# Exercises: Generics

## Array Swap

Write a generic function that takes in **4 arguments**:

* **a** – array of the generic type
* **aIndex** – number
* **b** – array of the generic type
* **bIndex** – number

The function then **swaps** the element at index **aIndex** in **a** with the element at index **bIndex** in **b**

The function doesn’t need to return anything; it directly mutates the arrays.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| let a = ['test', '123'];  let b = ['a', 'b', 'c'];  swap<string>(a, 0, b, 2);  console.log(a)  console.log(b) | ['c', '123']  ['a', 'b', 'test'] |
| let a = [20, 30 , 40];  let b = [1, 2, 3, 4, 5];  swap<number>(a, 0, b, 2);  console.log(a)  console.log(b) | [3, 30, 40]  [1, 2, 20, 4, 5] |

## CountableSet<T>

You are given the generic interface **CountableSet<T>**, which keeps track of unique items and their count (The set shouldn’t keep multiple copies, instead just manage a counter for each item).

|  |
| --- |
| **CountableSet<T>** |
| interface CountableSet<T> {      add(item:T): void;      remove(item: T): void;      contains(item: T): boolean;      getNumberOfCopies(item: T): number;  } |

Your task is to create a class **CountedSet<T>** that implements the **CountableSet<T>** interface, using the following logic:

* **add**(item:T) – if item **T is not in the set** – **add it** and **set count to 1**, otherwise just increment its **counter**.
* **remove**(item: T) – if item **T is not in the set** or is but **has count 0** – **does nothing**. If the **item is in the set** **and has count > 0**, **decrement** the counter.
* **contains**(item: T) – if **item exists in the set and has a count > 0** – returns **true**, otherwise returns **false**.
* **getNumberOfCopies**(item: T) – **returns** **the number of copies of the item** in the set, **if the item doesn’t exist** in the set **returns 0**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| let countedSet = new CountedSet<string>();  countedSet.add('test');  countedSet.add('test');  console.log(countedSet.contains('test'));  console.log(countedSet.getNumberOfCopies('test'));  countedSet.remove('test')  countedSet.remove('test')  countedSet.remove('test')  console.log(countedSet.getNumberOfCopies('test'));  console.log(countedSet.contains('test')); | true  2  0  false |
| let codesCounterSet = new CountedSet<200 | 301 | 404 | 500>();  codesCounterSet.add(404);  codesCounterSet.add(200);  console.log(codesCounterSet.contains(404));  console.log(codesCounterSet.getNumberOfCopies(200));  codesCounterSet.add(205);   //TS Error  codesCounterSet.getNumberOfCopies(350);    //TS Error | true  1  //TS Error: argument '205'  not assignable to type  //TS Error: argument '350' not assignable to type |

## Mechanic

You are given a generic class **Mechanic**

|  |
| --- |
| **Mechanic** |
| class Mechanic<T> {      technicalInspection(car: T): boolean { return true; }  } |

Add a constraint to its generic parameter so it allows **technicalInspection** to be called only with objects meeting the following structure:

* **engine** - object
  + **horsepower: number**
* **tires -** object
  + **model: string**
  + **airPressure: number**
* **body –** object
  + **material: string**

### Examples

You can use the bellow code to check if you implemented the Type constraint correctly.

|  |
| --- |
| **Sample code** |
| let mechanic = new Mechanic();  let someCar = { engine: { horsepower: 350, type: 'diesel' }, tires: { model: 'BRIT', airPressure: 33 }, body: { material: 'aluminum' } };  let notACar = { vroom: false };  let maybeCar = { tires: { model: 'BRIT' }, body: { material: 'aluminum' } };  let maybeCar2 = { engine: { horsepower: 220 }, tires: { model: 'BRIT', wear: 'High', airPressure: 33 }, body: { material: 'aluminum' } };  let maybeCar3 = { engine: { horsepower: 250 }, tires: { model: 'Nie' }  };  let maybeCar4 = { engine: { horsepower: 220, type: 'electric' }, tires: { model: 'BRIT' }, body: { material: 'steel', weight: 2670 } };  let maybeCar5 = { engine: { horsepower: '220', type: 'electric' }, tires: { model: 'BRIT', airPressure: 28 }, body: { material: 'steel', weight: 2670 } };  mechanic.technicalInspection(someCar); //ok  mechanic.technicalInspection(maybeCar2); //ok  mechanic.technicalInspection(maybeCar4); //TS Error  mechanic.technicalInspection(notACar); //TS Error  mechanic.technicalInspection(maybeCar); //TS Error  mechanic.technicalInspection(maybeCar3); //TS Error  mechanic.technicalInspection(maybeCar5); //TS Error |

## Conditional Number

Write a generic function that accepts a single parameter, the type of the accepted parameter depends on the generic argument of the function:

* If the generic argument is number, the function should accept a number parameter and print it formatted to the 2nd decimal place
* If the generic argument is anything else, the function should accept a string argument and print it as is.

