

## IMMORTAL PLAYER CHARACTERS

White Paper V1.0.1 Playchemy, Inc. 19th April, 2018

<b>Overview</b>	<b>4</b>
Player Ownership	4
Blockchain Integration	5
Distribution	5
<b>IPC Data</b>	<b>6</b>
IPC ID	6
IPC Age	6
DNA String	6
Attribute String	7
Experience	8
Metadata	8
<b>IPC Marketplace</b>	<b>9</b>
Sell Price	9
Beneficiary	9
ERC-721 Compliance	9
<b>Realms</b>	<b>10</b>
Realm Features	10
Meta-Realms	10
Parallel Play	10
Merchandise	11
Fantasy Realm Example	11
Character Sheet Proof of Concept	12
<b>IPC Developers</b>	<b>13</b>
Interpretation	13
Experiences	13
Developer Verification	13
IPC Ownership Verification	14
<b>Smart Contracts</b>	<b>15</b>
IpcAccessControl	15
IpcReleaseControl	15
ERC165	16
ERC721	16
ERC721Enumerable	16
IpcCreation	16
IpcMarketplace	17



IpcModification	18
IpcExperience	18
ERC721Metadata	19
IpcCore	19
Contract Code	19



# Immortal Player Characters

Immortal Player Characters (IPCs) are characters that can be played in many different games. They are stored on the Ethereum blockchain and improve themselves by gaining experience in games that support the IPC standard.

## Overview

Players spend a lot of time in games improving their player characters. The player accompanies their character on a journey, becomes familiar with its game world, and gains new experiences. This gaining of experience is often quantified in games as “leveling-up” the character, which improves the character in a distinct way. The more time and effort you pour into leveling a character up, the better it gets. As time goes on, players can get very attached to their characters.

All games are self-contained. Characters and environments from one game can't move to another game. Within a single game is all the programming, game rules, graphical content, and characters it will ever use. This means that in any given game, all the time, effort, and money players put into their characters and account can only enrich the characters and account of that particular game.

Eventually, players stop playing certain games. Sometimes it is merely because the story has come to an end. Most game companies stop producing more and more things to do with your character, dooming them to live in a state of limbo. In the case of online games, if the company shuts down, so do the servers, and the characters wink out of existence forever.

How many hours and dollars have been lost improving characters only for them to be abandoned? Players should be able to keep their characters beyond the game where they first met. Characters should be able to play and explore in new games. The time, effort, and money put into a character should be reflected in that character's value.

This is what Immortal Player Characters enable.

## Player Ownership

Until now, the accounts you create, characters you play, and items you earn in video games have been completely and unquestionably owned by the game



company. But with IPCs, the ownership is given back to the players. Ownership is tracked and verified using blockchain technology on the Ethereum network. Once created, an IPC is assigned to a specific Ethereum wallet address, meaning the owner of the wallet is the owner of the IPC.

An owner is free to customize and play with their IPCs in any games that support the IPC standard. Playing with an IPC in games gives it experience points, adding to its value.

At any time the owner may sell or gift his or her IPC to another person. The value added to your character can finally add to the value of the player!

## **Blockchain Integration**

The Ethereum network enables IPCs to become truly immortal. All data stored in the Ethereum network is immutable, which means it cannot be destroyed or changed once written. Instead, all changes or updates to the data are stored as transactions on the blockchain. IPC information is stored publicly in a decentralized smart contract, which is used to enable creation, transfer of ownership (selling, trading, etc), and leveling up of IPCs, as well as facilitating the exporting of IPC data into 3<sup>rd</sup> party games and apps. Because they are stored on the blockchain, these functions ensure IPCs will live on far beyond any person or company's lifetime.

## **Distribution**

The Immortal Player Character Contract distributes IPCs through central issuance, meaning that IPCs are bought from and originate through the contract owners. There will be a set USD price for all newly created IPCs. The reason for doing so is to preserve a standard "buy-in" price for new players. This way, we ensure we don't "price out" potential new players.



# IPC Data

What does it mean to be an IPC? What sets one IPC apart from another? When IPCs are created, they are given an identification number (IPC ID), a time of birth timestamp, a DNA string, an Attribute string, and an experience counter. This is the most basic form an IPC can be.

## IPC ID

As the name suggests, the IPC ID is used to identify an IPC. IPCs are created into one dynamic array, and their IPC ID is the index at which they can be found. The IPC ID of the first IPC (the genesis character) is #1, the next IPC created will have the IPC ID #2, and so on. The IPC ID is important because it is the way players load an IPC into different game worlds. A player will provide games with their IPC's identification number and the game will use that number to verify ownership, then load the IPC from the blockchain.

## IPC Age

At the moment of its creation, each IPC saves its exact time of birth as a timestamp. From then on, the IPC's age can be calculated by subtracting its birth timestamp from the current time. Games can choose to utilize an IPCs age in any way they want. For example, in a fantasy game, an older sorcerer might be able to learn more spells than his or her younger counterpart. The earlier an IPC was created, the older it will be, making it more powerful and valuable as time goes on.

## DNA String

Players experience a game through its player characters. They are what ground the player in a game's world and story. IPCs, however, live outside the context of any specific game. What does it mean to play as a character that can inhabit multiple completely different worlds? How can an IPC that is an Elvish wizard in a fantasy game load into a cartoon game about ponies? What does it mean to be an elf in one world and a pony in another?

IPC data does not contain any physical traits or characteristics. The blockchain doesn't know if the IPC has dark skin or light skin or no skin at all. Instead, each IPC contains a genetic signature; a string of 32 bytes known as its DNA string. The DNA string is used as a reference to manifest any characteristic that a specific game's world requires. Each individual byte represents a physical trait, and its value represents a specific variation of that trait. These traits will be standardized as development finalizes.



DNA string bytes can represent big things such as race or gender, as well as small things such as left or right handedness, eye color, or hair length. An IPC might be born with a rare trait like ambidexterity that automatically makes it quite valuable for certain games.

An example of DNA string interpretation: Say the fourth and fifth byte of the 32-byte DNA string represent skin and hair color, and the byte-values from 0 to 255 represent spectrums of light to dark skin and hair. If an IPC has the DNA values 130 and 255, it would have tan skin and black hair. The reason the blockchain doesn't just store "tan skin" and "black hair" as fixed values is because there could be many games where skin and hair need to be interpreted differently. In a game where the IPC is a pony, these bytes might represent "tan coat" and "black mane" instead.

This is how an IPC can be both an Elf in Middle Earth, and also a pony in My Little Pony World. Each game world has a responsibility to interpret the DNA string in a way that fits its style. This also ensures any given IPC shares common traits with all versions of itself across many worlds.

## Attribute String

How do IPCs differentiate between themselves? In addition to a DNA string that determines appearance, IPCs are born with a 32-byte Attribute string that determines their strengths and weaknesses. Like attributes from a tabletop role-playing game, the attribute string represents a character's level of ability in a variety of ways. Each byte of data in the Attribute seed represents an ability like power, pain tolerance, speed, muscle control, stamina, memory, etc.

Upon creation, the each of these attribute bytes are randomly generated, or "rolled" like a 6-sided die. Two IPCs might have identical dna, but because of the differences in their attributes, they would be distinct from one another. The Attribute string and the DNA string combined make an almost infinite number of possible IPCs. No two would be alike.

Attribute strings can be used by game developers in creative ways to create new attributes from component attribute byte-values. For example, in a tabletop role-playing game, the character's Dexterity could be composed from the bytes corresponding to muscle control, speed, and stamina. Or in a game where an IPC is a doctor, the game developers can create a new attribute called Surgical Skill, composed of memory, simulation, and muscle control.

In some cases, low attributes might correspond with a stronger IPC. For example, in a horror game, the developers might create a Sanity attribute that is the sum of pain tolerance and fortitude, subtracting memory and imagination. In this case, an IPC with low memory and imagination would be better suited for this game. Any attribute string has the potential to be valuable.



This mixing of component attributes allows for an endless number of possible implementations for game developers to explore.

## Experience

What makes a player play games with an IPC? If age, Dna, and Attributes were the only indicators of value, an IPC that has played through a thousand worlds would have the same value as an IPC that was never touched. Because of this, IPCs need a representation of how many experiences they have had; an experience (XP) counter.

When an IPC plays through a game, it levels up and improves in that game world, but it also gains XP to be stored on the blockchain. IPCs with very high XP have seen and done a lot. Like with age, game developers are free to utilize this XP counter any way they want. For example, an assassin IPC with a lot of XP might have a better chance of remaining undetected, or an archer IPC with high XP might have greater accuracy.

IPCs that have experienced more will be more powerful and valuable, incentivizing players to play more games. This means that IPC players will always be on the hunt for more games to play, which makes adopting the IPC standard very tempting for game developers.

## Metadata

IPCs store their metadata in an ERC721 JSON Schema on the immortalplayercharacters.io website. For now, this metadata is composed of the IPC name and an image. This image represents the most recent instantiation of the IPC to have been played. For example, an IPC named Bodo Fraggins plays a fantasy tabletop RPG-style game and his metadata image is of a hobbit. Then if his owner brings him to play in a different game where everyone is a race car, Bodo's metadata image would be of a race car.





# IPC Marketplace

An IPC becomes more powerful the more it experiences and the older it is. An IPC's attributes and DNA might make it better than others in some games. But what does it mean for an IPC to become more valuable? How does its value manifest itself?

## Sell Price

All IPC owners set a sell price for each of their IPCs at which anybody can buy it. Every IPC can be bought or traded. Players cannot make their IPCs unbuyable, they can only set their buy price arbitrarily high. This open market is what bestows all IPCs with value. A better IPC can be listed and sold at a higher price. Any IPC can gain value with time and effort to be sold at a profit.

## Beneficiary

What happens if a player wants to transfer his IPC to a friend for a discount? IPC owners can list a beneficiary address with a special price for each of their IPCs. This is so that an IPC could have a high sell price and also have a discounted price for a friend.

## ERC-721 Compliance

What if a third-party cryptocurrency exchange website wants to list IPCs? This would give IPCs a great deal of exposure.

