Selected Robot Documentation

The `becker` library that accompanies this book includes many methods, particularly those related to robots and the cities they inhabit. Full documentation is available on the Web at *www.learningwithrobots.com/doc*. For the times that a computer isn't available, a briefer form of the documentation is printed below.

City

```
public class City extends java.lang.Object
```

A `City` contains intersections joined by streets and avenues. `Intersections` may contain `Things` such as `Walls`, `Streetlights`, and `Flashers`, as well as `Robots`.

Constructor Summary

```
City()
```

Construct a new `City` using the defaults stored in the `becker.robots.ini` file.

```
City(int numVisibleStreets, int numVisibleAvenues)
```

Construct a new `City` that displays streets 0 through numVisibleStreets – 1 and avenues 0 through numVisibleAvenues – 1.

```
City(int firstVisibleStreet, int firstVisibleAvenue,
     int numVisibleStreets, int numVisibleAvenues)
```

Construct a new `City` that displays streets firstVisibleStreet through numVisibleStreets – 1 and avenues firstVisibleAvenue through numVisibleAvenues – 1.

```
City(String fileName)
```

Construct a new `City` by reading information to construct it from a file.

```
City(java.util.Scanner in)
```

Construct a new `City` by reading information to construct it from a file.

Method Summary

```
protected void customizeIntersection(Intersection intersection)
```

Customize an `Intersection`, perhaps by adding `Things` to it.

```
IIterate<Light> examineLights()
```

Examine all the `Light` objects in this `City`, one at a time.

```
IIterate<Robot> examineRobots()
```

Examine all the `Robot` objects in this `City`, one at a time.

```
IIterate<Thing> examineThings()
```

Examine all the `Thing` objects in this `City`, one at a time.

```
IIterate<Thing> examineThings(IPredicate aPredicate)
```

Examine all the `Thing` objects in this `City` that match `aPredicate`, one at a time.

```
protected Intersection getIntersection(int avenue, int street)
```

Obtain a reference to a specified `Intersection` within this `City`.

```
boolean isShowingThingCounts()
```

Is this `City` showing the number of `Things` on each `Intersection`?

```
protected void keyTyped(char key)
```

This method is called when this `City`'s display has the focus and a key is typed.

```
protected Intersection makeIntersection(int avenue, int street)
```

Make an `Intersection` that will appear at the specified avenue and street.

```
void save(String indent, java.io.PrintWriter out)
```

Save a representation of this `City` to a file for later use.

```
void setFrameTitle(String title)
```

Set the title of the implicitly created frame, if there is one.

```
void setSize(int width, int height)
```

Set the size of the implicitly created frame, if there is one.

```
void setThingCountPredicate(Predicate pred)
```

Set the predicate for what kinds of `Things` to count when showing the number of `Things` on each `Intersection`.

```
void showFrame(boolean show)
```

Should this `City` be shown in a frame? The default is to show it.

```
void showThingCounts(boolean show)
```

Show the number of `Things` on each `Intersection`, counted according to the predicate set with the method `setThingCountPredicate`.

Direction

```
public enum Direction
```

Constants that define directions within a `City`.

Field Summary

```
public static int EAST
public static int NORTH
public static int WEST
public static int SOUTH
```

Method Summary

```
Direction left()
```

Which `Direction` is left of this `Direction`?

```
Direction right()
```

Which `Direction` is right of this `Direction`?

```
Direction opposite()
```

Which `Direction` is opposite of this `Direction`?

Flasher

```
public class Flasher extends Light
```

A `Flasher` is commonly used to mark construction hazards on streets and avenues. `Flashers` are small enough for a `Robot` to pick up and carry. They do not obstruct the movement of `Robots`. Like all `Lights`, they can be turned on and off. Unlike some kinds of `Lights`, when `Flashers` are "on," their lights cycle on and off. When `Flashers` are turned "off," their lights stay off.

Constructor Summary

Flasher(City aCity, int aStreet, int anAvenue)

Construct a new Flasher, initially turned off.

Flasher(City aCity, int aStreet, int anAvenue, boolean isOn)

Construct a new Flasher.

Flasher(Robot heldBy)

Construct a new Flasher held by a Robot.

Method Summary

protected void save(String indent, java.io.PrintWriter out)

Save a representation of this Flasher to an output stream.

void turnOff()

Turn the Flasher off.

void turnOn()

Turn the Flasher on so that it begins to flash.

Intersection

public class Intersection extends Sim implements ILabel

Karel the Robot lives in a city composed of intersections connected by roads. Roads that run north and south (up and down) are called "Avenues" and roads that run east and west are called "Streets."

