C# Programming
Data and Types

Data in Programs

- Programs are made up of data and operations that work on that data
- C# programs contain *variables* that hold the data to be processed
- The program must declare each variable before it is used
- Variables are of a particular data *type*

Declaring variables

```
...
static void Main()
{
      double width, height, woodLength, glassArea;
      string widthString, heightString;
...
```

- A variable is a place you can store things
- You can think of it as a box with a name

Variables

- Boxes to put things in
- Hold items of a particular **type**
  - The type of the box determines what you can put in it
    - integer, double, float, string
- Converting between different types is not always automatic
  - We have to explicitly convert between string and double
- You choose the **identifier** for your variables

Identifiers

- A string of text you use to identify something that you have created
  - Starts with a letter or _
  - Contains letters, numbers and _ characters:

    `Fred Height99 The_Score theScore`
- The identifier should reflect what the variable is being used for:

  `width` and **`height`**

Variable Types

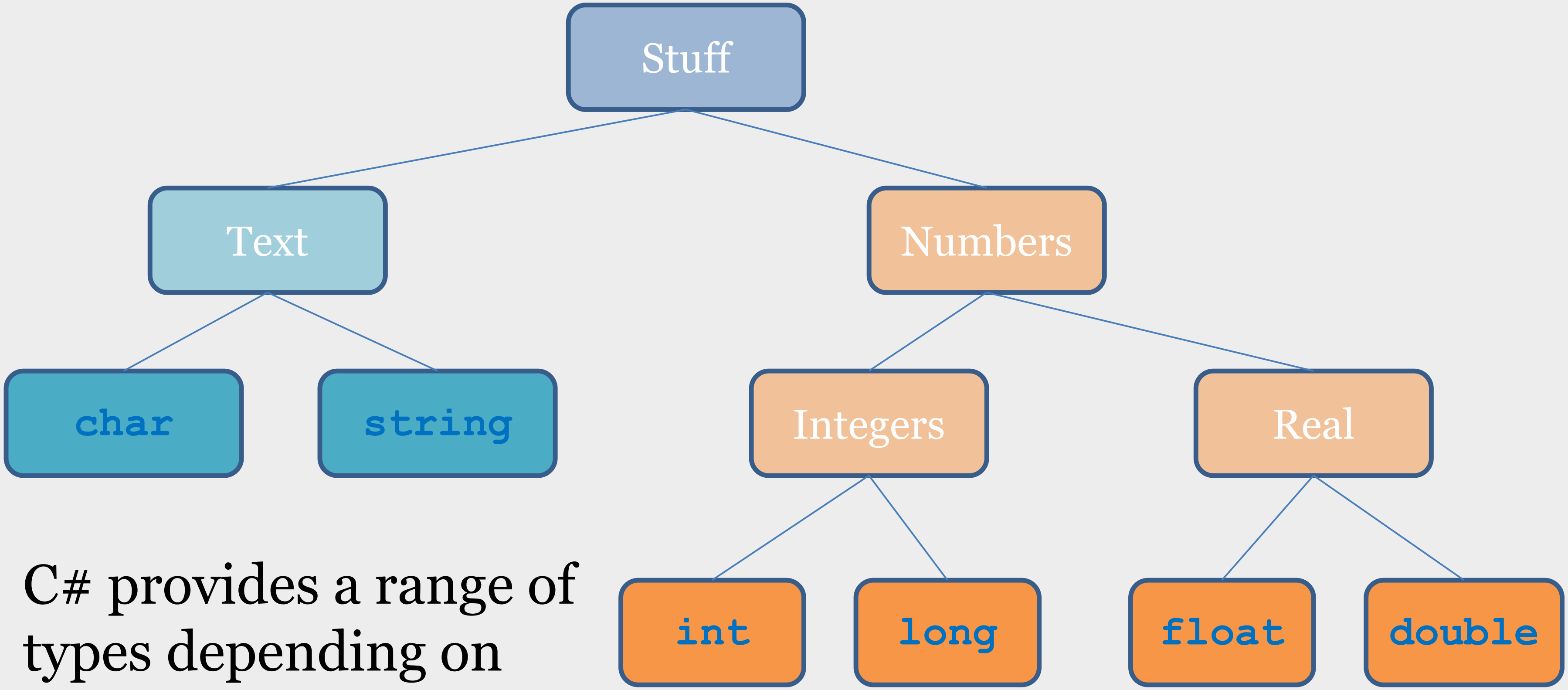
```
...
static void Main()
{
        double width, height, woodLength, glassArea;
        string widthString, heightString;
...
```

- The program stores numbers for the input values and output values
- It also stores the strings entered by the user