The IPC smart contract follows and conforms to the ERC-721 standard for non-fungible tokens (NFT), which is a standardized interface for NFT ownership and transfers. The inclusion of this in the IPC contract will allow IPCs to be listed on third-party exchange platforms and third-party NFT wallets.



# Realms

So far players can create, buy and sell IPCs. The next step is to instantiate them into a realm. Realms are game worlds compliant with the IPC standard. For example, World of Warcraft, Dungeons and Dragons, and Pokemon could all have their own IPC realms. Instantiating an IPC into a realm means to interpret its DNA into a set of features consistent with that world. IPCs can be instantiated into every existing realm. Realms may exist as standalone games, or be created as smart contracts that publicly store specific IPC info.

## Realm Features

Before they enter their first realm, IPCs are composed of just their age, DNA, attributes, and experience. But inside a game world, an IPC gains many new things, all specified by that realm's developer. An IPC gains a corporeal form dictated by that realm's interpretation of its DNA string and gains attributes dictated by that realm's interpretation of its attribute string. It might also gain an inventory of equipment and loot, a character level, a list of skills, abilities, and spells, a friends list... The sky's the limit when adding features to a realm. These features are non-transferrable between realms and are called realm-specific. For example, if a realm were to use a form of currency to buy things only redeemable in that realm, it would be a realm-specific currency.

## Meta-Realms

There may be realm smart contracts that aren't tied to any specific game or universe, but instead be a meta-realm that executes an *idea* that ties multiple realms together. For example, there may be a realm that has permanent death (permadeath). This means that once an IPC dies in this realm, it may no longer be played in that realm. The information of how many times an IPC has died can be stored in a meta-realm smart contract, so that all games can know whether an IPC has died in any permadeath games. A game could, for example, give the IPC's avatar a scar for each permadeath he has experienced.

## Parallel Play

One aspect of using on-chain realms is that they are not bound to any single game. Realms exist outside of a game, so that multiple games may use the features of one realm. The rewards earned from one game might be stored into that game's realm, and accessible by other games that also utilize that realm. This allows for "parallel play" where loot obtained in one game automatically show up in your inventory for another game. A game doesn't have to be popular to bestow value to an IPC. And a new game can piggy-back off a more



established game by allowing players use their progress in the new game.

Parallel play could enable modders that don't have the expertise, time, or money to make a fully featured game to create side-worlds that load data from other games and provide extra content in the form of more bosses or new areas.

## Merchandise

Once an IPC is instantiated into a realm, it has a definite form. Because the IPC is, at its core, a collectible, owners may want to purchase items that represent their specific IPC. Initially, collectible cards and 3D printed miniatures of IPCs will be available for purchase through the [ImmortalPlayerCharacters.io](http://ImmortalPlayerCharacters.io) website.

Realm creators will be responsible for creating the IPC avatars that will appear as the IPC's metadata image.

## Fantasy Realm Example

For example, a developer might create a Fantasy Realm. A player provides his IPC ID to load its properties and discovers that his IPC is a male dwarf. Because he is one of the oldest IPCs, he has long white hair and a thick braided beard. The dwarf IPC discovers the skills Mining, Axe Proficiency, Shield Proficiency, and has the attributes Strength, Dexterity, and Intelligence. The realm uses his Attribute string to calculate his maximum hit-points and magic-points. He starts with an empty inventory, except for 100 copper coins, a realm-specific currency. Because the IPC has 100 XP, he can upgrade one of his skills right from the start.


Now that there is a realm, there need to be games that utilize it. In this example, the realm developer might create a Dungeons and Dragons-style game, where players can load their IPC's Fantasy Realm avatar and attributes into a multiplayer tabletop campaign. They let the player keep all their earned campaign loot and experience by storing it in the Fantasy Realm's inventory. Each campaign may also provide characters with campaign-specific items, which do not register inside the realm's inventory. At the end of a campaign, the IPCs that participated each get an experience point.

A second developer creates a separate game that is a player-vs-player (PvP) battle tournament. The tournament game reads from the Fantasy Realm's IPC inventories and allows players to "buy in" using either 1000 copper or the equivalent value in loot. The winner gets to keep all the loot and the money. The Fantasy Realm attributes are loaded, as well as any relevant skills and spells, and the Fantasy Realm IPCs are matched up and battle each other. The mechanics of the battles are entirely decided by the developer.



## Character Sheet Proof of Concept

What does an IPC's DNA and Attributes look like when instantiated into a realm? The following is an example character sheet of a fantasy game realm. The DNA is interpreted into race, gender, etc, and attributes are interpreted into strength, dexterity, etc.

IPC #201																																									
<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 15%;">Name</td> <td>Lathlaeril</td> </tr> <tr> <td>Birth</td> <td>3/22/2018 3:41 PM</td> </tr> <tr> <td>XP</td> <td>0</td> </tr> <tr> <td>Price</td> <td>\$1,000,000.00</td> </tr> </table>	Name	Lathlaeril	Birth	3/22/2018 3:41 PM	XP	0	Price	\$1,000,000.00																																	
Name	Lathlaeril																																								
Birth	3/22/2018 3:41 PM																																								
XP	0																																								
Price	\$1,000,000.00																																								
<table style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th colspan="2" style="text-align: center;">Attributes</th> </tr> </thead> <tbody> <tr> <td style="width: 10%;">8</td> <td style="width: 10%;">STR</td> <td style="width: 5%; text-align: center;">{</td> <td style="width: 75%;"> <ul style="list-style-type: none"> <li>1 Force</li> <li>2 Sustain</li> <li>5 Tolerance</li> </ul> </td> </tr> <tr> <td>8</td> <td>DEX</td> <td style="text-align: center;">{</td> <td> <ul style="list-style-type: none"> <li>1 Speed</li> <li>6 Precision</li> <li>1 Reaction</li> </ul> </td> </tr> <tr> <td>11</td> <td>INT</td> <td style="text-align: center;">{</td> <td> <ul style="list-style-type: none"> <li>2 Memory</li> <li>4 Processing</li> <li>5 Reasoning</li> </ul> </td> </tr> <tr> <td>15</td> <td>CON</td> <td style="text-align: center;">{</td> <td> <ul style="list-style-type: none"> <li>6 Healing</li> <li>3 Fortitude</li> <li>6 Vitality</li> </ul> </td> </tr> <tr> <td>12</td> <td>LCK</td> <td></td> <td></td> </tr> </tbody> </table>	Attributes		8	STR	{	<ul style="list-style-type: none"> <li>1 Force</li> <li>2 Sustain</li> <li>5 Tolerance</li> </ul>	8	DEX	{	<ul style="list-style-type: none"> <li>1 Speed</li> <li>6 Precision</li> <li>1 Reaction</li> </ul>	11	INT	{	<ul style="list-style-type: none"> <li>2 Memory</li> <li>4 Processing</li> <li>5 Reasoning</li> </ul>	15	CON	{	<ul style="list-style-type: none"> <li>6 Healing</li> <li>3 Fortitude</li> <li>6 Vitality</li> </ul>	12	LCK			<table style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th colspan="2" style="text-align: center;">DNA</th> </tr> </thead> <tbody> <tr> <td style="width: 15%;">Race</td> <td>Elf</td> </tr> <tr> <td>Subrace</td> <td>Wood Elf</td> </tr> <tr> <td>Gender</td> <td>Male</td> </tr> <tr> <td>Height</td> <td>6'3"</td> </tr> <tr> <td>Skin Color</td> <td>Pale</td> </tr> <tr> <td>Hair Color</td> <td>Light Brown</td> </tr> <tr> <td>Eye Color</td> <td>Orange</td> </tr> <tr> <td>Handedness</td> <td>Right</td> </tr> </tbody> </table>	DNA		Race	Elf	Subrace	Wood Elf	Gender	Male	Height	6'3"	Skin Color	Pale	Hair Color	Light Brown	Eye Color	Orange	Handedness	Right
Attributes																																									
8	STR	{	<ul style="list-style-type: none"> <li>1 Force</li> <li>2 Sustain</li> <li>5 Tolerance</li> </ul>																																						
8	DEX	{	<ul style="list-style-type: none"> <li>1 Speed</li> <li>6 Precision</li> <li>1 Reaction</li> </ul>																																						
11	INT	{	<ul style="list-style-type: none"> <li>2 Memory</li> <li>4 Processing</li> <li>5 Reasoning</li> </ul>																																						
15	CON	{	<ul style="list-style-type: none"> <li>6 Healing</li> <li>3 Fortitude</li> <li>6 Vitality</li> </ul>																																						
12	LCK																																								
DNA																																									
Race	Elf																																								
Subrace	Wood Elf																																								
Gender	Male																																								
Height	6'3"																																								
Skin Color	Pale																																								
Hair Color	Light Brown																																								
Eye Color	Orange																																								
Handedness	Right																																								



# IPC Developers

When a player buys an IPC, their first thought is “what can I play?”. When there are thousands of IPC owners all looking for games to play, the few games that are available draw a lot of attention. Discoverability is something that a lot of game developers struggle with, and that is what the IPC standard can provide. So what does a developer need to do to have their game available to IPC owners?

## Interpretation

It is meaningless for an IPC to be playing in a game but that game not utilize any interpretation of the IPC’s DNA or attributes. If an IPC becomes Luke Skywalker no matter what, then it doesn’t matter if the IPC is tall or short, or strong or weak. Games that are available to IPCs need to reinforce a player’s character by making it recognizable. Even something as simple as making the in-game character’s hair color derived from the IPC’s DNA can be enough to make a character distinct and recognizable as a specific IPC.

## Experiences

Because the IPC contract is public, any developer can instantiate an IPC into their game and have it be playable by its owner. But for this play to have meaning, it needs to be documentable in the IPC contract. Having earnable experiences provides a record of that IPC’s play in a developer’s game. Developers that provide more experiences that are easier to achieve immediately make their game more desirable to new players. However, because the logging of XP is done on the blockchain, there is an inherent cost to gifting experiences. This creates a natural balance between how much developers are willing to spend on acquiring new players and how many experiences they can create.

