Robocode Game Basics

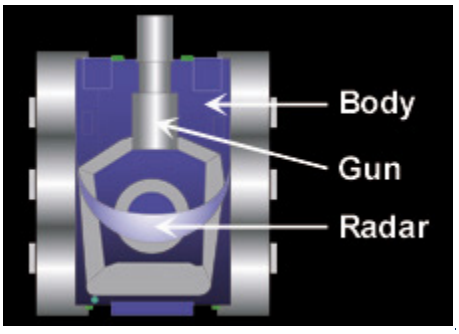
Goal

The goal of robocode battles is to be the winner by achieving the highest score in each round.

Robot Anatomy

> **Be Aware!**
>
> Robots submitted for competition must extend the ***robocode.Robot*** class, **not** the *robocode.JuniorRobot* or *robocode.AdvancedRobot* classes. Winning robots will be examined, and code reviews performed.

A robot consists of three individual parts:



- **Body** - Carries the gun with the radar on top. The body is used for moving the robot ahead and back, as well as turning left or right.
- **Gun** - Mounted on the body and is used for firing energy bullets. The gun can turn left or right.
- **Radar** - Mounted on the gun and is used to scan for other robots when moved. The radar can turn left or right. The radar generates onScannedRobot events when robots are detected.

Sample Robot

On the installation of robocode, you will find a robot that has some basic behavior already filled in that can get you started. Please **rename this robot** to reflect the team or individual name, by changing the class name and saving the java file as that class name.

Robot, Gun, and Radar rotation

Max rate of rotation of robot:	(10 - 0.75 * abs(velocity)) deg / turn. The faster you're moving, the slower you turn.
Max rate of rotation of gun:	20 deg / turn. This is added to the current rate of rotation of the robot.

Max rate of rotation of radar:	45 deg / turn. This is added to the current rate of rotation of the gun.

Rotating the whole robot is expensive. The gun can be rotated a lot more per player turn than the whole robot. The radar can be rotated even more per player turn that the gun. Moving the radar independently will result in a higher chance of detecting an enemy robot. However this introduces some issues, as the radar, gun and robot body can all be facing different directions and this must be accounted for when firing at the detected robot.

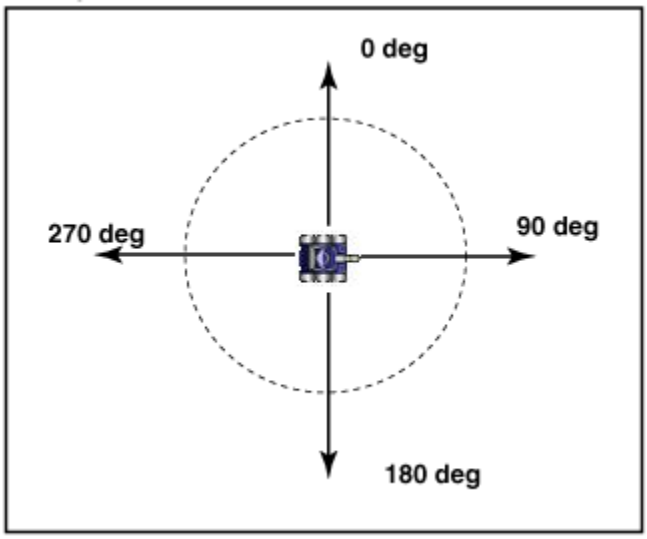
Independent movement of the gun and radar can be set by calling the following methods in the run method (before the main loop).

- setAdjustGunForRobotTurn
- setAdjustRadarForRobotTurn

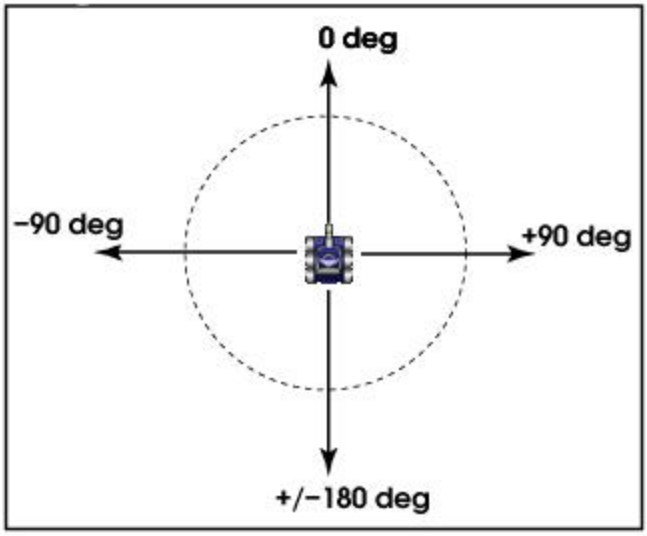
> **In the Openet supplied sample robot, by default, the gun and radar are bound together and turn independently of the robots body.**

