Object-Oriented Programming Using Microsoft Visual C# .NET

Using Classes and Objects

OOP in C# language

What is Class?

- The formal definition of class:

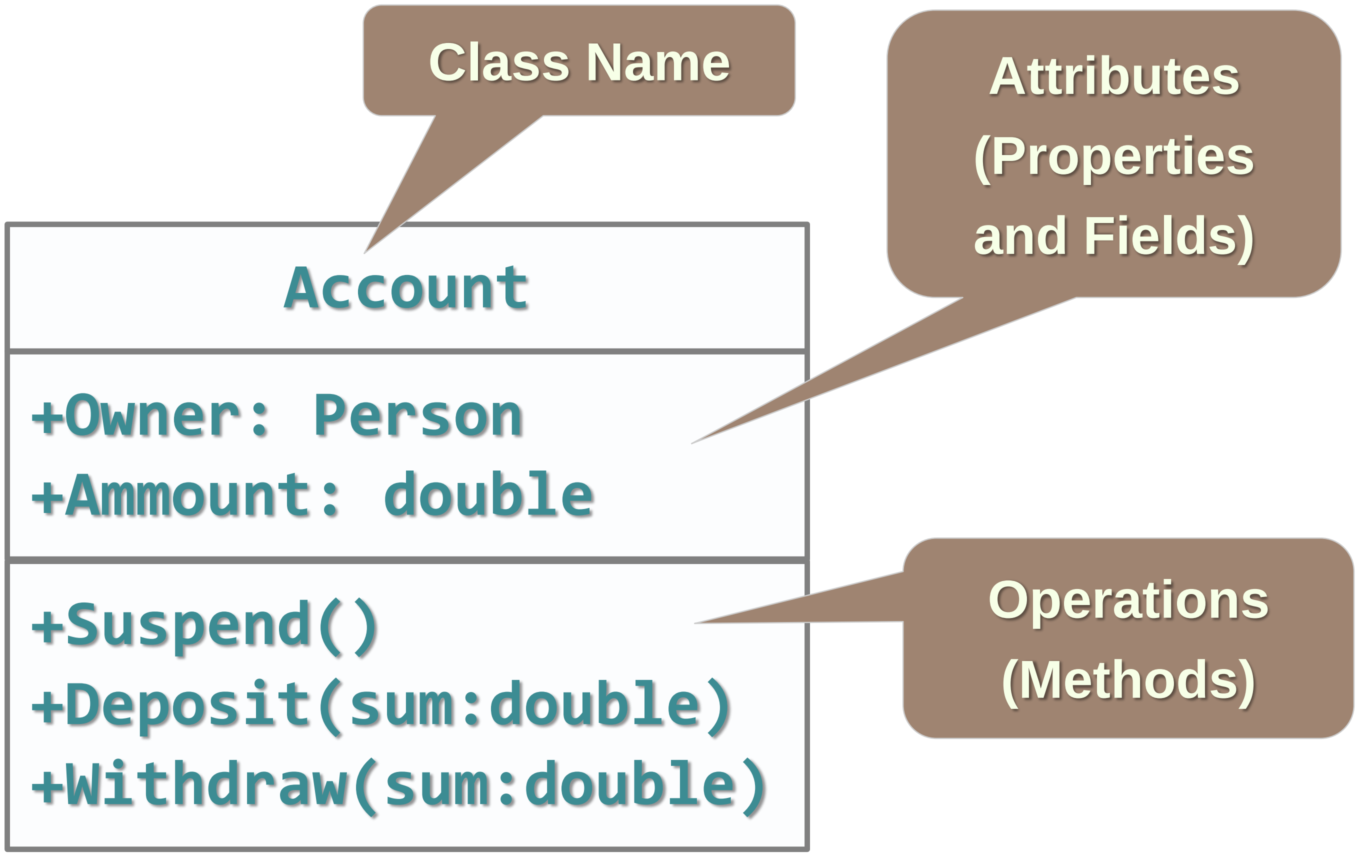
Classes act as templates from which an instance of an object is created at run time. Classes define the properties of the object and the methods used to control the object's behavior.

Definition by Google

Classes

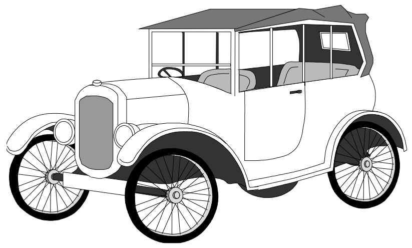
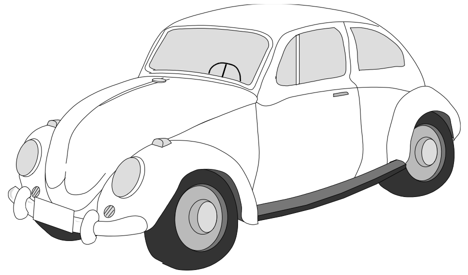
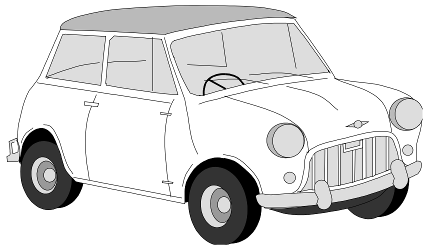
- Classes provide the structure for objects
  - Define their prototype, act as template
- Classes define:
  - Set of attributes
    - Represented by variables and properties
    - Hold their state
  - Set of actions (behavior)
    - Represented by methods
- A class defines the methods and types of data associated with an object

