BC# Reference Manual

BC# is a primitive (though definitely nontrivial!) subset of the real C# language. The following is a description of the language. Like many such brief descriptions, it may be ambiguous. In such cases, use your best judgment and document your decisions.

1 Lexical Issues

- **Identifiers**: Identifiers are class names or variable names. Case is significant when comparing identifiers.
  
  Variable names (local variables and method parameters) start with a lowercase letter, and can consist of any number of following letters, underscores, or digits.
  
  A “basic” class name starts with a capital letter, and can consist of any number of following letters, underscores, digits. A BC# class name is a sequence of one or more basic class names separated by periods (e.g., Foo or System.Object), with no whitespace.

- **Comments**: A comment is either a line comment, beginning with // and going to the end of the line, or a delimited comment, starting with /* and ending with the next */. Comments do not nest: The character sequences /* and // have no special meaning within a comment.

- **Whitespace**: White space and indentation (and comments) do not affect the meaning of the program, except to separate tokens. Whitespace characters include space, newline, carriage return, tab, form-feed, and vertical tab.

  Information on other tokens and keywords is interspersed throughout the rest of this document.

2 Programs

A Program is a collection of class definitions. Exactly one of the classes must be named Start, and this class must contain a member static void Main whose argument is an array of strings.\(^1\)

The names of all classes must be distinct, and the names of all fields and methods within a single class must be distinct. In particular, BC# does not support overloading.

Classes are all defined mutually-recursively (any class can refer to any other class in the file without needing forward declarations).

\(^1\)It is probably not a good idea to enforce this constraint in the parser.
Data types

The types of BC# are:

1. `int`. Integer constants can be written either in base ten digits, or in hexadecimal digits (preceded by `0x` or `0X`).

2. `char`. A single ASCII character. Written as a single character in single quotes, e.g., `'a'` or `'
'`.

3. `bool`. The two constants of this type are `false` and `true`. (These names are also reserved words.)

4. `string`. String constants are delimited by double quotes, and must be on a single line. String constants can contain escape codes (e.g., `\n` to represent the newline character, or `\"`, which is a double quote character rather than the end-of-string marker).
   Alternatively, a string constant can be written `@"..."` to mean a literal string with no expansion of escape codes. Such strings can even span multiple lines (in which case the string includes newline characters), e.g.,

   ```
   string s = "Line1
   Line2\n\n\n";
   ```

   defines a string containing a single newline characters, five occurrences of the letter n, and three backslashes, while

   ```
   string s = "Line1\nLine2\n\n\n";
   ```

   defines a string with four newline characters, two actual n's and no backslashes.

5. The class name `Object`.

6. Array types (`t[]` for any BC# type `t`).

7. User-defined class names, all of which (eventually) inherit from `Object`.

The above-named built-in types (e.g., `int` and `Object`) are also reserved words.

The only implicit conversions are

- From `char` to `int`.

- From any class to a class that it inherits from.

Note that there is no conversion from `int` to `bool` or vice-versa, and we do not convert array types.

---

2 This is zero-x, not the word ox.
3 real C# actually uses Unicode.
3 Expressions

Every expression has a type, which describes the sort of value it returns.

- Constants and identifiers.
- Unary (-, preincrement ++, predecrement --) and binary (+, -, *, /) arithmetic operators.
- Arithmetic, string, and character comparisons (<, <=, >=, >) returning a boolean.
- Unary (!) and short-circuiting boolean (&&, ||) logical operators.
- Binary equality (==) and inequality (!=), for any two values of the same type. This implements integer equality, string equality, or address-equality, depending on the type of the arguments.
- Conditional expression (e1 ? e2 : e3). The values of e2 and e3 must be convertible to a common type.
- Assignment (lvalue = rvalue). The value stored is also the value of the whole assignment operation. Assignment is right-associative.
- Binary string concatenation (s1 + s2), returning a brand-new string without side
- Array indexing (e[n]), or extracting a character from a string (e[n]) to obtain a character.
- Values can be created by (new C(...)) for any class C. The arguments are passed to the constructor.
- For any non-array type t, an array can be created by new t[e], where e is the run-time expression providing the length. All elements will be default initialized (meaning that that if you want an array of objects, the constructor had better take zero arguments). If you want an array of arrays, however, the syntax is new t[e][][].
- The constant null, which can be treated as having any user-defined class type or type Object.
- The constant this can be used in methods to refer to the enclosing object.
- e.l extracts the value of the field l from the object resulting from evaluating e. Alternatively, C.l extracts the value of the static field l from the class C.
- e.m(...) is invocation of the virtual method m in the object resulting from evaluating e. The ..., must be a collection of comma-separated arguments in parentheses. Alternatively, C.m(...) invokes the static method m in the class C.
• e.toString() returns the value of the expression e expressed as a string, if e is an integer
(in which case you get a string representation of the integer), character (in which case you
get a 1-character string), or boolean.\(^4\)

