Ludii Tutorials Documentation

Release 1.0.0

The Ludii Team

Jun 20, 2022

Getting Started

Ludii is a general game system designed to play, evaluate and design a wide range of games, including board games, card games, dice games, mathematical games, and so on. These pages provide tutorials for designing games in Ludii, and various programmatic use cases of Ludii (implementing, testing and evaluating Artificial Intelligence in Ludii, running games for game evaluation or other purposes, etc.).

CHAPTER 1

Installing Ludii

1.1 Prerequisites

Ludii requires Java version 8 or higher to be installed on your computer. Java can be downloaded from: https://www.java.com/download/. Ludii should run correctly on any major operating system. It has been verified to run correctly on the following operating systems:

- Windows 10.
- OS X El Capitan 10.11.6, OS X Mojave 10.14.3.
- Ubuntu 16.04, Ubuntu 18.04, Ubuntu 19.04.

1.2 Download and Installation

The latest version of Ludii may always be downloaded from Ludii's downloads page. This page also contains an archive of older versions of Ludii, and various extra downloads (such as documentation).

No additional installation steps are required – after downloading Ludii, it can be used directly. However, because Ludii occasionally writes files in the directory that it is run from, it may be convenient to place it in a directory of its own somewhere.

1.3 Running Ludii

The easiest way to launch Ludii is to double-click the downloaded `Ludii.jar` file. Alternatively, it may be launched by navigating to the directory containing the `Ludii.jar` file in a command prompt, and entering:

```
java -jar Ludii.jar
```

The Ludii downloads page also contains an extensive user guide, which explains how to use Ludii.

CHAPTER 2

Installing the Ludii Tutorials Repository

Note: Following these instructions for installing the Ludii Tutorials Repository is only required if you are interested in running some of the code examples from various programming tutorials. This is irrelevant for non-programming use cases of Ludii, such as game design.

The Ludii Tutorials repository on GitHub provides various code examples to go along with some of the tutorials on these pages. This page lists the steps required to run these code examples locally:

1. Clone the repository from: https://github.com/Ludeme/LudiiTutorials.
2. Create a `Java` project in your favourite IDE, using the source code in the cloned repository.
3. Suppose that the repository was cloned in the directory `<install_dir>/LudiiTutorials`, which already contains `src` and `docs` directories. Create a new directory `<install_dir>/LudiiTutorials/libs` alongside them, and place the `Ludii.jar` file (downloaded from Ludii's downloads page) in it.
4. Set up the project in your IDE to use the `Ludii.jar` file as a library. Most of the code requires this as a dependency.
5. Also set the project to use the other two `.jar` files that are already included in `<install_dir>/LudiiTutorials/libs` as libraries; these are only required for the unit tests in this repository.
6. The code examples for various programming tutorials can all be found in the `<install_dir>/LudiiTutorials/src/ludii_tutorials` package. Each of these `.java` files has a main method, which means that it can be run directly to see that tutorial's code in action.

CHAPTER 3

Writing .lud Descriptions – Basics

3.1 The .lud Format

Game descriptions for Ludii are written in text files with a `.lud` extension. The language used to describe games for Ludii is defined by a class grammar approach; it is automatically derived from the *ludeme* classes available in Ludii's `.jar` file.

Note: A full, detailed Ludii Language Reference may be downloaded from Ludii's downloads page.

The basic premise of the language is that ludemes are described as their name, followed by a whitespace-separated list of arguments, all wrapped up in a pair of parentheses:

```
(ludemeName arg1 arg2 arg3 ...)
```

Generally, the "outer" ludeme (the first one that is visible in a game description file) will be of the type `(game ...)`. Arguments may be of any of the following types:

1. *Ludemes*: many ludemes can be used as arguments of other ludemes, which ultimately results in games being described as **trees of ludemes**.
2. *Strings*: typically used to provide meaningful names to games, pieces, regions, etc. Strings are always written in a pair of double quotes, for example: `"Pawn"`. By convention, names usually start with an uppercase symbol.
3. *Booleans*: the boolean constants `true` and `false` may be used for any boolean (function) parameters.
4. *Integers*: integer constants can simply be written directly in any `.lud` descriptions, without requiring any special syntax: `1`, `-1`, `100`, etc.
5. *Floats*: any number containing a dot will be interpreted as a float constant. For example: `0.5`, `-1.2`, `5.5`, etc. In ludemes that expect floats as argument, numbers without dots (such as just `1`) cannot be used, and the same number should be written to include a decimal component instead (e.g., `1.0`).

As a first example, the following code shows the full game description for *Tic-Tac-Toe*:

```
(game "Tic-Tac-Toe"
   (players 2)
   (equipment
      {
         (board (square 3))
         (piece "Disc" P1)
         (piece "Cross" P2)
      }
   )
   (rules
      (play (move Add (to (sites Empty))))
      (end (if (is Line 3) (result Mover Win)))
   )
)
```

3.2 Viewing Ludii's Built-in Game Files

The `Ludii.jar` file is not only a runnable program, but also an archive containing files. It can be extracted or opened like any regular `.zip` archive, which allows for the individual files inside it to be inspected. One of the top-level directories inside it is the `/lud/` directory. Under this directory, all of the `.lud` files for all the built-in Ludii games can be found. They may all serve as examples for game designers.

CHAPTER 4

Writing Amazons in .lud Format

This tutorial provides a step-by-step walkthrough of how to implement the game *Amazons*, from scratch, in the `.lud` format.

Amazons is played on a `10x10` board. Each player has four amazons (chess queens), with other pieces used as arrows. Every turn consists of two moves. First, a player moves one of their amazons like a Chess queen, without crossing or entering a space occupied by another amazon or arrow. Second, it shoots an arrow to any space on the board that is along the unobstructed path of a queen's move from that place. The last player able to make a move wins.

Note: For each of the following steps, the Ludii Tutorials GitHub repository contains a `.lud` file with the contents written in that step. They can all be loaded in Ludii and "played", although some of them may not be particularly interesting to play!

4.1 Step 1: A Minimum Legal Game Description

We start out with the minimum description that results in a legal game description that may be loaded in Ludii by defining the number of players, the board by its shape and its size, the most used playing rules, and a basic ending rule.

```
(game "Amazons"
    (players 2)
    (equipment
        {
            (board (square 10))
        }
    )
    (rules
        (play
            (forEach Piece)
        )

```

(continues on next page)

(continued from previous page)

```
        (end
            (if
                (no Moves Next)
                (result Mover Win)
            )
        )
    )
)
```

Line 2 defines that we wish to play a two-player game, where it is implied by default to be an alternating-move game. Line 3 defines the equipment, and is used to list all the items used in the game. Line 5 defines that we wish to use a square board of size 10. By default, the square board is tiled by squares. Line 8 is used to define the rules of the game; the minimum rules to compile are the playing and the ending rules. Lines 9-11 describe the playing rules by using one of the simplest `play` rules available in Ludii; `(forEach Piece)`, which simply defines that Ludii should loop through all pieces owned by a player, and extract legal moves from the piece types to generate the list of legal moves for a mover. Finally, lines 13-18 describe the ending rules. Here we want the player who last made a move to win the game whenever the next player has no move.

4.2 Step 2: Defining the Pieces

In this step, we add the pieces to the equipment.

```
(game "Amazons"
    (players 2)
    (equipment
        {
            (board (square 10))
            (piece "Queen" Each)
            (piece "Dot" Neutral)
        }
    )
    (rules
        (play
            (forEach Piece)
        )

        (end
            (if
                (no Moves Next)
                (result Mover Win)
            )
        )
    )
)
```

Line 6 defines that each player should have a piece type labelled `"Queen"`. Ludii will automatically label these as `"Queen1"` and `"Queen2"` for players 1 and 2, respectively. Additionally, in line 7 we define a "Dot" piece type, which is not owned by any player. This is the piece type that we will use in locations that players block by shooting their arrows.

