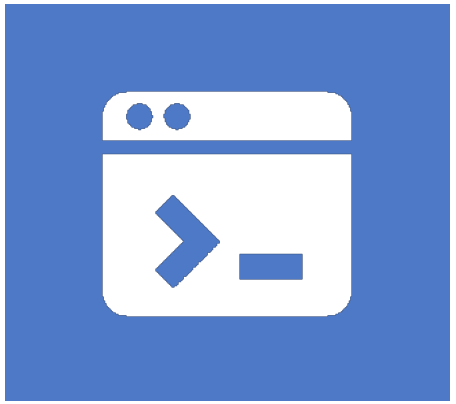
Advanced Game Programming



Week 9

Homework Review

Python and Tool Development

Trees in Practice

Class Based

What is a tree?

- What is a graph?
  - A graph is a set of nodes, with an arbitrary number of connections (edges) between nodes.
  - These edges can be either directed or undirected
  - These edges can be weighted or unweighted.
- A tree is a specialized graph:
  - All nodes except one (root) must have precisely one parent node,
  - All nodes can have an arbitrary number of children.
- What is special about trees?
  - From any node, you can reach any other node in the tree.
  - There are no cycles (you can’t return to a node from any other node.
  - The number of edges in a tree is exactly one less than the number of nodes.

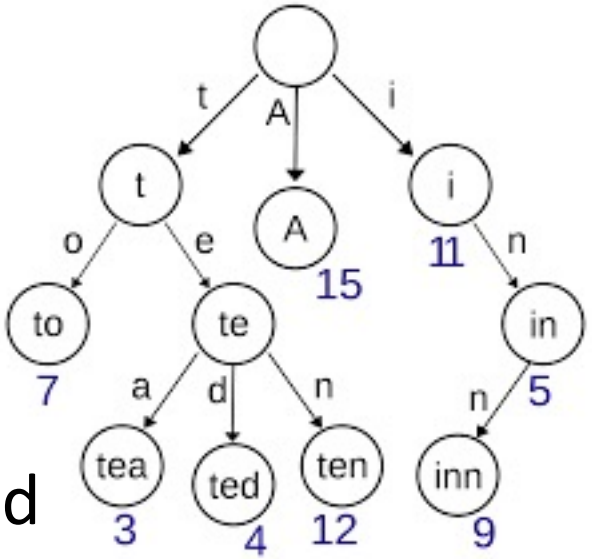
Types of Trees

- Binary Trees
- Pathfinding tree
- Prefix Trie

How to Implement Trees

- There’s no default implementation of trees in C#
- Trees are often highly data-dependent and can be implemented many ways.
  - Do you need to be able to go up and down from an individual node?
- Serialization needs can vary widely depending on the data

Simple Prefix Trie



- From the root, there's an edge for each letter
- From each following node, there's an edge for each valid prefix
- Each node has a bool for whether or not it's the end of a word.
- If there's not an edge, fall out of tree (not valid word)
- If the node you end on is false, it's not a valid word
- Otherwise, it's a word.

Prefix Trie Advantages

- PROS:
  - Lookup up data in a trie is faster in the worst case than a hash table
    - O(m) time (where m is the length of the search string)
    - HashTables are typically O(1), but worst case are O(N) technically
  - Looking up a prefix is far faster O(|prefix|)
  - A hash table can have collisions, a trie has no collisisions
  - No need to hash inputs
  - Can reduce quickly to an alphabetical ordering of entries.
- CONS:
  - Trie lookup is slower than hash table lookup, particularly if stored on hard disk
    - Random access memory instead of main memory lookup
  - Some trie implementations are larger than a hash table

Simple Implementation

```
public class Node
{
    private List<Node> children = new List<Node>();
    private char letter;
    private bool endsValidWord;
```

