Welcome to CodeRally!

The 2003 Java Challenge gives you the opportunity to pit your Java programming skills against other teams in a world of car rally racing. Each team will write a Java class which represents and controls a RallyCar. Your car (class) will be placed on a simulated race track along with cars from other teams.

Rally cars can move around on a track and obtain important information such as the location of various objects on the track and the current capabilities of other cars. Cars can collide with other cars, throw spare tires to distract other cars, and they can enter a protected mode to (temporarily) protect themselves from being hurt by other cars.

The rally pits cars against each other in a series of matches. A match consists of up to six cars competing with each other. Each car starts a match positioned in a random location and facing a random direction on a finite two-dimensional grid, with the same amount of fuel and spare tires as the other cars. Driving around the track uses fuel. The track contains places where cars can go to get additional fuel and different places they can go to pick up spare tires. If a car runs out of fuel it can no longer move, so it is important to constantly check your fuel and refuel if necessary.

During each match, a car can accumulate “points.” Cars can earn points in four ways: by successfully throwing a spare tire at another car, by passing checkpoints in order or out of order around the track, and by the amount of fuel remaining at the end of a match. Cars with the highest point totals from each match advance to subsequent rounds.

The rest of this guide shows how to create a car and how to invoke its various capabilities. You should work now to understand the code structure that your car must implement, and discuss the strategy for your car to use during the game. It is permissible for team members to discuss code structure and strategy with their coaches and alternates during this time.

The coding part of the Java Challenge starts when you get access to your contest machine. You will use the Eclipse development environment to create and test your car. You must use Eclipse to develop your car code, since the simulated track runs under Eclipse. Only the three members of the Java Challenge team (which may be any three members of your World Finals team, including alternates) may be on the contest floor during the coding phase.

You can test your car during the coding phase in two ways. First, there will be a private version of the CodeRally track environment on your machine. The private track contains a collection of sample cars with various capabilities (but no cars from other teams). You may run your car on this private track to see how it performs. Only your team will be able to see the results of running your car on the private track.

The second way to test a car is to submit it to the public CodeRally track. When you submit your car to the public track, you will obtain a snapshot on your machine of all other cars that have been submitted up to that point in time. You will be able to run a public version of the CodeRally track to watch the performance of your car against other teams’ submitted cars (and vice versa).
The performance of your car on the public track during the coding phase does not count. It is strictly an opportunity for you to see how your car performs against other teams’ cars in order to consider making changes. Spectators will also be able to see the public track during the coding portion of the Java Challenge.

It is not a requirement to submit your car to the public track during coding phase. However, all teams must submit their final cars prior to the end of the coding phase, which will be two and a half hours from the start of coding (unless announced otherwise).

The final versions of all cars (the last submission you make) will compete in a Tournament which will take place at the dinner tonight hosted by IBM. The contest staff will initially group cars randomly into matches with up to five other cars. After each round, cars will be regrouped according to the points they accumulate and a new round will begin.

All cars will compete in at least three rounds. After that, eliminations will take place based on points and a new series of rounds will begin. Each car will start each new series of rounds with zero points and random regrouping. The winners of the 2003 Java Challenge will be the teams whose cars earn the most points in the final Tournament Round.

The remainder of this guide is intended to help you understand how to design and implement your car. JavaDoc files are available, in both printed form and on the machine your team will use, describing the classes and interfaces that relate to coding your car.

Good Luck in the Java Challenge CodeRally!
Coding Your Car – Overview

When you start Eclipse on your machine and open the Java Challenge project, you will find a skeleton for the class `RallyCar.java`. This is the class that will contain the code making up your car. You may add fields and declare additional methods, but all code you write for your car must be inside this class.

The `RallyCar` class contains stubs for certain methods required in cars; you will have to fill in the code in these methods. Modifying these methods is the primary manner in which you create the “personality” of your car. You may also add other fields and methods to the `RallyCar` class to further define its characteristics.

When your team has a version of the `RallyCar` class that you wish to test in the private track, save your `RallyCar` code and then click the “Run the private track” button on the Eclipse toolbar, which has the following icon: 🟩 (Note: this icon has a red checkered flag in it; don’t confuse it with the other similar icon which has a green checkered flag.) If you are not sure about the function of an icon, move the mouse over it to see a ToolTip display of its function.