Classes – Example

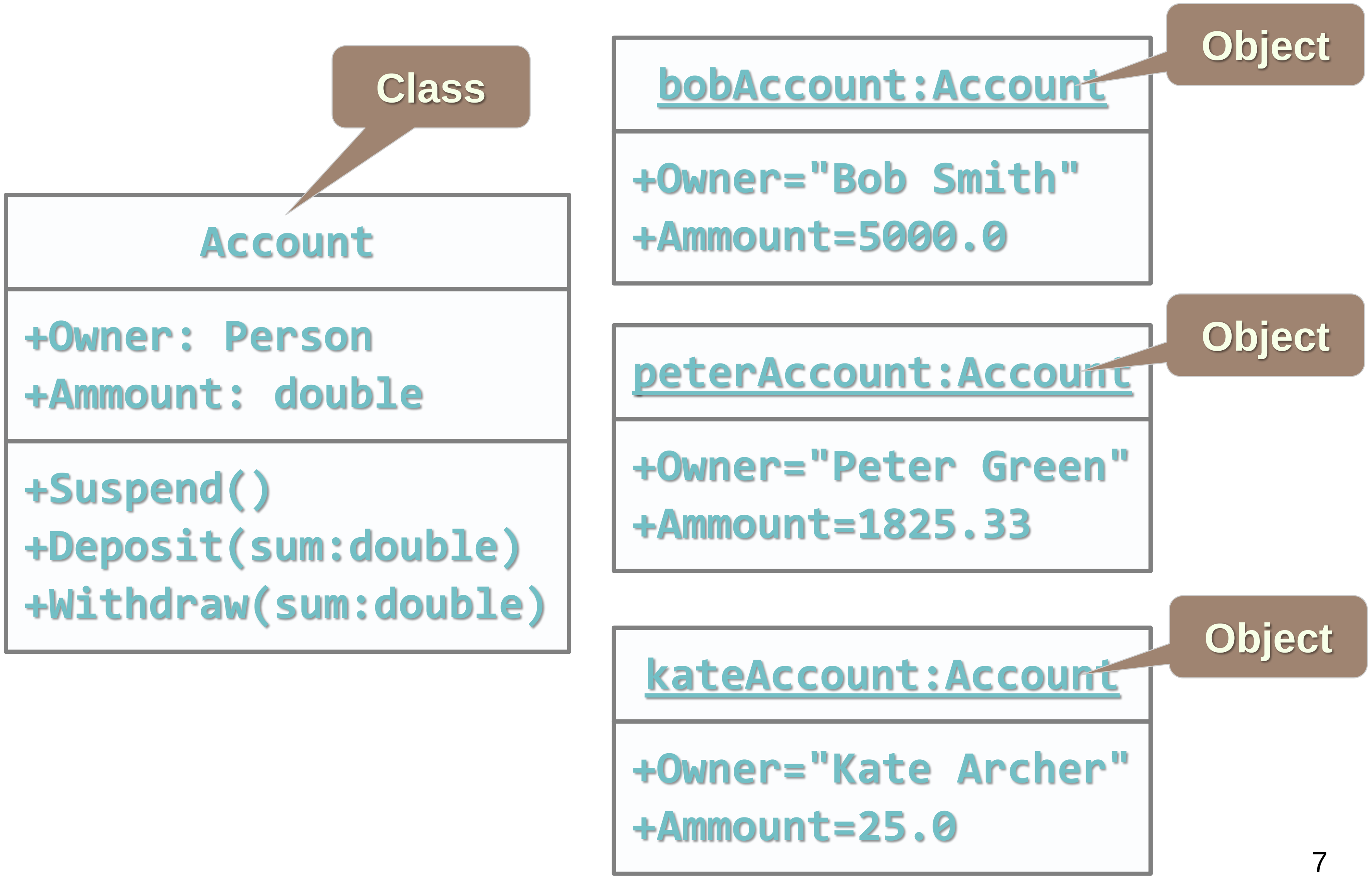


Objects

- An object is a concrete instance of a particular class
- Creating an object from a class is called instantiation
- Objects have
  - State: Set of values associated to their attributes (Objects store information)
  - Identity: Objects are distinguishable from one another
  - Behavior: Objects can perform tasks

Objects – Example



Classes in C#

Using Classes and their Class Members

Classes in C#

- Basic units that compose programs
- Implementation is encapsulated (hidden)
- Classes in C# can contain:
  - Fields (member variables)
  - Properties
  - Methods
  - Constructors
  - Inner types
  - Etc. (events, indexers, operators, …)

Classes in C# – Examples

- Example of classes:
  - `System.Console`
  - `System.String` (`string` in C#)
  - `System.Int32` (`int` in C#)
  - `System.Array`
  - `System.Math`
  - `System.Random`

Declaring Objects

- An instance of a class can be defined like any other variable

- Instances cannot be used if they are not initialized

Fields and Properties

Accessing Fields and Properties

Fields

- Fields are data members of a class
- Can be variables and constants
- Accessing a field doesn't invoke any actions of the object
- Example:
  - `String.Empty` (the `""` string)

Accessing Fields

- Constant fields can be only read
- Variable fields can be read and modified
- Usually properties are used instead of directly accessing variable fields

Properties

- Properties look like fields (have name and type), but they can contain code, executed when they are accessed
- Usually used to control access to data fields (wrappers), but can contain more complex logic
- Can have two components (and at least one of them) called accessors
  - get for reading their value
  - set for changing their value

Properties (2)

- According to the implemented accessors properties can be:
  - Read-only (get accessor only)
  - Read and write (both get and set accessors)
  - Write-only (set accessor only)
- Example of read-only property:
  - `String.Length`

Instance and Static Members

Accessing Object and Class Members

Instance and Static Members

- Fields, properties and methods can be:
  - Instance (or object members)
  - Static (or class members)
- Instance members are specific for each object
  - Example: different dogs have different name
- Static members are common for all instances of a class
  - Example: `DateTime.MinValue` is shared between all instances of `DateTime`

Accessing Members – Syntax

- Accessing instance members
  - The name of the instance, followed by the name of the member (field or property), separated by dot (".")

```
<instance_name>.<member_name>
```

- Accessing static members
  - The name of the class, followed by the name of the member

```
<class_name>.<member_name>
```

Instance and Static Members – Examples

- Example of instance member
  - `String.Length`
    - Each string object has different length
- Example of static member
  - `Console.ReadLine()`
    - The console is only one (global for the program)
    - Reading from the console does not require to create an instance of it

Methods

Calling Instance and Static Methods

Methods

- Methods manipulate the data of the object to which they belong or perform other tasks
- Examples:
  - `Console.WriteLine(…)`
  - `Console.ReadLine()`
  - `String.Substring(index, length)`
  - `Array.GetLength(index)`

Instance Methods

- Instance methods manipulate the data of a specified object or perform any other tasks
  - If a value is returned, it depends on the particular class instance
- Syntax:
  - The name of the instance, followed by the name of the method, separated by dot

```
<object_name>.<method_name>(<parameters>)
```

Static Methods

- Static methods are common for all instances of a class (shared between all instances)
  - Returned value depends only on the passed parameters
  - No particular class instance is available
- Syntax:
  - The name of the class, followed by the name of the method, separated by dot

```
<class_name>.<method_name>(<parameters>)
```

Calling Static Methods – Examples

```
using System;

double radius = 2.9;
double area = Math.PI * Math.Pow(radius, 2);
Console.WriteLine("Area: {0}", area);
// Area: 26,4207942166902

double precise = 8.7654321;
double round3 = Math.Round(precise, 3);
double round1 = Math.Round(precise, 1);
Console.WriteLine(
    "{0}; {1}; {2}", precise, round3, round1);
// 8,7654321; 8,765; 8,8
```

