# TypeScript – Retake Exam – 22 June 2025

## Flying Machine

Your task is **implement** **the** **class** **FlyingMachine**, which represents a flying machine that can move in altitude using either passive means like a lifting gas or active means like fuel using engines.

### Skeleton code

You are provided with a skeleton that comes with some files.

#### Contracts

In the **/contracts** folder of your skeleton you’ll be provided with some files, that the **FlyingMachine** class relies on or is meant to work with. You **MUST NOT add/remove or change any files** in the /contracts folder - during evaluation the **/contracts** folder in your project will be swapped with the default one from the skeleton, so **any changes you made in the folder WILL be lost**.

* **flyer.ts** – contains the interface that the **FlyingMachine** should implement.
* **liftingGas.ts** – contains the interface for a **LiftingGas**
* **propellant.ts** – contains the interface for a **Propellant**
* **activeLift.ts** – contains the interface for **ActiveLift** components, active lift components rely on a **Propellant** to generate altitude changes
* **passiveLift.ts** – contains the interface for **PassiveLift** components, passive lift components rely on a **LiftingGas** to generate altitude changes
* **heliumBaloon.ts** – an implementation of a **PassiveLift** component, generates altitude changes passively by relying on a **LiftingGas**
* **solidRocketBooster.ts** – an implementation of an **ActiveLift** component, generates altitude changes by burning a **Propellant**

#### Additional Files

You are also provided with some additional files:

* **decorators.ts** - a decorators file that features the empty decorator functions linked to contract classes.
* **flyingMachine.ts** - the empty implementation of the **FlyingMachine** class
* **index.ts** - a file that you can use for testing, see the Examples section bellow

You are **free to add or modify any files** **outside of the /contracts** folder as you see fit.

#### Skeleton explanation

Aside from the interfaces, you are given the following implementations:

**HeliumBaloon** – an implementation of the **PassiveLift** **interface**, a PassiveLift component consists of at least a:

1. **maxHeight** – a public readonly property that represents the maximum altitude that this **PassiveLift** component can reach.
2. **getAltitudeChange** – a method that **calculates the** **change in altitude**. A **PassiveLift** component **may or may not** use data from the LiftingGas or the current Flying machine’s altitude in the calculation, in this case the **HeliumBaloon** relies on the LiftingGas’s **liftingPower** and the **current altitude** to calculate the change in altitude.
3. A **PassiveLift** component’s constructor should always receive a **maxHeight** parameter andoptionallyany other parameters the implementation requires.

**SolidRocketBooster** – an implementation of the **ActiveLift** **interface**, an **ActiveLift** component should consists of at least a:

1. **fuelConsumptionRate** – a public readonly property that represents the amount of fuel used in a single **move** method call.
2. **liftPerFuelUnit** – a public readonly property that represents the amount of altitude generated **per unit of fuel consumed** in a **move** method call.
3. **optimalWeight** – an optional public readonly property that represents the optimal weight of the flyer for generating lift
4. **getAltitudeChange** – a method that **calculates the change in altitude**. An **ActiveLift** component calculates the altitude change for burning Propellant, may also optionally use the FlyingMachine’s current weight and the optional optimalWeight property in the calculation.
5. An **ActiveLift** component’s constructor should always receive **fuelConsumptionRate** and **liftPerFuelUnit** parametersand optionally an **optimalWeight** parameter.

### Tasks

You will be scored based on the following 3 tasks: Functionality, Structure and Decoration.

#### Functionality (25 points)

The **FlyingMachine** **class** should have:

* **TotalMetersMoved** – A public static property/accessor on the class itself, that should keep track of the **sum of total meters moved (sum of absolute altitude change) achieved by ALL FlyingMachine instances**, this value should **only be readable** **and not modifiable**
  + **TotalMetersMoved** should have an initial value of 0
  + All altitude numbers and altitude changes in this problem, can be considered to be in meters for simplicity (i.e. there are no different units of measurements or conversions, so don’t worry about that)
  + The calculation should use the **absolute value of altitude change**, for example:
    - moving 10 meters up and 20 meters down = 30 total meters moved
* **Constructor –** The **FlyingMachine**’sconstructorshouldtake4parameters**:**
  + liftDevice – the **ActiveLift/PassiveLift** component of the **FlyingMachine**
  + gas – the **LiftingGas/Propellant** component of the **FlyingMachine**
  + baseWeight: number – the base weight of the flying machine
  + altitude: number – the starting altitude of the flying machine
    - If the provided starting altitude **< 0**, set the starting altitude to **0**
    - If the the Flying machine is of type **"Passive"** and the provided starting altitude **> maxHeight**, set the starting altitudeto **maxHeight**

