

# 1. Data Structures and Classes

See **Appendix B (§3.2)** for a list of changes from the first version.

by Ronny Chan, Corey Fung, Frank Han, Hyebeen Jung.

## 1.1. Serialization Framework (serialization.srsf.\*)

The following data structures involve the serialization and organization of data in the program. The `serialization.srsf.*` API supports saving and loading data to and from *Simple Relational String Format* data files, see **Appendix A (§3.1)** for format specifications.

### 1.1.1. `serialization.srsf.SerializationContext`

#### Description

The context in which the collection of objects is loaded. `Lazy<T>` should be used in conjunction with `LazyResolver<T>` to resolve object references within the context, thus, object references in serialized SRSF text format must be self-contained within the `SerializationContext`.

#### Constructor

```
public SerialiationContext(String directory)
```

#### Fields

Field Name	Field Type	Field Description
directory	String	The root directory of the serialization context
serializers	HashMap<String, Serializer>	The serializers, keyed on the name of the class it is responsible for. See <code>Serializer</code>
collection	HashMap<String, List>	The loaded collections of objects, keyed on the name of the class of each collection.

#### Methods

```
<T> void addSerializer(Serializer<T>, Class<T>)
```

Adds a `Serializer` of type `T` to the serialization context. All serializers must be added before loading any object collections.

```
//pseudocode implementation
serializers.put(class.getName(), serializer)
//example usage
serializer.addSerializer(new MyTypeSerializer(), Type.class);
```

```
<T> void loadCollection(Class<T>)
```

Loads the collection of type `T` from the file `Name.srsf` in the context directory, where `Name` is the simple name of the class `T`. For example, the class `pokemon.data.PokemonType` must load its information from `PokemonType.srsf` in the context directory.

```
lines = loadFile(directory + class.getName() + ".srsf");
currentBlock = new HashMap<String, KeyValuePair>();
for line in lines:
    if line is "==" getCollection(class).add(currentBlock)
    else
        value = line.split("|")
        currentBlock.add(value[0], new KeyValuePair(value[0], value[1]))
```

```
<T> void saveCollection(Class<T>)
```

Saves the collection of type `T` to the file `Name.srsf` in the context directory, where `Name` is the simple name of the class `T`. For example, the class `pokemon.data.PokemonType` will save it's information to `PokemonType.srsf` in the context directory.

```

file.WriteLine("!!" + class.getName())
for object in getCollection(class.getName()):
    for kvp in serializers.get(class.getName()).deserialize(object):
        file.WriteLine(kvp.key + "|" + kvp.value)
file.WriteLine("---")

```

**<T> List<T> getCollection(Class<T>)**

Gets the loaded collection of items of type T as a List.

### 1.1.2. interface *serialization.srsf.LazyResolver*

#### Description

An interface that represents a factory object used to resolve, or provide an instance of, an object at a later time, thus deferring initialization after object initialization for use with Lazy.

#### Methods

**T resolve()**

Resolves the lazy object by creating a new instance of the object. This method acts as a factory for objects of type T.

```

//example implementation
LazyResolver<MyClass> resolver = new LazyResolver() {
    //anonymous class
    @Override
    public MyClass resolver() {
        return new MyClass();
    }
}

```

### 1.1.3. *serialization.srsf.Lazy*

#### Description

Represents a lazily evaluated object type. By providing a LazyResolver<T> as an object factory, Lazy<T> provides an object only upon request, this can be thought of as a promise to instantiate an object. Once instantiated, the instance is cached, and subsequent requests for the object must return the same instance.

#### Constructor

```
public Lazy(LazyResolver<T> resolver)
```

#### Fields

Field Name	Field Type	Field Description
resolver	LazyResolver<T>	The resolver for this Lazy<T>.
isCreated	boolean	Whether the value has been created or not.
value	T	The cached value.

#### Methods

**T getValue()**

Returns the cached or instantiates a new value of type T with the provided LazyResolver

```

//example implementation
if not isValueCreated():
    value = resolver.resolve()
return value

```

**boolean isValueCreated()**

Returns whether or not the instance or value has been created with the LazyResolver; in other words whether getValue() had previously been called.

### 1.1.4. abstract *serialization.srsf.Serializer*

#### Description

Represents a serializer to convert from the key value pair representation of an SRSF block to objects of type T, and vice versa.

## Constructor

```
public Serializer(SerializationContext context)
```

## Fields

Field Name	Field Type	Field Description
context	SerializationContext	See getContext()

## Methods

```
protected serialization.srsf.SerializationContext getContext()
```

Gets the SerializationContext associated with this serializer.

```
abstract T deserialize(HashMap<String, KeyValuePair>)
```

Converts from the key value pair representation of an SRSF block to an instance of T.

```
/* example implementation given following block

$foo|Hello
$bar|World
---
*/
Serializer<FooBar> serializer = new Serializer() {
    @Override
    public FooBar deserialize(HashMap<String, KeyValuePair> kvp) {
        return new FooBar(kvp.get("$foo").asString(),
            kvp.get("$bar").asString()); //FooBar { foo: "Hello", bar: "World" }
    }
}
```

```
abstract HashMap<String, String> serialize(T)
```

Converts from the instance of T to a hashmap of String/Value pairs ready for saving.