Variables and Type

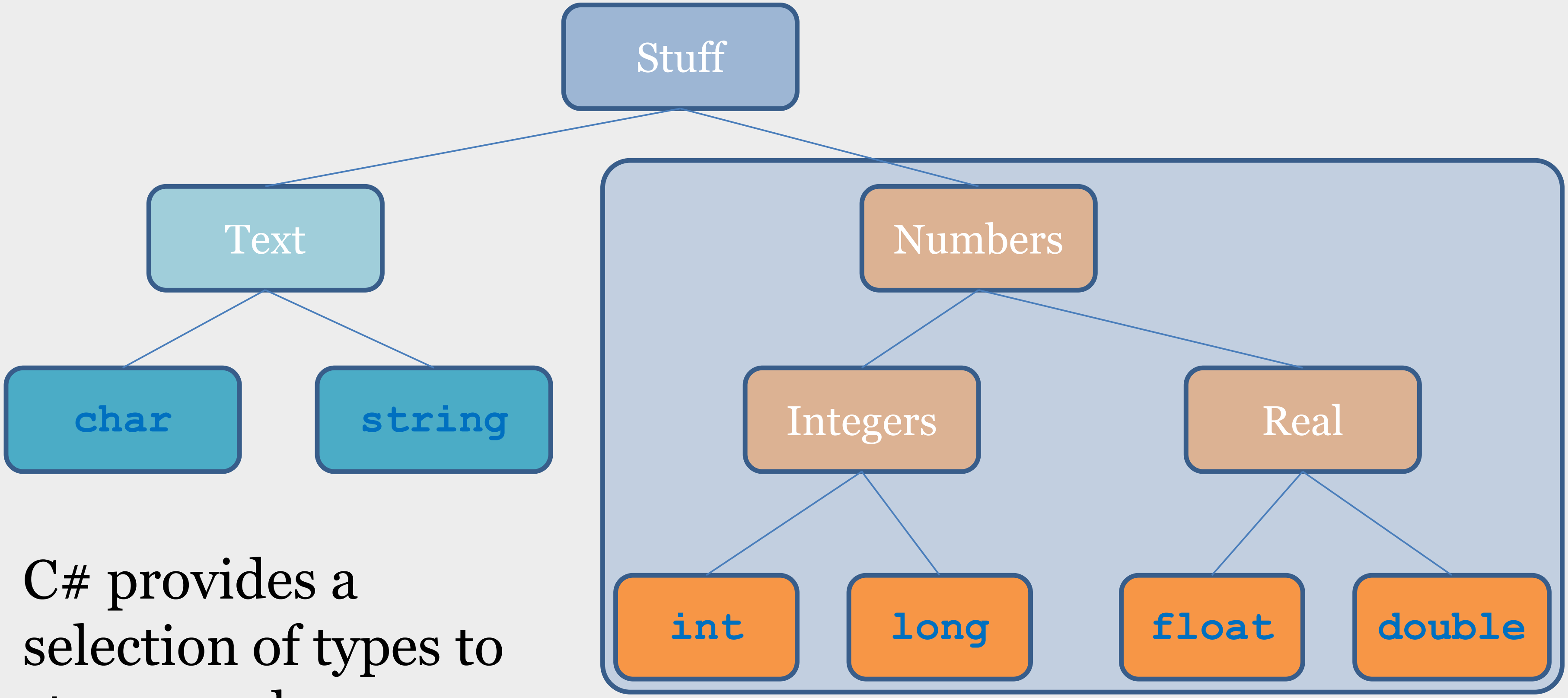
- The *type* of a variable determines what a program can store in it
- The C# language is *strongly typed* in that the compiler will prevent you from combining data types ways it things are wrong
- This is to make programs more reliable