## Developer Verification

The experience-gifting system has the potential for exploitation, so there is a verification process for a game developer to become a registered developer for IPCs. The verification process checks that the game developer has an interpretation system in place, and requires the developer to update the IPC’s metadata image automatically when the IPC is loaded into the game. After a developer is verified, he or she can create any number of experiences on the IPC contract for players to earn. Experiences must follow some rules: Unobtainable experiences are not allowed, experiences must be awarded on the contract if they are earned in the game, and vice versa, experiences may not be



awarded to IPCs that have not earned the experience in the game. Any violation of these rules and the developer will lose their IPC developer status.

## **IPC Ownership Verification**

It is important that IPCs are only playable by their owners, so developers that want to use IPCs need a way to verify IPC ownership. Playchemy Inc will provide a login service for players to verify their owned IPCs, and will eventually develop a metamask-style plugin for Unity to read the blockchain. IPC developers will need to either use these systems or utilize something similar in order to make sure players don't have access to IPCs they don't own.



# Smart Contracts

Immortal Player Character blockchain functionality is divided into interfaces and contract segments in a chain of inheritance.

## IpcAccessControl

The IpcAccessControl contract segment is derived from the Ownable contract by OpenZeppelin. IpcGod replaces Ownable's Owner, and three more address types are defined. IPC functions are divided into 5 levels of access: ipcGod, ipcExec, ipcAdmin, ipcDeveloper, and Public.

IpcGod is able to execute all functions and change the addresses associated with all levels of access. IpcGod, like Owner, can relinquish its access to another address using the relinquishGodhood function.

IpcExec is the address with the next-highest clearance. IpcExec can make top-level changes such as pricing and changing tranche variables, as well as releasing new tranches. IpcExec is in charge of the vetting and supervision of IpcAdmins and IpcDevelopers.

IpcAdmins are stored in a list of addresses. More admins can be created by ipcExec or ipcGod, and likewise, admin access can be revoked. Admins have access to remote IPC creation and modification. This is so that off-chain users can still have access to the world of IPCs.

IpcDevelopers are stored in an (address=>bool) mapping; an address is either a registered developer or it is not. IpcDevelopers have access to the functions outlined in the IpcExperience contract segment; Experience creation and gifting.

Every function not restricted to the above access levels is publicly accessible to anyone.

## IpcReleaseControl

Inherits IpcAccessControl

IPCs are released in batches called tranches. The IpcReleaseControl contract segment manages new IPC creation by keeping track of the number and price



of the IPCs released per tranche, as well as dictating when the next tranche gets released.

## **ERC165**

ERC165 is an interface used to query whether the IPC contract supports the interface and, if yes, which version of the interface. This allows an external source to adapt the way in which the IPC contract is interfaced. This is so future changes to an interface standard does not necessarily require the IPC contract to make any changes.

## **ERC721**

The ERC721 interface follows and conforms to the ERC-721 standard for non-fungible tokens (NFT). It is a standardized interface for NFT ownership and transfers. The inclusion of this contract segment will allow IPCs to be listed on third-party exchange platforms and third-party NFT wallets.

## **ERC721Enumerable**

The ERC721Enumerable interface allows the tokens outlined in the ERC721 interface to be counted and queried. This is not included for tokens that require privacy, but IPCs are public.

## **IpcCreation**

Inherits IpcReleaseControl, ERC165, ERC721, and ERC721Enumerable.

The IpcCreation contract segment defines all ownership data structures and handles all IPC creation. It is also where the Immortal Player Character itself is defined. Because this is the first segment that contains payable functions, the MarketPrice dollar to ether conversion interface is defined here.

USD to ETH conversion is handled by an external contract which is updated every 10 minutes to reflect the latest USD to ETH conversion rate. All prices defined in the IPC contract is in USD.





The `Ipc` struct is the essence of an IPC. It contains a name, a 32-byte string of dna, a 32-byte `attributeSeed`, an experience counter, and a time of birth integer.

Randomized creation of an IPC costs the least, and IPCs created with this function have their `attributeSeeds` and DNA randomized.

IPC seed creation gives players IPCs which have their `timeOfBirth`, `attributeSeed`, and `dna` member variables set to 0. These seeds cannot be used in many ipc-related protocols.

IPC Seeds have the option of being “substantiated” with a custom dna string. This can be done before or after attributes are generated. Attributes are always randomized. This is to allow a player to have his dna inspired by his attributes.

Random number generation is done by concatenating the sender’s address, the block’s timestamp, and a nonce that is incremented each time the random number generator is called. The concatenated result is run through the keccak256 hashing algorithm. This process will generate a pseudo-random number that is impossible to determine ahead of time.

IPCs are pushed into an array of all existing IPCs. A separate (`uint=>address`) `ipcToOwner` mapping is used to store IPC ownership. In order to look up a specific address’s owned IPCs is done off-chain by iterating through the entire list of IPCs and checking the address against their owners. This is done in a view-restricted `tokensOfOwner(_address)` function. Calculating this off-chain greatly reduces the gas cost of transferring ownership.

## IpcMarketplace

Inherits `IpcCreation`.

The `IpcMarketplace` contract segment stores the IPC ownership transfer functions as well as the owner-chosen pricing information for each IPC. Transfers can be done in three different ways: `transfer`, `approval / takeOwnership`, and `buyIpc`.

The ERC-721 standard requires a `transfer(address _to, uint _tokenId)` function, as well as an `approval / takeOwnership` system which protects against IPCs getting sent to incorrect addresses. The preferred way of transferring ownership is the `buyIpc(uint _ipclId, uint _newPrice)` function which takes the IPC from its owner without need for approval, as long as enough ether is sent with the function call.

IPC pricing information is stored in a `IpcMarketInfo` struct. The struct contains one address where a beneficiary may be set with a special beneficiary price.



This is so that transfers between friends or accounts can be done safely without the need for the ERC-721 transfer functions. The struct also contains an approval address which approves a single address to take ownership of the IPC for nothing in return. This is included in order to be ERC-721 compliant.

Players can set the prices and special addresses for their IPCs using public function calls, or by requesting these done by an admin.

## IpcModification

Inherits IpcMarketplace.

The IpcModification contract segment handles the modification of IPC names and their DNA. These can be changed by IPC owners through a public function call or by an admin. Changing DNA requires a price be paid for each increment or decrement of a DNA byte.

An example of DNA modification:

0xf141b2c55ca16dc00019d3feb6527f4b22edf0a139f66313dd2fff0b1ea0c768

The sixth byte in the dna string might represent skin tone with a value of 0x00 being the fairest and 0xFF being the darkest. The sixth byte in the example dna string is underlined as a1, or 161 in decimal. If the owner of this IPC wanted his skin tone to be as light as possible, he would have to pay 161 times the price to modify DNA to decrement the dna byte 161 times.

## IpcExperience

Inherits IpcModification.

IpcExperience is the contract segment where IpcDeveloper functions are defined. This layer includes experience creation, the purchasing of XP, and functions to grant XP to IPCs.

Experiences are goals that developers set for their players to try and achieve. Developers register these experiences to the IPC contract by function call with a string description of the experience. These experiences can be awarded to an IPC as XP, increasing that IPC's XP counter, which affect the IPC in realm-specific ways.

XP is a currency-type ownable by developers with the single purpose of granting to IPCs. Developers may grant XP as rewards for in-game achievements. Each XP gained by an IPC represents one experience from a developer. An IPC may only be granted one XP per experience.



To grant XP, developers must buy XP from the IPC contract. The IpcExperience counter tracks each developer's xpBalance.

The purpose of granting XP is for a developer to gain discoverability by rewarding players that try their game. This is counteracted by each XP having a cost associated with it, so developers can't endlessly grant XP. IPC owners are incentivized to keep trying new IPC-integrated games, which benefits both the players and developers.

## ERC721Metadata

The Metadata interface allows external web3 sources to query token name and image through a standardized ERC721 URI format. This allows 3rd party dapps to be able to display the correct IPC image and name next to the owned IPC.

## IpcCore

Inherits IpcExperience, and ERC721Metadata.

The IpcCore contract segment is the final layer of the IPC contract, and provides the IPC interface for which external contracts or dapps can read IPC data. There is a single public function `getIpc(uint _ipcid)` which returns all the data inside the IPC's struct.

This layer contains the constructor function for the contract, which sets the `ipcGod` address and the `MarketPlace` external contract address. The first tranche is initialized here.

This layer also provides a pointer for an updated IPC contract if ever the IPC contract needs to hard fork.

## Contract Code

```
pragma solidity ^0.4.19;
```

```
//-----  
/// @title IPC Access Control  
/// @dev defines access modifiers for God, Exec, Admin, and Developer access
```



```

pragma solidity ^0.4.19;

//-----
/// @title IPC Access Control
/// @dev defines access modifiers for God, Exec, Admin, and Developer access
/// defines functions to assign access levels to addresses. To avoid conflicts
/// of interest, an address with any of these levels of access cannot own IPCs.
//-----
contract IpcAccessControl {
    address ipcGod; // Assigns Exec, Admin, and Cashier addresses
    address ipcExec; // Manages top-level variable modification
    address ipcCashier; // Address to where money is sent
    mapping (uint=>address) indexToAdmin; // list of admin-level access
    mapping (address=>bool) ipcDeveloper; // developer-level access
    uint totalAdmins;
    uint public totalDevelopers;
    bool locked; // protects against re-entrancy attacks

    modifier onlyIpcGod() {
        require (msg.sender == ipcGod);
        _;
    }

    modifier onlyExecOrHigher() {
        require (msg.sender == ipcExec || msg.sender == ipcGod);
        _;
    }

    modifier onlyAdminOrHigher() {
        require (_checkIfAdmin(msg.sender) || msg.sender == ipcExec || msg.sender ==
ipcGod);
        _;
    }

    modifier onlyDeveloper() {
        require(ipcDeveloper[msg.sender]);
        _;
    }

    modifier noHigherAccess(address _address) {
        require(_checkIfAdmin(_address) == false &&
_address != ipcExec &&

