.NET Framework Fundamentals

Compiled by Pakita Shamoi

C#

Very similar to Java

70% Java, 10% C++, 5% Visual Basic, 15% new

As in Java

- Object-orientation (single inheritance)
- Interfaces
- Exceptions
- Threads
- Namespaces (like Packages)
- Strong typing
- Garbage Collection
- Reflection
- Dynamic loading of code
- ...

As in C++

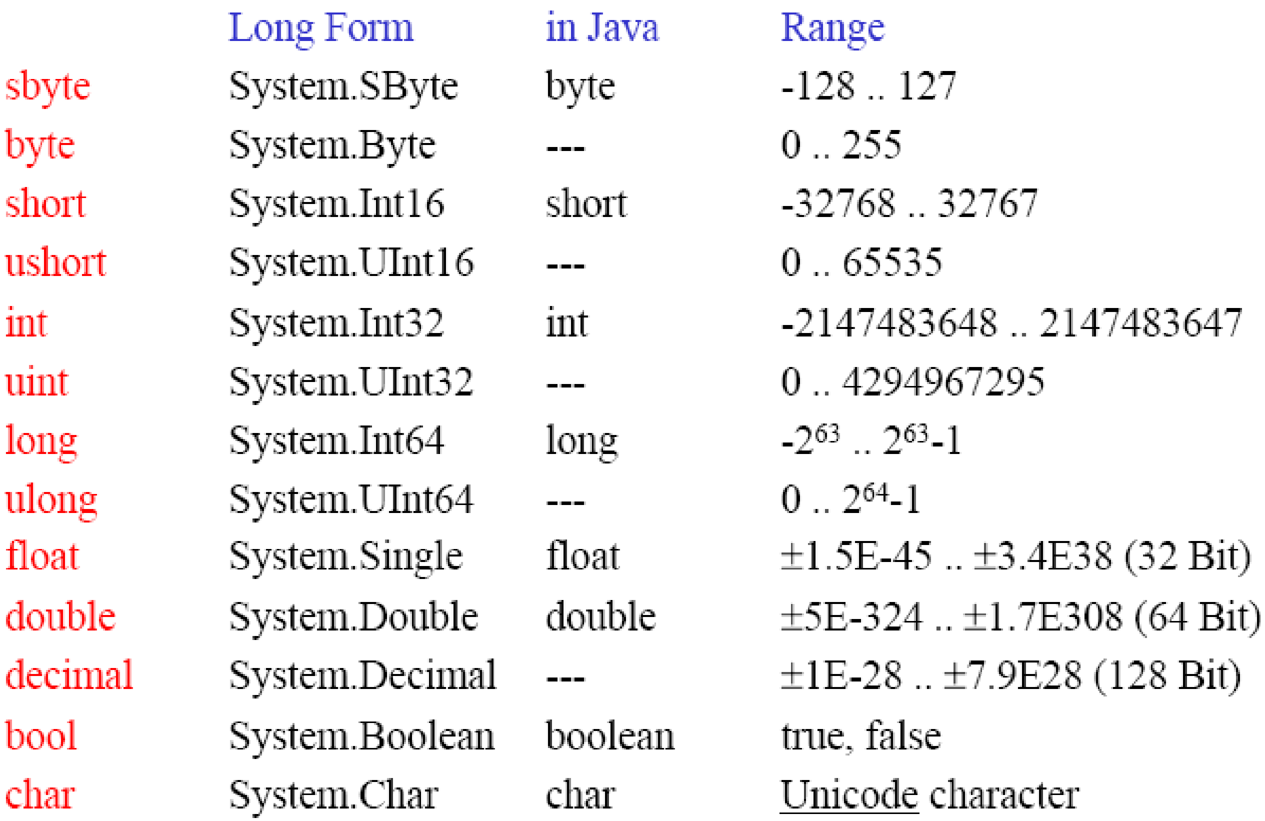
- (Operator) Overloading
- Pointer arithmetic in unsafe code
- Some syntactic details

Value types

- Value types are variables that contain their data directly instead of containing a reference to the data
- Instances of value types are stored in an area of memory called the *stack*
- There are three general value types:
  - Built-in types
  - User-defined types
  - Enumerations
- When you assign between value-type variables, the data is copied from one variable to the other and stored in two different locations on the stack.

Built-in value types

- Base types provided with the .NET Framework
- You choose a numeric type based on the size of the values you expect to work with and the level of precision you require.



	Long Form	in Java	Range
sbyte	System.SByte	byte	-128 .. 127
byte	System.Byte	---	0 .. 255
short	System.Int16	short	-32768 .. 32767
ushort	System.UInt16	---	0 .. 65535
int	System.Int32	int	-2147483648 .. 2147483647
uint	System.UInt32	---	0 .. 4294967295
long	System.Int64	long	-2^{63} .. 2^{63}-1
ulong	System.UInt64	---	0 .. 2^{64}-1
float	System.Single	float	±1.5E-45 .. ±3.4E38 (32 Bit)
double	System.Double	double	±5E-324 .. ±1.7E308 (64 Bit)
decimal	System.Decimal	---	±1E-28 .. ±7.9E28 (128 Bit)
bool	System.Boolean	boolean	true, false
char	System.Char	char	Unicode character

VTs function as objects - you can call methods on them. In fact, it is common to use the *ToString* method when displaying values as text.

Declaring value types

- Implicit constructor - no need for new keyword

- You should always explicitly initialize the variable within the declaration :

```
 bool b = false;
```

- Declare the variable as nullable if you want to be able to determine whether a value has not been assigned. When it can be useful?

```
Nullable<bool> b = null;
// Shorthand notation
bool? b = null;
```

- Use Has-Value to detect whether or not a value has been set:

```
if (b.HasValue)Console.WriteLine(b);
else Console.WriteLine("b is not set.");
```

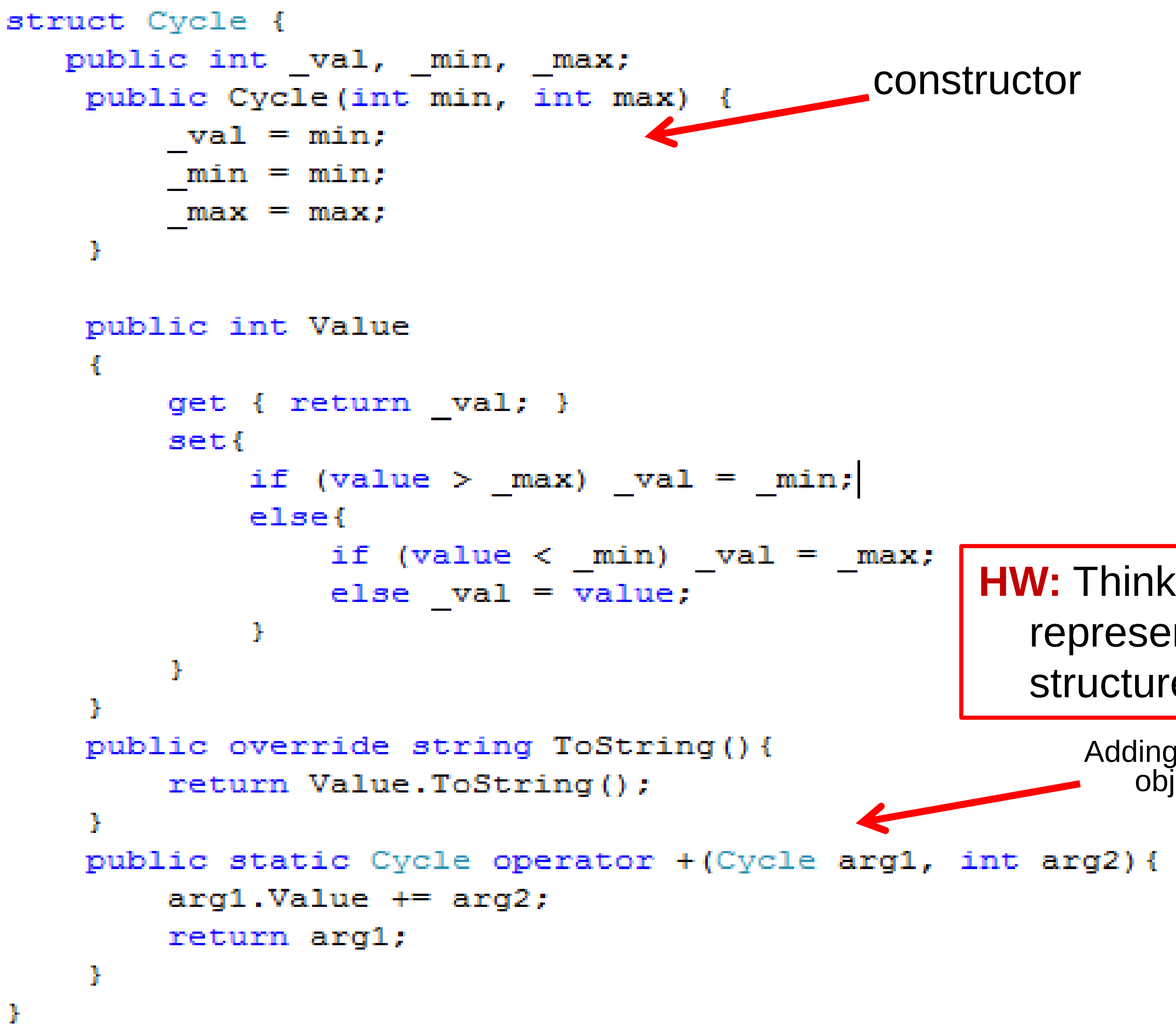
User-defined types

- Also called structures or simply structs
- Behave nearly identical to classes
- Structures are a composite of other types that make it easier to work with related data.

