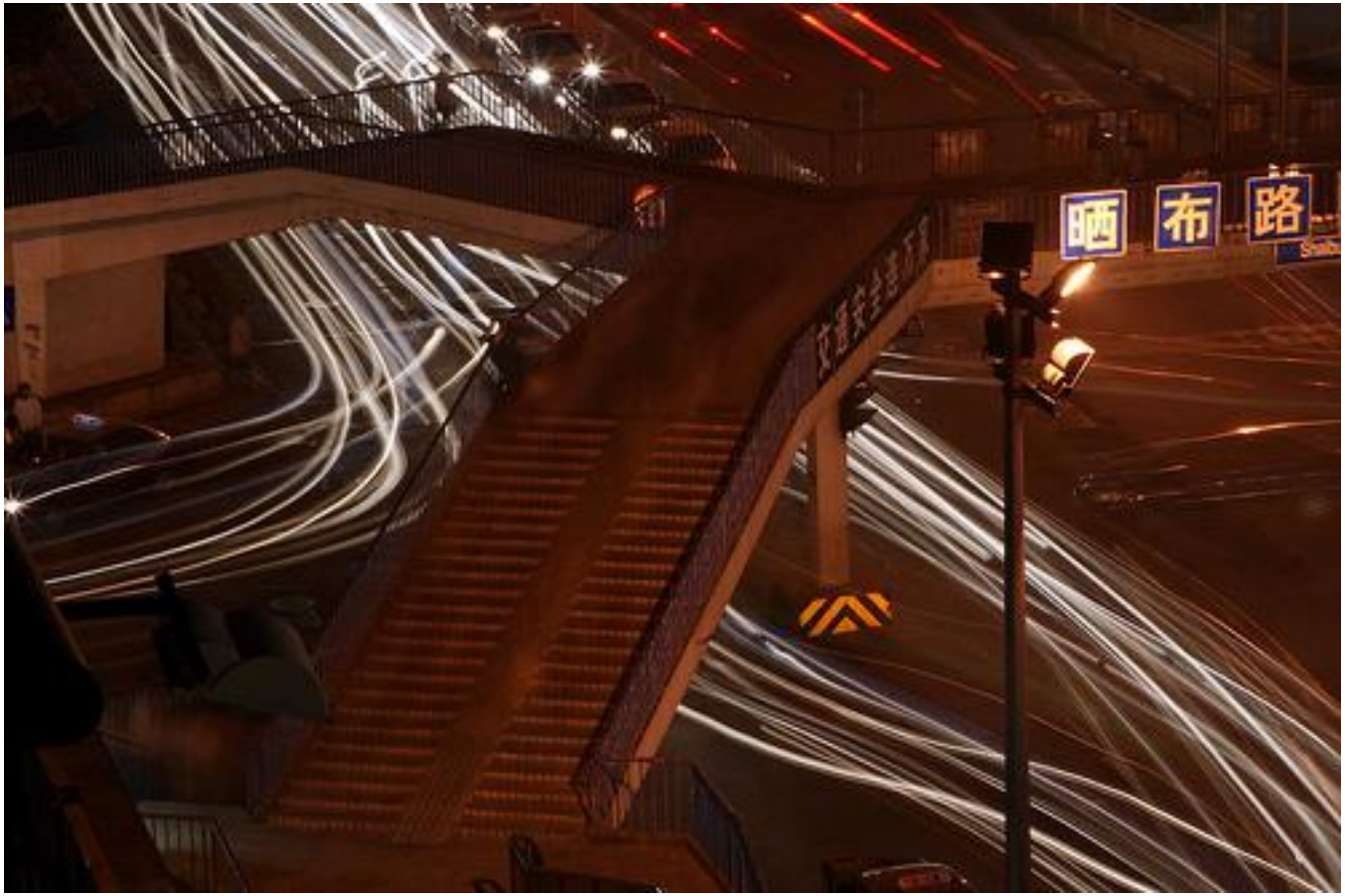
C# : Control Flow

Finding the path to a solution

Overview

- Branching
- Iterating
- Jumping
- Exceptions



Branching

```
if (age <= 2)
    ServeMilk();
else if (age < 21)
    ServeSoda();
else
{
    ServeDrink();
}
```

```
if (age <= 2)
{
    if(name == "Scott")
    {
        // ...
    }
}
```

```
string pass = age > 20 ? "pass" : "nopass";
```

Switching

- Restricted to integers, characters, strings, and enums
- No “fall throughs” like in C++
- Case labels are constants
- Default label is optional

```
switch(name) {
    case "Scott":
        ServeSoda();
        break;
    case "Poona":
        ServeMilk();
        ServeDrink();
        break;
    default:
        ServeMilk();
        break;
}
```