```



```

    _address != ipcGod);
    _;
}

// protects payable functions against re-entrancy attacks
modifier noReentrancy() {
    require(!locked);
    locked = true;
    _;
    locked = false;
}

//-----
// FUNCTIONS - high to low clearance
//-----
function renounceGodhood(address _newGod) external onlyIpcGod {
    ipcGod = _newGod;
}

function setExec(address _newExec) external onlyIpcGod {
    ipcExec = _newExec;
}

function setCashier(address _newCashier) external onlyIpcGod {
    ipcCashier = _newCashier;
}

function addAdmin(address _newAdmin) external onlyExecOrHigher {
    indexToAdmin[totalAdmins] = _newAdmin;
    totalAdmins++;
}

function removeAdmin(address _adminToRemove) external onlyExecOrHigher {
    for (uint i = 0; i < totalAdmins; ++i) {
        if (indexToAdmin[i] == _adminToRemove) {
            if (i != totalAdmins - 1) {
                address swapAddress = indexToAdmin[totalAdmins - 1];
                indexToAdmin[i] = swapAddress;
            }
            totalAdmins--;
        }
    }
}

```



```

}

function getAllPositions() external view onlyExecOrHigher returns (address[]) {
    address[] memory positions = new address[](totalAdmins + 3);
    positions[0] = ipcGod;
    positions[1] = ipcExec;
    positions[2] = ipcCashier;
    for (uint i = 3; i < positions.length; ++i) {
        positions[i] = indexToAdmin[i];
    }
    return positions;
}

function getAdmins() external view onlyAdminOrHigher returns (address[]) {
    address[] memory admins = new address[](totalAdmins);
    for (uint i = 0; i < totalAdmins; ++i) {
        admins[i] = indexToAdmin[i];
    }
    return admins;
}

function changeDeveloperStatus(address developer, bool value) external
onlyExecOrHigher {
    require(ipcDeveloper[developer] != value);
    if (value == true) {
        totalDevelopers++;
    } else {
        totalDevelopers--;
    }
    ipcDeveloper[developer] = value;
}

// withdraws money somehow stuck in the contract
function withdraw() external onlyAdminOrHigher {
    ipcCashier.transfer(address(this).balance);
}

function _checkIfAdmin(address _address) internal view returns(bool) {
    for(uint i = 0; i < totalAdmins; ++i){
        if (_address == indexToAdmin[i]) {
            return true;
        }
    }
}

```



```

    }
    return false;
  }
}

//-----
/// @title IPC Release Control
/// @dev Keeps track of the number and price of the IPCs released per tranche
/// and dictates when the next tranche gets released
//-----
contract IpcReleaseControl is IpcAccessControl {

    bool autoTrancheRelease = true; // whether tranches release by themselves
    uint128 public totalIpcs; // all ipcs in existence
    uint trancheSize = 1000; // number of IPCs to be created per release
    uint ipcCap; // equal to previous ipcCap + trancheSize
    uint priceIncreasePerTrancheInCents = 1; // how much more each IPC costs per
tranche
    uint public ipcPriceInCents = 25;

    function changeTrancheSize(uint _newSize) external onlyExecOrHigher {
        require (_newSize > 0);
        trancheSize = _newSize;
    }

    function changePriceIncreasePerTranche(uint _newPriceIncrease) external
onlyExecOrHigher {
        priceIncreasePerTrancheInCents = _newPriceIncrease;
    }

    function releaseNewTranche() public {
        if(autoTrancheRelease == false) {
            require (msg.sender == ipcExec || msg.sender == ipcGod);
        }
        require(totalIpcs >= ipcCap);
        ipcCap += trancheSize;
        ipcPriceInCents += priceIncreasePerTrancheInCents;
    }

    function setAutoTrancheRelease(bool value) external onlyExecOrHigher {
        autoTrancheRelease = value;
    }
}

```



```

    }
}

//-----
/// @title ERC-165 Standard Interface Detection
/// @dev see https://github.com/ethereum/EIPs/blob/master/EIPS/eip-165.md
/// Note: the ERC-165 identifier for this interface is 0x01ffc9a7
//-----
interface ERC165 {
    /// @notice Query if a contract implements an interface
    /// @param _interfaceId The interface identifier, as specified in ERC-165
    /// @dev Interface identification is specified in ERC-165. This function
    /// uses less than 30,000 gas.
    /// @return `true` if the contract implements `interfaceId` and
    /// `interfaceId` is not 0xffffffff, `false` otherwise
    function supportsInterface(bytes4 _interfaceId) external view returns (bool);
}

//-----
/// @title ERC-721 Non-Fungible Token Standard
/// @dev See https://github.com/ethereum/EIPs/blob/master/EIPS/eip-721.md
/// Note: the ERC-165 identifier for this interface is 0x6466353c
//-----
interface ERC721 {
    // Events
    /// @dev This emits when ownership of any NFT changes by any mechanism.
    /// This event emits when NFTs are created (`from` == 0) and destroyed
    /// (`to` == 0). Exception: during contract creation, any number of NFTs
    /// may be created and assigned without emitting Transfer. At the time of
    /// any transfer, the approved address for that NFT (if any) is reset to none.
    event Transfer(address indexed _from, address indexed _to, uint256 indexed
    _tokenId);

    /// @dev This emits when the approved address for an NFT is changed or
    /// reaffirmed. The zero address indicates there is no approved address.
    /// When a Transfer event emits, this also indicates that the approved
    /// address for that NFT (if any) is reset to none.
    event Approval(address indexed _owner, address indexed _approved, uint256 indexed
    _tokenId);
}

```





```

    /// @dev This emits when an operator is enabled or disabled for an owner.
    /// The operator can manage all NFTs of the owner.
    event ApprovalForAll(address indexed _owner, address indexed _operator, bool
_approved);

    /// @notice Count all NFTs assigned to an owner
    /// @dev NFTs assigned to the zero address are considered invalid, and this
    /// function throws for queries about the zero address.
    /// @param _owner An address for whom to query the balance
    /// @return The number of NFTs owned by `_owner`, possibly zero
    function balanceOf(address _owner) external view returns (uint256 balance);

    /// @notice Find the owner of an NFT
    /// @param _tokenId The identifier for an NFT
    /// @dev NFTs assigned to zero address are considered invalid, and queries
    /// about them do throw.
    /// @return The address of the owner of the NFT
    function ownerOf(uint256 _tokenId) external view returns (address owner);

    /// @notice Transfers the ownership of an NFT from one address to another address
    /// @dev Throws unless `msg.sender` is the current owner, an authorized
    /// operator, or the approved address for this NFT. Throws if `_from` is
    /// not the current owner. Throws if `_to` is the zero address. Throws if
    /// `_tokenId` is not a valid NFT. When transfer is complete, this function
    /// checks if `_to` is a smart contract (code size > 0). If so, it calls
    /// `onERC721Received` on `_to` and throws if the return value is not
    /// `bytes4(keccak256("onERC721Received(address,uint256,bytes)"))`.
    /// @param _from The current owner of the NFT
    /// @param _to The new owner
    /// @param _tokenId The NFT to transfer
    /// @param data Additional data with no specified format, sent in call to `_to`
    function safeTransferFrom(address _from, address _to, uint256 _tokenId, bytes
data) external payable;

    /// @notice Transfers the ownership of an NFT from one address to another address
    /// @dev This works identically to the other function with an extra data
parameter,
    /// except this function just sets data to []
    /// @param _from The current owner of the NFT
    /// @param _to The new owner
    /// @param _tokenId The NFT to transfer
    function safeTransferFrom(address _from, address _to, uint256 _tokenId) external

```



payable;

```

/// @notice Transfer ownership of an NFT -- THE CALLER IS RESPONSIBLE
/// TO CONFIRM THAT `_to` IS CAPABLE OF RECEIVING NFTS OR ELSE
/// THEY MAY BE PERMANENTLY LOST
/// @dev Throws unless `msg.sender` is the current owner, an authorized
/// operator, or the approved address for this NFT. Throws if `_from` is
/// not the current owner. Throws if `_to` is the zero address. Throws if
/// `_tokenId` is not a valid NFT.
/// @param _from The current owner of the NFT
/// @param _to The new owner
/// @param _tokenId The NFT to transfer
function transferFrom(address _from, address _to, uint256 _tokenId) external

```

payable;

```

/// @notice Set or reaffirm the approved address for an NFT
/// @dev The zero address indicates there is no approved address.
/// @dev Throws unless `msg.sender` is the current NFT owner, or an authorized
/// operator of the current owner.
/// @param _approved The new approved NFT controller
/// @param _tokenId The NFT to approve
function approve(address _approved, uint256 _tokenId) external payable;

```

```

/// @notice Enable or disable approval for a third party ("operator") to manage
/// all your asset.
/// @dev Emits the ApprovalForAll event
/// @param _operator Address to add to the set of authorized operators.
/// @param _approved True if the operators is approved, false to revoke approval
function setApprovalForAll(address _operator, bool _approved) external;

```

```

/// @notice Get the approved address for a single NFT
/// @dev Throws if `_tokenId` is not a valid NFT
/// @param _tokenId The NFT to find the approved address for
/// @return The approved address for this NFT, or the zero address if there is
none

```

```
function getApproved(uint256 _tokenId) external view returns (address);
```

```

/// @notice Query if an address is an authorized operator for another address
/// @param _owner The address that owns the NFTs
/// @param _operator The address that acts on behalf of the owner
/// @return True if `_operator` is an approved operator for `_owner`, false
otherwise

```



```

function isApprovedForAll(address _owner, address _operator) external view returns
(bool);
}

//-----
/// @title ERC-721 Non-Fungible Token Standard, optional enumeration extension
/// @dev See https://github.com/ethereum/EIPs/blob/master/EIPS/eip-721.md
/// Note: the ERC-165 identifier for this interface is 0x780e9d63
//-----
interface ERC721Enumerable /* is ERC721 */ {
    /// @notice Count NFTs tracked by this contract
    /// @return A count of valid NFTs tracked by this contract, where each one of
    /// them has an assigned and queryable owner not equal to the zero address
    function totalSupply() external view returns (uint256);

    /// @notice Enumerate valid NFTs
    /// @dev Throws if `_index` >= `totalSupply()`.
    /// @param _index A counter less than `totalSupply()`
    /// @return The token identifier for the `_index`th NFT,
    /// (sort order not specified)
    function tokenByIndex(uint256 _index) external view returns (uint256);

    /// @notice Enumerate NFTs assigned to an owner
    /// @dev throws if __owner is the zero address
    /// @param _owner An address to query for owned NFTs
    /// @return The token identifiers for all NFTs assigned to _owner in order of
creation
    function tokensOfOwner(address _owner) external view returns (uint256[]);

    /// @notice Get the token Id of the '_index'th NFT assigned to an owner
    /// @dev Throws if `_index` >= `balanceOf(_owner)` or if
    /// `_owner` is the zero address, representing invalid NFTs.
    /// @param _owner An address where we are interested in NFTs owned by them
    /// @param _index A counter less than `balanceOf(_owner)`
    /// @return The token identifier for the `_index`th NFT assigned to `_owner`
    function tokenOfOwnerByIndex(address _owner, uint256 _index) external view returns
(uint256 _tokenId);
}

/// @title Metadata extension to ERC-721 interface

