人狼知能エージェント作成の手引き

はじめに

この手引きは**Java**でエージェントを作成する人向けのものです．Python，C#エージェントを作成する方は，各自調べてください．

開発環境を整える

Eclipseのインストール

Javaエージェントの開発には，Eclipseを使用します．Jarファイルの出力が簡単であったり，補完機能の優秀さ，ライブラリの導入のしやすさなどのメリットがあるためです．

ここでは，Pleiadesプラグインが導入されたEclipseである"Pleiades All in One"を使用します．

> Pleiades All in One は Windows、Mac 向けに Eclipse 本体と日本語化を行うための Pleiades プラグインおよびプログラミング言語別に便利なプラグインをまとめたパッケージです。また、Full Edition の場合は JDK のインストールや環境変数の設定が不要で Eclipse の各種設定も自動で行われるため、ダウンロードして起動すれば、すぐに日本語化された Eclipse を利用できます。(公式サイトより引用)

- Pleiades All in One ダウンロードページリンク

32bit版しか動かないなど，特別な理由がない限りは最新版を使用して貰えば問題はありません．OSにあったJavaのFullEditionをダウンロード・インストールしましょう．ここでは，Macを使用しているものとして説明を進めていきます．

インストール後，起動すると"ワークスペースとしてのディレクトリー選択"という画面が表示されます．デフォルトの `../workspace` はアプリ内のディレクトリですので，任意の場所を指定しておくと，管理が楽になります．(Eclipseを消してもソースは残る・Eclipseの新バージョンを入れたときの移行作業が楽など．)

コンテンツ・アシスト(コード補完)の設定

デフォルトでは，`.` を押さないとコード補完が働かないようになっているので，入力の最中に補完候補が出力されるように設定を変更します．

1. 設定(Preferences)(`⌘cmd + ,`)を開き，フィルターを入力に"Java"と入力
2. Java > エディター > コンテンツ・アシスト を開く
3. 表示された内容の一番下に"自動有効化"の項目があることを確認する
4. Javaの自動有効化トリガー: に指定されている `.` を以下で置き換える
   `.abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ`
5. 適用して閉じる で設定を閉じる

人狼知能プラットフォームのダウンロード

プラットフォーム本体とJavaDocをそれぞれダウンロードします．下位互換はありませんので，できるだけ最新のものを使います．

- 人狼知能プラットフォーム ダウンロードページリンク

2020/05/01現在，プラットフォームの最新バージョンは0.6.0，JavaDocの最新バージョンは0.5.2のものです.

プラットフォームのzipは展開しておいてください．JavaDocはzipファイルのままで大丈夫です.

プロジェクトの作成

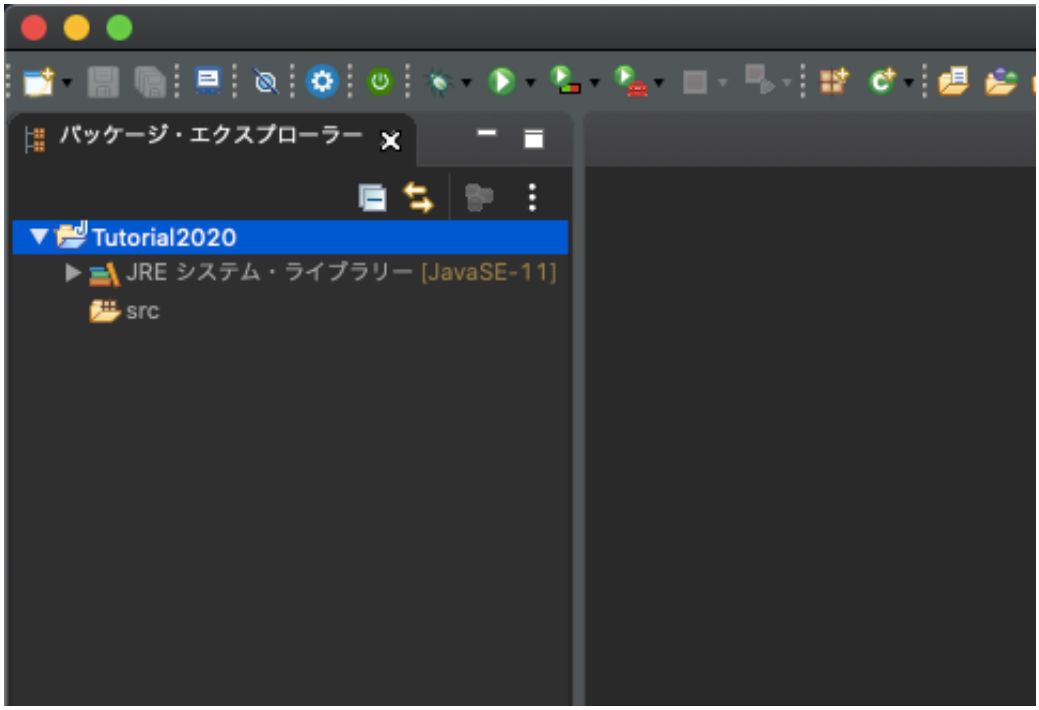
人狼知能エージェントを作成するためのプロジェクトを作成します.

新規Javaプロジェクトの作成

1. Eclipseを起動し，メニューバーから，ファイル > 新規 > Javaプロジェクト
2. プロジェクト名は任意の名前にする．説明の便宜上，ここでは"Tutorial2020"とする.
3. JREは "実行環境JREの使用: JavaSE-11" を選択する
4. プロジェクト・レイアウトは"ソースおよびクラス・ファイルのフォルダーを個別に作成"になっていることを確認する
5. ワーキング・セットにチェックが付いていないことを確認する
6. 次へ をおす
7. "ソース"タブ内の"module-info.javaファイルの作成"のチェックを外す
8. 完了 をおす

