Inspector Math Expressions

Table of Content

Introduction

Inspector Math Expressions is a tiny Editor extension,
which allows you to edit formulas **directly in Inspector** without recompilation code!

Why is the best approach?
Good game balance is an important part of the game.
Formulas are one of main part, which is constantly being edited. Every time recompile the project - is **a terrible idea**!
Especially if you have a game designer in a team, he could easily edit balance formulas directly in the game without modifying the code.
If your game has already been published, and you want to upgrade the game balance - it is **not a problem**.
Simply update asset bundles with formulas - and you have a new game balance (*update asset bundles is not included in this asset*). Especially it is important for iOS, which updated version of the application can take up to 10 days.

With **Inspector Math Expressions** you will:

- Math expressions as fields of class
- Math expressions as independent ScriptableObject
- Editing math expressions inside Inspector window
- Easy creation of new math expressions templates
- Basic set of mathematical functions for use inside the formula (**+,-,/,*,^, (), sqrt, abs, ln, floor, ceil, round, sin, cos, tan, asin, acos, atan, atan2, min, max, rnd, clamp, E, PI**)
- Extension capabilities with own math functions.

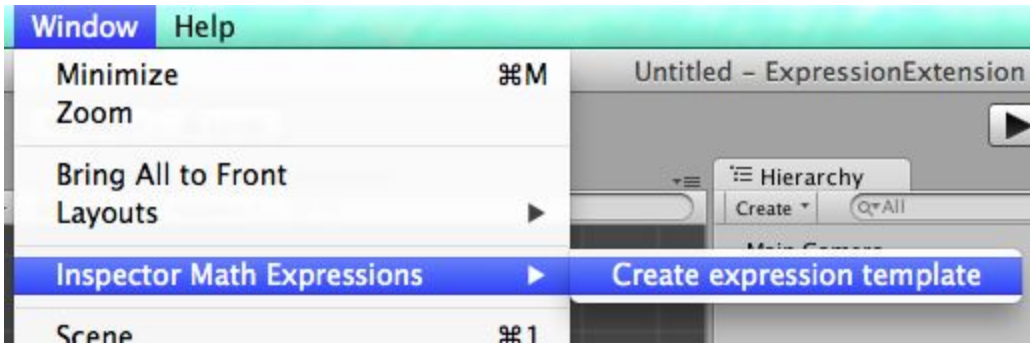
Supports all platforms.

Installation

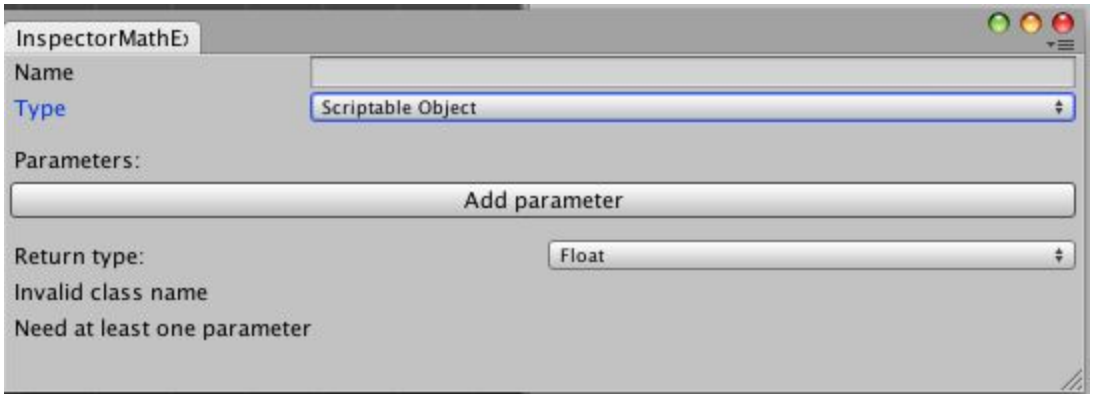
Asset does not require special installation and it can be located in any folder of the project.