```



```

/// @dev See https://github.com/ethereum/EIPs/blob/master/EIPS/eip-721.md
/// Note: the ERC-165 identifier for this interface is 0x5b5e139f
interface ERC721Metadata {

    /// @dev ERC-165 (draft) interface signature for ERC721
    // bytes4 internal constant INTERFACE_SIGNATURE_ERC721Metadata = // 0x2a786f11
    //     bytes4(keccak256('name()')) ^
    //     bytes4(keccak256('symbol()')) ^
    //     bytes4(keccak256('deedUri(uint256)'));

    /// @notice A descriptive name for a collection of deeds managed by this
    /// contract
    /// @dev Wallets and exchanges MAY display this to the end user.
    function name() external pure returns (string _name);

    /// @notice An abbreviated name for deeds managed by this contract
    /// @dev Wallets and exchanges MAY display this to the end user.
    function symbol() external pure returns (string _symbol);

    /// @notice A distinct URI (RFC 3986) for a given token.
    /// @dev If:
    /// * The URI is a URL
    /// * The URL is accessible
    /// * The URL points to a valid JSON file format (ECMA-404 2nd ed.)
    /// * The JSON base element is an object
    /// then these names of the base element SHALL have special meaning:
    /// * "name": A string identifying the item to which `_tokenId` grants
    /// ownership
    /// * "description": A string detailing the item to which `_tokenId` grants
    /// ownership
    /// * "image": A URI pointing to a file of image/* mime type representing
    /// the item to which `_tokenId` grants ownership
    /// Wallets and exchanges MAY display this to the end user.
    /// Consider making any images at a width between 320 and 1080 pixels and
    /// aspect ratio between 1.91:1 and 4:5 inclusive.
    function tokenURI(uint256 _tokenId) external view returns (string);
}

//-----
/// @title Coin Market Price Smart Contract
/// @dev see https://github.com/hunterlong/marketprice/blob/master/README.md

```



```

/// Updates every 2 hours
//-----
interface MarketPrice {
    function ETH(uint _id) external constant returns (uint256);
    function USD(uint _id) external constant returns (uint256);
    function EUR(uint _id) external constant returns (uint256);
    function GBP(uint _id) external constant returns (uint256);
    function updatedAt(uint _id) external constant returns (uint);
}

//-----
/// @title IPC Creation
/// @dev Defines all ownership data structures and handles all IPC creation
/// defines ERC-721 Enumerable's ownership queries
//-----
contract IpcCreation is IpcReleaseControl, ERC165, ERC721, ERC721Enumerable {

    /// @dev This emits when an IPC is created by any mechanism. Exception:
    /// during contract creation in the event of an update, existing IPCs
    /// will be created and assigned without emitting Created.
    event Created(uint tokenId, address indexed owner, string name);

    /// @dev This emits when an IPC is fully substantiated with dna, attributes,
    /// and a time of birth.
    event Substantiated(uint tokenId, bytes32 dna, bytes32 attributes);

    /// @dev This emits when an IPC's dna is modified
    event DnaModified(uint indexed tokenId, bytes32 to);

    // currency converter
    MarketPrice priceConverter;

    // contains all IPC information
    struct Ipc {
        string name;
        bytes32 attributeSeed;
        bytes32 dna;
        uint128 experience;
        uint128 timeOfBirth;
    }
}

```



```

    Ipc[] public Ipcs; // array of IPCs
    mapping (uint => address) public ipcToOwner; // IPC to owner address
    mapping (address => uint) public ownerIpcCount; // how many IPCs an address
owns
    mapping (uint => uint) ipcSeedToCustomizationPrice; // the price to customize an
IPC seed
    mapping (uint => bool) ipcToAdminAuthorization; // whether or not an admin
can modify IPC

    uint public customizationPriceMultiplier = 4;
    uint nonce = 0;

    modifier onlyOwnerOrAdmin(uint _costInCents, uint _ipcId) {
        uint costInWei = _convertCentsToWei(_costInCents);
        require (
            msg.sender == ipcToOwner[_ipcId] && msg.value >= costInWei ||
            (ipcToAdminAuthorization[_ipcId] &&
            (_checkIfAdmin(msg.sender) ||
            msg.sender == ipcExec ||
            msg.sender == ipcGod))
        );
        _;
    }

    // forward declaration
    function setIpcPrice(uint _ipcId, uint _newPrice) public onlyOwnerOrAdmin(0,
_ipcId);

    //-----
    // IPC ADMIN FUNCTIONS
    //-----

    function changeCustomizationMultiplier(uint _newMultiplier) external
onlyExecOrHigher {
        customizationPriceMultiplier = _newMultiplier;
    }

    function updateMarketPriceContract(address _newAddress) external onlyExecOrHigher
{
        priceConverter = MarketPrice(_newAddress);
    }

```



```

function createAndAssignRandomizedIpc(
    string _name,
    uint _price,
    address _owner
) external onlyAdminOrHigher noHigherAccess(_owner) {
    require(bytes(_name).length <= 32 && totalIpcs < ipcCap);
    _makeIpc(_price, _owner, _name, _generateRandomNumber(),
_generateRandomNumber(), uint128(now));
    emit Substantiated(totalIpcs, Ipcs[totalIpcs - 1].dna, Ipcs[totalIpcs -
1].attributeSeed);
    if (totalIpcs >= ipcCap && autoTrancheRelease) {
        releaseNewTranche();
    }
}

function createAndAssignIpcSeed(
    string _name,
    uint _price,
    address _owner
) external onlyAdminOrHigher noHigherAccess(_owner) {
    require(bytes(_name).length <= 32 && totalIpcs < ipcCap);
    _makeIpc(_price, _owner, _name, 0, 0, 0);
    ipcSeedToCustomizationPrice[totalIpcs] = ipcPriceInCents *
customizationPriceMultiplier;
    if (totalIpcs >= ipcCap && autoTrancheRelease) {
        releaseNewTranche();
    }
}

//-----
// USER FUNCTIONS
//-----

/// @notice Create a fully substantiated IPC with randomized attributes and dna
/// @dev Throws if name is longer than 32 bytes. Throws if msg.value is too low.
/// Throws if msg.sender is an admin, ipcExec, or ipcGod.
/// @param _name Name to assign to the IPC. The longer the name, the more gas
needed
/// @param _price Initial buy price for the IPC. Price calculated in USD cents.
function createRandomizedIpc(
    string _name,
    uint _price

```



```

) external payable noReentrancy noHigherAccess(msg.sender) {
    require(bytes(_name).length <= 32 && totalIpcs < ipcCap);
    uint ipcPriceInWei = _convertCentsToWei(ipcPriceInCents);
    require (msg.value >= ipcPriceInWei);
    _makeIpc(_price, msg.sender, _name, _generateRandomNumber(),
_generateRandomNumber(), uint128(now));
    emit Substantiated(totalIpcs, Ipcs[totalIpcs - 1].dna, Ipcs[totalIpcs -
1].attributeSeed);
    msg.sender.transfer(msg.value - ipcPriceInWei);
    ipcCashier.transfer(ipcPriceInWei);
    if (totalIpcs >= ipcCap && autoTrancheRelease) {
        releaseNewTranche();
    }
}

/// @notice Create an unsubstantiated IPC Seed with no attributes or dna
/// @dev Throws if name is longer than 32 bytes. Throws if msg.value is too low.
/// Throws if msg.sender is an admin, ipcExec, or ipcGod.
/// @param _name Name to assign to the IPC. The longer the name, the more gas
needed
/// @param _price Initial buy price for the IPC. Price calculated in USD cents.
function createIpcSeed(
    string _name,
    uint _price
) external payable noReentrancy noHigherAccess(msg.sender) {
    require(bytes(_name).length <= 32 && totalIpcs < ipcCap);
    uint ipcPriceInWei = _convertCentsToWei(ipcPriceInCents);
    require (msg.value >= ipcPriceInWei);
    _makeIpc(_price, msg.sender, _name, 0, 0, 0);
    ipcSeedToCustomizationPrice[totalIpcs] = ipcPriceInCents *
customizationPriceMultiplier;
    msg.sender.transfer(msg.value - ipcPriceInWei);
    ipcCashier.transfer(ipcPriceInWei);
    if (totalIpcs >= ipcCap && autoTrancheRelease) {
        releaseNewTranche();
    }
}

/// @notice Rolls attributes on an IPC Seed. Does not substantiate.
/// @dev Throws unless `msg.sender` is the current owner or an authorized IPC
administrator.
/// Throws if attributes were already rolled.

