IGB190: Game Mechanics Implementation

Data-Driven Game Design
Damage Formulas and Item Creation

> **ⓘ Worksheet Disclaimer**
> The worksheets for the second part of this unit are designed to simulate the experience of working as a **junior in a game development studio**. Each week, you'll be presented with a variety of new tasks, much like you would in a professional environment (though likely not so varied). While the assignments are framed as though you're being given specific responsibilities within a team, remember that **this is your project and you own the work**.

> **Goals**
> This week, you will be **creating and implementing a set of basic items** for your experience (the "legendary" effects will be developed in a future workshop). Additionally, you will be **implementing a custom damage formula** for your game to handle the armor attribute, and **creating developer cheats** to assist you in future testing and mechanic development.

Activity 1: Item Overview

Your studio has just finished the initial implementation of the item system, and needs you to create an **item suite** for the experience. Your team has been asked to create a mix of items with varying types and rarities. To assist you, the studio has created a short summary below explaining basic information about the system - *make sure you read this carefully before you start designing items*.

Items come in one of three rarities: **Common**, **Rare**, and **Legendary**. The rarity of an item determines its drop chance and provides a rough indicator of how "good" an item is. Drop rates can be modified by adjusting the monster properties in the **GameManager** object within the Main scene.

There are five **types** of items that can be equipped. These are **weapons**, **amulets**, **armor**, **boots**, and **rings**.

There is a sixth "**Other**" type which can be used to create non-equippable items. For example, the game may not allow a player to open a door until they have a key in their inventory. You won't be worrying about these this week, but these may become important in future activities.

The player can equip one of each of the item types, except for rings which allow **two** to be equipped at once. Players equip items by dragging them from their inventory onto the appropriate slot on their character.

Purchasing Items from the Marketplace

The **Marketplace** allows players to purchase items for gold. Players can purchase items at any time (the window is opened with the **C** or **I** key), and when purchased, they are immediately added to the player's inventory.

Selling Items at the Marketplace

The player can also drag items they don't want into the "**Sell**" slot to sell an item. The amount of gold the player is given for selling an item is a **ratio of its purchase price**. This can be changed in the game manager.

Item Attributes

Items will modify the attributes/stats of the player. There are many different stats an item may modify, but not all stats should necessarily be available on all item types.

- **Max Health** (e.g., +50 Maximum Health)
- **Max Health Percent** (e.g., +20% Maximum Health)
- **Resource** (e.g., +50 Maximum Rage)
- **Resource Percent** (e.g., (e.g., +20% Maximum Rage)
- **Damage** (e.g., +10 Damage)
- **Damage Percent** (e.g., +20% Damage)
- **Movement Speed Percent** (e.g., +20% Movement Speed)
- **Attack Speed Percent** (e.g., +20% Attack Speed)
- **Critical Strike Chance** (e.g., +10% Critical Strike Chance)
- **Critical Strike Damage** (e.g., +50% Critical Strike Damage)
- **Armor** (e.g., +50 Armor)
- **Resource Cost Reduction** (e.g., -20% Ability Resource Cost)
- **Resource Regeneration** (+20% Rage Generation)
- **Damage Reduction Percent** (e.g., -20% Damage Taken)

Item Custom Logic

The game also allows custom logic to be added to items while they are equipped (e.g., causing health to be restored to the player when they kill enemies with the item equipped).

To not overwhelm the player though, these effects are intended to be reserved for Legendary items.

Activity 2: Create Stat Costs and Item Budgets

Your studio has asked you and your team to create a mix of items with varying types and rarities. Before you start creating these items though, you need to do some design work to determine appropriate **point values** for each stat, and appropriate **stat budgets** for each item type.

1. Open this spreadsheet and select **File > Make a copy**. Then go to the **Stat Budgets** tab. You should then share this document with your team so that you can work on this activity collaboratively.

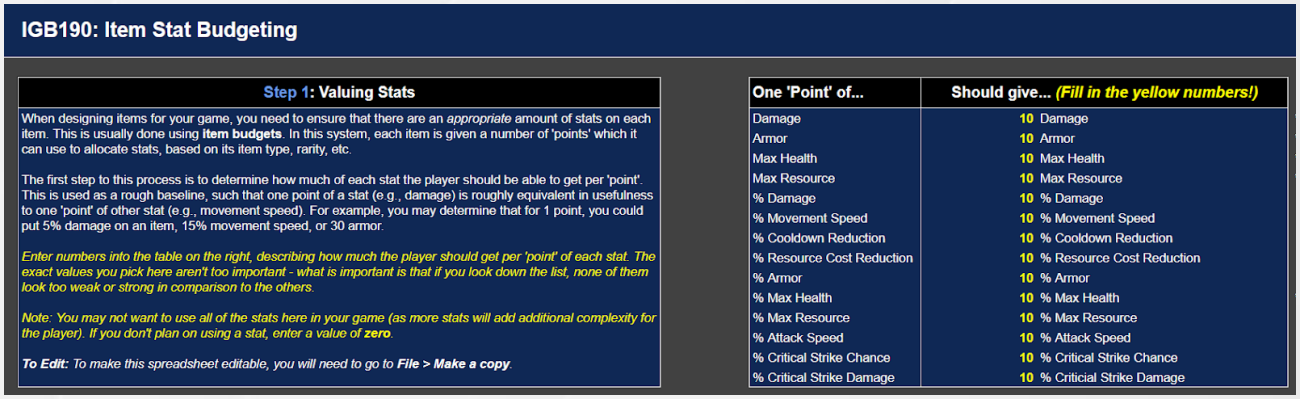
Stat budgets are used to determine an appropriate amount of stats to give an item (though they can also be used more broadly, e.g. to choose appropriate stats for enemies).

A **stat budget** is a single number (e.g. 10), which describes how many total **'points'** worth of stats you can assign to an item. You then determine how much of each stat the player should be able to get per 'point'. For example, you may decide that the player can get **10 health**, **3% damage**, or **5% movement speed** per 'point'.