Iterating

```
for(int i = 0; i < age; i++)
{
    Console.WriteLine(i);
}
```

```
while(age > 0)
{
    age -= 1;
    Console.WriteLine(age);
}
```

```
do
{
    age++;
    Console.WriteLine(age);
} while (age < 100);
```

Iterating with foreach

- **Iterates a collection of items**
  - Uses the collection's GetEnumerator method

```
int[] ages = {2, 21, 40, 72, 100};
foreach (int value in ages)
{
    Console.WriteLine(value);
}
```

```
int[] ages = {2, 21, 40, 72, 100};
IEnumerator enumerator = ages.GetEnumerator();
while(enumerator.MoveNext())
{
    Console.WriteLine((int)enumerator.Current);
}
```

Jumping

- break
- continue
- goto
- return
- throw

```
foreach(int age in ages) {
    if(age == 2) {
        continue;
    }
    if(age == 21) {
        break;
    }
}
```

```
foreach(int age in ages) {
    if(age == 2) {
        goto skip;
    }
    // ...
    skip:
        Console.WriteLine("Hello!");
}
```

Returning and Yielding

- You can use return in a void method
- You can use yield to build an IEnumerable

```
void CheckAges()
{
    foreach (int age in ComputeAges())
    {
        if (age == 21) return;
    }
}
```

```
IEnumerable ComputeAges()
{
    yield return 2;
    yield return 21;
    for(int i = 22; i < 32; i++)
    {
        yield return i;
    }
}
```

Throwing

- **Use throw to raise an exception**
  - Exceptions provide type safe and structured error handling in .NET
- **Runtime unwinds the stack until it finds a handler**
  - Exception may terminate an application

```
if(age == 21)
{
    throw new ArgumentException("21 is not a legal value");
}
```

Built-in Exceptions

- **Dozens of exceptions already defined in the BCL**
  - All derive from System.Exception

Type	Description
System.DivideByZeroException	Attempt to divide an integral value by zero occurs.
System.IndexOutOfRangeException	Attempt to index an array via an index that is outside the bounds of the array.
System.InvalidCastException	Thrown when an explicit conversion from a base type or interface to a derived type fails at run time.
System.NullReferenceException	Thrown when a null reference is used in a way that causes the referenced object to be required.
System.StackOverflowException	Thrown when the execution stack is exhausted by having too many pending method calls.
System.TypeInitializationException	Thrown when a static constructor throws an exception, and no catch clauses exists to catch it.

Handling Exceptions

- **Handle exceptions using a try block**
  - Runtime will search for the closest matching catch statement

```
try
{
    CheckAges();
}
catch(DivideByZeroException ex)
{
    Console.WriteLine(ex.Message);
    Console.WriteLine(ex.StackTrace);
}
```

Chaining Catch Blocks

- **Place most specific type in the first catch clause**
- **Catching a System.Exception catches everything**
  - … except for a few “special” exceptions

```
try {
    // ...
}
catch(DivideByZeroException ex)
{
    // ...
}
catch(Exception ex)
{
    // ...
}
```

Finally

- **Finally clause adds finalization code**
  - Executes even when control jumps out of scope

```
FileStream file = new FileStream("file.txt", FileMode.Open);
try
{
}
finally
{
    file.Close();
}
```

```
using(FileStream file1 = new FileStream("in.txt", FileMode.Open))
using(FileStream file2 = new FileStream("out.txt", FileMode.Create))
{
    // ...
}
```

Re-throwing Exceptions

- **For logging scenarios**
  - Catch and re-throw the original exception
- **For the security sensitive**
  - Hide the original exception and throw a new, general error
- **For business logic**
  - Useful to wrap the original exception in a meaningful exception

```
try
{
    // ...
}
catch(Exception ex)
{
    // log the error ...
    throw;
}
```

```
try
{
    // ...
}
catch(DivideByZeroException ex)
{
    throw new
       InvalidAccountValueException("...", ex);
}
```

Custom Exceptions

- Derive from a common base exception
- Use an Exception suffix on the class name
- Make the exception serializable

```
[Serializable]
public class InvalidAccountException : Exception
{
    public InvalidAccountException() { }
    public InvalidAccountException(string message) : base(message) { }
    public InvalidAccountException(string message, Exception inner)
        : base(message, inner) { }
    protected InvalidAccountException(
      System.Runtime.Serialization.SerializationInfo info,
      System.Runtime.Serialization.StreamingContext context)
        : base(info, context) { }
}
```

Summary

- Flow control statements fall into three categories
  - Branching
  - Looping
  - Jumping
- Exceptions are the error handling mechanism in .NET
  - Throw exceptions (built-in or custom)
  - Catch exceptions