msdn training

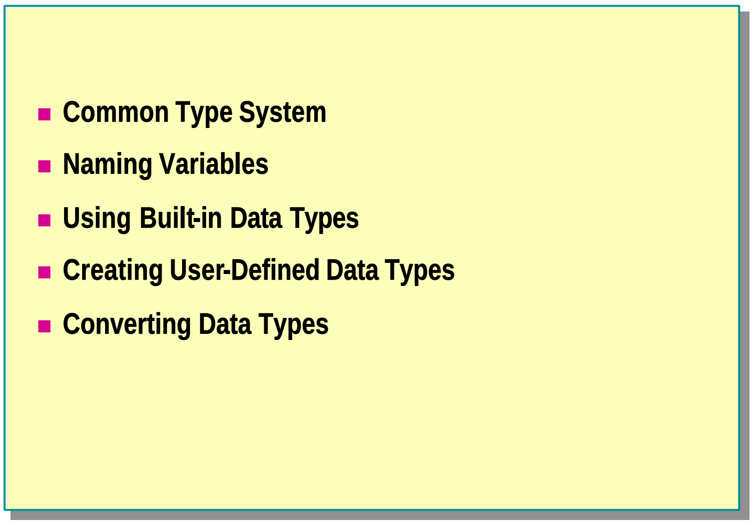
Module 3: Using Value-Type Variables

Contents

This course is based on the prerelease Beta 1 version of Microsoft® Visual Studio .NET. Content in the final release of the course may be different from the content included in this prerelease version. All labs in the course are to be completed with the Beta 1 version of Visual Studio .NET.

Microsoft

Overview



- Common Type System
- Naming Variables
- Using Built-in Data Types
- Creating User-Defined Data Types
- Converting Data Types

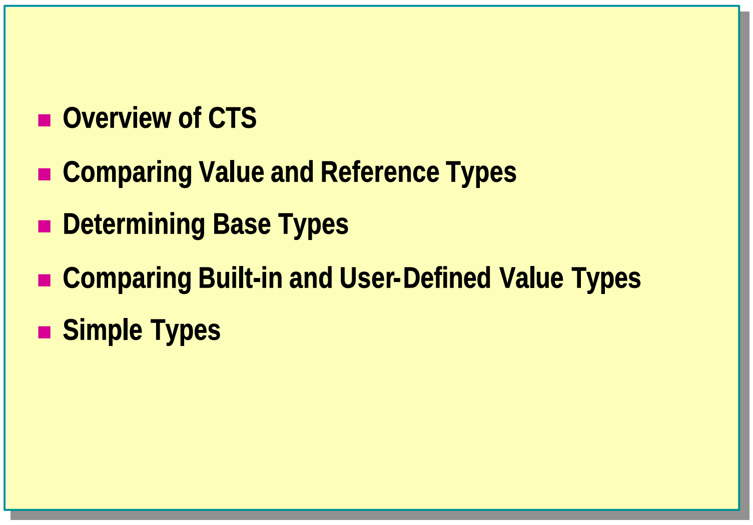
All applications manipulate data in some way. As a C# developer, you need to understand how to store and process data in your applications. Whenever your application needs to store data temporarily for use during execution, you store that data in a variable. Before you use a variable, you must define it. When you define a variable, you reserve some storage for that variable by identifying its data type and giving it a name. After a variable is defined, you can assign values to that variable.

In this module, you will learn how to use value-type variables in C#. You will learn how to specify the type of data that variables will hold, how to name variables according to standard naming conventions, and how to assign values to variables. You also will learn how to convert existing variables from one data type to another and how to create your own variables.

After completing this module, you will be able to:

- Describe the types of variables that you can use in C# applications.
- Name your variables according to standard C# naming conventions.
- Declare a variable by using built-in data types.
- Assign values to variables.
- Convert existing variables from one data type to another.
- Create and use your own data types.

◆ Common Type System

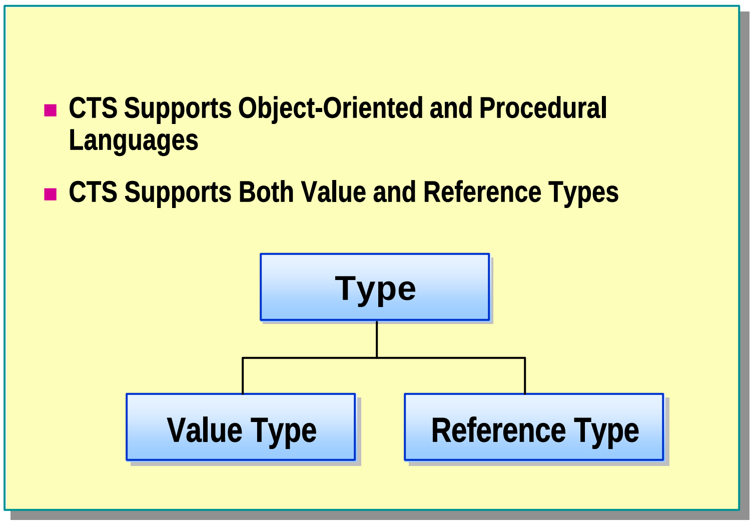


- Overview of CTS
- Comparing Value and Reference Types
- Determining Base Types
- Comparing Built-in and User-Defined Value Types
- Simple Types

Every variable has a data type that determines what values can be stored in the variable. C# is a type-safe language, meaning that the C# compiler guarantees that values stored in variables are always of the appropriate type.

The Common Language Runtime includes a Common Type System (CTS) that defines a set of built-in data types that you can use to define your variables. In this section, you will learn how the CTS works so that you can choose the appropriate data types for your variables. You also will see examples of value-type variables, including simple data types.

Overview of CTS



- CTS Supports Object-Oriented and Procedural Languages
- CTS Supports Both Value and Reference Types

When you define a variable, you need to choose the right data type for your variable. The data type determines the allowable values for that variable, which, in turn, determine the operations that can be performed on that variable.

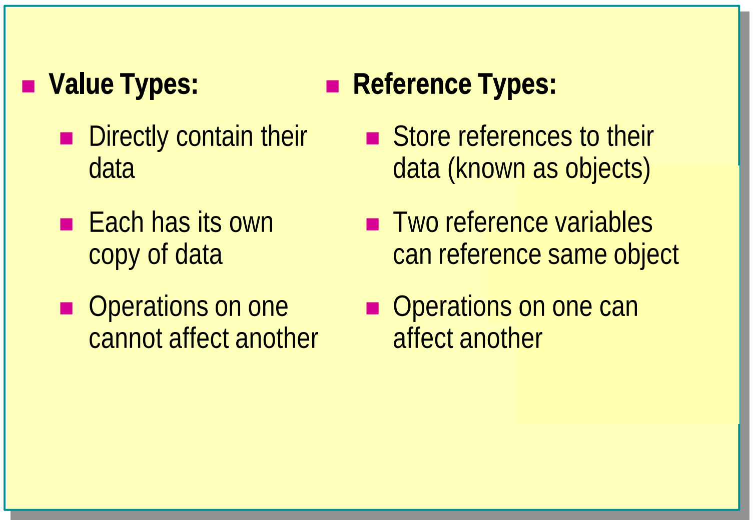
CTS

CTS is an integral part of the Common Language Runtime. The compilers, tools, and the runtime itself share CTS. It is the model that defines the rules that the runtime follows when declaring, using, and managing types. CTS establishes a framework that enables cross-language integration, type safety, and high-performance code execution.

