# Event System

## Event

This class provides a generic representation of an event, allowing you to define events with a specific type T where T is a template

### Properties:

**T m_type**: Type of the event. Generally will be an enum or string type.

### Methods:

**GetType()**: Returns the event type.

## Event Core Types

Every event has to inherit from class Event. These events are defined as “core” events as they have been predefined for the engine architecture.

#### Player Input Events:

Group of objects that inherit from Event type “PlayerInputEvents” enum.
*PlayerMoveEvent,  PlayerJumpEvent, PlayerAttackEvent, PlayerDieEvent.*

#### enum PlayerInputEvents:

Enum that contains all the event types predefined. *(MOVE, JUMP, ATTACK, DIE).*

#### Game State Events:

Group of objects that inherit from Event type “GameStateEvents” enum.
*StartGameEvent,  PauseGameEvent, GameOverEvent, ExitGameEvent.*

#### GameStateEvents:

Enum that contains all the event types predefined. *(START, PAUSE, GAME_OVER, EXIT).*

Entity Interactions Events:

Group of objects that inherit from Event type “EntityInteractionsEvents” enum.
*CollissionEvent,  TriggerEvent, PickUpEvent.*

EntityInteractionsEvents:

Enum that contains all the event types predefined. *(COLLISION, TRIGGER, PICKUP).*

Every Event class inheriting from Event can contain its own data which can be used later on the received callback. Follow the example class below:

```
class PlayerAttackEvent : public Event<PlayerInputEvents>
{
public:
    PlayerAttackEvent() : Event<PlayerInputEvents>(PlayerInputEvents::ATTACK) {}
    virtual ~PlayerAttackEvent() = default;
    float damage = -1;
};
```

# EventHandler

This class handles a map containing the events from a type and its listeners.

Properties:

private

**unordered_map<T, vector<pair<ListenerID, Func>>> m_listeners:** Map to handle listeners for predefined (core) events.

**unordered_map<T, vector<pair<ListenerID, FuncArgs>>> m_genericListeners**: Map to handle listeners for generic (custom) events.

**ListenerID listenerIndex:** Listener id generation index.

using

*using Func = function<void(const Event<T>&)>;*
*using FuncArgs = function<void(const Event<string>&, const vector<any>&)>;*
*using ListenerID = int;*

Methods:

public

**EventHandler()**: default constructor.

**~EventHandler():** default destructor.

**ListenerID AddListener(T type, const Func& callback):**
Add a listener to the type of event.
**type:** Event type *(Generally enum or string)*.
**callback:** Callback listener callback with an event type as a parameter.
returns Listener id generated.

**ListenerID AddListener(T type, const FuncArgs& callback):**
Add a listener to the type of event.
**type:** Event type *(Generally enum or string)*.
**callback:** Callback listener callback with an event type as a parameter.
returns Listener id generated.

**bool RemoveListener(const T& type, const ListenerID & id, bool isGenericEvent = true):**
Removes a listener based on its type and listener ID.
**type:** Type of the event where to remove the listener.
**id:** Listener ID generated when added.
**isGenericEvent:** If it is a core type event or a custom one. Depending on which one it will search in the core map or the generic map for events.
returns  Operation success result.

**void DispatchEvent(const Event<T>& event):**
Dispatch the event to all its listeners.
**event:** Event to be dispatched.

**void DispatchEvent(const Event<string>& event, const vector<any>& args):**
Dispatch the event to all its listeners.
**event:** Event to be dispatched.
**args:** event args to be dispatched.

# EventManager

This class manages all the events and event handlers. Other objects can subscribe, unsubscribe and send events through this object.

Properties:

private

**EventHandler<PlayerInputEvents>* m_playerInputEventsHandler:** Player Input event handler.

**EventHandler<GameStateEvents>* m_gameStateEventsHandler:** Game State event handler.

**EventHandler<EntityInteractionsEvents>* m_entityInteractionsEventsHandler:** Entity Interactions event handler.

**EventHandler<string>* m_genericEventsHandler:** Generic event handler.

**static unique_ptr<EventManager> m_instance:** Singleton instance.

Methods:

Public

**EventManager():** Constructor.

**~EventManager():** Destructor.

**static EventManager* getInstance():** Returns Event Manager Singleton.

**EventHandler<PlayerInputEvents>* GetPlayerInputEventHandler()const:** Returns player input event handler.

**EventHandler<GameStateEvents>* GetGameStateEventHandler()const:** Returns game state event handler.

**EventHandler<EntityInteractionsEvents>* GetEntityInteractionsEventHandler()const:** Returns entity interactions event handler.

**EventHandler<string>* GetGenericEventHandler()const:** Returns generic event handler.

**static void AddEventListener(const string& eventID, function<void(const Event<string>&, const vector<any>&)> callback):**
Add a new listener to the generic event handler.
**eventID:** Event identifier.
**callback:** Function that contains the event to subscribe and the event args.

this example below shows how to add a new listener using a lambda expression:

```
//Example of a generic event subscription with a lambda
EventManager::AddEventListener(CUSTOM_EVENT_1, [](const Event<string>& e, const vector<any>& args) {
    try
    {
        int code = any_cast<int>(args[0]);
        auto message = any_cast<string>(args[1]);
        cout << "Event received: " << e.GetType() << ", Code: " << code << ", Message: " << message << endl;
    }
    catch (const bad_any_cast& ex)
    {
        cerr << "Error: " << ex.what() << endl;
    }
});
```

Definitions

***AddPlayerInputEventListener, AddGameStateEventListener, AddEntityInteractionsEventListener, AddGenericEventListener***
Add listeners: Adds a new listener to an eventhandler based on the event type. Binds callback function with sender. Returns the listener ID.

***RemovePlayerInputEventListener, RemoveGameStateEventListener, RemoveEntityInteractionsEventListener, RemoveGenericEventListener***
Remove listeners: Removes a new listener from an eventhandler based on the event type and a listener ID. Returns success result.

***SendPlayerInputEvent, SendGameStateEvent, SendEntityInteractionsEvent, SendGenericEvent***
Send a new event using an event handler based on the event type.