### 1.1.5. *serialization.srsf*.KeyValuePair

## Description

A helper class to represent a single SRSF key value pair (\$name|value).

## Constructor

```
public KeyValuePair(String key, String value)
```

## Fields

Field Name	Field Type	Field Description
value	String	The string value of the pair
key	String	The string key of the pair

## Methods

```
String getKey()
```

Gets the key of the key value pair.

```
String asString()
```

Gets the value as a string. If the string value is equal to the magic string @@NULL@@, returns null.

```
String[] asStringArray()
```

Gets the value as a string array. If the string value is equal to the magic string @@NULL@@, returns null.

```
int asInt()
```

Attempts to get the value as an integer. If the string value is equal to the magic string @@NULL@@, returns 0.

```
int[] asIntArray()
```

Attempts to get the value as an integer array. If the string value is equal to the magic string @@NULL@@, returns null.

**double asDouble()**

Attempts to get the value as a double. If the string value is equal to the magic string @@NULL@@, returns 0.

**double[] asDoubleArray()**

Attempts to get the value as a double array. If the string value is equal to the magic string @@NULL@@, returns null.

**boolean asBoolean()**

Attempts to get the value as a boolean. If the string value is equal to the magic string @@NULL@@, returns false.

**boolean[] asBooleanArray()**

Attempts to get the value as a boolean array. If the string value is equal to the magic string @@NULL@@, returns null.

## 1.2. Simple Relational String Format Schema (`serialization.srsf.schema.*`)

### 1.2.1. *serialization.srsf.schema.Schema*

#### Description

Represents the SRSF Schema. See **Appendix A (§3.1)** for details.

#### Constructor

```
public Schema(String schemaName, String outputType, HashMap<String, String> schemaProperties)
```

#### Fields

Field Name	Field Type	Field Description
schemaProperties	HashMap<String, String>	See getSchemaProperties()
schemaName	String	See getSchemaName()
outputType	String	See getOutputType()

#### Methods

**HashMap<String, String> getSchemaProperties()**

Gets the field names (properties) and types as String/String key value pairs. See **Appendix A (§3.1)** for details.

**String getSchemaName()**

Gets the name of the schema.

**String getOutputType()**

Gets the output type of the schema.

### 1.2.2. *serializations.srsf.schema.SchemaSerializer* extends *Serializer<Schema>*

#### Description

A serializer to convert from the schema SRSF format to a Schema object.

#### Methods

**@Override Schema deserialize(HashMap<String, KeyValuePair>)**

Converts schema SRSF files to Schema objects

**@Override HashMap<String, String> serialize()**

Converts the schema to string/string key value pairs.

## 1.3. Text-based Menu Framework (`menu.text.*`)

The following classes assist with building text-based menus, allowing options to be separately run and tested.

### 1.3.1. *menu.text.MenuBuilder*

#### Description

Represents a menu with various options. Menus may contain options that spawn sub-menus. Exposes a fluent API to prepare various menu options, and a menu loop to display and execute the menu options.

#### Constructor

```
public MenuBuilder()
```

#### Fields

Field Name	Field Type	Field Description
options	List<MenuOption>	The list of all menu options for this menu
exit	MenuOption	The menu option used to exit this menu. Will cease the menu loop, and go back to the previous menu, or exit the application. The exit option is always run when the user selects a menu option less than 1.
error	ErrorHandler	The general handler for uncaught exceptions, usually displays an error message to the user

#### Methods

**MenuBuilder option(MenuOption)**

Adds the given option to this menu, and returns the updated MenuBuilder

```
new MenuBuilder()
    .option(new MyOption());
```

**MenuBuilder exit(MenuOption)**

Sets the given option to the menu exit option, and returns the updated MenuBuilder

```
new MenuBuilder()
    .exit(new MyExitOption());
```

**MenuBuilder error(ErrorHandler)**

Sets the given error handler menu error handler, and returns the updated MenuBuilder.

```
new MenuBuilder()
    .error(new MyErrorHandler());
```

**void run()**

Starts the menu loop, and run until the user chooses to exit.

```
new MenuBuilder()
    .option(new OptionOne())
    .option(new OptionTwo())
    .exit(new MyExitOption())
    .error(new MyErrorHandler())
    .run();
```

**@Override toString()**

Returns the string representation of the menu to be displayed to the user.

### 1.3.2. *abstract menu.text.MenuOption*

#### Description

An abstract class to represent an option in a menu.