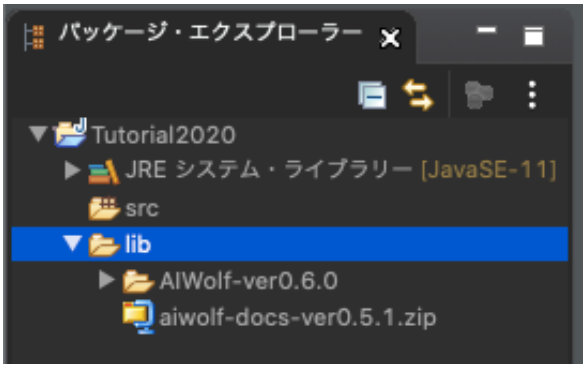
プロジェクトにライブラリを追加する

1. パッケージ・エクスプローラに表示されているプロジェクトを右クリック （以降，『プロジェクトを右クリック』はこの操作を指します）

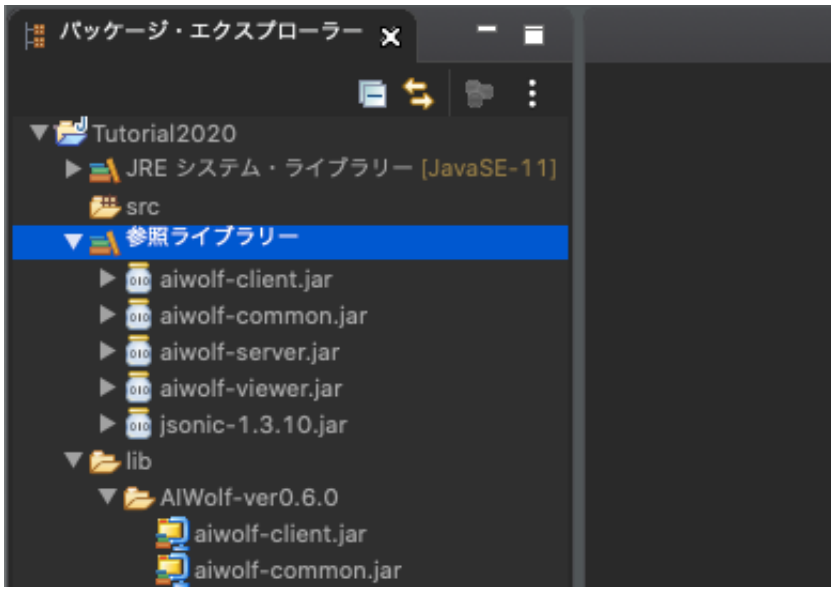


2. 表示されたメニューから 新規 > フォルダ
3. 親フォルダーを入力または選択 に該当するプロジェクト名が入力されていることを確認する
4. フォルダ名 に"lib"と入力し，完了をおす
5. ダウンロードしておいたJavaDocのzipファイルと，展開しておいたプラットフォームのフォル

ダをコピーして，プロジェクトのlibの中に貼り付ける．ここまで完了すると，以下の画像のようになる



6. lib > AIWolf-ver0.6.0 の中から，以下の5つのjarファイルを選択する (⌘cmdを押しながら選択することで，複数項目選択できる)
    - aiwolf-client.jar
    - aiwolf-common.jar
    - aiwolf-server.jar
    - aiwolf-viewer.jar
    - jsonic-1.3.10.jar
7. 選択項目を右クリックし，ビルド・パス > ビルド・パスに追加 を選択．参照ライブラリーが追加され，先程の5つのjarが入っていればOK.



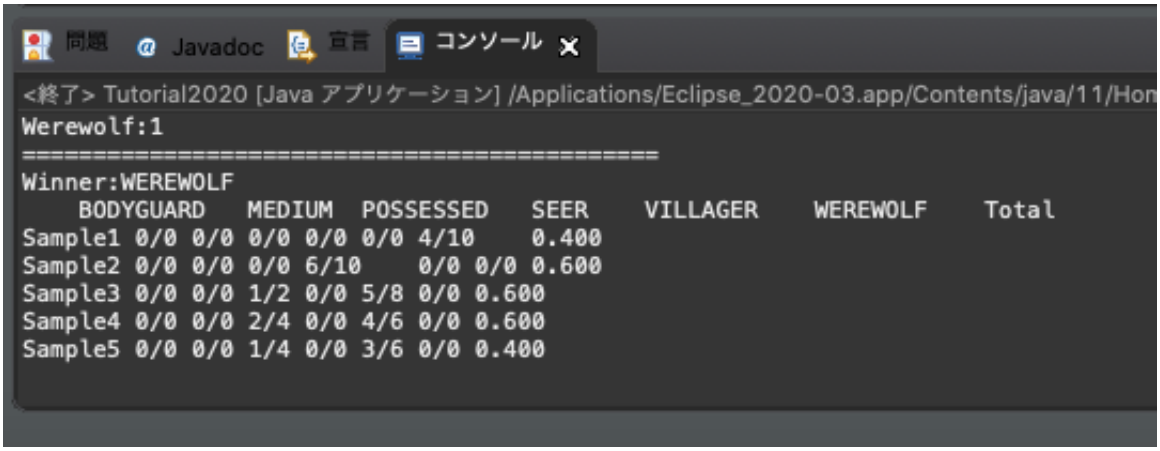
8. 参照ライブラリーを右クリックし，ビルド・パス > ビルド・パスの構成 を選択.
9. クラスパス の折りたたみ内にある，aiwolf-common.jarの折りたたみを展開し，"Javadocロケーション:(なし)" をダブルクリック.
10. 表示された項目で，"アーカイブ内のJavadoc"を選択し，"ワークスペース・ファイル"を選択し，アーカイブ・パス横の参照をおす
11. プロジェクトのlib内にある，JavaDocのzipファイルを選択し，OKを押す
12. アーカイブ内のパスに"docs"と入力し，検証を押す．「ロケーションは有効のようです」と表示されたら，そのウィンドウはOKで閉じる．その後，OKや適用して閉じるを押して，メインの画面まで戻ってくる.

設定ファイルの書き換え / 動作確認

サンプルエージェントを動かして，動作確認を行います．

1. lib > AIWolf-ver0.6.0 > AutoStarter.ini を開く
2. `setting=./SampleSetting.cfg` を `setting=./lib/AIWolf-ver0.6.0/SampleSetting.cfg` に書き換えて保存
3. Eclipse上部にあるアイコンたちの中から，実行の横の下三角を押す．

4. 実行の構成を選択
5. Javaアプリケーションの上で 右クリック > 新規構成． フィルターで絞り込んだ状態だと，作成した構成が表示されないことがあるので，ここではフィルターの使用はしない方が良い．
6. 名前は任意のものに変えておく．プロジェクト名や起動方法に関する名前を組み合わせてつけておくと便利である．ここでは，"Tutorial2020"とする．
7. メイン・クラスの検索を押し，"AutoStarter - org.aiwolf.ui.bin"を選択し，OKを押す
8. 引数のタブに移動し，プログラムの引数に以下を入力する `./lib/AIWolf-ver0.6.0/AutoStarter.ini`
9. 適用 を押した後に，実行を押す
10. コンソールタブにゲームのログが表示されたら，正しく設定が終わっている．実行時に最初に表示されるエラーは仕様なので無視して構わない．