- The simplest example is *System.Drawing.Point*

```
    Point p = new Point(20, 30);
     // Move point diagonally
     p.Offset(-1, -1);
Console.WriteLine("Point X {0}, Y {1}", p.X, p.Y);
```

Example



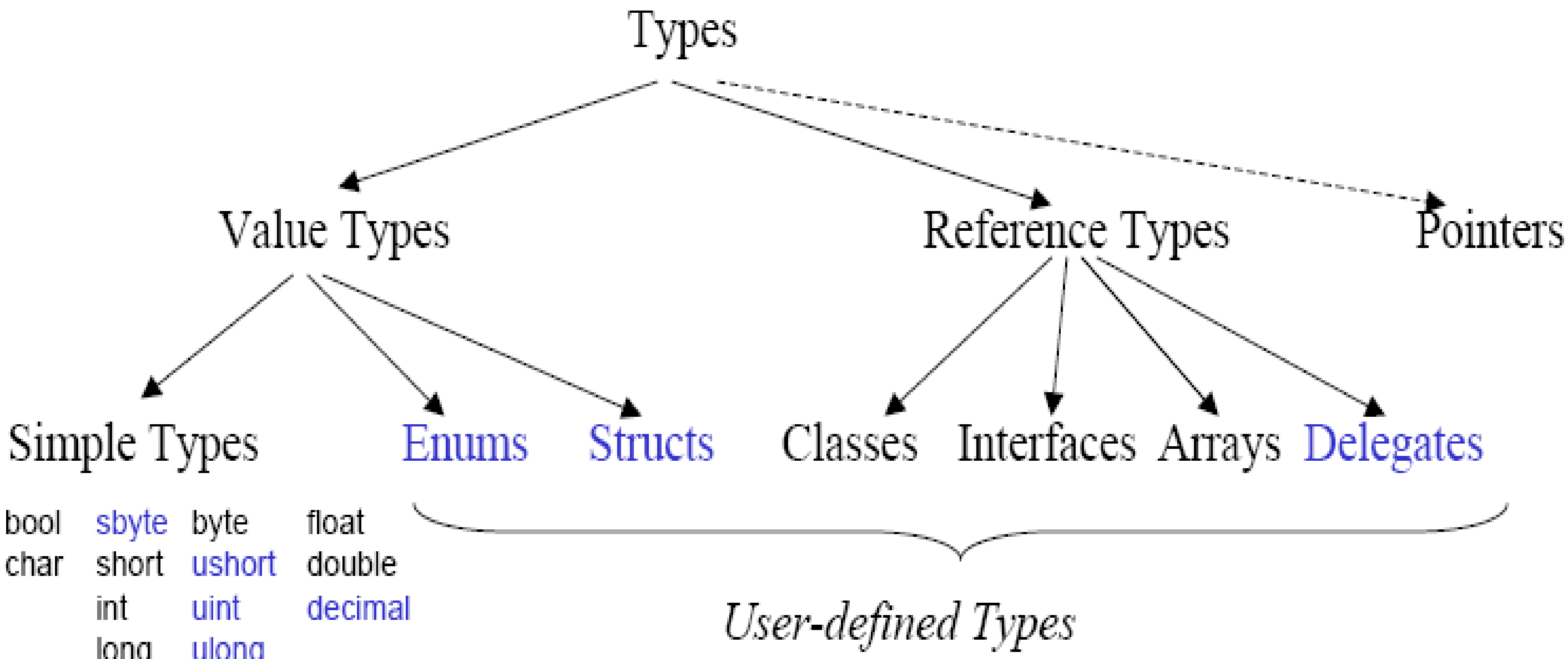
```
struct Cycle {
   public int _val, _min, _max;
    public Cycle(int min, int max) {
        _val = min;
        _min = min;
        _max = max;
    }

    public int Value
    {
        get { return _val; }
        set{
            if (value > _max) _val = _min;
            else{
                if (value < _min) _val = _max;
                else _val = value;
            }
        }
    }
    public override string ToString(){
        return Value.ToString();
    }
    public static Cycle operator +(Cycle arg1, int arg2){
        arg1.Value += arg2;
        return arg1;
    }
}
```

constructor

Adding number to cycle object

Let’s create a type that cycles through a set of integers between min and max values set by the constructor:

HW: Think, what can you represent by using this structure.



Types

Value Types

Reference Types

Pointers

Simple Types

Enums

Structs

Classes

Interfaces

Arrays

Delegates

bool sbyte byte float

char short ushort double

int uint decimal

long ulong

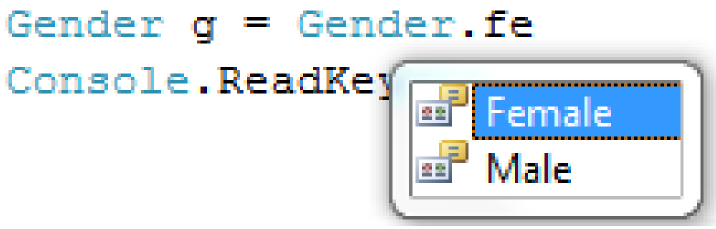
User-defined Types

All types are compatible with *object*

- can be assigned to variables of type *object*
- all operations of type *object* are applicable to them

Enumerations

- Enumerations are related symbols that have fixed values. Use enumerations to provide a limited set of choices for a value
- The purpose of enumerations is to simplify coding and improve code readability



```
Gender g = Gender.fe
Console.ReadKey
```

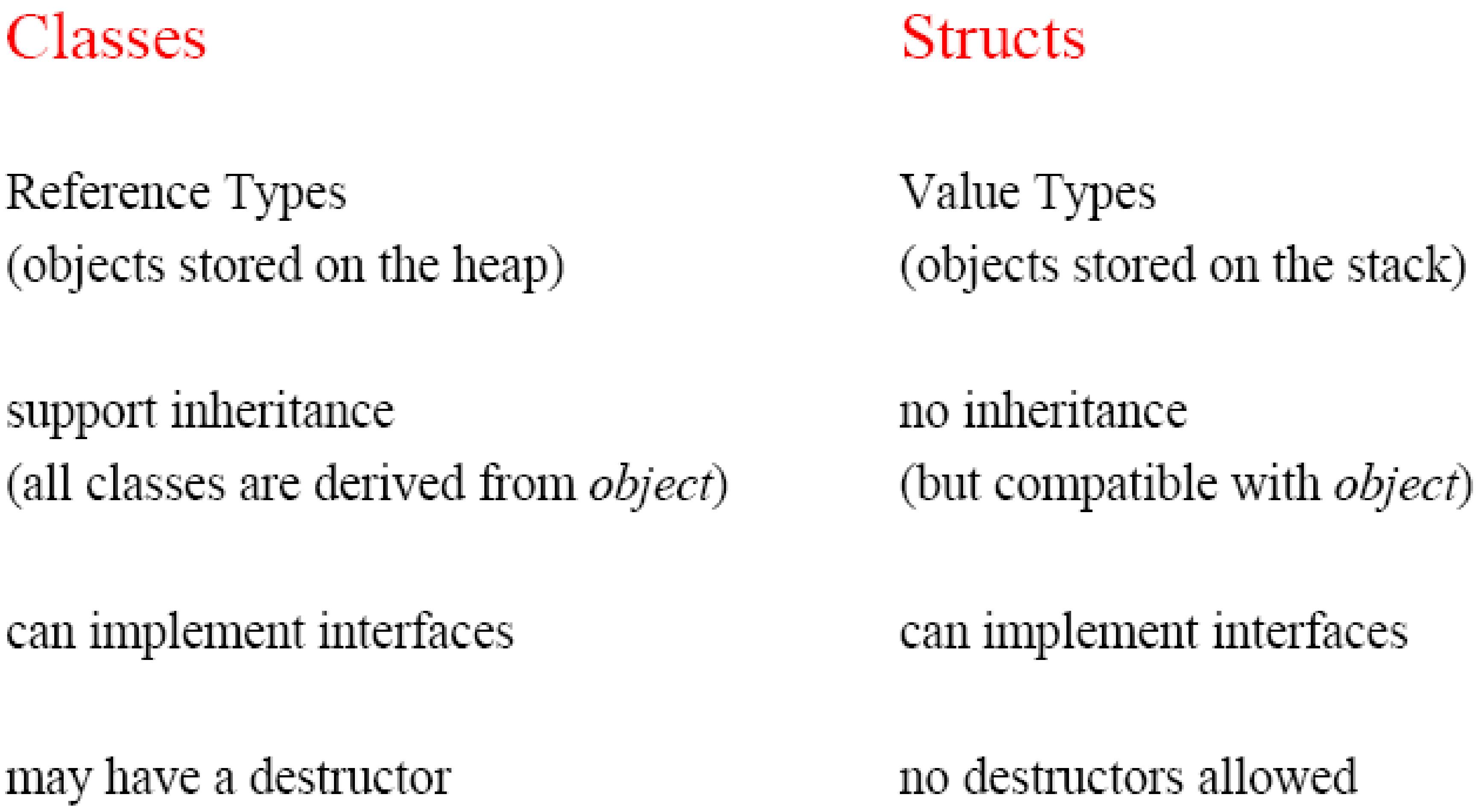
```
enum Gender { Male, Female};
```

Usage:

```
Gender g = Gender.Female;
Console.WriteLine(g); Console.WriteLine((int)g);
```

```
enum Color {red, blue, green}    // values: 0, 1, 2
enum Access {personal=1, group=2, all=4}
```