Checking Children

```
Frequently called  2 usages
public bool HasChild(char letterToCheck)
{
    return children.Any(n :Node => n.letter == char.ToUpper(letterToCheck));
}

Frequently called  1 usage
public Node GetChild(char letterToGet)
{
    return children.First(n :Node => n.letter == char.ToUpper(letterToGet));
}
```

Adding a Child

```
public Node GetOrAddChild(char letterToAdd, bool isEndOfWord = false)
{
    if (HasChild(letterToAdd))
    {
        return children.First(n :Node => n.letter == char.ToUpper(letterToAdd));
    }

    var toAdd = new Node { letter = char.ToUpper(letterToAdd), endsValidWord = isEndOfWord };

    children.Add(toAdd);

    return toAdd;
}
```

Initializing the Tree

```
public void AddWord(string word)
{
    if (word.Length == 0)
    {
        endsValidWord = true;
        return;
    }

    GetOrAddChild(word[0]).AddWord(word.Substring( startIndex: 1,  length: word.Length - 1));
}
```

Checking A Word

```
Frequently called  2 usages
public bool IsWord(string word)
{
    if (word.Length == 0)
    {
        return endsValidWord;
    }

    if (!HasChild(word[0]))
        return false;

    return GetChild(word[0]).IsWord(word.Substring( startIndex: 1,  length: word.Length - 1));
}
```

Filling the Tree

```
private Node CreateTree()
{
    var toReturn = new Node();

    foreach (var word :string  in wordlist.text.Split( params separator: '\n'))
    {
        if (string.IsNullOrEmpty(word)) continue;

        toReturn.AddWord(word);
    }

    return toReturn;
}
```

Using in a Scene

```
private void Awake()
{
    var stopwatch = new Stopwatch();
    stopwatch.Start();
    root = CreateTree();
    stopwatch.Stop();
    Debug.Log( message: "Time creating prefix trie: " + stopwatch.Elapsed.ToString( format: "g"));
}

Event function
private void Update()
{
    if (string.IsNullOrWhiteSpace(input.text))
    {
        display.text = "Enter text to see if it's a word in English.";
    }
    else
    {
        display.text = root.IsWord(input.text) ? "<color=green>Valid" : "<color=red>Invalid";
    }
}
```

Prefix Trie

Example Code.

Serialization

How to Serialize Tries

- Graphs can be infinitely recursive
- Can simply serialize as binary:
  - Write the number of possible characters in trie (*m*).
  - Write the depth of the trie (*d*)
  - Then, write *m* by *d* nodes out, one for each possible character for every depth
    - If it doesn't exist, write 0
    - If it does exist, write 1 if it's not the end of a word, and 2 if it is the end of a word.
- How is this inefficient?
- How can we account for those inefficiencies?

Serializing a General Graph

```
[DataContract]
class Node {
    [DataMember]
    public Node AnotherNode { get; set; }
}
```

