**SORT BY NAME sample (TypeScriptApp solution)**

|  |
| --- |
| 1. Start with **sortByName-00**   **Result:**  function sortByName(a) {  var result = a.slice(0);  result.sort(function (x, y) {  return x.name.localCompare(y.name);  });  return result;  } |
| 1. Closers from Google. Add at the beginning the **sortByName-01** code   **Result:**  /\*\*  \* @param {Array} a  \*/  function sortByName(a) {  var result = a.slice(0);  result.sort(function (x, y) {  return x.name.localCompare(y.name);  });  return result;  } |
| 1. Show the simple typing annotation with TypeScript by specifying the type of **a** variable in **sortByName** function: **any[]**   **Result:**  function sortByName(a: any[]) {  var result = a.slice(0);  result.sort(function (x, y) {  return x.name.localCompare(y.name);  });  return result;  } |
| 1. Show that having the type of variable **a** specified, you can benefit from code completion. Try a. |

|  |
| --- |
| 1. Add an interface that describes an entity to give more sense to the objects that are sorted in the code. Annotate the **a** array with Entity type. Use **sortByName-02.** Note that **localCompare** function name is incorrect which actually would be a bug in a dynamic language like JavaScript.   **Result:**  interface Entity {  name: string;  }  function sortByName(a: Entity[]) {  var result = a.slice(0);  result.sort(function (x, y) {  return x.name.localCompare(y.name);  return x.name.localeCompare(y.name);  });  return result;  } |
| 1. Show the refactoring available in Visual Studio which is very powerful also for TypeScript code. For this add a declaration of a new variable **name**. Refactor the property **name** of the **Entity** type with **entityName**.   **Result:**  interface Entity {  entityName: string;  }  function sortByName(a: Entity[]) {  var result = a.slice(0);  result.sort(function (x, y) {  var name = "abc";  return x.entityName.localeCompare(y.name);  });  return result;  } |
| 1. Add to the entity 2 new properties called **price** and **inStock**. After this make an instance of **Entity** type called **e**. If you don’t specify a value for **inStock** property you will get an error which says that **inStock** should have a value when an **Entity** type is instantiated. To solve the problem you can specify that **inStock** is an *optional property*.   **Result:**  interface Entity {  name: string;  price: number;  inStock?: number;  }  var e: Entity = {  name: "Hammer",  price: 5.95  };  function sortByName(a: Entity[]) {  var result = a.slice(0);  result.sort(function (x, y) {  return x.name.localeCompare(y.name);  });  return result;  } |
| 1. Show that interfaces in TypeScript can have also functions. Add **sortByName-03**. Also go and implement the added functions.   **Result:**  interface Entity {  name: string;  price: number;  getName(): string;  setName(value: string): void;  }  var e: Entity = {  name: "Hammer",  price: 5.95,  getName: function () {  return this.name;  },  setName: function (newName) {  this.name = newName;  }  };  function sortByName(a: Entity[]) {  var result = a.slice(0);  result.sort(function (x, y) {  return x.name.localeCompare(y.name);  });  return result;  } |

|  |
| --- |
| 1. Clean the code to minimal.   **Result:**  interface Entity {  name: string;  price: number;  }  function sortByName(a: Entity[]) {  var result = a.slice(0);  result.sort(function (x, y) {  return x.name.localeCompare(y.name);  });  return result;  } |
| 1. Add at the end of the code **sortByName-04.**   **Result:**  interface Entity {  name: string;  price: number;  }  function sortByName(a: Entity[]) {  var result = a.slice(0);  result.sort(function (x, y) {  return x.name.localeCompare(y.name);  });  return result;  }  var products = [  { name: "name 2", price: 2.0, id: 2 },  { name: "name 1", price: 1.0, id: 1 },  { name: "name 5", price: 5.0, id: 5 },  { name: "name 4", price: 4.0, id: 4 },  { name: "name 3", price: 3.0, id: 3 }  ];  var sorted = sortByName(products); |
| 1. First note that the type of **products** variable was inferred as being an array of objects that have **name**, **price** and **id** properties. Also, just because **products** satisfies at the minimum the definition of **Entity** type, the code will be ok and **sorted** will infer the type **Entity**. This means that if you write sorted[0]. you will see that **id** property will be lost. |