Intersections may contain Things such as Flashers, Walls, and Streetlights. Some kinds of Things block Robots from entering or exiting an Intersection. It is possible to build Things that are one-way, blocking Robots from entering but not exiting (or vice versa) an Intersection.

Constructor Summary

Intersection(City aCity, int aStreet, int anAvenue)

Construct a new Intersection.

Method Summary

```
protected void addSim(Sim theThing)
```

Add a `Sim` to this `Intersection`.

```
int countThings()
```

Determine the number of `Things` currently on this `Intersection`.

```
int countThings(IPredicate pred)
```

Determine the number of `Things` currently on this `Intersection` that match the given predicate.

```
protected boolean entryIsBlocked(Direction dir)
```

Determine whether something on this `Intersection` blocks `Robots` from entering this `Intersection` from the given direction.

```
IIterate<Light> examineLights(IPredicate aPredicate)
```

Examine all the `Light` objects on this `Intersection` that match the given predicate, one at a time.

```
IIterate<Robot> examineRobots(IPredicate aPredicate)
```

Examine all the `Robot` objects on this `Intersection` that match the given predicate, one at a time.

```
IIterate<Thing> examineThings(IPredicate aPredicate)
```

Examine all the `Thing` objects on this `Intersection` that match the given predicate, one at a time.

```
IIterate<Thing> examineThings()
```

Examine all the `Thing` objects on this `Intersection`, one at a time.

```
protected boolean exitIsBlocked(Direction dir)
```

Determine whether something on this `Intersection` blocks `Robots` from exiting the `Intersection`.

```
int getAvenue()
```

Get the avenue intersecting this `Intersection`.

```
protected Intersection getIntersection()
```

Return a reference to this `Intersection`.

```
String getLabel()
```

Get the label for this `Intersection`.

```
Intersection getNeighbor(Direction dir)
```

Get the `Intersection` neighboring this one in the given direction.

```
int getStreet()
```

Get the street intersecting this `Intersection`.

```
protected void removeSim(Sim s)
```

Remove the given `Sim` (`Robot`, `Flasher`, `Streetlight`, `Wall`, and so on) from this `Intersection`.

```
protected void save(String indent, java.io.PrintWriter out)
```

Save a representation of this `Intersection` to an output stream.

```
void setLabel(String aLabel)
```

Set a label for this `Intersection`.

```
String toString()
```

Report the internal state of this `Intersection`.

IPredicate

```
public interface IPredicate
```

A predicate says whether something is true or false about a `Sim`. A class implementing the `IPredicate` interface does this via the `isOK` method, which returns `true` if some condition about a `Sim` is true, and `false` otherwise.

A typical use for a predicate is to find a certain kind of `Thing` for a `Robot` to examine—for example, a `Light`. To do this, define a class implementing `IPredicate` as follows:

```
public class ALightPred implements Predicate
{    //return true if the Sim passed is a Light, false otherwise
    public boolean isOK(Sim s)
    {   return s instanceof Light;
    }
}
```

In a subclass of `Robot`, invoke the `examineThings` method like this:

```
Light light = this.examineThing(new ALightPred()).next();
```

which will return a `Light` from the current `Intersection`, if there is one, and throw an exception if there is not. The `isBesideThing` method in the `Robot` class can be used to determine if the specified kind of `Thing` is available.

The `IPredicate` class also defines a number of useful predicates as constants. For example, to pick up a `Thing` that is a `Flasher`, one could write

```
karel.pickThing(IPredicate.aFlasher);
```

Field Summary

static IPredicate **aFlasher**

A predicate to test whether something is a `Flasher`.

---

static IPredicate **aLight**

A predicate to test whether something is a `Light`.

---

static IPredicate **anyFlasher**

A predicate to test whether something is a `Flasher` or a subclass of `Flasher`.

---

static IPredicate **anyLight**

A predicate to test whether something is a `Light` or a subclass of `Light`.

---

static IPredicate **anyRobot**

A predicate to test whether something is a `Robot` or a subclass of `Robot`.

---

static IPredicate **anyStreetlight**

A predicate to test whether something is a `Streetlight` or a subclass of `Streetlight`.

---

static IPredicate **anyThing**

A predicate to test whether something is a `Thing` or a subclass of `Thing`.

---

static Predicate **anyWall**

A predicate to test whether something is a `Wall` or a subclass of `Wall`.

---

static Predicate **aRobot**

A predicate to test whether something is a `Robot`.