Classes vs. Structs

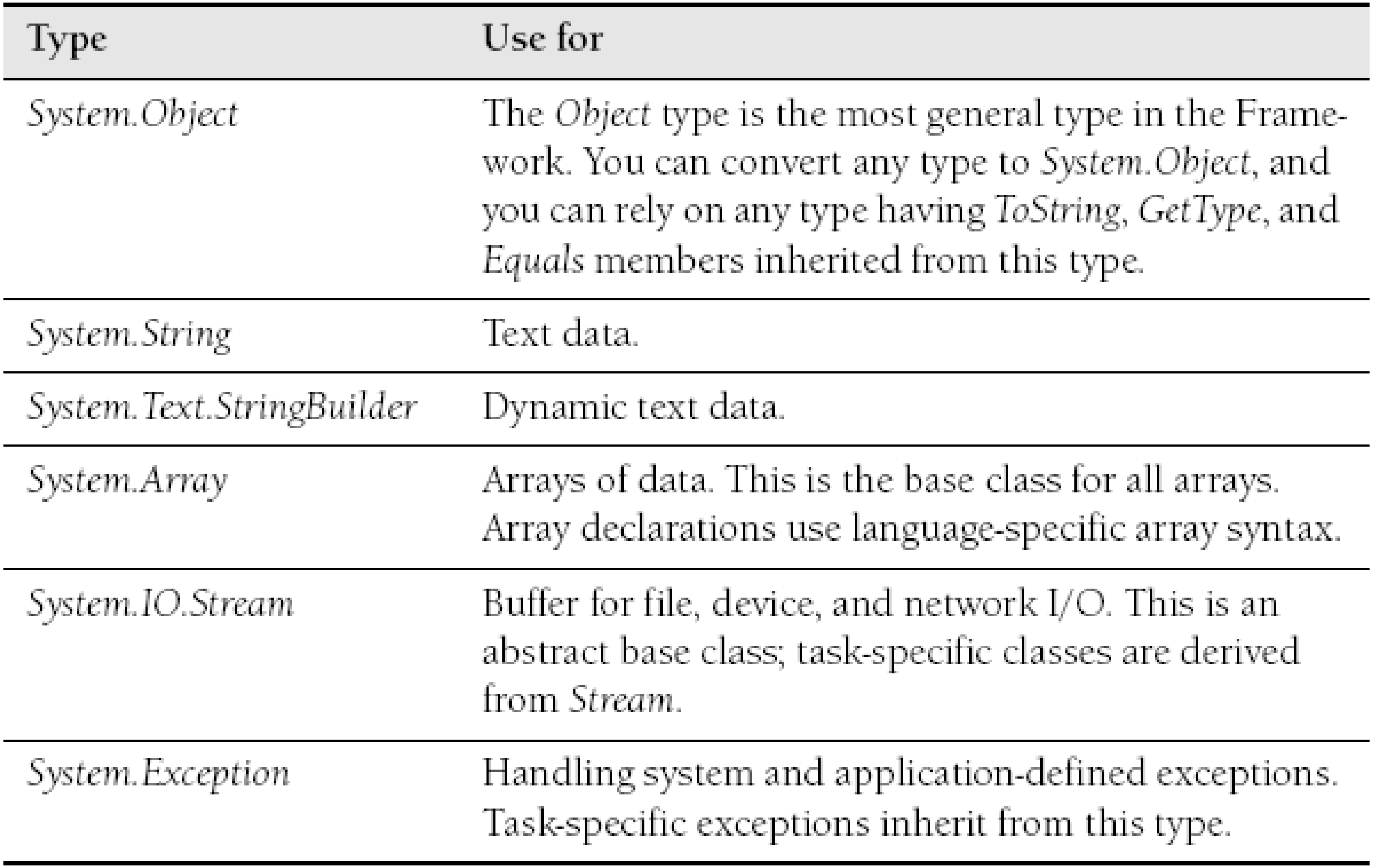


Classes	Structs
Reference Types (objects stored on the heap)	Value Types (objects stored on the stack)
support inheritance (all classes are derived from *object*)	no inheritance (but compatible with *object*)
can implement interfaces	can implement interfaces
may have a destructor	no destructors allowed

Reference types

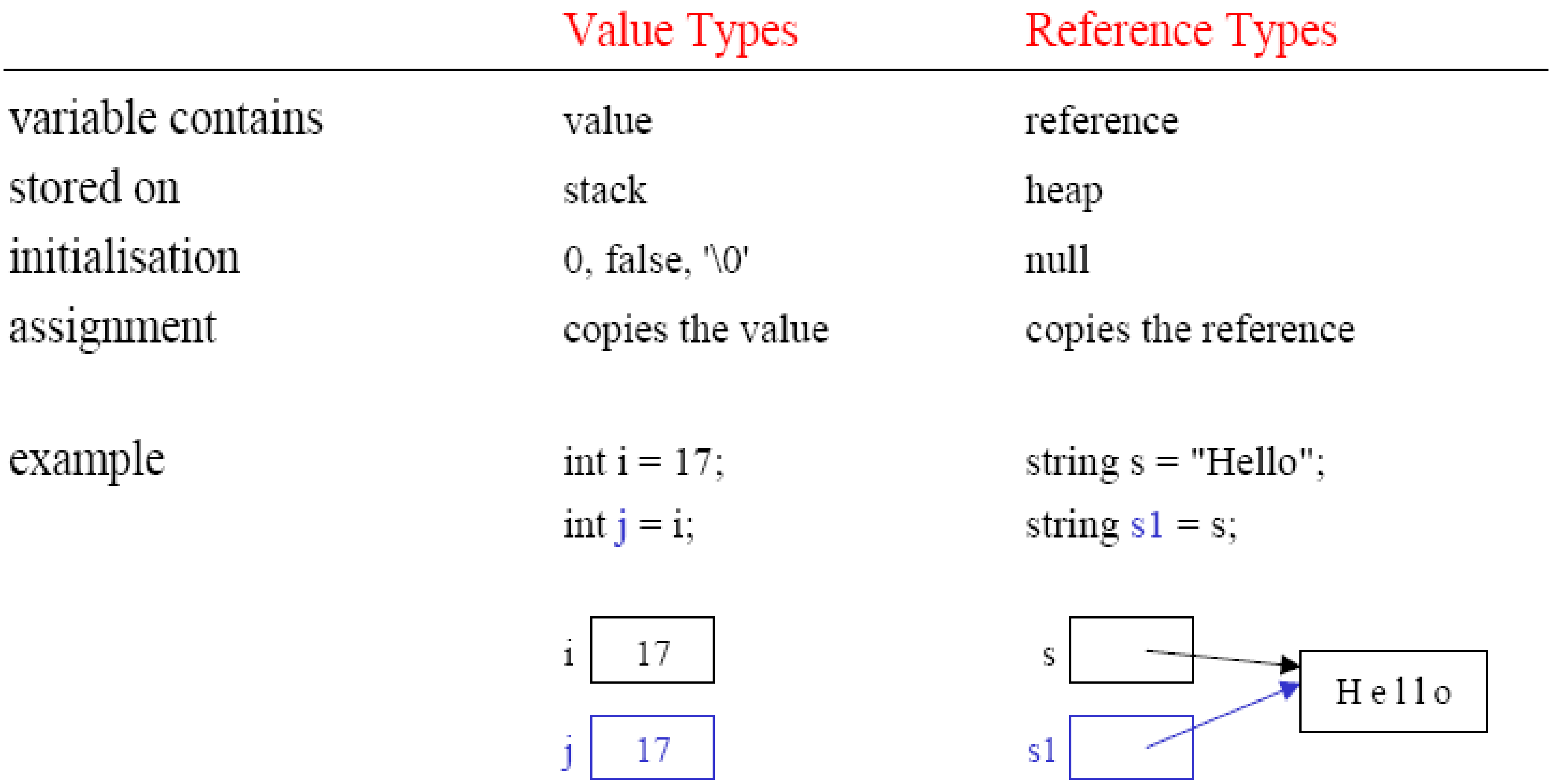
- Reference types store the address of their data, also known as a *pointer*, on the *stack.*
- The actual data that address refers to is stored in an area of memory called the *heap.*
- The runtime manages the memory used by the heap through a process called *garbage collection*
- Assigning a reference variable to another instance merely creates a second copy of the reference, which refers to the same memory location on the heap as the original variable

Common reference types



Type	Use for
System.Object	The *Object* type is the most general type in the Framework. You can convert any type to *System.Object*, and you can rely on any type having *ToString*, *GetType*, and *Equals* members inherited from this type.
System.String	Text data.
System.Text.StringBuilder	Dynamic text data.
System.Array	Arrays of data. This is the base class for all arrays. Array declarations use language-specific array syntax.
System.IO.Stream	Buffer for file, device, and network I/O. This is an abstract base class; task-specific classes are derived from *Stream*.
System.Exception	Handling system and application-defined exceptions. Task-specific exceptions inherit from this type.