|  |
| --- |
| 1. To solve the problem, **generics** should be used. To do so, change the declaration of the sortByName function by adding the generics. Try now to use **sorted** variable like this sorted[0].   **Result:**  interface Entity {  name: string;  price: number;  }  function sortByName<T>(a: T[]) {  var result = a.slice(0);  result.sort(function (x, y) {  return x.name.localeCompare(y.name);  });  return result;  }  var products = [  { name: "name 2", price: 2.0, id: 2 },  { name: "name 1", price: 1.0, id: 1 },  { name: "name 5", price: 5.0, id: 5 },  { name: "name 4", price: 4.0, id: 4 },  { name: "name 3", price: 3.0, id: 3 }  ];  var sorted = sortByName(products);  sorted[0]. |
| 1. Note that there are some errors with **name** property which seems do not exist. This actually means that **T** can be of any type and we are really interested in constraining **T** to the **Entity** type. To make it happen we can use **extends**:   **Result:**  interface Entity {  name: string;  price: number;  }  function sortByName<T extends Entity>(a: T[]) {  var result = a.slice(0);  result.sort(function (x, y) {  return x.name.localeCompare(y.name);  });  return result;  }  var products = [  { name: "name 2", price: 2.0, id: 2 },  { name: "name 1", price: 1.0, id: 1 },  { name: "name 5", price: 5.0, id: 5 },  { name: "name 4", price: 4.0, id: 4 },  { name: "name 3", price: 3.0, id: 3 }  ];  var sorted = sortByName(products);  sorted[0]. |
| 1. **MOVE TO PLAYGROUND.** Show that even generics are tools for being productive and are not there to add cost in performance to actual JavaScript. |
| 1. Add at the end of the code **sortByName-05.**   **Result:**  interface Entity {  name: string;  price: number;  }  function sortByName<T extends Entity>(a: T[]) {  var result = a.slice(0);  result.sort(function (x, y) {  return x.name.localeCompare(y.name);  });  return result;  }  var products = [  { name: "name 2", price: 2.0, id: 2 },  { name: "name 1", price: 1.0, id: 1 },  { name: "name 5", price: 5.0, id: 5 },  { name: "name 4", price: 4.0, id: 4 },  { name: "name 3", price: 3.0, id: 3 }  ];  var sorted = sortByName(products);  document.body.innerText = JSON.stringify(sorted, null, 4); |

|  |
| --- |
| 1. Now, let’s generalize the **sortByName** method to be able to sort by any key or property. Remove the current method and add the code **sortByName-06:**   **Result:**  interface Entity {  name: string;  price: number;  }  function sortBy<T>(a: T[], keyOf: (item: T) => any): T[] {  var result = a.slice(0);  result.sort(function (x, y) {  var kx = keyOf(x);  var ky = keyOf(y);  return kx > ky ? 1 : kx < ky ? -1 : 0;  });  return result;  }  var products = [  { name: "name 2", price: 2.0, id: 2 },  { name: "name 1", price: 1.0, id: 1 },  { name: "name 5", price: 5.0, id: 5 },  { name: "name 4", price: 4.0, id: 4 },  { name: "name 3", price: 3.0, id: 3 }  ];  var sorted = sortBy(products, x => x.price);  document.body.innerText = JSON.stringify(sorted, null, 4); |

**POO sample (TypeScriptApp solution)**

|  |
| --- |
| 1. Let’s look at how the classes can be defined. Use **POO-01:**   **Result:**  class Point {  x: number;  y: number;  }  var p = new Point();  p.x = 10;  p.y = 20; |
| 1. If I’d like to pass the X and Y to the constructor for the Point type, but I have to add a constructor to it. Use **POO-02:**   **Result:**  class Point {  x: number;  y: number;  constructor(x: number, y: number) {  this.x = x;  this.y = y;  }  }  var p = new Point(10, 20);  p.x = 10;  p.y = 20; |
| 1. Now let’s see how functions, static types are generated. Use **POO-03:**   **Result:**  class Point {  x: number;  y: number;  constructor(x: number, y: number) {  this.x = x;  this.y = y;  }  dist() {  return Math.sqrt(this.x \* this.x + this.y \* this.y);  }  static origin = new Point(0, 0);  }  var p = new Point(10, 20); |
| 1. Also show that the property declaration and initialization of them can happen directly in constructor and they also can be optional parameters to constructor. Use **POO-04:**   **Result:**  class Point {  constructor(public x: number = 0, public y: number = 0) {  }  dist() {  return Math.sqrt(this.x \* this.x + this.y \* this.y);  }  static origin = new Point(0, 0);  }  var p = new Point(10, 20); |
| 1. Let’s go deeper and show the inheritance. Use **POO-05** and add the following code at the end:   **Result:**  class Point {  constructor(public x: number = 0, public y: number = 0) {  }  dist() {  return Math.sqrt(this.x \* this.x + this.y \* this.y);  }  static origin = new Point(0, 0);  }  class Point3D extends Point {  constructor(x: number, y: number, public z: number) {  super(x, y);  }  }  var p = new Point(10, 20); |