When your team wishes to submit `RallyCar` to the public track, save your code and click the “Submit the code” button on the Eclipse toolbar, which has the following icon: ☑.

Once you have submitted your car to the public track, you can run your car in the public track and view its performance against other submitted cars. To run the public track, click the “Run the public track” button, which has the following icon: ☑ (Note: this icon has a green dot in it; don’t confuse it with the other similar icon which has a red dot.)

The class `Car.java` is the superclass of `RallyCar`. It defines a number of methods that are inherited by `RallyCar`. These methods can be extremely useful in controlling your car.

You should not modify anything in the `Car` class. In fact, when you run your car, it will actually run with a different version of the `Car` class from the one you see in your environment. In particular, the `Car` class you will see contains some dummy initialization and return value code that will be replaced when you run in the private or public track.

In addition to the above classes, your environment will contain three Java interfaces that define the interfaces presented by various components of the track:

- `IObject.java`

  This is the interface of all objects in the simulated track. Every object implements this interface, which declares methods `getX()` and `getY()` that return the location of the object on the track. All track coordinates are non-negative values of type `double`. 
Java Challenge 2003

• ISpareTire.java

This interface extends IObject and defines the interface of all spare tires that are currently active in the simulated track. Every spare tire implements this interface, which declares methods getHeading() and getSpeed(). Thus every spare tire contains methods that allow cars to determine important characteristics of the spare tire as the tire moves across the track.

• ICar.java

This interface extends IObject and defines the interface for all cars on the simulated track. Thus it defines methods your car can invoke either on itself or on an opponent’s car. These methods are described in further detail below and in the JavaDocs for the CodeRally environment.

The CodeRally Track Simulation

Identification

There are three method stubs in the RallyCar class that your team must fill in to identify your car. The first is getSchoolName(), which must return a string of no more than 25 characters giving the name of your School or University as registered for the World Finals. Cars that do not assign their World Finals registered school name in getSchoolName() will not be allowed to compete in the Tournament.

The second required identification method is getName(), which must return a string of no more than 25 characters assigning a team-chosen name to your car. You may choose any name you wish for your car, but the contest staff reserves the right to modify or eliminate cars whose names it deems are inappropriate.

The third identification method is getColor(), which must return a byte constant chosen from the predefined car colors given in the Car class. You can use this method to assign a color to your car, which determines its appearance in the graphical display of the CodeRally environment. The default value returned by getColor() is CAR_BLUE.

The identification methods must not do any computations other than returning the specified constant values.

Initialization

When your car is placed into a track, the simulator invokes the initialize() method in your car. Put any initialization code you want to have executed into this method. You may make use of the entire API at this time. Be aware that the simulator will provide only a limited amount of time (1 second) for your initialization code to execute before it begins the game. If your initialization code fails to complete within the time limit, your car will enter the track in an uninitialized state, with unpredictable results.
Moving Your Car

Once the simulator finishes its timed calls to each car’s `initialize()` routine, it calls the `move()` method in each car in sequential order. This happens once every clock tick. The code in your car’s `move()` method determines what actions it takes during the course of a game. In addition to the input parameters to `move()`, which give some status information, methods are available to your car to query its own status, to change variables such as the desired direction and speed at which it should move, to query the status of other cars, to find the location of objects on the track (for example, the gas stations which can be used to refuel or places where you can pick up spare tires), and to throw spare tires from your car.

`move()` has four parameters that provide information about what happened during the previous movement cycle. These parameters specify (1) how much time (in milliseconds) your `move()` method used the previous time it was called; (2) whether your car hit a wall during the previous cycle, (3) whether your car collided with another car during the previous cycle, and (4) whether your car was hit by a spare tire from another car during the previous cycle. The first parameter is an `int`, the second is a `boolean`, the third and fourth parameters return an `ICar` reference to the corresponding car (or null, if no collision or hit occurred). The first parameter is useful in determining whether your car is in danger of exceeding the maximum amount of time allowed to complete a move.

CodeRally Track Details

A CodeRally track is a two-dimensional world of 1010 units in X by 580 units in Y, with the origin in the top left corner. There is a wall around the outside edge of the track, and cars cannot go beyond the wall. There are no walls on the interior of the world. Cars can move freely about the world, unless they bump into another car. Objects move in directions called headings, which are measured in integer degrees. Zero degrees is “straight up”. All headings are positive numbers in the range 0..359 and increase in the clockwise direction.