Value types vs. reference types



	Value Types	Reference Types
variable contains	value	reference
stored on	stack	heap
initialisation	0, false, '\0'	null
assignment	copies the value	copies the reference

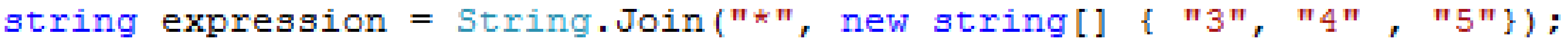
example	int i = 17;
	int j = i;
		string s = "Hello";
		string s1 = s;

i
17
j
17
s
s1
Hello

HW: To better understand the difference between value and reference types, pass through the example on page 18 (struct Number and class Number)

String

-
  ```
  string s = "this is some text to search";
   s = s.Replace("search", "replace");
  ```
- Strings of type *System.String* are immutable in .NET. That means any change to a string causes the runtime to create a new string and abandon the old one. To avoid this use:
  - String class's Concat, Join, or Format methods to join multiple items in a single statement.
  - Use the StringBuilder class to create dynamic (mutable) strings.



```
string expression = String.Join("*", new string[] { "3", "4" , "5"});
```

Another important feature of the String class is that it overrides operators from System.Object – "+", "!=", "=", "=="

Arrays

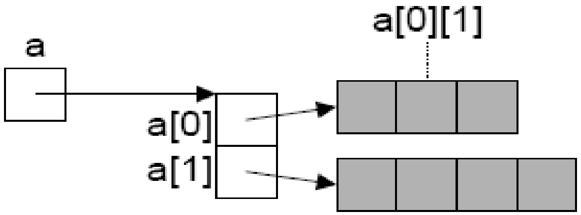
- 1-dimensional

```
int[] a = new int[3];
int[] b = new int[] {3, 4, 5};
int[] c = {3, 4, 5};
SomeClass[] d = new SomeClass[10];   // Array of references
SomeStruct[] e = new SomeStruct[10];  // Array of values (directly in the array)

int len = a.Length;   // number of elements in a
```

- 2-dimensional

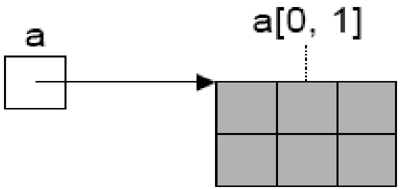
Jagged (like in Java)

```
int[][] a = new int[2][];
a[0] = new int[3];
a[1] = new int[4];

int x = a[0][1];
int len = a.Length;  // 2
len = a[0].Length;   // 3
```



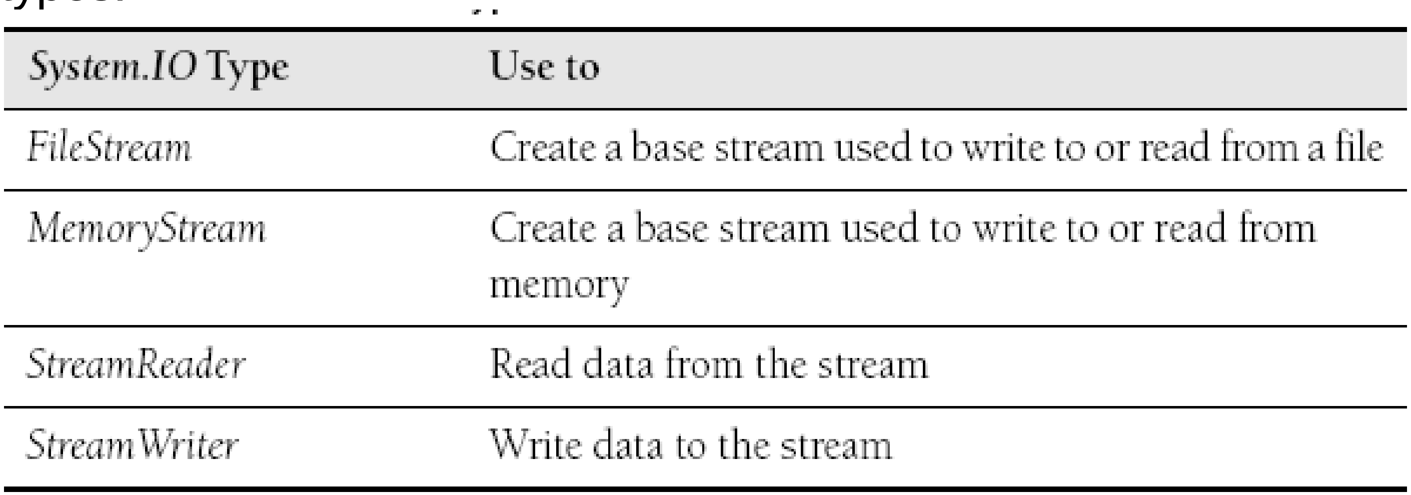
Rectangular (more compact, more efficient access)

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

int x = a[0, 1];
int len = a.Length;       // 6
len = a.GetLength(0); // 2
len = a.GetLength(1); // 3
```

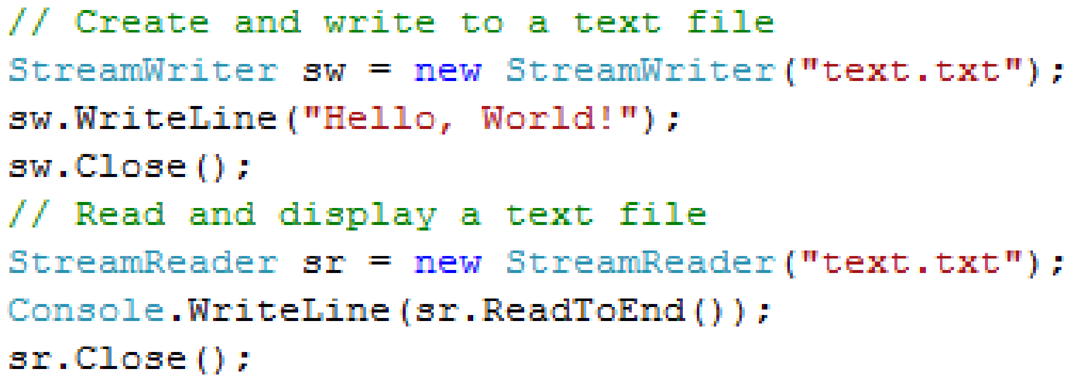


Streams

- Provide means for reading from and writing to the disk and communicating across the network.
- The *System.IO.Stream* type is the base type for all task-specific stream types.



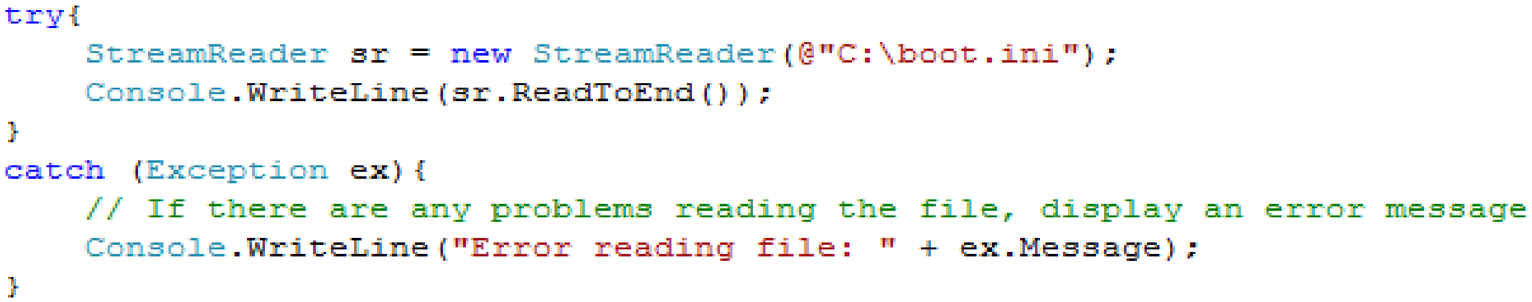
System.IO Type	Use to
FileStream	Create a base stream used to write to or read from a file
MemoryStream	Create a base stream used to write to or read from memory
StreamReader	Read data from the stream
StreamWriter	Write data to the stream



