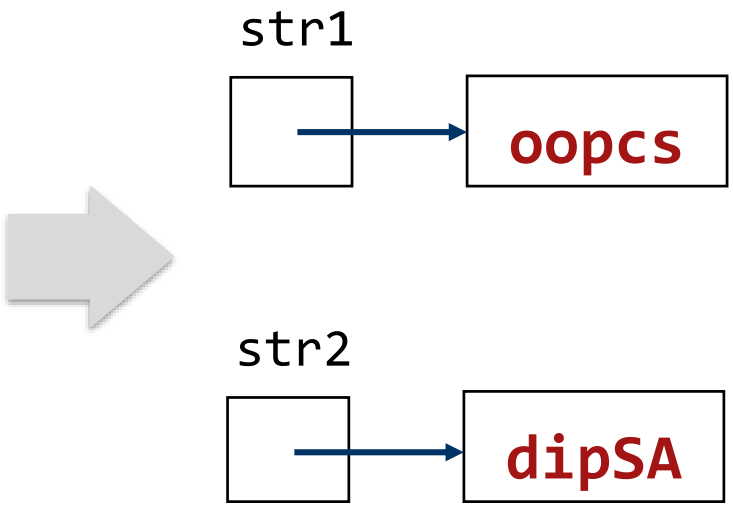


Some common value types include **bytes**, **short**, **int**, **long**, **float**, **double**, **bool**, **Char**

2 Categories of Data Types

A **reference type** variable **stores references** to the data **(NOT real values)**

```
string str1 = "oopcs";
string str2 = "dipSA";
```



Some common reference types include **string** and **object**

Topics

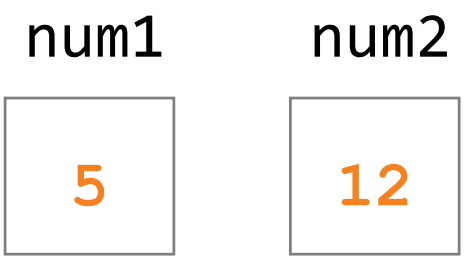
- Data Types
- **Assignment on variables**
  - **Value-type vs Reference-type variables**
  - **Alias**
- Garbage Collection (Self-Study)
- Passing Arguments to Methods
- The *this* reference
- Static variables
- Static methods

Assignment on Value-Type Variables

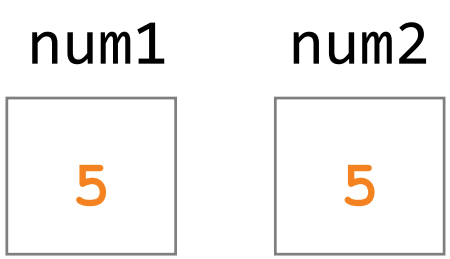
On assignment (operator =), the **real value** of a value-type variable is **copied**

```
int num1 = 5;
int num2 = 12;

num2 = num1;
```

Before



After



Object Instantiation - Revisit

The keyword *new* is used to **instantiate** objects

```
class Car
{
   private String model;
   private String color;

   public Car(String model, String color)
   {
      this.color = color;
      this.model = model;
   }

   public Car() :
          this("Toyota Camry", "red") { }
   // Setters and getters omitted for brevity
}
```

```
new Car();
```

How can we **access** the object just created, e.g. to retrieve its *model*?

Accessing Objects

We often need a **variable** that **references** to the **object**, and use it to **access** the object