```





```

/// @param _ipcId IPC Identifier to roll attributes
function rollAttributes(uint _ipcId) external onlyOwnerOrAdmin(0, _ipcId) {
    Ipc storage myIpc = Ipcs[_ipcId - 1];
    require (myIpc.attributeSeed == 0);    // can only roll attributes once
    myIpc.attributeSeed = _generateRandomNumber();
}

/// @notice Rolls custom dna on an IPC Seed. If attributes not rolled, roll
attributes. Substantiates.
/// @dev Throws unless `msg.sender` is the current owner or an authorized IPC
administrator.
/// Throws if msg.value is too low. Throws if IPC is already substantiated.
/// @param _ipcId IPC Identifier for IPC to customize DNA
/// @param _dna A custom bytes32
function customizeDna(
    uint _ipcId,
    bytes32 _dna
) public payable noReentrancy
onlyOwnerOrAdmin(ipcSeedToCustomizationPrice[_ipcId], _ipcId) {
    Ipc storage myIpc = Ipcs[_ipcId - 1];
    require (myIpc.timeOfBirth == 0);
    myIpc.timeOfBirth = uint128(now);
    myIpc.dna = _dna;
    if (myIpc.attributeSeed == 0) {
        myIpc.attributeSeed = _generateRandomNumber();
    }
    emit Substantiated(totalIpcs, Ipcs[totalIpcs - 1].dna, Ipcs[totalIpcs -
1].attributeSeed);
    msg.sender.transfer(msg.value - ipcSeedToCustomizationPrice[_ipcId]);
    ipcCashier.transfer(ipcSeedToCustomizationPrice[_ipcId]);
}

/// @notice Rolls randomized dna on an IPC Seed. If attributes not rolled,
/// rolls attributes. Substantiates.
/// @dev Throws unless `msg.sender` is the current owner or an authorized IPC
administrator.
/// Throws if IPC is already substantiated.
/// @param _ipcId IPC Identifier for IPC to randomize DNA
function randomizeDna(uint _ipcId) external onlyOwnerOrAdmin(0, _ipcId) {
    Ipc storage myIpc = Ipcs[_ipcId - 1];
    require (myIpc.timeOfBirth == 0);
    myIpc.timeOfBirth = uint128(now);

```



```

myIpc.dna = _generateRandomNumber();
if (myIpc.attributeSeed == 0) {
    myIpc.attributeSeed = _generateRandomNumber();
}
emit Substantiated(totalIpcs, Ipcs[totalIpcs - 1].dna, Ipcs[totalIpcs -
1].attributeSeed);
}

/// @notice Changes whether or not an admin is authorized to make changes to an
IPC
function changeAdminAuthorization(uint _ipcId, bool _authorization) external
onlyOwnerOrAdmin(0, _ipcId) {
    ipcToAdminAuthorization[_ipcId] = _authorization;
}

//-----
// ERC-721 FUNCTIONS
//-----
function totalSupply() external view returns (uint) {
    return totalIpcs;
}

function tokenByIndex(uint _index) external view returns (uint) {
    require (_index < totalIpcs);
    return _index + 1; // IPC ID is ALWAYS index + 1
}

function balanceOf(address _owner) external view returns (uint) {
    require (_owner != 0);
    return ownerIpcCount[_owner];
}

function tokensOfOwner(address _owner) external view returns (uint[]) {
    require(ownerIpcCount[_owner] > 0);
    uint counter = 0;
    uint[] memory result = new uint[](ownerIpcCount[_owner]);
    for (uint i = 1; i <= Ipcs.length; i++) {
        if(ipcToOwner[i] == _owner) {
            result[counter] = i;
            counter++;
        }
    }
}

```



```

    return result;
}

function tokenOfOwnerByIndex(address _owner, uint _index) external view returns
(uint) {
    require (_index <= ownerIpcCount[_owner]);
    uint counter = 0;
    for (uint i = 0; i < Ipcs.length; i++) {
        if (ipcToOwner[i] == _owner) {
            if (counter == _index) {
                return i;
            } else {
                counter++;
            }
        }
    }
}

function ownerOf(uint _tokenId) external view returns (address owner) {
    owner = ipcToOwner[_tokenId];
}

//-----
// INTERNAL FUNCTIONS
//-----

function _generateRandomNumber() internal returns (bytes32) {
    nonce++;
    return keccak256(now, msg.sender, nonce);
}

function _makeIpc(
    uint _price,
    address _owner,
    string _name,
    bytes32 _dna,
    bytes32 _attributeSeed,
    uint128 _timeOfBirth
) internal {
    uint id = Ipcs.push(Ipc(_name, _attributeSeed, _dna, 0, _timeOfBirth));
    ipcToOwner[id] = _owner;
    ownerIpcCount[_owner]++;
    ipcToAdminAuthorization[id] = true; // default admin access
}

```



```

    setIpcPrice(id, _price);
    emit Created(id, _owner, _name);
    emit Transfer(0, _owner, id); // send the Transfer event
    totalIpcs++;
}

function _convertCentsToWei(uint centsAmount) internal view returns(uint) {
    uint ethCent = priceConverter.USD(0); // $0.01 worth of wei
    return (ethCent * centsAmount); // centsAmount worth of wei
}
}

/// @dev Note: the ERC-165 identifier for this interface is 0xf0b9e5ba
interface ERC721TokenReceiver {
    /// @notice Handle the receipt of an NFT
    /// @dev The ERC721 smart contract calls this function on the recipient
    /// after a `transfer`. This function MAY throw to revert and reject the
    /// transfer. This function MUST use 50,000 gas or less. Return of other
    /// than the magic value MUST result in the transaction being reverted.
    /// Note: the contract address is always the message sender.
    /// @param _from The sending address
    /// @param _tokenId The NFT identifier which is being transferred
    /// @param data Additional data with no specified format
    /// @return `bytes4(keccak256("onERC721Received(address,uint256,bytes)"))`
    /// unless throwing
    function onERC721Received(address _from, uint256 _tokenId, bytes data) external
returns(bytes4);
}

//-----
/// @title IPC Marketplace
/// @dev Defines ownership transfer functions and owner-chosen pricing information.
/// Transfers can be done in three different ways: safeTransferFrom, transferFrom,
/// and buyIpc.
//-----
contract IpcMarketplace is IpcCreation {

    /// @dev Emits whenever an IPC is bought using the buyIpc function
    event Bought(uint indexed _tokenId, address _seller, address indexed _buyer, uint
price);

```



```

    /// @dev Emits whenever the price of an IPC changes. Does not emit for beneficiary
    price change
    event PriceChanged(uint indexed _tokenId, uint from, uint to);

    struct IpcMarketInfo {
        uint32 sellPrice;
        uint32 beneficiaryPrice;
        address beneficiaryAddress;
        address approvalAddress;    // used for ERC721-required approval and
takeOwnership functions
    }

    mapping (uint => IpcMarketInfo) public ipcToMarketInfo;
    mapping (address => mapping (address => bool)) ownerToOperator;
    uint public maxIpcPrice = 100000000;    // 1 million

    function setMaxIpcPrice(uint _newPrice) external onlyExecOrHigher {
        maxIpcPrice = _newPrice;
    }

    /// @notice Change the sell price of an owned IPC
    /// @dev throws unless msg.sender is the owner of the IPC or an IPC administrator
    /// @param _ipcId The IPC Identifier for the IPC whose price is changing
    /// @param _newPrice The new price for the IPC
    function setIpcPrice(uint _ipcId, uint _newPrice) public onlyOwnerOrAdmin(0,
_ipcId) {
        uint from = ipcToMarketInfo[_ipcId].sellPrice;
        if (_newPrice > maxIpcPrice) {
            _newPrice = maxIpcPrice;
        }
        ipcToMarketInfo[_ipcId].sellPrice = uint32(_newPrice);
        emit PriceChanged(_ipcId, from, _newPrice);
    }

    /// @notice Gives a beneficiary address a discounted or inflated price.
    /// @dev There may only be one beneficiary at a time. Throws unless msg.sender
    /// is the owner of the IPC or an approved IPC administrator.
    /// @param _ipcId The IPC Identifier for the IPC to approve a beneficiary
    /// @param _beneficiaryAddress The beneficiary's wallet address
    /// @param _beneficiaryPrice The special price for the beneficiary
    function setSpecialPriceForAddress(

```



```

    uint _ipcId,
    address _beneficiaryAddress,
    uint _beneficiaryPrice
) external onlyOwnerOrAdmin(0, _ipcId) {
    ipcToMarketInfo[_ipcId].beneficiaryPrice = uint32(_beneficiaryPrice);
    ipcToMarketInfo[_ipcId].beneficiaryAddress = _beneficiaryAddress;
}

/// @notice Obtains ownership of an ipc. Must send at least the buyout price.
/// @dev Throws unless _ipcId is valid. Throws if msg.value is too low.
/// Buying sets beneficiaryAddress to 0 address. Emits Transfer event. Emits
/// Bought event.
/// @param _ipcId The IPC Identifier for the IPC to be bought
/// @param _newPrice The new price of the IPC once bought
function buyIpc(uint _ipcId, uint _newPrice) public payable noReentrancy {
    require(_ipcId > 0 && _ipcId <= totalIpcs);
    IpcMarketInfo storage ipcToBuy = ipcToMarketInfo[_ipcId];
    uint priceInCents;
    uint priceInWei;
    if (msg.sender == ipcToBuy.beneficiaryAddress) {
        priceInCents = ipcToBuy.beneficiaryPrice;
    } else {
        priceInCents = ipcToBuy.sellPrice;
    }
    priceInWei = _convertCentsToWei(priceInCents);
    require (msg.value >= priceInWei);
    address seller = ipcToOwner[_ipcId];
    _transferOwnership(msg.sender, seller, _ipcId);
    emit Bought(_ipcId, seller, msg.sender, priceInCents); // send buy event
    ipcToBuy.sellPrice = uint32(_newPrice);
    msg.sender.transfer(msg.value - priceInWei); // send the excess value back
    seller.transfer(priceInWei); // send the rest to the seller
}

//-----
// ERC721-required transfer functions
//-----

function safeTransferFrom(
    address _from,
    address _to,
    uint256 _tokenId,

```



```

    bytes data
  ) external payable noHigherAccess(_to) {
    require (
      msg.sender == ipcToOwner[_tokenId] || // IPC owner
      msg.sender == ipcToMarketInfo[_tokenId].approvalAddress || // Approved
address
      ownerToOperator[ipcToOwner[_tokenId]][msg.sender] == true // Approved
operator
    );
    require (_tokenId != 0 && _tokenId <= totalIpcs);
    require (_from == ipcToOwner[_tokenId]);
    require (_to != 0);
    if (msg.sender == ipcToMarketInfo[_tokenId].approvalAddress) {
      require (_to == msg.sender);
    }
    _transferOwnership(_to, _from, _tokenId);
    if (_isContract(_to)) {
      ERC721TokenReceiver tokenReceiver = ERC721TokenReceiver(_to);
      bytes4 returnValue = tokenReceiver.onERC721Received(_from, _tokenId,
data);
      require (returnValue ==
bytes4(keccak256("onERC721Received(address,uint256,bytes)")));
    }
  }

function safeTransferFrom(
  address _from,
  address _to,
  uint256 _tokenId
) external payable noHigherAccess(_to) {
  require (
    msg.sender == ipcToOwner[_tokenId] || // IPC owner
    msg.sender == ipcToMarketInfo[_tokenId].approvalAddress || // Approved
address
    ownerToOperator[ipcToOwner[_tokenId]][msg.sender] == true // Approved
operator
  );
  require (_tokenId != 0 && _tokenId <= totalIpcs);
  require (_from == ipcToOwner[_tokenId]);
  require (_to != 0);
  if (msg.sender == ipcToMarketInfo[_tokenId].approvalAddress) {
    require (_to == msg.sender);
  }
}