```
// Create and write to a text file
StreamWriter sw = new StreamWriter("text.txt");
sw.WriteLine("Hello, World!");
sw.Close();
// Read and display a text file
StreamReader sr = new StreamReader("text.txt");
Console.WriteLine(sr.ReadToEnd());
sr.Close();
```

Exceptions

- Exceptions are unexpected events that interrupt normal execution of the application.
  - For example, if you are reading a large text file from a removable disk and the user removes the disk, the runtime will throw an exception.



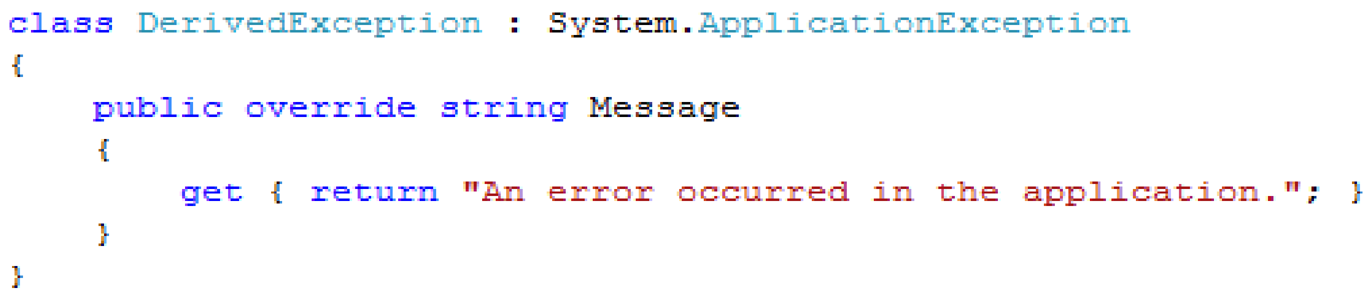
```
try{
    StreamReader sr = new StreamReader(@"C:\boot.ini");
    Console.WriteLine(sr.ReadToEnd());
}
catch (Exception ex){
    // If there are any problems reading the file, display an error message
    Console.WriteLine("Error reading file: " + ex.Message);
}
```

• In addition to the base Exception class, there are other exception classes to describe different types of events, all derived from System.SystemException. Having multiple exception classes allows you to respond differently to differene errors.

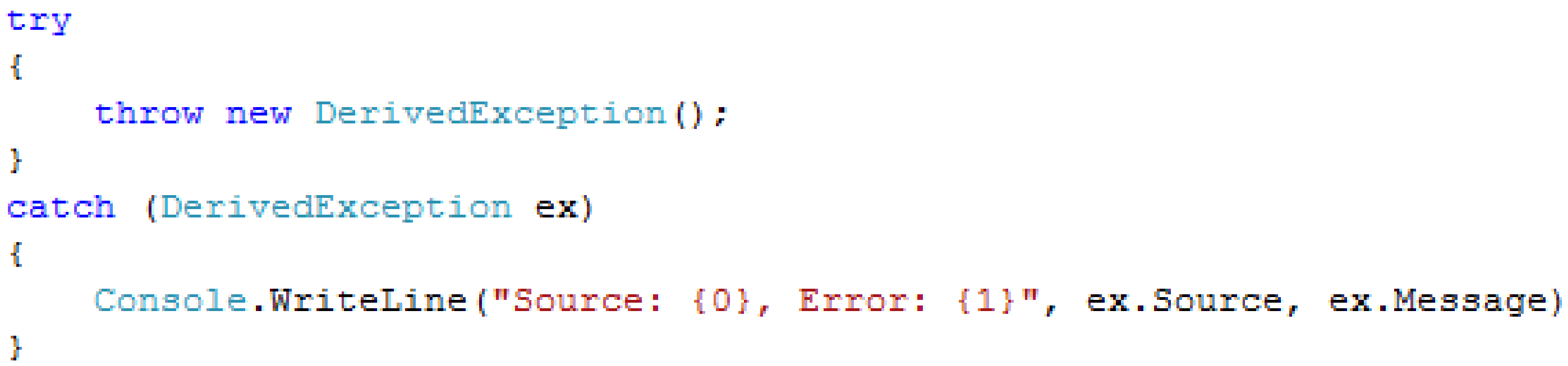
• The Finally block runs after the Try block and any Catch blocks have finished executing, whether or not an exception was thrown. Therefore, you should use a Finally block to close any streams or clean up any other objects that might be left open if an exception occurs.

Inheritance

- Use inheritance to create new classes from existing ones
- You can easily create a custom exception class by inheriting from *System.ApplicationException:*



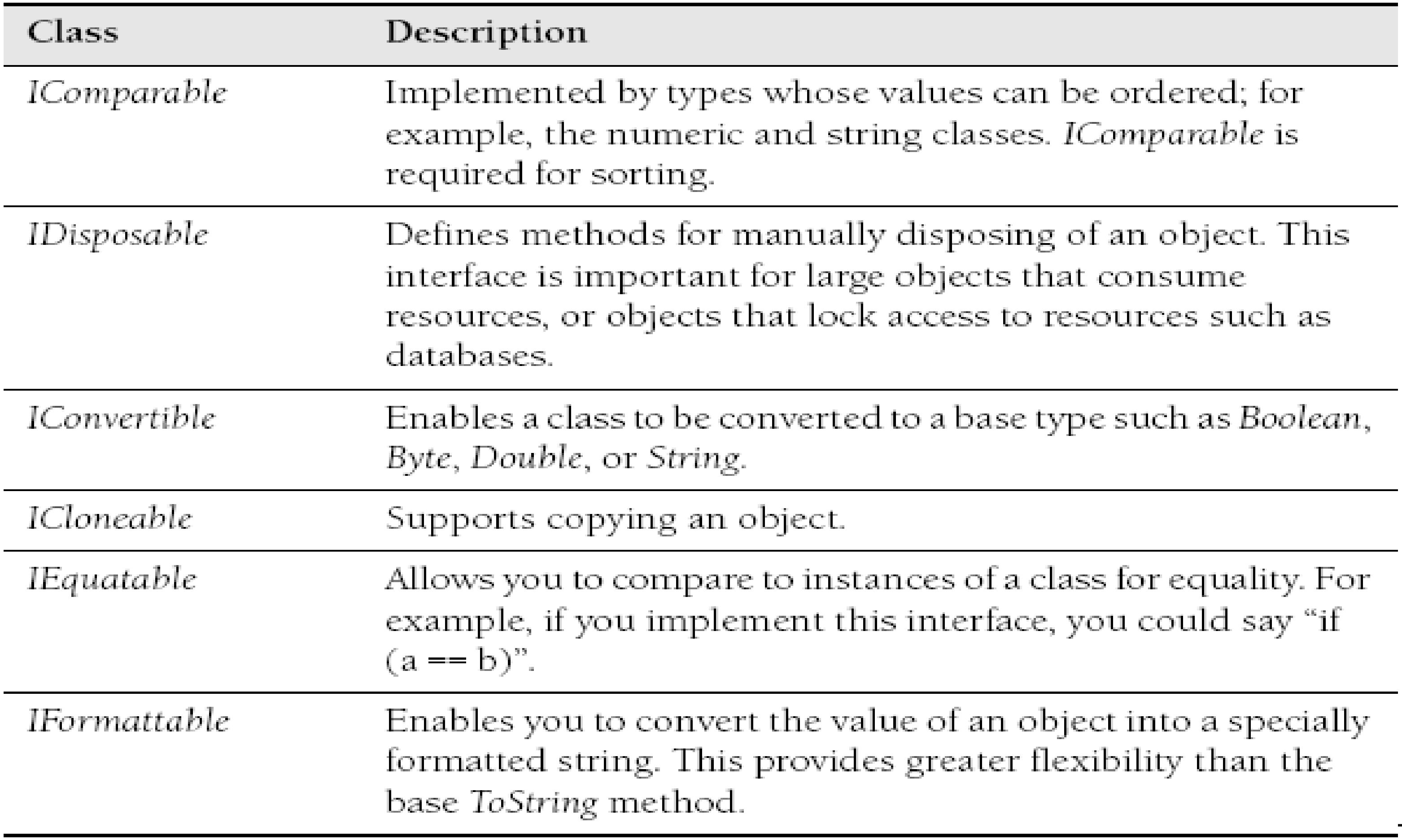
```
class DerivedException : System.ApplicationException
{
    public override string Message
    {
        get { return "An error occurred in the application."; }
    }
}
```



