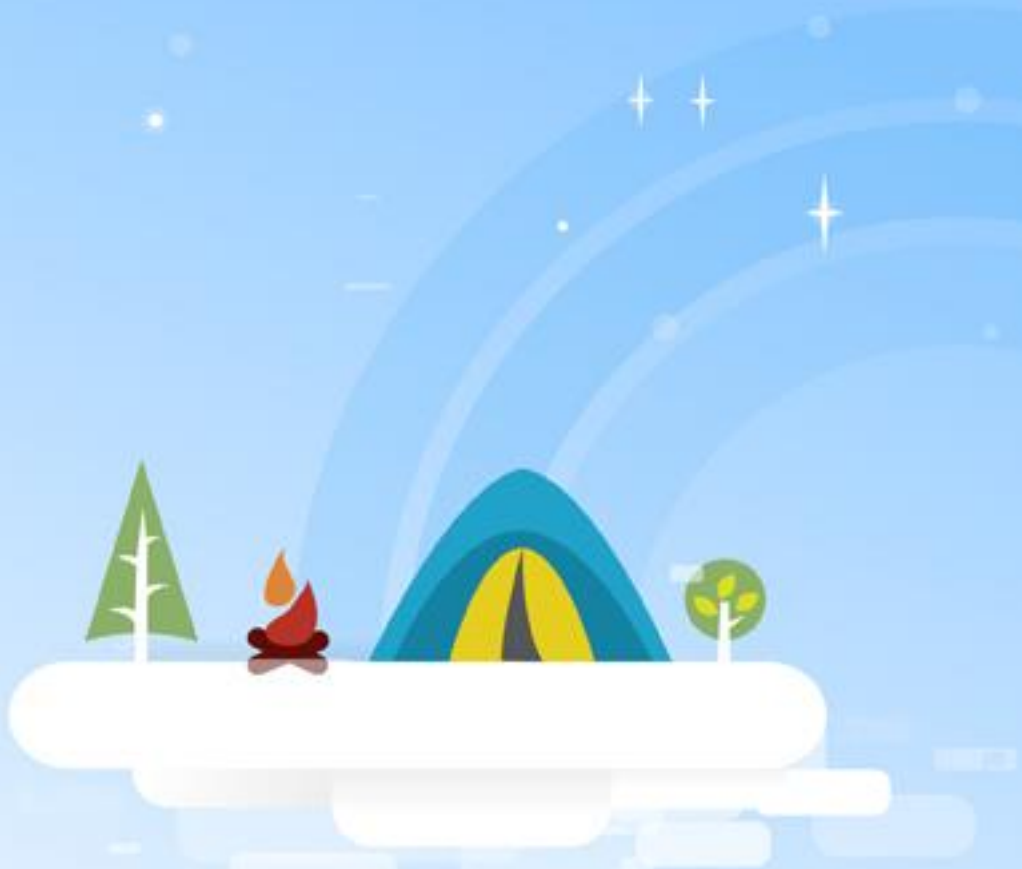

# Methods Design

Ridi Ferdiana | ridi@acm.org
Version 1.1.0

# Overview

Using Methods
Using Parameters
Using Overloaded Methods
Using Polymorpishm

# Using Methods

# Defining Methods

## Main is a method

- Use the same syntax for defining your own methods

```
using System;

class ExampleClass
{
    static void ExampleMethod( )
    {
        Console.WriteLine("Example method");
    }
    static void Main( )
    {
        // ...
    }
}
```

# Calling Methods

## After you define a method, you can:

- Call a method from within the same class
  - Use method's name followed by a parameter list in parentheses
- Call a method that is in a different class
  - You must indicate to the compiler which class contains the method to call
  - The called method must be declared with the **public** keyword
- Use nested calls
  - Methods can call methods, which can call other methods, and so on

# Using the return Statement

Immediate return

Return with a conditional statement

```
static void ExampleMethod( )
{
   int numBeans;
   //...

   Console.WriteLine("Hello");
   if (numBeans < 10)
      return;
   Console.WriteLine("World");
}
```

