Object Oriented Software Development

4. C# data types, objects and references

Data in programs

- Programs need to store information in computer's **memory**
- Information must be available to the program when needed
- Simple data values
  - Numbers, characters, etc.
- Objects
  - Attributes, behaviour which can be invoked
- In C#, even simple data values are objects

Variables

- Program accesses data through **variables**
- A variable is the program's "label" for, or **reference** to, an object
- Each variable has a specific **scope**
  - Instance variables – belong to an object, can be accessed by any code in that object's class
  - Method parameters - can only be accessed in the method where they are declared
  - Local variables – can only be accessed in the method or code block where they are declared

Areas of operating memory

- Stack
  - Keeps track of executing code
  - "what's been called?"
  - Stores variables needed by executing code
  - Only last added items can be accessed
  - Like a stack of boxes
- Heap
  - Keeps track of objects
  - Objects can be accessed by any code at any time

Stack and heap example

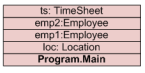
- **ClassesDemo project**
- **Employee.cs**
- **TimeSheet.cs**
- **Program.cs**
- What happens in memory when program runs?

What goes on the stack?

- **Program.Main** executes
- Variables are placed on stack:

```
Location loc = new Location();
Employee emp1 = new Employee("Michael",
"michael", loc, "1234");
Employee emp2 = new Employee("Susan",
"susan", loc, "4321");
TimeSheet ts = new TimeSheet();
```

Stack



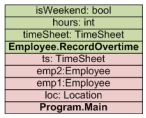
Method called from Main

- **RecordOvertime** method of an Employee object is called
- Parameters stored on stack

```
public void RecordOvertime(TimeSheet
   timeSheet, int hours, bool isWeekend)
{


emp1.RecordOvertime(ts, 5, true);
```

Stack



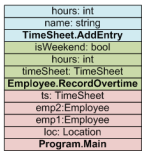
Method calls another method

- **AddEntry** method of the TimeSheet object is called
- Parameters stored on stack

```
public void AddEntry(string name, int hours)
{

timeSheet.AddEntry(name, hours * 2);
```

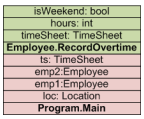
Stack



Stack and heap example

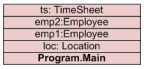
- **AddEntry** method finishes
  - Variables removed from stack

Stack



- **RecordOvertime** method finishes
  - Variables removed from stack

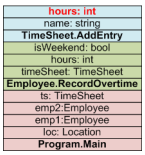
Stack



What is actually stored?

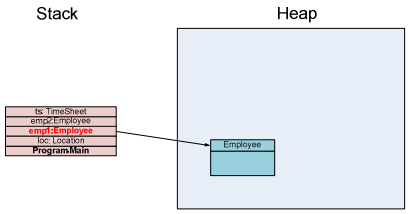
- Example:
  - **hours** parameter in call to AddEntry
  - Integer value
  - Memory to hold an integer value is allocated on the stack
  - Actual integer value stored on stack

Stack



What is actually stored?

- Another example:
  - **emp1** variable in Main, of type Employee
  - Points to an Employee **object**

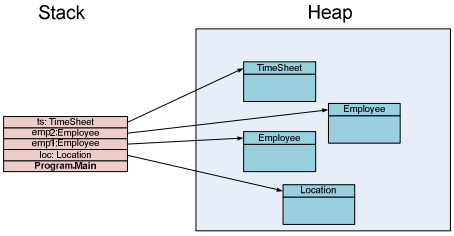


Object references

- Memory to store attributes of **object** is allocated on **heap**
- The **stack** just stores a pointer, or **reference**, to the object
- The reference is essentially an address in heap memory which the program can go to in order to find the correct object

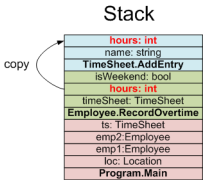
Object references

- Main creates several objects on the heap and references to these on the stack



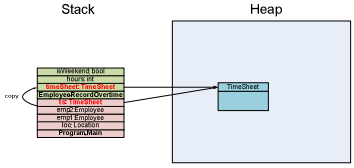
Parameter passing

- RecordOvertime passes integer *value* **hours** to AddEntry
- Copy is made of integer *value* and added to stack as parameter of AddEntry