```
try
{
    throw new DerivedException();
}
catch (DerivedException ex)
{
    Console.WriteLine("Source: {0}, Error: {1}", ex.Source, ex.Message)
}
```

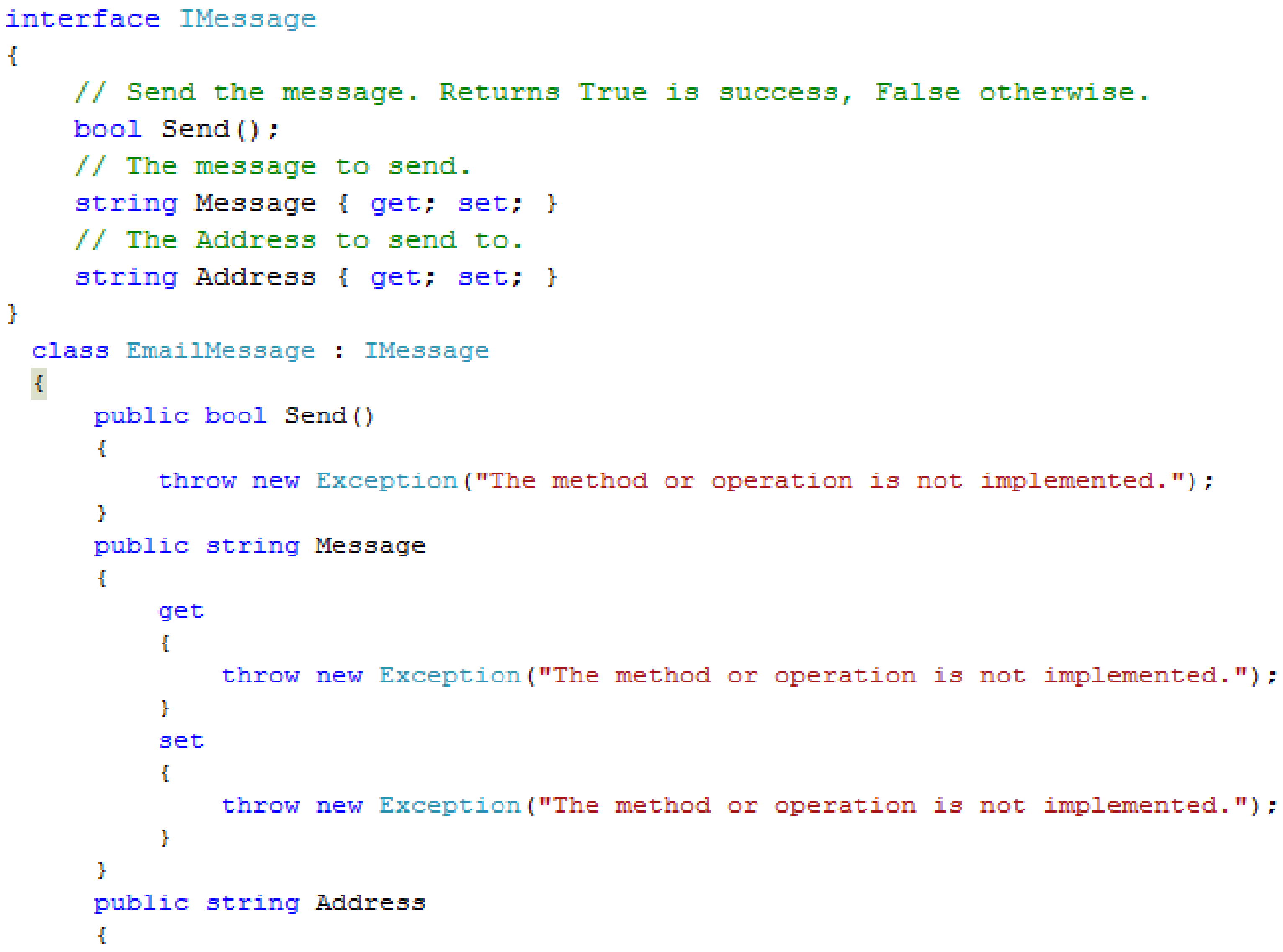
Interfaces

- Define a common set of members that all classes that implement the interface must provide.
  - For example, the IComparable interface defines the CompareTo method, which enables two instances of a class to be compared



Class	Description
IComparable	Implemented by types whose values can be ordered; for example, the numeric and string classes. *IComparable* is required for sorting.
IDisposable	Defines methods for manually disposing of an object. This interface is important for large objects that consume resources, or objects that lock access to resources such as databases.
IConvertible	Enables a class to be converted to a base type such as *Boolean*, *Byte*, *Double*, or *String*.
ICloneable	Supports copying an object.
IEquatable	Allows you to compare to instances of a class for equality. For example, if you implement this interface, you could say "if (a == b)".
IFormattable	Enables you to convert the value of an object into a specially formatted string. This provides greater flexibility than the base *ToString* method.

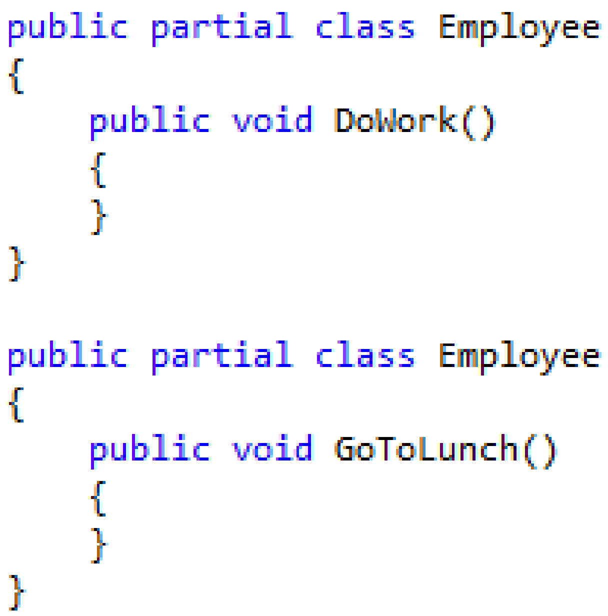
Creating the interface



```
interface IMessage
{
    // Send the message. Returns True is success, False otherwise.
    bool Send();
    // The message to send.
    string Message { get; set; }
    // The Address to send to.
    string Address { get; set; }
}
 class EmailMessage : IMessage
 {
     public bool Send()
     {
         throw new Exception("The method or operation is not implemented.");
     }
     public string Message
     {
         get
         {
             throw new Exception("The method or operation is not implemented.");
         }
         set
         {
             throw new Exception("The method or operation is not implemented.");
         }
     }
     public string Address
     {
```

Partial class

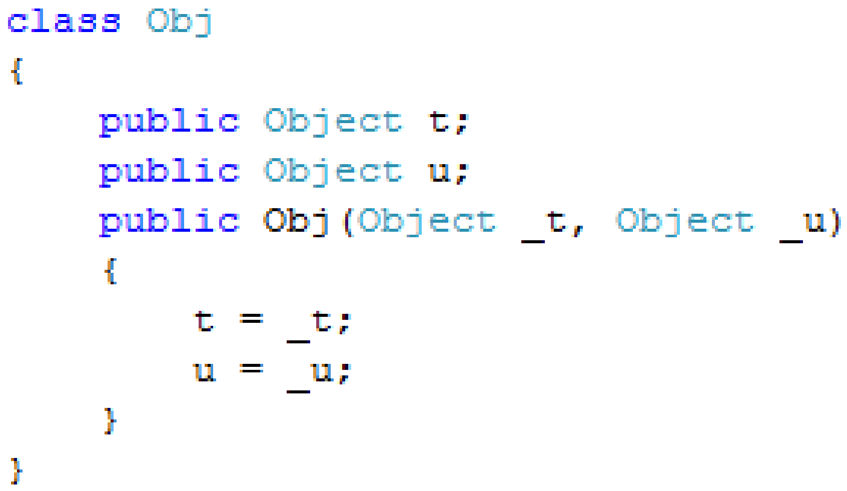
- Partial classes allow you to split a class definition across multiple source files.
- The benefit of this approach is that it hides details of the class definition so that derived classes can focus on more significant portions.



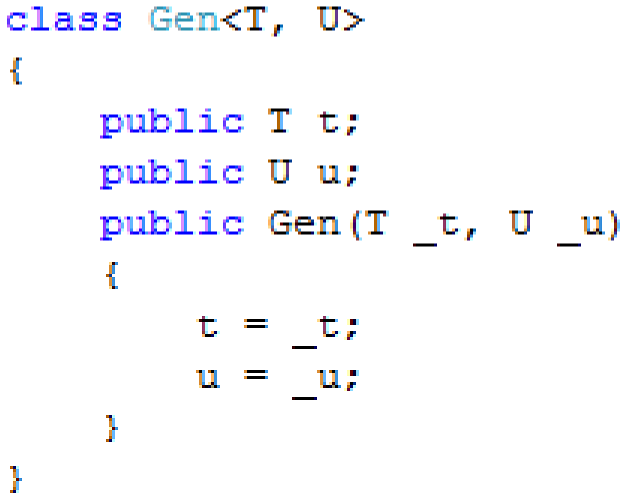
```
public partial class Employee
{
    public void DoWork()
    {
    }
}

public partial class Employee
{
    public void GoToLunch()
    {
    }
}
```

Generics

- Generics allow you to define a type while leaving some details unspecified.
- Instead of specifying the types of parameters you can allow code that uses your type to specify it.
- Generic classes - *Dictionary, Queue,SortedDictionary, and SortedList*
- *Compare:*

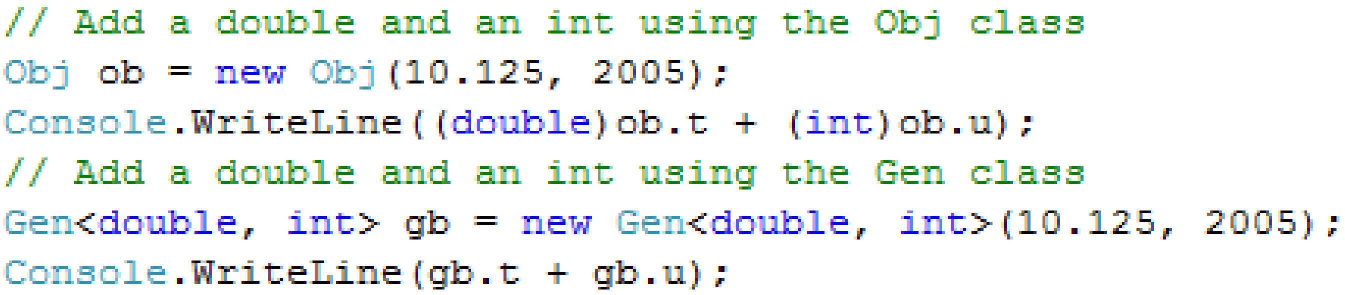


```
class Obj
{
    public Object t;
    public Object u;
    public Obj(Object _t, Object _u)
    {
        t = _t;
        u = _u;
    }
}
```



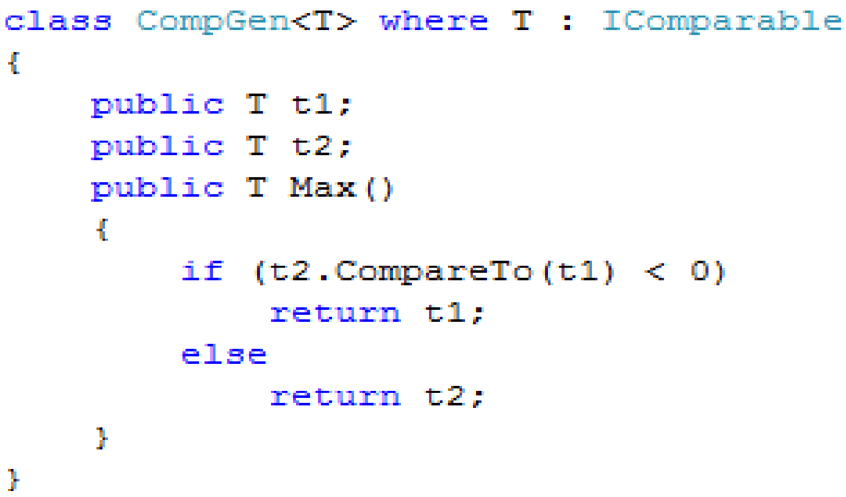
```
class Gen<T, U>
{
    public T t;
    public U u;
    public Gen(T _t, U _u)
    {
        t = _t;
        u = _u;
    }
}
```

Generics (cont.)