C# Data Types



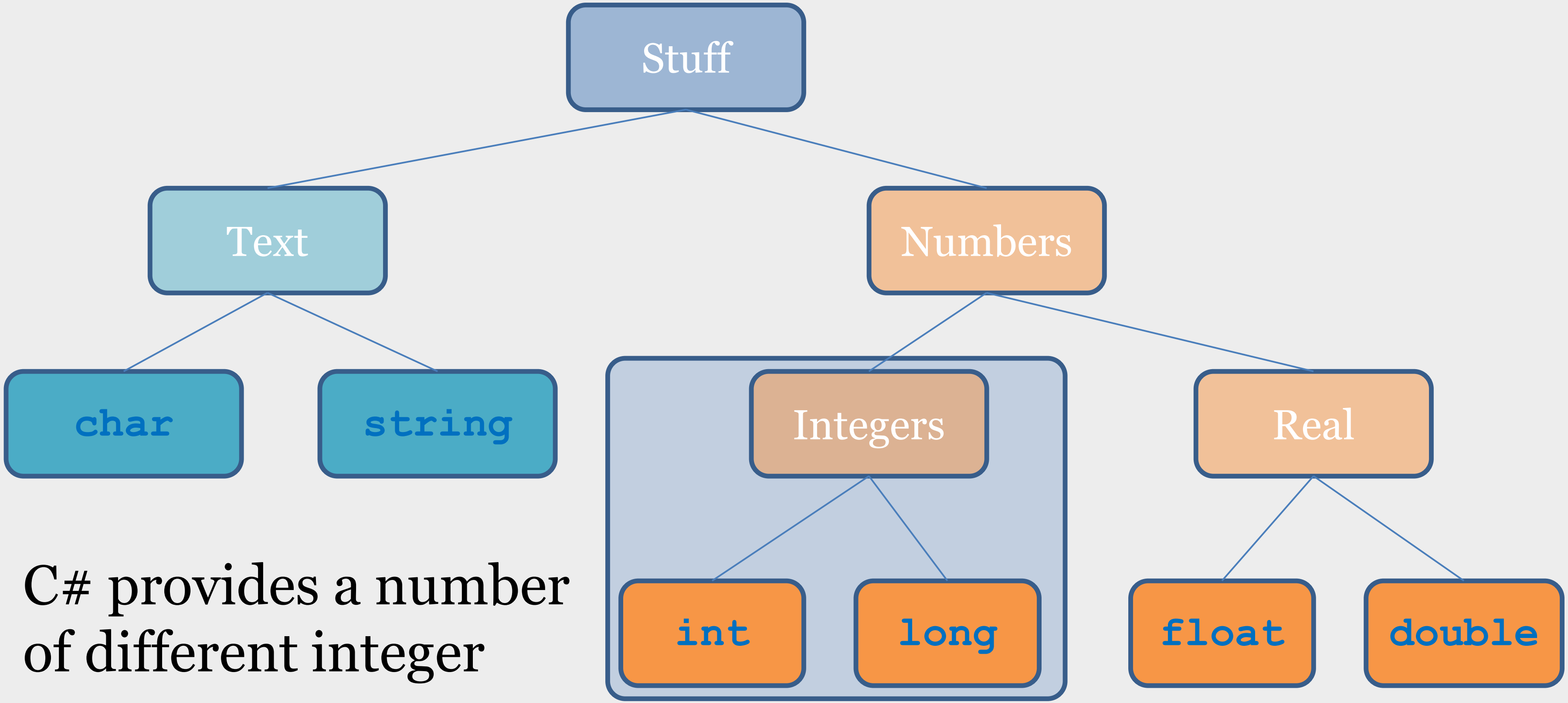
C# provides a range of types depending on what is to be stored

Numeric Data Types



C# provides a selection of types to store numbers

Integer values



C# provides a number of different integer types

Using Integers

- If you have no need to store fractions, you should use integers
- Computers can manipulate integers more quickly than floating point
  - This is particularly true for smaller devices
- Even things that you think should be real numbers can often be integer
  - The price of something can be stored in pence

Storing Integer Values

- Integer values are held exactly
  - i.e. the pattern of bits held in computer memory exactly matches the integer value it is supposed to represent
- The more bits that are used to hold an integer value, the greater the range
- Integers use “2’s complement” notation to hold negative numbers

C# Integer Variable Types

`sbyte`	8 bits	-128 to 127
`byte`	8 bits	0 to 255
`short`	16 bits	-32768 to 32767
`ushort`	16 bits	0 to 65535
`int`	32 bits	-2147483648 to 2147483647
`uint`	32 bits	0 to 4294967295
`long`	64 bits	-9223372036854775808 to 9223372036854775807
`ulong`	64 bits	0 to 18446744073709551615
`char`	16 bits	0 to 65535