References to same object

- Main creates TimeSheet object *reference* and passes it as a parameter to RecordOvertime
  - Parameter contains a copy of the *reference*
  - There can be many references in the stack to a single object in the heap



Releasing memory

- If a program keeps allocating memory and never releases it for re-use, the memory will fill up and the computer will crash
- Stack memory is released whenever a method finishes
- The memory allocated for that method is removed from the stack and is available for re-use

Garbage collection

- Releasing heap memory is more complicated
- As long as there is at least one reference to an object on the heap then the object is kept "alive"
- Objects with no references are eligible to be removed and their memory released
- Removed by the **garbage collector**
- Employee objects in the example will be removed when Main method finishes

Garbage collection

- Garbage collector (GC) runs periodically and removes "orphaned" objects
- Can't be sure exactly when it will do so
- GC is a feature of a managed language
  - e.g. .NET languages, Java, PHP
  - Programmer does not have to manage memory
- Unmanaged languages require programmer to explicitly release memory
  - e.g. C++, C

Value types and reference types

- The .NET type system defines two categories of data type, or object type
- **Value types**
  - Values can be stored on the stack
  - Derived from **System.ValueType**
  - Examples of built-in framework value types:
    - Byte, Int16, Int32, Int64, Single, Double, Decimal, Char, Boolean
  - C# has built-in types which are aliases for these:
    - byte, short, int, long, float, double, decimal, char, bool

Value types and reference types

- **Reference types**
  - Objects stored in the heap
  - References stored on the stack
  - Types derived from **System.Object**
  - Examples of reference types:
    - String (C# alias is string)
    - all classes, including classes in your project
    - arrays (see later)
    - delegates (see later)
    - Interfaces (see later)

Boxing and unboxing

- Boxing
  - Converting value type to reference type

```
int i = 123;
// The following line boxes i.
object o = i;
```

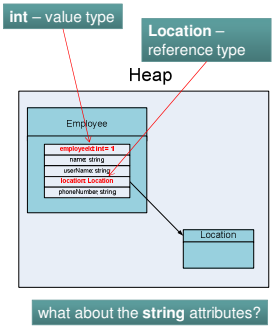
- Unboxing
  - Converting reference type to value type

```
o = 123;
i = (int)o;  // unboxing
```

- We will look again at boxing and type conversions later

A closer look at an object

- Objects on the heap have attributes which need to be stored
- Value type attribute data is stored with object
- Reference type attributes are stored as references to other objects on the heap



Creating value types

- There are two kinds of value type in .NET
- **struct**
  - Similar to a class, but stored as a value type
  - Local variable of struct type will be stored on the stack
  - Built-in values types, e.g. Int32, are structs
- **enum**
  - An enumeration type
  - Consists of a set of named constants

struct

- Example in TimeSheet.cs - rewrite TimeSheet as a struct rather than a class

```
public struct TimeSheet
{
```

- **struct** can contain instance variables, constructors, properties, methods
- Can't explicitly declare default constructor
  - Compiler generates default constructor

struct

- Instance can be created without **new** key word

```
TimeSheet ts;
```

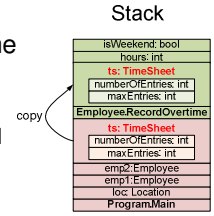
- With **class**, this would create a **null** reference
- With **struct**, this creates instance with fields set to default values

```
ts = new TimeSheet();
```

- This explicitly calls default constructor

Method call with struct parameter

- Revisit earlier example with TimeSheet as a struct
- Main creates TimeSheet *struct* instance and passes it as a parameter to RecordOvertime
  - Parameter contains a copy of the *struct*
  - A copy of **whole struct** placed on stack



struct vs. class

- TimeSheet example is a small struct, but structs can have large numbers of instance variables
- Passing large structs as parameters can use a lot of stack memory
- On the other hand, creating objects on the heap is expensive in terms of performance compared to creating structs
- No definitive rules, but take these factors into account when deciding

enum

- **enum** is a good way of storing and naming constant values

```
enum Days { Sat, Sun, Mon, Tue, Wed, Thu, Fri };
```