**Hint**: You should use conditional types.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| conditionalNumber<number>(20.3555);  conditionalNumber<string>('wow');  conditionalNumber<boolean>('a string'); | 20.36  wow  a string |
| conditionalNumber<boolean>(30);  conditionalNumber<number>('test'); | //TS error: type 'number' is not assignable to parameter of type 'string'  //TS error: type 'string' is not assignable to parameter of type 'number' |

**Extra Task**: As an extra task, you can try expanding the constraint on the generic parameter, so that:

* If the generic argument is number – accepts only a number parameter
* If the generic argument is string – accepts only a string parameter
* On any other generic argument type – rejects all parameters

## Caching Logger

You are given 2 enums and an interface:

* enum **LoggingLevel** – the log severity
* enum **LoggingFormat** – the possible log formats for the log message
* Interface **CachingLogger** – where the generic parameters are of type **LoggingLevel** and **LoggingFormat**

|  |
| --- |
| **LoggingLevel** |
| enum LoggingLevel {      Info = "Info",      Error = "Error",      Warning = "Warning",      Debug = "Debug",  } |

|  |
| --- |
| **LoggingFormat** |
| enum LoggingFormat {      Standard = "[%level][%date] %text",      Minimal = "\*%level\* %text"  } |

|  |
| --- |
| **CachingLogger** |
| interface CachingLogger<T extends LoggingLevel, V extends LoggingFormat> {      cachedLogs: Map<T, string[]>      log(logLevel: T, message: string): void;      getFormat(): V  } |

Your task is to create a class that implements the **CachingLogger** interface.

* The class should take in a **LoggingFormat** value in its constructor – the format for the log messages
* Using the log method should use the passed in **LoggingLevel** parameter and the internal logging format to **format the message** and **print it on the console**
  + The date in the format uses **toISOString()**
* A **copy of each formatted message** should also be **saved in the cachedLogs** **Map** under the correct LoggingLevel.

### Examples

|  |
| --- |
| **Input** |
| let logger = new Logger<LoggingLevel, LoggingFormat>(LoggingFormat.Standard);  logger.log(LoggingLevel.Info, "This is an info message.");  logger.log(LoggingLevel.Info, "Another message.");  logger.log(LoggingLevel.Error, "Something went wrong.");  logger.log(LoggingLevel.Warning, "Be careful with the type assertions.");  logger.log(LoggingLevel.Debug, "Running the debugger.");  console.log('-----------')  console.log([...logger.cachedLogs.entries()].map(x => x[1].join('\n')).join('\n')) |
| **Output** |
| [Info][2024-01-01T22:00:00.000Z] This is an info message.  [Info][2024-01-01T22:00:00.000Z] Another message.  [Error][2024-01-01T22:00:00.000Z] Something went wrong.  [Warning][2024-01-01T22:00:00.000Z] Be careful with the type assertions.  [Debug][2024-01-01T22:00:00.000Z] Running the debugger.  -----------  [Info][2024-01-01T22:00:00.000Z] This is an info message.  [Info][2024-01-01T22:00:00.000Z] Another message.  [Error][2024-01-01T22:00:00.000Z] Something went wrong.  [Warning][2024-01-01T22:00:00.000Z] Be careful with the type assertions.  [Debug][2024-01-01T22:00:00.000Z] Running the debugger. |

|  |
| --- |
| **Input** |
| let logger = new Logger<LoggingLevel, LoggingFormat>(LoggingFormat.Minimal);  logger.log(LoggingLevel.Info, "Just a simple message.");  logger.log(LoggingLevel.Error, "A Problem happened.");  console.log('-----------')  console.log(logger.getFormat());  console.log([...logger.cachedLogs.entries()].map(x => x[1].join('\n')).join('\n')) |
| **Output** |
| \*Info\* Just a simple message.  \*Error\* A Problem happened.  -----------  \*%level\* %text  \*Info\* Just a simple message.  \*Error\* A Problem happened. |

|  |
| --- |
| **Input** |
| let logger = new Logger<LoggingLevel, LoggingFormat>("%text"); //TS Error  let wronglogger = new Logger<string, LoggingLevel>();          //TS Error  logger.log("%s", "Running the debugger.");                     //TS Error  logger.log({format: "Test %s"}, "Running the debugger.");      //TS Error |

## Choose Type

Your task is to implement a generic mapped type **Choose** - extracts a subtype from a provided type. The type should have 2 generic parameters:

* T – the **type to get the keys from**
* K – a type representing the **keys to extract from T**

**Hint**: You can create mapped type by iterating one set of keys but getting the types for the keys from somewhere else. Type constraints can help you ensure these keys exist in the target type.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| type test = {  name: string,  age: number,  test:() => string;  }  type extracted = Choose<test, 'name' | 'age'> | //type extracted = {  // name: string,  // age: number  //} |
| type anotherType = {      time: Date,      duration: number,      test: () => string,      val: 200 | 300,      user: {          name: string,          age: number      }  }  type nestedUserAndTime = 'user' | 'time'  type extracted2 = Choose<anotherType, nestedUserAndTime> | //type extracted2 = { // user: { // name: string; // age: number; // }; // time: Date; //} |
| type test3 = { name: string, age: number }  type error = Choose<test3, 'wow'> | //TS Error 'wow' is not a key in test3 |