Math expression creation

After installation choose menu item *“Window->Inspector Math Expression->Create expression template*” will be available:

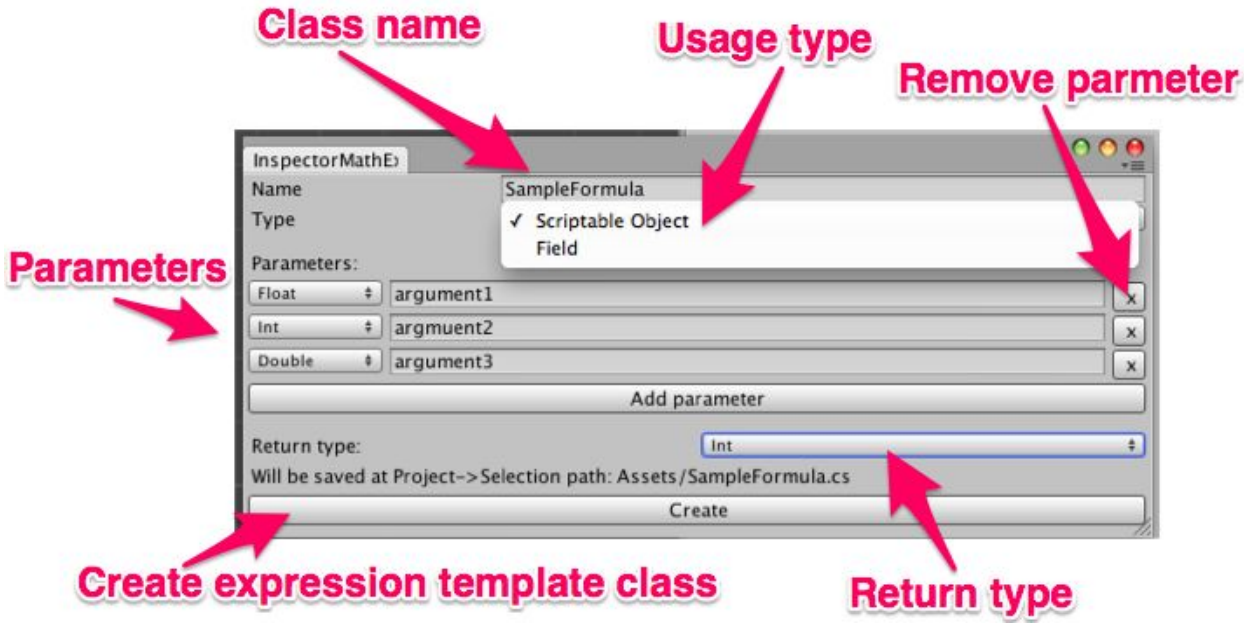


To start creating math expression template click on it, after that creation window will show:



Name - expression template class name.
Type - way of usage
ScriptableObject - expression as ScriptableObject
Field - expression as filed of any class
Parameters - list of parameters which will be used in expression
Parameter can be one of the three types Int, Float, Double (can be used any other type, but not directly, ***find below***)
Return type - expression return type (Int, Float, Double).
Button “Add parameter” - adding new parameter.
Errors - list of errors which should be fixed before creation.

Example of creation:

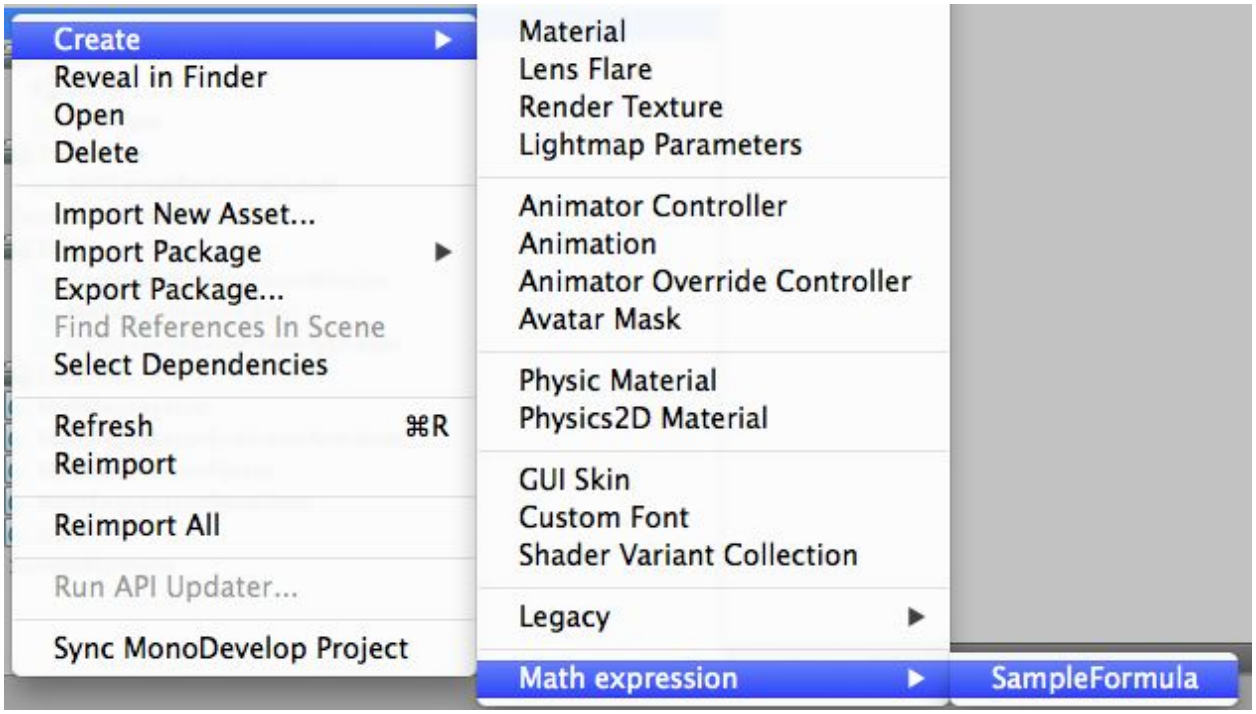


ScriptableObject

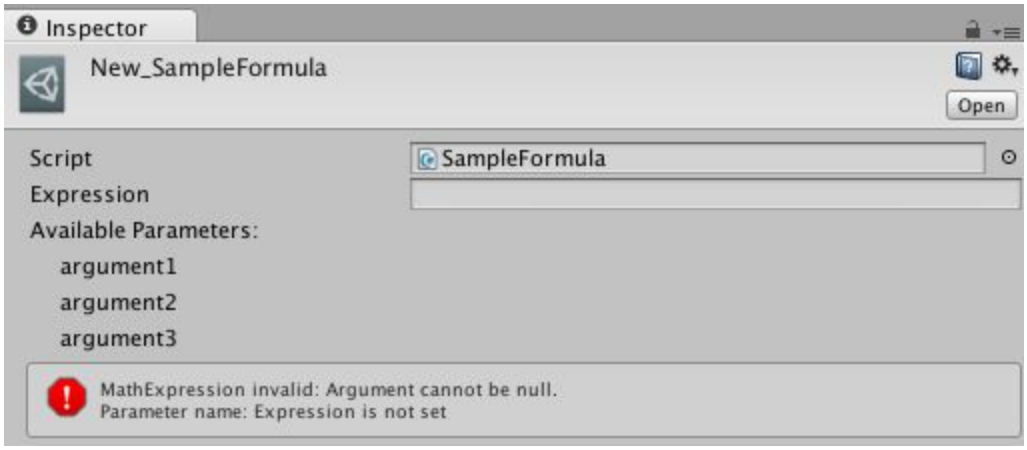
Class of expression template is created with type ScriptableObject:

```
using InspectorMathExpressions;

public class SampleFormula : MathExpression<SampleFormula>
{
   [MathExpressionEvaluator]
   public int Get(float argument1, int argument2, double argument3) {
       return (int)EvaluateMathExpression(argument1, argument2, argument3);
   }
#if UNITY_EDITOR
   public static partial class ScriptableObjectCreators
   {
      [UnityEditor.MenuItem("Assets/Create/Math expression/SampleFormula")]
      public static void CreateSampleFormula() {
         ScriptableObjectUtility.CreateAsset<SampleFormula>();
      }
   }
#endif
}
```

