Obecné informace

- Zadání musíte splnit v časovém limitu 3 hodiny, který začíná po vysvětlení zadání a zodpovězení dotazům k zadání. Je možné pokládat dotazy i během testu, ale tak, aby nerušily vaše kolegy při práci.
- Vaše řešení bude testováno v prostředí ReCodEx. Je tedy nutné dbát na detaily ze zadání (např. co se týče přesného obsahu chybových/informačních hlášení).
- Hodnotí se také tvar odevzdaného souboru, tedy navrhněte vhodnou architekturu, dodržujte programátorské konvence, dbejte na úpravu, pište hezký, dobře členěný a hlavně čitelný kód. Programy bez rozumné dekompozice (např. obsahující pouze pár metod), případně programy s nečitelnými zdrojovými kódy a jim podobné nebudou uznány!
- Je dovoleno používat tištěné i elektronické materiály, pokud to není přímo řešení dané (nebo podobné) úlohy nebo její dílčí části. Nezapomeňte, že pokud kopírujete nějaký kód, který Vám nepatří, je nutné uvést zdroj! Není dovolena jakákoliv forma komunikace s kýmkoliv z jakéhokoliv důvodu (kromě zkoušejících) - do toho se počítá i to, že není dovolena komunikace s libovolnou umělou inteligencí (AI) nad rámec standardní IntelliSense ve Visual Studiu (tj. je zakázáno používat služby jako ChatGPT, GitHub Copilot, apod.).
- Až budete s Vaším řešením spokojení, přihlaste se a vyčkejte na zkoušejícího, který s Vámi řešení projde. Pokud zkoušející najde ve Vašem řešení nějaké problémy, můžete je v rámci časového limitu odstranit. Pokud ze zápočtového testu samovolně odejdete bez toho, aby zkoušející Vaše řešení spolu s Vámi procházel, tak jste zápočtový test nesložili (a to i pokud Vám v ReCodExu prošly testy na 100%) - osobní předvedení zkoušejícímu na místě je povinnou součástí testu.
- Řešení je nutné splnit na 100% správnosti.

Zadání: Tower Defense Engine

Vaším úkolem je naprogramovat aplikaci, která simuluje strategickou hru typu Tower Defense. Aplikace na standardním vstupu přijímá příkazy, které manipulují s herním světem (přidávají věže, nepřátele) a posouvají herní čas. Příkazy se provádí sekvenčně v pořadí, v jakém je uživatel na standardním vstupu zadává.

Předpokládejte, že **vstup je vždy validní**. Nemusíte ošetřovat chybné vstupy, neznámé příkazy ani chyby v logice (např. umístění věže na políčko mimo meze herní mapy).

Aplikace musí být spustitelná s jedním argumentem příkazové řádky, který určuje délku herní mapy.

Povinná struktura aplikace

Pro implementaci zpracování příkazů **musíte** vyjít z připravené kostry kódu.

Stáhněte si kostru aplikace ___ JE NA KONCI DOKUMENTU

Důležité pokyny k implementaci:

- Využití rozhraní `ICommand` a připravených tříd je **povinné**.
- Kostru však můžete (a budete muset) **libovolně upravovat a rozšiřovat**.
- Všimněte si, že připravená metoda `Execute` nemá přístup k herní mapě. Je na vás, jak ji upravíte (změna parametrů metody, předání mapy v konstruktoru apod.), aby příkazy mohly reálně ovlivňovat hru.
- Načítání vstupu a jeho parsování (rozdělení řádku na příkaz a argumenty) si musíte naimplementovat sami.

Specifikace herního světa

Hra se odehrává na jednorozměrné mapě o délce `N`, kde `N` je číslo zadané jako argument při spuštění programu (`TrackLength`). Pozice na mapě jsou indexovány od `0` do `N-1`.

- **Nepřátelé** začínají vždy na pozici `0` a pohybují se směrem k `N`.
- **Věže** jsou umístěny na fixních pozicích a útočí na nepřátele v dosahu.
- **Konec hry:** Pokud se jakýkoliv živý nepřítel dostane na pozici `>= N`, hra končí prohrou.

Na jedné pozici se může vyskytovat více entit (například více nepřátel nebo nepřítel a věž).

Herní smyčka a výpis

Po zpracování **každého** příkazu (ať už jde o přidání entity nebo ukončení kola) se na standardní výstup vypíše aktuální stav hry (více v sekci Formát výstupu).