```



```

    }
    bytes memory data;
    _transferOwnership(_to, _from, _tokenId);
    if (_isContract(_to)) {
        ERC721TokenReceiver tokenReceiver = ERC721TokenReceiver(_to);
        bytes4 returnValue = tokenReceiver.onERC721Received(_from, _tokenId,
data);
        require (returnValue ==
bytes4(keccak256("onERC721Received(address,uint256,bytes)")));
    }
}

function transferFrom(
    address _from,
    address _to,
    uint256 _tokenId
) external payable noHigherAccess(_to) {
    require (
        msg.sender == ipcToOwner[_tokenId] || // IPC owner
        msg.sender == ipcToMarketInfo[_tokenId].approvalAddress || // Approved
address
        ownerToOperator[ipcToOwner[_tokenId]][msg.sender] == true // Approved
operator
    );
    require (_tokenId != 0 && _tokenId <= totalIpcs);
    require (_from == ipcToOwner[_tokenId]);
    require (_to != 0);
    if (msg.sender == ipcToMarketInfo[_tokenId].approvalAddress) {
        require (_to == msg.sender);
    }
    _transferOwnership(_to, _from, _tokenId);
}

function approve(address _to, uint256 _tokenId) external payable {
    require (_tokenId <= totalIpcs && msg.sender == ipcToOwner[_tokenId]);
    ipcToMarketInfo[_tokenId].approvalAddress = _to;
    emit Approval(msg.sender, _to, _tokenId); // send the Approval event
}

function setApprovalForAll(address _operator, bool _approved) external {
    ownerToOperator[msg.sender][_operator] = _approved;
    emit ApprovalForAll(msg.sender, _operator, _approved); // send the

```





ApprovalForAll event

```

    }

    function getApproved(uint256 _tokenId) external view returns (address) {
        require(_tokenId != 0 && _tokenId <= totalIpcs);
        return ipcToMarketInfo[_tokenId].approvalAddress;
    }

    function isApprovedForAll(address _owner, address _operator) external view returns
(bool) {
        return ownerToOperator[_owner][_operator];
    }

    function _transferOwnership(address _to, address _from, uint256 _tokenId) internal
{
        ownerIpcCount[_from]--;          // remove IPC from seller's list of owned
        ipcToOwner[_tokenId] = _to;      // change owner to buyer
        ipcToMarketInfo[_tokenId].beneficiaryAddress = 0; // remove any beneficiary
        if(ipcToMarketInfo[_tokenId].approvalAddress != 0) {
            emit Approval(msg.sender, 0, _tokenId);
            ipcToMarketInfo[_tokenId].approvalAddress = 0; // remove any pending
approval
        }
        ownerIpcCount[_to]++;           // add IPC to buyer's list of owned
        emit Transfer(_from, _to, _tokenId); // send the Transfer event
    }

    function _isContract(address addr) private view returns (bool) {
        uint size;
        assembly { size := extcodesize(addr) }
        return size > 0;
    }
}

//-----
/// @title IPC Modification
/// @dev Handles modification of IPC names and DNA
//-----
contract IpcModification is IpcMarketplace {

    /// @dev Emits whenever the name of an IPC is changed. Emits on IPC creation.

```



```

event NameChanged(uint indexed _tokenId, string _to);

uint public priceToModifyDna = 100;
uint public priceToChangeName = 100;
uint public dnaModificationLevelRequirement = 1000000;
uint public nameModificationLevelRequirement = 1;

modifier levelRequirement(uint _req, uint _ipcId) {
    require(Ipcs[_ipcId - 1].experience >= _req);
    _;
}

function changeDnaModificationLevelRequirement(uint _newReq) external
onlyExecOrHigher {
    dnaModificationLevelRequirement = _newReq;
}

function changeNameModificationLevelRequirement(uint _newReq) external
onlyExecOrHigher {
    nameModificationLevelRequirement = _newReq;
}

function changePriceToModifyDna(uint _newPrice) public onlyExecOrHigher {
    priceToModifyDna = _newPrice;
}

/// @notice Changes the name of an IPC
/// @dev Throws unless msg.sender is the IPC's owner or an authorized
administrator.
/// Throws if msg.value is too low. Throws if _newName is longer than 32 bytes.
/// Throws if IPC's experience is too low. Emits the NameChanged event.
/// @param _ipcId The IPC Identifier for the IPC whose name will be changed.
/// @param _newName Set the IPC's name to this string.
function changeIpcName(
    uint _ipcId,
    string _newName
) public payable
noReentrancy
onlyOwnerOrAdmin(priceToChangeName, _ipcId)
levelRequirement(nameModificationLevelRequirement, _ipcId)
{
    require(bytes(_newName).length <= 32);
}

```



```

uint index = _ipcId - 1;
Ipcs[index].name = _newName;
emit NameChanged(_ipcId, _newName);
if (msg.sender == ipcToOwner[_ipcId]) {
    uint price = _convertCentsToWei(priceToChangeName);
    msg.sender.transfer(msg.value - price);
    ipcCashier.transfer(price);
}
}

/// @notice Changes a specific byte of an IPC's dna by an amount. The price
/// scales with modification amount.
/// @dev Throws unless msg.sender is the IPC's owner or an authorized
administrator.
/// Throws if msg.value is too low. Throws if _byteToModify is greater than 31.
/// Throws unless _ipcId is a valid IPC. Throws if modifying the _byteToModify by
/// _modifyAmount overflows or underflows the byte value. Emits the DnaModified
event.
/// @param _ipcId The IPC Identifier for the IPC whose DNA will be modified
/// @param _byteToModify The index of the byte to modify. Must be less than 32
/// @param _modifyAmount The amount by which to increase or decrease the DNA byte
value
function modifyDna(
    uint _ipcId,
    uint _byteToModify,
    int _modifyAmount
) public payable
noReentrancy
levelRequirement(dnaModificationLevelRequirement, _ipcId)
{
    // check enough money was sent
    uint costInWei;
    if(_modifyAmount < 0) {
        costInWei = _convertCentsToWei(priceToModifyDna * uint(_modifyAmount *
-1));

        require (
            _checkIfAdmin(msg.sender) ||
            (msg.sender == ipcToOwner[_ipcId] && msg.value >= costInWei)
        );
    } else {
        costInWei = _convertCentsToWei(priceToModifyDna * uint(_modifyAmount));
        require (

```



```

        _checkIfAdmin(msg.sender) ||
        (msg.sender == ipcToOwner[_ipcId] && msg.value >= costInWei)
    );
}
Ipc storage myIpc = Ipcs[_ipcId - 1];
// requirements
require (_ipcId < totalIpcs && myIpc.timeOfBirth != 0); // require valid IPC
require (_byteToModify < 32); // require valid byte index (0-31)

// calculate new dna value
require (int(myIpc.dna[_byteToModify]) + _modifyAmount < 256 &&
    int(myIpc.dna[_byteToModify]) + _modifyAmount >= 0);
int newDnaValue = int(myIpc.dna[_byteToModify]) + _modifyAmount;

// construct an array of bytes as new dna
bytes memory newDna = new bytes(32);
for (uint i = 0; i < 32; ++i) {
    if (i == _byteToModify) {
        newDna[i] = byte(newDnaValue);
    } else {
        newDna[i] = myIpc.dna[i];
    }
}

// convert the array of bytes into a fixed-size bytes32 array
bytes32 tempDnaBytes32;
assembly {
    tempDnaBytes32 := mload(add(newDna, 32))
}

// set ipc dna to modified dna value
myIpc.dna = tempDnaBytes32;

// send event
emit DnaModified(_ipcId, tempDnaBytes32);

// send money
if (msg.sender == ipcToOwner[_ipcId]) {
    msg.sender.transfer(msg.value - costInWei);
    ipcCashier.transfer(costInWei);
}
}

```



```

}

//-----
/// @title IPC Experience
/// @dev Defines developer functions: experience creation, purchasing of XP, and
/// functions to grant XP to IPCs.
//-----
contract IpcExperience is IpcModification {
    /// @dev This emits any time an IPC is gifted an XP by a developer
    event ExperienceEarned(uint indexed tokenId, address indexed developer, uint
indexed xpId);

    struct Developer {
        uint32 experienceCount;
        uint32 xpBalance;
        string name;
    }

    // contains information about a specific achievable experience
    struct Experience {
        address developer;
        string description;
    }

    // stores the developer info
    mapping (address => Developer) addressToDeveloper;
    // stores whether an IPC has been granted xp for a specific experience
    mapping (uint => mapping(uint => bool)) public ipcIdToExperience;
    // array of all experiences
    Experience[] public experiences;

    // pricing data
    uint xpPriceInCents = 1;

    function changeXpPrice(uint _newAmount) public onlyExecOrHigher {
        xpPriceInCents = _newAmount;
    }

    function setDeveloperName(address _address, string _name) external
onlyAdminOrHigher {
        require (_address != 0 && ipcDeveloper[_address]);