人狼知能の作成に入る前に

授業でJavaは習っていると思います．基本的な文法については省略しますので，わからないところがあれば各自Google先生に聞いてください． ここでは，人狼知能の作成にあたって知っておいた方がいいものを一部抜粋して紹介します．

ArrayList

```
List<E> list = new ArrayList<E>();
```

`E` には任意の型が入ります．例： `List<String> list = new ArrayList<String>();` ．

ArrayListは要素を追加した順番が保証されているリストです． `list.add("xxx");` を繰り返すたびに，リストの最後列に追加されていきます．順番が大事になるリスト，例えば占った対象のエージェントを記録しておくためのリスト(先頭から1日目に占った，2日目に占った…と記録できる)などに使えます．順番が必要ではないリストでも，ArrayListを使って損はないので，よく使われるListです．

HashMap

```
Map<K, V> map = new HashMap<K, V>();
```

`K` , `V` には任意の型が入ります．例： `Map<String, Integer> map = new HashMap<String, Integer>();` ．

HashMapは重複しない `K` をキーとして値 `V` を記録する，他言語ではディクショナリ型・辞書型などと呼ばれたりもするものです．人狼知能では，エージェントに対して紐づけておきたい情報がある時に使用することが多いです．例えば，エージェント①は占い師とカミングアウトしている，といった情報を保存するときなどに使います． `K` (キー)の重複は許されていないため，もし同じ `K` で `V` (値)を保存しようとすると，元々保存されていた `K` に対する `V` が上書きされます．これは，情報の更新が容易であると言う点では，利点と捉えることができます．

人狼知能プロジェクトのクラス・列挙型

これらを全て説明しようとすると，膨大な量になってしまいますので，他のドキュメントを読んでいただくことで説明の代わりとしたいと思います． 人狼知能プラットフォームのダウンロード でダウンロードしてきたJavaDocですが，展開せずそのままzipファイルで扱っていました．このzipを展開して，中の `index.html` を開いてみてください．人狼知能の基本的なクラスや列挙型の説明を，htmlで見ることができます．時間のある時に一通り目を通していただれば．人狼知能プラットフォームで扱うデータにはどのようなものがあるのか，わかっていただけると思います．

とりあえず，優先的に見てもらいたいのは **org.aiwolf.common.data** のページです．ここに **Agent** クラスとはどのようなものか， **Role** 列挙型とはどのようなものかなどが書いてあります．作成しながら読むでも問題ありませんので，JavaDocの中を見れば説明書きがあるということを覚えておいてください．

インターフェース・実装

インターフェースは，メソッドの仕様のみを定義したものであり，実際の処理は書いていないものです．このインターフェースをもとにクラスを作成することを **実装** と言います．人狼知能で使われているインターフェースを例に挙げます．

```
public interface Player {

  /** プレイヤーの名前を返します。*/
  String getName();

  /** ゲームの情報が更新されたときに呼び出されます。*/
  void update(GameInfo gameInfo);
```

```
  /** ゲーム開始時に呼び出されます。*/
  void initialize(GameInfo gameInfo, GameSetting gameSetting);

  /** 1日の始まりに呼び出されます。*/
  void dayStart();

  /**
   * 各プレイヤーはこのメソッドに意見を返すことで会話することができます。
   * 戻り値の文字列は人狼知能プロトコルに沿ったものにする必要があります。 nullを返した場合は、SKIPとみなされます。
   */
  String talk();

  /** 各人狼はこのメソッドに意見を返すことで人狼同士での会話ができます。 */
  String whisper();

  /** 追放したいエージェントに投票します。 */
  Agent vote();

  /** 人狼によって、襲撃したいエージェントを決めます。*/
  Agent attack();

  /** 占い師によって、占うエージェントを決めます。*/
  Agent divine();

  /** 狩人によって、護衛するエージェントを決めます。</div> */
  Agent guard();

  /**
   * ゲーム終了時に呼び出されます。
   * このメソッドが呼び出される前に、ゲームの情報のすべての情報は更新されます。
   */
  void finish();
}
```

人狼知能のエージェントは，このPlayerインターフェースを実装して作成します．インターフェースで必須のメソッドの名前・引数・戻り値が決められているので，Playerを実装したクラスは全てPlayerとして扱うことができます．例えば以下のように2つのクラスがあるとします．

```
class AgentA implements Player {
  (省略)
}
```

```
class AgentB implements Player {
  (省略)
}
```

`implements` は実装するインターフェースを指定する時に使うものです． 本来，これらのクラスを呼び出す時は， `AgentA A = new AgentA();` や `AgentB B = new AgentB();` としますが，どちらもPlayerを実装して作成しているので， `Player A = new AgentA();` と `Player B = new AgentB();` とすることができます．これにより，全てのエージェントを扱いやすくしているわけです．

継承

継承は，既にあるクラスを引き継いで新たにクラスを作成するものです．元々あったクラスを親クラス，新たに作成したものを子クラスと呼んだりします．子クラスは親クラスの変数やメソッドも使うことができます．例えば以下のようなクラスがあるとします．

```
class Human {
  String name;
  int age;

  public Human(String name, int age){
    this.name = name;
    this.age = age;
  }

  void printName(){
    System.out.println(name);
  }

  void printAge(){
    System.out.println(age);
  }

  void printAll(){
    System.out.println(name + "," + age);
  }

  void printlnAll(){
    System.out.println(name);
    System.out.println(age);
  }
}
```

Humanクラスは，名前と年齢を引数にして作成ができ，各メソッドで情報を表示できます．このHumanクラス(親クラス)はそのままに，新たに性別も管理できるクラス(子クラス)を作成も作成したいとします．すると以下のように書けます．

```
class HumanGender extends Human {
  String gender;

  public HumanGender(final String name, final int age, final String gender) {
    super(name, age);
    this.gender = gender;
  }
```

```
  void printGender(){
    System.out.println(gender);
  }

  @Override
  void printAll() {
    System.out.println(name + "," + age + "," + gender);
  }

  @Override
  void printlnAll() {
    super.printlnAll();
    System.out.println(gender);
  }
}
```

順番に説明していきます.

- class名の横についている `extends` が継承する元のクラス(親クラス)を指定するためのものです.
- コンストラクタ内の `super()` は親クラスのコンストラクタを呼び出すものです.
- `@Override` は親クラスのメソッドを上書きする指示です. printAllメソッドでは表示する内容を変える必要があるので, `@Override` してメソッドを書き換えています.
- printlnAllメソッドは, 既存の出力の下に, 性別の出力を追加すれば良いので, `super.printlnAll();` と書いています. これは, `@Override` で上書きする前の親クラスのprintlnAllメソッドを実行してくださいと言う意味です. よって, 名前と年齢が表示された後, 性別が表示される処理となります.