C# defines several categories of variables. In this module, you will learn about two kinds:

- Value-type variables
- Reference-type variables

Comparing Value and Reference Types



- **Value Types:**
  - Directly contain their data
  - Each has its own copy of data
  - Operations on one cannot affect another
- **Reference Types:**
  - Store references to their data (known as objects)
  - Two reference variables can reference same object
  - Operations on one can affect another

Value Types

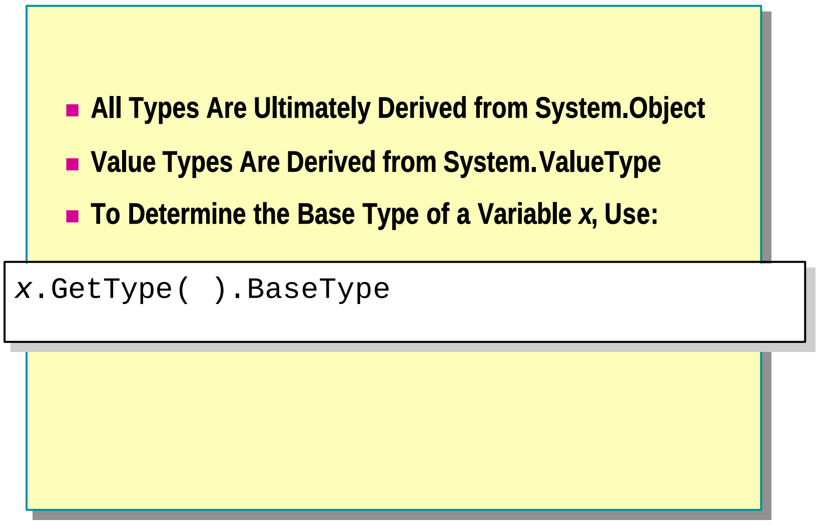
Value-type variables directly contain their data. Each value-type variable has its own copy of the data, so it is not possible for operations on one variable to affect another variable.

Reference Types

Reference-type variables contain references to their data. The data for reference-type variables is stored in an instance. It is possible for two reference-type variables to reference the same object, so it is possible for operations on one reference variable to affect the object referenced by another reference variable.

For more information about reference types, see Module 8, "Using Reference-Type Variables," in Course 2124A, *Introduction to C# Programming for the Microsoft .NET Platform (Prerelease)*.

Determining Base Types



- **All Types Are Ultimately Derived from System.Object**
- **Value Types Are Derived from System.ValueType**
- **To Determine the Base Type of a Variable *x*, Use:**

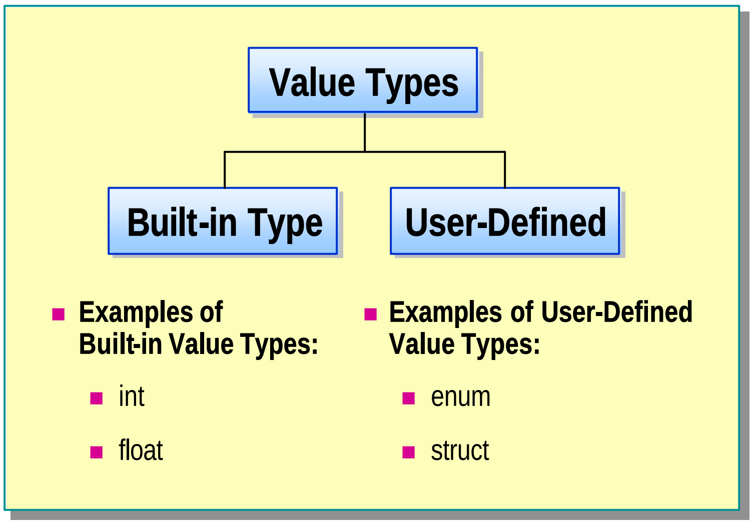
```
x.GetType( ).BaseType
```

All of the base data types are defined in the **System** namespace for C#. All types are ultimately derived from **System.Object**.

To determine the base data type of variable *x*, you can use the following code:

```
x.GetType( ).BaseType
```

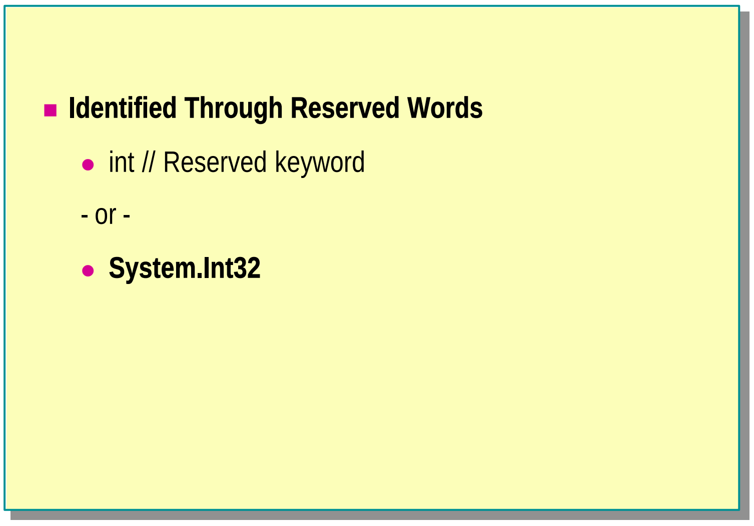
Comparing Built-in and User-Defined Value Types


Value Types
Built-in Type
User-Defined
Examples of Built-in Value Types:
int
float
Examples of User-Defined Value Types:
enum
struct

Value types include built-in and user-defined data types. The difference between built-in and user-defined types in C# is minimal because user-defined types can be used in the same way as built-in ones. The only real difference between built-in data types and user-defined data types is that you can write literal values for the built-in types. All value types directly contain data, and they cannot be **null**.

You will learn how to create user-defined data types such as enumeration and structure types in this module.

Simple Types



- **Identified Through Reserved Words**
  - int // Reserved keyword

  - or -

  - **System.Int32**

Built-in value types are also referred to as basic data types or simple types. Simple types are identified by means of reserved keywords. These reserved keywords are aliases for predefined structure types.

A simple type and the struct type it aliases are *completely indistinguishable*. In your code, you can use the reserved keyword or you can use the struct type. The following examples show both:

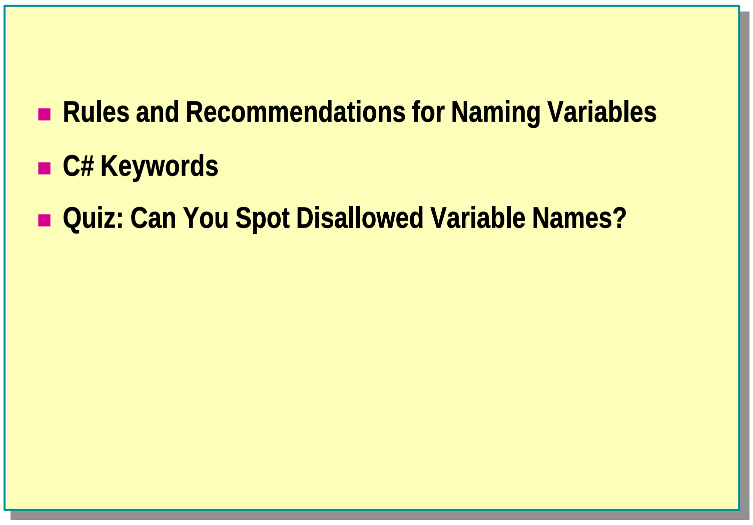
```
byte // Reserved keyword
--Or--
System.Byte // Struct type


int // Reserved keyword
--Or--
System.Int32 // Struct type
```

For more information about the sizes and ranges of built-in value types, search for " Value Types" in the Microsoft® Visual Studio.NET Help documents.

The following table lists common reserved keywords and their equivalent aliased struct type.