---

static Predicate **aStreetlight**

A predicate to test whether something is a `Streetlight`.

```
static Predicate aThing
```

A predicate to test whether something is a `Thing`.

```
static Predicate aWall
```

A predicate to test whether something is a `Wall`.

```
static Predicate canBeCarried
```

A predicate to test whether the `Thing` is something that a `Robot` can carry.

Method Summary

```
boolean isOK(Sim theSim)
```

Returns `true` if a certain condition is true about `theSim`, `false` otherwise.

Light

```
public abstract class Light extends Thing
```

A `Light` is a kind of `Thing` that can be turned on to make it brighter and turned off to make it darker. Some `Lights` can be moved (`Flasher`) while others can't (`Streetlight`).

The `Light` class itself is abstract, meaning programmers cannot construct an instance of `Light`. It must be extended to create a class that can be instantiated. This class does define a common interface for all `Lights` so that any `Light` may be turned on or off without knowing what specific kind of `Light` it is (polymorphism).

Constructor Summary

```
Light(City aCity, int aStreet, int anAvenue)
```

Construct a new `Light` with the same default appearance as a `Thing`.

```
Light(City aCity, int aStreet, int anAvenue, Direction
    orientation, boolean canBeMoved, Icon anIcon)
```

Construct a new `Light`.

```
Light(Robot heldBy)
```

Construct a new `Light` held by a `Robot`.

Method Summary

`boolean isOn()`

Determine whether the `Light` is turned on.

`void turnOff()`

Turn the `Light` off.

`void turnOn()`

Turn the `Light` on.

Robot

`public class Robot extends Sim implements ILabel, IColor`

`Robot`s exist on a rectangular grid of roads and can move, turn left ninety degrees, pick things up, carry things, and put things down. A `Robot` knows which avenue and street it is on and which direction it is facing. Its speed can be set and queried.

More advanced features include determining if it is safe to move forward, examining `Thing`s on the same `Intersection` as themselves, and determining if they are beside a specific kind of `Thing`. They can pick up and put down specific kinds of `Thing`s and determine how many `Thing`s they are carrying.

Constructor Summary

`Robot(City aCity, int aStreet, int anAvenue, Direction aDir)`

Construct a new `Robot` at the given location in the given `City` with nothing in its backpack.

`Robot(City c, int str, int ave, Direction aDir, int numThings)`

Construct a new `Robot` at the given location in the given `City` with the given number of `Thing`s in its backpack. Override `makeThing` to customize the kind of `Thing` added to the backpack.

Method Summary

```
protected void breakRobot(String msg)
```

This method is called when this `Robot` does something illegal such as trying to move through a `Wall` or picking up a nonexistent object. An exception is thrown that stops this `Robot`'s operation.

```
boolean canPickThing()
```

Determine whether this `Robot` is on the same `Intersection` as a `Thing` it can pick up.

```
int countThingsInBackpack()
```

How many `Thing` objects are in this `Robot`'s backpack?

```
int countThingsInBackpack(IPredicate kindOfThing)
```

How many of a specific kind of `Thing` are in this `Robot`'s backpack?

```
IIterate<Light> examineLights()
```

Examine all the `Light` objects that are on the same `Intersection` as this `Robot`, one at a time.

```
IIterate<Robot> examineRobots()
```

Examine all the `Robot` objects that are on the same `Intersection` as this `Robot`, one at a time.

```
IIterate<Thing> examineThings(IPredicate aPredicate)
```

Examine all the `Thing` objects that are on the same `Intersection` as this `Robot` and match the given predicate, one at a time.

```
boolean frontIsClear()
```

Can this `Robot` move forward to the next `Intersection` safely?

```
int getAvenue()
```

On which avenue is this `Robot`?

```
Direction getDirection()
```

Which `Direction` is this `Robot` facing?

```
String getLabel()
```

What is the string labeling this `Robot`?

```
double getSpeed()
```

How many moves and/or turns does this `Robot` complete in one second?

```
int getStreet()
```

On which street is this `Robot`?

```
double getTransparency()
```

Get this `Robot`'s transparency.

```
boolean isBesideThing(IPredicate aPredicate)
```

Determine whether this `Robot` is on the same `Intersection` as one or more instances of the specified kind of `Thing`.

```
protected Thing makeThing(int nOf, int total)
```

Make a new `Thing` to place in this `Robot`'s backpack. Override this method in a subclass to control what kind of `Thing` is made when a `Robot` is constructed with `Thing`s in its backpack.

```
void move()
```