#### Fields

Field Name	Field Type	Field Description
------------	------------	-------------------

Field Name	Field Type	Field Description
name	String	See getName()

#### Methods

**String** getName()

Gets the name of the menu option to be displayed to the user

**abstract void** run()

Runs the menu option once.

#### 1.3.3. *abstract menu.text.ErrorHandler*

##### Description

An abstract class to represent an error handler.

##### Methods

**abstract void** handle(Exception)

Handles the given exception.

### 1.4. Pokémon Data Structures (pokemon.data.\*)

The following classes have constructors that take in all fields as constructor parameters.

#### 1.4.1. *pokemon.data.PokemonSpecies*

##### Description

**PokemonSpecies** defines a single species of Pokemon, and include all relevant information.

**PokemonSpecies** should be immutable upon creation, none of the properties of the object or child objects should be publically mutable after creation.

##### Constructor

```
public PokemonSpecies(int nummber, String name, double weight,
    Lazy<PokemonType> primaryType, Lazy<PokemonType>secondaryType,
    Lazy<PokemonSpecies> evolution, Lazy<PokemonSpecies>)
```

##### Fields

Field Name	Field Type	Field Description
number	int	See getNumber()
name	String	See getName()
primaryType	Lazy<PokemonType>	See getPrimaryType()
secondaryType	Lazy<PokemonType>	See getSecondaryType()
weight	double	See getWeight()
evolution	Lazy<PokemonSpecies>	See getEvolution()
preEvolution	Lazy<PokemonSpecies>	See getPreEvolution()

##### Methods

**String** getName()

The name of the Pokémon species, for example, 'Pikachu'. This name should be the same for instances of the same Pokémon Species.

**PokemonType** getPrimaryType()

The primary type of the Pokémon species.

**PokemonType** getSecondaryType()

The secondary type of the Pokémon species.

**double** getWeight()

The average weight of the Pokémon

**int** getNumber()

The unique Pokédex number of the Pokémon species. This number must be unique for each different species.

**PokemonSpecies getNextEvolution()**

The Pokémon species next in the evolutionary chain of the Pokémon. This merely indicates the Pokémon next in the evolutionary chain, and does not specify the requirements for the Pokémon species to evolve into its evolution. If this Pokémon species is the final stage in the evolutionary chain, then this property returns null.

**PokemonSpecies getPreviousEvolution()**

The Pokémon species previous in the evolutionary chain of the Pokémon. This merely indicates the Pokémon previous in the evolutionary chain, and does not specify the requirements for the previous evolution species to evolve into the current Pokémon species. If this Pokémon species is the first in its evolutionary chain, then this property returns null.

**boolean equals(PokemonSpecies)**

PokemonSpecies defines its own equality by comparing the species number `getNumber()`. Two PokemonSpecies are equal if they have the same number, although in practice there should only be one instance of each unique PokemonSpecies throughout the lifetime of the application.

**@Override boolean equals(Object)**

Default equals(Object) override, defers to equals(PokemonSpecies).

#### 1.4.2. *pokemon.data.PokemonMove*

##### Description

Represents a single move along with its base damage, type, and name of the move.

##### Constructor

```
public PokemonMove(String name, Lazy<PokemonType> type,
                    double baseDamage, double selfDamage)
```

##### Fields

Field Name	Field Type	Field Description
name	String	See getName()
type	Lazy<PokemonType>	See getType()
baseDamage	double	See getBaseDamage()
selfDamage	double	See getSelfDamage()

##### Methods

**String getName()**

The name of the move, for example, 'Hyper Beam'.

**int getDamage()**

The damage of a move before any multipliers. Can be negative to apply a healing effect.

**int getSelfDamage()**

The damage applied to the Pokémon that uses this move. Can be negative to apply a healing effect.

**PokemonType getType()**

The type of the move. The type is important in figuring out the damage given to the other Pokémon in battles as each type is strong and weak to other types.

**boolean equals(PokemonMove)**

PokemonMove defines its own equality by its name. Two moves are equal if they have the same name `getName()`, ignoring case, however in practice there should only be a single instance of a unique move throughout the lifetime of the application.

**@Override boolean equals(Object)**

Default equals(Object) override. Defers to equals(PokemonMove) for equality.

#### 1.4.3. *pokemon.data.PokemonTeam*

##### Description

The team is a collection of individual Pokemon, usually up to 6. This class is mutable by the switchPosition and setActivePokemon methods.

##### Constructor

```
public PokemonTeam(ArrayList<Lazy<Pokemon>> pokemon)
```

##### Fields

Field Name	Field Type	Field Description
pokemon	ArrayList<Lazy<Pokemon>>	The ArrayList holding the six pokemon.

##### Methods

```
void switchPosition(int, int)
```

Swaps the positions of the Pokémon in the two specified slots.

```
List<Pokemon> getPokemon()
```

Get all Pokémon in the team.

```
Pokemon getActivePokemon()
```

Gets the first Pokémon in the team.

```
void setActivePokemon(int)
```

Sets the active Pokémon to the Pokémon at the given index. This has the effect of swapping the Pokémon at the given index with the first Pokémon in the team.