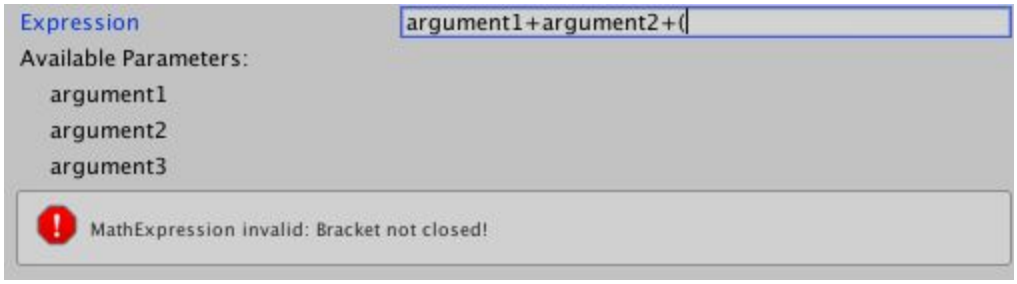
Menu item **”Create->Math expression->SampleFormula”** will be available after creation class



Click on it to create an instance of math expression.



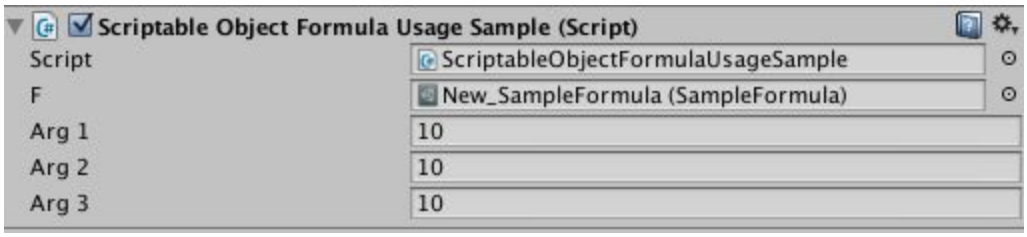
In “Expression” field, enter a formula according to the mathematical rules. If formula has a syntax error Help Box with error message will appear:



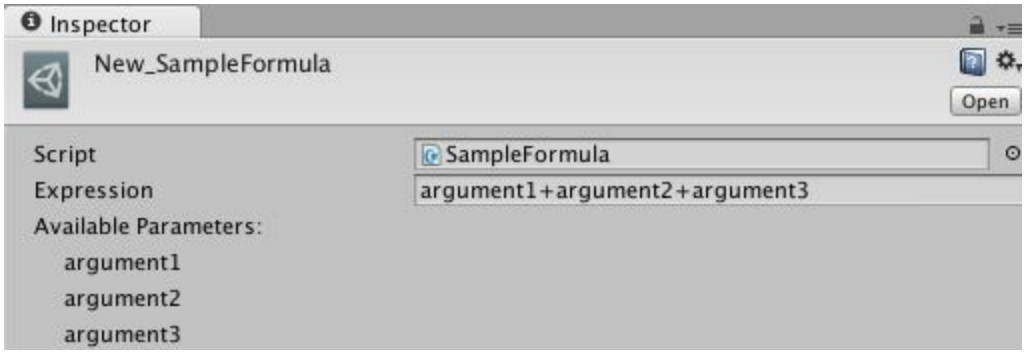
Expression instance can be use in any other object and scripts as object reference:

```
public class ScriptableObjectFormulaUsageSample : MonoBehaviour {
    public SampleFormula f;

    public float arg1;
    public int arg2;
    public double arg3;

    void Start() {
        Debug.Log(f.Expression + "= " + f.Get(arg1, arg2, arg3));
    }
}
```

