BattleBugs Class Specifications

Enumerated Constants:

`Act`

MOVE, *TURN_CLOCKWISE*, *TURN_COUNTER_CLOCKWISE*, *ATTACK*, *GET_BONUS*, *USE_BONUS*, *DEFEND*

"This enumerated constant is used to specify the Act your bug should perform with ActToken and is returned by `getLastAct()`. It can be used as a `String` to get a nicely formatted version for printing to your bug's tab."

When using these terms you use `Act.MOVE` or whichever act you want.

`Attribute`

ATTACK, *DEFENSE*, *HEALTH*, *MAXHEALTH*, *POINTS*, *DIRECTION*, *DISTANCE*

"When methods return values based upon a given attribute, use these constants to specify the attribute."

When using these terms you use `Attribute.ATTACK` or whichever attribute you want.

`Bonus`

HEALTH, *POINTS*, *MAX_HEALTH*, *ATTACK*, *DEFENSE*, *MYSTERY*

"When a method's return value is based upon a given bonus type, use these constants to specify the bonus. Each constant can also be used to get basic information about it by using:

`Bonus.HEALTH` .METHODNAME where METHODNAME can be:

`int DEFAULT_VALUE()`
"Returns the default value for that bonus. The default value represents how much that bonus will add to your matching attribute. If `Bonus.HEALTH.DEFAULT_VAL` returns 100, then using that bonus will add 100 points to your bug's health."

`double randomPercent()`
"Returns the random percentage chance of that type of bonus appearing on the board."

`String toString()`
"Returns the bonus as a string suitable for printing to the output tab of your bug."

`Taunt`

NICE_TRY, *GOING_TO_GET_YOU*, *COMING_FOR_YOU*, *I_AM_KING_KONG*, *I_AM_KING_OF_THE_WORLD*, *RUN_FOR_IT*

"This enumerated type is used to send taunts to the Announcer tab. Some taunts must be directed at someone. Use the `Taunt. REQUIRES_TARGET()` method to determine if a target is required for a given taunt."

Constants:

```
    int SORT_ASCENDING=0;
    int SORT_DESCENDING=1;
```

"When calling methods that return a list in sorted order, use these constants to specify the desired sort order."

When using these terms you use `Informer.SORT_ASCENDING` or `Informer.SORT_DESCENDING`

Informer Methods:

`int turnTowards(Target t)`
"Return the absolute direction to the target `t` "

`int adjacentOpponentsCanAttack()`
"Returns the number of bugs that are adjacent to your bug and are facing towards your bug so they might choose to attack you this round."

`ArrayList<Target> getTargetList()`
"Returns a complete list of bugs currently on the board in unsorted order. The list may have a size of 0 if you your bug is the sole survivor, but will never be `null`."

`ArrayList<Target> getAdjacentTargetsFacingMeList()`
"Returns a list of bugs that are adjacent to your bug and are facing towards your bug so they might choose to attack you this round. The list may have a size of 0 if you your bug has no adjacent opponent bugs facing your bug, but the list will never be `null`."

`boolean isTargetAdjacentClockwise()`
"Returns `true` only if there is an opponent adjacent one turn facing clockwise to your current direction."

`boolean isTargetAdjacentCounterClockwise()`
"Returns `true` only if there is an opponent adjacent one turn facing counter-clockwise to your current direction."

`boolean canMoveClockwise()`
"Returns `true` only if there is an empty space one turn facing clockwise one step forward. In other words, if you turned clockwise, then moved, returns `true` if that space would be open right now."

This would most likely only be useful in evaluating which direction to turn if you are trying to identify an open direction to move into. This might be useful if trying to break contact with an opponent.

`boolean canMoveCounterClockwise()`
"Returns `true` only if there is an empty space one turn facing counter-clockwise one step forward. In other words, if you turned counter-clockwise, then moved, returns `true` if that space would be open right now."

See above.

`ArrayList<Target> getAdjacentTargetList()`
"Returns a list of bugs that are adjacent to your bug. The list may have a size of 0 if you your bug has no adjacent opponent bugs, but the list will never be `null`."

`ArrayList<Target> getSortedTargetList(int order, Attribute attribute)`
"Returns a non-`null` list of bugs sorted in either ascending or descending order based upon the specified attribute. The list might have a size of 0 if your bug is the only one remaining on the board. "

`ArrayList<BonusTarget> getBonusTargetList()`
"Returns a complete list of bonuses currently on the board sorted by distance. The list may have a size of 0 if there are no bonuses on the board, but will never be `null`."