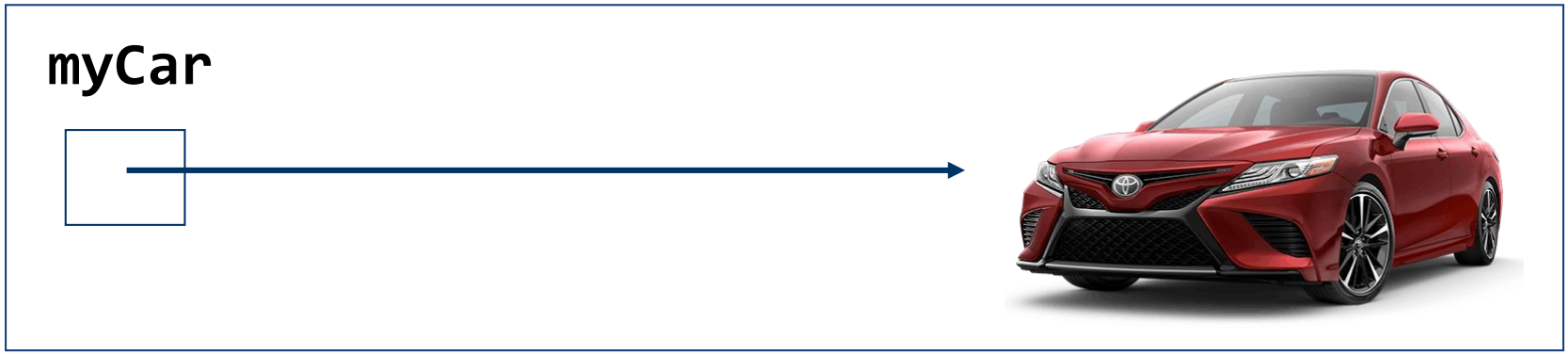
```
class Car
{
  private String model;
  private String color;

  public Car(
      String model, String color)
  {
    this.color = color;
    this.model = model;
  }

  public Car()
    : this("Toyota Camry", "red")
  { }

  // Setters and getters omitted
}
```

```
Car myCar = new Car();
Console.WriteLine(
    myCar.getModel());
```

Objects involve References

A reference variable stores a **reference** and **not the real value** of the object

```
Car myCar = new Car("Toyota Camry", "red");
```

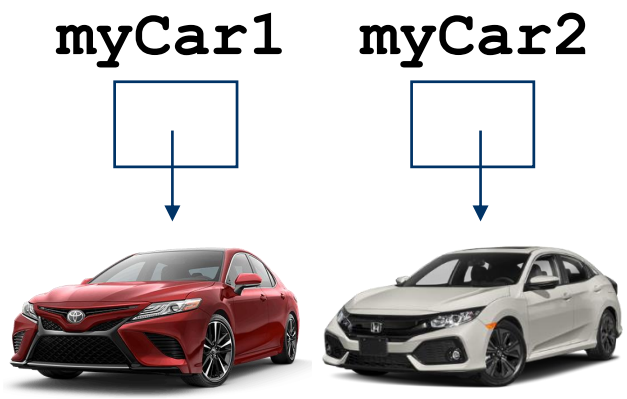


Assignment on Reference Type Variables

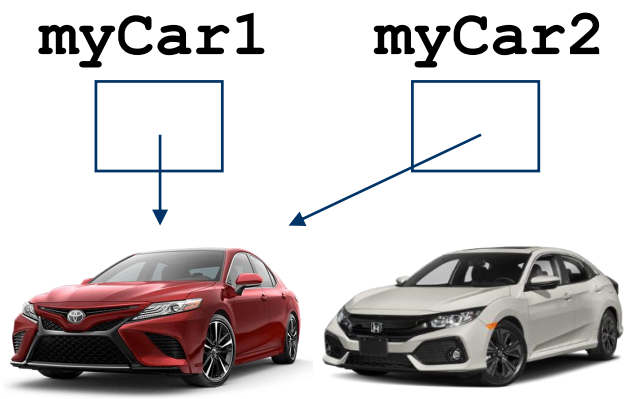
On assignment (operator =), **references**, i.e., memory location addresses, not values, **are copied**

```
Car myCar1 = new Car("Toyota Camry", "red");
Car myCar2 = new Car("Honda Civic", "white");
myCar2 = myCar1;
```