The figure below describes the world:
The world has the following characteristics:

- The world is driven by a ticking clock whose value can be read using `getClockTicks()`.
- Each car starts the match with 100 fuel units and 3 spare tires.
- Setting steering and throttle causes a car to move continuously with those settings until it is instructed to do otherwise, although it may be blocked by walls or other cars.
- Cars can change throttle and steering instantaneously. Speed and direction will not change instantaneously because the cars have inertia.
- The minimum throttle of a car (`MIN_THROTTLE`) is -50 units and the maximum throttle (`MAX_THROTTLE`) is 100; the maximum rate of change of speed from a stopped position (except in collisions) is 8 units per tick.
- The minimum steering setting (`MAX_STEER_LEFT`) is -10, and the maximum steering setting (`MAX_STEER_RIGHT`) is 10. The rate of change of heading is dependent on speed and can be found via the `getChangeInHeading()` method.
- The location of a car on the track is a point. Cars are 60 units long and 40 units wide, centered at their location.
- To keep the detection of a round spare tire hitting a rectangular car simple, a spare tire will hit a car when its location passes within 40 units of the car’s location.
- Spare tires thrown by a car move at a constant velocity of 12 units per tick until they hit a car or a wall, at which time they disappear. Checkpoints, fuel depots, and spare tire depots do not affect spare tires as they move across the world. Spare tires do not hit each other if they pass over the same location.
- The maximum amount of fuel a car can have is 100 units.
- The maximum number of spare tires a car can have is 5.
- Whenever a car’s location is within 25 units of a fuel depot, the car’s fuel is increased at a constant rate of 1 unit per clock tick up to the maximum. There are 3 randomly placed fuel depots during each match.
- Whenever a car’s location is within 25 units of a spare tire depot, and the car has less than 5 spare tires, the car will pick up a spare tire every 5 clock ticks. There are 3 randomly placed spare tire depots during each match.
- Cars can protect themselves against collisions with spare tires or other cars by entering “protect mode”. A car moving in protect mode consumes fuel at twice the normal rate. Protect mode lasts for 50 clock ticks.
- Colliding with another car will cause your momentum to be transferred to the other car and both cars will lose 10 fuel units.
• Throwing a spare tire and successfully hitting another car will increase your points by 10. The car that is hit will lose 10 fuel units, its `move()` method will not be called for 10 clock ticks, and it will be pushed in the direction that the spare tire was traveling. You will not get any points for hitting a car which is in protected mode, nor will it affect that other car.

• Once a car throws a spare tire, that car will be unable to throw another spare tire for 25 clock ticks from the time the tire was thrown.

• The time limit to complete a single move is 500 milliseconds. If a car’s `move()` method does not return within 500 milliseconds of the time when it is called, the `move()` will not be called again for the rest of the match and the most recent steering and throttle settings will be maintained.

• There are a number of ordered checkpoints placed in a route on each track. Passing within 25 units of any checkpoint will give you 2 points, but going to the next successive checkpoint will give you 6 points. Returning to the same checkpoint twice in a row will not give you any points.

• Points are earned according to the following table. Remember that it is total points earned that determines which cars advance during elimination rounds.

<table>
<thead>
<tr>
<th>Action</th>
<th>Points Earned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Passing any checkpoint</td>
<td>2</td>
</tr>
<tr>
<td>Passing a checkpoint in successive order</td>
<td>6</td>
</tr>
<tr>
<td>Hitting another car with a spare tire</td>
<td>10</td>
</tr>
<tr>
<td>For each 10 units of fuel left after a match</td>
<td>1</td>
</tr>
</tbody>
</table>
General Information, Caveats, Constraints, and Restrictions