#### 1.4.4. *pokemon.data.PokemonType*

##### Description

Representing a certain typing of Pokemon, PokemonType stores the name of the type and it's weaknesses to other types (e.g Fire weak to Water, Rock, Ground), its strengths (e.g Fire strong against Grass, Ice, Bug), and its immunity (e.g. Ground is immune to Electric type). PokemonType is immutable upon creation.

##### Constructor

```
public PokemonType(String name, Lazy<List<PokemonType>> strongAgainst,  
                    List<Lazy<PokemonType>> weakAgainst, List<Lazy<PokemonType>> immuneAgainst)
```

##### Fields

Field Name	Field Type	Field Description
strongAgainst	List<Lazy<PokemonType>>	See getStrengths()
weakAgainst	List<Lazy<PokemonType>>	See getWeaknesses()
immuneAgainst	List<Lazy<PokemonType>>	See getImmunities()
name	String	See getTypeName()

##### Methods

```
List<PokemonType> getStrengths()
```

Gets the types this type is strong against.

```
List<PokemonType> getWeaknesses()
```

Gets the types this type is weak against.

```
List<PokemonType> getImmunities()
```

Gets the types this type is immune against.

```
boolean isWeakAgainst(PokemonType)
```

Checks if the given type is weak against this type.



**boolean isStrongAgainst(PokemonType)**

Checks if the given type is strong against this type.

**boolean isImmuneAgainst(PokemonType)**

Checks if the given type is immune against this type.

**String getName()**

Gets the name of the type.

**boolean equals(PokemonType)**

PokemonType defines its own equality based on whether or not two types have the same name `getName()`, ignoring case, although in practice there should only be one instance of a certain PokemonType throughout the instance of the application.

**@Override boolean equals(Object)**

Default `equals(Object)` override. Defers to `equals(PokemonType)` for equality.

#### 1.4.5. *pokemon.data.Pokemon*

##### Description

Represents an individual Pokémon belonging to the user or player character, including all information, such as species, move set, nickname, level, and health points. This class is mutable by the `setNickname` and `setLevel` methods.

##### Constructor

```
public Pokemon(String id, Lazy<PokemonSpecies> species, List<Lazy<PokemonMove>> moves,
               String name, int level)
```

##### Fields

Field Name	Field Type	Field Description
species	Lazy<PokemonSpecies>	See <code>getSpecies()</code>
moves	List<Lazy<PokemonMove>>	See <code>getMoves()</code>
name	String	See <code>getNickname()</code>
level	int	See <code>getLevel()</code>
id	String	

##### Methods

**PokemonSpecies getSpecies()**

Gets the species of this Pokemon

**List<PokemonMove> getMoves()**

Gets the list of moves for this Pokemon

**String getNickname()**

Gets the given nickname of this Pokémon. If not set, then return the name of the Pokémon species.

**void setNickname(String)**

Sets the given nickname of this Pokémon. To unset, set the nickname to the empty string.

**int getLevel()**

The level of this Pokémon. The higher the level is, the greater the base damage of each Pokémon. Also, the health point increases every time the Pokémon levels up.

**void setLevel(int)**

Sets the level of this Pokémon.

**int getHP()**

Gets the amount of health points of this Pokemon.

**void setHP(int)**

Sets the amount of health points of this Pokemon.

**boolean isFainted()**

The status of the Pokémon that indicates whether it is alive or fainted. If the Pokemon's the health point is zero, the Pokémon is fainted thus it returns true. Otherwise, it returns false.

## 1.5. Pokémon Core Facilities (pokemon.core.\*)

The following classes contain the behavior of the program regarding Pokedex and Battle functionality, and are not meant to be saved to a file. These classes will take in all dependencies in their constructor.

### 1.5.1. *pokemon.core.Pokedex*

#### Description

Represents a sortable and searchable collection of *PokemonSpecies*.

#### Constructor

`public Pokedex(List pokemonSpecies)`

#### Fields

Field Name	Field Type	Field Description
<code>pokemonSpecies</code>	<code>List&lt;PokemonSpecies&gt;</code>	The list of Pokémon species defined in this Pokedex.

#### Methods

`List<PokemonSpecies> searchPokemonByType(PokemonType)`

Searches for and returns all *PokemonSpecies* with the given *PokemonType*.

`List<PokemonSpecies> searchPokemonByName(String)`

Searches for and returns all *PokemonSpecies* with a name matching the given string.

`List<PokemonSpecies> getAllPokemon()`

Gets all the Pokémon in the Pokedex.

`PokemonSpecies getPokemon(int)`

Gets the Pokémon with the given number.