Before



After



Quiz

How does memory look like while running this snippet?

```
class Person
{
  private string name;
  private string country;
  private int age;
  // Getters and Setters

  public static void Main() {
    Person person1 = new Person();
    person1.SetName("Arian");
    person1.SetCountry("USA");
    person1.SetAge(24);

    Person person2 = new Person();
    person2.SetName("Bryan");
    person2.SetAge(20);

    Person person3 = person2;
  }
}
```

Quiz

What is the output of the following program?

```
class Student {
  private string name;

  public Student(
          string name) {
    this.name = name;
  }

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

```
public static void Main() {
  Student student1 =
        new Student("Chandler");
  Student student2 =
        new Student("Joey");

  student2 = student1;
  student2.Name = "Ross";

  Console.WriteLine(student1.Name);
  Console.WriteLine(student2.Name);
}
```

Alias

- **Two or more references** that refer to the **same object** are called ***alias***
- The object (and its member data) can be **accessed** using **different reference variables**
- Changing the object's state (e.g., instance variables) through one reference changes it for **all its aliases**

Next

In the last quiz, what will happen to the `Student` object named *"Joey"*?

Topics

- Data Types
- Assignment on variables
- **Garbage Collection (Self-Study)**
- Passing Arguments to Methods
- The *this* reference
- Static variables
- Static methods

Garbage Collection

Self study

- When an object **no longer** has any valid **references** to it, it can no longer be **accessed** by the program
  - It serves no useful purpose, and is therefore called **garbage**
- The .NET runtime **automatically releases its memory** and restores these spaces to the system for future use
  - The process is called garbage collection

What does it mean to us (developers)?

Efficient usage of Memory

Self study

- Do not keep references to variables that are **not used anymore**
- Another good reason to **use modular programming**

Topics

- Data Types
- Assignment on variables
- Garbage Collection
- **Passing Arguments to Methods**
  - **Default: passed-by-value**
    - **Value Type vs Reference Type arguments**
  - Using *ref*: passed-by-reference
  - Changing the values of passed arguments
- The *ref* reference
- Static variables
- Static methods

Method Arguments - Revisit

As you have known, **passed arguments** when calling a method may be **literals**, **variables**, or **values returned** from **other methods**

```
Console.WriteLine(
                min(2, 3));
```

```
int n1 = 5;
int n2 = 4;
Console.WriteLine(
  min(n1, n2));
```

```
int m1 = 9;
int m2 = 7;
int m3 = 8;
int smallest =
  min(min(m1, m2), m3);
Console.WriteLine(smallest);
```

```
static int min(
  int num1, int num2)
{
   if (num1 < num2)
      return num1;

   return num2;
}
```

How does the **passing arguments** to methods really work?

Passing Arguments to Methods

- By default, C# arguments **are passed by value**
- That is, a **local copy** of an **argument's value** is made and passed to the called method

Passing Arguments to Methods

- For **value-type** arguments, the called method receives a local copy with **the real value**. Therefore
  - All the arguments of value-type **will not be changed** by the method called
- For **reference-type** arguments, the called method receives a local copy with **the reference**. Therefore
  - The **content** of the object **can be changed**
  - The **reference** itself **cannot be changed**

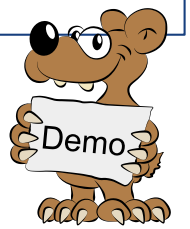
Why? **Hint**: look at the "*how memory looks like quiz*" again

If you understand why, you don't need to remember the rules above

Passing Arguments - Quiz 1

What is the output of the following program?

```
public static void Main() {
    int num = 1;
    Console.WriteLine("Before calling the method, " +
        " num is " + num);
    ChangeValue(num);
    Console.WriteLine("After calling the method, " +
        " num is " + num);
}
public static void ChangeValue(int n) {
    n = n + 1;
    Console.WriteLine("n is " + n);
}
```



Passing Arguments - Quiz 2

What is the output of the following program?