```
// Add a double and an int using the Obj class
Obj ob = new Obj(10.125, 2005);
Console.WriteLine((double)ob.t + (int)ob.u);
// Add a double and an int using the Gen class
Gen<double, int> gb = new Gen<double, int>(10.125, 2005);
Console.WriteLine(gb.t + gb.u);
```

- Generics would be extremely limited if you could only write code that would compile for any class
- To overcome this limitation, use constraints (interface, reference or value type, constructor, base class) to place requirements on the types



```
class CompGen<T> where T : IComparable
{
    public T t1;
    public T t2;
    public T Max()
    {
        if (t2.CompareTo(t1) < 0)
            return t1;
        else
            return t2;
    }
}
```

Events

- An event is a message sent by an object to signal the occurrence of an action.
  - The action could be caused by user interaction, such as a mouse click, or it could be triggered by some other program logic.

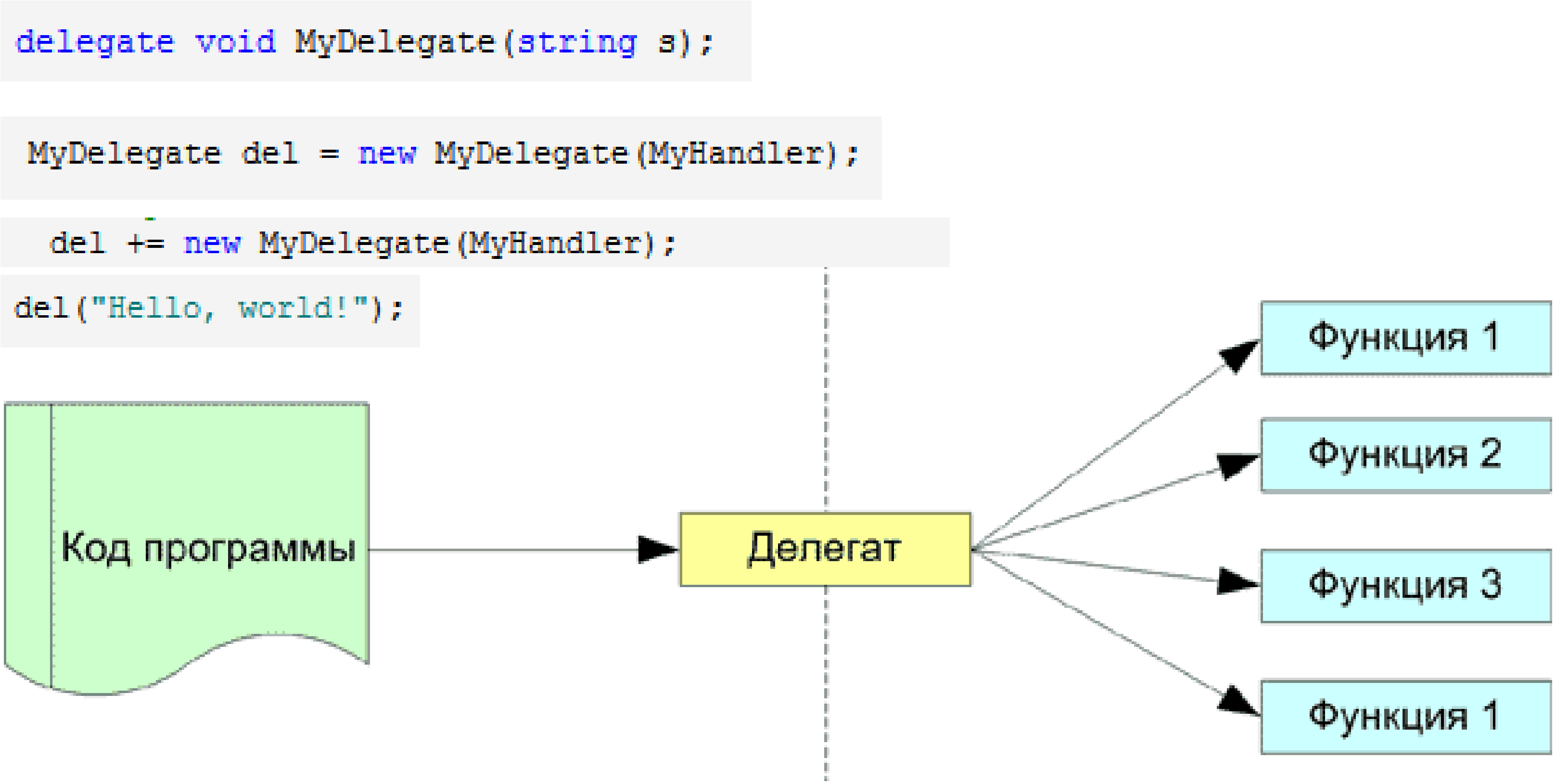
The object that raises the event is called the event sender.

The object that captures the event and responds to it is called the event receiver.

- An intermediary between the source and the receiver. Is a delegate, which provides the functionality of a function pointer

Delegates

- A delegate is a class that can hold a reference to a method.
- Unlike other classes, it has a signature, and it can hold references only to methods that match its signature.



```
delegate void MyDelegate(string s);
```

```
 MyDelegate del = new MyDelegate(MyHandler);
```

```
  del += new MyDelegate(MyHandler);
```

```
del("Hello, world!");
```

Serialization

- To enable a class to be serialized, you must add the *Serializable attribute:*

```
[Serializable]
class ShoppingCartItem
{
}
```

Without the Serializable attribute, a class is not serializable.

Converting between types

- Often, you need to convert between two different types.
  - For example, you might need to determine whether an Integer is greater or less than a Double
- C# allows *implicit conversion* if the destination type can accommodate all possible values from the source type. That is called a *widening conversion*

```
int i = 1;
double d = 1.0001;
d = i; // Conversion allowed
```

- If the range of the source type exceeds that of the destination type, the operation is called a *narrowing conversion* and requires *explicit conversion.*
  - Use type.*ToString*, type.*Parse*, *(type) cast* operator

Boxing & Unboxing

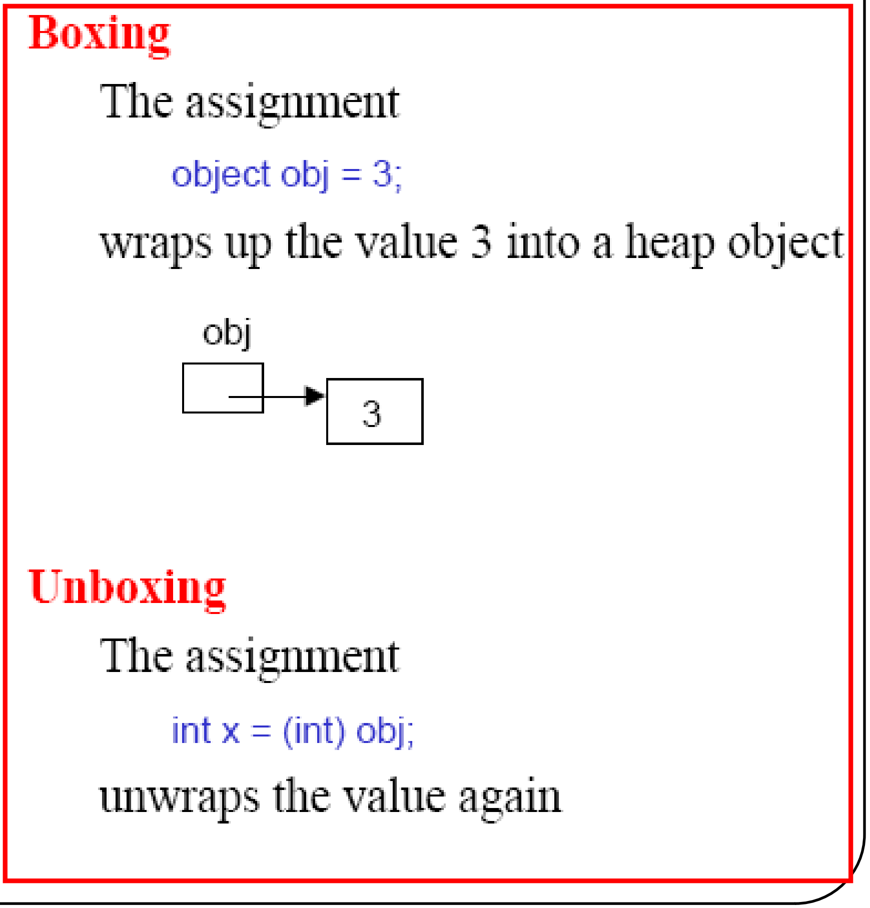
- Boxing converts a value type to a reference type, and unboxing converts a reference type to a value type.

- Let’s convert *Integer* to an *Object :*

```
int i = 123;
object o = (object) i;
```

- And vice versa:

```
object o = 123;
int i = (int) o;
```



Boxing

The assignment

```
object obj = 3;
```

wraps up the value 3 into a heap object

Unboxing

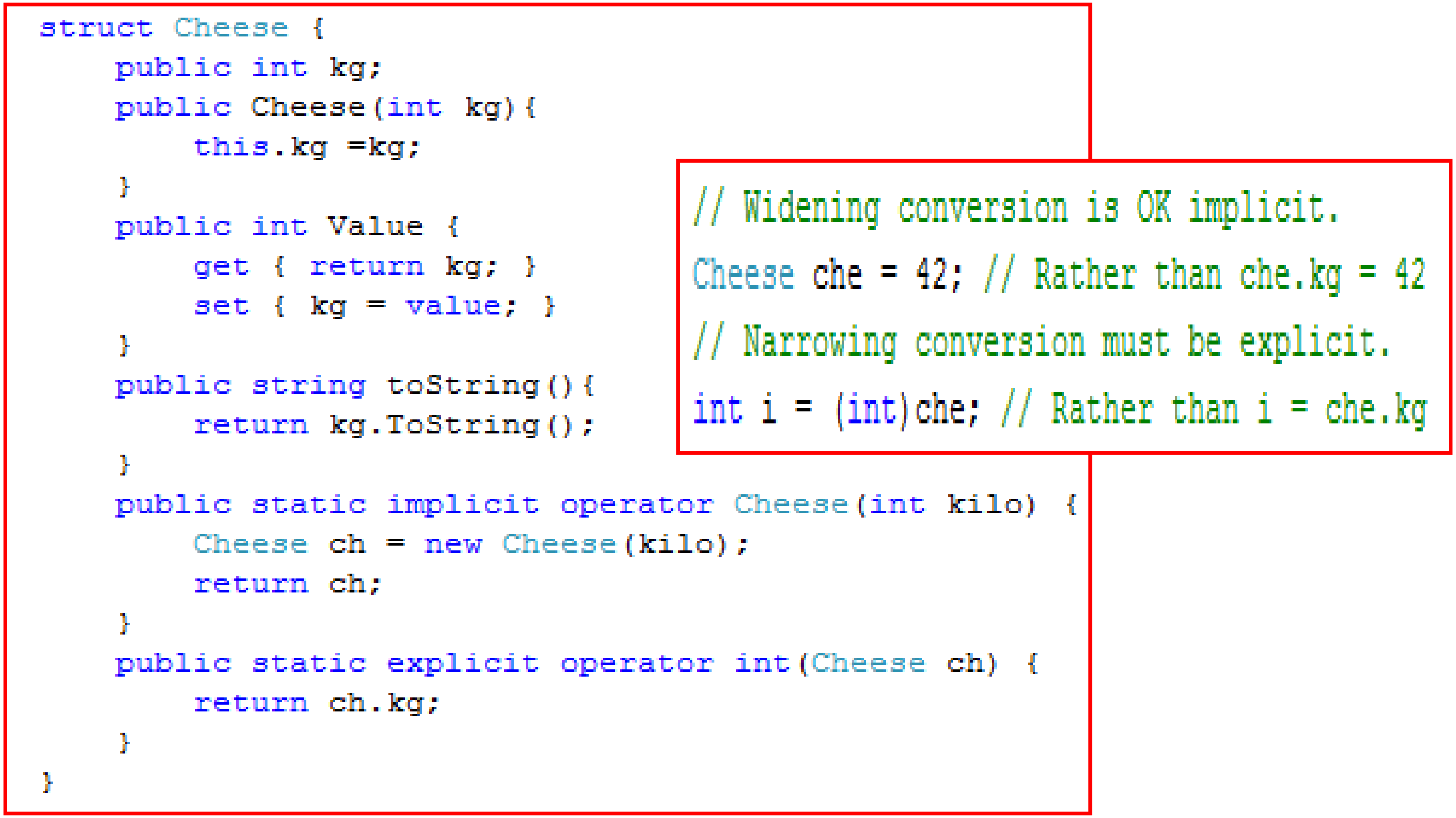
The assignment

```
int x = (int) obj;
```

unwraps the value again

Conversion in custom types

- Use the implicit keyword for conversions that don’t lose precision and explicit keyword for conversions that could lose precision.

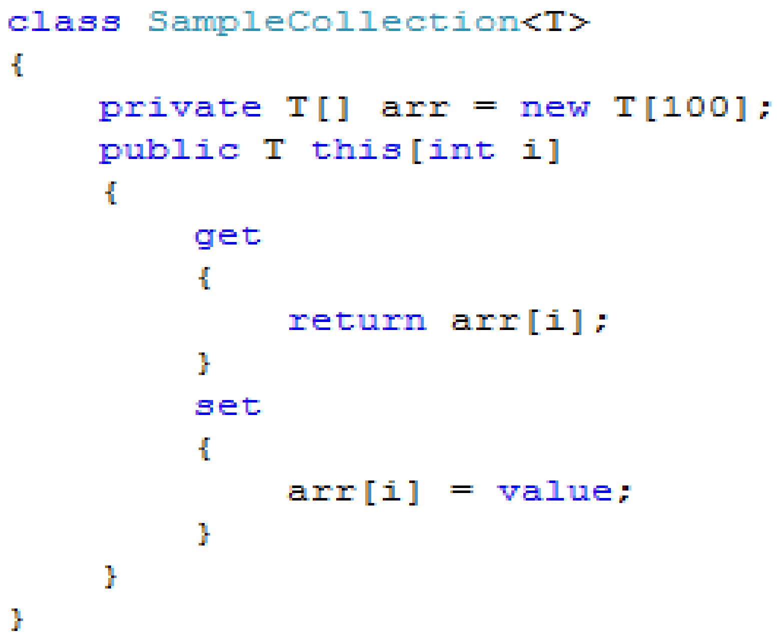


```
struct Cheese {
    public int kg;
    public Cheese(int kg){
        this.kg =kg;
    }
    public int Value {
        get { return kg; }
        set { kg = value; }
    }
    public string toString(){
        return kg.ToString();
    }
    public static implicit operator Cheese(int kilo) {
        Cheese ch = new Cheese(kilo);
        return ch;
    }
    public static explicit operator int(Cheese ch) {
        return ch.kg;
    }
}
```

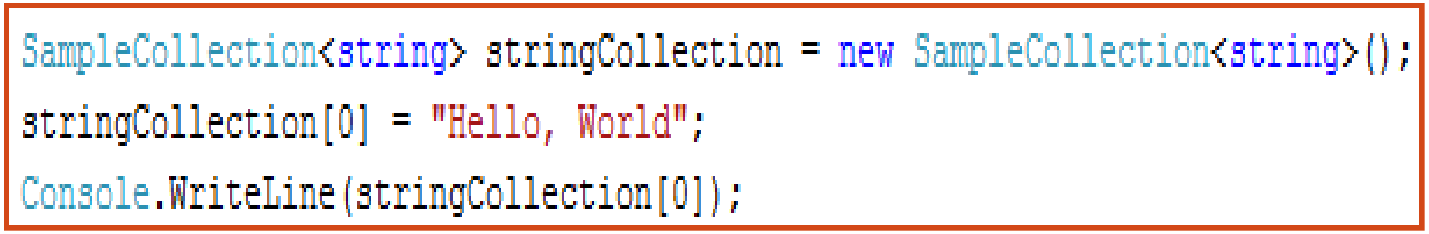
```
// Widening conversion is OK implicit.
Cheese che = 42; // Rather than che.kg = 42
// Narrowing conversion must be explicit.
int i = (int)che; // Rather than i = che.kg
```

Indexers

- Indexers allow instances of a class or struct to be indexed just like arrays:



```
class SampleCollection<T>
{
    private T[] arr = new T[100];
    public T this[int i]
    {
        get
        {
            return arr[i];
        }
        set
        {
            arr[i] = value;
        }
    }
}
```



```
SampleCollection<string> stringCollection = new SampleCollection<string>();
stringCollection[0] = "Hello, World";
Console.WriteLine(stringCollection[0]);
```

Summary

- The .NET Framework includes a large number of built-in types that you can use directly or use to build your own custom types.
- Value types directly contain their data, offering excellent performance. However, value types are limited to types that store very small pieces of data - 16 bytes or shorter.
- You can create user-defined types that store multiple values and methods. In object-oriented development environments, a large portion of your application logic will be stored in user-defined types.
- Enumerations improve code readability by providing symbols for a set of values.

Summary

- Reference types contain the address of data rather than the actual data.
- When you copy a value type, a second copy of the value is created. When you copy a reference type, only the pointer is copied. *Therefore, if you copy a reference type and then modify the copy, both the copy and the original variables are changed.*
- The .NET Framework includes a large number of built-in reference types that you can use directly or use to build your own custom types.
- Strings are immutable; use the *StringBuilder* class to create a string dynamically.
- Use streams to read from and write to files and memory.
- Use the *Catch* clause within *Try blocks* to filter exceptions by type.
- *Close and dispose* resources in the *Finally clause* of a *Try block.*

Summary

- Use inheritance to create new types based on existing ones.
- Use interfaces to define a common set of members that must be implemented by related types.
- Partial classes split a class definition across multiple source files.
- Events allow you to run a specified method when something occurs in a different section of code.

Summary

- Widening conversions occur implicitly
- Narrowing conversions require explicit conversion in C#
- Boxing allows any type to be treated as a reference type.
- You must specifically implement conversion operators to enable conversion in custom types.

Programming practice

- Create any class you like
- Add some fields, properties, methods.
  - Feel the difference between a property and a field.
- Implement the following features in your class:
  - Indexers
  - Implicit conversion
  - Explicit conversion
  - Overload +, - or * operators for your data structure
- Test your class