`static void sortByName(List<PokemonSpecies>)`

Sorts the given list of *PokemonSpecies* by their name in lexicographical order. See *PokemonSpeciesNameComparator*

`static void sortByNumber(List<PokemonSpecies>)`

Sorts the given list of *PokemonSpecies* by their number in ascending order. See *PokemonSpeciesNumberComparator*

`static void sortByWeight(List<PokemonSpecies>)`

Sorts the given list of *PokemonSpecies* by their weight in ascending order. See *PokemonSpeciesWeightComparator*

### 1.5.2. *pokemon.core.battle.BattleState*

#### Description

An enumeration of possible battle states.

#### Enumeration

Enumeration	Description
<code>PLAYER_ONE_MOVE</code>	The battle is waiting for the first player to select their move
<code>PLAYER_TWO_MOVE</code>	The battle is waiting for second player
<code>PLAYER_ONE_FAINTED</code>	Player one has a fainted Pokémon and must select a new one from their team.

Enumeration	Description
PLAYER_TWO_FAINTED	Player two has a fainted Pokémon and must select a new one from their team.
PLAYER_ONE_VICTORY	The battle has been completed, player two has exhausted their team, and player one is the victor.
PLAYER_TWO_VICTORY	The battle has been completed, player one has exhausted their team, and player two is the victor.

### 1.5.3. *pokemon.core.battle.BattleManager*

#### Description

Manages a battle between two `PokemonTeams`. `BattleManager` is a state machine intended to be run in a loop until one team has encountered a victory state.

#### Constructor

```
public BattleManager(PokemonTeam teamOne, PokemonTeam teamTwo)
```

#### Fields

Field Name	Field Type	Field Description
state	<code>BattleState</code>	The current state of the battle
teamOne	<code>PokemonTeam</code>	The Pokémon team of the first player
teamTwo	<code>PokemonTeam</code>	The Pokémon team of the second player

#### Methods

```
BattleState getState()
```

Gets the current state of the battle.

```
PokemonTeam getTeamOne()
```

Gets the first team of Pokémon in the battle.

```
PokemonTeam getTeamTwo()
```

Gets the second team of Pokémon in the battle.

```
void applyMove(PokemonMove, Pokemon, Pokemon)
```

Applies the effects of the given move to the casting and target Pokémon. This method calculates the amount of damage dealt to each Pokémon after the move is applied, and determines the new `BattleState` after damage calculations.

```
//pseudocode implementation
if state == BattleState.PLAYER_ONE_MOVE
    isWeakTo = pokemonTwo.getSpecies().isWeakAgainst(move.Type())
    isStrongTo = pokemonTwo.getSpecies().isStrongAgainst(move.Type())

    if isWeakTo multiplier = 2
    if isStrongTo multiplier = 0.5
    else multiplier = 1

    pokemonTwo.setHp(pokemonTwo.getHp() - multiplier * move.getDamage())
    boolean loss = false
    for pokemon in getTeamTwo():
        loss = loss || pokemon.isFainted()
    if loss state = BattleState.PLAYER_ONE_VICTORY else
    if pokemonTwo.isFainted() state = BattleState.PLAYER_TWO_FAINTED
    else state = PLAYER_TWO_MOVE
    return
...
repeat for player two move state.
```

## 1.6. Poké@mon Serialization Classes (`pokemon.serialization.*`)

### Pokémon Serializer Classes

The following classes extend `serialization.srsf.Serializer` and are used to serialize their specified types. Provided is a schema and an example, see **Appendix A (§3.1)** for format

details.

#### 1.6.1. *pokemon.serialization.PokemonTypeSerializer* extends *Serializer<PokemonType>*

Serializes `pokemon.data.PokemonType`

##### Schema

```
$schemaName|PokemonType
$outputType|pokemon.data.PokemonType
@name|string
@strongAgainst|[string!!PokemonType]
@weakAgainst|[string!!PokemonType]
@immuneAgainst|[string!!PokemonType]
---
```

##### Example

```
$name|WATER
$strongAgainst|[FIRE]
$weakAgainst|[GRASS]
$immuneAgainst|[]
---
```

##### Methods

**@Override *PokemonType* deserialize(HashMap<String, KeyValuePair>)**

Converts from a block of the above example to a *PokemonType*.

**@Override *HashMap<String, String>* serialize(*PokemonType*)**

Converts from a *PokemonType* object to it's key value pair representation.

#### 1.6.2. *pokemon.serialization.PokemonSpeciesSerializer* extends

*Serializer<PokemonSpecies>*

Serializes `pokemon.data.PokemonSpecies`

##### Schema

```
$schemaName|PokemonSpecies
$outputType|pokemon.data.PokemonSpecies
@name|string
@index|int
@primaryType|string!!PokemonType
@secondaryType|string!!PokemonType
@weight|double
@evolution|int!!PokemonSpecies
@preevolution|int!!PokemonSpecies
---
```

##### Example

```
$index|1
$name|Bulbasaur
$primaryType|GRASS
$secondaryType|@@NUL@@
$weight|10
$evolution|2
$preevolution|0
---
```

##### Methods

**@Override *PokemonSpecies* deserialize(HashMap<String, KeyValuePair>)**

Converts from a block of the above example to a *PokemonSpecies*. `$evolution` and `$preevlotion` are represented as the Pokémon number.