Move this `Robot` from the `Intersection` it currently occupies to the next `Intersection` in the `Direction` it is currently facing, leaving it facing the same `Direction`.

```
void pickThing()
```

Attempt to pick up a movable `Thing` from the current `Intersection`.

```
void pickThing(IPredicate kindOfThing)
```

Attempt to pick up a particular kind of `Thing` from the `Intersection` this `Robot` currently occupies.

```
void pickThing(Thing theThing)
```

Attempt to pick up a particular `Thing` from the `Intersection` this `Robot` currently occupies.

```
void putThing()
```

Take something out of this `Robot`'s backpack and put it down on the `Intersection` this `Robot` currently occupies.

```
void putThing(IPredicate kindOfThing)
```

Attempt to take a particular kind of `Thing` out of this `Robot`'s backpack and put it down on the `Intersection` the `Robot` currently occupies.

```
void putThing(Thing theThing)
```

Attempt to put down a particular `Thing` on the `Intersection` this `Robot` currently occupies.

```
protected void save(String indent, java.io.PrintWriter out)
```

Save a representation of this `Robot` to an output stream.

```
void setLabel(String theLabel)
```

Set a label to identify this `Robot`.

```
void setSpeed(double movesPerSecond)
```

Set this `Robot`'s speed.

```
void setTransparency(double trans)
```

Set this `Robot`'s transparency.

```
void turnLeft()
```

Turn this `Robot` left by 90 degrees or one-quarter turn.

RobotRC

```
public class RobotRC extends Robot
```

A remote control robot, `RobotRC` for short, can be directed from a computer keyboard. The `City`'s view must have the keyboard focus when the program is running for the `Robot` to receive the instructions from the keyboard. When the `City`'s view has the focus, it will have a thin black outline. Shift the focus between the speed control and the start/stop button on the `City`'s view with the tab key.

Constructor Summary

```
RobotRC(City aCity, int aStreet, int anAvenue, Direction aDir)
```

Construct a new `RobotRC` with nothing in its backpack.

```
RobotRC(City aCity, int aStreet, int anAvenue, Direction aDir,
    int numThings)
```

Construct a new `RobotRC`.

Method Summary

```
protected void keyTyped(char key)
```

This method makes the robot respond to the user's key presses as shown in Table E-1. It may be overridden to make the robot respond differently.

Keys	Response
m, M	move
r, R	turn right
l, L	turn left
u, U	pick up a `Thing`
d, D	put down a `Thing`

RobotSE

```
public class RobotSE extends Robot
```

A new kind of `Robot` with extended capabilities, such as `turnAround` and `turnRight`.

Constructor Summary

```
RobotSE(City aCity, int aStreet, int anAvenue, Direction aDir)
```

Construct a new `RobotSE` with nothing in its backpack.

```
RobotSE(City aCity, int aStreet, int anAvenue, Direction aDir,
    int numThings)
```

Construct a new `RobotSE`.

Method Summary

```
boolean isFacingEast()
```

Determine whether this `Robot` is facing east.

```
boolean isFacingNorth()
```

Determine whether this `Robot` is facing north.

```
boolean isFacingSouth()
```

Determine whether this `Robot` is facing south.

```
boolean isFacingWest()
```

Determine whether this Robot is facing west.

```
void move(int howFar)
```

Move the given distance.

```
void pickAllThings()
```

Pick up all the Things that can be carried from the current Intersection.

```
void pickAllThings(Predicate kindOfThing)
```

Pick up all of the specified kind of Things from the current Intersection.

```
void putAllThings()
```

Put down all the Things in this Robot's backpack on the current Intersection.

```
void putAllThings(Predicate kindOfThing)
```

Put down all of the specified kind of Things from the Robot's backpack on the current Intersection.

```
void turnAround()
```

Turn this Robot around so it faces the opposite Direction.

```
void turnLeft(int numTimes)
```

Turn this Robot left the given number of times.

```
void turnRight()
```

Turn this Robot 90 degrees to the right.

```
void turnRight(int numTimes)
```

Turn this Robot right the given number of times.

Sim

```
public abstract class Sim extends java.lang.Object
```

A Sim is an element of a City that participates in the simulation, namely a Thing (such as Walls or Lights), a Robot, or an Intersection.

Since this class is abstract it cannot be instantiated; only subclasses may be instantiated. This class exists both to ensure that basic services required for the simulation are present and to provide common implementations for required several services.

Constructor Summary

```
Sim(City aCity, int aStreet, int anAvenue, Direction orientation,
    Icon theIcon)
```

Construct a new `Sim`.