Reserved keywords	Alias for struct type
`sbyte`	`System.SByte`
`byte`	`System.Byte`
`short`	`System.Int16`
`ushort`	`System.UInt16`
`int`	`System.Int32`
`uint`	`System.UInt32`
`long`	`System.Int64`
`ulong`	`System.UInt64`
`char`	`System.Char`
`float`	`System.Single`
`double`	`System.Double`
`bool`	`System.Boolean`
`decimal`	`System.Decimal`

◆ Naming Variables



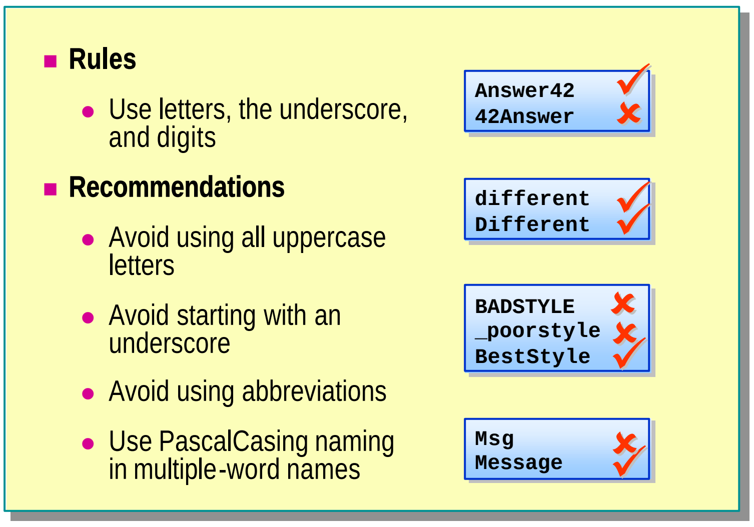
- Rules and Recommendations for Naming Variables
- C# Keywords
- Quiz: Can You Spot Disallowed Variable Names?

To use a variable, you first choose a meaningful and appropriate name for the variable. Each variable has a name that is also referred to as the *variable identifier.*

When naming variables, follow the standard naming conventions recommended for C#. You also need to be aware of the C# reserved keywords that you cannot use for variable names.

In this section, you will learn how to name your variables by following standard naming rules and recommendations.

Rules and Recommendations for Naming Variables



- **Rules**
  - Use letters, the underscore, and digits
- **Recommendations**
  - Avoid using all uppercase letters
  - Avoid starting with an underscore
  - Avoid using abbreviations
  - Use PascalCasing naming in multiple-word names

```
Answer42      ✓
42Answer      ✗

different     ✓
Different     ✓

BADSTYLE      ✗
_poorstyle    ✗
BestStyle     ✓

Msg           ✗
Message       ✓
```

When naming variables, observe the following rules and recommendations.

Rules

The following are the naming rules for C# variables:

- Start each variable name with a letter or underscore character.
- After the first character, use letters, digits, or the underscore character.
- Do not use reserved keywords.
- If you use a disallowed variable name, you will get a compile-time error.

Recommendations

It is recommended that you follow these recommendations when naming your variables:

- Avoid using all uppercase letters.
- Avoid starting with an underscore.
- Avoid using abbreviations.
- Use PascalCasing naming in multiple-word names.

PascalCasing Naming Convention

To use the PascalCasing naming convention, capitalize the first character of each word. Use PascalCasing for classes, methods, properties, enums, interfaces, fields, namespaces, and properties, as shown in the following example:

```
void InitializeData( );
```

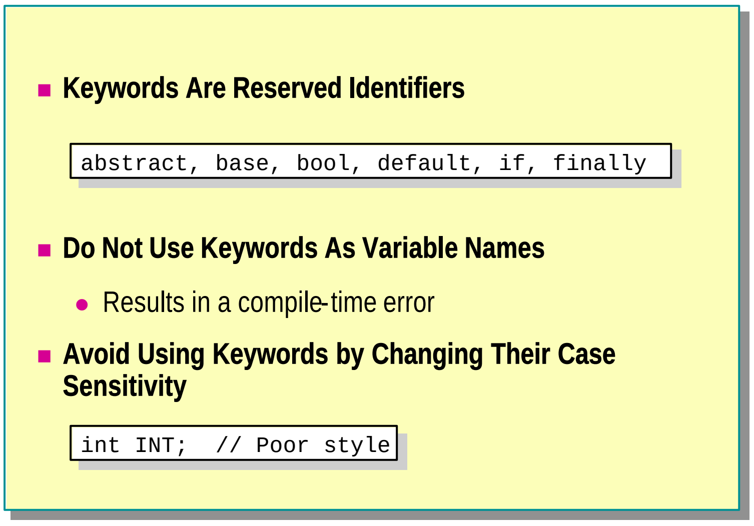
camelCasing Naming Convention

To use the camelCasing naming convention, capitalize the first character of each word except for the first word. Use camelCasing for variables that define fields and parameters, as shown in the following example:

```
int loopCountMax;
```

For more information about naming conventions, see "Naming Guidelines" in the .NET Framework SDK Help documents.

C# Keywords



- **Keywords Are Reserved Identifiers**

```
abstract, base, bool, default, if, finally
```

- **Do Not Use Keywords As Variable Names**
  - Results in a compile-time error
- **Avoid Using Keywords by Changing Their Case Sensitivity**

```
int INT;  // Poor style
```

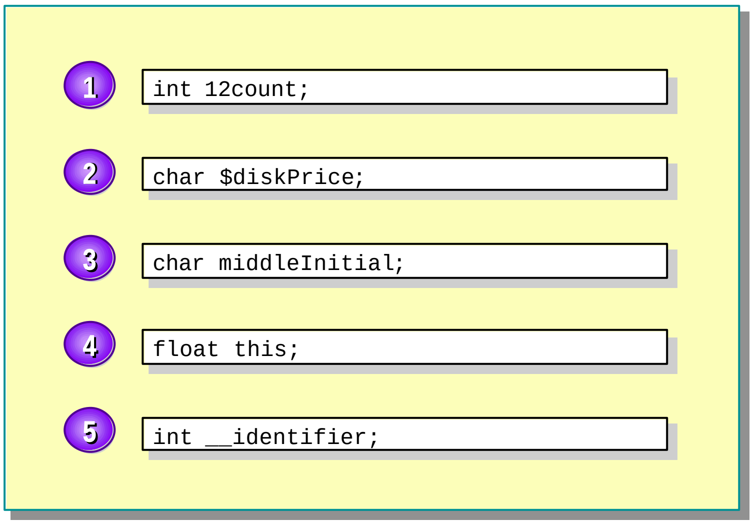
Keywords are reserved, which means that you cannot use any keywords as variable names in C#. Using a keyword as a variable name will result in a compile-time error.

Keywords in C#

The following is a list of keywords in C#. Remember, you cannot use any of these words as variable names.

abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	do	double	else	enum
event	explicit	extern	false	finally
fixed	float	for	foreach	goto
if	implicit	in	int	interface
internal	is	lock	long	namespace
new	null	object	operator	out
override	params	private	protected	public
readonly	ref	return	sbyte	sealed
short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void
while				

Quiz: Can You Spot the Disallowed Variable Names?