For example, with these numbers, an item with a **10-point budget** could assign all of them to health and give **+100 Health**, or all of them to damage and give **+30% Damage**. It could also split its budget and give some to each (e.g., +50 Health, +9% Damage, and +10% Movement Speed).

The exact numbers you choose here don't matter too much, as you can adjust how many points are on each item if you want to make the items weaker or stronger later. Instead, you should aim to **ensure that one point of a stat is roughly equivalent to one point of any other stat** (i.e., there shouldn't be any clear "winners" here). For example, if the player could get 50% increased damage with one 'point', or 5% attack speed with one 'point', it would be extremely uncommon for someone to choose the item with attack speed.

2. Read the information for "Step 1", and then decide on how much of each stat you should be able to get for a single 'point'. Remember that all of these should be generally equivalent (though their usefulness to the player will change depending on their build, playstyle etc).



IGB190: Item Stat Budgeting

Step 1: Valuing Stats

When designing items for your game, you need to ensure that there are an *appropriate* amount of stats on each item. This is usually done using **item budgets**. In this system, each item is given a number of 'points' which it can use to allocate stats, based on its item type, rarity, etc.

The first step to this process is to determine how much of each stat the player should be able to get per 'point'. This is used as a rough baseline, such that one point of a stat (e.g., damage) is roughly equivalent in usefulness to one 'point' of other stat (e.g., movement speed). For example, you may determine that for 1 point, you could put 5% damage on an item, 15% movement speed, or 30 armor.

Enter numbers into the table on the right, describing how much the player should get per 'point' of each stat. The exact values you pick here aren't too important - what is important is that if you look down the list, none of them look too weak or strong in comparison to the others.

Note: You may not want to use all of the stats here in your game (as more stats will add additional complexity for the player). If you don't plan on using a stat, enter a value of ***zero****.*

To Edit: *To make this spreadsheet editable, you will need to go to* ***File > Make a copy****.*

One 'Point' of...	Should give... *(Fill in the yellow numbers!)*
Damage	10 Damage
Armor	10 Armor
Max Health	10 Max Health
Max Resource	10 Max Resource
% Damage	10 % Damage
% Movement Speed	10 % Movement Speed
% Cooldown Reduction	10 % Cooldown Reduction
% Resource Cost Reduction	10 % Resource Cost Reduction
% Armor	10 % Armor
% Max Health	10 % Max Health
% Max Resource	10 % Max Resource
% Attack Speed	10 % Attack Speed
% Critical Strike Chance	10 % Critical Strike Chance
% Critical Strike Damage	10 % Criticial Strike Damage

ⓘ Stat Budgets
This also allows for some more advanced systems. For example, a drawback of -100 armor would allow more points to be allocated elsewhere. You may also choose to create particularly powerful legendary effects, but forgo stats entirely on those items.

3. Now scroll down in the spreadsheet and read the information for “Step 2” to enter in an appropriate point budget for each of the armor types and rarities.

Step 2: How many points should each item type and rarity be given?

Now that you know how much of each stat is given per "point", you need to consider how many total points each item type and rarity will be allocated. You will then be able to split those points however you like when designing each items. For example, if a common weapon has 20 points, you may choose to allocate 10 points of the damage stat, 5 points of the % attack speed stat, and 5 points to the Max Health stat.

Discuss with your team and decide on an appropriate point budget value for each item type and rarity below.

Weapon	
Rarity	Point Budget
Common	10
Rare	10
Legendary	10

Armor	
Rarity	Point Budget
Common	10
Rare	10
Legendary	10

Boots	
Rarity	Point Budget
Common	10
Rare	10
Legendary	10

Amulet	
Rarity	Point Budget
Common	10
Rare	10
Legendary	10

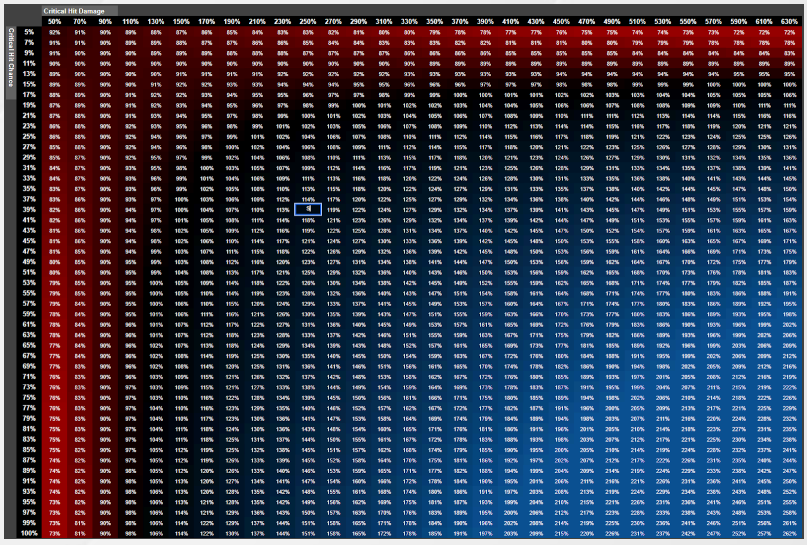
Ring	
Rarity	Point Budget
Common	10
Rare	10
Legendary	10

Understanding Critical Strike

Balancing critical strike chance and critical strike damage with other stats in a game can be very challenging. Unlike many other stats in your game, these stats scale exponentially together. Often, these stats are very poor if a player has not built their character around them, and ***extremely*** strong if they have.

Go to the math spreadsheet you have been using for these activities, and select the **Critical Strike Simulations** tab. This tab shows you the strength of critical strike in your game, based on the point costs you have specified.

This shows you how strong a player would be if they had “spent” their item budget on that much critical strike chance and damage ***instead of*** stacking damage.



Values below 100% **(Red)** indicate that critical strike is worse, while values above 100% **(Blue)** indicate that critical strike is better.

For critical strike to feel “fair” it should ideally be worse than damage at low amounts, and better than damage at high amounts (similar to the below table). If critical strike chance or damage is always better/worse, you may want to reconsider your weightings.