Constant field

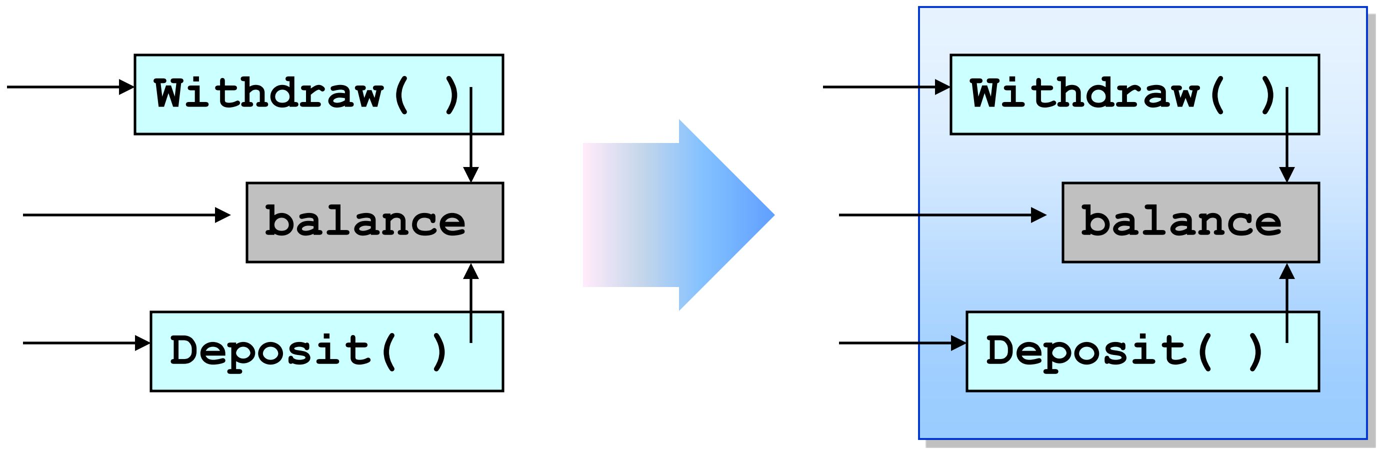
Static method

Static method

Static method

Combining Data and Methods

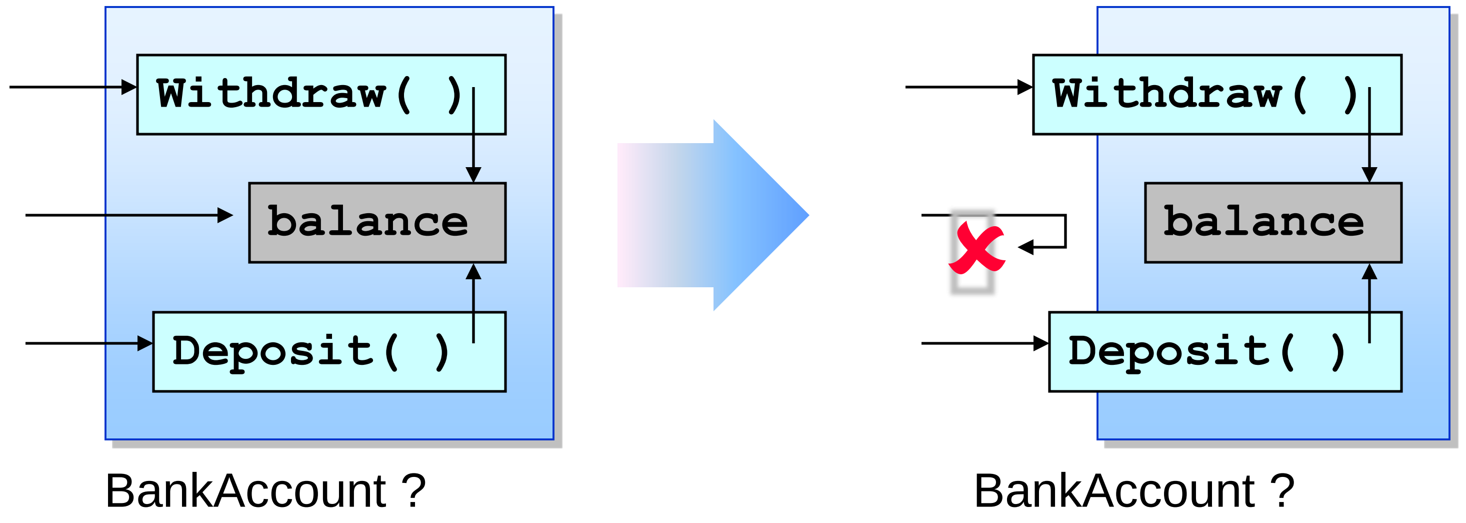
- Combine the data and methods in a single *capsule*
- The capsule boundary forms an inside and an outside



BankAccount ?   BankAccount ?

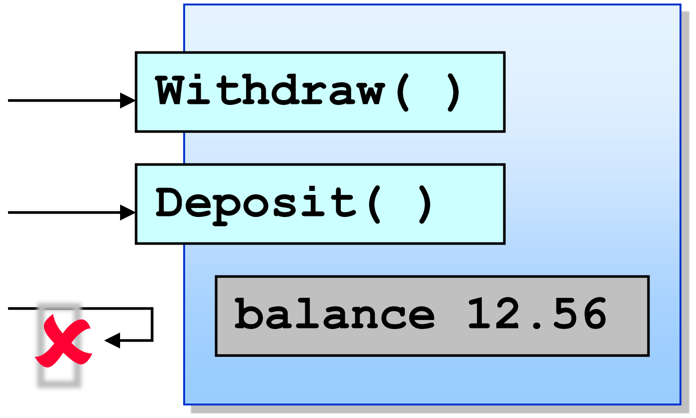
Controlling Access Visibility

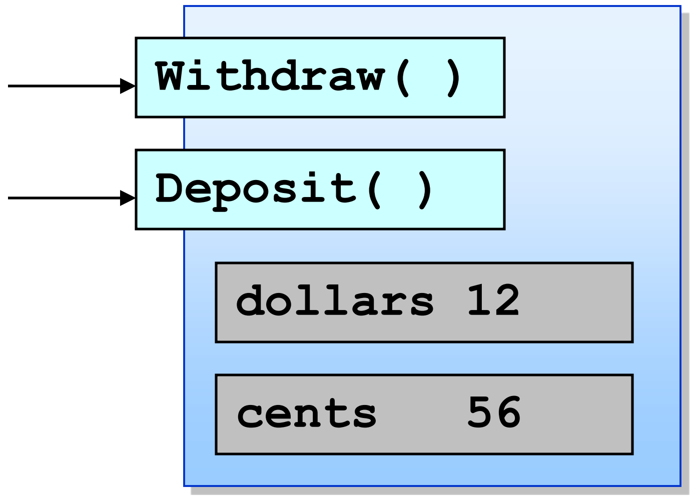
- Methods are *public*, accessible from the outside
- Data is *private*, accessible only from the inside



Why Encapsulate?

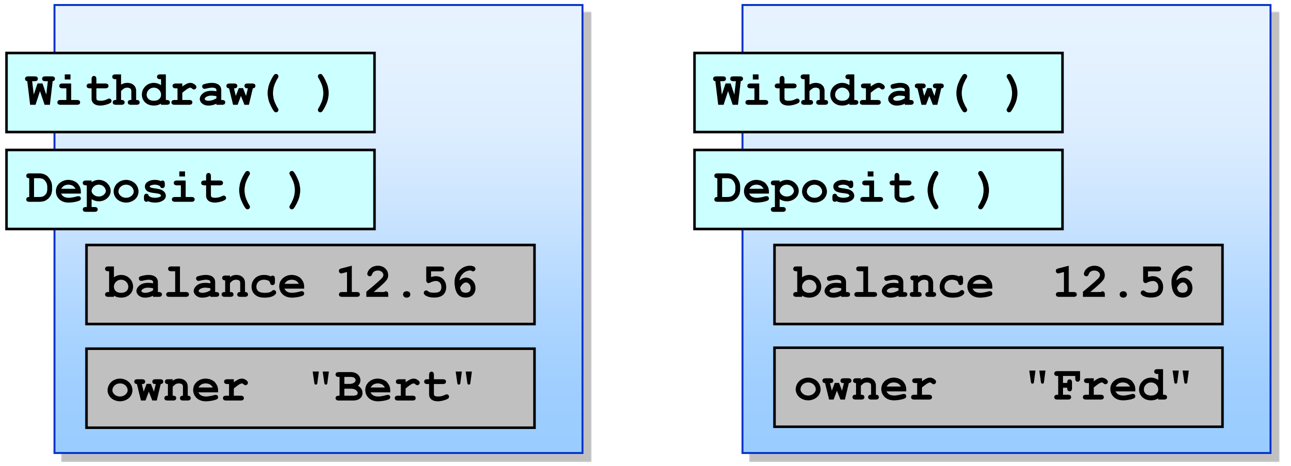
- Allows control
  - Use of the object is solely through the public methods
- Allows change
  - Use of the object is unaffected if the private data type changes