Implement the following methods and properties from the **Flyer** interface:

1. **Property/Accessor weight** –returns the current weight for the flying machine
   1. this property/accessor should only allow reading its value and should prevent changes to the value
   2. The weight should include both the **FlyingMachine’s** **base weight** (passed in the constructor) and the **weight of the** **Propellant** (if the FlyingMachine is of type **"Active"** and has any propellant):
2. **Property/Accessor altitude** – returns the current altitude of the FlyingMachine
   1. This property/accessor should only allow reading its value and should prevent changes to the value
3. **Method** **move()** – should change the FlyingMachine’s altitude, based on its LiftDevice and Gas and return a message based on the absolute amount of altitude change:
   1. If the absolute altitude change = 0 – return the message **"Flyer stayed in place"**
   2. If the absolute altitude change > 0 – return the message **"Flyer moved <absoluteAltitudeChange> meters <direction>" where:**
      1. **absoluteAltitudeChange** is the absolute value of the change in altitude formattedto **2 decimal places**
      2. **direction** is either **"up"** if the FlyingMachine gained altitude or **"down"** if the FlyingMachine lost altitude
   3. If the FlyingMachine is of type **"Active"**, the altitude change only happens if there is enough fuel:
      1. If there is **not enough fuel**, the altitudeChange should instead **be set to 0**
      2. If there is enough fuel, use the altitudeChange calculated by the ActiveLift’s **getAltitudeChange** method and **after that** reduce the **fuelAmount** by the ActiveLift’s **fuelConsumptionRate**
      3. As long as there is enough fuel, fuel should always be consumed on a move command, regardless if it leads to an actual altitudeChange or not
   4. The FlyingMachine should ensure that its **altitude cannot go lower than 0** **or higher than maxHeight** (in case of a FlyingMachine of type **"Passive"**)
      1. In case the altitude change would put the **FlyingMachine bellow 0 altitude**, instead limit the altitude change so that the FlyingMachine **reaches exactly 0 altitude**
      2. In case the altitude change would put the **FlyingMachine above the maxHeight**, instead limit the altitude change so that the FlyingMachine would **reach exactly maxHeight**
4. **Method checkStatus()** – returns information about the status of the FlyingMachine in the following format:
   1. In case the FlyingMachine is of the **"Active"** type:  
      **"Flyer altitude: <altitude> meters**

**Flyer weight: <weight>**

**Fuel left: <fuelAmount>"**

* 1. In case the FlyingMachine is of the **"Passive"** type:

**"Flyer altitude: <altitude> meters**

**Flyer weight: <weight>**

**Max height: <maxHeight>"**

In both cases **altitude** in this method displays only the **integer part** of the actual value

#### Structure (45 Points)

Your **FlyingMachine** implementation should match the following structure requirements:

**NOTE:** These tasks will only award points if the underlying functionality also exists and works as intended.

1. The **FlyingMachine** should fully implement the **Flyer** **interface**.
2. The **FlyingMachine** class constructor should enforce the following type constraints on these parameters:
   * liftDevice - object compatible with either the **ActiveLift** or **PassiveLift interface**
   * gas – object compatible with either the **LiftingGas** or **Propellant** **interface**
3. The **FlyingMachine** class should be **generic** with exactly 1 generic parameter that accepts the string literal value **"Active"** or **"Passive"**
   * In case a **FlyingMachine** is initialized with **"Active"**, it should only be able to accept a liftDevice of type **ActiveLift** and gas of type **Propellant**
   * In case a **FlyingMachine** is initialized with **"Passive"** it should only be able to accept a liftDevice of type **PassiveLift** and gas of type **LiftingGas**
4. Ensure that all code and implementations written by you, do **NOT** use the **any** type
   * The only allowed usage of **any**, is for the argument types as part of the constructor type in decorators:
     1. **"{ new(...args: any[]): {} }"**
     2. **"constructor(...args: any[])"**