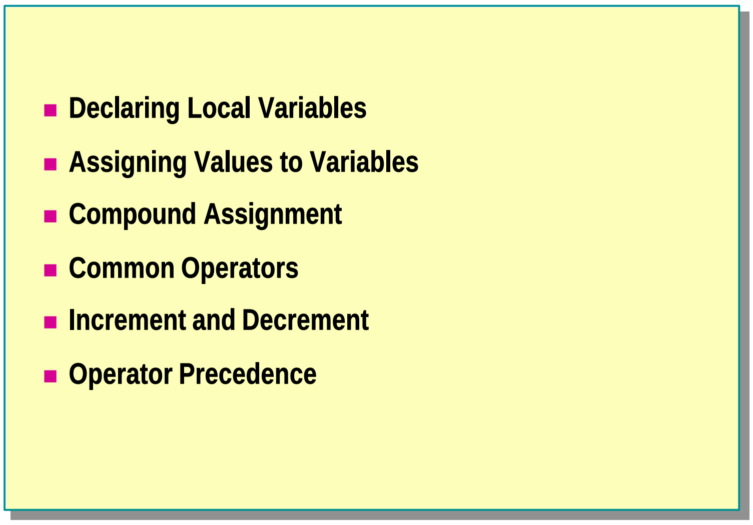


1.
```
int 12count;
```
2.
```
char $diskPrice;
```
3.
```
char middleInitial;
```
4.
```
float this;
```
5.
```
int __identifier;
```

Quiz Answers

1. **Disallowed** Variable names cannot begin with a digit.
2. **Disallowed** Variable names must start with a letter or an underscore.
3. **Allowed**. Variable names can start with a letter.
4. **Disallowed** Keywords (**this**) cannot be used to name variables.
5. **Allowed**. Variable names can start with an underscore.

◆ Using Built-in Data Types

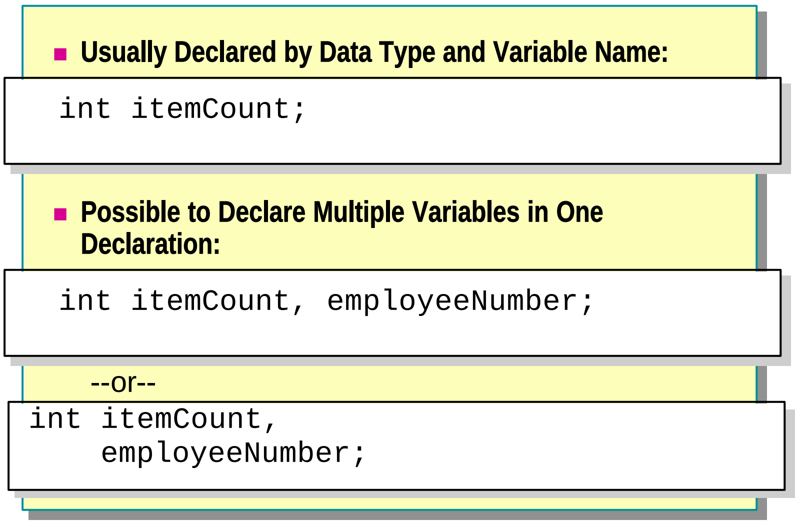


- Declaring Local Variables
- Assigning Values to Variables
- Compound Assignment
- Common Operators
- Increment and Decrement
- Operator Precedence

To create a variable, you must choose a variable name, declare your variable, and assign a value to your variable, unless it has already been automatically assigned a value by C#.

In this section, you will learn how to create a local variable by using built-in data types. You will also learn which variables are initialized, which variables are not initialized, how to use operators to assign values to variables, and how to define readonly variables and constants.

Declaring Local Variables



- **Usually Declared by Data Type and Variable Name:**

```
int itemCount;
```

- **Possible to Declare Multiple Variables in One Declaration:**

```
int itemCount, employeeNumber;
```

--or--

```
int itemCount,
    employeeNumber;
```

Variables that are declared in methods, properties, or indexers are called local variables. Generally, you declare a local variable by specifying the data type followed by the variable name, as shown in the following example:

```
int itemCount;
```

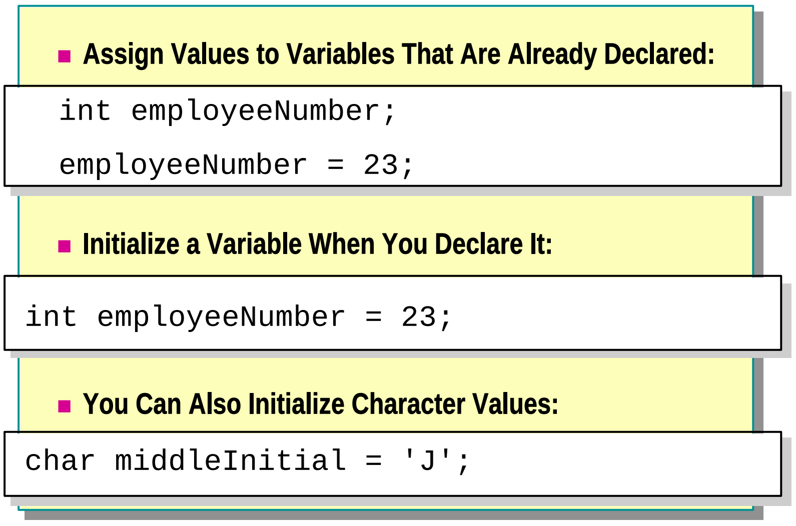
You can declare multiple variables in a single declaration by using a comma separator, as shown in the following example:

```
int itemCount, employeeNumber;
```

In C#, you cannot use uninitialized variables. The following code will result in a compile-time error because the *loopCount* variable has not been assigned an initial value:

```
int loopCount;
Console.WriteLine ("{0}", loopCount);
```

Assigning Values to Variables



- Assign Values to Variables That Are Already Declared:

```
int employeeNumber;
employeeNumber = 23;
```

- Initialize a Variable When You Declare It:

```
int employeeNumber = 23;
```

- You Can Also Initialize Character Values:

```
char middleInitial = 'J';
```

You use assignment operators to assign a new value to a variable. To assign a value to a variable that is already declared, use the assignment operator (=), as shown in the following example:

```
int employeeNumber;
employeeNumber = 23;
```

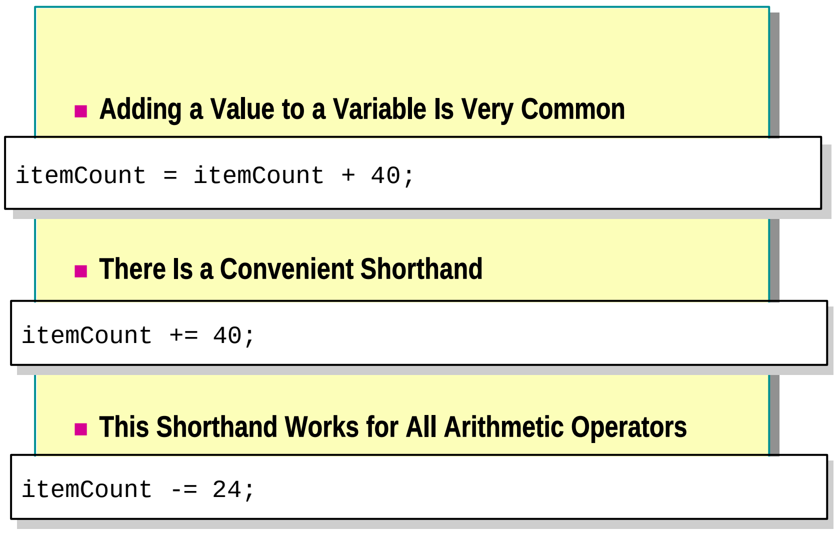
You can also initialize a variable when you declare it, as shown in the following example:

```
int employeeNumber = 23;
```

You can use the assignment operator to assign values to character type variables, as shown in the following example:

```
char middleInitial = 'J';
```

Compound Assignment



- Adding a Value to a Variable Is Very Common

```
itemCount = itemCount + 40;
```