```



```

    addressToDeveloper[_address].name = _name;
}

//-----
// DEVELOPER FUNCTIONS
//-----
/// @notice Grants xp to ipc
/// @dev Throws if developer XP balance is 0. Throws if developer not
/// the experience's owner. Throws if the IPC is not substantiated. Throws
/// if IPC already earned the experience. Emits ExperienceEarned event.
/// @param _ipcId The IPC Identifier for the IPC receiving the XP
/// @param _xpId The XP Identifier for the experience given
function grantXpToIpc(uint _ipcId, uint _xpId) public onlyDeveloper {
    Ipc storage ipc = Ipcs[_ipcId - 1];
    require (
        addressToDeveloper[msg.sender].xpBalance > 0 &&
        experiences[_xpId].developer == msg.sender &&
        ipc.timeOfBirth != 0 &&
        ipcIdToExperience[_ipcId][_xpId] == false
    );
    ipcIdToExperience[_ipcId][_xpId] = true;
    ipc.experience++;
    addressToDeveloper[msg.sender].xpBalance--;
    emit ExperienceEarned(_ipcId, msg.sender, _xpId);
}

/// @notice Grants XP to multiple IPCs. Costs significantly less gas than
one-by-one.
/// @dev Throws if developer's XP balance is too low to complete the operation.
/// Throws if arrays are not equal in length. Checks each IPC against each
experience.
/// If valid, grants the XP. If not, continues to the next IPC and XP.
/// It is up to the caller to make sure the IPC and XP arrays align correctly.
/// @param _ipcIdArray An array of IPC Identifiers for the IPCs to receive XP
/// @param _xpIdArray An array of XP to grant to IPCs.
function grantBulkXp (uint[] _ipcIdArray, uint[] _xpIdArray) public onlyDeveloper
{
    require(addressToDeveloper[msg.sender].xpBalance >= _ipcIdArray.length &&
        _ipcIdArray.length == _xpIdArray.length);
    for (uint i = 0; i < _ipcIdArray.length; ++i) {
        if (
            addressToDeveloper[msg.sender].xpBalance > 0 &&

```



```

        experiences[_xpIdArray[i]].developer == msg.sender &&
        Ipcs[_ipcIdArray[i]].timeOfBirth != 0 &&
        ipcIdToExperience[_ipcIdArray[i]][_xpIdArray[i]] == false
    ) {
        ipcIdToExperience[_ipcIdArray[i]][_xpIdArray[i]] = true;
        Ipcs[_ipcIdArray[i] - 1].experience++;
        emit ExperienceEarned(_ipcIdArray[i], msg.sender, _xpIdArray[i]);
    }
}
addressToDeveloper[msg.sender].xpBalance -= uint32(_ipcIdArray.length);
}

/// @notice Developer-only function to purchase Experience Points
/// @dev Sends the developer however many experience points his msg.value
/// can afford. Sends the remainder back to the buyer. Throws if sender is
/// not a developer.
function buyXp() external payable onlyDeveloper noReentrancy {
    uint xpPriceInWei = _convertCentsToWei(xpPriceInCents);
    uint xpBought = msg.value / xpPriceInWei;
    addressToDeveloper[msg.sender].xpBalance += uint32(xpBought);
    ipcCashier.transfer(xpPriceInWei * xpBought);
    msg.sender.transfer(msg.value - (xpPriceInWei * xpBought));
}

/// @notice Registers a new experience into the idToExperience mapping
/// @dev Throws if the sender is not a developer. The longer the _description
/// the more gas this function costs. If the description is very long
/// it is recommended to store a metadata uri conforming to RFC3986 syntax
/// described here https://tools.ietf.org/html/rfc3986
/// @param _description String describing the experience
function registerNewExperience(string _description) external onlyDeveloper
returns(uint) {
    uint experienceId = experiences.push(Experience (msg.sender, _description)) -
1;
    addressToDeveloper[msg.sender].experienceCount++;
    return experienceId;
}

/// @notice Invalidates an existing Experience
/// @dev Sets the experience's developer to the 0 address, which makes the
/// experience impossible to access. If the experience needs to be reactivated
/// it needs to be created as a new experience.

```



```

/// @param _xpId The Experience Identifier for the Experience to remove
function removeExperience(uint _xpId) external onlyDeveloper {
    require (experiences[_xpId].developer == msg.sender);
    addressToDeveloper[msg.sender].experienceCount--;
    experiences[_xpId].developer = 0;
}

/// @notice Developer-only getter that returns XP price in cents
/// @dev Throws if msg.sender is not a developer
function getXpPrice() external view onlyDeveloper returns (uint) {
    return xpPriceInCents;
}

/// @notice Developer-only getter that returns their XP balance
/// @dev Throws if msg.sender is not a developer
function getXpBalance() external view onlyDeveloper returns (uint) {
    return addressToDeveloper[msg.sender].xpBalance;
}

/// @notice Public getter that shows all the Experiences owned by a developer
/// @dev Throws if the developer owns no Experiences.
/// @param _developer The address to search for owned Experiences.
/// @return An array of Experience Identifiers owned by _developer
function experiencesOfDeveloper(address _developer) external view returns (uint[])
{
    uint counter = addressToDeveloper[_developer].experienceCount;
    require(counter > 0);
    uint[] memory result = new uint[](counter);
    for (uint i = 0; i < Ipcs.length; i++) {
        result[counter] = i;
    }
    return result;
}
}

//-----
// IPC CORE
// - provides the IPC interface for which external contracts or dapps can read IPC
data
//-----
contract IpcCore is IpcExperience, ERC721Metadata {

```





```

address public mostCurrentIpAddress = address(this);
mapping (bytes4 => bool) supportedInterfaces;
string ipcUrl;

// Constructor - called once and only once when contract is created
function IpcCore() public {
    ipcGod = msg.sender;
    ipcCap = trancheSize;

    supportedInterfaces[0x01ffc9a7] = true;    // ERC-165
    supportedInterfaces[0x6466353c] = true;    // ERC-721
    supportedInterfaces[0x780e9d63] = true;    // ERC721Enumerable
    supportedInterfaces[0xf0b9e5ba] = true;    // ERC721TokenReceiver
    supportedInterfaces[0x5b5e139f] = true;    // ERC721Metadata

    ipcUrl = "https://www.immortalplayercharacters.com/ipc/";

    //priceConverter = MarketPrice(0x2138FfE292fd0953f7fe2569111246E4DE9ff1DC);
// main
    priceConverter = MarketPrice(0x97d63Fe27cA359422C10b25206346B9e24A676Ca);    //
testnet
}

function updateIpcContract(address _newAddress) external onlyExecOrHigher {
    mostCurrentIpAddress = _newAddress;
}

function updateIpcUrl(string _newUrl) external onlyExecOrHigher {
    ipcUrl = _newUrl;
}

function addSupportedInterface(bytes4 _newInterfaceId) external onlyExecOrHigher {
    supportedInterfaces[_newInterfaceId] = true;
}

function removeSupportedInterface(bytes4 _interfaceId) external onlyExecOrHigher {
    supportedInterfaces[_interfaceId] = false;
}

/// @notice Returns all ipc stats to the caller.
/// @dev note: If the caller was an external contract, Ipc.name will be

```



unreadable.

```

    /// @param _ipcId The IPC Identifier for the IPC to be read
    /// @return name, attributeSeed, dna, experience, and timeOfBirth of an IPC
    function getIpc(uint _ipcId) external view returns (
        string name,
        bytes32 attributeSeed,
        bytes32 dna,
        uint128 experience,
        uint128 timeOfBirth
    ) {
        Ipc storage ipc = Ipcs[_ipcId - 1];
        name = ipc.name;
        experience = ipc.experience;
        attributeSeed = ipc.attributeSeed;
        dna = ipc.dna;
        timeOfBirth = ipc.timeOfBirth;
    }

    /// @notice Converts name to bytes32 for external contract use.
    /// @dev returns a bytes32 containing the IPCs name followed by 0s. If the
    /// IPCs name ends with 0s they will be indistinguishable from the 0s added
    /// by this method.
    /// @param _ipcId The IPC Identifier for the IPC whose name is being looked up
    /// @return byte32 containing the IPCs name followed by 0s
    function getIpcName(uint _ipcId) external view returns (bytes32 result) {
        bytes memory nameBytes = new bytes(32);
        Ipc storage ipc = Ipcs[_ipcId - 1];
        if (bytes(ipc.name).length == 0) {
            return 0x0;
        }
        for (uint i = 0; i < bytes(ipc.name).length; ++i) {
            nameBytes[i] = bytes(ipc.name)[i];
        }

        assembly {
            result := mload(add(nameBytes, 32))
        }
    }

    /// @notice Check how much wei an existing IPC costs
    /// @dev updates roughly once every ten minutes
    /// @param _ipcId The IPC Identifier for the IPC price to look up

```



```

/// @return The amount the IPC costs in wei
function getIpcPriceInWei(uint _ipcId) external view returns (uint) {
    return _convertCentsToWei(ipcToMarketInfo[_ipcId].sellPrice);
}

// ERC721Metadata functions
function name() external pure returns (string) {
    return "ImmortalPlayerCharacter";
}

function symbol() external pure returns (string) {
    return "IPC";
}

// returns url + IPC as a string
function tokenURI(uint _tokenId) external view returns (string) {
    uint ipcId = _tokenId;
    require (_tokenId > 0 && _tokenId <= totalIpcs);
    bytes32 ipcIdBytes32;
    while(_tokenId > 0) {
        ipcIdBytes32 = bytes32(uint(ipcIdBytes32) / (2 ** 8));
        ipcIdBytes32 |= bytes32((( _tokenId % 10) + 48) * 2 ** (8 * 31));
        ipcId /= 10;
    }

    bytes memory bytesString = new bytes(32);
    for (uint i = 0; i < 32; ++i) {
        byte char = byte(bytes32(uint(ipcIdBytes32) * 2 ** (8 * i)));
        if (char != 0) {
            bytesString[i] = char;
        }
    }

    bytes memory newStringBytes = new bytes(bytes(ipcUrl).length +
bytesString.length);
    uint counter = 0;

    for (i = 0; i < bytes(ipcUrl).length; i++) {
        newStringBytes[counter++] = bytes(ipcUrl)[i];
    }
    for (i = 0; i < bytesString.length; i++) {
        newStringBytes[counter++] = bytesString[i];
    }

```



```
    }  
    return string(newStringBytes);  
}  
  
// ERC165 function  
function supportsInterface(bytes4 _interfaceId) external view returns (bool) {  
    return supportedInterfaces[_interfaceId];  
}  
}
```