|  |
| --- |
| 1. Let’s introduce modules. Use **POO-06** to show this:   **Result:**  module Utils {  export class Point {  constructor(public x: number = 0, public y: number = 0) {  }  dist() {  return Math.sqrt(this.x \* this.x + this.y \* this.y);  }  static origin = new Point(0, 0);  }  class Point3D extends Point {  constructor(x: number, y: number, public z: number) {  super(x, y);  }  }  }  var p = new Utils.Point(10, 20); |

**TRACKER sample (TypeScriptApp solution)**

|  |
| --- |
| 1. Let’s take a look at a very simple but a very common issue in JavaScript code. Use **Tracker-01**:   **Result:**  class Tracker {  count = 0;  start() {  window.onmousemove = function (e) {  this.count++;  console.log(this.count);  }  }  }  var t = new Tracker();  t.start(); |
| 1. Run it in browser and you will find that the common issue is related to **this** pointer. To solve the problem use lamba expressions which will come with EcmaScript 6 specification:   **Result:**  class Tracker {  count = 0;  start() {  window.onmousemove = e => {  this.count++;  console.log(this.count);  }  }  }  var t = new Tracker();  t.start(); |

**NODE JS sample (NodeJS Backend solution)**

|  |
| --- |
| 1. Add to the solution the NuGet package called **node.TypeScript.DefinitelyTyped**. After adding it, remove from the package directory any other version than final (like **node-0.8.8.d.ts**). |
| 1. Open the **NodeJS console** to have it prepared to compile the project. Also make sure that in project properties you will compile it as **CommonJS** module. |
| 1. Open **server.ts** file and reference the imported package. Use **nodeJS-01** for this:   **Result:**  /// <reference path="scripts/typings/node/node.d.ts" /> |
| 1. Import from the package the **http** module. Use **nodeJS-02** for this:   **Result:**  /// <reference path="scripts/typings/node/node.d.ts" />  import http = require("http"); |
| 1. Your app will export for external reuse a function called **nodeServer**. The definition of this function is in **nodeJS-03**:   **Result:**  /// <reference path="scripts/typings/node/node.d.ts" />  import http = require("http");  export function nodeServer(port: number, message: string) {  } |
| 1. The **nodeServer** function will create a server using **NodeJS** and to do so it will use the **createServer** function from **http** module. Use **nodeJS-04** for this:   **Result:**  /// <reference path="scripts/typings/node/node.d.ts" />  import http = require("http");  export function nodeServer(port: number, message: string) {  http.createServer((req, res) => {  }).listen(port);  } |

|  |
| --- |
| 1. Add some logging functionality for the server by adding some info to the response. Use **nodeJS-05** for this:   **Result:**  /// <reference path="scripts/typings/node/node.d.ts" />  import http = require("http");  export function nodeServer(port: number, message: string) {  http.createServer((req, res) => {  res.writeHead(200, { "Content-Type": "text/html" });  res.write("<h1>" + message + "</h1>");  res.end();  }).listen(port);  } |
| 1. On the client side, in **client.ts**, import the current definition of the server. Use nodeJS-06 for this:   **Result:**  import server = require("./server"); |
| 1. Call the nodeServer. Use nodeJS-07 for this:   **Result:**  import server = require("./server");  server.nodeServer(1337, "Hello ITCamp!");  console.log("Listening on 1337..."); |
| 1. The code is done. Compile the code and go to **NodeJS** console to execute: **node client** |
| 1. Open the browser at <http://localhost:1337> and see the result |