⚠ Critical Strike Imbalances

If your table looks very red or very blue, you should reconsider the values you have chosen for critical strike (or damage percent). You should also ensure that the graph doesn’t look too skewed towards critical strike chance or critical strike damage (i.e. does the left-side or top-side look noticeably redder/worse than the other? If so, one may be overvalued compared to the other.

Note: *If critical strikes have other utility in your game, you may want to also consider the impact they would have. For example, if attacks that critically strike ignore armor, there are additional bonuses not captured here (and critical strike stats should be more “costly”).*

ⓘ Balancing the Stats

Remember that the player can wear six items, and balance accordingly. If an item can give the player +40% movement speed, do you want that to be available on all item slots? Would 340% movement speed feel fun, or awkward?

***Note**: It may take several adjustments here before you reach stat values and item budgets you are happy with for your first implementation.*

⚠ Different Budgets per Stat

It is entirely up to you how you allocate the points on an item to each of the possible stats. You may want to have your experience only increase a single stat, but increase it by *a lot*. You could also have items increase several stats, but make the individual gains much smaller. You also don't need to give an equal amount of points to each stat - a weapon could assign 7 points to damage, and 3 to attack speed.

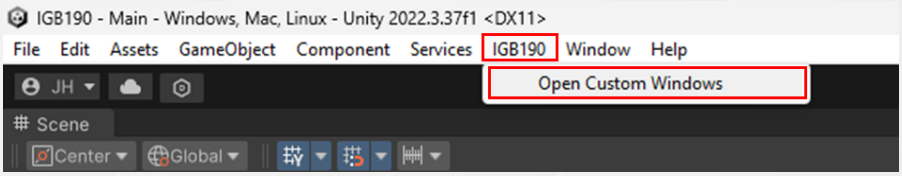
Remember that when you submit your final assignment though, you need to be able to justify why you thought this would be a good decision, given the stated design pillars, and how your testing showed you were reinforcing those pillars.

Activity 3: Create an Item Suite

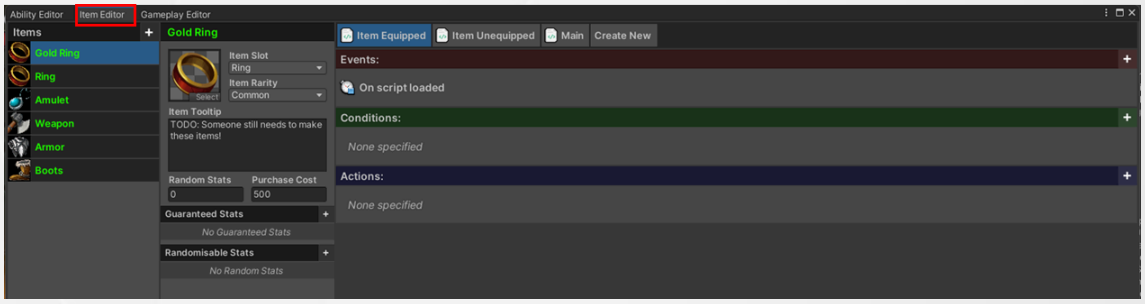
Now that you have a rough understanding of how many stats you should give to each item, you will need to create the initial suite of items. The studio expects you to create at least **two items for each item slot and rarity** (i.e. two common amulets, two rare amulets, two legendary amulets, two common boots, ...). This means that you will need at least **30 total items** (2 Unique Items x 5 Item Slots x 3 Rarities = 30).

You have been asked to ensure that the set of items **makes thematic sense**, is **reasonably well-balanced**, there are **no game-breaking playstyles**, and the items can be **clearly understood by users**.

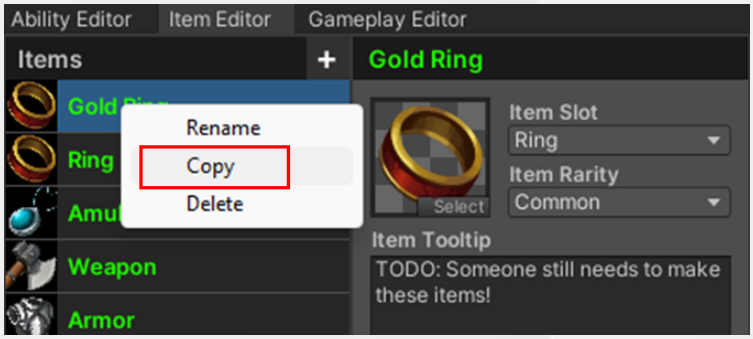
To help you get started, a brief set of instructions on how to create your first item is shown below.

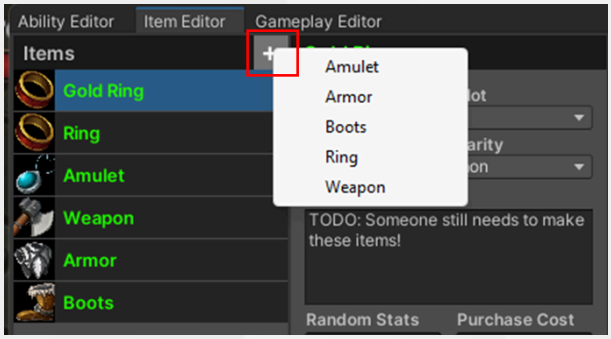


Open the **Item Window** by selecting **IGB190 > Open Custom Windows** and select the **Item Editor** from the top tab.



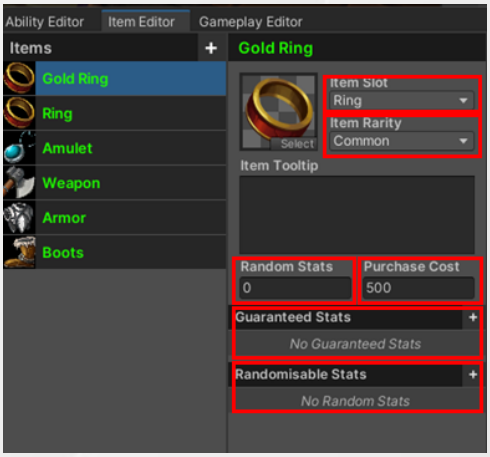
Create a new item by either right-clicking on an existing item and selecting **'Copy'** to create a new item with the same data as an existing item, or press the '**+**' button to create a new item from a basic template.