子クラスは親クラスのメソッドも変数も使用できるので,

```
HumanGender hg = new HumanGender("Taro", 15, "man");
hg.printName(); //output: Taro
hg.printAge(); // output: 15
hg.printGender(); // output: man
hg.printAll(); //output: Taro,15,man
hg.printlnAll(); /* output: Taro
                            15
                            man
*/
```

とHumanクラスのメソッドも使うことができます.

人狼知能の作成

Eclipseでの人狼知能の作成は, 新規プロジェクトの作成を含めて以下のような流れで行います.

1. 新規プロジェクトの作成・ライブラリの追加
2. パッケージの追加
3. AbstractRoleAssignPlayerクラスを継承したクラスの作成
4. BasePlayerの作成

5. 各役職のクラスを作成

パッケージの追加

Javaは同じ名前のクラスが複数存在すると，間違ったクラスを読み込んでしまいエラーを起こす可能性があります．人狼知能で例えるならば，村人に関する処理が書いてあるクラスに"Villager"と名付けたとします．他のエージェントも同じように村人に関する処理を書いた"Villager"クラスを持っていたら，どちら読み込めば正しく動くのか分からなくなってしまいます．

名前の衝突が起こらないようにクラス名を長く複雑にすればいいわけではありません．ここでパッケージの登場です．XXパッケージのVillagerを読み込む・yyパッケージのVillagerを読み込むとしてあげれば，クラス名は短く・分かりやすいまま，正確にクラスを読み込むことができます．

パッケージ名は，ドメイン名を逆さにしたもの+プロジェクト固有の単語にすることが一般的です．ハイフン `-` はアンダーバー `_` に，変えます．自分のドメインを所有しているなら，それを使って貰えば良いのですが，多くの人は持っていないと思いますので，メールアドレスを逆さにしてパッケージ名にします．

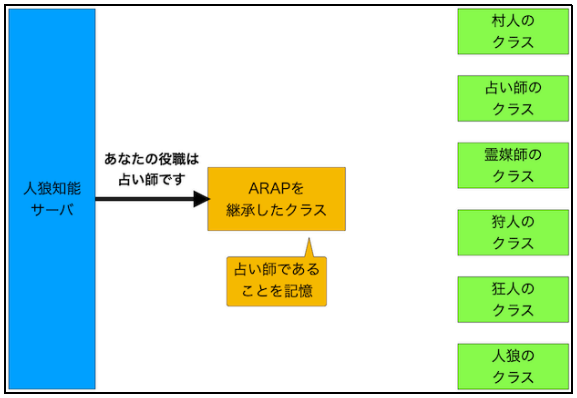
メールアドレスが `k10000kk@aitech.ac.jp` とするとき，パッケージ名は `jp.ac.aitech.k10000kk` となります． `@` も `.` に置き換えましょう．ここにプロジェクト固有の単語を追加します．今回はAIWolfTutorial2020にちなんで `aiwt2020` とします． では先ほど作成したプロジェクトに，パッケージを追加します．

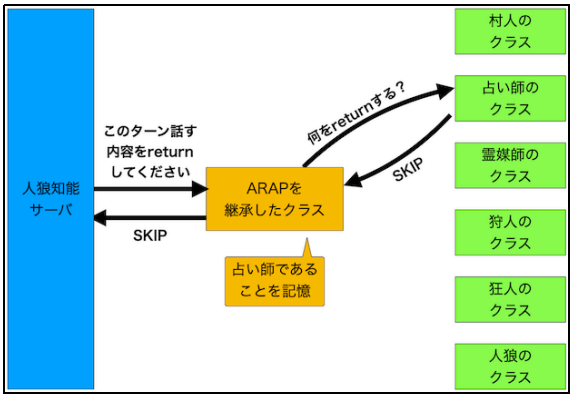
1. パッケージ・エクスプローラーのsrc上で右クリック > 新規 > パッケージ を選択
2. 名前: にパッケージ名を入力します． この資料では `jp.ac.aitech.maslab.aiwt2020` としますが，各自のメールアドレスを逆さにしたもの+ `aiwt2020` としてください． `aiwt2020` の部分も変えていただいて構いません．
3. 完了を押します．

パッケージ・エクスプローラのsrcを右クリック > Finder をして貰えればわかると思いますが，パッケージ名の `.` ごとにフォルダわけされています． この先， プレイヤに関する内容は `jp.ac.aitech.maslab.aiwt202.player` ， よく使う便利な関数群は `jp.ac.aitech.maslab.aiwt2020.util` に作るなど，パッケージを有効活用して整理しましょう．

AbstractRoleAssignPlayerを継承したクラスの作成

人狼知能では一般的に，各役職のクラスを作成して，与えられた役職をもとに，どのクラスを使用するかを決定します． この，どのクラスを使うかを決定してくれるのがAbstractRoleAssignPlayer(ARAP)を継承したクラスです．村人のときはこのクラス，占い師のときはこのクラス，と予め設定しておくことで，与えられた役職のクラスに処理を投げてくれます．





実際にこのクラスを作ってみます.

1. 先ほど作成したパッケージの下に `player` パッケージを追加(しなくても良いが，分かりやすくするため)． パッケージ・エクスプローラの src > jp.ac.aitech.maslab.aiwt2020(各自で作ったパッケージ) を右クリック
2. 新規 > パッケージ
3. 名前の欄に，作成済みのパッケージの名前が表示されているので，後ろに `.player` を追加
   例： `jp.ac.aitech.maslab.aiwt2020.player`
4. 作成したplayerパッケージ上で右クリック > 新規 > クラス
5. パッケージ: の欄が， `[作成してあったパッケージ].player` なっていることを確認
6. 名前: の欄に任意の名前を入力する．今回は `RoleAssignPlayer` とします.
7. スーパークラス : が親クラスを指定する部分． 横の参照を押す
8. 検索欄に"ARAP"と入力して，表示された"AbstractRoleAssignPlayer"を選択して，OKを押す
9. 完了 を押す

出来上がったファイルに対して次の編集を加えます.

1. getNameメソッド内の `return null` のnullをエージェントの名前に変える． (過去の研究室メンバーの名前の一例: Army, Kanpyo, Pocky, daisyo, Tomato, PaSeRi ...)
2. コンストラクタを作成する(中身はあとで追加していくので，今は空で構わない)

今は空のコンストラクタですが，エージェントを作り終えたら，コンストラクタの中で紐付けを行います．以下にその例を示します.

```
public RoleAssignPlayer() {
  setBodyguardPlayer(new BodyGuardPlayer());
  setMediumPlayer(new MediumPlayer());
  setPossessedPlayer(new PossesedPlayer());
  setSeerPlayer(new SeerPlayer());
  setVillagerPlayer(new VillagerPlayer());
  setWerewolfPlayer(new WerewolfPlayer());
}
```

このように，set○○Playerを呼び出して，その中に各役職のクラスのインスタンス(newしたもの)を渡すことで紐づけることができます．今はまだ，どの役職のクラスも作っていないので，空で大丈夫です.