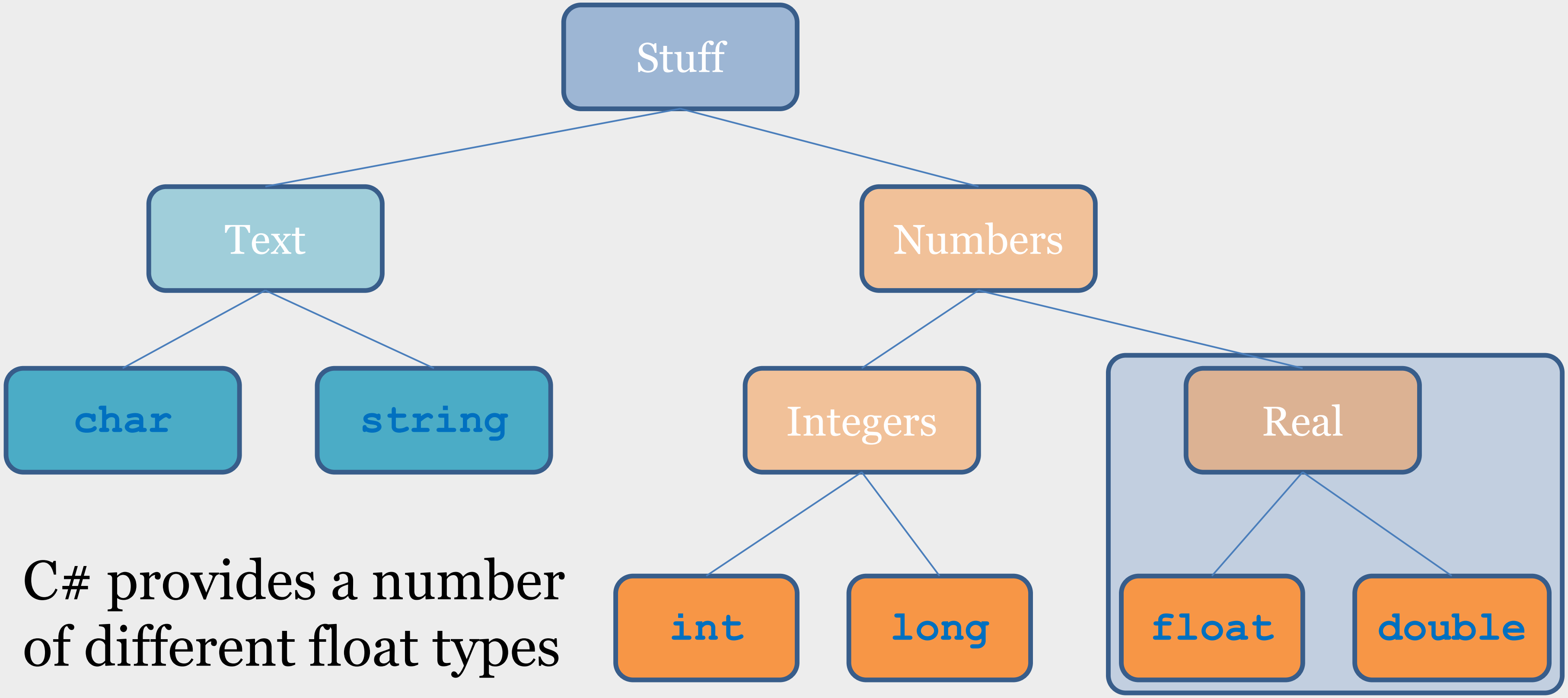
- These are the integer types provided by C#
- Note that the unsigned types do not store negative numbers

Integer “literals” in C#

```
int i;
i = 99;
byte b;
b = 100;
```

- A “literal” is a value in the program that is literally “just there”
- In C# program a integer literal value is given with no decimal point

Real values



C# provides a number of different float types

Using Real Numbers

- Real numbers are used when you need a fractional part
  - Working out averages
  - Any kind of real world calculations
- C# provides a range of real number types which have different range and precision
- You choose the one that fits the problem

Range and Precision

- Floating point values are held in C# to a particular *range* and *precision*
  - Range: the biggest and smallest numbers I can store
  - Precision: the number of digits of accuracy available
- Each type has a particular range and precision

Storing Real Numbers

- Real numbers are held as “binary fractions”
- The value ¾ would be represented as:
  - “a half plus a quarter”
- This means that the value 0.1 (a tenth) cannot be represented exactly on a computer in this way
- Instead we use enough bits to ensure that values are held sufficiently accurately

C# Real Variable Types

`float`	32 bits	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$ 7 digits of precision
`double`	64 bits	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$ 15 digits of precision
`decimal`	128 bits	$\pm 1.0 \times 10^{-28}$ to $\pm 7.9 \times 10^{28}$ 28 digits of precision

- These are the real types provided by C#
- decimal is provided for use in high precision finance calculations