- There Is a Convenient Shorthand

```
itemCount += 40;
```

- This Shorthand Works for All Arithmetic Operators

```
itemCount -= 24;
```

Adding a Value to a Variable Is Very Common

The following code declares an int variable called *itemCount*, assigns it the value 2, and then increments it by 40:

```
int itemCount;
itemCount = 2;
itemCount = itemCount + 40;
```

There Is a Convenient Shorthand

The code to increment a variable works, but it is slightly cumbersome. You need to write the identifier that is being incremented twice. For simple identifiers this is rarely a problem, unless you have many identifiers with very similar names. However, you can use expressions of arbitrary complexity to designate the value being incremented, as in the following example:

```
items[(index + 1) % 32] = items[(index + 1) % 32] + 40;
```

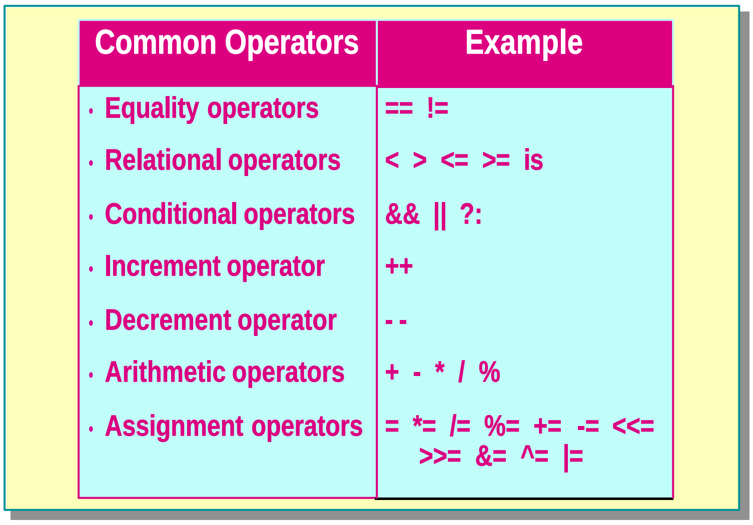
In these cases, if you needed to write the same expression twice you could easily introduce a subtle bug. Fortunately, there is a shorthand form that avoids the duplication:

```
itemCount += 40;
items[(index + 1) % 32] += 40;
```

This Shorthand Works for All Arithmetic Operators

```
var += expression; // var = var + expression
var -= expression; // var = var - expression
var *= expression; // var = var * expression
var /= expression; // var = var / expression
var %= expression; // var = var % expression
```

Common Operators



Common Operators	Example
Equality operators	== !=
Relational operators	< > <= >= is
Conditional operators	&& || ?:
Increment operator	++
Decrement operator	- -
Arithmetic operators	+ - * / %
Assignment operators	= *= /= %= += -= <<= >>= &= ^= |=

Expressions are constructed from *operands* and *operators*. The operators of an expression indicate which operations to apply to the operands.

Examples of operators include the concatenation and addition operator (+), the subtraction operator (-), the multiplication operator (*), and the division operator (/). Examples of operands include literals, fields, local variables, and expressions.

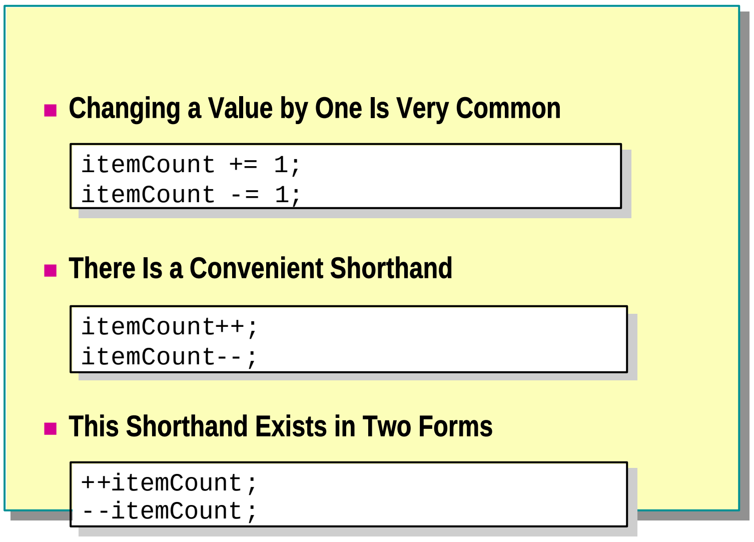
Common Operators

Some of the most common operators used in C# are described in the following table.

Type	Description
Assignment operators	Assign values to variables by using a simple assignment. For the assignment to succeed, the value on the right side of the assignment must be a type that can be implicitly converted to the type of the variable on the left side of the assignment.
Relational logical operators	Compare two values.
Logical operators	Perform bitwise operations on values.
Conditional operator	Selects between two expressions, depending on a Boolean value.
Increment operator	Increases the value of the variable by one.
Decrement operator	Decreases the value of the variable by one.
Arithmetic operators	Performs standard arithmetic operations.

For more information about the operators available in C#, see "Expressions" in the C# Language Specification in the Visual Studio.NET Help documents.

Increment and Decrement



- **Changing a Value by One Is Very Common**

```
itemCount += 1;
itemCount -= 1;
```

- **There Is a Convenient Shorthand**

```
itemCount++;
itemCount--;
```

- **This Shorthand Exists in Two Forms**

```
++itemCount;
--itemCount;
```

Changing a Value by One is Very Common

You often want to write a statement that increments or decrements a value by one. You could do this as follows:

```
itemCount = itemCount + 1;
itemCount = itemCount – 1;
```

However, as just explained, there is a convenient shorthand for this:

```
itemCount += 1;
itemCount -= 1;
```

This shorthand form is the preferred idiomatic way for C# programmers to increment or decrement a value.

Convenient Shorthand

Incrementing or decrementing a value by one is so common, that this shorthand method has an even shorter shorthand form!

```
itemCount++; // itemCount += 1;
itemCount--; // itemCount -= 1;
```

The ++ operator is called the increment operator and the – operator is called the decrement operator. You can think of ++ as an operator that changes a value to its successor and – as an operator that changes a value to its predecessor.

Once again, this shorthand is the preferred idiomatic way for C# programmers to increment or decrement a value by one.

Note C++ is called C++ because it was the successor to C!

This Shorthand Exists in Two Forms

You can use the ++ and – operators in two forms.

1. You can place the operator symbol *before* the identifier, as shown in the following examples. This is called the *prefix* notation.

   ```
   ++itemCount;
   --itemCount;
   ```

2. You can place the operator symbol *after* the identifier, as shown in the following examples. This is called the *postfix* notation.

   ```
   itemCount++;
   itemCount--;
   ```

In both cases, the *itemCount* is incremented (for ++) or decremented (for --) by one. So why have two notations? To answer this question, you first need to understand assignment in more detail:

An important feature of C# is that assignment is an operator. This means that besides assigning a value to a variable, an assignment expression itself has a value, or outcome, which is the value of the variable after the assignment has taken place. In most statements the value of the assignment expression is discarded, but it can be used in a larger expression, as in the following example:

```
int itemCount = 0;
Console.WriteLine(itemCount = 2); // Prints 2
Console.WriteLine(itemCount = itemCount + 40); // Prints 42
```

Compound assignment is also an assignment. This means that a compound assignment expression, besides assigning a value to a variable, also has a value—an outcome itself. Again, in most statements the value of the compound assignment expression is discarded, but it can be used in a larger expression, as in the following example:

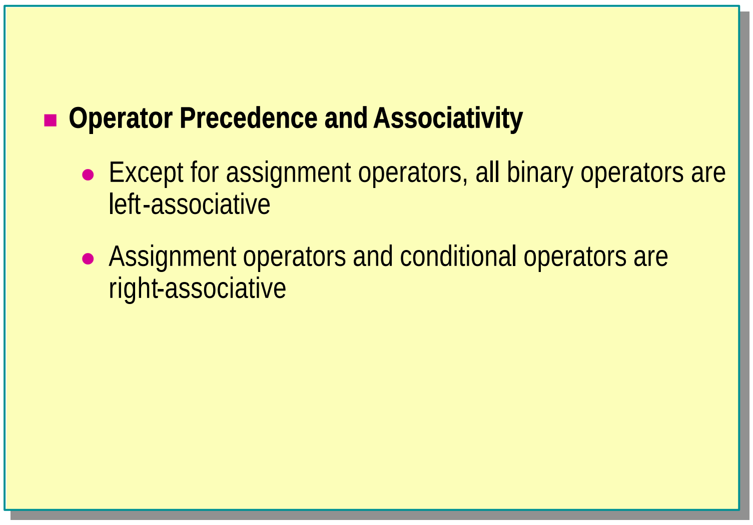
```
int itemCount = 0;
Console.WriteLine(itemCount += 2); // Prints 2
Console.WriteLine(itemCount -= 2); // Prints 0
```

Increment and decrement are also assignments. This means, for example, that an increment expression, besides incrementing a variable by one, also has a value, an outcome itself. Again, in most statements the value of the increment expression is discarded, but again it can be used in a larger expression, as in the following example:

```
int itemCount = 42;
int prefixValue = ++itemCount;  // prefixValue == 42
int postfixValue = itemCount++; // postfixValue = 44
```

The value of the increment expression differs depending on whether you are using the prefix or postfix version. In both cases itemCount is incremented. That is not the issue. The issue is what is the value of the increment expression. The value of a prefix increment/decrement is the value of the variable *before* the increment/decrement takes place. The value of a postfix increment/decrement is the value of the variable *after* the increment/decrement takes place.

Operator Precedence



- **Operator Precedence and Associativity**
  - Except for assignment operators, all binary operators are left-associative
  - Assignment operators and conditional operators are right-associative

Operator Precedence

When an expression contains multiple operators, the precedence of the operators controls the order in which the individual operators are evaluated. For example, the expression x + y * z is evaluated as x + (y * z) because the multiplicative operator has higher prec edence than the additive operator. For example, an *additive-expression* consists of a sequence of *multiplicative-expressions* separated by + or - operators, thus giving the + and - operators lower precedence than the *, /, and % operators.

Associativity

When an expression contains the same operator many times, the associativity controls the order in which the operators are performed. For example, x + y + z is evaluated as (x + y) + z. This is particularly important for assignment operators. For example, x = y = z is evaluated as x = (y = z).

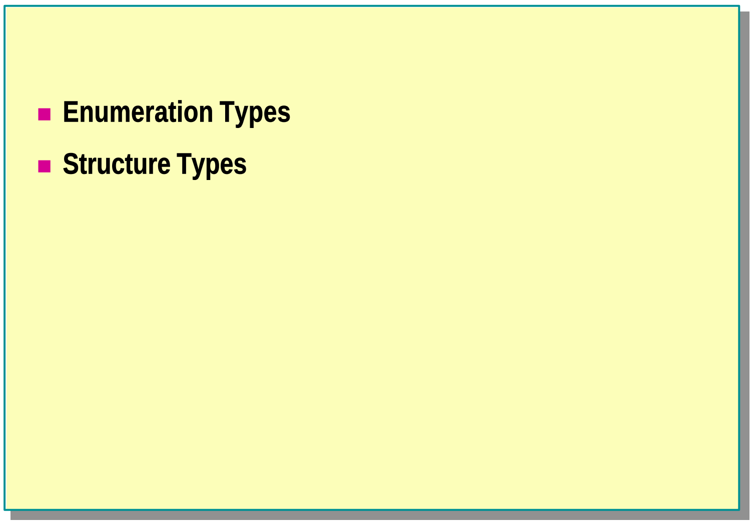
- Except for the assignment operators, all binary operators are *left-associative*, meaning that operations are performed from left to right.
- The assignment operators and the conditional operator (?:) are *right-associative*, meaning that operations are performed from right to left.

You can control precedence and associativity by using parentheses. For example, x + y * z first multiplies y by z and then adds the result to x, but (x + y) * z first adds x and y and then multiplies the result by z.

The following table summarizes operators in order of precedence, from highest to lowest.

Category	**Operators**
Primary	(x) x.y f(x) a[x] x++ x-- new typeof sizeof checked unchecked
Unary	+ - ! ~ ++x --x (T)x
Multiplicative	* / %
Additive	+ -
Shift	<< >>
Relational	< > <= >= is
Equality	== !=
Logical AND	&
Logical XOR	^
Logical OR	|
Conditional AND	&&
Conditional OR	||
Conditional	?:
Assignment	= *= /= %= += -= <<= >>= &= ^= |=

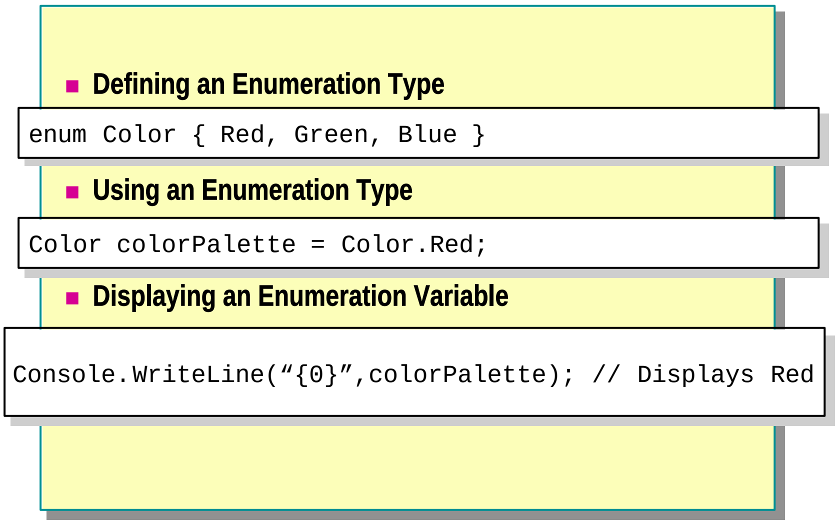
◆ Creating User-Defined Data Types



- Enumeration Types
- Structure Types

In this section, you will learn how to create user-defined enumeration (enum) and structure (struct) data types.

Enumeration Types



- Defining an Enumeration Type

```
enum Color { Red, Green, Blue }
```

- Using an Enumeration Type

```
Color colorPalette = Color.Red;
```

- Displaying an Enumeration Variable

```
Console.WriteLine(“{0}”,colorPalette); // Displays Red
```

Enumerators are useful when a variable can only have a specific set of values.

Defining an Enumeration Type

To declare an enumeration, use the **enum** keyword followed by the enum variable name and initial values. For example, the following enumeration defines three integer constants, called enumerator values.

```
enum Color { Red, Green, Blue }
```

By default, enumerator values start from 0. In the preceding example, *Red* has a value of 0, *Green* has a value of 1, and *Blue* has a value of 2.

You can initialize an enumeration by specifying integer literals.