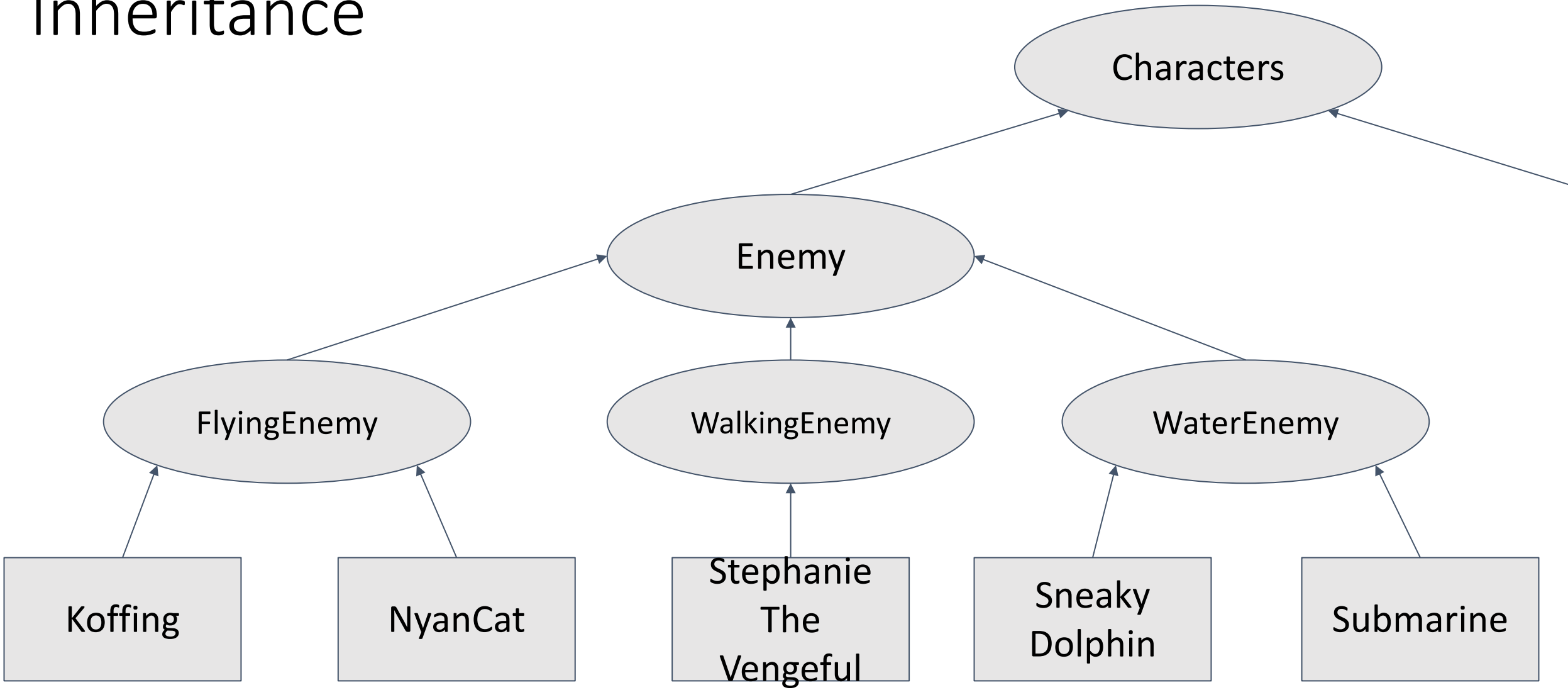
```
static class Program
{
    static void Main()
    {
        Node a = new Node(), b = new Node();
        a.AnotherNode = b;
        b.AnotherNode = a;

        DataContractSerializer dcs = new DataContractSerializer(
            typeof(Node), null, int.MaxValue, false, true, null);
        using (MemoryStream ms = new MemoryStream())
        {
            dcs.WriteObject(ms, a);
            // Example:
            ms.Position = 0;
            Node c = (Node) dcs.ReadObject(ms);
            Console.WriteLine(ReferenceEquals(c, c.AnotherNode.AnotherNode));
        }

    }
}
```

