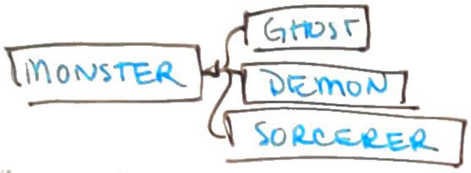
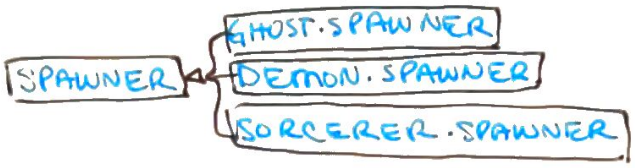
##GameProgrammingPatterns_Revisited_Prototype##

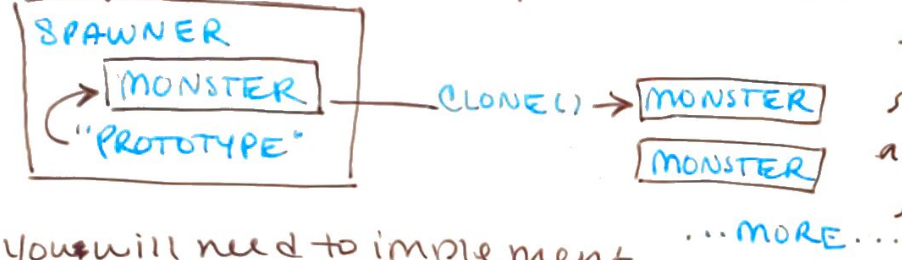
*ex. spawners in a Gauntlet-style game





"an object can spawn other objs similar to itself"
"Any monster can be used as a prototypal monster used to generate other versions of itself."

- give base class an abstract clone() method which when implemented provides a new obj === in class & state to itself
- Then define a single spawner which holds a hidden monster prototype template used to stamp out more monsters



*since the spawner holds state we can make one for a fast/slow ghost, big/small, strong/weak, etc. by creating a prototype of that ghost.

*you will need to implement clone() in each monster class & working through a deep vs. shallow clone

*This isn't saving us a ton of code & assumes each monster has its own class. *tend to use Component & Type Object to avoid needing separate classes per entity

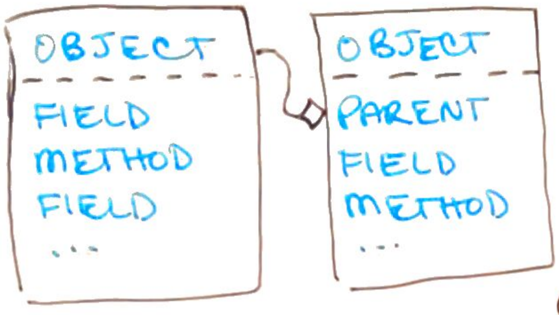
*Could create spawn[monster] functions instead & then store a function ptr in the spawner

*Can also use Templates so we're not hard coding monster classes

The Prototype Language Paradigm

"...the defining characteristic of OOP is that it tightly binds state & behavior together."

*Talks about the Self language that has delegation & no classes but still binds data & behavior. We look for inherited methods w/a ptr to its parent (which can be changed at runtime w/ dynamic inheritance)



- parent objs let us reuse behavior + state across ~~[illegible]~~ ∞ objs
- make new instances of self w/ cloning
- The author found self lang. difficult + not fun to use bc. complexity was pushed to the user

"In self it's as if every obj supports the Prototype design pattern automatically. Any obj. can be cloned."

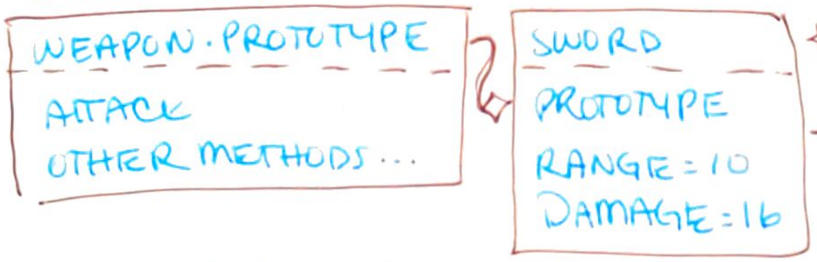
- Javascript was inspired by self + contains prototypes
  "An obj can also have another obj, called its "prototype", that it delegates to if a field access fails."
  ↳ but js doesn't have built-in cloning

Typical way to define types + objs in JS.
① ctor func. create new obj. + init. fields
② invo~~ke~~ke w/ new
↳ invokes body of Weapon() w/ $\boxed{\text{this}}$ bound to new, empty obj
body adds ~~so~~ fields + now-filled obj. is returned
+ wires up blank obj to delegate to a prototype obj
which you can get w/ Weapon. prototype

* state is added to ctor body (to define behavior)
* methods are added to prototype obj.

Weapon. prototype. attack = function() { ... }

- every obj. returned by $\boxed{\text{new}}$ delegates to weapon. prototype
it looks like this:



* create w/ "new" w/ obj that represents type (ctor)
* state stored in instance
* Behavior is delegated to prototype + is stored in separate obj. shared by all objs of a certain type

"the syntax + idioms of [J.S.] encourage a class based approach"

* delegation can be useful for Data Modeling + reusing data
Data entities are maps or property bags
* pick the simplest data entity + delegate to it

```
{
  "name": "goblin grunt",
  "min" : 20,
  "max": 30,
  "resists": ["cold", "poison"],
  "weaknesses": ["fire", "light"]
}
```

```
{"name": "goblin wizard",
 "prototype": "goblingrunt",
 "spells": ["fireball"]
}
```

```
{"name": "goblin archer",
 "prototype": "goblingrunt",
 "attacks" : ["short bow"]
}
```

* makes it easier to tell the difference b/w types & ↑ flexibility for designers
* prototypes are saying "if I don't have the info, look over here"