Using an Enumeration Type

You can declare a variable *colorPalette* of **Color** type by using the following syntax:

```
Color colorPalette;       // Declare the variable
colorPalette = Color.Red; // Set value
```

- Or -

```
colorPalette = (Color)0; // Type casting int to Color
```

Displaying an Enumeration Value

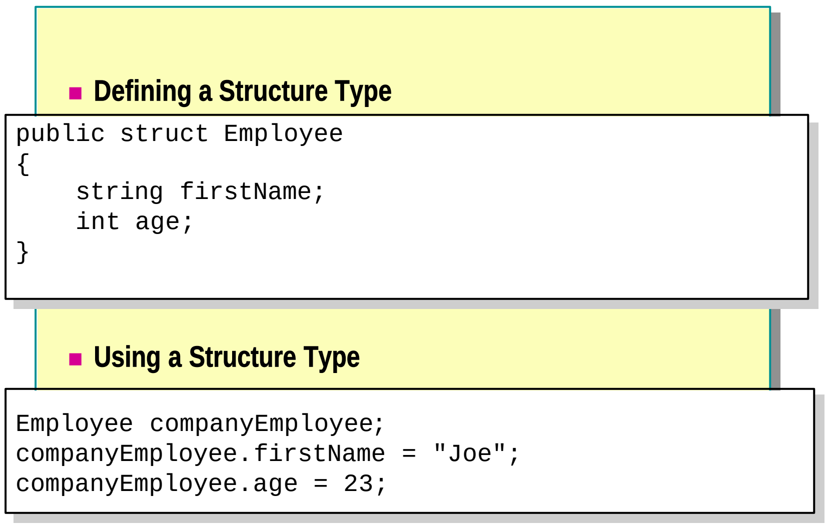
To display an enumeration value in readable format, use the following statement:

```
Console.WriteLine("{0}",colorPalette);
```

Alternatively, you can use the format method as shown in the following example:

```
Console.WriteLine(colorPalette.Format());
```

Structure Types



- Defining a Structure Type

```
public struct Employee
{
    string firstName;
    int age;
}
```

- Using a Structure Type

```
Employee companyEmployee;
companyEmployee.firstName = "Joe";
companyEmployee.age = 23;
```

You can use structures to create objects that behave like built-in value types. Because structs are stored inline and are not heap allocated, there is less garbage collection pressure on the system than there is with classes.

In the .NET Framework, simple data types such as int, float, and double are all built-in structures.

Defining a Structure Type

You can use a structure to group together several arbitrary types, as shown in the following example:

```
public struct Employee
{
    string firstName;
    int   age;
}
```

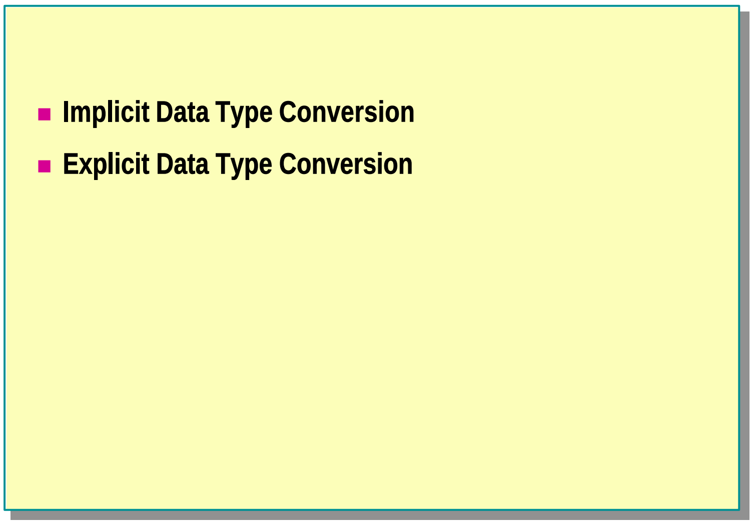
This code defines a new type called **Employee** that consists of two elements: first name and age.

Using a Structure Type

To access elements inside the struct, use the following syntax:

```
Employee companyEmployee;          // Declare variable
companyEmployee.firstName =  "Joe" // Set value
companyEmployee.age = 23;
```

◆ Converting Data Types



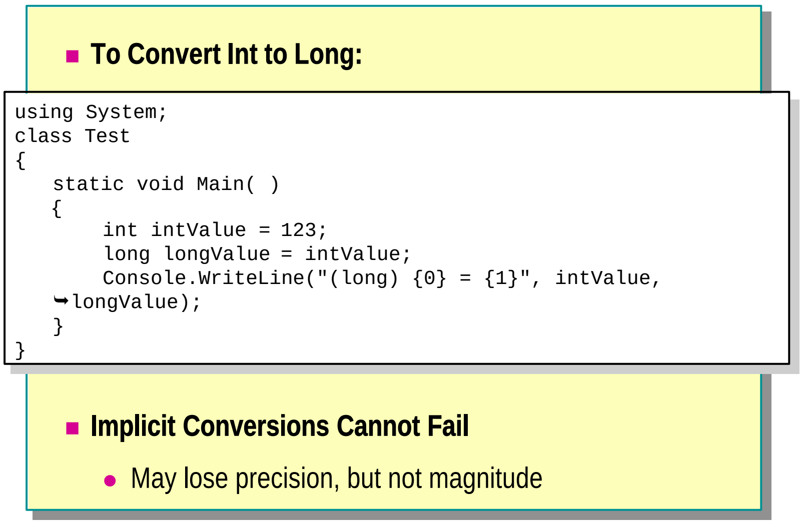
- Implicit Data Type Conversion
- Explicit Data Type Conversion

In C#, there are two types of conversion:

- Implicit data type conversion
- Explicit data type conversion

You will see examples of how to perform both implicit and explicit data conversion in this section.

Implicit Data Type Conversion



- **To Convert Int to Long:**

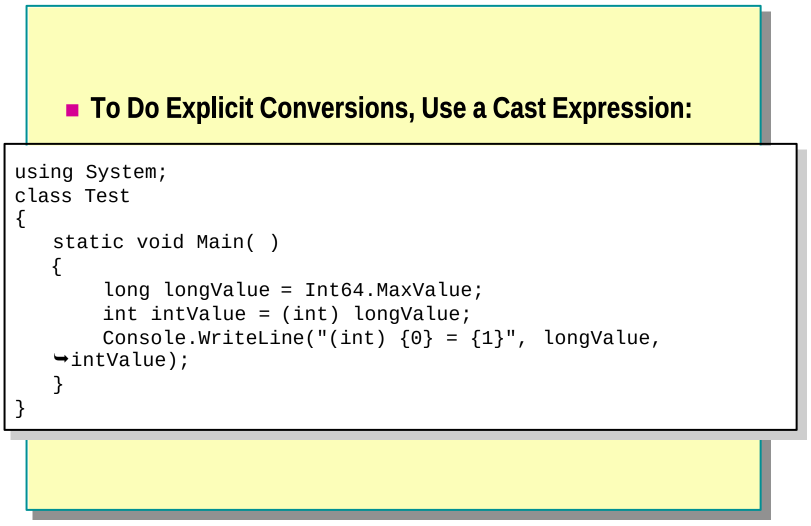
```
using System;
class Test
{
   static void Main( )
   {
       int intValue = 123;
       long longValue = intValue;
       Console.WriteLine("(long) {0} = {1}", intValue,
   ➥longValue);
   }
}
```

- **Implicit Conversions Cannot Fail**
  - May lose precision, but not magnitude

Converting from an int data type to a long data type is implicit. This conversion always succeeds, and it never results in a loss of information. The following example shows how to convert the variable *intValue* from an int to a long:

```
using System;
class Test
{
    static void Main( )
    {
        int intValue = 123;
        long longValue = intValue;
        Console.WriteLine("(long) {0} = {1}", intValue,
➥longValue);
    }
}
```

Explicit Data Type Conversion



- To Do Explicit Conversions, Use a Cast Expression:

```
using System;
class Test
{
   static void Main( )
   {
       long longValue = Int64.MaxValue;
       int intValue = (int) longValue;
       Console.WriteLine("(int) {0} = {1}", longValue,
   ➥intValue);
   }
}
```