```
public static void Main()
{
   Student myStudent = new Student("Monica");
   Console.WriteLine("Before calling the method, "
      + "name is " + myStudent.Name);

   ChangeValue(myStudent);
   Console.WriteLine("After calling the method, "
      +  " name is " + myStudent.Name);
}
public static void ChangeValue(Student myStudent)
{
   myStudent = new Student("Pheobe");
   Console.WriteLine(myStudent.Name);
}
```

```
class Student {
  private string name;
  public Student
       (string name) {
    this.name = name;
  }
  public string Name {
    get {
      return name;
    }
    set {
      name = value;
    }
  }
}
```

Passing Arguments - Quiz 3

What is the output of the following program?

```
public static void Main()
{
   Student student = new Student("Monica");
   Console.WriteLine("Before calling the method, "
      + "name is " + student.Name);

   ChangeValue(student);
   Console.WriteLine("After calling the method, "
      + " name is " + student.Name);
}
public static void ChangeValue(Student student)
{
   student.Name = "Phoebe";
   Console.WriteLine(student.Name);
}
```

```
class Student {
  private string name;
  public Student
       (string name) {
    this.name = name;
  }
  public string Name {
    get {
      return name;
    }
    set {
      name = value;
    }
  }
}
```

Topics

- Data Types
- Assignment on variables
- Garbage Collection
- **Passing Arguments to Methods**
  - Default: passed-by-value
    - Value Type vs Reference Type arguments
  - **Using *ref*: passed-by-reference**
  - **Changing the values of passed arguments**
- The *ref* reference
- Static variables
- Static methods

Passing Arguments to Methods

- By default, C# arguments are **passed by value**
- But we can make C# arguments **passed by reference** using the `ref` keyword
- Much more **complicated and hard-to-maintain** code

Passing Arguments - Quiz

What is the output of the following program?

```
public static void Main() {
   int num = 1;

   Console.WriteLine("Before calling the method, " +
      " num is " + num);
   ChangeValue(ref num);
   Console.WriteLine("After calling the method, " +
      " num is " + num);
}

public static void ChangeValue(ref int n) {
   n = n + 1;
   Console.WriteLine("n is " + n);
}
```

Question

In the last few examples, we **make change** to the values of **passed arguments**. Is it a good idea?

Changing the Values of Passed Arguments

- Most of the time, this is a **bad practice**, because
  - The called function can **unexpectedly corrupt** the caller's data, i.e., **side effects**

- A method is much **easier to be maintained** when
  - All **inputs are unchanged**
  - It only does **one thing**
  - There's only **one output**

Is there any exception?

Topics

- Data Types
- Assignment on variables
- Garbage Collection
- Passing Arguments to Methods
- **The *this* reference**
- Static variables
- Static methods

`this` Reference

- Keyword *`this`* references to the **current instance**
- Keyword *`this`* is used to
  1. Refer to **instance methods** and **variables** (optional)
  2. Resolve **naming conflicts**
  3. Refer to the **constructor** of the **current class**
  4. …

Keyword `this`

Keyword *`this`* can be used to refer to **instance variables**, **properties,** or **methods** (optional)

```
public class Circle {
  private double radius;

  public double GetArea() {
    return this.radius *
       this.radius *
                   Math.PI;
  }
  public void Print() {
    Console.WriteLine(
      "{0}, {1}",
        this.radius,
        this.GetArea());
  }
}
```

≈

```
public class Circle {
  private double radius;

  public double GetArea()
  {
    return radius *
       radius * Math.PI;
  }
  public void Print()
  {
    Console.WriteLine(
       "{0}, {1}",
          radius,
          GetArea());
  }
}
```

Keyword `this`

If there is a **name conflict**, keyword ***this*** must be used to **resolve** it

```
public class Circle
{
  private double radius;

  public Circle(double radius)
  {
     this.radius = radius;
  }
}
```

```
public class Circle
{
   private double radius;

   public Circle(double radius)
   {
      radius = radius;
   }
}
```

Keyword `this`

Because *`this`* references the **current instance** itself, it can also be used to **invoke another constructor**

```
public class Circle
{
   private double radius;

   public Circle(double radius)
   {
      this.radius = radius;
   }

   public Circle()  :  this(0.0)
   {
      // More statements can be inserted here
   }
}
```

The parameter list must **match** in terms of the *quantity*, *sequence* & *types*

Statements in the called constructor will be executed **before** statements the caller constructor

Quiz What is the output of the following program?