Heading vs Bearing

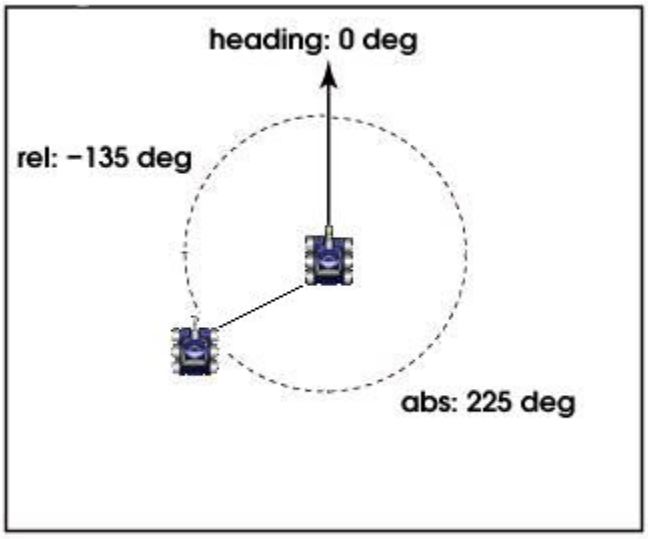
A heading is a value between 0 and +360 degrees. You can get your degree heading by calling getHeading(). The robot below has a heading of 90 deg.



There are occasions where you get a *Bearing*, that is to say, a degree measurement from -180 to 180 that represents your offset from something (a wall, another robot, etc.)



The following illustration shows both the relative and absolute bearing from one robot to another



To face the robot with an abosolute bearing of 225 in the most efficient way, you need to turn your robot -135 deg rather than +225 deg.

> **Note**: That because a bearing is a value from -180 to 180, calling turnRight() will actually make you turn *left* if the bearing is negative -- this is by design and it is what you want to have happen.
>
> **Note**: When onScannedRobot method is run, the bearing of the scanned robot is relative to your robot's heading, not the gun or radar heading.

Energy

You lose energy every time you hit a wall, you are hit by an enemy bullet, ram an enemy, or you fire your gun. The

amount of energy you lose by being hit is `4 * bullet power + 2 * max(bullet power - 1 , 0)`. So the maximum amount is 16.0. When you fire, you spend a quantity of energy equal to the power of the bullet fired. When one of your bullets hits an enemy, you collect back `3 * bullet power` energy. When you hit an enemy bot, each bot takes 0.6 damage.

You can't kill yourself, so when your energy drops to zero because you hit a wall or you fire, your bot gets disabled. It will not be able to move nor fire. If you are lucky enough and one of your bullets in the air hits an enemy, you will get some energy back and recover from disabled status.

If your bot is disabled but your energy is > 0, you may have called a getXXX() function - such as getVelocity() - too many times a turn. The limit is 10000 getXXX() function calls per turn.

Strategies

The sample bot you have been given performs some basic movement, scanning its gun in a 360 deg arc, then moving forward a little. It also has some logic to point the gun and fire at a robot when it scans it. This logic is not optimal and has some bugs in it.

Explore the robocode.Robot API in detail. There are a number of event driven and status methods that can be overridden to gather more information and react to the environment.