Simulace času probíhá v kolech. Příkaz `END_ROUND` ukončí aktuální kolo a provede následující kroky v přesně tomto pořadí:

1. **Pohyb nepřátel:** Všichni nepřátelé se posunou o svou aktuální rychlost.
2. **Útok věží:** Všechny věže provedou útok dle svých pravidel.
3. **Odstranění mrtvých:** Nepřátelé, jejichž životy klesnou na 0 a méně, jsou odstraněni.
4. **Výpis stavu:** Vypíše se aktuaní pozice nepřátel a veží (více ve "Formát výstupu")
5. **Kontrola prohry:** Pokud je živý nepřítel mimo mapu, vypíše se "`Game Over!`" a program končí.

Specifikace Entit

Nepřátelé

Vrabec:

- **Jméno**: Sparrow
- **HP:** 10
- **Rychlost:** 2 políčka za tah.

Ježek:

- **Jméno**: Hedgehog

- **HP:** 15
- **Rychlost:** 1 políčko za tah.
- **Schopnost:** Má krunýř. Každé obdržené poškození se dělí dvěma (celočíselné dělení). Vždy však utrží poškození alespoň 1.

Bobr:

- **Jméno**: Beaver
- **HP:** 5
- **Rychlost:** Začíná na 1. Po každém pohybu se jeho rychlost zvýší o 1, až do maxima 3.
- *(Příklad: 1. tah se posune o 1, 2. tah o 2, 3. tah o 3, 4. tah o 3...)*

Věže

Věže stojí na zadané pozici.

Kaktus:

- **Jméno**: Cactus
- **Dosah:** Pouze políčko, na kterém stojí.
- **Útok:** Udělí **5 poškození** všem nepřátelům na své pozici.

Kanón na vrabce:

- **Jméno**: Cannon
- **Dosah:** Políčko věže +/- 1 (celkem 3 políčka).
- **Útok:** Udělí **3 poškození** všem **Vrabcům** (Sparrow) v dosahu. Ostatní ignoruje.

Vuvuzela:

- **Jméno**: Vuvuzela
- **Dosah:** 2 políčka na každou stranu (Věž +/- 2).
- **Útok:** Udělí **1 poškození** všem nepřátelům v dosahu.
- **Mechanika přehřátí (Cooldown):** Po každém úspěšném útoku (pokud v tomto tahu někoho zranila) se Vuvuzela musí chladit a následující **2 tahy** neútočí.

Formát příkazů

Jednotlivé argumenty vstupních příkazů jsou odděleny jednou nebo více mezerami.

```
ADD_ENEMY enemyType
```

- Přidá na začátek mapy (pozice 0) nového nepřítele typu `enemyType`.
- Povolené typy: `HEDGEHOG`, `SPARROW`, `BEAVER`.

```
PLACE_TOWER towerType position
```

- Postaví věž typu `towerType` na pozici `position` (celé číslo).
- Povolené typy: `CACTUS`, `CANNON`, `VUVUZELA`.

```
END_ROUND
```

- Provede simulaci jednoho herního kola.

Formát výstupu

Po **každém** příkazu (včetně `ADD_ENEMY` atd.) vypište seznam nepřátel a věží.

- Pozice vypisujte vždy jako **třímístné číslo** zarovnané nulami (např. `000`, `005`, `012`). *Hint: Ve formátovacím řetězci lze použít `"{0:D3}"`.*
- Sekce `Enemies:` je vždy první, následuje sekce `Towers:`.
- Uvnitř sekcí musí být položky seřazeny **abecedně (lexikograficky)** podle jejich textové reprezentace (celého řádku, který se tiskne).
- *Hint:* V C# se vám může hodit metoda `List<string>.Sort()`, která řadí textové řetězce lexikograficky.

Vzhled řádků:

- Nepřítel: `[<pozice>][<Jméno>: HP=<životy>]`
- Věž: `[<pozice>][<Jméno>]`

Vizualizace: Pro snadnou kontrolu vašeho výstupu můžete použít tento vizualizátor: https://erunno.github.io/tower-defence-viz. Do vizualizátoru stačí zkopírovat celý výstup vaší konzole.

Poznámka: Příklady níže lze vložit i se symboly > a <.

Příklady

Řádky se znaménkem větší než (>) jsou ty, které program vypisuje na standardní výstup. Řádky se znaménkem menší než (<) jsou ty, které uživatel píše na standardní vstup.