Float “literals” in C#

```
double d;
d = 0.1;
float f;
f = 0.1f;
```

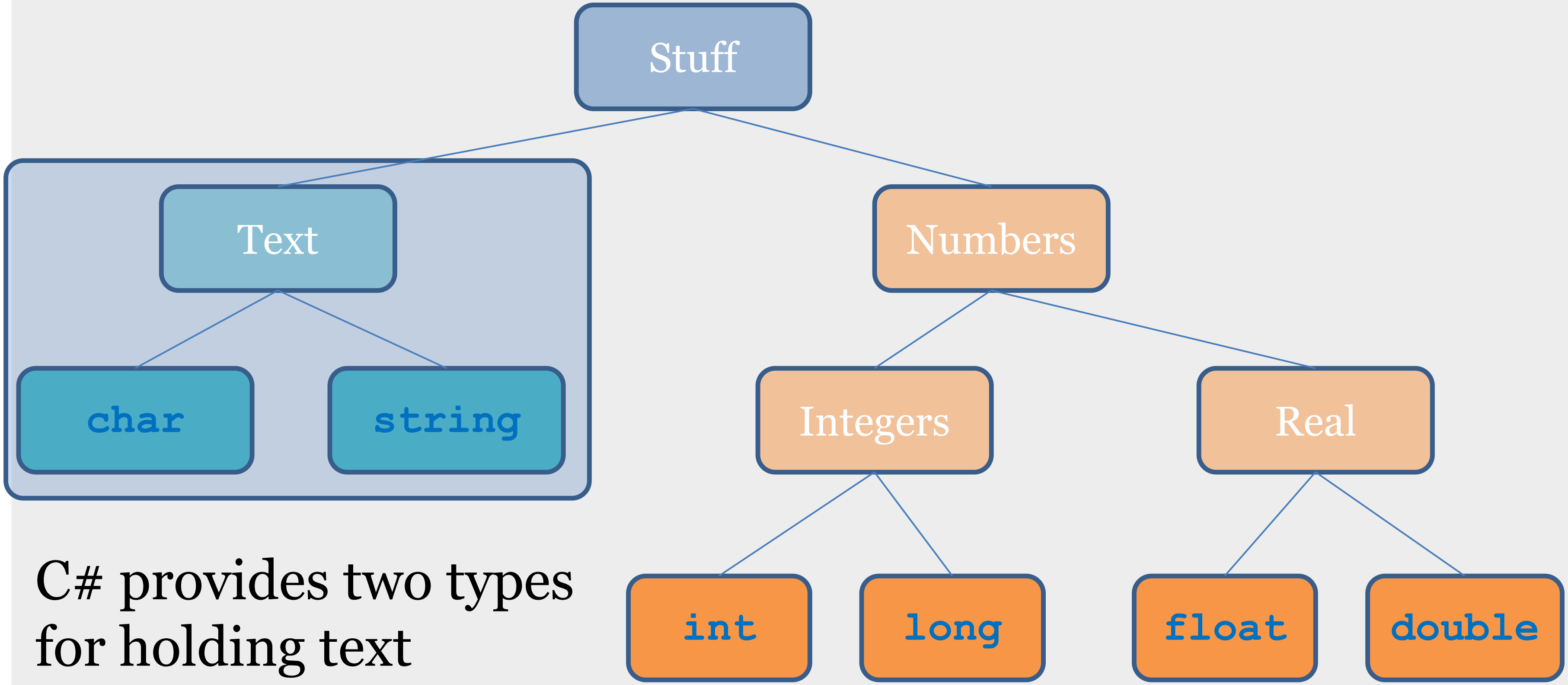
- A literal floating point value is always treated as if it was of `double` type by the compiler
- To create a literal value of type `float` you have to add an f on the end of the literal value