`ArrayList<BonusTarget> getBonusTargetByType(Bonus type)`
"Returns a list of bonuses of the specified type currently on the board sorted by distance. The list may have a size of 0 if there are no bonuses on the board, but will never be `null`."

`boolean canMove()`
"Returns `true` if the space in front of your bug is empty."

`int getAttack()`
"Returns your bug's current attack value."

`Bonus getEffect()`
"Returns an enumerated Bonus if your bug has picked one up, and `null` if your bug doesn't currently have one that is can use."

`int getDefense()`
"Returns your bug's current defense value."

`int getMaxHealth()`
"Returns your bug's current Maximum Health value."

`int getHealth()`
"Returns your bug's current Health value."

`int getPoints()`
"Returns your bug's current number of points."

`boolean foeInFront()`
"Returns `true` if the space in front of your bug has another bug in it."

`boolean bonusInFront()`
"Returns `true` if the space in front of your bug has a bonus in it."

`BonusTarget BonusInFront()`
"Returns a `BonusTarget` for the Bonus in front of your bug or `null` if there is no bonus."

`int getDirection()`
"Returns the absolute direction your bug is currently facing (0=North)."

`Target targetFoeInFront()`
"Returns a reference to the BattleBug located directly in front of your BattleBug, or `null` if the space in front of your BattleBug has no BattleBug."

`String sortOrderString(int sortOrder)`
"Returns a string of 'ascending' or 'descending' for use in sending output to your bug's tab. (0=ascending, 1=descending."

If you want to output to your bug's tab what sort order you used to generate a list, you can use `Informer.sortOrderString(Informer.SORT_ASCENDING)` *for example to print the string "ascending.*

`getTargetEffect(Target t)`
"Returns the `Bonus` type the target is carrying or `null` if the target bug isn't currently carrying a bonus."

`int field(Attribute attribute, Target t)`
"Returns the value of the `Target` for a given `attribute` enumerated type. To get the health of a target use `myInformer.field(Attribute.HEALTH , t)`."

`void print(String s)`
"Use this method to output a string to your bug's tab."

`boolean haveBonus()`
"Returns true if you have a bonus that you can use."
`Bonus whatTypeOFBonus()`
"Returns the enumerated `Bonus` your bug is carrying or `null` if your bug has no bonus."

`Act getLastAct() {`
"Returns the enumerated `Act` your bug performed last turn (returns Defend on the first round)."

`void taunt(Taunt taunt)`
"Use this method to send a `Taunt` to the Announcer tab. For example `myInformer.taunt(Taunt.NICE_TRY)`."

If the taunt selected requires a target, using this method will result in no text being sent to the announcer tab (see the next entry for a safe way to use the `taunt` *method).*

`void taunt(Taunt taunt, Target target)`
"Use this method to send a `Taunt` to the Announcer tab when the taunt requires a target. For example `myInformer.taunt (Taunt.GOING_TO_GET_YOU, target)`."

If the taunt selected doesn't require a target, using this method will still result in the taunt text being sent to the announcer tab. This version of the `taunt` *method may also be used to trigger a taunt with no target using the form:* `Taunt.GOING_TO_GET_YOU, null)`

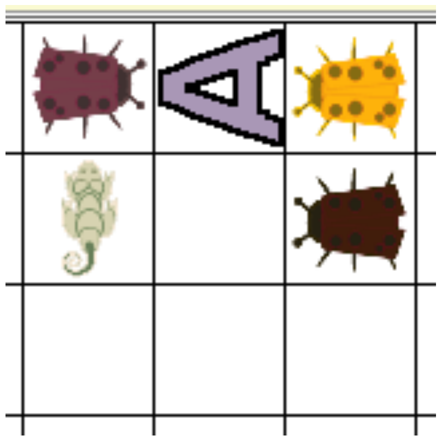
`int getOpenRelDirectionClockwise()`
"Returns the number of degrees to the first open adjacent space turning clockwise (returns 360 if no spaces are open)."

`BonusTarget targetBonusInFront()`

"Returns a reference to the Bonus located directly in front of your BattleBug, or `null` if the space in front of your BattleBug has no Bonus."

`int getOpenRelDirectionCounterClockwise()`
"Returns the number of degrees to the first open adjacent space turning counter-clockwise. Since the direction is relative to your current facing it will always be negative (returns -360 if no spaces are open)."

`int getOpenRelDirection()`
"Use this method to examine the adjacent squares to your current location in both the clockwise and counter-clockwise direction from your current facing. The returned value will be a ± relative direction representing the closest open square. This method returns 0 if the square in front of your bug is open and will randomly select clockwise or counterclockwise if there are open squares in both directions an equal number of degrees. A returned value more or less than 180 degrees indicates that there is NO open square surrounding your bug."