ちなみに，紐づけられたクラスがない場合は，公式サンプルが代わりにゲームをしてくれるので，村人だけできたらなら，村人だけ紐づけて貰えば問題なく動きます.

BasePlayerの作成

人狼知能では，各役職ごとにクラスを作成しますが，どの役職でも行う共通の処理は存在します．例えば，他のエージェントの発話を読み取る作業です．こういった処理を，役職ごとのクラスに毎回書いていくのは，無駄も多く，修正が必要になった時に，全てのクラスに手を加えなければいけなくなります．

そこで，BasePlayerと言うクラスを作成し，そこに共通の処理を書き，各役職のクラスには+αの処理だけ書けば良いといった構成にします．つまり，継承をここで使うのです．

では実際にBasePlayerを作成してみましょう．

1. BasePlayerクラスを作成する
    1. パッケージ・エクスプローラのsrc > 任意のパッケージ名 > player を右クリック
    2. 新規 > クラス
    3. 名前: には `BasePlayer` と入力
    4. インターフェース: 横の追加を押す
    5. 検索欄に "Player"と入力し，表示された "Player - org.aiwolf.common.data ..."を選択してOK
    6. 完了 を押す

BasePlayer.javaが作成され，インターフェース・実装で示したPlayerインターフェースのメソッドが事前に入力されたクラスが出来上がりました．ここに共通の処理を書き込んでいきます．

2. インスタンス変数の作成

    以下のインスタンス変数を追加します．

```
/** このエージェント */
Agent me;

/** 日付 */
int day;

/** 最新のゲーム情報 */
GameInfo currentGameInfo;

/** 自分以外の生存エージェントリスト */
List<Agent> aliveOthers;

/** 追放されたエージェントリスト */
List<Agent> executedAgents = new ArrayList<Agent>();

/** 襲撃されたエージェントリスト */
List<Agent> killedAgents = new ArrayList<Agent>();

/** 発言された占い結果報告リスト */
List<Judge> divinationList = new ArrayList<Judge>();

/** 発言された霊媒結果報告リスト */
List<Judge> identList = new ArrayList<Judge>();

/** 発言用Talkキュー */
Deque<Content> talkQueue = new LinkedList<Content>();
```

```
/** カミングアウト状況 */
Map<Agent, Role> comingoutMap = new HashMap<Agent, Role>();

/** talkListの読み込みヘッド */
int talkListHead;

/** その日投票するターゲット(候補) */
Agent voteCandidate;

/** その日投票するターゲット(宣言済) */
Agent declaredVoteCandidate;
```

Eclipseの機能でimport(使用するパッケージの指定)は自動でおこなわれますが，クラス名の重複があると，補完が働かないことがあります．チュートリアルで作るBasePlayerに使用する全てのパッケージのimport文を以下に示しますので，補完が働かなかった場合は参考にしてください．

```
import java.security.SecureRandom;
import java.util.ArrayList;
import java.util.Deque;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Random;

import org.aiwolf.client.lib.Content;
import org.aiwolf.client.lib.Topic;
import org.aiwolf.common.data.Agent;
import org.aiwolf.common.data.Judge;
import org.aiwolf.common.data.Player;
import org.aiwolf.common.data.Role;
import org.aiwolf.common.data.Status;
import org.aiwolf.common.data.Talk;
import org.aiwolf.common.net.GameInfo;
import org.aiwolf.common.net.GameSetting;
```

3. ユーティリティメソッドを作成する

ユーティリティ(役に立つ)メソッドをいくつか作成しておきます．実装によって作成されたメソッドよりも上にまとめておくと，後から見やすいです．

```
/** エージェントが生存しているかどうかを返す */
boolean isAlive(Agent agent) {
  return currentGameInfo.getStatusMap().get(agent) == Status.ALIVE;
}

/** エージェントが襲撃されたかどうかを返す */
boolean isKilled(Agent agent) {
  return killedAgents.contains(agent);
}

/** エージェントがカミングアウトしているかどうかを返す */
```

```
boolean isCo(Agent agent) {
  return comingoutMap.containsKey(agent);
}

/** 役職がカミングアウトされたかどうかを返す */
boolean isCo(Role role) {
  return comingoutMap.containsValue(role);
}

/** その役職をカミングアウトした人数を返す */
int countCoRole(Role role) {
  int i=0;
  if(!comingoutMap.isEmpty()) {
    for(Role r : comingoutMap.values()) {
      if (r == role) i++;
    }
  }
  return i;
}

/** リストからランダムに選んで返す */
<T> T randomSelect(List<T> list) {
  if(list.isEmpty()) {
    return null;
  } else {
    Random random = new Random();
    return list.get(random.nextInt(list.size()));
  }
}

/** 追放されたエージェントを生存者リストから削除し，追放リストに追加する */
private void addExecutedAgent(Agent agent) {
  if(agent != null) {
    aliveOthers.remove(agent);
    if(!executedAgents.contains(agent)) {
      executedAgents.add(agent);
    }
  }
}

/** 襲撃されたエージェントを生存者リストから削除し，襲撃リストに追加する */
private void addKilledAgent(Agent agent) {
  if(agent != null) {
    aliveOthers.remove(agent);
    if(!killedAgents.contains(agent)) {
      killedAgents.add(agent);
    }
  }
}

/**
 * 主語を省略されたtalkを扱えるようにするため，主語を発話者に置き換えるメソッド
```

```
   * @param content
   * @param newSubject
   */
  static Content replaceSubject(Content content, Agent newSubject) {
    if (content.getTopic() == Topic.SKIP || content.getTopic() == Topic.OVER) {
      return content;
    }
    if (newSubject == Content.UNSPEC) {
      return new Content(Content.stripSubject(content.getText()));
    } else {
      return new Content(newSubject + " " +
 Content.stripSubject(content.getText()));
    }
  }
```