4.3 Step 3: Defining the Starting Rules

We extend the game description listed above by adding `start` rules to place the pieces on the board:

```
1  (game "Amazons"
2      (players 2)
3      (equipment
4          {
5              (board (square 10))
6              (piece "Queen" Each)
7              (piece "Dot" Neutral)
8          }
9      )
10     (rules
11         (start
12             {
13                 (place "Queen1" {"A4" "D1" "G1" "J4"})
14                 (place "Queen2" {"A7" "D10" "G10" "J7"})
15             }
16         )
17         (play
18             (forEach Piece)
19         )
20
21         (end
22             (if
23                 (no Moves Next)
24                 (result Mover Win)
25             )
26        )
27     )
28 )
```

Lines 11-16 ensure that any game is started by placing objects of the two different types of queens in the correct starting locations. The labels used to specify these locations can be seen in Ludii by enabling "Show Coordinates" in Ludii's *View* menu.

4.4 Step 4: Step 4: Adding the Final Rules for *Amazons*

To complete the game of *Amazons*, we need to allow players to move their queens and to shoot an arrow after moving a queen. This is implemented in the following game description:

```
1  (game "Amazons"
2      (players 2)
3      (equipment
4          {
5              (board (square 10))
6              (piece "Queen" Each (move Slide (then (moveAgain))))
7              (piece "Dot" Neutral)
8          }
9      )
10     (rules
11         (start
12             {
13                 (place "Queen1" {"A4" "D1" "G1" "J4"})
```

(continues on next page)

(continued from previous page)

```
                (place "Queen2" {"A7" "D10" "G10" "J7"})
            }
        )
        (play
            (if (is Even (count Moves))
                (forEach Piece)
                (move Shoot (piece "Dot0"))
            )
        )

        (end
            (if
                (no Moves Next)
                (result Mover Win)
            )
        )
    )
)
```

To make the queens able to move, inside the queen pieces, we have added the following: `(move Slide (then (moveAgain)))`. By default, the `(move Slide)` ludeme defines that the piece is permitted to slide along any axis of the used board, as long as we keep moving through locations that are empty. No additional restrictions – in terms of direction or distance, for example – are required for queen moves. We have appended `(then (moveAgain))` in the queen moves. This means that, after any queen move, the same player gets to make another move.

In lines 18-21, the `play` rules have been changed to no longer exclusively extract their moves from the pieces. Only at even move counts (0, 2, 4, etc.) do we still make a queen move (using `(forEach Piece)`. At odd move counts, the moves are defined by `(move Shoot (piece "Dot0"))`. This rule lets us shoot a piece of type `"Dot0"` into any empty position, starting from the location that we last moved to – this is the location that our last queen move ended up in. This game description implements the full game of *Amazons* for Ludii.

Once pieces are defined, their names are internally appended with the index of the owning player. For example, the above description defines a "Queen" piece for players 1 and 2, then the subsequent description refers to "Queen1" for "Queen" pieces belonging to Player 1 and "Queen2" for "Queen" pieces belonging to Player 2. The "Dot" piece is referred to as "Dot0", indicating that this is a neutral piece not owned by any player. Note that pieces can also be referred to by their undecorated names in the game description, e.g. "Queen" or "Dot", in which case the reference applies to all pieces with that name belonging to any player.