If you use the **any** type anywhere else, points will be deducted for this task

1. Implement a new **Glider** class that implements the **PassiveLift** interface and uses the following functionality:
   * The Glider’s constructor should accept a:
     1. **maxHeight: number** – the max altitude that can be reached with the Glider
     2. **descentSpeed: number** – used in the **getAltitudeChange** calculation
   * **maxHeight: number** – a public readonly property that represents the maximum altitude that the Glider can reach.
   * **liftCoefficient:** number – a public static readonly property that represents the lift coefficient of all gliders – initialized with a predefined value of **0.6**
   * **getAltitudeChange** – calculates the altitude change using the **descentSpeed** and l**iftCoefficient**. A glider cannot gain altitude, but instead always **loses altitude** equal to:
     1. **(descentSpeed \* (1 - liftCoefficient))**
2. All **literal** and **advanced types** in the **FlyingMachine** and implementations written by you should be replaced with equivalent **type aliases** for easier readability – **reuse interfaces/type aliases where possible**.
3. **All access modifiers** of properties and methods **in your code**, **that are not part of the interfaces, the provided implementations or the required structure explained in this document**, should use either the **private or protected** access modifiers
   * In other words helper methods or extra properties you define to help you achieve the required functionality should **not be public**

#### Decoration (30 Points)

You are tasked to change the functionality of the following class:

* **SolidRocketBooster** -Your task is to change how the altitudeChange of the SolidRocketBooster is calculated, if the SolidRocketBooster instance was created with the **optimalWeight** parameter and the **optimalWeight** **is** **lower than** the FlyingMachine’s **current weight**, the altitudeChange produced from **getAltitudeChange** method **should be halved**. You are **NOT allowed** to directly modify the files inside the **/contracts** folder.

**IMPORTANT:** The modifications should change the class definition itself – check Example 6 to see that the expected modifications are directly available on the **SolidRocketBooster** class definition.

* Creating a derived class from **SolidRocketBooster** and modfiying the functionality there, while the functionality remains unchanged in **SolidRocketBooster will award NO points.**
* You are expected to use valid TS and typization to accomplish this task, so any of the following can result in penalties to the points from this task:
  + TS errors in your implementation or because of incompatible decorators
  + Solutions that avoid or ignore TS’s type checks (i.e. you should use appropriate types instead of trying to avoid the type checks)
  + Unnecessary mutation or redefinition of existing members (i.e. try to have the smallest impact on the existing definition)

### Examples

This is an example of how the **FlyingMachine** class is **intended to be used**:

* Examples 1-4 show the basic functionality
* Example 5 shows the usage of the Glider class
* Example 6 is meant to test the modified SolidRocketBooster (that can use the optimalWeight and the FlyingMachine’s current weight in its calculations)

**NOTE:** Examples 1-5 don’t use the optional optimalWeight parameter, so they should work regardless of your implementation of the Decoration task

|  |
| --- |
| **Example 1** |
| let activeLift = new SolidRocketBooster(10, 7.5);  let propellant: Propellant = {      fuelAmount: 20,      get fuelWeight() { return this.fuelAmount \* 2 }  };  let flyer = new FlyingMachine<'Active'>(activeLift, propellant, 1100, 0);  console.log(flyer.move());  console.log(flyer.move());  console.log(flyer.move());  console.log(flyer.checkStatus()) |
| **Output 1** |
| **Flyer moved 75.00 meters up**  **Flyer moved 75.00 meters up**  **Flyer stayed in place**  **Flyer altitude: 150 meters**  **Flyer weight: 1100**  **Fuel left: 0** |

|  |
| --- |
| **Example 2** |
| let passiveLift = new HeliumBaloon(30000);  let helium: LiftingGas = { liftingPower: 12 }  let flyer = new FlyingMachine<'Passive'>(passiveLift, helium, 350, 0);  console.log(flyer.move());  console.log(flyer.move());  console.log(flyer.move());  console.log(flyer.move());  console.log(flyer.checkStatus())  console.log(FlyingMachine.TotalMetersMoved); |
| **Output 2** |
| **Flyer moved 12.00 meters up**  **Flyer moved 12.00 meters up**  **Flyer moved 11.99 meters up**  **Flyer moved 11.99 meters up**  **Flyer altitude: 47 meters**  **Flyer weight: 350**  **Max height: 30000**  **47.971207679232** |