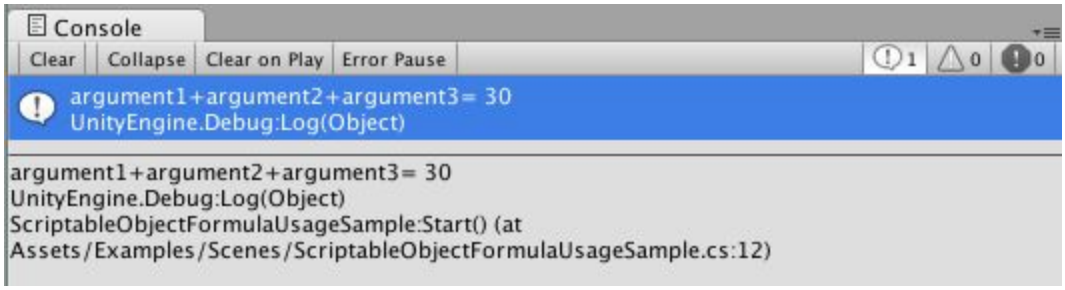
Component ScriptableObjectFormulaUsageSample in Inspector:



Field F contain reference to object New_SampleFormula:



Console log after running script:



Field

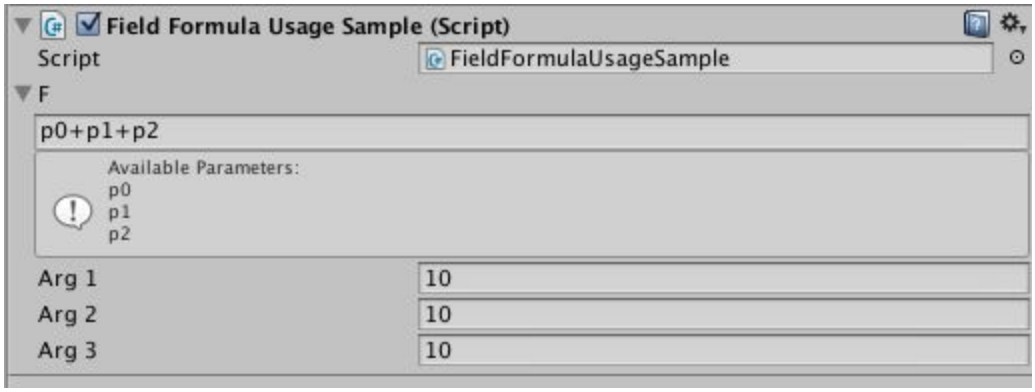
Class of expression template is created with type Field:

```
using InspectorMathExpressions;

[System.Serializable]
public class SampleFieldFormula : MathExpressionSerialized<SampleFieldFormula>
{
    [MathExpressionEvaluator]
    public int Get(float p0, int p1, double p2) {
        return (int)EvaluateMathExpression(p0, p1, p2);
    }
}
```