Select the item you want to modify in the left window, and then modify the attributes of the item using the settings to the right. You can now modify any of the below properties on the item.

- **Item Slot**: The item slot can be **Weapon**, **Amulet**, **Armor**, **Boots**, or **Ring**. This determines which slot the item can be equipped in.
- **Item Rarity**: The item rarity can be **Common**, **Rare**, or **Legendary**. This affects the color of the item and its drop rate in-game.
- **Item Icon**: The item icon can be modified by pressing the 'Select' button in the bottom-left of the icon, and choosing a new image.
- **Item Tooltip**: The item tooltip only changes the text when you hover over the item in-game. It **does not change the functionality** for the item at all. It can be used to describe custom logic you implement using the visual editor (e.g., "*You do 30% more damage while moving.*"), or to add flavor text to items.



- **Purchase Cost**: This determines the cost that the item will have in the marketplace. All items are automatically added to the marketplace.
- **Random Stats**: This determines the number of stats in the 'Randomised Stats' list that will be added to the final item. For example, a "Random Stats" value of **3** will choose three stats from that list.
- **Guaranteed Stats**: This list determines the stats that the final item is **guaranteed to have**. For example, if you want a pair of boots to be guaranteed to have movement speed, it should be included in this list.
- **Randomisable Stats**: This list determines the stats that the final item could randomly have. Only include the random stats here that you want to be possible (e.g., you may not want damage to be a valid random stat on a pair of boots).

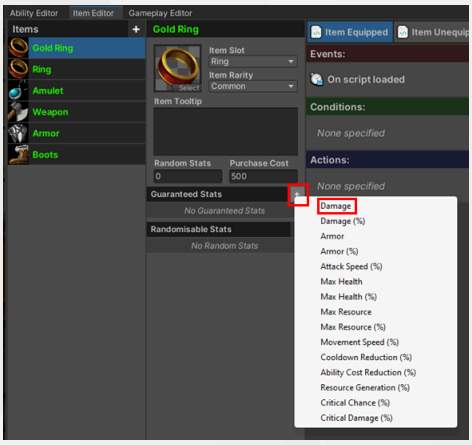
Adding Stats

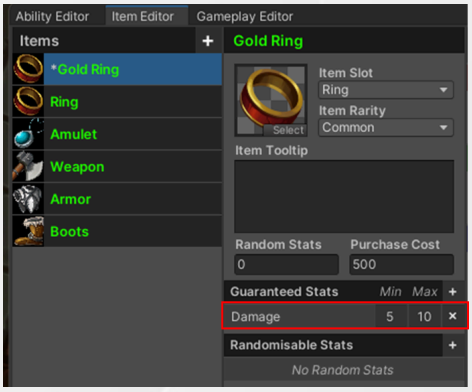
To add random or guaranteed stats to the item, click on the corresponding '**+**' button. The list shown contains all valid stats that an item can have.

When you select one, it will now be added to the list on the item. Each stat has a minimum and maximum value.

When the item is created and the random stats are generated, each stat will be given a value somewhere between these ranges (inclusive). This means that this **Gold Ring** could currently drop with **+5 Damage**, **+6 Damage**, ..., all the way up to **+10 Damage**.

Note: Your values here should be informed by the stat budgets you created earlier (i.e. you should know roughly how much **Max Health** a **Ring** should have).





Removing Stats

Stats can be removed from an item by pressing the '**x**' button on the far right of the stat.

Item Design Advice

When designing items, here are some things you may want to consider:

1. **Item Purchase Costs**: Item purchase costs should scale with stat budgets (i.e. items with more stats should cost more).
2. **Item Stat Count**: Carefully consider how many unique stats each item should have. Is it better to have fewer unique stats on each item, but give more of them? Or vice-versa? What will be fun for the player? What will be confusing?
3. **Rarity Differences**: Will rare and legendary items have more unique stats, have higher stat ranges, or something else?
4. **Stat Availability**: Should certain stats only be available on certain item types? If so, which ones?
5. **Randomness**: How much randomness do you want in the item system? Should all stats have a random range? Should item stats be randomised at all?
6. **Build Diversity**: Given you need at least two items for each type and rarity, what will the two different items do differently? Will one focus on toughness and the other on damage?

ⓘ Making a Unique Experience

To better separate your work from the work done by other groups, you should try to customise the assets you are using. This means that you should try to use different item icons, ability icons, fonts, level components, and other key assets in the experience.

In this activity, you should consider finding unique items. Icons for items and abilities can be sound on the Unity Asset Store, itch.io, Kenny, and many other sources.

For example, https://captaincatsparrow.itch.io/110-free-armor-and-jewelry-icons has a visually matching set of equipment icons:

There are also quite a few free ability icon packs, such as https://captaincatsparrow.itch.io/40-free-barbarian-skill-icons has some barbarian-specific icons (though the same creator has dozens of free icon packs):

Activity 4: Understanding Armor Formulas

A good damage formula in an ARPG is crucial for ensuring balanced gameplay, as it directly impacts how rewarding combat feels and may help to create meaningful player choices in builds and strategies. Your studio has asked you to further develop the damage formula for the player and monsters to consider armor (it currently does nothing!) and other important factors.

In this activity, you will be updating the damage formula in your ARPG to:

1. Consider **armor** in the calculation
2. Consider whether **randomness** should be used in the calculation
3. Consider whether the damage formula will use any **special critical hit logic**
4. Consider whether **stealth help** should be given to the player

For now though, you will just be considering armor. As you likely have very little past experience with armor formulas, overviews of three common armor formulas in games are described below. These are: **flat damage reduction**, **percentage-damage reduction**, and **capped percentage-damage reduction**.

ⓘ Armor Formula #1: Flat Damage Reduction

In a flat damage reduction system, armor reduces the damage dealt by a fixed amount per attack. For example, 10 armor will reduce the damage you take on each attack by 10. If you take a 50 damage hit, or a 1000 damage hit, the reduction will be the same.