Float “literals” in C#

```
double d;
d = 0.1;
float f;
f = 0.1f;
```

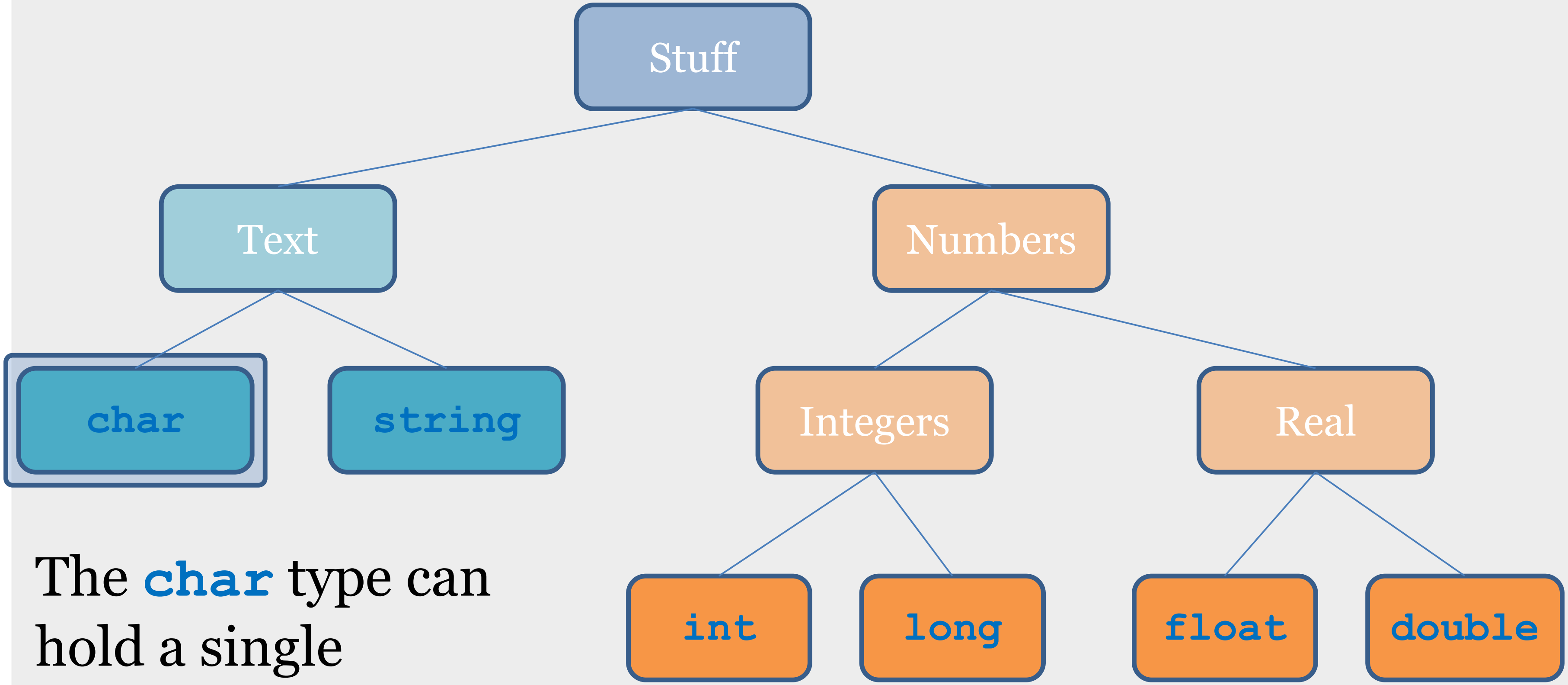
- If you leave out the f in the above code the program will fail to compile
- The compiler will not let a program put a value into a variable if it thinks the type might not be able to hold it correctly

Text



C# provides two types for holding text

Individual Characters



The **char** type can hold a single character

Using Characters

- You use a char type if you want to store a single character
- It can be a letter, digit, punctuation character, control character or space
- This character will be held as a single value using the UNICODE standard

Character Codes

- Computers store everything as patterns of bits
- For a computer to store text we have to map these patterns to particular characters
- C# uses the UNICODE standard to perform this mapping

The UNICODE Standard

- UNICODE is a standard for characters
- Each character is stored in a 16 bit value
- This allows for over 64,000 characters
- You may have heard of an 8 bit code called ASCII
- The ASCII character set is mapped onto the first 128 values of UNICODE

Character literal values

```
char commandKey;
commandKey = 'A';
```

- A character literal value is written in the program enclosed in single quotes
- This is how the compiler can tell which is the character to be used
- Upper and lower case characters are different

Control Codes

- Some characters are not printed on the screen, but instead have a control behaviour
  - Carriage return
  - Take a new line
  - Sound an alert
  - Tab
- C# uses *escape sequences* to allow a program to use these codes