Inheritance , Composition and Reflection

Inheritance



Composition

Flying

Walking

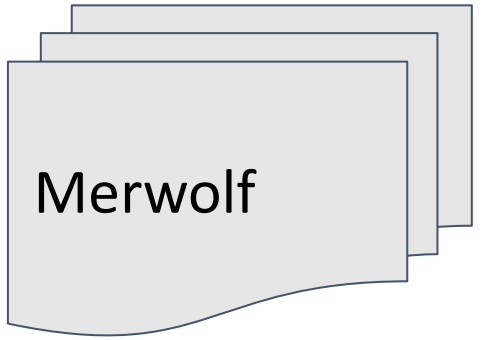
Swimming

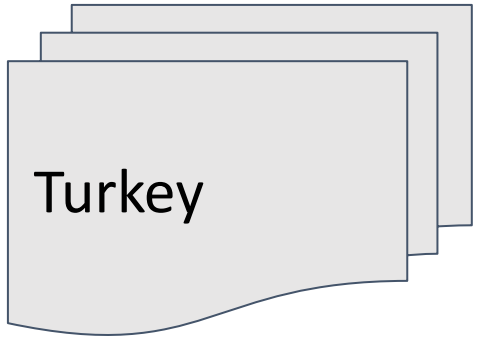
Talking

Punching

Dunking

Psychic Readings





Reflection

- Reflection is inspecting a class:
  - Methods
  - Constructors
  - Parameters
  - Properties
- Not widely used in game programming
  - Could be used in creating a game engine – to know how to expose methods and variables in the editor
  - Your IDE uses reflection to see what to suggest when you press ”.”
  - Also can be used for object serialization
- How is it different than composition?

Reflection on Class

```
Type t = typeof(SimpleClass);
BindingFlags flags = BindingFlags.Instance | BindingFlags.Static | BindingFlags.Public |
                     BindingFlags.NonPublic | BindingFlags.FlattenHierarchy;
MemberInfo[] members = t.GetMembers(flags);
Debug.Log($"Type {t.Name} has {members.Length} members:");
foreach (var member in members)
{
    string access = "";
    string stat = "";
    var method = member as MethodBase;
    if (method != null)
    {
        if (method.IsPublic)
            access = " Public";
        else if (method.IsPrivate)
            access = " Private";
        else if (method.IsFamily)
            access = " Protected";
        else if (method.IsAssembly)
            access = " Internal";
        else if (method.IsFamilyOrAssembly)
            access = " Protected Internal ";
        if (method.IsStatic)
            stat = " Static";
    }
    var output = $"{member.Name} ({member.MemberType}): {access}{stat}, Declared by {member.DeclaringType}";
    Debug.Log(output);
}
```

Output:

```
// The example displays the following output:
// Type SimpleClass has 9 members:
// ToString (Method): Public, Declared by System.Object
// Equals (Method): Public, Declared by System.Object
// Equals (Method): Public Static, Declared by System.Object
// ReferenceEquals (Method): Public Static, Declared by System.Object
// GetHashCode (Method): Public, Declared by System.Object
// GetType (Method): Public, Declared by System.Object
// Finalize (Method): Internal, Declared by System.Object
// MemberwiseClone (Method): Internal, Declared by System.Object
// .ctor (Constructor): Public, Declared by SimpleClass
```

Inheritance

- Pros:
  - Allows polymorphic behavior.
  - Is initially simple and convenient.
- Cons:
  - Can become complex or clumsy over time as more behavior and relations are added
- Inheritance (“is a” relationship)
  - Need polymorphism
  - Need to process many kinds of things similarly

Composition

- Pros:
  - Straightforward to incrementally extend relations and behavior.
- Cons:
  - No polymorphism (less convenient to use related information and behavior)
- Composition (“has a” relationship)
  - Describing attributes or behavior instead of core identity
  - Interfaces - middle ground

Ability and Inventory Systems

What Are the Considerations?

- Lots of redundant code for different items or abilities
- Every part of game code will touch these classes
- When any outside system changes, odds are good some item or ability will break
- It’s hard to define invariants (levels of all audio clips, etc.)

Why Not Lean Fully On Inheritance?

- Brittle Base Class Problem
  - Base Class will end up coupled to every system *any* derived class needs to talk to
  - Makes it difficult to change base class without breaking something
- Useless Base Class Problem
  - If base class becomes a list of methods that *might* do something in derived classes, you end up having nested checks and having almost none of the benefits of inheritance
- Deep Inheritance Tree Problem
  - The deeper your inheritance tree, the less useful your base case is in identifying behavior.