**@Override *HashMap<String, String>* serialize(*PokemonSpecies*)**

Converts from a *PokemonSpecies* object to it's key value pair representation.

### 1.6.3. *pokemon.serialization.PokemonMoveSerializer* extends *Serializer<PokemonMove>*

Serializes `pokemon.data.PokemonMove`

#### Schema

```
$schemaName|PokemonMove
$outputType|pokemon.data.PokemonMove
@name|string
@type|string!!PokemonType
@baseDamage|int
@selfDamage|int
---
```

#### Example

```
$name|Hyper Beam
$type|NORMAL
$baseDamage|100
$selfDamage|50
---
```

#### Methods

**@Override *PokemonMove* deserialize(*HashMap<String, KeyValuePair>*)**

Converts from a block of the above example to a *PokemonMove*. *\$evolution* and *\$preevlotion* are represented as the Pokémon number.

**@Override *HashMap<String, String>* serialize(*PokemonMove*)**

Converts from a *PokemonMove* object to it's key value pair representation.

### 1.6.4. *pokemon.serialization.PokemonSerializer* extends *Serializer<Pokemon>*

Serializes `pokemon.data.Pokemon`

#### Schema

```
$schemaName|Pokemon
$outputType|pokemon.data.Pokemon
@species|int!!PokemonSpecies
@nickName|string
@level|int
@moves|[string!!PokemonMove]
@hp|int
@id|string
---
```

#### Example

```
$species|1
$nickName|Bulby
$level|10
$moves|[LEECHSEED,TACKLE,@@NULL@@,@@NULL@@]
$hp|100
$id|A9810DJ12D
---
```

#### Methods

**@Override *Pokemon* deserialize(*HashMap<String, KeyValuePair>*)**

Converts from a block of the above example to a *Pokemon*

**@Override *HashMap<String, String>* serialize(*Pokemon*)**

Converts from a *Pokemon* object to it's key value pair representation.

### 1.6.5. *pokemon.serialization.PokemonTeamSerializer* extends *Serializer<PokemonTeam>*

Serializes `pokemon.data.PokemonTeam`

#### Schema

```

$schemaName|PokemonTeam
$outputType|pokemon.data.PokemonTeam
@pokemon|[string!!Pokemon]
---

```

### Example

```

$pokemon|[A9810DJ12D]
---

```

### Methods

**@Override PokemonTeam deserialize(HashMap<String, KeyValuePair)**

Converts from a block of the above example to a PokemonTeam. Pokémon are stored as references by the unique Pokémon ID.

**@Override HashMap<String, String> serialize(PokemonTeam)**

Converts from a PokemonTeam object to it's key value pair representation.

### Pokemon Resolver Classes

Resolves loaded Pokémon information from a serialization context.

#### 1.6.6. *pokemon.serialization.PokemonSpeciesResolver* implements **LazyResolver<<PokemonSpecies>>**

Lazily resolves a Pokémon species from the Pokémon number

#### Fields

Field Name	Field Type	Field Description
context	SerializationContext	The context to resolve the types from
number	int	The Pokémon number

#### Methods

**@Override PokemonSpecies resolve()**

Resolves the Pokémon species from the serialization context

#### 1.6.7. *pokemon.serialization.PokemonIdResolver* implements **LazyResolver<Pokemon>>**

Lazily resolves a Pokémon from the Pokémon ID

#### Fields

Field Name	Field Type	Field Description
context	SerializationContext	The context to resolve the types from
id	String	The Pokémon ID

#### Methods

**@Override Pokemon resolve()**

Resolves the Pokémon from the serialization context

#### 1.6.8. *pokemon.serialization.PokemonTypeResolver* implements **LazyResolver<PokemonType>>**

Lazily resolves a single Pokémon type from their type names.

#### Fields

Field Name	Field Type	Field Description
context	SerializationContext	The context to resolve the types from
typeName	String	The name of the type

#### Methods

**@Override PokemonType resolve()**

Resolves the types from the serialization context

### 1.6.9. *pokemon.serialization.PokemonMoveResolver* implements

`LazyResolver<PokemonMove>>`

Lazily resolves a single Pokémon types from their type names.

#### Fields

Field Name	Field Type	Field Description
context	SerializationContext	The context to resolve the types from
moveName	String	The name of the move

#### Methods

`@Override PokemonType resolve()`

Resolves the move from the serialization context

## 1.7. *Menu Options (pokemon.menu.\*)*

The following classes extend `menu.text.MenuOption` to provide the UI for the application.

### 1.7.1. *pokemon.menu.PokedexMenu*

A menu option to display and search the Pokedex.