Escape Sequence

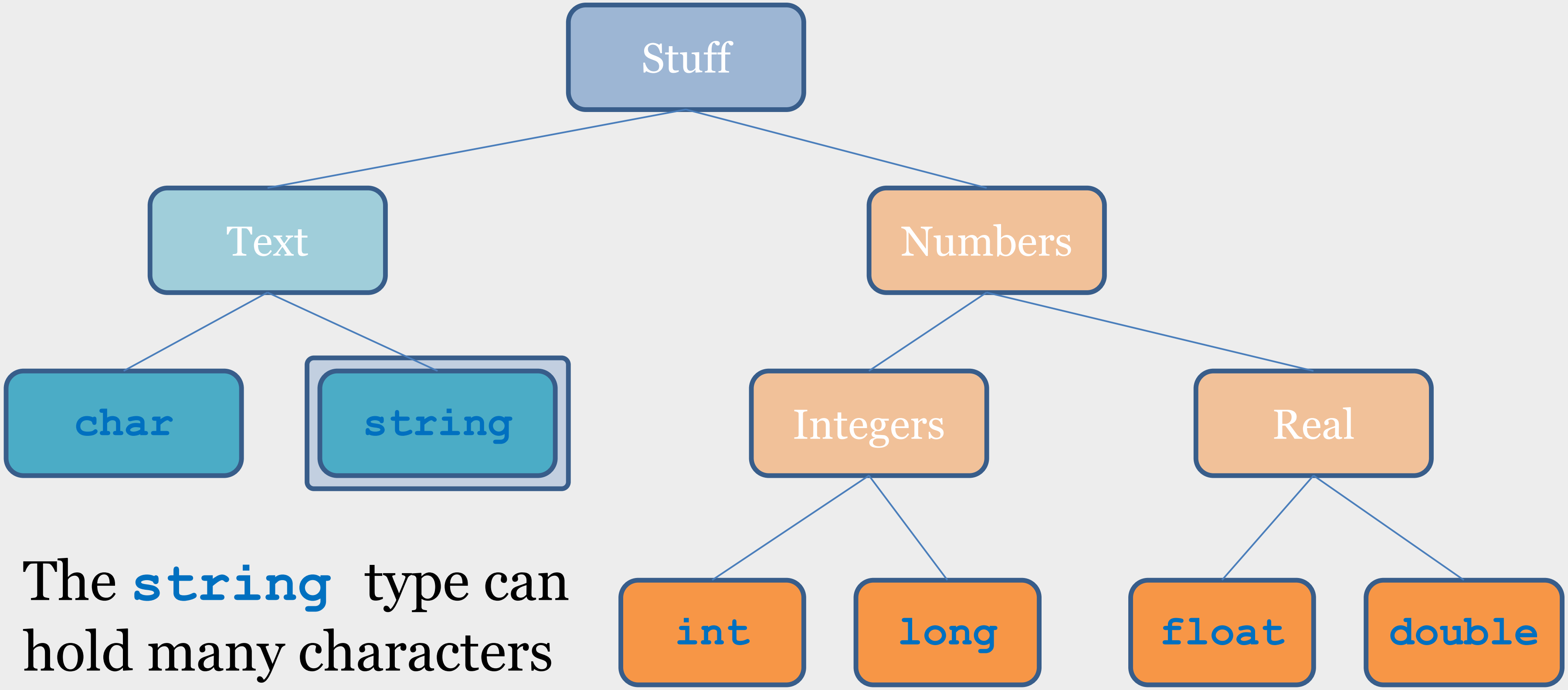
```
char newLine;
newLine = '\n';
```

- The escape sequence is the backslash (\) character followed by a letter that identifies the required control character
- Letter n means "newline"

Escape Sequence Values

Character	Escape Sequence name
\'	Single quote
\"	Double quote
\\	Backslash
\0	Null
\a	Alert
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical quote

Strings of Characters



The **string** type can hold many characters

Using Strings

- You can use a string everywhere you need to store some text:
  - Names
  - Addresses
  - The book “War and Peace”
- Strings can get very long indeed
- They also provide a bunch of useful text behaviours

Storing Strings

- There is only one `string` type
- String storage is managed automatically by the C# runtime system
- A storage area of the right size is created for each string that is stored
- You don’t need to worry about reserving memory for strings or releasing it when you have finished

String literal values

```
string name;
name = "Rob Miles";
```

- A string literal is enclosed in double quotes
- You can put escape sequences in the string as well – they must be preceded by the \ character as used in chars

Verbatim String literal values

```
string backslash;
backslash = @"A backslash : \";
```

- If you don’t want to use escape sequences in your string literal you can put an @ in front of it
- This means the string is used verbatim

Multi-Line Verbatim Strings

```
string address;
address = @"University of Hull
Cottingham Road
Hull";
```

- A verbatim string can spread over several lines
- The line breaks are preserved