The formula for calculating flat damage reduction is quite straightforward:

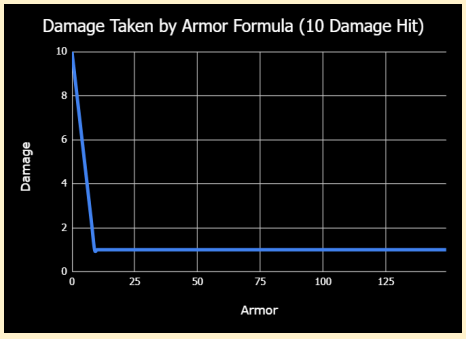
$$\text{Damage Taken} = \text{Attack Power} - \text{Armor Value}$$

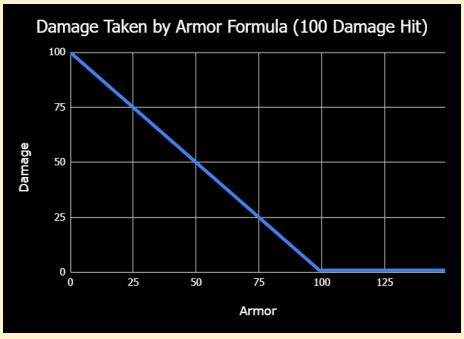
Adding this to the damage formula in your game, the code may look something like this (similar syntax can be used to retrieve the current value for any stat on the unit):

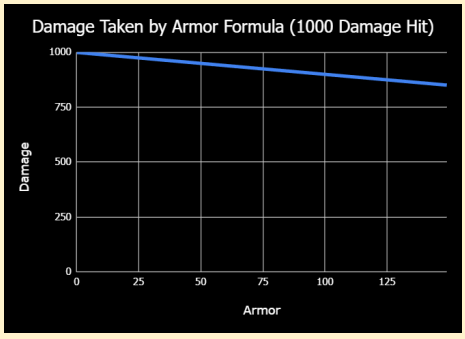
```
amount = amount - stats[Stat.Armor].GetValue();
```

Benefits: This system is easy to implement and easy for players to understand. It is effective against smaller attacks, making low-damage hits feel less impactful while keeping high damage attacks feeling powerful.

Drawbacks: It is harder to balance, because armor has little effect on high-damage attacks (e.g., reducing 10 damage on a 20 damage attack is a 50% reduction, but reducing 10 damage on a 100 damage attack is only a 10% reduction). Notice how armor acts very differently depending on the strength of the attack.







Examples: Diablo II, The Elder Scrolls: Morrowind, Dungeons and Dragons, Darkest Dungeon, Divinity: Original Sin 2, Minecraft, Faster than Light.

ⓘ Armor Formula #2: Percentage-Damage Reduction

In a percentage damage reduction formula, all damage is reduced by the same percentage. This percentage is based on the current armor of the unit (e.g., if a unit has 200 armor, all damage they take may be reduced by 30%).

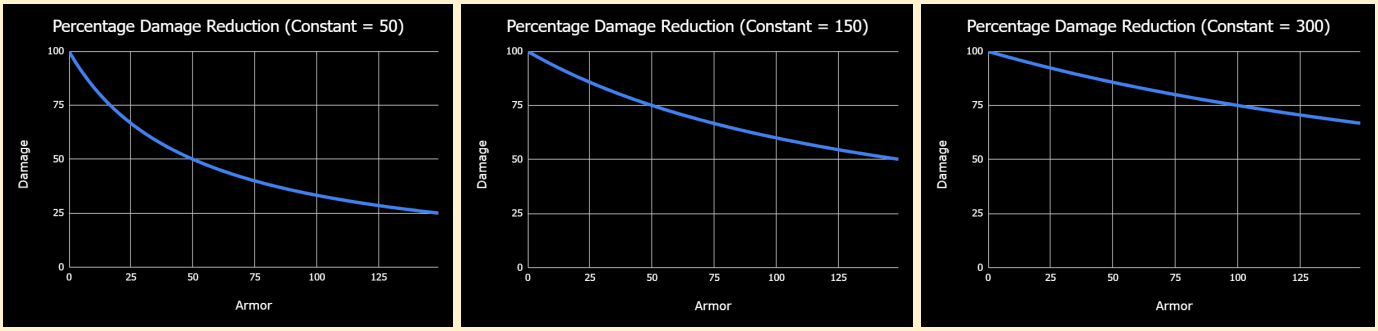
The formula for calculating percentage damage reduction is:

$$\text{Damage Taken} = \text{Attack Power} \times \left(1 - \frac{\text{Armor}}{\text{Armor} + \text{Constant}}\right)$$

When adding this formula to your game, the code may look something like this:

```
float armor = stats[Stat.Armor].GetValue();
float constant = 100f; // Choose your value here.
amount = amount * (1 - armor / (armor + constant));
```

The “Constant” variable can be adjusted to control the “strength” of the damage reduction. The Constant describes the total amount of armor required to achieve 50% damage reduction. This value should always be greater than zero, and increasing this value will make armor less powerful. Here are some example curves using different constants:



Benefits: This system works well across all damage ranges, providing a more consistent experience. It's also easy to balance and more scalable, especially for late-game encounters with high damage values.

Drawbacks: It can be harder for players to understand the exact impact of each additional point of armor, and because armor affects all damage taken equally, it has minimal impact on gameplay decisions.

Examples: League of Legends, The Elder Scrolls V: Skyrim, The Witcher 3: Wild Hunt.

ⓘ Armor Formula #3: Scaled Percentage-Damage Reduction

This formula acts the same as the standard percentage damage reduction formula, but adds an additional scaling factor to determine when the armor should be most valuable. This can make it so that players are encouraged to get at least a small amount of armor, or better reward players that stack a large amount of it.