Why Not Lean Fully on Composition

- Alphabet Soup
  - Too many small components means too much searching through sets of components
  - What do you do if multiple components have conflicting implementations of Update loops?
- Everything's General
  - You end up making a lot of special case components to negate the effects of other components

A Mixed Solution

```
public class Item {
    private Effect[] effects;
    private uint cost;
    private bool unique;

    public class Effect {
        private Func<Actor> used, removed;
    }

    public class PermanentEffect : Effect { }

    public class OngoingEffect : Effect { }
}
```

```
public class Power {
    private Cast[] onCast;

    private class Effect {
        private Func<Actor, Actor> used;
        private float recharge;
    }

    private class AreaOfEffect : Effect {
        private Func<Actor, Actor> used;
        private float range, radius;
    }

    private class HealingEffects : Effect {
        private float amountHealed;
        private bool canTargetPet;
    }
}
```

Why Use Helper Classes

- It reduces the number of methods in the base class
- Code in the helper class is usually easier to maintain.
- It lowers the coupling between the base class and other systems.

- Other helper examples:
  - Sound Players
  - Visualizations
  - Serializer / Deserializer

Alternative – Data Driven Solution

```
public class Item {
    private string referenceID;
    private Dictionary<string, string> values;
```

```
public T ValueOf<T>(string key)
{
    if (!values.ContainsKey(key.ToUpper()))
    {
        Debug.Log("Couldn't find key " + key);
        return (T) (object) 0;
    }

    switch (Type.GetTypeCode(typeof(T)))
    {
        case TypeCode.Boolean :
            return (T) (object) bool.Parse(values[key.ToUpper()]);
        case TypeCode.String :
            return (T) (object) values[key.ToUpper()];
        case TypeCode.Int32 :
            return (T) (object) int.Parse("0" + values[key.ToUpper()]);
        case TypeCode.Single :
            return (T) (object) float.Parse("0" + values[key.ToUpper()]);
        case TypeCode.Byte:
            return (T) (object) byte.Parse("0" + values[key.ToUpper()]);
        case TypeCode.UInt32:
            return (T) (object) uint.Parse("0" + values[key.ToUpper()]);
        case TypeCode.Char:
            return (T) (object) char.Parse(values[key.ToUpper()]);
        default:
            Debug.Log("Type " + typeof(T) + " isn't implemented: going to try casting a string.");
            break;
    }
    return (T) (object) (values.ContainsKey(key.ToUpper()) ? values[key.ToUpper()] : "");
}
```

Data Driven Control

- Create a language you can parse into the game

```
839#credit&-19#stamina
```

- Create tools that output to that language
- Example Code

Facade Pattern

```
public class CarFacade
{
    private readonly CarAccessories accessories;
    private readonly CarBody body;
    private readonly CarEngine engine;
    private readonly CarModel model;

    public CarFacade()
    {
        accessories = new CarAccessories();
        body = new CarBody();
        engine = new CarEngine();
        model = new CarModel();
    }

    public void CreateCompleteCar()
    {
        Console.WriteLine("******** Creating a Car **********");
        model.SetModel();
        engine.SetEngine();
        body.SetBody();
        accessories.SetAccessories();

        Console.WriteLine("******** Car creation is completed. **********");
    }
}
```

Template Method

- Template method in a base (usually abstract) class that calls a series of methods in order.
- Subclasses of the base class fill in the empty or variant parts of the template
- Example of *inversion of control* because lower-level code determines what to run

Template Method Example

```
abstract class CardPlayEffect
{
    public virtual void OnReveal() {}
    public abstract void OnPlace();
    public virtual void TriggerEnemyTraps() {}
    public abstract void RegisterTurnEndEffects();

    public void PlayCard()
    {
        OnReveal();
        OnPlace();
        EffectNeighbors();
        TriggerEnemyTraps();
        RegisterTurnEndEffects();
    }
}
```

Considerations

- How easy is it to add or remove items?
- How easy is it for non-technical designers to add items?
- How easy is it to change implementation of items?
- How easy is it to change implementation of systems that interface with items?