• e.Length returns the number of characters in the string e.\(^5\)

• Console.In.readLine() gets a line of input from the standard input and returns it as a
string.\(^6\)

4 L-values

An l-value is something that can appear on the left-hand-side of an assignment statement, or as
the operand of a a preincrement/predecrement operator. Permissible l-values are:

• Identifiers

• Array\(^7\) accesses e1[e2]

• static fields X.f, or object fields e.f.

5 Statements

Statements are executed for their side effects, and do not return any values.

• The empty statement ; does nothing.

• If e is an expression, then e ; is a statement that evaluates e (for its side-effects) and throws
away the result.

• A block statement is a sequence of zero or more statements enclosed in { and }.

• if (e) stmt1 and if (e) stmt1 else stmt2 are statements if e is a boolean expression,
and stmt1 and stmt2 are statements. The parentheses are required.

• while (e) stmt is a statement if if e is a boolean expression, and stmt is a statement. The
parentheses are required.

• for (e1; e2; e3) stmt is the usual loop structure.\(^8\)

• return; and return e; are statements; the former is used in void functions, and the latter
is used if we need to return the value of the expression e.

\(^4\)Don’t worry about this in the parser.
\(^5\)ditto
\(^6\)ditto
\(^7\)Not String accesses...string objects are immutable.
\(^8\)For simplicity, e1 must be an expression — probably an assignment — and not a variable declaration.
• A variable declaration \( T \ x = e \); defines a variable \( x \) of type \( T \) and initial value \( e \). The scope of the variable is the rest of the enclosing statement.

• \texttt{Console.write(e)}; is a statement for any integer, character, boolean, or string expression \( e \). It displays the value to the standard output stream.

6 Classes

A class declaration has the form

```c
class \texttt{identifier} : \texttt{identifier} {  
optional-fields  
constructor  
optional-methods  
};
```

where the first identifier is a class name, and then after the colon is the user-defined class being inherited from. (One cannot inherit from the other built-in types, e.g., \texttt{int} or \texttt{String}.) The colon and the superclass can be omitted, in which case the class implicitly inherits from \texttt{Object}. The semicolon at the end is also optional.

Each field may be static or not. static fields have a type and an initial value; non-static fields just have a type. All BC# fields are (implicitly) public.

A method may be static, virtual, or override (exactly one of the three — override means the same as virtual except that a method of the same name with the same parameter and return types must have been inherited). It must have a return type or be declared as returning void, have a name, and have a parenthesized, comma-separated list of formal parameters). The body of a method is a block statement ending with a \texttt{return}. Within a method, one must always use this to refer to methods and fields in the surrounding object.

```c
class Point : Object {  
static int points_created = 0;  
int x;  
int y;  
Point(int x0, int y0) {  
this.x = x0; // Recall that the reference to this is required  
this.y = y0; // Ditto  
++Point.points_created;  
return;  
}  
virtual void move(int dx, int dy) {  
this.x = this.x + dx;
```
this.y = this.y + dy;
return;
}
};

after which one can say

// ...
Point myPoint = new Point(7,3);
myPoint.move(3,4);
myPoint.move('a',-0xFF);
// ...

Constructors have an optional call to their base-class constructor. The syntax is : base(...)
where ... is a comma-separated set of arguments. E.g.,

class Super {
  int f;

  Super(int n) {
    this.f = n;
    return;
  }
};

class Derived : Super {
  int f2;
  Derived(int m1, int m2) : base(m1+m2) {
    // The base-class constructor initializes this.f to m1+m2
    this.f2 = m1 * m2;
    return;
  }
};

If there is no explicit base call, (as in Super’s and Point’s constructors), it is equivalent to
writing : base().