#### Fields

Field Name	Field Type	Field Description
pokedex	Pokedex	The Pokedex to use

#### Methods

`@Override run()`

Runs the Pokedex menu option.

### 1.7.2. *pokemon.menu.TeamMenu*

A menu option to manage your Pokémon Team

#### Fields

Field Name	Field Type	Field Description
team	PokemonTeam	The Pokémon team to manage

#### Methods

`@Override run()`

Runs the Pokémon team menu option.

### 1.7.3. *pokemon.menu.BattleMenu*

A menu option to start a Pokémon Battle. Is responsible for managing the `pokemon.core.battle.BattleManager`

#### Fields

Field Name	Field Type	Field Description
battleManager	BattleManager	The Pokémon battle manager to run the battle menu.

#### Methods

`@Override run()`

Runs the Pokémon battle with the battle manager.

### 1.7.4. *pokemon.menu.MainMenu*

The main menu or application entry point

#### Methods

`static void main(String[])`

The entry point for the application. Responsible for initial loading of types, instantiation of the `SerializationContext` and `MenuBuilder`, and managing the overall lifecycle of the program. The main method should instantiate the `SerializationContext` in the data folder of the application startup directory, which will contain the `Schema.srsf` file.

## 1.8. Pokémon Comparators (`pokemon.data.comparators.*`)

The following classes implement `Comparator<T>` for Pokémon Data Types

### 1.8.1. `pokemon.data.comparators.PokemonSpeciesNameComparator` implements `Comparator<PokemonSpecies>`

Compares two `PokemonSpecies` by their name in lexicographical order.

#### Methods

`int compare(PokemonSpecies, PokemonSpecies)`

Compares two Pokémon species by their name `getName()` in lexicographical order. Defers to `String.compare` for comparison.

### 1.8.2. `pokemon.data.comparators.PokemonSpeciesNumberComparator` implements `Comparator<PokemonSpecies>`

Compares two `PokemonSpecies` by their Pokémon number.

#### Methods

`int compare(PokemonSpecies, PokemonSpecies)`

Compares two Pokémon species by their number `getNumber()` Defers to `Integer.compare` for comparison.

### 1.8.3. `pokemon.data.comparators.PokemonSpeciesWeightComparator` implements `Comparator<PokemonSpecies>`

Compares two `PokemonSpecies` by their weight.

#### Methods

`int compare(PokemonSpecies, PokemonSpecies)`

Compares two Pokémon species by their weight `getWeight()` Defers to `Double.compare` for comparison.

## 2. Algorithms and concepts

### 2.1. Accessors

Methods to get an access or to get the value of a field within a class

```
<data type> get<fieldName>()
    return <fieldName>
```

### 2.2. Mutators

Methods to change the value of a field within a class

```
<data type> set<fieldName>(<variable>)
```

### 2.3. Binary Search

This will be used when searching for a specific Pokémon using the ID in a list of Pokémon sorted by ID.

```
<item> BinarySearch(list L, item)
    middle = middle value of list
    if middle's Field = item's Field
```



```

        Then return item;
    If middle's Field > item's Field
    Then
list L2 = first half of L
    return BinarySearch(L2, item)
    If middle's Field < item's Field
    Then
list L2 = second half of L
    return BinarySearch(L2, item)

```

## 2.4. Sequential Search

This will be used search for a Pokémon by name and type.

```

SequentialSearch(list L, String x)
    For each item in the list
        If (y matches x)
            Return y;

```

## 2.5. Bubble Sort

This will be used to sort Pokémon in Pokedex by name, ID, and weight

```

BubbleSort (List L)
    for upperbound equals L.length - 1 down to 1 and sorted is false
Set sorted to true
for j from 0 to upperbound - 1
    If j > j + 1
        set sorted to false
        swap j and j + 1

```

## 2.6. Recursion

This will be used as part of the Binary Search Refer to Binary Search

## 2.7. File Input/Output

This will be used to store or load teams of Pokémon.

## 2.8. Generics

This concept is used during serialization for Lazy instantiation (`Lazy<PokemonType>`), use of dynamic arrays (`List<Pokemon>`), and hash maps (`HashMap<String, String>`). A generic class or algorithm allows the types to be specified later when needed.

## 2.9. Dynamic Arrays

A dynamic array or a List is a resizable array of indefinite size. Used to store Pokemon, types, and allow lazy instantiation of array-like types.

## 2.10. Interfaces

A contract for a class to implement. Used to define the `LazyResolver<T>` interface used during serialization. Also used for `PokemonSpecies` comparators.

## 2.11. Hash Maps

A data structure that maps keys to values. Used during serialization to represent key value pairs.

# 3. Appendices

## 3.1. Appendix A — Simple Relational String Format

*Simple Relational String Format*, or SRSF defines a generic method to load and save a collection of objects into text files. Objects are loaded into a root serialization context from formatted '.srsf' record files, and are accessible as collections of loaded objects. In order to resolve self-references, object references are lazily evaluated and resolved within the context before being produced into objects by the converter. Record files are expected to contain a collection of objects of the same type, and will be serialized as a list or array.