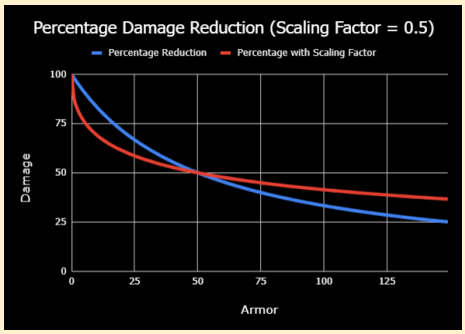
The formula for this is as follows:

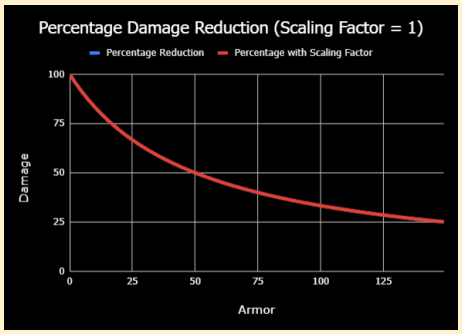
$$\text{Damage Taken} = \text{Attack Power} \times \left(\frac{1}{1 + \left(\frac{\text{Armor}}{\text{Constant}} \right)^{\text{Scaling Factor}}} \right)$$

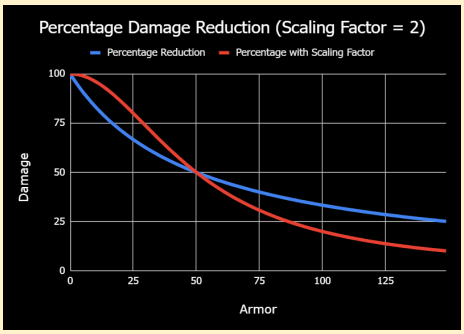
When adding this formula to your game, the code may look like:

```
float armor = stats[Stat.Armor].GetValue();
float constant = 100f; // Chooses your value here.
float scalingFactor = 1.5f; // Choose your value here
amount = amount * (1.0f / (1.0f + Mathf.Pow(armor / constant, scalingFactor)));
```

The scaling factor should always be **greater than zero**. Values between 0 and 1 will make each point of armor more powerful if you don’t have much, and less powerful if you have a lot. A value greater than 1 will make each point of armor less powerful if you don’t have much, and more powerful if you have a lot. Here are some example curves using different scaling factors (note that a scaling factor of one is equivalent to the standard percentage curve):







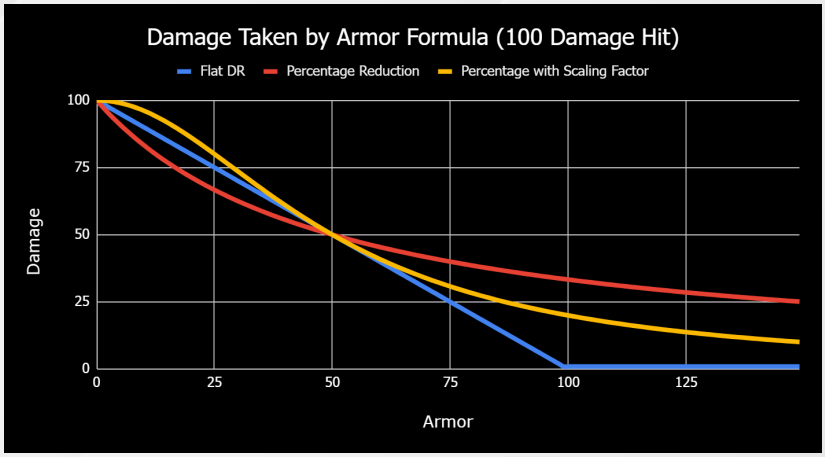
Benefits: The scaling factor offers flexibility in tuning the "diminishing returns" curve. You can make armor fall off more or less sharply depending on your game design needs, allowing for fine-grained control over balance.

Drawbacks: The scaling factor adds complexity to balancing armor. You need to carefully tune the constant and scaling factor to ensure the right balance of effectiveness at low and high armor levels. It can also penalize players that don't realize that they shouldn't take a very small or very large amount of armor (depending on your chosen scaling factor).

Examples: World of Warcraft, Warframe, Guild Wars 2, Black Desert Online.

To better understand these three formulas, you can once again go back to the spreadsheet you used when developing your item budgets in the previous activities. Go to the **Armor Formulas** tab. This tab will simulate how much damage the player would take depending on their armor, using the three formulas above (and the parameters you have set for each).

You should see a graph in the bottom-right highlighting how much damage the player would take with various armor values.



Try changing the **Incoming Damage**, **Constant**, and **Scaling Factor** variables to see how it affects this graph.

1. **Incoming Damage**: This will only affect the Flat DR curve, as the other two formulas always reduce damage taken by the same percentage, regardless of the size of the hit.

2. **Constant**: This value controls how effective the damage reduction is for the two percentage methods. A constant value of 50 means that 50 armor is needed for the player to halve the damage they are taking.

3. **Scaling Factor**: This determines when the armor is most effective. In the above graph, the scaling factor has made armor less effective at low values, but more effective at high values (compared to the standard percentage reduction formula).

Activity 5: Implement the Armor Formula

Determine the armor formula you like the most, and make a note of the values you have chosen for the variables (if you're using one of the percentage-reduction formulas).

Open the **Unit** script in the project, found at **Core > Scripts > Unit** and search for the **ApplyDamageFormula** method. Scroll up to where that formula was explained, and copy the formula into the method to modify the incoming damage amount.

Note: *You can also create a custom formula here. For example, you may want to consider a more unique damage formula, such as having critical strikes completely ignore armor.*

Activity 6: Incorporating Randomness

To make everything feel a bit less deterministic, you can also apply damage ranges to the attacks. This will ensure that when the player hits 30 enemies with an attack, they do not all take the *exact* same amount of damage. This formula usually looks like the following:

$$\text{Damage Taken} = \text{Damage} \times (1 + \text{Random Factor})$$

To add this calculation to your damage formula, go to the **ApplyDamageFormula** method in the Unit script again, and add the following two lines of code (you can choose to apply this before or after the armor modifier):

```
const float randomFactor = 0.05f;
amount *= (1 + Random.Range(-randomFactor, randomFactor));
```

This would cause the damage of an attack to be randomly modified (it could be reduced by up to 5%, or increased by up to 5% for a total 10% variance in possible damage).