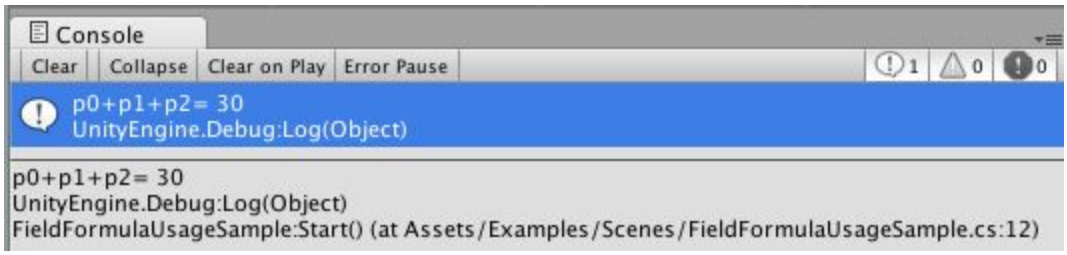
Expression can be used as field of any class:

```
public class FieldFormulaUsageSample : MonoBehaviour {
    public SampleFieldFormula f;

    public float arg1;
    public int arg2;
    public double arg3;

    void Start() {
        Debug.Log(f.Expression + "= " + f.Get(arg1, arg2, arg3));
    }
}
```

Component **FieldFormulaUsageSample** in Inspector:



Console log after running script:



Input math expression

The expression in “Expression” field is entered according to the standard mathematical rules.
Each parameter can be used unlimited number of times.
By default there are following mathematical operations:

f_sqrt - function System.Math.Sqrt .
f_abs - function System.Math.Abs .
f_ln - function System.Math.Log .
f_floor - function System.Math.Floor .
f_ceiling - function System.Math.Ceiling .
f_round - function System.Math.Round .
f_sin - function System.Math.Sin .
f_cos - function System.Math.Cos .
f_tan - function System.Math.Tan .
f_asin - function System.Math.Asin .
f_acos - function System.Math.Acos .
f_atan - function System.Math.Atan .
f_atan2 - function System.Math.Atan2 .
f_min - function System.Math.Min .
f_max - function System.Math.Max .
f_clamp - function Clamp.
f_clamp01 - function Clamp в пределах 0,1.

f_rnd - System.Random.NextDouble() call.

And constants:

k_pi - System.Math.PI

k_e - System.Math.E

Basic set can be extended own functions by modifying class **InspectorMathExpressions.Utils.**

If you need to change a set of functions and constants only for a particular template, you have to override properties Constants and Functions:

```
protected override Consts Constants {
    get {
        var c = base.Constants;
        c.Add("G", () => 9.8);
        return c;
    }
}

protected override Funcs Functions {
    get {
        var f = base.Functions;
        f.Add("CircleArea", (p) => System.Math.PI*p.FirstOrDefault()*p.FirstOrDefault());
        return f;
    }
}
```

Dice notation

Expressions supports dice notation, but '**@'** is used instead of '**d**' and '**#**' is used instead of **z**. For example 2d6 will be in "Expression" field 2@6. There are few examples: 2z8+5 : 2#8+5, 4d4-4 : 4@4-4, d100 : 1@100, etc. In any time you can change @ and # to d and z in **Utils.cs** file but it can lead to unreadable expressions with some arguments, for example expression *chargeCount@chargeForce* will be *chargeCountdchargeForce*.

Extended usage

Parameters objects of all types can be use as expression templates, but parameter should be converted to value of type Int, Float, or Double. For example, you can calculate value of resulting damage to the enemy when hit him:

```
[System.Serializable]
public class DamageCalculator : MathExpressionSerialized<DamageCalculator>
{
    public float Calc(Hero myHero, Hero enemyHero) {
        return DoCalc(myHero.level,
                      enemyHero.level,
                      myHero.attack,
                      enemyHero.block);
    }

    [MathExpressionEvaluator]
    float DoCalc(int myHeroLevel,
                 int enemyHeroLevel,
                 float myAttack,
                 float enemyBlock) {
        return (float)EvaluateMathExpression(myHeroLevel,
                                             enemyHeroLevel,
                                             myAttack,
                                             enemyBlock);
    }
}
```

External libraries

Asset uses a library UniLinq - an analogue of Linq, which works for mobile platforms.