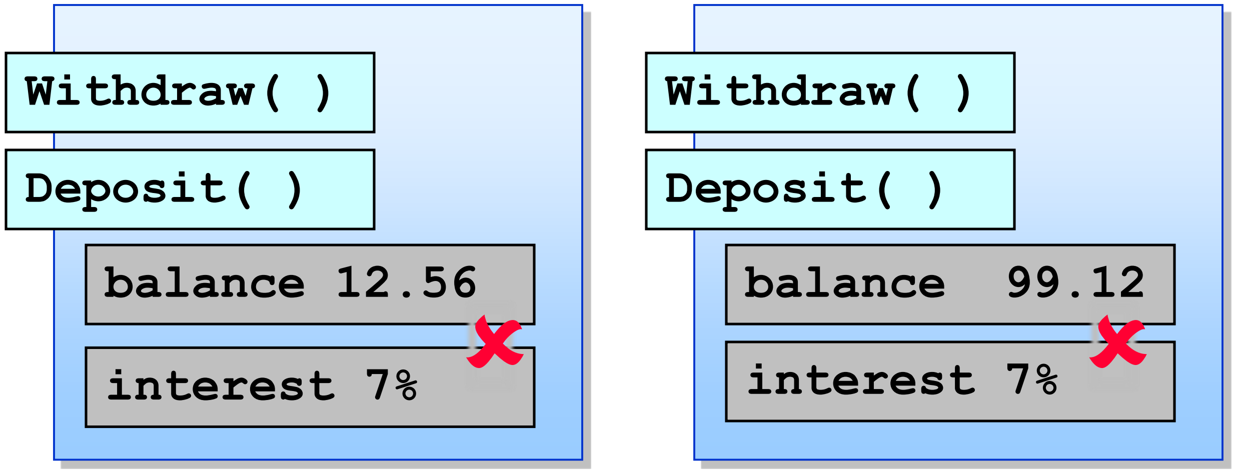
Object Data

- Object data describes information for *individual* objects
  - For example, each bank account has its <u>own</u> balance. If two accounts have the same balance, it is only a coincidence.



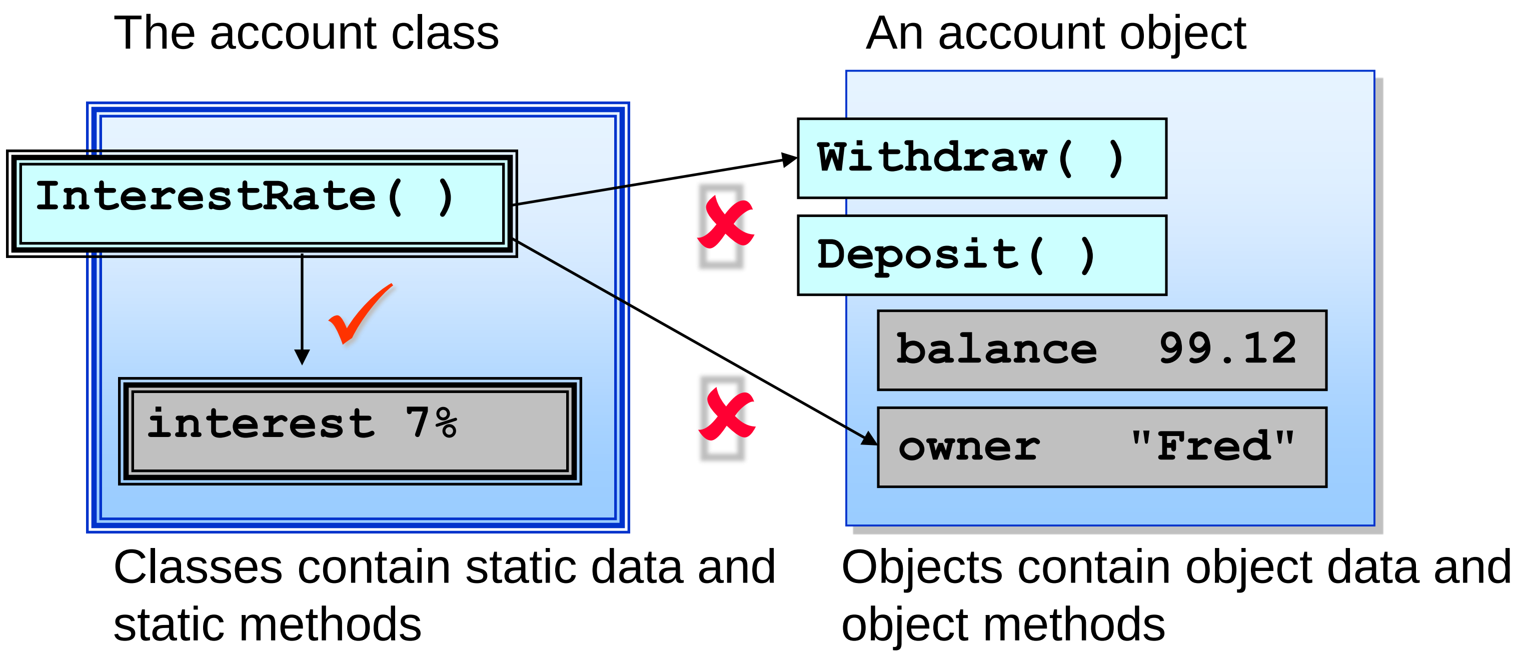
Using Static Data

- Static data describes information for *all* objects of a class
  - For example, suppose all accounts share the same interest rate. Storing the interest rate in every account would be a bad idea. Why?



Using Static Methods

- Static methods can only access static data
  - A static method is called on the class, not the object



Defining Simple Classes

- Data and methods together inside a class
- Methods are public, data is private

```
class BankAccount
{
    public void Withdraw(decimal amount)
    { ... }
    public void Deposit(decimal amount)
    { ... }
    private decimal balance;
    private string name;
}
```

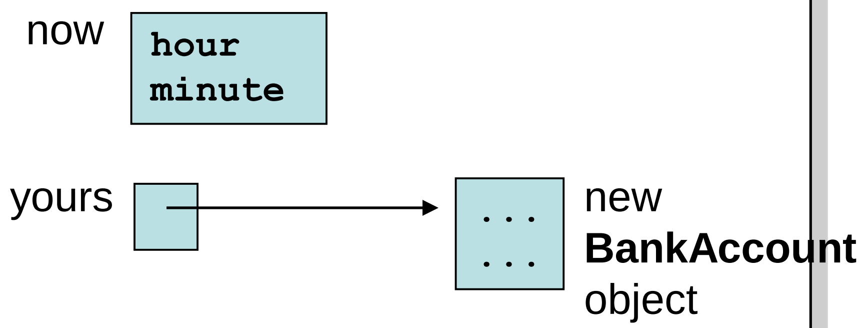
Public methods describe accessible behaviour

Private fields describe inaccessible state

Instantiating New Objects

- Declaring a class variable does not create an object
  - Use the **new** operator to create an object

```
class Program
{
    static void Main( )
    {
        Time now;
        now.hour = 11;
        BankAccount yours = new BankAccount( );
        yours.Deposit(999999M);
    }
}
```



Using the this Keyword

- The this keyword refers to the object used to call the method
  - Useful when identifiers from different scopes clash

```
class BankAccount
{
    ...
    public void SetName(string name)
    {
        this.name = name;
    }
    private string name;
}
```

If this statement were
name = name;
What would happen?

Constructors

- **Constructors** are *special methods* used to assign initial values of the fields in an object
  - Executed when an object of a given type is being created
  - **Have the same name as the class** that holds them
  - **Do not return a value**
- A class may have **several** constructors with different set of parameters

Constructors (2)

- Constructor is invoked by the new operator

```
<instance_name> = new <class_name>(<parameters>)
```

- Examples:

```
DateTime dt = new DateTime(2009, 12, 30);
```

```
DateTime dt = new DateTime(2009, 12, 30, 12, 33, 59);
```

```
Int32 value = new Int32(1024);
```

Parameterless Constructors

- The constructor without parameters is called default constructor
- Example:
  – Creating an object for generating random numbers with a default seed

```
using System;
...
Random randomGenerator = new Random();
```

Parameterless constructor call

The class `System.Random` provides generation of pseudo-random numbers

Constructor With Parameters

- Example
  - Creating objects for generating random values with specified initial seeds

```
using System;
...
Random randomGenerator1 = new Random(123);
Console.WriteLine(randomGenerator1.Next());
// 2114319875

Random randomGenerator2 = new Random(456);
Console.WriteLine(randomGenerator2.Next(50));
// 47
```

Namespaces

Organizing Classes Logically into Namespaces

What is a Namespace?