4. initializeメソッドの実装

initializeメソッドは，初期化メソッドになります．前回のゲームで使用していたデータを，今回の試合に持ち込まないように，各変数などを初期化する作業などを行います． 今回は，以下のように各種変数・List・Mapを初期化します．

```
  @Override
  public void initialize(GameInfo arg0, GameSetting arg1) {
    // 手元のゲーム情報を最新のもので初期化
    currentGameInfo = arg0;
    // 人狼知能プラットフォームには0日目が存在するので，初期化は-1
    day = -1;
    // 自分の情報をgameInfoの情報で初期化
    me = currentGameInfo.getAgent();
    // 生存者リストをgameInfoから複製し，自分をリストから削除
    aliveOthers = new ArrayList<>(currentGameInfo.getAliveAgentList());
    aliveOthers.remove(me);
    // 追放リストはclearで初期化
    executedAgents.clear();
    // 襲撃リストはclearで初期化
    killedAgents.clear();
    // 占いリストはclearで初期化
    divinationList.clear();
    // 霊媒リストはclearで初期化
    identList.clear();
    // カミングアウトマップはclearで初期化
    comingoutMap.clear();
  }
```

5. dayStartメソッドの実装

dayStartメソッドは，日ごとに更新したい情報についてや，初期化したいものについて実装するメソッドです．その日の会話が始まる前に1度だけ呼ばれます．

今回は，日ごとに初期化したいリストなどについての処理と，前日に追放・襲撃されたプレイヤについての情報の処理を実装します．

```
  @Override
  public void dayStart() {
```

```
    // 発言用キューの初期化(前日に発言できずに残ってしまった内容を間違って話さないように)
    talkQueue.clear();
    // talkListはその日に話された内容しか入っていないので，ヘッドを0に戻す
    talkListHead = 0;
    // 誰に投票するかの変数もnullで初期化
    voteCandidate = null;
    // 宣言済みのほうもnullで初期化
    declaredVoteCandidate = null;
    // 昨日追放されたエージェントをリストに追加
    addExecutedAgent(currentGameInfo.getExecutedAgent());
    // 昨日襲撃されたエージェントをリストに追加
    if(!currentGameInfo.getLastDeadAgentList().isEmpty()) {
      addKilledAgent(currentGameInfo.getLastDeadAgentList().get(0));
    }
  }
```

6. updateメソッドの実装

   updateメソッドは，最新のゲーム情報(人狼知能をJavaで書く場合は，GameInfoと呼ぶ)を受け取るメソッドであり，initializeを除く他のメソッドの前に呼ばれます．GameInfoから得られる情報は，**人狼知能プロジェクトのクラス・列挙型** を参考に，org.aiwolf.common.netにあるGameInfoクラスのページを見てください．

   今回は，updateメソッドで得られる最新のゲーム情報から，日付の更新・今日追放されたエージェントについて更新(村人や狂人以外は，能力行使の前にその情報が欲しい為)・各エージェントの発話について更新，の大きく分けて3つを実装します．

   - 日付の更新 dayStartの前にも呼び出されるので，その時は処理を実行しないように組んである．

     ```
     @Override
     public void update(GameInfo gameInfo) {
       // 手元のゲーム情報を最新のもので更新
       currentGameInfo = gameInfo;
       // 1日の最初の呼び出しは，dayStart()メソッドの前なので，何もしない
       if(currentGameInfo.getDay() == day + 1) {
         day = currentGameInfo.getDay();
         return;
       }

       /** この先にまだ追記します **/
     }
     ```

   - 今日追放されたエージェントについて更新 dayStartでも追放されたエージェントについての更新をしました．なぜここにも記述するかですが，直前に追放されたエージェントの情報は，占い対象や襲撃対象・護衛対象などを決める時に必要になるからです． 今日追放されたエージェントは `GameInfo.getLatestExecutedAgent();` ， 前日追放されたエージェントは `GameInfo.getExecutedAgent();` と違うメソッドから取得するので，一応dayStartでもupdateでも情報更新を行っています．しかし，今日追放されたエージェントによって情報を更新した場合，前日追放されたエージェントでも情報の更新を行うと，内容の重複が起こってしまうので，その辺りは `addExecutedAgent()` ユーティリティメソッドでうまいことやっています．

```
@Override
public void update(GameInfo gameInfo) {
  /** 日付の更新についての処理がここに書かれています **/

  // 2回目以降の呼び出し
  // (夜限定) 追放されたエージェントを登録
  addExecutedAgent(currentGameInfo.getLatestExecutedAgent());

  /**この先にまだ追記します **/
}
```

- 各エージェントの発話についての処理

  各エージェントの発話は，`GameInfo.getTalkList();`で取得できるリストに，発言された順番に入っています．その日の発話全てが入っており，前日のものは入っていません．よって，リストの何番目以降が，このターンに発話された内容なのかを知っておく必要があります．それが`int talkListHead`です．まず，talkListHeadを使ってリストをfor文で読み取ります．その後，talkLIstHeadの値を更新します．

```
@Override
public void update(GameInfo gameInfo) {
  /** 日付の更新についての処理がここに書かれています **/

  /** 今日追放されたエージェントについて更新についての処理がここに書かれています **/

  // talkListから，カミングアウト・占い報告・霊媒報告を抽出
  for(int i=talkListHead; i<currentGameInfo.getTalkList().size(); i++) {

    /** この中にまだ追記します **/

  }
  // ヘッドを上書きする
  talkListHead = currentGameInfo.getTalkList().size();
}
```

  そして，for文の中でtalkListの中からtalkを取得し，それについて処理を行います．

```
@Override
public void update(GameInfo gameInfo) {
  /** 日付の更新についての処理がここに書かれています **/

  /** 今日追放されたエージェントについて更新についての処理がここに書かれています **/

  // talkListから，カミングアウト・占い報告・霊媒報告を抽出
  for(int i=talkListHead; i<currentGameInfo.getTalkList().size(); i++) {
    // リストからtalkを取得
    Talk talk = currentGameInfo.getTalkList().get(i);
    // 発話者を取得
    Agent talker = talk.getAgent();
    // 発話者が自分なら処理しない
    if(talker == me) {
      continue;
```

```
      }
      // 扱いやすいようにContent型にする
      Content content = new Content(talk.getText());
      // subject(主語)がUNSPEC(未特定)の場合は，発話者に入れ替える
      if(content.getSubject() == Content.UNSPEC) {
        content = replaceSubject(content, talker);
      }

      /** この中にまだ追記します **/
    }
    // ヘッドを上書きする
    talkListHead = currentGameInfo.getTalkList().size();
  }
```