4.5 Step 5: Improving Graphics

The game description above plays correctly, but does not look appealing because it uses Ludii's default colours for the board. This can be easily improved by adding graphics metadata:

```
(game "Amazons"
    (players 2)
    (equipment
        {
            (board (square 10))
            (piece "Queen" Each (move Slide (then (moveAgain))))
            (piece "Dot" Neutral)
        }
    )
    (rules
        (start
```

(continues on next page)

(continued from previous page)

```
            {
                (place "Queen1" {"A4" "D1" "G1" "J4"})
                (place "Queen2" {"A7" "D10" "G10" "J7"})
            }
        )

        (play
            (if (is Even (count Moves))
                (forEach Piece)
                (move Shoot (piece "Dot0"))
            )
        )

        (end
           (if
             (no Moves Next)
             (result Mover Win)
           )
        )
    )
)

(metadata
    (graphics
        {
            (piece Scale "Dot" 0.333)
            (board Style Chess)
        }
    )
)
```

Line 37 makes the “Dot” pieces smaller, and line 38 applies a Chess style to the board.

CHAPTER 5

Ludii Programming Terminology

This page describes some of the Ludii terminology and core concepts relevant for programmatic users of Ludii:

Game In Ludii, the `Game` type refers to a type of object that contains all the rules, equipment, functions etc. required to play. A single object of this type is instantiated when Ludii compiles the contents of a `.lud` file.

Trial A `Trial` in Ludii corresponds to a record of a game played at a particular time (i.e., where a `Game` object would be "the game of *Chess*", a `Trial` object would be "a game of *Chess* as played by these persons at this time". Trials in Ludii store the full history of moves applied throughout the trial, as well as any already-determined player rankings.

State A `State` stores all the relevant properties of a game state (minus the history of moves, which is contained in the Trial as described above).

Context A `Context` object in Ludii describes the context of a current trial being played, and is generally the most convenient object to pass around through methods. It provides pointers to the "higher-level" `Game` object, as well as the "lower-level" `Trial` and `State` objects.

Action `Action` objects in Ludii are atomic objects that, when applied to a game state, modify a single property of it. Note that these do **not** correspond directly to the decisions that players can make during gameplay. Users of Ludii will generally not need to interact with these low-level objects directly.

Move `Move` objects are wrappers around one or more `Action` objects. Sometimes they may even contain references to additional rules that should be executed to compute additional Actions to apply after the Actions that it directly contains have been applied to a game state. Moves correspond to the decisions that players can actually directly make when playing.

CHAPTER 6

Programmatically Loading Games

6.1 Loading a Game by Name

Ludii's `player.utils.loading.GameLoader` class provides static helper methods that may be used to programmatically load games. The simplest such method only takes a single argument; a `String` representing the name of a game. This argument should always include a `.lud` extension, and at least the filename of the game to load. Note that this can only be used to load games that are built into the `Ludii.jar` file, and not for loading games from external `.lud` files. It may be called as follows:

```
final Game ticTacToe = GameLoader.loadGameFromName("Tic-Tac-Toe.lud");
final Game chess = GameLoader.loadGameFromName("/Chess.lud");
```

It is also allowed to prepend any part of the "folder structure" under which the `.lud` file is stored inside `Ludii.jar`, starting from the top-level `/lud/` folder. Normally Ludii should be smart enough to know which game you wish to load as long as the full filename (without folders) is provided, so this should normally not be necessary. For example, it knows that `Chess.lud` refers to the game of *Chess*, even though that name could also be a match for other games such as *Double Chess.lud*. However, to avoid any risk of ambiguities, it can be useful to include a part of the folder structure (or even just a single `/`, as in the second line of the example code above) in the provided name.

6.2 Listing all Built-in Ludii Games

A list of names for **all** built-in games in your copy of `Ludii.jar`, all of which may be used in `GameLoader.loadGameFromName(...)` calls, can be obtained using the following code:

```
final String[] allGameNames = FileHandling.listGames();
```

This produces an array of Strings that looks as follows:

```
/lud/board/hunt/Adugo.lud
/lud/board/hunt/Baghchal.lud
```

(continues on next page)

(continued from previous page)

```
/lud/board/hunt/Cercar La Liebre.lud
...
```

Note: On some operating systems, the very first symbol in every String in this array may be a backslash instead of a forward slash. They may be freely replaced by forward slashes in game loading calls, and they should still load correctly.