- Namespaces are used to organize the source code into more logical and manageable way
- Namespaces can contain
  - Definitions of classes, structures, interfaces and other types and other namespaces
- Namespaces can contain other namespaces
- For example:
  - `System` namespace contains `Data` namespace
  - The name of the nested namespace is `System.Data`

Full Class Names

- A full name of a class is the name of the class preceded by the name of its namespace

```
<namespace_name>.<class_name>
```

- Example:
  - `Array` class, defined in the `System` namespace
  - The full name of the class is `System.Array`

Including Namespaces

- The using directive in C#:

```
using <namespace_name>
```

- Allows using types in a namespace, without specifying their full name

Example:

```
using System;
DateTime date;
```

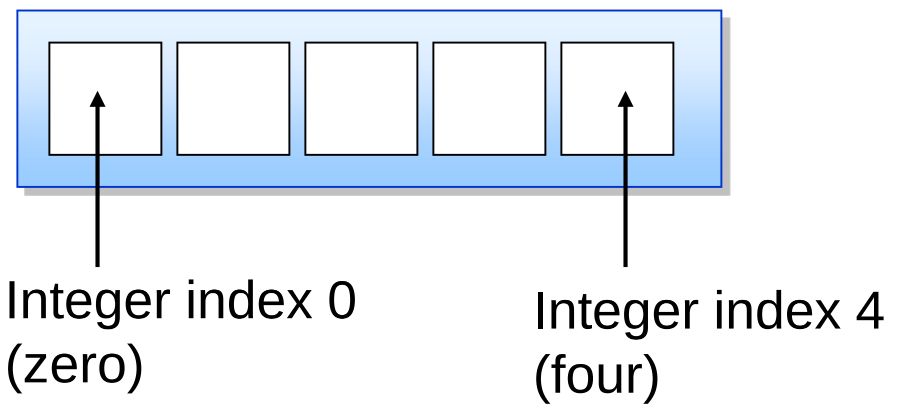
instead of

```
System.DateTime date;
```

ARRAYS AND COLLECTIONS

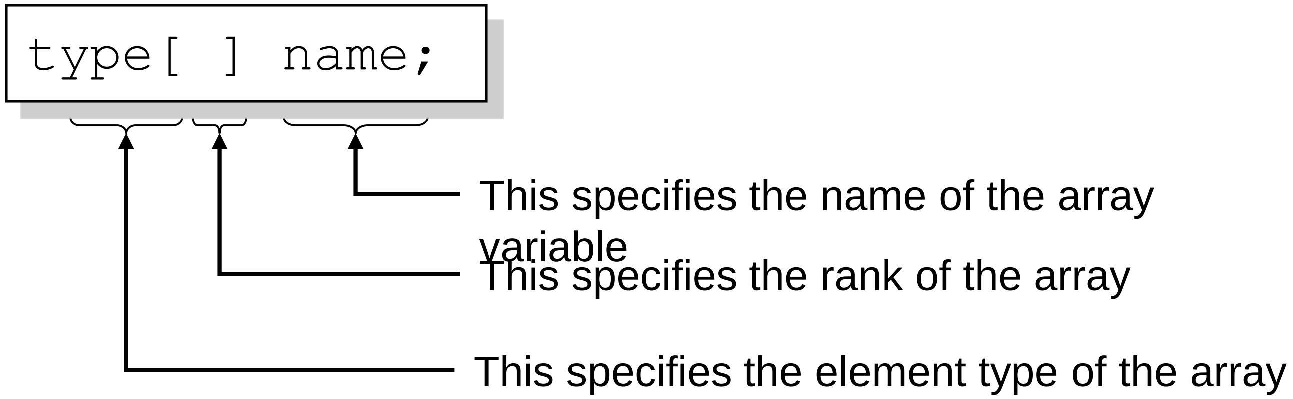
What Is an Array?

- An array is a sequence of elements
  - All elements in an array have the same type
  - Structs can have elements of different types
  - Individual elements are accessed using integer indexes



Array Notation in C#

- You declare an array variable by specifying:
  - The element type of the array
  - The rank of the array
  - The name of the variable

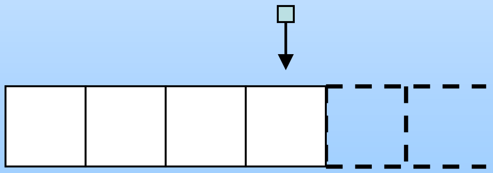

type[ ] name;
This specifies the name of the array variable
This specifies the rank of the array
This specifies the element type of the array

Array Rank

- Rank is also known as the array dimension
- The number of indexes associated with each element