さて，`/** この中にまだ追記します **/`がまだ残っていますが，ここには発話を読み取るためのメソッドを呼び出す文を書きます．先にメソッドを作りましょう． そもそも，なぜ発話を読み取る部分だけ別メソッドにするのか？という点ですが，これは人狼知能プロトコルver3.6を用いて話せる内容にORやANDなどの演算子が含まれることに起因します．例えばORは「Agent[01]は人狼だと思う OR Agent[01]は狂人だと思う」といった使い方ができます．これにより，1度に送られてくる文章に2つ以上の文が含まれている可能性を視野に入れなければなりません．よって，ORやANDなどが含まれている場合は，その前後の文をそれぞれ読み取る必要があります．では，文を解析するメソッドと，演算子を解析するメソッドを一気に作りましょう．updateメソッドの下に作っておくと見やすいでしょう．

```
  @Override
  public void update(GameInfo gameInfo) {
    /** updateメソッドの内容はここでは省略 **/
  }

  // 再帰的に文を解析するメソッド
  void parseSentence(Content content) {
    switch(content.getTopic()) {
    // カミングアウト
    case COMINGOUT:
      comingoutMap.put(content.getTarget(), content.getRole());
      return;
    // 占い報告
    case DIVINED:
      divinationList.add(new Judge(
          day,
          content.getSubject(),
          content.getTarget(),
          content.getResult()
      ));
      return;
    // 霊媒報告
    case IDENTIFIED:
      identList.add(new Judge(
          day,
          content.getSubject(),
          content.getTarget(),
```

```
            content.getResult()
        ));
        return;
      // 演算子(OR,AND,NOT...)
      case OPERATOR:
        parseOperator(content);
        return;
      // それ以外はここでは無視
      default:
        break;
      }
    }

    // 演算子の分析をするメソッド
    void parseOperator(Content content) {
      switch(content.getOperator()) {
      case BECAUSE:
        // [0]に理由，[1]に結論が入っているので，結論だけ処理
        parseSentence(content.getContentList().get(1));
        break;
      case DAY:
        // 特定の日付について言及しているが，内容だけ処理
        parseSentence(content.getContentList().get(0));
        break;
      // AND(全て真)，OR(1つは真)，XOR(どちらかを主張≒どちらかが真)
      case AND:
      case OR:
      case XOR:
        for(Content c : content.getContentList()) {
          parseSentence(c);
        }
        break;
      // それ以外はここでは無視
      default:
        break;
      }
    }
```

複雑なように見えて，していることは至って単純です．parseSenteceメソッドから始まり，演算子で結ばれているなら，parseOperatorメソッドへ処理を移し，そこから含まれている文ごとにparseSenteceメソッドで処理を行うといった流れです．この2つのメソッドを作った上で，updateメソッドのfor文の最後に一行加えて，メソッドの完成です．

```
    @Override
    public void update(GameInfo gameInfo) {
      /** 日付の更新についての処理がここに書かれています **/

      /** 今日追放されたエージェントについて更新についての処理がここに書かれています **/

      // talkListから，カミングアウト・占い報告・霊媒報告を抽出
      for(int i=talkListHead; i<currentGameInfo.getTalkList().size(); i++) {
        // リストからtalkを取得
```

```
      Talk talk = currentGameInfo.getTalkList().get(i);
      // 発話者を取得
      Agent talker = talk.getAgent();
      // 発話者が自分なら処理しない
      if(talker == me) {
        continue;
      }
      // 扱いやすいようにContent型にする
      Content content = new Content(talk.getText());
      // subject(主語)がUNSPEC(未特定)の場合は，発話者に入れ替える
      if(content.getSubject() == Content.UNSPEC) {
        content = replaceSubject(content, talker);
      }
      // 演算子がある場合は再帰的に処理する必要があるため，別メソッドへ
      parseSentence(content);
    }
    // ヘッドを上書きする
    talkListHead = currentGameInfo.getTalkList().size();
  }
```

7. talkメソッドの実装

talkメソッドではこのターン話す内容をreturnします．今回の実装では，話す内容はtalkQueueに入れているので，その先頭から順番に返していくようにします．

```
@Override
public String talk() {
  // 発話する内容がない時は，SKIPを返す
  if (talkQueue.isEmpty()) {
    return Talk.SKIP;
  }
  // キューの先頭を取ってくる
  Content content = talkQueue.poll();
  // 主語が自分なら，省略する．
  if (content.getSubject() == me) {
    return Content.stripSubject(content.getText());
  }
  return content.getText();
}
```

話す内容がない時はSKIPが返るようにしてあること，主語が省略できる場合は省略していることなどがポイントです．なぜ主語を省略するのかですが，公式がそうしているから真似ただけに過ぎません．省略しなくてもいい気がしますが，スマートさを求めてそうしてあります．

8. voteメソッドの実装

voteメソッドは，その日に投票したい相手を返すメソッドです．今回の実装では，投票候補を入れいている変数 `voteCandidate` がありますので，原則それを返します．nullが入っていた場合は，生存者からランダムに返すようにします．

```
@Override
public Agent vote() {
  if(voteCandidate == null) {
    return randomSelect(aliveOthers);
  }
  return voteCandidate;
}
```

9. その他のメソッド

   各エージェントに共通する部分をBasePlayerで書いているので，これ以外のwhisper/attack(人狼専用)・divine(占い師専用)・guard(狩人専用)には特に何も書きません．また，finishメソッドでは，今回の実装では何もおこなわないため，同様に何も書きません．getNameメソッドについては，先に作ったAbstractRoleAssignPlayerを継承したクラスが代わりに名前を返してくれるので，こちらも特に何も書き替えません．

村人プレイヤーの作成

さて，BasePlayerができたので，実際に村人を作ってみましょう．今回作る村人は，実装が簡単な代わりに超無能に仕上げます．

1. VillagerPlayerクラスの作成する
   1. パッケージ・エクスプローラのsrc > 任意のパッケージ名 > player を右クリック
   2. 新規 > クラス
   3. 名前: には `VillagerPlayer` と入力
   4. スーパークラス : 横の参照を押す
   5. 検索欄に "BasePlayer"と入力し，表示された "BasePlayer - [your package name]"を選択して OK
   6. 完了 を押す

ここで，Plyaerインターフェースを実装しなくていいの？と思われるかもしれませんが，BasePlayerが実装しており，それを継承してVillagerPlayerを作成しているので，VillagerPlayer自体にはPlayerの実装は必要なくなっています．また，継承は実装と違って，予めメソッドが作られることはありません．自分で必要な分だけ書く必要があることに注意してください．

2. talkメソッドに書き加える

   村人用にtalkメソッドに処理を書き加えます．VillagerPlayerに `talk` と入力してください．予測変換に `talk():String - 'BasePlayer'でのメソッドのオーバーライド` と出てくると思いますので，それを選択します．すると，以下のようなメソッドが出来上がると思います．

```
@Override
public String talk() {
  // TODO 自動生成されたメソッド・スタブ
  return super.talk();
}
```