You can convert variable types explicitly by using a cast expression. The following example shows how to convert the variable *longValue* from a long data type to an int data type by using a cast expression:

```
using System;
class Test
{
    static void Main( )
    {
        long longValue = Int64.MaxValue;
        int intValue = (int) longValue;
        Console.WriteLine("(int) {0} = {1}", longValue,
➥intValue);
    }
}
```

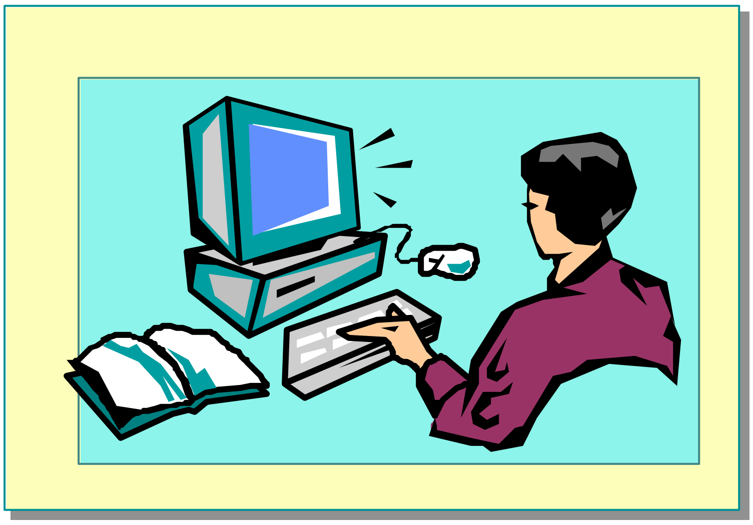
Because an overflow occurs in this example, the output is as follows:

```
(int) 9223372036854775807 = -1
```

To avoid such a situation, you can use the **checked** statement to raise an exception when a conversion fails, as follows:

```
using System;
class Test
{
  static void Main( )
  {
     checked
     {
        long longValue = Int64.MaxValue;
        int intValue = (int) longValue;
        Console.WriteLine("(int) {0} = {1}", longValue,
➥intValue);
     }
  }
}
```

Lab 3: Creating and Using Types



Objectives

After completing this lab, you will be able to:

- Create new data types.
- Define and use variables.

Prerequisites

Before working on this lab, you should be familiar with the following:

- The Common Type System
- Value-type variables in C#

Scenario

In Exercise 1, you will write a program that creates a simple **enum** type and then sets and prints the values by using the **Console.WriteLine** statement.

In Exercise 2, you will write a program that uses the **enum** type declared in Exercise 1 in a **struct**.

If time permits, you will add input/output functionality to the program you wrote in Exercise 2.

Starter and Solution Files

There are starter and solution files associated with this lab. The starter files are in the *install folder*\Labs\Lab03\Starter folder and the solution files are in the *install folder*\Labs\Lab03\Solution folder.

Estimated time to complete this lab: 60 minutes

Exercise 1
Creating an enum Type

In this exercise, you will create an enumerated type for representing different types of bank accounts (checking and savings). You will create two variables by using this **enum** type, and set the values of the variables to Checking and Deposit. You will then print the values of the variables by using the **System.Console.WriteLine** function.

↙ To create an enum type

1. Open the Enum.cs file in the *install folder*\Labs\Lab03\Starter\BankAccount folder.
2. Add an **enum** called **AccountType** before the class definition as follows:

   ```
   public enum AccountType { Checking, Deposit }
   ```

   This **enum** will contain Checking and Deposit types.
3. Declare two variables of type **AccountType** in **Main** as follows:

   ```
   AccountType goldAccount;
   AccountType platinumAccount;
   ```

4. Set the value of the first variable to Checking and the value of the other variable to Deposit as follows:

   ```
   goldAccount = AccountType.Checking;
   platinumAccount = AccountType.Deposit;
   ```

5. Add two **Console.WriteLine** statements to print the value of each variable as follows:

   ```
   Console.WriteLine("The Customer Account Type is
   {0}",goldAccount);
   Console.WriteLine("The Customer Account Type is
   {0}",platinumAccount);
   ```

6. Compile and run the program.

Exercise 2
Creating and Using a Struct Type

In this exercise, you will define a **struct** that can be used to represent a bank account. You will use variables to hold the account number (a **long**), the account balance (a **decimal**), and the account type (the **enum** that you created in Exercise 1). You will create a **struct** type variable, populate the **struct** with some sample data, and print the result.

⇙ To create a struct type

1. Open the Struct.cs file in the *install folder*\Labs\Lab03\Starter\StructType folder.
2. Add a **public struct** called **BankAccount** that contains the following fields.

Type	Variable
public long	*accNo*
public decimal	*accBal*
public AccountType	*accType*

3. Declare a variable *goldAccount* of type **BankAccount** in **Main**.

```
BankAccount goldAccount;
```

4. Set the *accType*, *accBal*, and *accNo* fields of the variable *goldAccount*.

```
goldAccount.accType = AccountType.Checking;
goldAccount.accBal = (decimal)3200.00;
goldAccount.accNo = 123;
```

5. Add **Console.WriteLine** statements to print the value of each element in the struct variable.

```
Console.WriteLine("Acct Number  {0}", goldAccount.accNo);
Console.WriteLine("Acct Type    {0}", goldAccount.accType);
Console.WriteLine("Acct Balance ${0}",goldAccount.accBal);
```

6. Compile and run the program.

If Time Permits
Adding Input/Output functionality

In this exercise, you will modify the code written in Exercise 2. Instead of using the account number 123, you will prompt the user to enter the account number. You will use this number to print the account summary.

To add input/output functionality

1. Open the StructType.cs file in the *install folder*\Labs\Lab03\Starter\Optional folder.
2. Add a **Console.Write** statement to prompt the user to enter the account number.

```
Console.Write("Enter account number: ");
```

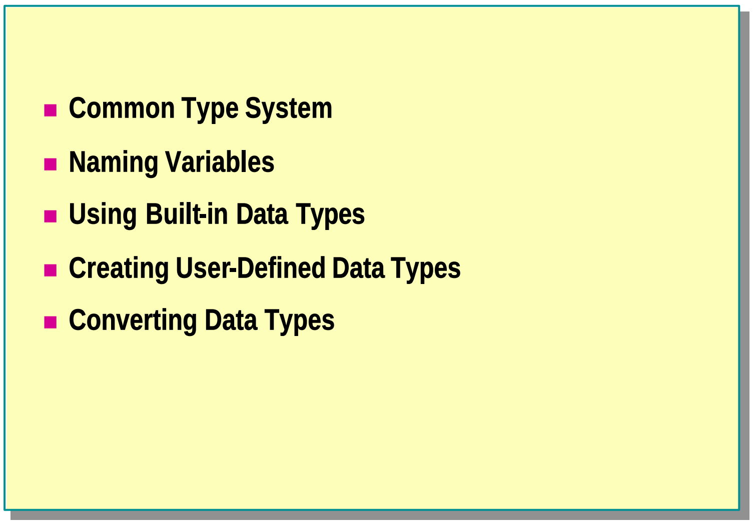
3. Read the account number by using a **Console.ReadLine** statement. Assign this value to *goldAccount.accNo*.

```
goldAccount.accNo = long.Parse(Console.ReadLine());
```

Note You need to use the **long.Parse** method to convert the string read by the **Console.ReadLine** statement into a decimal value before assigning it to *goldAccount.accNo*.

4. Compile and run the program.

Review



- Common Type System
- Naming Variables
- Using Built-in Data Types
- Creating User-Defined Data Types
- Converting Data Types

1. What is the Common Type System?

2. Can a value type be **null**?

3. Can you use uninitialized variables in C#? Why?

4. Can there be loss of magnitude as a result of an implicit conversion?