When called this method is called for **A** *the number -90 is returned because* **A** *would need to turn 270 clockwise to find an open square but only 90 degrees counter-clockwise.*

If there were a another bug or bonus located in the empty square then the method would return a value of < -180 degrees or > 180 degrees.

`The current version should always return -225 when your bug is surrounded but this should NOT be assumed to always be the case.`

`int getRow()`
"Returns the current row your bug occupies ranging from 0 to the maximum number of rows - 1."

`int getCol()`
"Returns the current column your bug occupies ranging from 0 to the maximum number of columns - 1."

`int getRows()`
"Returns the number of rows in the current grid."

`int getCols()`
"Returns the number of columns in the current grid."

`int getTurnsUntilPenalty()`
"Returns the number of turns remaining until the next penalty damage will be applied if no BattleBug damages deals damage to an opponent."

`public int getPenaltyDamage()`
"Returns the amount of damage penalty that will get applied to every BattleBug if no damage is being done (a.k.a. the 'peace dividend') when `getTurnsUntilPenalty()` reaches 0."

Target Methods:

`boolean targetFacingMe()`
"Returns `true` if this target is facing the bug."

`String getTargetName()`
"Returns the name of the target for sending to your bugs tab."

`String getTargetCreator`
"Returns the name of the target's creator for sending to your bugs tab."

`long getTargetVersion`
"Returns the version of the target's bug. "

`int getTargetHealth()`
"Returns the target's current health. "

`int getTargetAttack()`
"Returns the target's attack value. "

`int getTargetDefense()`
"Returns the target's current defense value. "

`int getTargetMaxHealth()`
"Returns the target's current maximum health value. "

`int getTargetPoints()`
"Returns the target's number of points. "

`int getTargetPointValue()`
"Returns the number of points earned for eliminating the Target. 10% of this amount is scored each time the Target is damaged. "

`int getTargetDirection()`
"Returns the target's current absolute direction. "

`Act getTargetsLastAct()`
"Returns the last `Act` performed by the target. "

`boolean targetValid()`
"Returns `true` if this is still a valid target. A target will become invalid if it has been removed from the board."

*When a target is being stored between turns, it is **important** to make sure it is still valid before trying to use the `Target` methods!*

`int getDirectionTo()`
"Returns the absolute direction toward this target (returns 0 if the target is no longer valid)."

An absolute direction is 0-360 degrees while a relative direction is ±180 degrees relative to your bug's *current direction.*

`String getName()`
"Returns the target bug's name (duplicates `getTargetName()`). "

`int getRelDirectionTo()`
"Returns the relative direction toward this target (returns 0 if the target is no longer valid)."

A relative direction is ±180 degrees relative to your bug's current direction.

`int getStepsDistance()`
"Returns the number of steps it would take to end up adjacent to, and facing this target."

A target located one square behind your bug would have a step distance of 4 because it would require four turns to face that target.

`int getRow()`
"Returns the current row the target bug occupies ranging from 0 to the maximum number of rows - 1."

`int getCol()`
"Returns the current column the target bug occupies ranging from 0 to the maximum number of columns - 1."

`boolean hasActed()`
"Returns `true` if the target has already taken it's turn this round."

BonusTarget Methods:

`String getLocation()`
"Returns the target's location in the form of row & column formatted string. "

`Bonus getEffect()`
"Returns the target's enumerated `Bonus` type. "

`int getDirectionTo()`
"Returns the absolute direction toward this target (returns 0 if the target is no longer valid)."

An absolute direction is 0-360 degrees while a relative direction is ±180 degrees relative to your bug's current direction.

`int getrelDirectionTo()`
"Returns the relative direction toward this target (returns 0 if the target is no longer valid)."

A relative direction is ±180 degrees relative to your bug's current direction.

`boolean isValid()`
"Returns `true` if this is still a valid target. A target will become invalid if it has been removed from the board."

*When a target is being stored between turns, it is __**important**__ to make sure it is still valid before trying to use the `Target` methods!*

`int getStepsDistance()`
"Returns the number of steps it would take to end up adjacent to, and facing this target."

A target located one square behind your bug would have a step distance of 4 because it would require four turns to face that target.

`int getRow()`
"Returns the current row the target bonus occupies ranging from 0 to the maximum number of rows - 1."

`int getCol()`
"Returns the current column the target bonus occupies ranging from 0 to the maximum number of columns - 1."