- The Java JRE being used for the Java Challenge is version 1.4.1.
- Cars may not define any constructors.
- Cars may not use static initialization blocks to initialize their cars.
- All the Java code for your car must be contained within the `RallyCar` class.
- Cars may not create their own threads, processes, print jobs, files, or other similar system functions.
- Cars may use `System.out.println()` to display information on the Eclipse console, but the time it takes to do this is charged against the car’s `move()` time limit. (Actions like this are relatively time-consuming.)
- You may run the simulation in Java debug mode. However, understand that when running either the private or public worlds in debug mode, all move and initialization time limits are automatically turned off. If your moves are taking a long time, you should test your car in run mode to ensure that it is not going over the time limits.
- You may not talk to anyone other than your own team members once the coding phase of the Java Challenge starts.
- You can submit any questions to the Contest Staff using the PC² “Clarification Request” system.
- Any team that submits a car containing code deemed by the contest staff to be intentionally designed to damage the CodeRally environment will be disqualified.
Example Rally Car Code

The following code snippets show simple examples of various operations which might be used inside a `move()` method. Note that these are separate code snippets, not a single complete `move()` method. Note also that these are only examples, intended to give you an idea of how to do things within your car. Winning cars will undoubtedly utilize sophisticated strategies which take full advantage of the range of method calls available to them.

The list of methods available to cars is documented in the JavaDoc descriptions of the classes and interfaces of the CodeRally environment. The primary challenge in the Java Challenge is for your team to decide on a strategy which uses the available methods to optimum advantage for your car.

[samples coming]
public abstract class Car extends java.lang.Object implements ICar

This class is the superclass of the RallyCar that you implement. This class contains helper methods to find out about the state of your car, to set the steering and throttle for your car, as well as entering protect mode and throwing spare tires.

Field Summary

| static byte | CAR_BLUE    |
| static byte | CAR_ORANGE  |
| static byte | CAR_PURPLE  |
| static byte | CAR_RED     |
| static byte | CAR_TEAL    |
| static byte | CAR_YELLOW  |
| static int  | MAX_CLOCK_TICKS |
| static int  | MAX_STEER_LEFT |
| static int  | MAX_STEER_RIGHT |
**enterProtectMode**

public boolean enterProtectMode()

Enter protect mode. This will shield your car from spare tires, but you will use twice the amount of fuel. Protect mode lasts for 50 clock ticks.

**Returns:**
boolean - true if the car entered protect mode, and false if the car was already in protect mode.

---

**getChangeInHeading**

public int getChangeInHeading()

**Description copied from interface: ICar**
Returns the current change of heading of the car, in degrees per turn. This method returns a positive number if the car is turning right, and a negative number if the car is turning left.

**Specified by:**
getChangeInHeading in interface ICar

**See Also:**
ICar.getChangeInHeading()

---

**getCheckpoints**

public IObject[] getCheckpoints()

Returns a list of checkpoints that you can pass to gain points. The first checkpoint returned is always the next one that you have to pass.

**Returns:**
com.ibm.rally.IObject[]

---

**getClockTicks**

public int getClockTicks()
Returns the number of clock ticks (turns) that have occurred during this match.

**Returns:**

```
int
```

---

### getColor

```
public abstract byte getColor()
```

Returns the color of your car. This method should return one of the six CAR_XXX color constants and will only be called once during initialization.

**Returns:**

```
byte
```

---

### getDistanceTo

```
public double getDistanceTo(double x, double y)
```

**Description copied from interface: IObject**

Returns the distance to the given location.

**Specified by:**

```
getDistanceTo in interface IObject
```

**See Also:**

```
IObject.getDistanceTo(double, double)
```

---

### getDistanceTo

```
public double getDistanceTo(IObject obj)
```

**Description copied from interface: IObject**

Returns the distance to the given object.

**Specified by:**

```
getDistanceTo in interface IObject
```

**See Also:**

```
IObject.getDistanceTo(IObject)
```

---

### getFuel

```
public int getFuel()
```

**Description copied from interface: ICar**

Returns the amount of fuel remaining in the car.

**Specified by:**
getFuel in interface ICar

See Also:
ICar.getFuel()

---

getFuelDepots

public IObject[] getFuelDepots()

Returns the available fuel depots for this match. When you are within 20 units of a fuel depot, your car will gain fuel.

Returns:
com.ibm.rally.IObject[]

---

getHeading

public int getHeading()

Description copied from interface: ICar
Return the car's heading, in degrees.
Specified by:
getHeading in interface ICar
See Also:
ICar.getHeading()

---

getHeadingTo

public int getHeadingTo(double x, double y)

Description copied from interface: IObject
Returns the heading (angle) to the given location.
Specified by:
getHeadingTo in interface IObject
See Also:
IObject.getHeadingTo(double, double)

---

getHeadingTo

public int getHeadingTo(IObject obj)

Description copied from interface: IObject
Returns the heading (angle) to the given object.
Specified by:
getName

public abstract java.lang.String getName()

    Returns the name of your car. This method must return a static string and will only be called once
during initialization.