- Enum has an underlying data type
  - int by default
  - in example, Days.Sat, Days.Sun, Days.Mon... represent values 0,1, 2,...
  - can set values explicitly

enum code example

- **EnumDemo project**
- **Employee.cs**
- **Program.cs**

enum example

- Previously indicated pay rate with boolean value isWeekend
- Replace this with **enum**, which allows more than simply true/false

```
public enum PayRate
{
    Day,
    Weekend,
    Holiday
}
```

enum example

- Change parameter in RecordOvertime to type PayRate

```
public void RecordOvertime(TimeSheet timeSheet, int hours,
    PayRate payRate)
{
    if (payRate == PayRate.Holiday)
    {
        timeSheet.AddEntry(name, hours * 3);
    }
    else if (payRate == PayRate.Weekend)
    {
        timeSheet.AddEntry(name, hours * 2);
    }
    else
    {
        timeSheet.AddEntry(name, hours);
    }
}
```


enum example

- Pass in enumeration value to method

```
emp1.RecordOvertime(ts, 5, PayRate.Holiday);
```

- Always refer to value by name, don't need to know or use underlying value

Warning!

- Classes, objects, instance variables, methods, references are fundamental OO concepts
- Value types (struct, enum) and properties are specific to the way in which .NET interprets the OO programming model
- Other languages do it slightly differently, e.g. Java has primitive types (for simple values) and classes – no structs

Creating C# types

- A program (or class library) consists of type definitions (classes, structs, etc)
- These define the types of objects which need to be created when the program runs
- Objects perform the program's required functions
- Program written in C# (source code), saved in file with **.cs** extension

Compiling C# types

- C# is a high-level, human-readable language
- Computer processor understands detailed, low-level instructions, called **machine code**
- In traditional languages, source code is converted to machine code by a **compiler**
- In .NET, source code is compiled to an intermediate language (**MSIL**)
- Similar to machine code, but is not specific to any real processor

Creating assemblies

- MSIL needs a special program called the Common Language Runtime (**CLR**)
- CLR converts MSIL to "native" machine code
- MSIL is contained in an **assembly**, which is a file with **.exe** or **.dll** extension

Including other assemblies

- Source code can use, or reference, types defined in other assemblies, e.g the .NET framework libraries
- Need **project reference** to these assemblies in Visual Studio so that compiler knows about the types in them
- Need using statements in your code to include classes from referenced assemblies

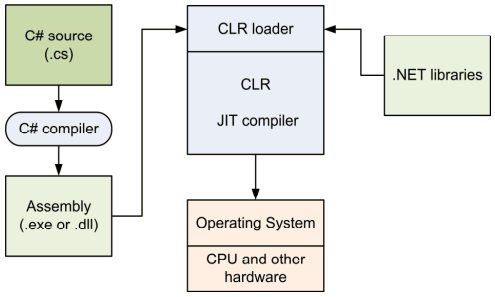
Building a Visual Studio project

- When you build a project in Visual Studio, the following happens:
  - All source code files in the project are compiled
  - An assembly is created, usually in a folder called bin
  - Any additional resources (text, images, etc) are copied into bin folder or embedded into assembly
  - Referenced assemblies may be copied into bin folder

How a C# program runs

- CLR is software which runs on top of host operating system
- CLR loads assembly and uses a Just-in-Time compiler (JIT) to translate MSIL code to native machine code which can be executed by CPU
- Also loads referenced assemblies
- Same MSIL code can be executed on different CPUs if CPU is supported by CLR

How a C# program runs



Further reading

- The following link leads to a comprehensive series of articles on the stack and heap in .NET
  - http://www.c-sharpcorner.com/uploadfile/rmcochran/csharp_memory01122006130034pm/csharp_memory.aspx?articleid=9adb0e3c-b3f6-40b5-98b5-413b6d348b91

What's next?

- We will go on to look at some more important concepts in object oriented programming: interfaces, polymorphism and inheritance