```
class ThisReferenceExample
{
    private int number;

    public ThisReferenceExample(int number)
    {
        this.number = number;
    }
    public void Method1()
    {
        Console.WriteLine(
                    "Start Method1");
        this.Method2();
    }
    public void Method2()
    {
        Console.WriteLine("Start Method2");
        Console.WriteLine(this.number);
    }
}
```

```
public static void Main()
{
  ThisReferenceExample ex =
    new ThisReferenceExample(1);
  ex.Method1();
}
```

Quiz What is the output of the following program?

```
class ThisReferenceExample2
{
    private int number;

    public ThisReferenceExample2(int number)
    {
        number = number;
    }
    public void Method1()
    {
        Console.WriteLine(
                "Start Method1");
        Method2();
    }
    public void Method2()
    {
        Console.WriteLine("Start Method2");
        Console.WriteLine(number);
    }
}
```

```
public static void Main()
{
  ThisReferenceExample2 ex =
    new ThisReferenceExample2(1);
  ex.Method1();
}
```

Quiz What is the output of the following program?

```
class ThisReferenceExample3
{
   private int number1;
   private int number2;
   public ThisReferenceExample3()
      : this(-1, -1) {}
   public ThisReferenceExample3(int number1)
      : this(number1, -1) {}
   public ThisReferenceExample3(
         int number1, int number2)
   {
      this.number1 = number1;
      this.number2 = number2;
   }
   public void Print()
   {
      Console.WriteLine("number1 is " +
         number1 + " and number2 is " +
         number2);
   }
}
```