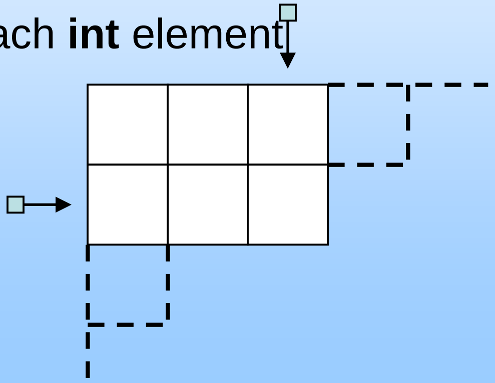
```
long[ ] row;
```

Rank 1: One-dimensional
Single index associates with each **long** element



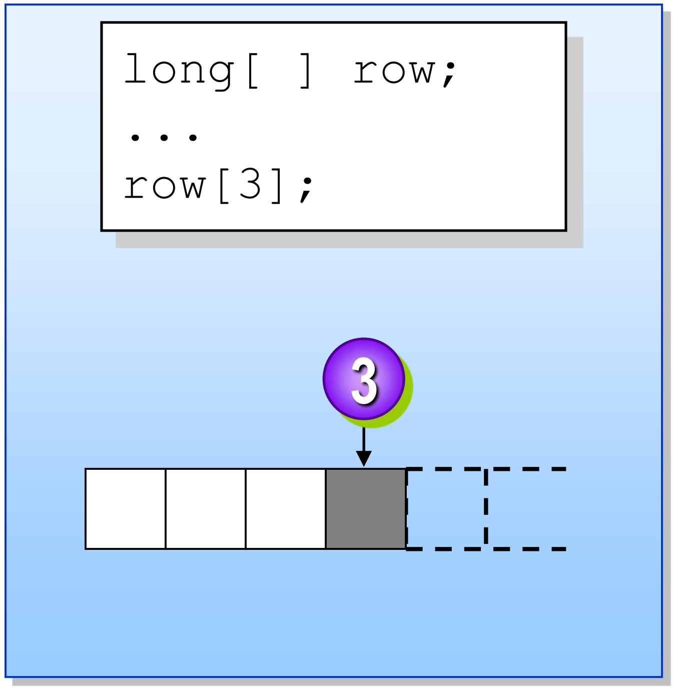
```
int[,] grid;
```

Rank 2: Two-dimensional
Two indexes associate with each **int** element

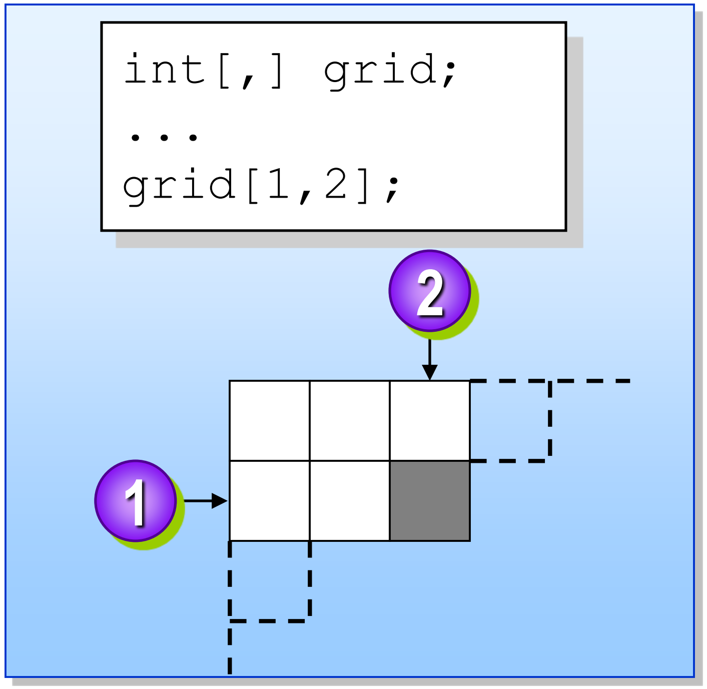


Accessing Array Elements

- Supply an integer index for each rank
  - Indexes are zero-based



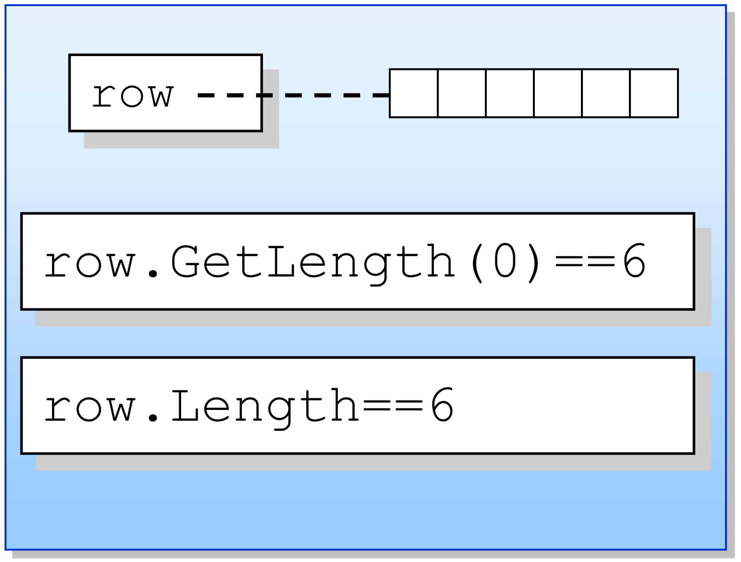
```
long[ ] row;
...
row[3];
```

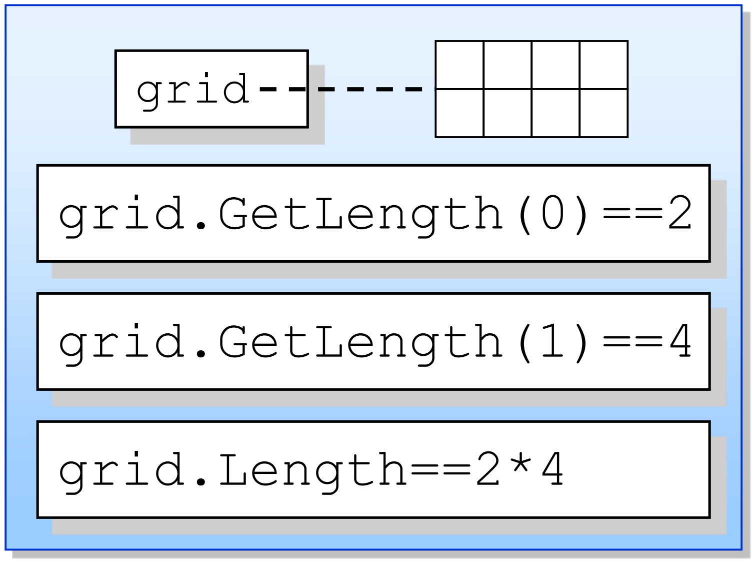


```
int[,] grid;
...
grid[1,2];
```

Checking Array Bounds

- All array access attempts are bounds checked
  - A bad index throws an IndexOutOfRangeException
  - Use the **Length** property and the **GetLength** method





Comparing Arrays to Collections

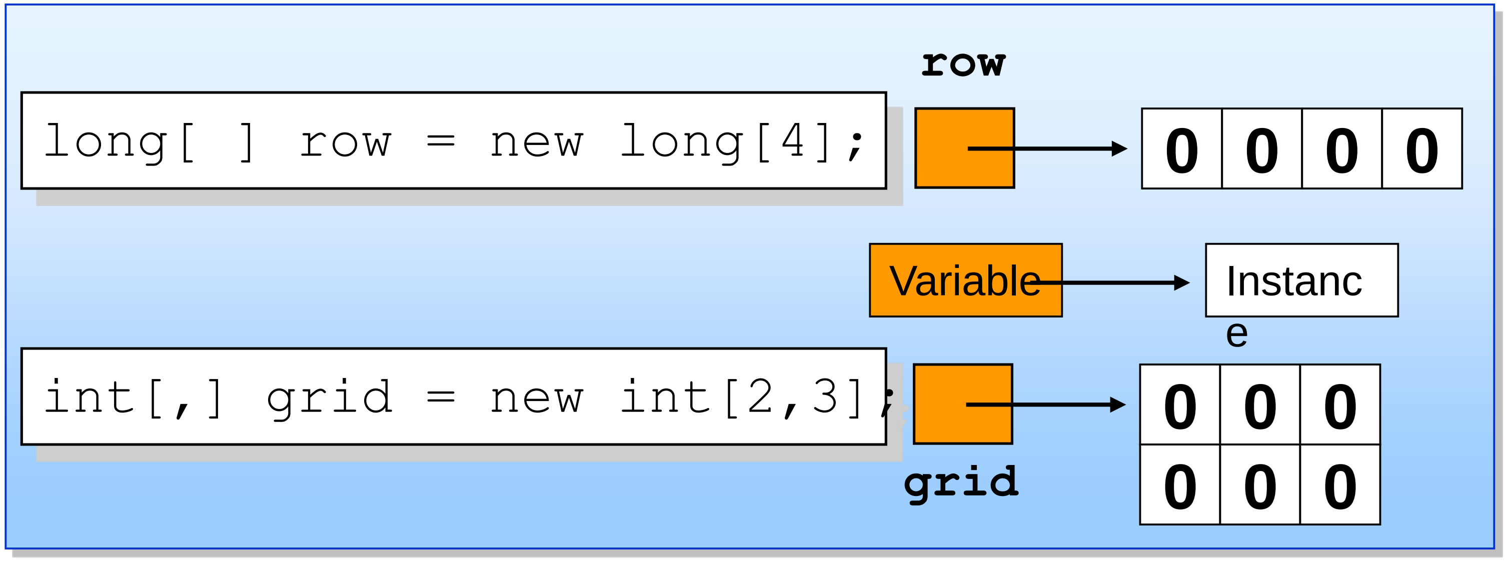
- An array cannot resize itself when full
  - A collection class, such as ArrayList, can resize
- An array is intended to store elements of one type
  - A collection is designed to store elements of different types
- Elements of an array cannot have read-only access
  - A collection can have read-only access
- In general, arrays are faster but less flexible
  - Collections are slightly slower but more flexible