> **⚠ Problems with Randomness**
>
> Please note that adding randomness to the damage formula can cause problems in some cases. For example, if you want the player to always be able to determine the exact amount of damage they will deal, players may find this confusing. If the damage numbers are already difficult to calculate though (e.g., due to players being unable to see enemy armor), this becomes a much less serious concern.

Activity 7: Stealth Help

Stealth help aims to assist the player without them ever being aware that it's happening. Stealth help can, for example, have the player take less damage when they're on low health to make the player less likely to die.

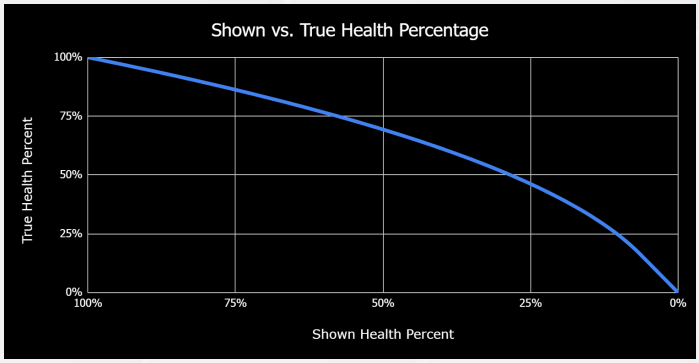
A formula for stealth help which does this could be:

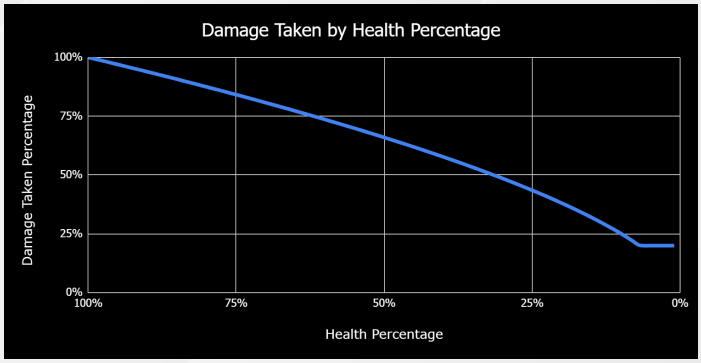
$$\text{Damage Taken} = \text{Damage} \times \max\left(\left(\frac{\text{Current Health}}{\text{Max Health}}\right)^{\text{Modifier}}, \text{Minimum Damage Percentage}\right)$$

In this situation, the **Modifier** value controls how significant the "help" should be. A modifier of **0 will provide no help**, and a modifier of **1 will provide 1% damage reduction per 1% of health lost**. For example, a player that has lost 80% of their health will take 80% less damage.

The **Maximum Damage Reduction** ensures that the player doesn't sit on very low health with significant damage reduction, which can make the help much more obvious to the player. For example, a value of **0.9** here would make it such that the help **cannot make an attack go below 10% of its intended value**.

Go back to the spreadsheet once again, and go to the **Stealth Help** tab. Try entering different values for the player health, the help modifier, and the maximum reduction to see how it affects the overall curve.





After choosing values you feel are appropriate, you will need to implement this in-game. As this help should only be added to the player, you shouldn't add this to the **ApplyDamageFormula** method on the **Unit** (as this would also apply help to all the monsters). Instead, you should add this code to the **TakeDamage method in the Player class** (add it before the current line of code in the method).

The code to implement this is shown below, but you should update the help modifier and max damage reduction percentage to values that are appropriate:

```
const float helpModifier = 1.0f;
const float maxDamageReduction = 0.1f;
float healthPerc = health / stats[Stat.MaxHealth].GetValue();
amount *= Mathf.Max(Mathf.Pow(healthPerc, helpModifier), 1.0f - maxDamageReduction);
```

⚠ Stealth Help
Consider whether stealth help is appropriate for your game, and if so, determine appropriate values for the help modifier and minimum damage percent. Remember that you need to adhere to the four design pillars you have been given.

Activity 5: Building a Developer Cheat Suite

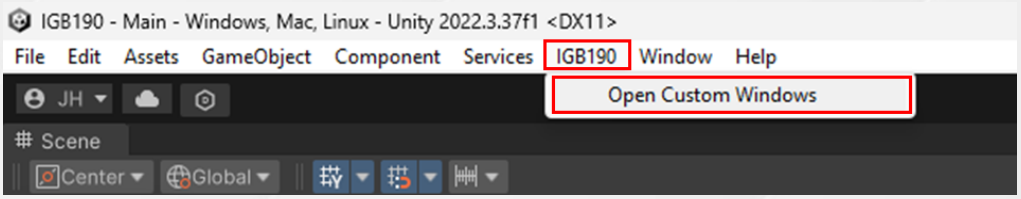
In game development, **developer cheats** are tools that allow you to quickly test various aspects of the game without following normal gameplay. This can allow you to quickly test specific parts of your game without needing to play through everything “properly”.

Your studio has asked you to create the following developer cheats for the game to help facilitate development, bug fixing, and future playtesting:

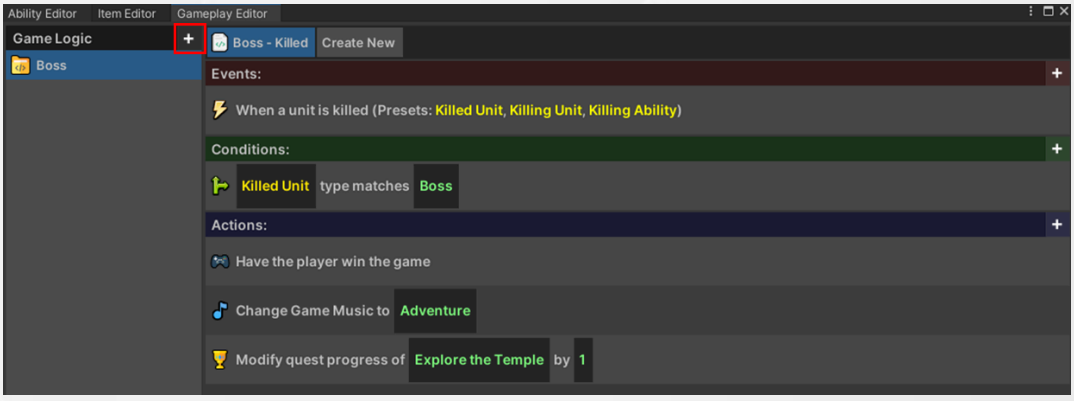
1. Give the player gold.
2. Spawn monsters.
3. Move the player to the boss.
4. Make the player invulnerable.
5. Give the player infinite resources.
6. Kill the player
7. Give the player specific item builds (full uncommon gear, full rare gear, full legendary gear).