```
public static void Main()
{
  new ThisReferenceExample3()
                          .Print();
  new ThisReferenceExample3(1)
                          .Print();
}
```

Next

Consider the following scenario

We need to implement a class called *SavingsAccount* that has some attributes

1. *name*: the name of the account's holder
2. *balance*: the current balance of the account
3. *basisInterestRate*: the current basis interest rate

- Certainly, *name* and *balance* are **different** for **each instance**
- However, the *basisInterestRate* is **the same for all the instances**

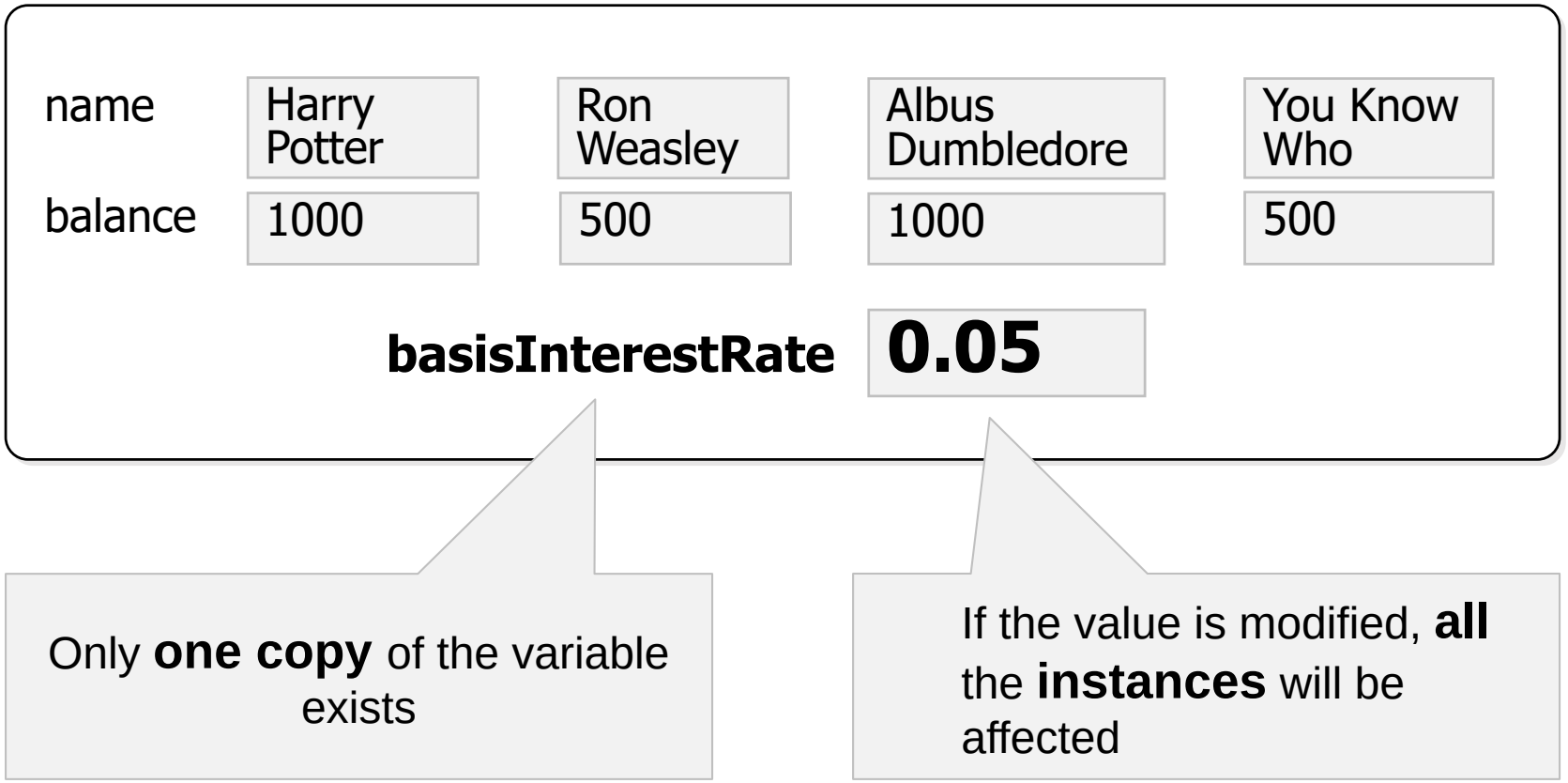
We know *name* and *balance* should be **instance's attributes**, but how to implement *basisInterestRate*?

Topics

- Data Types
- Assignment on variables
- Garbage Collection (Self-Study)
- Passing Arguments to Methods
- The *this* reference
- **Static variables**
- Static methods

Static Variables

A *static* variable (aka **class variable**) is **shared** by **all instances** of the class



Static Variables

A class variable is declared using keyword *static*

```
public class SavingsAccount {
  public static float
     basisInterestRate = 0.05f;

  private string name;
  private float balance;

  public SavingsAccount(string name,
      float initBalance) {
     this.name = name;
     this.balance = initBalance;
  }

  public void EarnInterest() {
     balance +=
     balance * basisInterestRate;
  }
  // … Print
}
```

```
public static void Main() {
  SavingsAccount sa1 =
     new SavingsAccount(
       "Harry Potter", 1000f);
  SavingsAccount sa2 =
     new SavingsAccount(
       "Ron Weasley", 500f);

  sa1.EarnInterest();
  sa2.EarnInterest();

  sa1.Print();
  sa2.Print();
}
```

```
SavingsAccount[name=Harry
Potter, balance=1050]
SavingsAccount[name=Ron
Weasley, balance=525]
```

Static Variables

- Can be accessed even **without an instance**
  - By using the class name
- Can be used even if **no instances** have been created in the system

Quiz

What is the output of the following program?