More advanced code to filter this list of games based on their properties is provided in `ListLudiiGames.java`.

6.3 Loading a Game from File

The `GameLoader.loadGameFromName()` method can only be used to load built-in games that ship with Ludii. Programmatically loading games from other files (such as any games you may have implemented yourself!) can be loaded using a similar `GameLoader.loadGameFromFile()` method, which takes a `File` object as argument instead of a `String`. An example, which loads the `.lud` file that we created at the end of *Writing Amazons in .lud Format*, is provided by the following code:

```
final Game ourOwnAmazons = GameLoader.loadGameFromFile(new File("resources/luds/
↪walkthrough_amazons/Step7.lud"));
```

6.4 Loading Games with Options

All of the examples discussed above load the default variants of the respective games. For each of the `GameLoader` methods described above, there is also a version that additionally takes a `List<String>` object as second argument. Whenever an empty list is provided, such a call will be identical to the calls without this argument, simply causing a game with its default *Options* to be loaded. If the list is not empty, Ludii will try to interpret each of the provided Strings as a description of an *Option* to be loaded (instead of the default option).

Note: If you try to load a game with options that are not defined for that game, Ludii will throw an exception.

By default, *Hex* in Ludii is played on an `11x11` board. The following code shows how to load a different variant of *Hex*, by using two non-default options; we play on a `19x19` board, and we invert the winning condition by selecting the "Misere" end rule:

```
final List<String> options = Arrays.asList("Board Size/19x19", "End Rules/Misere");
final Game hex = GameLoader.loadGameFromName("Hex.lud", options);
System.out.println("Num sites on board = " + hex.board().numSites());
```

In this code, the last line is used to verify that we did indeed correctly load a board of size `19x19` instead of the default `11x11` board; it prints that we have `361` sites on the board, which is correct! The `11x11` board would only have `121` sites.

Note: This tutorial uses example code from the following source files:

- GameLoading.java.
- ListLudiiGames.java.

CHAPTER 7

Running Trials

In this tutorial, we look at how to run Ludii trials programmatically. This is one of the core parts of Ludii that will be of interest to practically any programmatic user of Ludii.

Note: This tutorial uses example code from the following source file:

- RunningTrials.java.

In this tutorial, we'll run trials for the game of *Hex*. So, let's load that game first, based on the tutorial on *Programmatically Loading Games*. We only need to do this a single time, and can re-use the resulting `Game` object for multiple trials (assuming we want to play multiple trials of the same game of course):

```
final Game game = GameLoader.loadGameFromName("Hex.lud");
```

Now we'll construct `Trial` and `Context` objects (refer back to *Ludii Programming Terminology* for what these mean). For this tutorial, it is sufficient to only instantiate one of each, because we *re-use* them by resetting their data whenever we're finished with one trial and ready to start the next one.

```
final Trial trial = new Trial(game);
final Context context = new Context(game, trial);
```

Running trials also requires `AI` objects, which select moves during the trials. In this tutorial, we use `RandomAI` objects because they are very fast. Ludii uses 1-based indexing for anything related to players. Therefore, we first insert a `null` entry in the list of `AI` objects that we create:

```
final List<AI> ais = new ArrayList<AI>();
ais.add(null);
for (int p = 1; p <= game.players().count(); ++p)
{
   ais.add(new RandomAI());
}
```

Finally, we implement the main loop that executes multiple trials (played by our random AIs), and inspects the rankings achieved at the end of every trial:

```
1  for (int i = 0; i < NUM_TRIALS; ++i)
2  {
3      game.start(context);
4
5      for (int p = 1; p <= game.players().count(); ++p)
6      {
7          ais.get(p).initAI(game, p);
8      }
9
10     final Model model = context.model();
11
12     while (!trial.over())
13     {
14         model.startNewStep(context, ais, 1.0);
15     }
16
17     final double[] ranking = trial.ranking();
18     for (int p = 1; p <= game.players().count(); ++p)
19     {
20         System.out.println("Agent " + context.state().playerToAgent(p) + " achieved␣
   ↪rank: " + ranking[p]);
21     }
22 }
```

In line 3, we start the new trial. This call resets any data from any previous trials in the `Context` and `Trial` objects, and should always be called before starting a new trial.