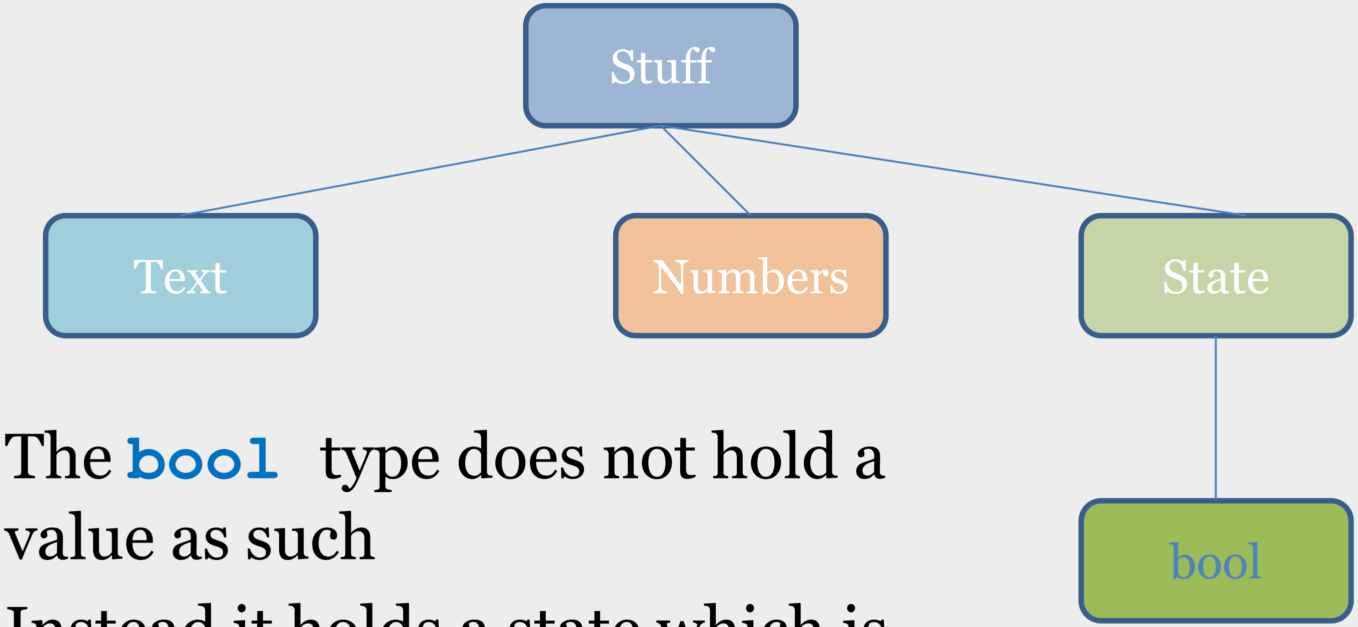
Taking newlines in strings

```
Console.WriteLine("Hello\nWorld");
```

```
Hello
World
```

- The newline character in a string will cause a new line to be taken at that point

Storing State



The **`bool`** type does not hold a value as such

Instead it holds a state which is either true or false

Storing State

- Some things that are to be stored are not values as such, but instead are *states*
  - "is a member of the club"
  - "input is valid"
  - "network OK"
- C# provides a `bool` type which is used to hold the states `true` or `false`

The bool type

- The `bool` type can only hold two possible values
  - true or false
- These could be held by a single bit in the computer memory
  - This is not usually how it is done however, as such a value would be hard to address

Bool literal values

```
bool ageIsValid;
ageIsValid = true;
```

- Variables of the `bool` type can be set to the values `true` or `false` and nothing else
- They can be used directly in conditions, as we shall see later

Choosing a Variable Type

- Price of an ice cream
- The possible types are:

  `sbyte` – hold an integer from -127 to +128

  `byte` – hold an integer from 0 to 255

  `short` – hold an integer from + or – 32,000

  `int` – hold an integer + or – 2,000,000,000

  `float` – hold a real with 7 digit precision

  `double` – hold a real with 15 digit precision
- Which would you choose?

Ice Cream Price

- I'd use `int` or `short`
- Although it will be priced in pounds and pence (e.g. 1.20) I don't want to use a real number since these are not what they are for
- An ice cream could cost more than 2.55 and so it has to be `short` or `int`

Choosing Another Variable

- Speed of a car in MPH
- The possible types are:
  - `sbyte` – hold an integer from -127 to +128
  - `byte` – hold an integer from 0 to 255
  - `short` – hold an integer from + or – 32,000
  - `int` – hold an integer + or – 2,000,000,000
  - `float` – hold a real with 7 digit precision
  - `double` – hold a real with 15 digit precision
- Which would you choose?

Speed of a Car

- This depends on the accuracy of the sensor and the way the result is to be displayed
  - `sbyte` is no good because the range is too small
  - `byte` is no good because you can't go backwards
- You can make a good case for just about any of the others

Identifiers

- Each item we create in a program must have an identifier (or name)
- We decide what the identifier is:

The identifier of an item must reflect what the item is to be used for.

C# Identifier Rules

- Used in the program use to identify something that you have created
  - Can only contain letters, digits and the underscore (_) character
  - Must start with a letter or underscore (_)

    `Width HeightString 99ImIllegal so$am$I`
- The case is significant:
  - `Fred` is a different identifier from `fred`

Summary

- Programs work by operating on data
- The data is stored in *variables* which are of a particular data *type*
- The type of a variable determines what you can put into it
- The programmer must select appropriate data types and create appropriate *identifiers* for variables in a program