◆ Creating Arrays

- Creating Array Instances
- Initializing Array Elements
- Initializing Multidimensional Array Elements
- Creating a Computed Size Array
- Copying Array Variables

Creating Array Instances

- Declaring an array variable does not create an array!
  - You must use **new** to explicitly create the array instance
  - Array elements have an implicit default value of zero

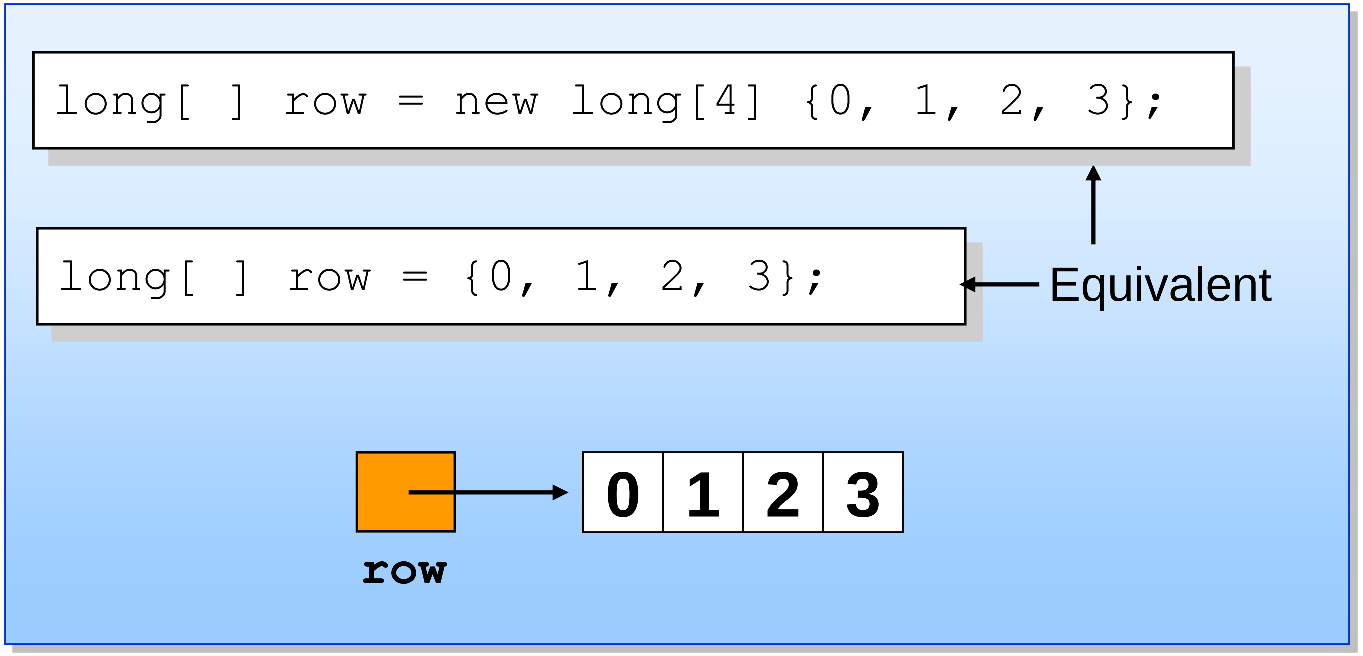


```
long[ ] row = new long[4];
```

```
int[,] grid = new int[2,3];
```

Initializing Array Elements

- The elements of an array can be explicitly initialized
  - You can use a convenient shorthand



```
long[ ] row = new long[4] {0, 1, 2, 3};
```

```
long[ ] row = {0, 1, 2, 3};
```

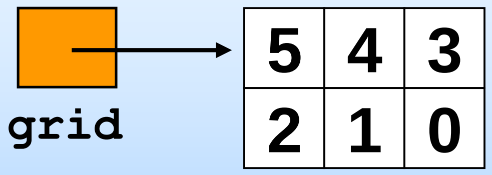
Initializing Multidimensional Array Elements

- You can also initialize multidimensional array elements
  - All elements must be specified

```
int[,] grid = {
          {5, 4, 3},
          {2, 1, 0}
};
```

Implicitly a new int[2,3] array

✓



```
int[,] grid = {
          {5, 4, 3},
          {2, 1    }
};
```

✗

Creating a Computed Size Array

- The array size does not need to be a compile-time constant
  - Any valid integer expression will work
  - Accessing elements is equally fast in all cases
    - Array size specified by compile-time integer constant:

```
long[ ] row = new long[4];
```

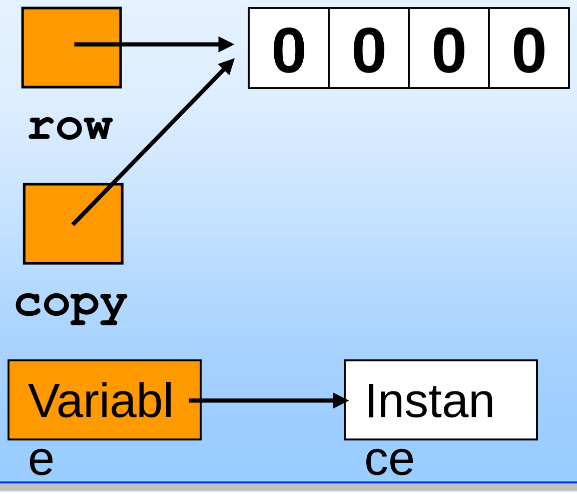
- Array size specified by run-time integer value:

```
string s = Console.ReadLine();
int size = int.Parse(s);
long[ ] row = new long[size];
```

Copying Array Variables

- Copying an array variable copies the array variable only
  - It does not copy the array instance
  - Two array variables can refer to the same array instance

```
long[ ] row = new long[4];
long[ ] copy = row;
...
row[0]++;
long value = copy[0];
Console.WriteLine(value);
```

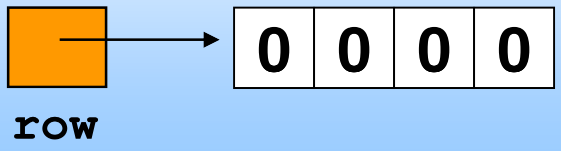


◆ Using Arrays

- Array Properties
- Array Methods
- Returning Arrays from Methods
- Passing Arrays as Parameters
- Command-Line Arguments
- Demonstration: Arguments for Main
- Using Arrays with foreach

Array Properties

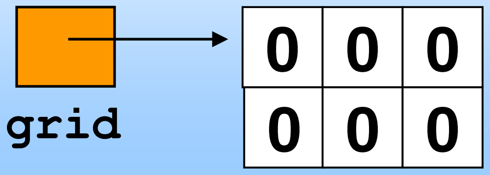
```
long[ ] row = new long[4];
```



```
row.Rank
```
1

```
row.Length
```
4

```
int[,] grid = new int[2,3];
```



```
grid.Rank
```
2

```
grid.Length
```
6

Array Methods

- Commonly used methods
  - **Sort** – sorts the elements in an array of rank 1
  - **Clear** – sets a range of elements to zero or **null**
  - **Clone** – creates a copy of the array
  - **GetLength** – returns the length of a given dimension
  - **IndexOf** – returns the index of the first occurrence of a value

Working with the ArrayList Class

- The `ArrayList` is a class defined in the `System.Collections` namespace.
- It represents a dynamically sized array of objects.
- Features of `ArrayList` class:
  - Allows you to add, remove, insert, and sort the items in it.
  - Contains items in the order of addition.

Working with the ArrayList Class

- Consists of an index identifier assigned to its items.
- Enables you to retrieve items in any order by means of their associated index numbers.