In lines 5-8, we allow our `AI` objects to perform any initialisation for the game. In this tutorial this would technically not be necessary, because Ludii's built-in `RandomAI` does not actually require any initialisation. But it is good practice to run this code before starting any new trial, because some algorithms may require initialisation.

In line 10, we obtain a `Model` to play this trial. This may be understood as an object that handles the "control flow" of a trial for us; it has different implementations for alternating-move games than for simultaneous-move games. By using this object, it is possible to run trials of either of those types of games using the same code.

In line 12, we keep looping until the trial is over (i.e. until a terminal game state has been reached).

Line 14 performs most of the work involved in running a trial. It checks which player(s) is/are to move, requests the corresponding `AI` objects to select their moves, and applies them to the game. In an alternating-move game, this call applies a single move to the game (selected by the current mover). In a simultaneous-move game, this call requests moves from all active players, and applies them as one large "combined move". The code used in this tutorial is the simplest version of the `startNewStep()` method. The final `1.0` argument denotes the amount of "thinking time" for AIs, in seconds. There are also more complex versions of the method that allow the user to assign iteration or search depth limits to AIs, or even control whether this method should return immediately and run in a background thread. By default, it blocks and only returns when any moves have been applied.

Finally, line 17 obtains the rankings of all the players, and lines 18-21 prints them. Note that rankings returned by the call in line 17 are indexed by "player indices", which refer to the "colours" of players in a game. In most games these indices will also continue to correspond to the indices for the list of `AI` objects, but in games that use the "Swap rule" this may not be the case. Before swapping, the default colours in *Hex* are red for Player 1, and blue for Player 2, which are controlled by the `AI` objects at indices 1 and 2, respectively. After swapping, the "player indices" remain unchanged. This means that even after swapping, Player 1 will still be red, and if the red player won, `ranking[1]` will return `1.0` (for the first rank). However, *Player 1* will after a swap be controlled by *Agent 2*, and the correct index to use in arrays such as the `ranking` array can be obtained using `context.state().playerToAgent(p)`.

CHAPTER 8

Ludii Programming Cheat Sheet

This page provides a cheat sheet of methods in Ludii that programmatic users (such as AI developers) are likely to require. On this page, we assume that you will at least have access to a `context` object of the type `Context`. Such an object is typically passed around as a wrapper around the "current game state", or can be instantiated by yourself as described on the *Running Trials* page.

By convention, we describe methods that should be called on `Context` objects as `context.method()`, methods that should be called on `Game` objects as `game.method()`, methods that should be called on `State` objects as `state.method()`, and methods that should be called on `Trial` objects as `trial.method()`. Note that references to `Game`, `State`, or `Trial` objects can always be obtained through `Context` objects.

8.1 Game methods

`game.start(final Context context)` Resets given `Context` object and starts it (applying any start rules to generate an initial game state).

`game.moves(final Context context).moves()` Returns the list of legal moves for the current game state in the given `Context` object.

- Will have to compute them on first call, but will immediately return the list on subsequent calls (until a move is applied to modify the game state, after which it will be necessary to re-compute).
- **Warning**: do **not** modify the returned list! Copy it first.

`game.apply(final Context context, final Move move)` Applies the given move to the current game state, causing a transition into a new state.

`game.players().count()` Returns the number of players for this game.

8.2 Context methods

`context.game()` Returns a reference to a `Game` object.

- The `Game` object encapsulates the rules of a game, i.e. *how* a game is played (and with what equipment). For example, this may represent "the game of Chess," as opposed to "this particular game of Chess played by Eric and Matthew."
- A single reference can safely be used across many trials running in parallel.

