Robocode Teacher's Guide

This document gives a crash course in Robocde. It is intended for teachers, but may also be of interest to advanced students.It is strongly recommended that all tutors read this.

The Robocode API

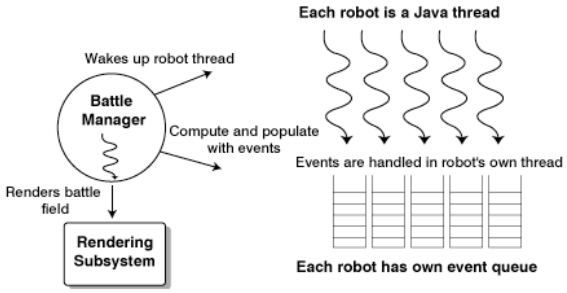
The API is composed of 28 classes, most of which are either Robots, Conditions, or Events. The most important aspects of these are explained below. A significant amount of the information is taken directly from the API, and anybody who plans to tutor a Robocode class should examine the official API reference before doing so; in particular the note about conventions used throughout the reference, explained at the beginning of the Robot class.

The Robocode IDE

Robocode has 2 integrated components:

- **Editor** - a rudimentary text editor is packaged; it supports java syntax highlighting and multiple open files and provides a built-in java compiler called jikes. It does not supprch functions, so is not well-suited to large files; however none of the sample or tutorial robots are large enough to make this a problem. There is a bug that seems to surface when switching between multiple open files, and causes the cursor to become invisible. It can be fixed by minimising then restoring the editor.
- **Battlefield** - This is where the robots compete. A robot in Robocode consists of one or more Java classes. These classes can be archived into a JAR package. The "Robot Packager" that can be activated from the battlefield GUI window is included for this purpose.

Implementation



Each robot is represented as a thread, with a master battle simulator thread, which operates as follows:


```
while (round is not over) do
   call the rendering subsystem to draw robots, bullets, explosions
   for  each robot do
       wake up the robot
       wait for it to make a blocking call, up to a max time interval
   end for
   clear all robot event queue
   move bullets, and generate event into robots' event queue if
applicable
   move robots, and generate event into robots' event queue if
applicable
   do battle housekeeping and generate event into robots' event queue
         if applicable
   delay for frame rate if necessary
end do
```

There is more detailed information in this article from Developerworks, of which this section is an extract.

How to Build a Robot

All robots must inherit from class `Robot`. A sublass `AdvancedRobot`, which adds some more complicated functionality, is provided. Both of these classes can be instantiated in a battle, although they will no act, as most of their methods are stubs which the user is expected to over-ride.

When a battle begins, the battle simulator thread calls `run()` for each competitor. This method will typically contain some initialisation code (executed once per round), followed by an infinite loop containing the general strategy of the robot. This is looped through continuously until one of the following occurs:

An event happens - This has the effect of diverting execution to an event-handling event. Several events (all implementing interface `Event`) are defined by robocode; more can be defined by the user by inheriting from `CustomEvent`. When an event occurs, an appropriate event-handling method is called immediately (all actions can be interrupted by events at any time). A user's robot should over-ride these methods, which have empty definitions within class `Robot`. The standard events are:

- `BulletHitBulletEvent`
- `BulletHitEvent`
- `BulletMissedEvent`
- `CustomEvent`
- `DeathEvent`
- `HitByBulletEvent`
- `HitRobotEvent`
- `HitWallEvent`
- `MessageEvent`
- `RobotDeathEvent`
- `ScannedRobotEvent`

- `SkippedTurnEvent`
- `WinEvent`

There is an event-handling method for each of these; their names begin with on and do not end in event (e.g. `ScannedRobotEvent` is handled by `onScannedRobot()`). Custom events are the one exception; they are handled by `onCustomEvent()`.

The time-limit is exceeded - This has the effect of Each robot is allocated a specific amount of milliseconds per turn. It is extremely unlikely that this will become a problem, since typically tens of milliseconds are allowed, and even very powerful robots will use 1 or 2 milliseconds only.

A blocking call is made - Blocking calls do not return until the robot completes some task, an event occurs, or some condition is met. The following commands are blocking calls:

- `void ahead(double distance)`
- `void back(double distance)`
- `void doNothing()`
- `void resume()`
- `void stop()`
- `void stop(boolean overwrite)`
- `void turnGunLeft(double degrees)`
- `void turnGunRight(double degrees)`
- `void turnLeft(double degrees)`
- `void turnRadarLeft(double degrees)`
- `void turnRadarRight(double degrees)`
- `void turnRight(double degrees)`
- `void waitFor(robocode.Condition condition)`

Non-blocking calls

These commands are used for queuing moves, rather than executing them instantly. Unlike the blocking calls above, non-blocking calls return immediately. Actions set up using non-blocking calls begin at the next blocking call. Most blocking calls have a corresponding non-blocking call, the name of which begins with 'set' (e.g. the non-blocking version of `ahead()` is `setAhead()`).

Non-blocking calls can be used to set up several moves that will happen simultaneously, e.g.:

```
// do some donuts
setTurnLeft(10000);
setAhead(10000);
waitFor(new TurnCompleteCondition(this));
```

Non-blocking moves of the same 'type' do not stack, e.g.

```
// only move back 100: 2nd call overrides 1st
setBack(100);
```



```
setBack(100);
waitFor(new MoveCompleteCondition(this));

// turns right 90: 2nd call overrides 1st
setTurnLeft(90);
setTurnRight(90);
waitFor(new TurnCompleteCondition(this));
```

All non-blocking methods are found in class `AdvancedRobot`. The following are non-blocking methods:

- `void setAhead(double distance)`
- `void setBack(double distance)`
- `void setResume()`
- `void setStop()`
- `void setStop(boolean overwrite)`
- `void setTurnGunLeft(double degrees)`
- `void setTurnGunRight(double degrees)`
- `void setTurnLeft(double degrees)`
- `void setTurnRadarLeft(double degrees)`
- `void setTurnRadarRight(double degrees)`
- `void setTurnRight(double degrees)`

The `execute()` method is unique; it begins execution of all commands queued using non-blocking calls, and returns immediately. In this example the robot will move while turning

```
setTurnRight(90);
setAhead(100);
execute();
while (getDistanceRemaining() != 0 && getTurnRemaining() != 0) {
    execute();
}
```

If more 300 or more non-blocking calls are made without a blocking call or `execute()`, the robot's turn is skipped.

Conditions

The `waitFor()` blocking call does not return until a given condition is met. The user can create custom conditions inheriting from class Condition.

2 Exceptional Cases

There a 2 cases given in the API in which Robocode behaves abnormally in order to simplify certain tasks:

- If you are in `onScannedRobot()`, and call `scan()`, and you still see a robot, then the system

will interrupt your `onScannedRobot()` call immediately and start it from the top. This is unusual because by default, events cannot interrupt events of the same prioriy (e.g. a `ScannedRobotEvent` cannot normally interrupt another `ScannedRobotEvent`). To set events as interruptible, use `AdvancedRobot.setInterruptible()`.

- If the gun and radar are aligned (and were aligned last turn), and the event is current, and you call `fire()` before taking any other actions, `fire()` will fire directly at the robot. In essence, this means that if you can see a robot, and it doesn't move, then fire will hit it. `AdvancedRobots` will NOT be assisted in this manner.

[Home | Introduction | My Second Robot]