    Specified by:
    getName in interface ICar

    Returns:
    java.lang.String

getNumberOfSpareTires

public int getNumberOfSpareTires()

    Description copied from interface: ICar
    Returns the number of spare tires currently available to be thrown from this car.

    Specified by:
    getNumberOfSpareTires in interface ICar

    See Also:
    ICar.getNumberOfSpareTires()

getOpponents

public ICar[] getOpponents()

    Returns an array of all the cars that you are currently competing against in this match. This array
will not include your own car, and will always return the cars in the same order.

    Returns:
    ICar[]

getPoints

public int getPoints()

    Description copied from interface: ICar
    Returns the number of points that this car has accumulated during this match.

    Specified by:
getPoints in interface ICar

See Also:
ICar.getPoints()

getPreviousCheckpoint

public int getPreviousCheckpoint()

Returns the index of the previous checkpoint that was reached, or -1 if the car has not passed any checkpoints yet during this match. You can go to checkpoints in any order, but you will get more points if you go to the next checkpoint in the list.

Returns:
int

g getSpareTireDepot

public IObject[] getSpareTireDepot()

Returns the available spare tire depots for this match. When you are within 20 units of a spare tire depot, your car will gain a spare tire every 25 ticks.

Returns:
com.ibm.rally.IObject[]

g getSpareTiresOnTrack

public ISpareTire[] getSpareTiresOnTrack()

Returns the spare tires that have been thrown but have not yet hit a wall or another car.

Returns:
com.ibm.rally.ISpareTire[]

g getSchoolName

public abstract java.lang.String getSchoolName()

Return the name of your school. This method must return a static string and will only be called once during initialization.

Specified by:
getSchoolName in interface ICar

Returns:
java.lang.String
getSpeed

public double getSpeed()

Description copied from interface: ICar
Return the car's current speed, in units per turn. This method returns the speed in the direction in which the car is heading, so the car may move (e.g. as the result of being hit sideways by another car) without the speed changing. This method will return a negative value if the car is going backwards.
Specified by:
getSpeed in interface ICar
See Also:
ICar.getSpeed()

getSteeringSetting

public int getSteeringSetting()

Returns the steering setting for this car. The steering setting will affect the turn rate of the car.
Returns:
int;

getThrottle

public int getThrottle()

getTrackHeight

public int getTrackHeight()

Returns the height of the track, in units.
Returns:
int

getTrackWidth

public int getTrackWidth()

Returns the width of the track, in units.
Returns:
int

**getX**

```java
public double getX()
```

*Description copied from interface: `IObject`*
Return the X position.

**Specified by:**
`getX` in interface `IObject`

**See Also:**
`IObject.getX()`

**getY**

```java
public double getY()
```

*Description copied from interface: `IObject`*
Return the Y position.

**Specified by:**
`getY` in interface `IObject`

**See Also:**
`IObject.getY()`

**initialize**

```java
public abstract void initialize()
```

Called to give you a chance to do initialization. This method will be called at the beginning of each match, and you will have a limited amount of time to do initialization.

**isHeadlightsOn**

```java
public boolean isHeadlightsOn()
```

Returns true if the car's headlights are on.

**Returns:**
`boolean`

**isInProtectMode**
public boolean isInProtectMode()

Returns true if the car is in protect mode.

Specified by:

isInProtectMode in interface ICar

Returns:

boolean

isReadyToThrowSpareTire

public boolean isReadyToThrowSpareTire()

Returns true if the car is ready to throw a spare tire, and false if does not have any spare tires, or if it is not ready to throw again after the last spare tire.

Returns:

boolean

move

public abstract void move(int lastMoveTime,
boolean hitWall,
ICar collidedWithCar,
ICar hitBySpareTire)

This method is called repeatedly to allow you to move your car. When you have run out of fuel, this method will no longer be called. The first parameter returns the length of time (in ms) that the last call to move() took.