## Holiday Manager

You are given 3 enums and 2 interfaces **Holiday** and **VacationManager**:

|  |  |  |
| --- | --- | --- |
| **TravelVacation** | **MountainVacation** | **BeachVacation** |
| enum TravelVacation {      Abroad = 'Abroad',      InCountry = 'InCountry'  } | enum MountainVacation {      Ski = 'Ski',      Hiking = 'Hiking'  } | enum BeachVacation {      Pool = 'Pool',      Sea = 'Sea',      ScubaDiving = 'ScubaDiving'  } |

|  |  |
| --- | --- |
| **Holiday** | **VacationManager** |
| interface Holiday {      set start(val: Date);      set end(val: Date);      getInfo(): string;  } | interface VacationManager<T, V> {      reserveVacation(holiday: T, vacationType: V): void;      listReservations(): string;  } |

Write a class **PlannedHoliday** that implements the Holiday interface, it should have the following functionality:

* **set** **start**(val:Date) – sets the start date of the holiday – if the start date is after the end date throw and error
* **set** **end**(val:Date) – sets the end date of the holiday – if the end date is before the start date throw and error
* **getInfo()** – returns info about the holiday in the following format `**Holiday: {startDate in format d/m/yyyy} - {endDate in format d/m/yyyy }`**

Write a generic class **HolidayManager** that implements **VacationManager** and has 2 generic types:

* The first type has to be compatible with the **Holiday** interface
* The second type has to be of type either **TravelVacation**, **MountainVacation** or **BeachVacation**

The Holiday manager has the following functionality:

* Should store connections between a holiday and a vacation type (you can choose an appropriate data structure to use)
* **reserveVacation(holiday, vacationType**) – connect the passed in holiday and the passed in vacationType and store them in the **HolidayManager**
* **listReservations()** – returns info about all stored holidays and their connected vacations, each in the format `**{holiday.getInfo()} => {vacationType}`**

### Examples

|  |
| --- |
| **Input** |
| let holiday = new PlannedHoliday(new Date(2024, 1, 1), new Date(2024, 1, 4));  let holiday2 = new PlannedHoliday(new Date(2025, 3, 14), new Date(2025, 3, 17));  let holidayManager = new HolidayManager<Holiday, TravelVacation>();  holidayManager.reserveVacation(holiday, TravelVacation.Abroad);  holidayManager.reserveVacation(holiday2, TravelVacation.InCountry);  console.log(holidayManager.listReservations()) |
| **Output** |
| Holiday: 1/2/2024 - 4/2/2024 => Abroad  Holiday: 14/4/2025 - 17/4/2025 => InCountry |

|  |
| --- |
| **Input** |
| let holiday = new PlannedHoliday(new Date(2022, 10, 11), new Date(2022, 10, 18));  let holiday2 = new PlannedHoliday(new Date(2024, 5, 18), new Date(2024, 5, 22));  let holidayManager = new HolidayManager<Holiday, BeachVacation>();  holidayManager.reserveVacation(holiday, BeachVacation.ScubaDiving);  holidayManager.reserveVacation(holiday2, BeachVacation.Sea);  console.log(holidayManager.listReservations()) |
| **Output** |
| Holiday: 11/11/2022 - 18/11/2022 => ScubaDiving  Holiday: 18/6/2024 - 22/6/2024 => Sea |

|  |
| --- |
| **Input** |
| let holiday3 = new PlannedHoliday(new Date(2021, 3, 14), new Date(2020, 3, 17));  let holiday4 = new PlannedHoliday(new Date(2024, 2, 1), new Date(2024, 1, 4)); |
| **Output** |
| //Runtime error: End date cannot be before start date  //Runtime error: End date cannot be before start date |

|  |
| --- |
| **Input** |
| let holiday = new PlannedHoliday(new Date(2024, 1, 1), new Date(2024, 1, 4));  let holiday2 = new PlannedHoliday(new Date(2025, 3, 14), new Date(2024, 3, 17));  let holidayManager = new HolidayManager<Holiday, MountainVacation>();  holidayManager.reserveVacation(holiday, BeachVacation.ScubaDiving);  holidayManager.reserveVacation(holiday2, TravelVacation.InCountry);  console.log(holidayManager.listReservations()) |
| **Output** |
| //TS errors: BeachVacation.ScubaDiving not assignable to MountainVacation  //TS errors: TravelVacation.InCountry not assignable to MountainVacation |

## Function Extraction

Your task is to implement a generic mapped type **AllFunctions** that extracts a subtype with only the function properties from a provided type. The type should have 1 generic parameter:

* T – the **type to get the function properties from**

**Hint**: You may need to compose and use multiple generic mapped types

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| type test = {  name: string,  age: number,  test:() => string;  }  type extracted = AllFunctions<test> | //type extracted = { //test: () => string; //} |
| type Employee = {      name: string,      salary: number,      work: () => void,      takeBreak: () => string  };  type extracted2 = AllFunctions<Employee>; | //type extracted2 = { //work: () => void; //takeBreak: () => string; //} |
| type Nope = {      name: string  };  type extracted3 = AllFunctions<Nope>; | //type extracted3 = {} |