`context.trial()` Returns a reference to a `Trial` object.

- Wrapper around the history of moves that have been played so far, from initial till current game state.

`context.state()` Returns a reference to a `State` object.

- Represents the current game state.

`new Context(final Context other)` Returns a deep copy of the given `other` object.

- Copy will have a different internal state for Random Number Generation, so any future stochastic events may play out differently for the copy than the original.

`new TempContext(final Context other)` Returns a copy of the given `other` object for temporary use.

- Modifications to the copy will not leak back into the original.
- Modifications to the original **can** leak back into previously instantiated `TempContext` objects, or even corrupt them.
- Can be more efficient than proper `Context` copies, when only temporarily required and discarded after use (before any new moves are applied to the original).

`context.active(final int who)` Returns whether the given player is still active.

- Did not already win or lose or tie or otherwise stop playing.

8.3 Trial methods

`trial.over()` Returns whether or not the trial is over (i.e., a terminal game state reached with no active players).

`trial.ranking()` Returns an array of rankings, with one value per player.

- First valid index is `1`, for the first player.
- A rank value of `1.0` is the best possible value, and a rank value of `K` is the worst possible value (for a game with `K` players).
- Entries for players that are still active are always `0.0`.

`trial.reverseMoveIterator()` Returns an iterator that allows iteration through all the moves that have been applied, in reverse order.

`trial.getMove(final int idx)` Returns the move at the given index.

- If all the last `X` moves are required, using the reverse move iterator can be significantly more efficient.

`trial.moveNumber()` The number of moves that have been played (excluding moves applied by game's start rules to generate initial game state).

`trial.numMoves()` Total number of moves applied (including moves applied by game's start rules to generate initial game state).

`trial.numInitPlacement()` The number of moves applied by the game's start rules to generate initial game state.

8.4 State methods

`state.mover()` Returns the current player to move (only correct for alternating-move games).

`state.playerToAgent(final int playerIdx)` For a given player index (corresponding to a "colour"), returns the index of the "agent" (human or AI) who is currently in control of that player index.

- Usually just returns `playerIdx` again, but can be different in game states where players have swapped colours during gameplay (commonly used in games such as Hex).

`state.owned()` Returns an object of type `Owned`, which is a data structure that stores which positions are occupied by any pieces for any player.

`state.stateHash()` Returns a (Zobrist) hash code for the state that only accounts for a limited number of state variables (intuitively: only for elements that can be visibly seen on the board, i.e. which pieces are where).

`state.fullHash()` Returns a (Zobrist) hash code for the state that accounts for (almost) all possibly-relevant state variables.

CHAPTER 9

Basic API for AI Development

Note: This tutorial expects AIs for Ludii to be implemented in Java. For experimental support for Python-based implementations, see https://github.com/Ludeme/LudiiPythonAI.

Ludii expects custom AIs to be written in Java, and extend the abstract `util.AI` class. This tutorial describes the basic functions that are likely to be useful to override. AIs implemented according to this tutorial can be loaded and used to play games in the Ludii app the following instructions from the Ludii Example AI repository.

9.1 Selecting Actions

The most important method for custom AIs, which must always be overridden, has the following signature:

```
public abstract Move selectAction
(
   final Game game,
   final Context context,
   final double maxSeconds,
   final int maxIterations,
   final int maxDepth
);
```

This method takes the following parameters:

- `game`: A reference to the game we're playing.
- `context`: A copy of the `Context` that we're currently in (see *Ludii Programming Terminology* for what a `Context` is). This also contains the game state in which we're expected to make a move.
- `maxSeconds`: The maximum number of seconds, after which the AI is expected to return a selected move. Ludii does not generally enforce this limit, though it will of course be enforced in competition settings.
- `maxIterations`: The maximum number of "iterations" that the AI is allowed to use, before it should return its moves. Here, we do not have a strict definition of what "iterations" should mean. Ludii does not ever enforce

this limit. It will mostly be of interest for AI researchers. For example, we use this ourselves in some research papers, where we restrict multiple different MCTS agents to a fixed MCTS iteration count, rather than a time limit.