The second parameter (hitWall) returns true if your car has just ran into a wall. If your car has collided with another car, the third parameter (collidedWithCar) will return that other car; otherwise this parameter will be null. If you have just been hit by a spare tire from another car, the fourth parameter (hitBySpareTire) returns that other car; otherwise this parameter will be null.

Parameters:

int - lastMoveTime
boolean - hitWall
ICar - collidedWithCar
ICar - hitBySpareTire

setHeadlightsOn

public void setHeadlightsOn(boolean on)

Turn the car headlights on or off. There's no reason to do this, but hey, why not?

Parameters:
boolean -

setSteeringSetting

public void setSteeringSetting(int steering)

Set the steering setting. This must be somewhere between the MAX_STEER_LEFT and MAX_STEER_RIGHT constants, and will affect the car's rate of turn.

Parameters:

steering -

setThrottle

public void setThrottle(int throttle)

Set the throttle (gas peddle) of the car. This must be somewhere between the MIN_SPEED (reverse) and MAX_SPEED (full throttle) constants. Increasing the throttle will make the car move faster, but will also burn more fuel per unit travelled.

Parameters:

throttle -

throwSpareTire

public boolean throwSpareTire()

Throw a spare tire out the front of the car. The spare tire will be thrown in the same heading as the car is currently facing. You can never be hit by a spare tire thrown by your car.

Returns:

boolean - true if the spare tire was thrown, and false if the car did not have any remaining spare tires or was not ready to throw again after the last thrown spare tire.
public interface ICar
extends IObject

This interface is used by all cars on the track. It allows you to find information about the other cars in your match.

### Method Detail

#### getChangeInHeading

defined as public int getChangeInHeading()

Returns the current change of heading of the car, in degrees per turn. This method returns a positive number if the car is turning right, and a negative number if the car is turning left.

**Returns:**

- int

#### getFuel

defined as public int getFuel()

Returns the amount of fuel remaining in the car.

**Returns:**

- int

#### getHeading

defined as public int getHeading()

Return the car's heading, in degrees.

**Returns:**

This interface is used by all cars on the track. It allows you to find information about the other cars in your match.

### Method Detail

#### getChangeInHeading

defined as public int getChangeInHeading()

Returns the current change of heading of the car, in degrees per turn. This method returns a positive number if the car is turning right, and a negative number if the car is turning left.

**Returns:**

- int

#### getFuel

defined as public int getFuel()

Returns the amount of fuel remaining in the car.

**Returns:**

- int

#### getHeading

defined as public int getHeading()

Return the car's heading, in degrees.

**Returns:**
int

getName

details

public java.lang.String getName()

    Returns the name of the car.
    Returns:
        java.lang.String

getNumberOfSpareTires

details

public int getNumberOfSpareTires()

    Returns the number of spare tires currently available to be thrown from this car.
    Returns:
        int

getPoints

details

public int getPoints()

    Returns the number of points that this car has accumulated during this match.
    Returns:
        int

gSchoolName

details

public java.lang.String getSchoolName()

    Returns the name of the school.
    Returns:
        java.lang.String

gSpeed

details

public double getSpeed()

    Return the car's current speed, in units per turn. This method returns the speed in the direction in which the car is heading, so the car may move (e.g. as the result of being hit sideways by another
car) without the speed changing. This method will return a negative value if the car is going backwards.

**Returns:**

double

---

**isInProtectMode**

```java
class Car {
    public boolean isInProtectMode() {
        // Returns true if the car is in protect mode.
        return true;  // Example return value
    }
}
```

**Returns:**

boolean
This page intentionally left blank
public interface IObject

This interface is implemented by all objects in the game. It allows you to get the location of any object, and to determine the distance and heading to or from the object.

Method Detail

getDistanceTo

public double getDistanceTo(double x,
    double y)

Returns the distance to the given location.

Parameters:
    double - x
double - y

Returns:
    double

getDistanceTo

public double getDistanceTo(IObject obj)

Returns the distance to the given object.

Parameters:
    IObject -

Returns:
    double

getHeadingTo

public int getHeadingTo(double x,
    double y)
Returns the heading (angle) to the given location.

**Parameters:**
- double - x
- double - y

**Returns:**
- int

---

**getHeadingTo**

public int getHeadingTo(IObject obj)

Returns the heading (angle) to the given object.