```
public static void Main() {
  SavingsAccount
     .basisInterestRate = 0.1f;

  SavingsAccount sa3 =
    new SavingsAccount(
      "Albus Dumbledore", 1000f);
  SavingsAccount sa4 =
    new SavingsAccount(
      "You Know Who", 500f);

  sa3.EarnInterest();
  sa4.EarnInterest();

  sa3.Print();
  sa4.Print();
}
```

```
public class SavingsAccount {
  public static float
    basisInterestRate = 0.05f;

  private string name;
  private float balance;

  public SavingsAccount(
      string name,
      float initBalance) {
    this.name = name;
    this.balance = initBalance;
  }
  public void EarnInterest() {
    balance = balance +
      balance*basisInterestRate;
  }
  public void Print() {
    Console.WriteLine(
      "SavingsAccount[name="
      + name + ", balance="
      + balance + "]");
  }
}
```

Static Variables

name	Harry Potter	Ron Weasley	Albus Dumbledore	You Know Who
balance	1000	500	1000	500
basic Interest Rate	**0.05**	**0.05**	**0.05**	**0.05**

If we implement the *basisInterestRate* as an **instance variable instead,** and the interest rate **changes** to 0.1, how many instances are needed to be updated?

Confused Terminology

- **Static** variables/methods = **class** variables/methods
- **Non-static** variables/methods = **instance** variables/methods
- **Attributes** = **fields** or **variables**

=> *These terminology is used **interchangeably***

Next

Consider the following scenario

```
class MyMath {
  public double Abs(double n)
  {
    if (n > 0) return n;
    return -n;
  }
}
```

```
class MyString {
  public string Concatenate(
     string s1, string s2) {
    return s1 + " " + s2;
  }
}
```

```
public static void Main()
{
 1 MyMath myMath = new MyMath();
   Console.WriteLine(myMath.Abs(-3));

 2 MyString myString = new MyString();
   Console.WriteLine(myString.
      Concatenate("Hello", "DipSA"));
}
```

Every time we need to use the methods of each class, we have to create an **unnecessary object** Can it be avoided?

Topics

- Data Types
- Assignment on variables
- Garbage Collection
- Passing Argument to Methods
- The *this* reference
- Static variables
- **Static methods**

Static Methods

No object is **needed** to call a *`static`* method

```
class SomeClass1
{
    public 1 static
        void AStaticMethod()
    {
        // Implementation
    }
}
```

```
class SomeClass2
{
    public 3 static void Method1()
    {
    2 SomeClass1.AStaticMethod();
    }

    public 3 void Method2()
    {
    2 SomeClass1.AStaticMethod();
    }
}
```

Static Methods

In the last scenario, methods can be converted to *`static`* and **invoked without any object**

```
class MyMath
{
   public 1 static
        double Abs(double n)
   {
      if (n > 0) return n;
      return -n;
   }
}
```

```
class MyString
{
   public 1 static
      string Concatenate(
         string s1, string s2)
   {
      return s1 + " " + s2;
   }
}
```

```
public static void Main()
{
   Console.WriteLine( 2 MyMath.Abs(-3));
   Console.WriteLine( 2 MyString.Concatenate(
                              "Hello", "DipSA"));
}
```

Static Methods

However, *static* methods **cannot** directly access **instance variables** (and therefore also **methods**)

```
class SavingsAccount {
   public static float basisInterestRate = 0.05f;
   private float balance;

   public float ComputeInterest() {
      return balance * basisInterestRate;
   }
   public static void Print() {
      Console.WriteLine("Account balance: " + balance +
         " and interest: " + ComputeInterest());
   }
}
```

Why not?

Static vs Non-static

- When to use *static*?
  - When the method/variable is NOT **dependent** on any **specific instance** of a class
- When NOT to use *static*?
  - When the method/variable is **dependent** on some **specific instance** of a class

Should setters/getters static or instance methods. Why?

Quiz

Static or non-static? Why?

1. In class `Circle`
   a. The *`radius`* variable?
   b. The *`PI`* variable?
   c. The *`getArea()`* method?
   d. The *`computePerimeter()`* method?

2. In class `Math`
   a. The *`pow()`* method
   b. The *`sin()`* method

Readings

- Visual C# 2012 How to Program, 5th edition – section 4.8, 4.9, 7.3, 7.16, 8.8, 10.4, 10.8. 10.9