   この `return super.talk();` というのは，親クラス(BasePlayer)のtalkメソッド `super.talk();` から返ってきた内容を `return` するという意味です．実際にこのメソッドを書き換えて説明します． `return` の前に追記して次のようにしてください．

```
@Override
public String talk() {
  // 投票対象が決まっていなければ，決めて発言する
  if(declaredVoteCandidate == null) {
    // 投票対象をランダムに生存者の中から決める
    voteCandidate = randomSelect(aliveOthers);
    // キューに発言を追加する
    talkQueue.add(new Content(new VoteContentBuilder(me, voteCandidate)));
    // 宣言済み投票対象を書き換える
    declaredVoteCandidate = voteCandidate;
  }
  return super.talk();
}
```

村人のtalkメソッドを呼ばれると，まずその日の投票対象を決めたかどうかを確認します．決まっていない場合は，ランダムに誰か一人を選んで，その人に投票することを宣言する文を作成してキューに入れます．その後に，BasePlayerのtalkメソッド `super.talk()` を実行します．

このように書くことで，VillagerPlayerクラスには発言する内容をキューに追加する処理を書くだけで済みます．実際に何をreturnするのかは，BasePlayerに書いたためです．

ちなみに，村人はこれで完成です．村人たる最低限は作り終えました．適当に投票するだけの無能であることは間違いありませんが，動きます．

村人プレイヤの紐付け

AbstractRoleAssignPlayerを継承したクラス のコンストラクタで，村人プレイヤを紐づけましょう．

```
public RoleAssignPlayer() {
  setVillagerPlayer(new VillagerPlayer());
}
```

村人プレイヤの動作チェック

実際に村人プレイヤを動かして，動作チェックを行います．

1. 設定ファイルの書き換え / 動作確認 でも書き換えたAutoStarter.ini(lib > AIWolf-ver0.6.0 > AutoStarter.ini) を開きます．
2. `Sample5,java,org.aiwolf.sample.player.SampleRoleAssignPlayer` の行の先頭に `#` を付けます．→ `#Sample5,java,org.aiwolf.sample.player.SampleRoleAssignPlayer` iniファイルは `;` でもコメントアウトとして扱うことがありますが，人狼知能プラットフォームでは，`#` のみコメントアウトとして扱っています．
3. `#` をつけた行の下に，自分の作成したエージェントの名前・言語・"AbstractRoleAssignPlayerを継承したクラス"のクラスパス・役職指定 を追記します．例としては以下のような感じです．

```
Tutorial,java,jp.ac.aitech.maslab.aiwt2020.player.RoleAssignPlayer,VILLAGER
```

役職指定は省略できますが，今回は村人の動作検証をしたいので，村人を指定します．

4. 設定ファイルの書き換え / 動作確認 で押した，実行ボタン横の▼ではなく，今回は実行ボタンそのものを押します．これは前回の構成と同じものを実行するようになっています．別の構成を実行したくなった時は，▼を押しましょう．自分のエージェントが村人のみで参加していたら成功です．

```
==========================================
Winner:VILLAGER
    BODYGUARD   MEDIUM  POSSESSED   SEER    VILLAGER    WEREWOLF    Total
Sample1 0/0 0/0 8/21    12/26   12/28   13/25   0.450
Sample2 0/0 0/0 13/25   17/26   8/23    14/26   0.520
Sample3 0/0 0/0 15/28   8/19    15/26   14/27   0.520
Sample4 0/0 0/0 15/26   12/29   14/23   10/22   0.510
Tutorial    0/0 0/0 0/0 0/0 49/100  0/0 0.490
```

他の役職のプレイヤの実装

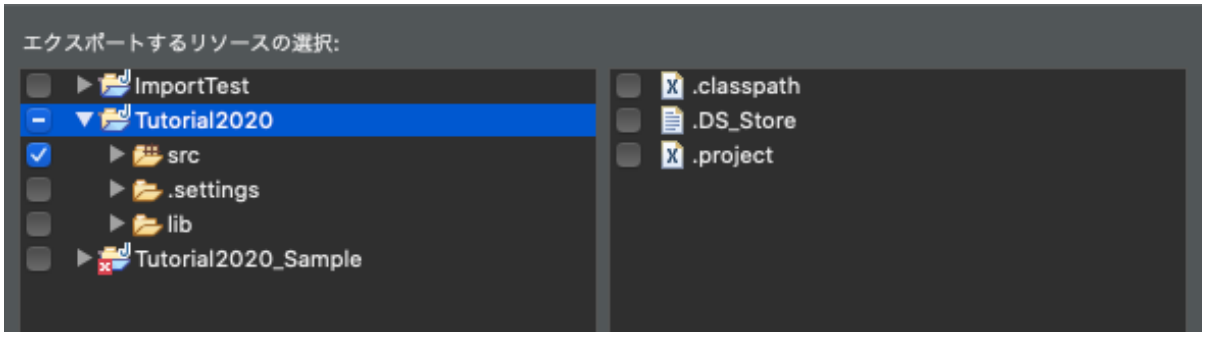
チュートリアルでは，他のプレイヤの実装については紹介しません． 人狼知能の開発者向けページ に，過去の大会参加者のソースコードをダウンロードできるページへのリンクが貼ってありますので，そこからJavaエージェントを見つけて，中を見てみてください．もしくはサンプルエージェントを見てみると良いでしょう．ソースコードは公式のGitHubにあります．

- 人狼知能公式GitHub SamplePlayer

エージェントのエクスポート

自分のエージェントを大会に参加させるには，jarファイルとしてエクスポートする必要があります．EclipseでjarファイルのエクスポートをするEclipseでjarファイルのエクスポートをする方法を説明します．

1. パッケージ・エクスプローラのsrc を右クリック > エクスポート
2. フィルタに `jar` と入力し，Javaの中にあるJARファイル を選択して次へ
3. エクスポートするプロジェクトが間違っていないか，srcのみが選択されているか(libなどを含んでいないか)を確認する



チェックではなく横線になっているのは，部分的に選択されていることを示すマークなので，Tutorial2020に横線がついているのは問題ない(Tutorial2020からすればsrcが部分的に選択されている)

4. エクスポート先を選択してください: の参照を押し，出力先のディレクトリと出力名(Save As の入力欄が出力名)を決める．決めたらSaveを押す
5. 完了を押すとエクスポートされる．特にエラーが吐かれなければ成功．

実行環境の構築(Coming soon...)

他言語や自分以外のエージェントを含めて対戦させる方法などを書きたい...