Common strategies to make your bot more survivable or more formidable are:

Basic Strategies

- Efficient detection
    - Scanning as frequently as possible increases the chances of detecting an enemy before they detect you
- Effective targeting of enemies
    - Get the gun pointed at the enemy as quickly and accurately as possible and fire
- Efficient movement
    - Turning 5 degrees right instead of 355 degrees left.
- Moving closer to or further away from enemies
    - An optimal distance for your chosen targeting and firing strategies will become clear from testing
- Managing your energy by varying the fire-power you apply.
    - The range of power to apply to a bullet is large - 0.1 to 3.0 - and has different applications depending on the situation. The bullets velocity decreases as the power increases.
- Unpredictable movement.
    - Don't be a sitting duck

Intermediate Strategies

- Escaping from walls and corners
    - Don't be a sitting duck
- Locking onto a particular enemy
    - Don't do other Robots jobs for them by softening up someone else's target.
- Taking the size of the arena into account.
    - A tight space may require a different strategy to a large open space

Advanced Strategies

- Predicting the enemies future position.
    - The information on an enemies present position, heading, velocity, energy and distance are all supplied when an enemy is scanned. With the right application, you could guess where they will be when firing a bullet.

- Reacting to events like being hit by a bullet or detecting an enemy has fired at you.
  - Again, use the information that's available. A bullet hit event can give you a heading pointing right back to the enemy who fired it.
- Pattern matching
  - Store and interpret information on scanned robots to detect if they move or fire in a particular predictable pattern.

> **Remember to test your strategy implementation a number of times in your own environment against the supplied sample bots to ensure it works in all situations.**

Coordinates and Direction Conventions

Coordinates System:	Robocode is using the Cartesian Coordinate System, which means that that the (0, 0) coordinate is located in the bottom left of the battle field.
Clockwise Direction:	Robocode is using a clockwise direction convention where 0 / 360 deg is towards "North", 90 deg towards "East", 180 deg towards "South", and 270 deg towards "West".

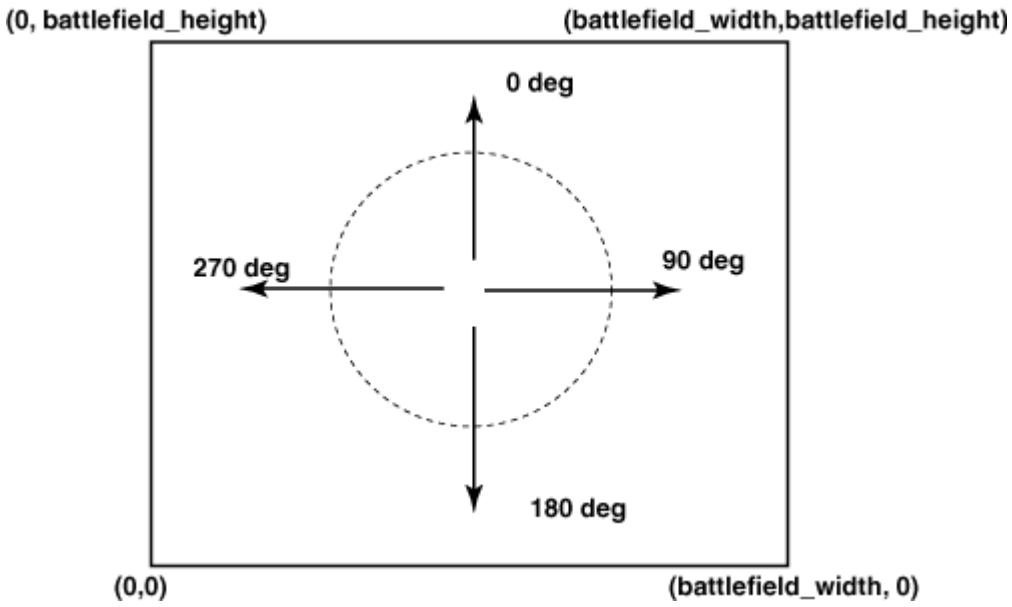


Figure 1:

Time and distance measurements in Robocode

Time (t):	Robocode time is measured in "ticks". Each robot gets one turn per tick. 1 tick = 1 turn.

Distance Measurement:	Robocode's units are basically measured in pixels, with two exceptions. First, all distances are measured with *double* precision, so you can actually move a fraction of a pixel. Second, Robocode automatically scales down battles to fit on the screen. In this case, the unit of distance is actually smaller than a pixel.