- `maxDepth`: The maximum depth that an AI is allowed to search, before it should return its move. Here, we do not have a strict definition of what "iterations" should mean. Ludii does not ever enforce this limit. It will mostly be of interest for AI researchers.

The method should be implemented to return a `Move` object that the agent wishes to be applied. A full example of how this method is implemented by the Example Random AI is shown below:

```
@Override
public Move selectAction
(
   final Game game,
   final Context context,
   final double maxSeconds,
   final int maxIterations,
   final int maxDepth
)
{
   FastArrayList<Move> legalMoves = game.moves(context).moves();

   if (legalMoves.isEmpty())
      return Game.createPassMove(context);

   // If we're playing a simultaneous-move game, some of the legal moves may be
   // for different players. Extract only the ones that we can choose.
   if (!game.isAlternatingMoveGame())
      legalMoves = AIUtils.extractMovesForMover(legalMoves, player);

   final int r = ThreadLocalRandom.current().nextInt(legalMoves.size());
   return legalMoves.get(r);
}
```

9.2 Initialisation and Cleanup

Ludii's abstract `AI` class has two methods, with default empty implementations, to perform initialisation and cleanup. These may be overwritten for agents if it is necessary to perform initialisation steps before starting to play (for instance to load data from files), or to perform cleanup after finishing a game:

```
public void initAI(final Game game, final int playerID){}
public void closeAI(){}
```

The `initAI()` method also tells the AI which player it is expected to start playing as in the upcoming trial. This is generally not important for AIs for alternating move-games – since they can always figure out who the current mover is directly from the state for which they're asked to make a move – but it is important for AIs that support simultaneous-move games. They can memorise this argument and know that that is the player for which they should return moves. This is why the Example Random AI has the following implementation:

```
@Override
public void initAI(final Game game, final int playerID)
{
   this.player = playerID;
}
```

For AIs loaded inside the Ludii app, it is always guaranteed that `initAI()` will be called at least once before an AI is requested to make a move in a given trial. Note that it is possible that the method will be called much more frequently than that (for instance if the user starts jumping back and forth through a trial). For programmers implementing their own experiments, it is important that they remember to call this method themselves, as shown in *Running Trials*. Similarly, Ludii will try to call `closeAI()` to allow for cleanup when possible, but AIs should not rely on this for them to function correctly.

Note: Examples of full AI implementations can be found in the Ludii Example AI repository on GitHub.

CHAPTER 10

Contact Info

1. For questions or suggestions directly related to these tutorial pages, please create an issue on GitHub.
2. For other questions, suggestions or remarks, it is preferred to use the Ludii Forums.
3. Alternatively, we may be contacted via `ludii.games@gmail.com`.

CHAPTER 11

Acknowledgements

This repository is part of the European Research Council-funded Digital Ludeme Project (ERC Consolidator Grant #771292), being run by Cameron Browne at Maastricht University's Department of Advanced Computing Sciences.

CHAPTER 12

Citing Ludii

The following `.bib` entry may be used for citing the use of Ludii in papers:

```
@inproceedings{Piette2020Ludii,
  author = "{\'E}. Piette and D. J. N. J. Soemers and M. Stephenson and C. F. Sironi␣
↪and M. H. M. Winands and C. Browne",
  booktitle = "Proceedings of the 24th European Conference on Artificial Intelligence␣
↪(ECAI 2020)",
  title = "Ludii -- The Ludemic General Game System",
  pages = "411-418",
  year = "2020",
  editor = "G. De Giacomo and A. Catala and B. Dilkina and M. Milano and S. Barro and␣
↪A. Bugarín and J. Lang",
  series = "Frontiers in Artificial Intelligence and Applications",
  volume = "325",
  publisher = "IOS Press"
}
```