Method Summary

```
Icon getIcon()
```

Return the icon used to display the visible characteristics of this `Sim`, based on the `Sim`'s current state.

```
protected abstract Intersection getIntersection()
```

Return the `Intersection` where this `Sim` is located.

```
protected void keyTyped(char key)
```

This method is called when a key is typed and keyboard input is directed to karel's world (the map, as opposed to a different window or the controls for karel's world).

```
protected void notifyObservers()
```

Notify any observers of this `Sim` (for instance, the user interface) that it has changed.

```
protected void notifyObservers(java.lang.Object changeInfo)
```

Notify any observers of this `Sim` (for instance, the user interface) that it has changed.

```
void setIcon(Icon theIcon)
```

Set the icon used to display this `Sim`.

Streetlight

```
public class Streetlight extends Light
```

A `Streetlight` is a kind of `Light` that lights an intersection. Like all `Lights`, it can be turned on and off. A `Streetlight` cannot be moved by a `Robot`.

Constructor Summary

```
Streetlight(City city, int aStreet, int anAvenue, Direction corner)
```

Construct a new `Streetlight`.

```
Streetlight(City city, int aStreet, int anAvenue, Direction corner,
    boolean isOn)
```

Construct a new `Streetlight`.

Method Summary

```
protected void save(String indent, java.io.PrintWriter out)
```

Save a representation of this `StreetLight` to an output stream.

```
void turnOff()
```

Turn the `Streetlight` off.

```
void turnOn()
```

Turn the `Streetlight` on.

Thing

```
public class Thing extends Sim
```

A `Thing` is something that can exist on an `Intersection`. All `Things` have a location (avenue and street). Some `Things` can be picked up and moved by a `Robot` (`Flashers`) while others cannot (`Streetlights`, `Walls`).

In addition to a location, all `Things` have an orientation, although it is common for the orientation to always have a default value. Examples where that is not the case include a `Wall` where the orientation determines which exit or entry into an `Intersection` is blocked, and a `Streetlight` where the orientation determines which corner of the `Intersection` it occupies.

Constructor Summary

```
Thing(City city, int aStreet, int anAvenue)
```

Construct a new `Thing` with a default appearance that can be carried.

```
Thing(City aCity, int aStreet, int anAvenue, Direction orientation)
```

Construct a new `Thing` with a default appearance that can be carried, in the given orientation.

```
Thing(City aCity, int aStreet, int anAvenue, Direction orientation,
        boolean canBeMoved, Icon anIcon)
```

Construct a new `Thing` with an appearance defined by `anIcon`.

```
Thing(Robot heldBy)
```

Construct a new `Thing` held by the given `Robot`.

Method Summary

```
boolean blocksIntersectionEntry(Direction entryDir)
```

Does this `Thing` block the entry of this `Intersection` from the given `Direction`?

```
boolean blocksIntersectionExit(Direction exitDir)
```

Does this `Thing` block the exit of this `Intersection` in the given `Direction`?

```
boolean canBeCarried()
```

Can this `Thing` be picked up, carried, and put down by a `Robot`?

```
protected Intersection getIntersection()
```

Return a reference to this `Thing`'s `Intersection`.

```
protected void save(String indent, java.io.PrintWriter out)
```

Save a representation of this `Thing` to an output stream.

```
void setBlocksEntry(boolean north, boolean south, boolean east,
    boolean west)
```

Set whether this `Thing` blocks a `Robot`'s entry from the given `Directions`.

```
void setBlocksEntry(Direction aDir, boolean block)
```

Set whether this `Thing` blocks a `Robot`'s entry from the given `Direction`.

```
void setBlocksExit(boolean north, boolean south, boolean east,
    boolean west)
```

Set whether this `Thing` blocks a `Robot`'s exit from the given `Directions`.

```
void setBlocksExit(Direction aDir, boolean block)
```

Set whether this `Thing` blocks a `Robot`'s exit from the given `Direction`.

```
void setCanBeCarried(boolean canCarry)
```

Set whether this `Thing` can be picked up and carried by a `Robot`.

Wall

```
public class Wall extends Thing
```

A `Wall` will block the movement of a `Robot` into or out of the `Intersection` that contains it, depending on the `Robot`'s direction of travel and the orientation of the `Wall`.

Constructor Summary

```
Wall(City city, int aStreet, int anAvenue, Direction orientation)
```

Construct a new `Wall`.

Method Summary

```
protected void save(String indent, java.io.PrintWriter out)
```

Save a representation of this `Wall` to an output stream.