Introducing Generics

- Benefits of using Generics:
  - Allows you to work with any data type.
  - Provides many of the advantages of the strongly typed collections.
  - Increases code performance by reducing the number of required casting operations.

Introducing Generics

Generic Class	Nongeneric Class	Description
List<T>	*ArrayList, StringCollection*	A dynamically resizable list of items
Dictionary<T,U>	*HashTable, ListDictionary, HybridDictionary, OrderedDictionary, NameValueCollection, StringDictionary*	A generic collection of name-value pairs
Queue<T>	*Queue*	A generic implementation of a FIFO list

Introducing Generics

Generic Class	Nongeneric Class	Description
Stack<T>	*Stack*	A generic implementation of a LIFO list
SortedList<T,U>	*SortedList*	A generic implementation of a sorted list of generic name/value pairs
Comparer<T>	*Comparer*	Compares two generic objects for equality

Introducing Generics

Generic Class	**Nongeneric Class**	**Description**
LinkedList<T>	*N/A*	A generic implementation of a doubly linked list
Collection<T>	*CollectionBase*	Provides the basis for a generic collection
ReadOnlyCollection<T>	*ReadOnlyCollectionBase*	A generic implementation of a set of read-only items

Lists

- The most common sort of collection is a List<T>. Once you create a List<T> object, it's easy to add an item, remove an item from any location in the list, peek at an item, and even move an item from one place in the list to another.
- Here's how a list works:
  - First you create a new instance of List<T>
  - You need to specify the type of object or value that the list will hold by putting it in angle brackets <> when you use the new keyword to create it.
  - List<Card> cards = new List<Card>();
- The <T> at the end of List<T> means it's *generic*.
- The T gets replaced with a type—so List<int> just means a List of ints.

Lists

- A List **resizes dynamically** to whatever size is needed.
- To put something into a List, use **Add().**
- To remove something from a List, use **Remove().**
- You can remove objects using their **index** number using **RemoveAt().**
- To find out where something is (and if it is) in a List, use **IndexOf().**
- To get the number of elements in a List, use the **Count** property.
- You can use the **Contains()** method to find out if a particular object is in a List.

Files

- Contain a sequence of bytes that represent information
- Persisting objects to files is important because
  - It eliminates need to reenter data
  - It allows information to be shared between people, or applications
- File format
  - Describes how file is organized
  - Determined by software used to create the file

Files (continued)

- File extension
  - Does not dictate what is contained within the file
- To identify a file’s location, you need
  - The drive letter of the disk
  - The folders and subfolders in which the file is contained
  - The filename
  - The file extension

The .NET System.IO Namespace

- Classes can be grouped into two functions
  - Classes that manipulate the file system
  - Classes for saving and retrieving data to and from external storage devices

System.IO.Directory

- Directory class
  - Exposes methods that can be used to create, delete, move, and list subfolders and files
- Because methods are defined as *static*
  - They must be called using class name and not using an instance variable name

System.IO.File

- File class
  - Exposes methods that can be used to copy, delete, and move files
- File methods
  - Defined as *static*
  - Must be called using the class name and not using an instance variable name

System.IO.Path

- Methods in the Path class
  - Do not actually manipulate folders and files
  - Can be used to manipulate the strings that define a folder’s or file’s path

Types of Files

- Data
  - Can be stored in a file as plain-text strings or as binary data
- Text files
  - Usually have a file extension of .txt
- Saving data to a binary data file
  - Executes more efficiently than saving data to a text file

Persistence Using Sequential Text Files

- .NET *StreamReader* and *StreamWriter* classes
  - Used to read and write sequential text files with C#
- Hard-coded paths
  - Not a good idea because a path that exists on your computer might not another computer
  - Can result in exceptions

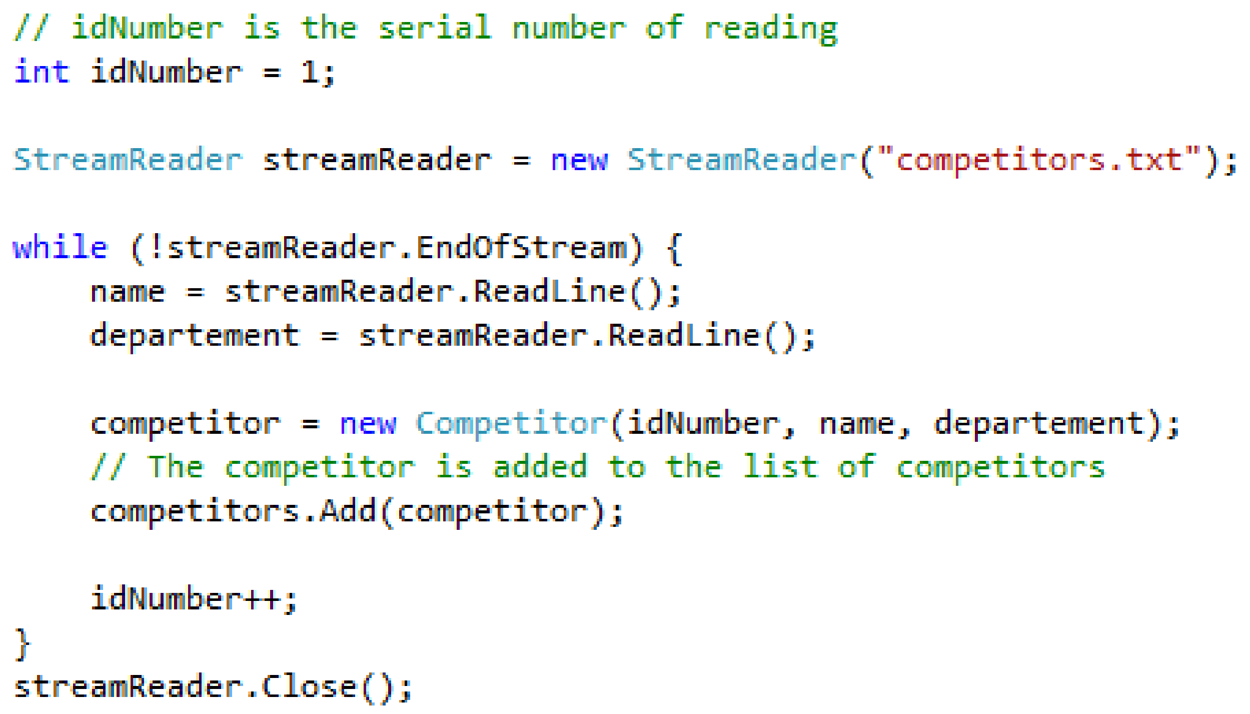
Fixed-Width Text Files

- Format
  - File contains zero or many records
  - Each record contains information about a single entity
  - Each record normally is terminated by a CRLF
- Contents of each field in each record is always the same number of characters
- Nothing in the file delimits where a field begins and ends

Fixed-Width Text Files (continued)

- Processing
  - First open the file
  - For reading or writing records
    - Use the *ReadLine* method of the *StreamReader* class or the *WriteLine* method of the *StreamWriter* class
  - File is closed when you are finished using it
- Parsing input records and formatting output records
  - Requires more tedious coding than with tab-delimited records

Example



```
// idNumber is the serial number of reading
int idNumber = 1;

StreamReader streamReader = new StreamReader("competitors.txt");

while (!streamReader.EndOfStream) {
    name = streamReader.ReadLine();
    departement = streamReader.ReadLine();

    competitor = new Competitor(idNumber, name, departement);
    // The competitor is added to the list of competitors
    competitors.Add(competitor);

    idNumber++;
}
streamReader.Close();
```