C# Type Conversions

Topics Covered

- Type Compatibility
- Implicit/Explicit Casts
- Numeric Casts
  - Loss of Information
  - Signed/Unsigned
  - Storage Size (bits)
  - Decimal Range/Precision
- Impossible Conversions
- Casting to object
- Casting from object
- Two-Step Conversions
  - is Assignments
- Type-Testing with is
- Safe Casts with as
- Boxing

Types and Type Compatibility

C# is strongly-typed. The types of all variables are known at compile-time, and the language has strong rules around how one type may be converted to other types.

This document explains these rules, how we can work with them, and how we can test for type-compatibilities in our programs.

When we convert an object of one type to another type, it is called *type conversion* or *type casting*. This presentations uses both expressions interchangeable.

Implicit and Explicit Casts

Some type conversions can occur *implicitly*.

For example, a variable of type **long** can accommodate all possible values of an **int**, so we can implicitly convert an **int** to a **long** using a simple assignment:

```
int i = 55;
long l = i; // implicit type conversion
```

In contrast, a long variable can hold values outside of the range of an int, so if we need to convert a long to an int, we must do it explicitly:

```
i = (int)l; // explicit type conversion
```

This signals to the compiler that we are aware of the possibility that information may be lost. This usage of a typename between parentheses is called a *conversion operator*.

Converting Numeric Types

The general rule for converting numeric types is simple:

- If no information is lost in a conversion, then it can be implicit.
- If information may be lost in a conversion, then it must be explicit.

byte, as the smallest unsigned integer type, can be implicitly cast to any other integer type except for sbyte. Conversion to sbyte must be explicit because values of byte > 127 cannot be represented by sbyte.

```
sbyte sb = -1;
byte b = sb;     // not allowed
byte c = (byte)sb;  // allowed
```

Explicit cast

```
int i = c;    // no data lost
uint ui = c;  // no data lost
long L = c;   // no data lost
```

Signed vs. Unsigned

Signed integer types cannot be implicitly cast to unsigned types because of the potential loss of information – the negative numbers.

The explicit casts are allowed:

```
int i = -53;
uint ui = i;     // not allowed
ulong uL = i;    // not allowed
ui = (uint)i;    // explicit, ok
uL = (ulong)i;   // explicit, ok
```

Storage Size

Implicit casts from a larger type to a smaller type fail because, again, of the loss of information.

Converting a larger-sized integer (long, 64 bits) to smaller (int, 32 bits; or short, 16 bits) must be done explicitly:

```
long L = 1001;
int i = L;     // not allowed
short s = L;   // not allowed
i = (int)L;    // explicit, ok
s = (short)L; // explicit, ok
```

Converting Floating Point Types

The same rules apply to floating point types float (32 bits) and double (64 bits).

double can accommodate all possible values of a float, so the implicit cast is allowed.

Converting from double to float requires an explicit cast:

```
float f = 2.0f;
double d = f;    // allowed
f = d;           // not allowed
f = (float)d;    // explicit, ok
```

Converting double/decimal

There are no implicit conversions to/from decimal. It has larger size (128 bits) than double (64 bits), but smaller range.

Thus, information may be lost by converting in either direction:

```
double d = 1.577;
decimal m = d;          // not allowed
m = (decimal)d;         // explicit, ok

decimal m2 = 1.577M;
double d2 = m2;         // not allowed
d2 = (double)m2;        // explicit, ok
```

Impossible Conversions

Some type conversions are simply impossible.

There is no implicit or explicit conversion from DateTime to long.

There is no implicit or explicit conversion from string to long.

Custom conversions are possible, but those must be designed into the class.

```
DateTime now = DateTime.Now;
string s = "This is a test";
long L = (long)now;    // not possible!
L = (long)s;           // not possible!
long ticks = now.Ticks; // a custom "conversion"
DateTime now2 = new DateTime(ticks);
```

Converting to Object

All C# types are objects, so variables of type object can represent any type.

```
object o = 125;          // int
o = 257L;                // long
o = 3.77;                // double
o = 3.77M;               // decimal
o = 1024ul;              // unsigned long
o = '9';                 // char
o = DateTime.Now;        // DateTime
o = "This is a test.";   // string
```

The tricky part comes when we need to convert from object back to the “actual” type!

Converting from Object

To convert from a variable of type object back to the actual or “underlying” type, we must use an explicit cast. These statements all work:

```
object o = 125;          // int
int i = (int)o;
o = 257L;                // long
long L = (long)o;
o = 3.77;                // double
double d = (double)o;
o = 3.77M;               // decimal
decimal m = (decimal)o;
o = DateTime.Now;        // DateTime
DateTime now = (DateTime)o;
```

“Two-Step” Conversions from Object

When explicitly converting from object to another type, we must be careful to be correct.

If the explicit cast is exactly correct, the cast will succeed.

If the explicit cast is not exactly correct, it will result in an exception.

This occurs even when the underlying implicit cast would otherwise work:

```
object o = 125;    // int
int i = (int)o;    // correct explicit cast, ok

byte b = (byte)i; // explicit cast int -> byte, ok
b = (byte)o;      // InvalidCastException!

long L = i;       // implicit cast int -> long, ok
L = (long)o;      // InvalidCastException!
```

To achieve a “two-step” conversion without an exception, we must make the two steps explicit:

```
object o = 125;   // int
byte b = (byte)(int)o;
```

Type-Testing: is

If you have any uncertainty about the type of an object, you must test that type before attempting a conversion.