Spuštění: `TowerDefense.exe 10`

Příklad 1: Základní mechaniky *(Všimněte si, že stav se vypisuje po každém příkazu)*

```
< PLACE_TOWER CACTUS 2
> Enemies:
> Towers:
> [002][Cactus]
< ADD_ENEMY HEDGEHOG
> Enemies:
> [000][Hedgehog: HP=15]
> Towers:
> [002][Cactus]
< END_ROUND
> Enemies:
> [001][Hedgehog: HP=15]
> Towers:
> [002][Cactus]
< END_ROUND
> Enemies:
> [002][Hedgehog: HP=13]
> Towers:
> [002][Cactus]
```

(Vysvětlení posledního kroku: Ježek se posune na pozici 2. Kaktus útočí (5 dmg). Ježek redukuje poškození (5/2 = 2 dmg). Ježkovi zbývá 13 HP.)

Příklad 2: Vuvuzela a Cooldown

```
< PLACE_TOWER VUVUZELA 4
> Enemies:
> Towers:
> [004][Vuvuzela]
```

```
> Enemies:
> [000][Sparrow: HP=10]
> Towers:
> [004][Vuvuzela]
< END_ROUND
> Enemies:
> [002][Sparrow: HP=9]
> Towers:
> [004][Vuvuzela]
< END_ROUND
> Enemies:
> [004][Sparrow: HP=9]
> Towers:
> [004][Vuvuzela]
```

(Vysvětlení: V prvním kole (`END_ROUND`) se Vrabec posune na pozici 2. Je v dosahu Vuvuzely (4 +/- 2). Dostane 1 dmg a Vuvuzela se přehřeje. V druhém kole Vrabec skočí na 4. Vuvuzela by ho zasáhla, ale má cooldown, takže neútočí.)

Příklad 3: Řazení výstupu *(Při použití lexikografického řazení je '001' (Beaver) před '001' (Hedgehog), protože B je před H. V druhém kole Bobr zrychlí, předběhne Ježka a vyhne se věži.)*

```
< PLACE_TOWER CACTUS 2
> Enemies:
> Towers:
> [002][Cactus]
< ADD_ENEMY HEDGEHOG
> Enemies:
> [000][Hedgehog: HP=15]
> Towers:
> [002][Cactus]
< ADD_ENEMY BEAVER
> Enemies:
> [000][Beaver: HP=5]
> [000][Hedgehog: HP=15]
> Towers:
> [002][Cactus]
< END_ROUND
> Enemies:
> [001][Beaver: HP=5]
> [001][Hedgehog: HP=15]
> Towers:
> [002][Cactus]
< END_ROUND
> Enemies:
> [002][Hedgehog: HP=13]
> [003][Beaver: HP=5]
> Towers:
> [002][Cactus]
```

(Vysvětlení posledního kroku: Bobr zvýší rychlost na 2 a posune se na pozici 3 (`1 + 2`). Ježek se posune na pozici 2. Kaktus útočí na pozici 2. Bobr je mimo dosah (v bezpečí). Ježek dostane zásah (zbývá 13 HP). Všimněte si, že ve výpisu je nyní `[002]` před `[003]`.)

ReCodEx Testy

Všechny testy jsou ke stažení zde. Vsupní soubory jsou pomenované jako `<test_id>_<track_length>.in`.

General Information

```
// ============================================================================
// MANDATORY APPLICATION STRUCTURE
//
// 1. You MUST use this Command pattern in your solution to handle commands.
// 2. You are free (and expected) to EXTEND this skeleton as you like and need.
//    - You can change the Execute method parameters (e.g., to pass the GameMap).
//    - You can add constructors, fields, and properties.
//    - You will need to implement your own input parsing logic (not provided here).
// ============================================================================

public interface ICommand
{
    // Command identifier (e.g., "ADD_ENEMY")
    public string Name { get; }

    // Method to execute the command.
    // NOTE: In this basic form, the method has no access to the game world.
    // It is up to you to modify the signature or the class to allow
    // manipulation of the map, towers, and enemies.
    public void Execute(string[] args);
}

public sealed class AddEnemyCommand : ICommand
{
    public string Name => "ADD_ENEMY";

    public void Execute(string[] args)
    {
        // args[0] = enemyType
        // Add logic to create and add the enemy to the game here.
    }
}

public sealed class PlaceTowerCommand : ICommand
{
    public string Name => "PLACE_TOWER";

    public void Execute(string[] args)
    {
        // args[0] = towerType
        // args[1] = position
        // Add logic to place the tower here.
    }
}

public sealed class EndRoundCommand : ICommand
{
    public string Name => "END_ROUND";

    public void Execute(string[] args)
    {
        // args are empty
        // Add logic to advance the round (move, attack, cleanup) here.
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Implement here (or in another class) the main game loop:
        // 1. Game initialization (read map size).
        // 2. Command registration.
        // 3. Loop for reading input, parsing lines, and calling the correct Command.
    }
}
```