As these are only intended to be used internally, they are happy for you to use simple keybindings to implement these actions.

1. To get started, go to **IGB190 > Open Custom Windows** and select the **Gameplay Editor** tab. This editor can be used to create custom logic for your game, such as giving the player quests, handling tutorial actions, and much more.



2. Create a new logic block by pressing the ‘+’ button in the top left of the window, and name the new block **Developer Cheats**. It will contain all of the visual scripts for each cheat.



You should now have an empty logic block, which looks like the following:

You can press the '**Create New**' button to add more scripts to the block, or right click an existing script here to rename or delete it. This will let you, for example, have one script per cheat.

Scripts in this system use an **Event-Condition-Action** system: when any of the events in a script occur, if all of the specified conditions are true, all of the specified actions will be run.Let's begin by creating an example cheat that will give the player gold. Click on the '+' button in the Events header.

1. Select **Input > On Key Down**. This will add the event to the list, but there will be blocks that you can fill in with additional information. In this case, you need to specify which key we want to check for. Click on the block and enter the **G** key. It should look like the following:

In this example, we will make the developer keybind for this key **Control+G**. This means that a condition is also needed to check that the user is holding down the control key.

2. Press the green '+' button in the conditions header, and select **Input > Key is Held**. For the key, type **LeftControl**. The information box below explains the codes for common keys.

ⓘ KeyCodes
The key you need to enter is the Unity KeyCode for the key. You can find a complete list of the KeyCodes here.

Important keys include:
- Common Letters: **A**, **G**, **H** etc.
- Function Keys: **F1**, **F2**, **F3** etc.
- Top Keyboard Numbers: **Alpha1**, **Alpha2**, **Alpha3** etc.
- Keypad Numbers: **Keypad1**, **Keypad2**, **Keypad3** etc.
- Common Modifiers: **LeftControl**, **LeftShift**, **LeftAlt** (and the right equivalents).
- Other Important Keys: **Space**, **Escape**, **Delete**, **Return**, **Enter**, **Backspace**.

3. Finally, you need to specify the action to add gold to the player. Click on the blue '**+**' in the actions header, and select **Player > Add Gold**. For the number, enter **5000**.

4. Save the script by pressing **Ctrl+S**. Logic blocks with unsaved logic will have an asterisk '*****' in front of their name.

✓ Try it out
Press play and test the command. If you press Control+G, the player should gain 5000 gold (the animation will take a while to "count up" this much gold, but you can already spend it in the shop).

ⓘ Debug Logs
If you're creating codes to test something that doesn't have an immediately obvious effect, you can add a console log action. This can be found in **UI > Show Debug Message**.

Try to create the remaining developer cheats on your own using the system. Here are some actions that may help you:

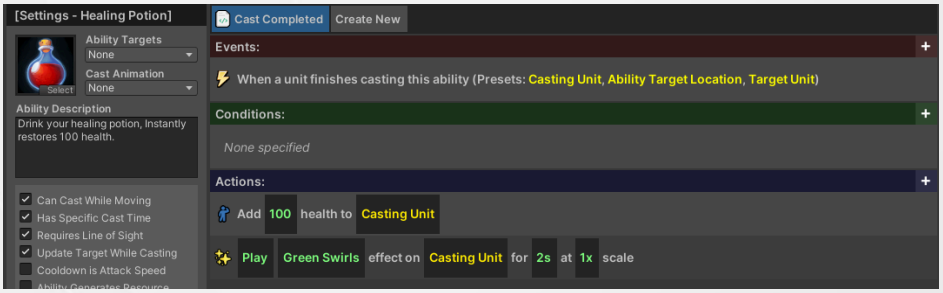
- **Unit > Spawn Monster** (Spawn a monster of the specified type)
- **Unit > Teleport** (Move a unit to the specified location)
- **Unit > Remove Health** (Has the unit lose the given amount of health)
- **Unit > Kill** (Kills the specified unit)
- **Unit > Increase Stat > Increase Stat on Unit by Percent** (e.g. increase damage by 10000%).
- **Player > Add Item** (Adds an item to the player's inventory)
- **Player > Add Experience** (Adds the specified amount of experience to the player)
- **Flow > Wait** (Waits a given amount of time before doing the remaining actions)
- **Flow > For Loop** (Runs the child actions a set number of times - e.g. spawn 50 units).
- **Flow > Disable Script** (Prevent the script from running again - e.g. tutorial actions only run once).

If you get stuck, you should consider looking at some of the abilities. The logic for every ability on each of the three characters was made entirely using this event-condition-action system. You will also be using this in later weeks to create your own abilities.

The Visual Script Editor

This visual scripting editor was built by your studio for the project to assist the team with designing items, abilities, monsters, and general game logic. You will use it next week to design abilities.

A powerful aspect of this system is that the editable parts of each action don't have to be a "hard-coded" value. Events often come with "**Preset**" variables which allow you to easily reference important information. For example, if you look at the Health Potion ability in the Ability Editor tab, it has the following logic:



The “unit finishes casting this ability” event specifies a **casting unit**, a **target unit**, and a **target location** (these are shown in yellow). When describing the ability logic, those options will be available as presents when you try to edit a matching block type.

For example, when specifying the unit that will gain the health, you can enter the “**casting unit**” here.

This approach means that abilities and items are specified using “generic” logic. A player ability (such as the health potion) can be added to the ability list on a monster and the monster will automatically start using it without any further prompting.