**Parameters:**
- IObject -

**Returns:**
- int

---

**getX**

public double getX()

Return the X position.

**Returns:**
- double

---

**getY**

public double getY()

Return the Y position.

**Returns:**
- double
public interface ISpareTire extends IObject

This interface is used by the spare tires that have been thrown across the track. Use this interface to determine the speed and heading for a spare tire.

### Method Detail

#### getHeading

public int getHeading()

Return the heading, in degrees.

**Returns:**

int

#### getSpeed

public double getSpeed()

Return the speed, in units per turn.

**Returns:**

double
public class RallyCar
extends Car

This is the class that you must implement to enable your car within the CodeRally track. Adding code to these methods will give your car it's personality and allow it to compete.

### Fields inherited from class com.ibm.rally.Car

- CAR_BLUE
- CAR_ORANGE
- CAR_PURPLE
- CAR_RED
- CAR_TEAL
- CAR_YELLOW
- MAX_CLOCK_TICKS
- MAX_STEER_LEFT
- MAX_STEER_RIGHT
- MAX_THROTTLE
- MIN_THROTTLE

### Method Detail

#### getColor

public byte getColor()

Description copied from class: Car

Returns the color of your car. This method should return one of the six CAR_XXX color constants and will only be called once during initialization.

Overrides:

- getColor in class Car

See Also:

- Car.getColor()

#### getName

public java.lang.String getName()

Description copied from class: Car

Returns the name of your car. This method must return a static string and will only be called once during initialization.

Overrides:
getName in class Car
See Also:
Car.getName()

getSchoolName

public java.lang.String getSchoolName()

Description copied from class: Car
Return the name of your school. This method must return a static string and will only be called once during initialization.
Overrides:
getSchoolName in class Car
See Also:
Car.getSchoolName()

initialize

public void initialize()

Description copied from class: Car
Called to give you a chance to do initialization. This method will be called at the beginning of each match, and you will have a limited amount of time to do initialization.
Overrides:
initialize in class Car
See Also:
Car.initialize()

move

public void move(int lastMoveTime,
 boolean hitWall,
 ICar collidedWithCar,
 ICar hitBySpareTire)

Description copied from class: Car
This method is called repeatedly to allow you to move your car. When you have run out of fuel, this method will no longer be called. The first parameter returns the length of time (in ms) that the last call to move() took.

The second parameter (hitWall) returns true if your car has just ran into a wall. If your car has collided with another car, the third parameter (collidedWithCar) will return that other car; otherwise this parameter will be null. If you have just been hit by a spare tire from another car, the fourth parameter (hitBySpareTire) returns that other car; otherwise this parameter will be null.
Overrides:
move in class Car

See Also:
Car.move(int, boolean, ICar, ICar)
Example Rally Car Code

The following code snippets show simple examples of various operations which might be used inside a `move()` method. Note that these are separate code snippets, not a single complete `move()` method. Note also that these are only examples, intended to give you an idea of how to do things within your car. Winning cars will undoubtedly utilize sophisticated strategies which take full advantage of the range of method calls available to them.

The list of methods available to cars is documented in the JavaDoc descriptions of the classes and interfaces of the CodeRally environment. The primary challenge in the Java Challenge is for your team to decide on a strategy which uses the available methods to optimum advantage for your car.

```java
/**
 * Go toward the first spare tire depot.
 */
public void move(int lastMoveTime, boolean hitWall, ICar collidedWithCar,
                  ICar hitBySpareTire) {
    // pick a spare tire depot
    IObject st = getSpareTireDepot()[0];

    // go toward the checkpoint
    int h = getHeadingTo(st);
    if (getHeading() > h)
        setSteeringSetting(MAX_STEER_LEFT);
    else
        setSteeringSetting(MAX_STEER_RIGHT);
    setThrottle(MAX_THROTTLE);
}

/**
 * Put the car in reverse for a few moves if you collide with another car.
 */
protected int wait;
public void move(int lastMoveTime, boolean hitWall, ICar collidedWithCar,
                 ICar hitBySpareTire) {
    if (collidedWithCar != null)
        wait = 10;
    if (wait > 0)
        setThrottle(MIN_THROTTLE);
    else
        setThrottle(MAX_THROTTLE);
    if (wait > 0)
        wait--;
}