Robot Movement Physics

Acceleration (a):	Robots accelerate at the rate of 1 pixel/turn/turn. Robots decelerate at the rate of 2 pixels/turn/turn. Robocode determines acceleration for you, based on the distance you are trying to move.
Velocity Equation(v):	v = at. Velocity can never exceed 8 pixels/turn. Note that technically, velocity is a vector, but in Robocode we simply assume the direction of the vector to be the robot's heading.
Distance Equation (d):	d = vt. That is, distance = velocity * time

Bullets

Damage:	4 * firepower. If firepower > 1, it does an additional damage = 2 * (power - 1).
Velocity:	20 - 3 * firepower.
GunHeat generated:	1 + firepower / 5. You cannot fire if gunHeat > 0. All guns are hot at the start of each round.
Power returned on hit:	3 * firepower.

Collisions

With Another Robot:	Each robot takes 0.6 damage. If a robot is moving away from the collision, it will not be stopped.
With a Wall:	AdvancedRobots take abs(velocity) * 0.5 - 1; (Never < 0).

Robocode Processing Loop

The order that Robocode runs is as follows:

1. Battle view is (re)painted.
2. All robots execute their code until they take action (and then paused).
3. Time is updated (time = time + 1).
4. All bullets move and check for collisions. This includes firing bullets.
5. All robots move (gun, radar, heading, acceleration, velocity, distance, in that order).
6. All robots perform scans (and collect team messages).
7. All robots are resumed to take new action.
8. Each robot processes its event queue.

Most of this can be gleamed by following the method calls from BaseBattle.runRound() and Battle.runTurn() in the robocode.battle module.

Event dispatch happens from within commands that take a turn. So the call stack when an event is delivered usually looks like this:

> ***Robocode internals Robot's run method Robocode internals Event handler***

However, event handlers can themselves make calls that take turns. If one of these happens to generate an event, we might suspect a call stack like

> ***Robocode internals Robot's run method Robocode internals First event handler Robocode internals Second event handler***

But this kind of nesting could lead to a stack overflow. Or—more commonly—cases where the first handler finishes up its actions long after the response-provoking situation has passed. Thus, Robocode takes special steps for events generated within event handlers; these measures are implemented in EventManager.processEvents(). In particular, the call stack will get as far as

> ***Robocode internals Robot's run method Robocode internals (including processEvents) First event handler Robocode internals (including processEvents)***

but then the inner processEvents will detect the impending nesting and throw an EventInterruptedException, which unwinds the stack to the catch block in the outer processEvents:

> ***Robocode internals Robot's run method Robocode internals (including processEvents)***

effectively canceling whatever the running event handler was up to. Next, the event-delivering loop in the outer processEvents resumes delivering events, letting the second event handler execute unnested:

> ***Robocode internals Robot's run method Robocode internals Second event handler***

It is possible, though not usually useful, to catch and respond to EventInterruptedExceptions in the first handler instead.

Scoring

Item	Explanation
Total Score	This is everything else added up, and determines each robot's rank in this battle.
Survival Score	Each robot that's still alive scores 50 points every time another robot dies.
Last Survivor Bonus	The last robot alive scores 10 additional points for each robot that died before it.
Bullet Damage	Robots score 1 point for each point of damage they do to enemies.
Bullet Damage Bonus	When a robot kills an enemy, it scores an additional 20% of all the damage it did to that enemy.
Ram Damage	Robots score 2 points for each point of damage they cause by ramming enemies.

Ram Damage Bonus	When a robot kills an enemy by ramming, it scores an additional 30% of all the damage it did to that enemy.

Firing Pitfall

Because bullets are fired before the gun is moved, calling setFire() will cause the bullet to leave at the current gun heading. This may seem counter-intuitive if you are used to thinking in terms of pointing a gun, then shooting. It is also inconvenient because you can't call `setTurnGun(...)` and `setFire(...)` right after each other (not if you need perfect accuracy, anyway). Most of the time, the error will be so small you won't notice it, but if you're testing a pattern matcher against `sample.Walls`, you will occasionally spot the bug.

To get the bullet to leave after turning the gun, you will need to use code like this:

```
long fireTime = 0;
void doGun() {
    if (fireTime == getTime() && getGunTurnRema
ining() == 0) {
        setFire(2);
    }

    // ... aiming code ...

    setTurnGunRight(...);
    // Don't need to check whether gun turn
will complete in single turn because
    // we check that gun is finished turning
before calling setFire(...).
    // This is simpler since the precise angle
your gun can move in one tick
    // depends on where your robot is turning.
    fireTime = getTime() + 1;
}
```