### 3.1.1. Serialization Context

The serialization context is the root context in which a set of SRSF files are loaded into. Each serialization context should be self contained in a folder, and contain at least a `Schema.srsf`, specifying the schema of the data files contained in the folder, and one or more `.srsf` record files describing the actual records. Saving and loading should occur only in the specified context folder, and all `.srsf` files in the context folder must be named as the simple name of the class it represents, for example, data representing instances of `pokemon.data.PokemonType` must be saved and loaded from `PokemonType.srsf`.

### 3.1.2. SRSF Tokens

Token	Example	Description
~~!srsf~~	~~!srsf~~	The magic header required at the beginning of all valid SRSF record files.
!!	!!PokemonType	The type descriptor, describing the Java type this record file is a collection of. If used after a primitive descriptor in a schema, indicates an object reference of the type.
\$	\$name	Indicates the key of the value.
@	@name	For a schema file, indicates the name of a serializable field.
[ ]	[WATER, GRASS]	For a value, indicates that the value represents an array of primitives.
,	[WATER, GRASS]	In an array value, separates the values of the array.
---	---	Block marker delimiter, indicates the end of a record block.
@@NULL@@	@@NULL@@	Indicates the null or default value
#	#Some comment is is written here.	Indicates that the line begins a comment.
	\$name Pokemon	The vertical bar symbol separates the key from the value in a line

### 3.1.3. SRSF Record Format

An SRSF file begins with the magic string `~~!srsf~~`, followed by the type marker `!!`, a type, and the block end marker. Thus, the first "block" in any SRSF format specifies the schema type for the rest of the file. Files may contain any amount of blocks delimited by the block marker delimiter, and each block must contain the data required to instantiate one instance of the type specified.

```

~~!srsf~~
!!PokemonType
---
$name|GRASS
$strongAgainst|[WATER,GROUND]
$weakAgainst|[FIRE]
#todo add more types
---
```

This example file denotes a collection of `PokemonType` objects, with a single instance. Containing the header block and one record block, the record block has the properties `name`, `strongAgainst`, and `weakAgainst`. `strongAgainst` and `weakAgainst` are string arrays, but as specified in a separate schema file, are in actuality typed to be `PokemonTypes` themselves, and must be deserialized accordingly.

### 3.1.4. SRSF Schema Files

The SRSF Schema format defines a schema for every single record file in the serialization context. Named `Schema.srsf`, this file should be the first to be loaded in any serialization context. One can consider the schema file to be a special case of Record, as they are serialized just the same as a normal record. However, they are privileged in that the `Schema.srsf` need not a schema specified for itself, and the Schema format is the only format specified to have metadata in it's definitions; thus each record shape will be different, unlike normal records. These schema files are usually used for deserialization, and for machine parsing of various record formats.

```
!!schema
---
$schemaName|PokemonType
$outputType|pokemon.data.PokemonType
@name|string
@strongAgainst|[string!!PokemonType]
@weakAgainst|[string!!PokemonType]
---
```

This schema defines the type for the previous `PokemonType` example. A schema has two required field, `schemaName`, indicating the name of the schema, generally the simple name of the type, and `outputType`, indicating the fully qualified name of the Java type it represents. The entries that begin with `@` are metadata properties and are used to define the shape of the schema. Rather than have values, they must contain either a primitive type, or a primitive that will be serialized into an object reference. SRSF primitives are not exactly the same as Java primitives, the list of SRSF primitives is defined as follows

SRSF Primitive	Definition	Java Equivalent
string	A string-like type.	java.lang.String
bool	A true or false value represented by the string literals <code>true</code> or <code>false</code> .	boolean
int	A machine-length integer guaranteed to be at least 30 bits large.	int
double	An IEEE-754 double precision floating-point number	double

Thus, each SRSF must be composed of, or have objects references derived, on the responsibility of the Serializer, these four SRSF primitive types.

## 3.2. Appendix B — Changelog

### 3.2.1. January 13th 2017

- Initial submission.

### 3.2.2. January 16th 2017

- Fixed a bug where the angle brackets used in generic types would not render, due to the document renderer interpreting the brackets as an HTML tag. Generic implementations and extensions should now read properly `extends Lazy<Pokemon>`, for example, rather than `extends Lazy<`.
- Improved organization in UML diagram to be easier to follow and interpret.
- Fixed a bug where the double bar was interpreted as a table-column separator by the document renderer, and breaking the SRSF token description in §3.1.2.
- Fixed appendices being placed in the wrong section, they are now under §3.
- Explicitly denote constructor signatures.
- Added section markers for references to Appendix A.
- Added changelog as Appendix B (§3.2).
- Added a list of authors.
- Add additional pseudocode examples
- Add syntax highlighting to code examples.
- Minor text fixes.