# Using Local Variables

## Local variables

- Created when method begins
- Private to the method
- Destroyed on exit

## Shared variables

- Class variables are used for sharing

# Returning Values

Declare the method with non-void type

Add a return statement with an expression

- Sets the return value
- Returns to caller

Non-void methods must return a value

```
static int TwoPlusTwo( ) {
    int a,b;
    a = 2;
    b = 2;
    return a + b;
}
```

```
int x;
x = TwoPlusTwo( );
Console.WriteLine(x);
```

# Using Parameters

# ◆ Using Parameters

Declaring and Calling Parameters
Mechanisms for Passing Parameters
Pass by Value
Pass by Reference
Output Parameters
Using Variable-Length Parameter Lists
Guidelines for Passing Parameters
Using Recursive Methods

# Declaring and Calling Parameters

- Declaring parameters
  - Place between parentheses after method name
  - Define type and name for each parameter
- Calling methods with parameters
  - Supply a value for each parameter

```
static void MethodWithParameters(int n, string y)
{ ... }

MethodWithParameters(2, "Hello, world");
```

# Mechanisms for Passing Parameters

- Three ways to pass parameters

| | |
|---|---|
| in | Pass by value |
| ref | Pass by reference |
| out | Output parameters |

# Pass by Value

Default mechanism for passing parameters:

- Parameter value is copied
- Variable can be changed inside the method
- Has no effect on value outside the method
- Parameter must be of the same type or compatible type

```
static void AddOne(int x)
{
      x++; // Increment x
}
static void Main( )
{
      int k = 6;
      AddOne(k);
      Console.WriteLine(k); // Display the value 6, not 7
}
```

# Pass by Reference

## What are reference parameters?

- A reference to memory location

## Using reference parameters

- Use the **ref** keyword in method declaration and call
- Match types and variable values
- Changes made in the method affect the caller
- Assign parameter value before calling the method

# Output Parameters

## What are output parameters?

- Values are passed out but not in

## Using output parameters

- Like **ref**, but values are not passed into the method
- Use **out** keyword in method declaration and call

```
static void OutDemo(out int p)
{
      // ...
}
int n;
OutDemo(out n);
```

# Variable-Length Parameter Lists

Use the params keyword

Declare as an array at the end of the parameter list

Always pass by value

```
static long AddList(params long[ ] v)
{
    long total, i;
    for (i = 0, total = 0; i < v.Length; i++)
        total += v[i];
    return total;
}
static void Main( )
{
    long x = AddList(63,21,84);
}
```

# Guidelines for Passing Parameters

## Mechanisms

- Pass by value is most common
- Method return value is useful for single values
- Use **ref** and/or **out** for multiple return values
- Only use **ref** if data is transferred both ways

## Efficiency

- Pass by value is generally the most efficient

# Demo

## Method and Parameter

# Polymorphism

# Polymorphism

- Polimorpishm : static and dynamic

# Static Polymorphism

- Function Overloading
- Operator Overloading

# Declaring Function Overloading

## Methods that share a name in a class

Distinguished by examining parameter lists

```
class OverloadingExample
{
    static int Add(int a, int b)
    {
        return a + b;
    }
    static int Add(int a, int b, int c)
    {
        return a + b + c;
    }
    static void Main( )
    {
        Console.WriteLine(Add(1,2) + Add(1,2,3));
    }
}
```

# Method Signatures

Method signatures must be unique within a class

Signature definition

**Forms Signature Definition**

- Name of method
- Parameter type
- Parameter modifier

**No Effect on Signature**

- Name of parameter
- Return type of method

# Using Overloaded Methods

## Consider using overloaded methods when:

- You have similar methods that require different parameters
- You want to add new functionality to existing code

## Do not overuse because:

- Hard to debug
- Hard to maintain

# Operator Overloading

## Manipulating operator to support ‘custom’ entity

```
public static Box operator+ (Box b, Box c) {
   Box box = new Box();
   box.length = b.length + c.length;
   box.breadth = b.breadth + c.breadth;
   box.height = b.height + c.height;
   return box;
}
```

# Dynamic Polymorphism

- implemented by abstract classes and virtual functions.
- Dynamic polymorphism is stopped when classes is sealed
- Absract can’t be sealed

# Demo
## Polymorphism

# Review

Two types of methods return-able value  and non return value (void)

Three types of parameters which are by value, by reference, and out parameter)

Overloaded Method different parameter same method name.

Abstract class and Virtual Method helps to implement Polymorphism