|  |
| --- |
| **Example 3** |
| let activeLift = new SolidRocketBooster(12, 10);  let propellant: Propellant = {      fuelAmount: 150,      get fuelWeight() { return this.fuelAmount \* 1.3 }  }  let flyer = new FlyingMachine<'Active'>(activeLift, propellant, 2500, 10);  let passiveLift = new HeliumBaloon(30000);  let helium: LiftingGas = { liftingPower: 20 }  let flyer2 = new FlyingMachine<'Passive'>(passiveLift, helium, 350, 0);  console.log(flyer.move());  console.log(flyer.move());  console.log(flyer.checkStatus())  console.log(flyer2.move());  console.log(flyer2.move());  console.log(flyer2.checkStatus())  console.log(FlyingMachine.TotalMetersMoved); |
| **Output 3** |
| **Flyer moved 120.00 meters up**  **Flyer moved 120.00 meters up**  **Flyer altitude: 250 meters**  **Flyer weight: 2663.8**  **Fuel left: 126**  **Flyer moved 20.00 meters up**  **Flyer moved 19.99 meters up**  **Flyer altitude: 39 meters**  **Flyer weight: 350**  **Max height: 30000**  **279.9866666666667** |

**Note:** This example is meant to check correct typization and constraints, the comments in the output are not the result of runtime execution, but errors that TS should flag on the red text in the input.

|  |
| --- |
| **Example 4** |
| let activeLift: ActiveLift = {      getAltitudeChange() { return 30; },      fuelConsumptionRate: 10,      liftPerFuelUnit: 1.5,      optimalWeight: undefined  };  let liftingGas: LiftingGas = { liftingPower: 8 }  let flyer = new FlyingMachine<'Active'>(activeLift, liftingGas, 500, 0);  let flyer2 = new FlyingMachine<'Passive'>(activeLift, liftingGas, 500, 0); |
| **Output 4** |
| **//TypeScript Error: 'LiftingGas' is not assignable to parameter of type 'Propellant'**  **//TypeScript Error: 'ActiveLift' is not assignable to parameter of type 'PassiveLift'** |

|  |
| --- |
| **Example 5** |
| let glider = new Glider(2500, 50);  let air: LiftingGas = { liftingPower: 0 };  let flyer = new FlyingMachine<'Passive'>(glider, air, 125, 55);  console.log(flyer.move());  console.log(flyer.checkStatus())  console.log(flyer.move());  console.log(flyer.move());  console.log(flyer.move());  console.log(flyer.checkStatus()); |
| **Output 5** |
| **Flyer moved 20.00 meters down**  **Flyer altitude: 35 meters**  **Flyer weight: 125**  **Max height: 2500**  **Flyer moved 20.00 meters down**  **Flyer moved 15.00 meters down**  **Flyer stayed in place**  **Flyer altitude: 0 meters**  **Flyer weight: 125**  **Max height: 2500** |

|  |
| --- |
| **Example 6** |
| let activeLift = new SolidRocketBooster(10, 7.5, 800);  let propellant: Propellant = {      fuelAmount: 150,      get fuelWeight() { return this.fuelAmount \* 2 }  };  let flyer = new FlyingMachine<'Active'>(activeLift, propellant, 505, 0);  console.log(flyer.checkStatus());  console.log(flyer.move());  console.log(flyer.checkStatus());  console.log(flyer.move());  console.log(flyer.checkStatus()); |
| **Output 6** |
| **Flyer altitude: 0 meters**  **Flyer weight: 805**  **Fuel left: 150**  **Flyer moved 37.50 meters up**  **Flyer altitude: 37 meters**  **Flyer weight: 785**  **Fuel left: 140**  **Flyer moved 75.00 meters up**  **Flyer altitude: 112 meters**  **Flyer weight: 765**  **Fuel left: 130** |