One way to do this is the **is** keyword to test for type compatibility:

```
object o = 125;   // int
if (o is byte)
{
    byte b = (byte)o;
}
if (o is long)
{
    long l = (long)o;
}
if (o is int)
{
    int i = (int)o;
}
```

In this code example, only the last if conditional will evaluate to true. The variable o was assigned to an integer, not a byte or long.

This example is written to highlight the type-testing logic, but there is a simpler way to write it.

is + Assignment

We can use is with pattern-matching to both test its type and declare/initialize a variable of that type in one statement:

```
object o = 125;   // int
if (o is byte)
{
  byte b = (byte)o;
}
if (o is long)
{
  long l = (long)o;
}
if (o is int)
{
  int i = (int)o;
}
```

```
object o = 125;   // int
if (o is byte b)
{
  // work with b
}
if (o is long l)
{
  // work with l
}
if (o is int i)
{
  // work with i
}
```

Casting with as

The as keyword performs a “safe” cast. as is used in assignments.

If the cast is valid, the assignment succeeds. If the cast is not valid, the assignment results in a null.

as can only be used with reference types.

In this example, the Array implements the ICloneable interface. The List does not:

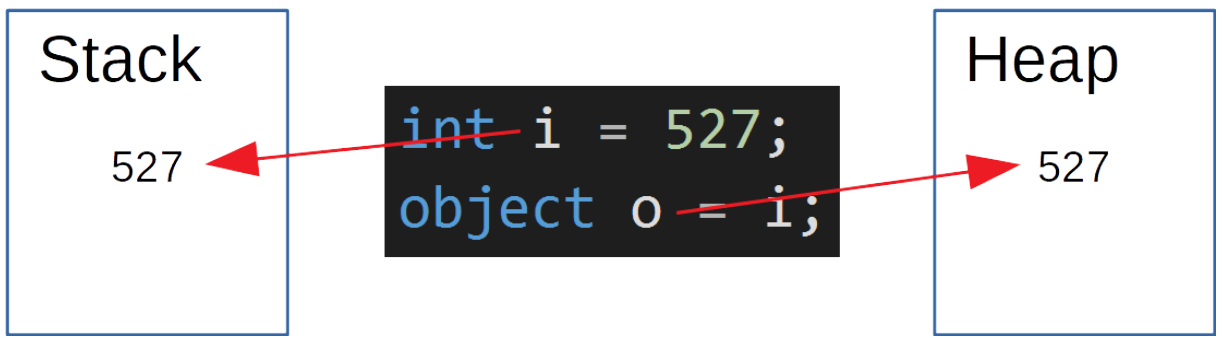
```
List<int> list = new List<int>();
int[] array = new int[] { 1, 2, 3, 4 };
ICloneable ic = list as ICloneable; // fails.  ic is null
ic = array as ICloneable; // succeeds. ic is set.
```

Boxing

A peculiarity arises when we assign a value type, such as int, to object, which is reference type.

Storage for variables of type int is allocated on the stack, while variables of type object point to memory in the heap.

When we assign a variable of type object from a variable of type int, memory in the heap is allocated to store that integer value.



This seemingly simple assignment results in new memory allocation.

This process is called *boxing*.

It should be avoided, especially in loops with many values.

Converting User-Defined Types

C# allows an application to define custom types. For example, a construction management application might define a **Building** type, with attributes you could expect for a building – the target date for completion, the construction materials used, the architectural designs.

Elsewhere, the same application might define a **Worker** type, representing a worker at a construction site. The Worker attributes might include construction skills and hourly or weekly availability.

Building and **Worker** are distinct types. There is no obvious way to convert an object of type Building to an object of type Worker.

The C# type system understands these distinctions and disallows inappropriate conversions.

Custom Type Conversion

Sometime we create classes where it makes sense to be able to convert them to another type.

Suppose we define a Fraction class, with Numerator, Denominator and Ratio properties shown below.

It is then reasonable to be able to cast this implicitly to a double.

```
public int Numerator { get; private set; }
public int Denominator { get; private set; }
public double Ratio => (double)Numerator / Denominator;

public static implicit operator double (Fraction fr)
{
    return fr == null ? 0 : fr.Ratio;
}
```

A class can define custom casting operators as shown here.

```
Fraction frA = new Fraction(1, 2),
  frB = new Fraction(3, 4);
// implicit & explicit casting:
double valueA = frA, valueB = (double)frB;
```

Topics Covered

- Type Compatibility
- Implicit/Explicit Casts
- Numeric Casts
  - Loss of Information
  - Signed/Unsigned
  - Storage Size (bits)
  - Decimal Range/Precision
- Impossible Conversions
- Casting to object
- Casting from object
- Two-Step Conversions
  - is Assignments
- Type-Testing with is
- Safe Casts with as
- Boxing

Questions

1. What is an *implicit* cast? What is an *explicit* cast?
2. What are the rules around converting integer types?
3. What are the rules around converting floating-point types?
4. Why are some type-conversions impossible?
5. Is there any type which cannot be represented by an object variable?
6. How do we convert from object back to its actual type?
7. What is a two-step conversion from object?
8. What happens when we attempt a two-step conversion from object?
9. How can we test if an object is convertible to a type?
10. What is *boxing*?